UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Departamento de Sistemas Informáticos y Computación

TESIS DOCTORAL

Sistemas de tipos en lenguajes lógico-funcionales

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Enrique Martín Martín

Directores

Francisco Javier López Fraguas
Juan Rodríguez Hortalá

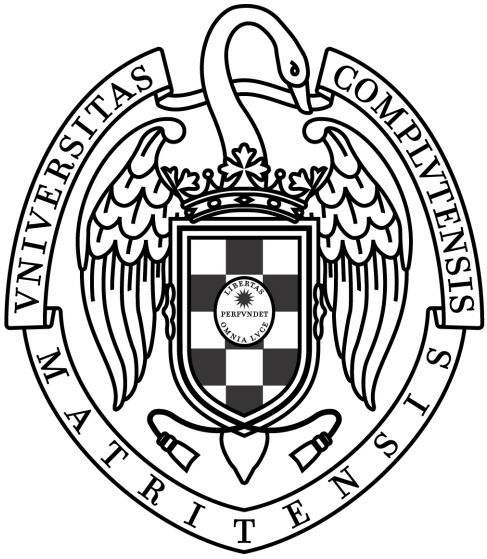
Madrid, 2013

Tesis doctoral

SISTEMAS DE TIPOS EN LENGUAJES LÓGICO-FUNCIONALES

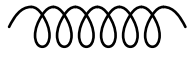
presentada por ENRIQUE MARTÍN MARTÍN *para la obtención del título de* Doctor en Ingeniería Informática *en el departamento de* Sistemas Informáticos y Computación *de la* Universidad Complutense de Madrid

Dirigida por los doctores FRANCISCO JAVIER LÓPEZ FRAGUAS *y* JUAN RODRÍGUEZ HORTALÁ



Finalizada en Madrid, *el 23 de abril de 2012*

Sistemas de tipos en lenguajes lógico-funcionales



Enrique Martín Martín

Tesis doctoral en formato publicaciones presentada por Enrique Martín Martín en el Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid para la obtención del título de doctor en ingeniería informática.

Finalización: 23 de abril de 2012.
Última revisión: 12 de julio de 2012.

Título:

Sistemas de tipos en lenguajes lógico-funcionales

Autor:

Enrique Martín Martín (emartinm@fdi.ucm.es)
Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Directores:

Francisco J. López Fraguas (fraguas@sip.ucm.es)
Juan Rodríguez Hortalá (juanrh@fdi.ucm.es)

Listado de cambios

12 de julio de 2012:

- Reemplazados los artículos contenidos en los apéndices A.1, A.2, A.3, A.4 y A.5 para cumplir con las cesiones de *copyright* firmadas con las distintas editoriales. Los artículos originales han sido reemplazados por versiones de los autores con los correspondientes mensajes de *copyright*.
- Modificado el listado de artículos en el apéndice A (página 130) para incluir mensajes de *copyright*.
- Corregida la errata en la Sección 5.3.4 (página 47): ambas expresiones $filter\ p\ (map\ h\ [zero])$ y $map\ h\ (filter\ (p \circ h)\ [zero])$ se pueden evaluar al valor $[\,]$.

Prólogo

Esta tesis sigue el formato de *tesis por publicaciones* según la normativa vigente de la Universidad Complutense de Madrid, y su organización responde a los requerimientos de dicho formato. La Parte I recoge la motivación, objetivos y principales contribuciones de la tesis. La Parte II presenta el estado actual del tema de la tesis, incluyendo una presentación de la programación lógico-funcional, las principales semánticas propuestas para este paradigma y los sistemas de tipos (tanto en lenguajes funcionales como lógico-funcionales). La Parte III contiene los distintos sistemas de tipos que componen esta tesis, realizando una presentación unificadora de las distintas propuestas. La Parte IV recoge las principales conclusiones de la tesis, así como algunas posibles líneas de trabajo futuro. Por último, la parte V contiene las publicaciones asociadas a la tesis en su formato y longitud original, además de dos de informes técnicos para disponer, dentro de la propia tesis, de las demostraciones formales de todos los resultados presentados.

Esta tesis ha sido desarrollada dentro del *Grupo de Programación Declarativa* (grupo de investigación reconocido por la UCM con referencia 910502). Durante la elaboración de esta tesis se ha contado con el apoyo de los proyectos de investigación *Foundations and Applications of Declarative Software Technologies — Software Tools and Multiparadigm Programming* (FAST-STAMP, referencia TIN2008-06622-C03-01), *Métodos RIgurosos para sistemas heTerogéneos y Móviles — Métodos Formales en Sistemas Software Heterogéneos* (MERIT-FORMS-UCM, referencia TIN2005-09207-C03-03), *Programa en métodos para el desarrollo de software fiable, de alta calidad y seguro de la Comunidad de Madrid* (PROMESAS-CM, referencia S-0505/TIC/0407) y *Programa de Métodos Rigurosos de Desarrollo de Software de la Comunidad de Madrid* (PROMETIDOS-CM, referencia S2009/TIC-1465). También se ha contado con las ayudas al grupo de investigación mediante las convocatorias de referencia UCM-BSCH-GR58/08-910502 y UCM-BSCH-GR35/10-A-910502.

Agradecimientos

En primer lugar, agradecer a Paco y Juan por todo el tiempo y la ayuda que me han dedicado, además de la confianza que han depositado en mí. Ha sido un auténtico placer tenerlos como directores y me siento un privilegiado por todo lo que he llegado a aprender a su lado. No me refiero únicamente al campo de la programación lógico-funcional y de los sistemas de tipos, sino de la investigación en sí. Quizá no llegue a ser un gran investigador, lo que sí es seguro es que, gracias a ellos, he llegado a ser un investigador bueno. Gracias por lo bien que me habéis tratado y por encontrar huecos para atenderme incluso en momentos en los que teníais agendas muy apretadas.

Por otro lado, esta tesis habría sido imposible de llevar a cabo sin el apoyo económico de las diferentes instituciones mencionadas en el prólogo. Desde aquí querría transmitirlas mi agradecimiento, particularmente a la Universidad Complutense de Madrid por haberme dado la oportunidad de compaginar la elaboración de esta tesis con la práctica de la docencia, actividad en muchos aspectos enriquecedora. También agradecer a los revisores anónimos —incluidos los menos anónimos, como Philip Wadler— que han examinado nuestros artículos. Sus interesantes comentarios nos han ayudado a mejorar nuestros trabajos, además de indicarnos nuevas referencias bibliográficas y enfoques distintos que no conocíamos.

Por último, y no por ello menos menos importante, el resto de personas. Gracias a todos los amigos y compañeros del despacho 220 (a uno y otro lado del muro, e incluso en Donostia). Realizar una tesis es, en ocasiones, una labor ingrata. Pero ellos han conseguido que, incluso en los malos momentos, ir a trabajar sea una de las cosas más agradables. También agradecer —y mucho— a mis amigos de Sonseca (¡CocoChufa!). Con ellos he pasado muchos buenos momentos, y gracias a su cariño he podido «desconectar», semana tras semana, de los trasiegos del trabajo.

Pero sobre todo, gracias a mi familia, y especialmente a mis padres: Encarni y Martín. Sin ellos, llegar hasta aquí no habría sido posible, ni habría tenido mucho sentido. También agradecer a mi hermana Marta la completa revisión gramatical y de estilo que ha realizado a la tesis, que ha mejorado considerablemente la presentación de la misma. Y para finalizar, gracias a Pá por los buenos momentos de los últimos meses, que hacen que el día a día sea —aún— mejor.

Índice

Índice de figuras

Resumen

La programación lógico-funcional es un paradigma de programación declarativa muy expresivo, fruto de la combinación de la programación funcional y la programación lógica. Entre sus principales características destacan la posibilidad de definir funciones indeterministas, los patrones de orden superior y el uso de variables libres que se ligan a valores adecuados durante el cómputo. Desde el punto de vista de tipos, los sistemas lógico-funcionales han adoptado de manera directa el sistema de tipos de Damas-Milner, proveniente del ámbito funcional, debido a su sencillez y a la existencia de tipos principales y métodos efectivos para la inferencia de tipos. Sin embargo, esta adaptación directa no maneja adecuadamente algunas de las principales características de los lenguajes lógico-funcionales como los patrones de orden superior o las variables libres, dando lugar errores de tipos durante la evaluación.

En esta tesis proponemos tres sistemas de tipos adecuados para la programación lógico-funcional cuyo objetivo es manejar correctamente estas características problemáticas desde el punto de vista de los tipos. Los sistemas de tipos propuestos, que tratan diferentes mecanismos de cómputos lógico-funcionales (reescritura y estrechamiento), dan solución a los mencionados problemas, proporcionando resultados técnicos de corrección. Además, suponen un mejora sobre propuestas previas de sistemas de tipos para programación lógico-funcional, ya que salvan algunas de sus limitaciones. Aparte de los resultados teóricos, en esta tesis también se han desarrollado implementaciones de los sistemas de tipos, integrándolos como fase de comprobación de tipos en ramas del sistema lógico-funcional Toy.

Palabras clave: Programación lógico-funcional, sistemas de tipos, patrones de orden superior, patrones opacos, polimorfismo, clases de tipos, programación genérica, variables extra, estrechamiento.

$ar(h)$	Aridad de un símbolo de constructora o función	Pág. 18
$fv(e)$	Variables libres de una expresión	Pág. 23
$bv(e)$	Variables ligadas de una expresión	Pág. 23
$var(e)$	Variables de una expresión	Pág. 24
$\mathcal{C}$	Contexto de un solo «hueco»	Fig. 4, pág. 23
$t, p \in Pat$	Patrones	Fig. 4, pág. 23
$\theta \in PSubst$	Sustitución de patrones	Fig. 4, pág. 23
$dom(\theta)$	Dominio de una sustitución	Pág. 24
$vran(\theta)$	Variables en el rango de una sustitución	Pág. 24
$\theta\|_A$, $\theta\|_{\setminus A}$	Restricción de una sustitución	Pág. 24
$e \to^l e'$	Paso de let-reescritura	Fig. 5, pág. 24
$e \rightsquigarrow^l_\theta e'$	Paso de let-estrechamiento	Fig. 6, pág. 27
τ	Tipo simple	Pág. 32
σ	Esquema de tipo	Pág. 32
$\mathcal{A}$	Conjunto de suposiciones de tipo	Página 52
$ftv(\sigma)$	Variables de tipo libres	Página 32
π	Sustitución de tipos	Pág. 32
$\sigma \succ \sigma'$	Instancia genérica	Pág 32
$\sigma \succ_{var} \tau$	Variante	Pág. 32
$\mathcal{A} \oplus \mathcal{A}'$	Unión de conjuntos de suposiciones	Pág. 32
$\mathcal{W}$	Algoritmo de inferencia Damas-Milner	Pág. 33
$\mathcal{A} \vdash e : \tau$	Derivación de tipos básica	Fig. 10, pág. 53
$\mathcal{A} \vdash^\bullet e : \tau$	Derivación de tipos $\vdash^\bullet$	Fig. 11, pág. 54
$critVar_{\mathcal{A}}(e)$	Variables críticas de una expresión	Def. 2, pág. 54
$wt^\bullet_{\mathcal{A}}(\mathcal{P})$	Programa bien tipado con respecto a $\vdash^\bullet$	Def. 3, pág. 55
$\mathcal{A} \Vdash e : \tau \vert \pi$	Inferencia de tipos básica	Fig. 12, pág. 57
$\mathcal{A} \Vdash^\bullet e : \tau \vert \pi$	Inferencia de tipos $\Vdash^\bullet$	Fig. 12, pág. 57
$\mathcal{B}(\mathcal{A}, \mathcal{P})$	Inferencia de tipos de un programa	Def. 5, pág. 59
Ψ	Eliminación de patrones compuestos	Fig. 13, pág. 61
$e \to^{lp} e'$	Paso de let-reescritura con manejo de let_m y let_p	Fig. 14, pág. 63
$\mathcal{A} \vdash^l e : \tau$	Derivación de tipos liberal	Fig. 15, pág. 68
$\mathcal{A} \Vdash^l e : \tau$	Inferencia de tipos liberal	Fig. 15, pág. 68
$wt^l_{\mathcal{A}}(\mathcal{P})$	Programa bien tipado liberal	Def. 6, pág. 68
$e \to^{lf} e'$	Paso de let-reescritura con fallo	Fig. 16, pág. 71
$nf_{\mathcal{P}}(e)$	Formas normales con respecto a $\to^{lf}$	Pág. 71
κ	Nombre de clase de tipos	Fig. 18, pág. 80
θ	Contexto de tipos	Fig. 18, pág. 80
ϕ	Contexto de tipos saturado	Fig. 18, pág. 80
ρ	Tipo sobrecargado	Fig. 18, pág. 80
$\mathcal{A} \vdash^e e : \tau$	Derivación de tipos $\vdash^e$	Fig. 23, pág. 93
$wt^e_{\mathcal{A}}(\mathcal{P})$	Programa bien tipado con respecto a $\vdash^e$	Def. 12, pág. 93
$wt^e_{\mathcal{A}}(\theta)$	Sustitución bien tipada	Def. 13, pág. 94
$e \rightsquigarrow^{lwt}_\theta e'$	Paso de let-estrechamiento bien tipado	Def. 15, pág. 96
$e \rightsquigarrow^{lmgu}_\theta e'$	Paso de let-estrechamiento reducido	Def. 16, pág. 98
$OIS(\mathcal{P})$	Transformación a programa OIS	Def. 17, pág. 100
$\mathcal{U}(\mathcal{P})$	Transformación a formato uniforme	Def. 18, pág. 101
$UTypes_{\mathcal{A}}$	Tipos inseguros con respecto a $\mathcal{A}$	Def. 19, pág. 104
$wt^r_{\mathcal{A}}(\mathcal{P})$	Programa bien tipado restringido	Def. 20, pág. 105

Figura 1: Resumen de notación

Parte I
Introducción

1. Presentación y motivación

La programación lógico-funcional (PLF) [8, 55, 130, 53] surge de la combinación de distintos paradigmas declarativos: la programación lógica, la programación funcional e incluso la programación con restricciones. Estos tipos de paradigmas se caracterizan por abstraer al programador de detalles como el orden de evaluación o la asignación, permitiéndole programar a un nivel de abstracción más alto y cercano al dominio del problema que los lenguajes imperativos clásicos como Java o C/C++. En otras palabras, *un programa declarativo es una descripción de las propiedades que debe cumplir una solución, en lugar de una secuencia de pasos que se deben llevar a cabo en cierto orden para construir dicha solución*. La combinación de estos paradigmas ha sido un área de investigación bastante activo en las últimas décadas, siendo Toy [93, 23] y Curry [54] los dos lenguajes de programación lógico-funcional más representativos de la corriente mayoritaria en el área, en la que se encuadra también esta tesis.

Siendo un combinación de distintos paradigmas, la programación lógico-funcional hereda interesantes características de ellos. De la programación funcional toma las funciones de orden superior (permitiendo definir funciones que aceptan otras funciones como argumento, y que pueden usarlas en su cuerpo), el sistema de tipos de Damas/-Milner y su polimorfismo (permitiendo que una misma definición de función sea válida para varios tipos distintos) y la evaluación perezosa (evitando así que la evaluación de una expresión implique la evaluación de la totalidad de sus subexpresiones). Del campo de la programación lógica, la programación lógico-funcional adopta la búsqueda indeterminista (una expresión puede evaluarse a distintos valores, que son mostrados por el sistema uno a uno) y las variables lógicas (variables libres que se ligan a valores durante la ejecución del programa). Finalmente, de la programación con restricciones hereda la posibilidad de añadir restricciones sobre distintos dominios (Herbrand, dominios finitos, números reales...), restricciones que son congeladas o van resolviéndose según avanza el cómputo lógico-funcional y se evalúan las expresiones. Todo ello hace que la programación lógico-funcional sea un paradigma de alto nivel de abstracción, ofreciendo a los usuarios una gran expresividad y comodidad a la hora de programar.

Por otro lado, los sistemas de tipos son análisis incluidos en los lenguajes cuyo fin es garantizar que ciertos tipos de errores no aparezcan durante la ejecución de los programas. Para ello, clasifican las construcciones del programa (expresiones, instrucciones, etc.) de acuerdo a la clase de valores que representan (su *tipo*), impidiendo su uso en lugares incompatibles. Los sistemas de tipos tienen una larga historia desde su aparición en la década de los 50, siendo utilizados extensivamente por lenguajes

de programación actuales como Java, C/C++/C#, Python... Sin embargo, podría decirse que es en el campo de la programación funcional (PF) donde han cobrado más importancia y han tenido un mayor desarrollo. Particularmente importante es el sistema de tipos de Damas/Milner (DM) [58, 103, 32, 31] desarrollado originalmente para ML, que ha sido la base de los sistemas de tipos para posteriores lenguajes funcionales (como Haskell) e incluso lógico-funcionales (Curry, Toy).

Debido a la gran importancia del sistema de tipos de Damas/Milner en programación funcional, la programación lógico-funcional ha incluido este sistema de tipos desde sus orígenes. Pero contrariamente a lo que ha pasado en programación funcional, donde se han propuesto una gran variedad de extensiones y mejoras al sistema de tipos (citando algunas: recursión polimórfica [108, 76, 57], tipos existenciales [105, 112, 79], clases de tipos [48, 12], polimorfismo de rango arbitrario [111, 118], tipos de datos algebraicos generalizados (GADTs) [28, 119, 134], programación genérica [60, 62, 64]...), los sistemas de tipos han recibido una escasa atención en programación lógico-funcional. De hecho, en PLF los sistemas se han limitado a adaptar de manera directa el sistema de Damas/Milner, omitiendo en muchos casos un tratamiento formal que demuestre su corrección en este paradigma. Esta omisión ha provocado que algunas de las características particulares de la PLF como los *patrones de orden de orden superior* [44] no hayan sido tratadas adecuadamente, dando lugar a errores de tipos. En el siguiente ejemplo, adaptado de [45], mostramos algunos de ellos.

Ejemplo 1 (*Casting* polimórfico y descomposición opaca) *Consideremos el siguiente programa escrito con sintaxis Toy, es decir, con las variables en mayúsculas y los símbolos de función/constructora en minúsculas. En este ejemplo y los siguientes utilizaremos las constructoras usuales para listas* $[\,]$ *y* $(:)$*, aparte del azúcar sintáctico* $[e_1, e_2, \ldots, e_n]$*.*

```
snd :: A -> B -> B          unpack :: (A -> A) -> B
snd X Y = Y                 unpack (snd X) = X

cast :: A -> B              not :: bool -> bool
cast X = unpack (snd X)     not true = false
                            not false = true
```

Este programa utiliza patrones de orden superior *(aplicaciones parciales de símbolos de constructora o funciones a otros patrones) en el lado izquierdo de la regla* $unpack$*. Este tipo de patrones es una característica de los lenguajes lógico-funcionales que no está presente en PF, y que permite distinguir de manera* intensional *distintas descripciones de una misma función* extensional*. Por ejemplo* $(snd\ true)$*,* $(snd\ [\,])$ *e* id *serían tres descripciones diferentes de una misma función identidad, que podrían aparecer en los lados izquierdos de las reglas para distinguir casos. Utilizando una adaptación directa del sistema de tipos DM,* $(snd\ X)$ *tendría tipo* $A \to A$ *con* X *de cualquier tipo* B*, por lo que la función* $unpack$ *tendría tipo* $(A \to A) \to B$*. Considerando este tipo*

para $unpack$, $cast$ *estaría bien tipada con tipo* $A \to B$, *convirtiéndose en una función de* casting polimórfico *[45, 18] que acepta un valor de cualquier tipo y lo devuelve exactamente igual pero con cualquier tipo, posiblemente diferente. Es sencillo ver cómo* $cast$ *destruye la preservación de tipos: la expresión* $not\ (cast\ [\,])$ *está bien tipada, puesto que* $(cast\ [\,])$ *puede tener cualquier tipo (en particular* $bool$*), pero al aplicar las reglas de* $cast$ *y* $unpack$ *obtenemos la reducción:*

$$not\ (\underline{cast\ [\,]}) \longrightarrow not\ (\underline{unpack\ (snd\ [\,])}) \longrightarrow not\ [\,]$$

donde claramente $not\ [\,]$ *está mal tipado. El origen de este problema radica en el propio patrón de orden superior* $(snd\ X)$ *de la regla* $unpack$*, que genera una situación de «*opacidad*». Decimos que el patrón de orden superior* $snd\ X$ *introduce opacidad sobre la variable* X *porque el tipo de esta no queda unívocamente fijado por el tipo del patrón. Nótese que esta opacidad nunca ocurre en los patrones de primer orden (variables o símbolos de constructoras* totalmente *aplicados a patrones de primer orden) usados en PF y en algunos sistemas de PLF como Curry debido a la transparencia de las constructoras, que siempre reflejan el tipo de sus argumentos.*

Aparte del problema del casting *polimórfico, los patrones de orden superior también pueden dar lugar a la llamada* descomposición opaca *[45]. Los diferentes sistemas de PLF proporcionan una función de igualdad estructural con el tipo usual* $A \to A \to bool$*. A diferencia de otras funciones predefinidas, dicha función no puede definirse mediante reglas debido a que estaría mal tipada en una adaptación directa de DM, por lo que los sistemas la implementan de manera* ad-hoc *como una primitiva (==)*[1]*. De esta manera la evaluación de una igualdad de patrones compuestos* $(s\ t_1 \ldots t_n) == (s\ t'_1 \ldots t'_n)$ *se reduce a la conjunción de igualdades sobre sus componentes* $t_1 == t'_1 \wedge \ldots \wedge t_n == t'_n$*. En este escenario, los patrones de orden superior pueden dar lugar a la pérdida de la preservación de tipos. Un ejemplo sencillo es la igualdad* $(snd\ true) == (snd\ [\,])$ *que está bien tipada ya que ambos lados pueden tener el mismo el tipo (por ejemplo* $bool \to bool$*). Sin embargo, aplicando las reglas* ad-hoc *de la igualdad obtendríamos*

$$\underline{(snd\ true) == (snd\ [\,])} \longrightarrow true == [\,]$$

donde la expresión $true == [\,]$ *está mal tipada. Como antes, el problema es producido por la opacidad de los patrones* $snd\ true$ *y* $snd\ [\,]$*, ya que el tipo de su argumento no queda reflejado en el tipo del patrón* $bool \to bool$*. De esta manera podemos comparar patrones del mismo tipo pero que contienen elementos de tipos diferentes, obteniendo errores de tipos debido al comportamiento estructural de la igualdad.*

[1] No es posible definir la igualdad utilizando clases de tipos ya que los sistemas actuales no soportan esta característica: Toy no la contempla, y Curry solo de manera experimental como una rama del sistema Münster Curry Compiler (MCC) [94] o el sistema Zinc Compiler [16], basado en MCC y en fase experimental.

El ejemplo anterior muestra cómo los patrones de orden superior dan lugar a diversos errores de tipos en un marco simple en el que solo se considera la *reescritura* de expresiones. No obstante, cuando consideramos un marco más complejo donde las variables libres de las expresiones se van ligando durante el cómputo a valores adecuados para poder aplicar las reglas del programa (*estrechamiento*), los errores de tipos aparecen de manera aún más sencilla. El siguiente ejemplo muestra algunos de ellos.

Ejemplo 2 (Problemas de tipos con estrechamiento) *Consideremos el siguiente programa escrito con sintaxis Toy.*

```
snd :: A -> B -> B          and :: bool -> bool -> bool
snd X Y = Y                 and true X = X
                            and false X = false
f :: (A -> A) -> bool
f (snd zero) = true
```

En este ejemplo asumiremos que disponemos de las constructoras $zero$ *y* $succ$ *para números naturales de Peano, con tipos* nat *y* $nat \to nat$ *respectivamente. Con estas funciones podemos formar la expresión* $succ\ (F\ zero)$*, que tiene tipo* nat *siempre que* F *tenga tipo* $nat \to nat$*. Sin embargo, es sencillo ver cómo una reducción de estrechamiento que ligue la variable de tipo funcional* F *puede llevar fácilmente a una expresión mal tipada:*

$$succ\ (\underline{F\ zero}) \leadsto_{[F \mapsto and\ false]} succ\ false$$

Este paso liga F *con* $and\ false$ *para aplicar la segunda regla de* and *a la expresión* $and\ false\ zero$*. En esta ocasión el problema radica en que, al tratarse de sistemas estáticamente tipados, no se ha llevado ninguna información de tipos a tiempo de ejecución. A la hora de buscar especulativamente ligaduras para* F *que permitan aplicar alguna regla de programa, el sistema no dispondrá de ninguna información que le permita discriminar entre ligaduras adecuadas e inadecuadas, con lo que puede elegir alguna que no preserve los tipos. Por lo tanto, el sistema ha ligado la variable* F *de tipo* $nat \to nat$ *con el patrón* $and\ false$ *de tipo* $bool \to bool$*, produciendo la expresión* $succ\ false$ *que no admite ningún tipo.*

También es posible encontrar errores de tipos en reducciones de estrechamiento que ligan variables de tipo no funcional, y en las que no intervienen patrones de orden superior. Consideremos, por ejemplo, la expresión $and\ true\ X$*, que tiene tipo* $bool$ *cuando* X *tiene tipo* $bool$*. Utilizando la primera regla de* and *podríamos realizar un paso de estrechamiento que no preserva los tipos:*

$$\underline{and\ true\ X} \leadsto_{[X \mapsto zero]} zero$$

ya que $zero$ *no puede tener tipo* $bool$*. El sistema no tiene información de que* X *es de tipo booleano, y lo liga incorrectamente al valor natural* $zero$*. En este caso se observa que el paso de estrechamiento liga la variable a un valor más concreto de lo que sería*

necesario para aplicar la regla. Considerando la primera regla de and*, sería suficiente con utilizar el* unificador más general *de la expresión y el lado izquierdo de la regla* $[X \mapsto X_1]$ *(siendo* X_1 *una variable fresca) para realizar un paso de estrechamiento:*

$$\underline{and\ true\ X} \leadsto_{[X \mapsto X_1]} X_1$$

Este paso no viola necesariamente la preservación de tipos, que dependería de la suposición de tipos para X_1*. En todo caso, el paso de estrechamiento utilizando la sustitución* $[X \mapsto zero]$ *sería un paso legítimo, pues nada restringe a usar unificadores más generales.*

Estos problemas de tipos que se producen a la hora de ligar variables no funcionales aparecen aún con más facilidad en presencia de patrones de orden superior, incluso utilizando unificadores más generales. Un claro ejemplo es la expresión $[f\ (snd\ X), X]$*, que tiene tipo* $[bool]$ *cuando* X *tiene tipo* $bool$*. Sin embargo, podemos realizar el siguiente paso de estrechamiento:*

$$[\underline{f\ (snd\ X)}, X] \leadsto_{[X \mapsto zero]} [true, zero]$$

donde X *ha sido ligada a* $zero$ *para aplicar la regla de* f*, produciendo la expresión* $[true, zero]$ *que está mal tipada debido a que los dos elementos de la lista son de distinto tipo. En este caso la causa del error es la misma que en los anteriores: el sistema no dispone de información de tipos para discriminar ligaduras adecuadas e inadecuadas de* X*. No obstante, la opacidad del patrón de orden superior* $snd\ zero$ *de la función* f *también juega un papel importante, ya que evita que el tipo* nat *de su argumento quede reflejado en el tipo de la función, permitiendo así que* $f\ (snd\ X)$ *esté bien tipado aun cuando* X *tiene tipo* $bool$*.*

Los anteriores ejemplos, aunque hayan sido presentados usando nociones intuitivas de reducción, muestran problemas de tipos que realmente aparecen en los sistemas de PLF actuales. En particular, los ejemplos se puede comprobar en Toy 2.3.2[2], donde todas las expresiones anteriormente presentadas estarían bien tipados. En la última versión del sistema *Portland Aachen Kiel Curry System*[3] (PAKCS 1.10.0), la versión de referencia de Curry, se produce descomposición opaca. También aparecen los problemas del *casting* polimórfico y de la ligadura de variables libres de primer orden, aunque en estos casos es necesario reformular las funciones que contienen patrones de orden superior en los lados izquierdos de las reglas para que utilicen guardas de igualdad en su lugar, ya que este tipo de patrones no es soportado por Curry[4]. Debido a la estrictez de la igualdad, técnicamente no se obtienen definiciones equivalentes

[2] `http://toy.sourceforge.net/`

[3] `http://www.informatik.uni-kiel.de/~pakcs/`

[4] Aunque los patrones de orden superior no están contemplados en el estándar de Curry [54], el sistema PAKCS sí que soporta algunos de ellos como casos particulares de *patrones funcionales* (*function patterns*) [7] —en general, PAKCS considera como patrón funcional cualquier

a las funciones con patrones de orden superior sino versiones estrictas de ellas; no obstante estas versiones presentan los mismos problemas de tipos. Un ejemplo de la mencionada reformulación se encuentra a continuación, considerando las funciones *unpack* y *f* presentadas en los ejemplos anteriores:

```
unpack :: (a -> a) -> b
unpack x | (x =:= snd y) = y where y free

f :: (a -> a) -> Bool
f x | x =:= (snd Zero) = True
```

(Nótese que Curry utiliza una sintaxis similar a la de Haskell, donde los símbolos de función y variables están en minúsculas, mientras que las constructoras y tipos comienzan en mayúsculas). PAKCS no adolece del problema de la ligadura especulativa de variables libres de orden superior, ya que el mecanismo de cómputo utilizado deja estas ligaduras suspendidas hasta que se ligan por otros medios (por ejemplo mediante una igualdad =:=).

Por todo ello se observa que el campo de los sistemas de tipos en programación lógico-funcional tiene mucho espacio para la investigación, ya que se necesitan sistemas de tipos rigurosamente formalizados que aborden adecuadamente los problemas específicos del paradigma. Esta tesis avanza en esa dirección.

2. Objetivos, contribuciones y estructura de la tesis

2.1. Objetivos de la tesis

El objetivo principal de esta tesis es realizar avances en los sistemas de tipos para programación lógico-funcional. Como se ha comentado, este es un campo que no ha recibido mucho interés de la comunidad lógico-funcional, en el que es posible y deseable mejorar el manejo que se hace de algunas de sus características problemáticas desde el punto de vista de los tipos.

En concreto, los objetivos de esta tesis son:

- Proponer sistemas de tipos rigurosamente formalizados para lenguajes lógico-funcionales y probar su corrección con respecto a las semánticas habituales para estos lenguajes. En particular, estamos interesados en utilizar semánticas operacionales de pequeño paso surgidas recientemente como la *let-reescritura* [90, 92, 131] o el *let-estrechamiento* [91, 92, 131], que proporcionan una descripción muy cercana a cómo evolucionan los cómputos lógico-funcionales.

expresión de la forma $f\ t_1 \ldots t_n$ con $n > 0$ y $\overline{t_n}$ patrones de primer orden [56](§3.2) —. Por tanto, los anteriores ejemplos con patrones de orden superior también serían válidos en PAKCS.

- Estudiar y proporcionar soluciones para manejar correctamente los patrones de orden superior en los lados izquierdos de las reglas, que dan lugar a errores de tipos como el *casting* polimórfico. Como se ha comentado, estos patrones generan una opacidad sobre sus componentes que hace que una aplicación directa del sistema de Damas-Milner no garantice la preservación de tipos durante la evaluación de expresiones. Trabajos anteriores [45] ya consideran este problema, ofreciendo como solución limitar el conjunto de programas considerados a aquellos que no contienen patrones de orden superior de una cierta clase de patrones «problemáticos». Sin embargo, hay algunos usos de estos patrones problemáticos que no comprometen la preservación de tipos. En consecuencia, nuestro objetivo es diseñar un sistema de tipos que acepte programas generales (sin restringir *a priori* los patrones que pueden aparecer) y detecte las situaciones donde los patrones de orden superior pueden producir errores de tipos, obteniendo una solución más general que en [45].

- Investigar posibles modificaciones sobre el sistema de tipos que den lugar a una disciplina de tipos más relajada, sin perder las propiedades de corrección. Actualmente los sistemas de PLF utilizan una adaptación directa del sistema de tipos DM, sistema cuya corrección —*well-typed programs cannot go wrong*— está demostrada con respecto a la semántica denotacional usual utilizada en PF [103]. Al utilizar otras semánticas más adecuadas a la PLF es posible que descubramos limitaciones de DM que se pueden relajar en PLF sin por ello perder la corrección del sistema de tipos.

- Proponer soluciones al problema de la descomposición opaca. Este es un problema que, como se ha visto, aparece de manera natural con la combinación de patrones de orden superior y la igualdad estructural *ad-hoc* de los sistemas de PLF. Este problema también aparece con la igualdad estructural y patrones de primer orden si se permiten *constructoras de datos existenciales* [112, 79] —constructoras cuyo tipo final no refleja el tipo de sus componentes, como por ejemplo la constructora $mkKey$ de aridad 2 y tipo $A \to (A \to nat) \to key$ para construir valores de tipo key—. En cambio, a pesar de la facilidad con la que aparece este problema, no conocemos ningún trabajo en el que se proponga alguna solución al respecto. Incluso en [45], trabajo de referencia sobre sistemas de tipos para PLF y donde se detecta originariamente este problema, sus resultados son correctos bajo la suposición de que no ocurren pasos de descomposición opaca durante la reducción de objetivos. Por ello, otro de los objetivos de esta tesis es estudiar el problema de la descomposición opaca y proponer soluciones para tratarlo.

- Manejar correctamente, desde el punto de vista de los tipos, las *variables extra* —aquellas variables que aparecen solamente en el lado derecho de una regla, sin aparecer en el lado izquierdo—. Estas variables son una característica muy

potente de la PLF, altamente relacionada con las variables libres y el estrechamiento. Mediante ellas, es posible definir funciones complejas de manera sencilla utilizando su potencia expresiva. Un ejemplo de ello es la función *last* que calcula el último elemento de una lista, que puede ser definida de manera concisa utilizando la concatenación de listas: `last Xs = if (Xs == Zs ++ [E]) then E` (es decir, E es el último elemento de la lista Xs si para alguna lista Zs concatenar $[E]$ al final de Zs da lugar a la lista Xs). Existe una íntima relación entre las variables extra y las variables libres, ya que al aplicar funciones con variables extra estas se introducen como variables libres en la expresión a evaluar, debiendo ser ligadas a valores durante el cómputo. A pesar de la gran expresividad de las variables extra, estas han sido usualmente omitidas en los trabajos de tipos para PLF (por ejemplo en [45, 9]). Uno de los objetivos de esta tesis es desarrollar sistemas de tipos para PLF que soporten variables libres en los objetivos y variables extra en las reglas, utilizando para ello una semántica de estrechamiento.

2.2. Contribuciones principales de la tesis

Las contribuciones principales de esta tesis pueden resumirse en:

- El desarrollo del sistema de tipos $\vdash^{\bullet}$, que maneja de manera segura los patrones de orden superior en los lados izquierdos evitando el *casting* polimórfico. Este sistema de tipos garantiza la preservación de tipos bajo las reducciones de let-reescritura. Además, viene acompañado con un algoritmo de inferencia para expresiones y programas. Esta última parte ha sido implementada en Prolog e integrada como una rama experimental del sistema Toy.
- Dentro del sistema de tipos $\vdash^{\bullet}$, se ha clarificado y formalizado los distintos grados de polimorfismo que se pueden asignar a las variables en una let-expresión ($let\ t = e_1\ in\ e_2$). Aunque estos distintos grados no son novedosos, su formalización es un aspecto interesante ya que los distintos sistemas de PF y PLF proporcionan diversos grados de polimorfismo a las let-expresiones sin formalizar (y en algunos casos sin documentar) su elección.
- El desarrollo de un sistema de tipos liberal para PLF que es correcto con respecto a la semántica de let-reescritura. Este sistema de tipos soporta características del estilo de las constructoras de datos existenciales [112, 79], los tipos de datos algebraicos generalizados (GADTs) [28, 134] o las funciones genéricas (permitiendo así la definición por reglas de la igualdad estructural de manera segura, evitando la descomposición opaca). Además, se ha demostrado que este sistema de tipos es lo más liberal posible garantizando preservación de tipos, utilizando un tipado estático. Aunque no se dispone de un algoritmo de inferencia de tipos sí proporciona un método de comprobación de tipos a partir un programa con

anotaciones de tipos para las funciones, método que ha sido implementado en Prolog e integrado en una rama experimental de Toy y en una interfaz web.

- Basado en el sistema de tipos liberal para PLF, se ha desarrollado una traducción para clases de tipos [150, 48] alternativa a la traducción clásica utilizando diccionarios. Esta traducción, en comparación con la clásica, destaca por su sencillez, por resolver problemas de soluciones no computadas que impedían aplicar directamente la traducción clásica —formulada para PF— a los lenguajes PLF y por obtener un rendimiento que puede llegar a ser mejor que el de los diccionarios.
- Desarrollo de un sistema de tipos que garantiza la preservación de tipos con respecto a la semántica de let-estrechamiento, para expresiones con variables libres y reglas con variables extra. Demostramos con precisión que si los pasos de let-estrechamiento se realizan con sustituciones bien tipadas entonces los tipos se preservan. Asegurar que estas sustituciones están bien tipadas requiere en general efectuar comprobaciones de tipos en tiempo de ejecución. Para evitar estas comprobaciones, definimos una clase de programas y un estrechamiento restringido para el cual no serían necesarias. Basándose en este estrechamiento restringido, demostramos que la evaluación de programas Curry mediante *estrechamiento necesario* y *residuación* preserva los tipos.

2.3. Estructura de la tesis

Como se ha dicho, esta tesis sigue el formato por publicaciones según la normativa vigente en la Universidad Complutense de Madrid. La Parte I, que concluye con esta sección, ha presentado la motivación (Sección 1) y los objetivos y principales contribuciones de la tesis (Sección 2). La Parte II contiene una exposición detallada de los aspectos principales del estado actual del tema de la tesis. En la Sección 3 se presenta el paradigma de la programación lógico-funcional, resaltando su potencial expresivo frente a otros paradigmas declarativos mediante ejemplos. En la Sección 4 se explican las distintas alternativas semánticas que se han usado en el paradigma lógico-funcional, las cuales son relevantes para enunciar y demostrar la corrección de los sistemas de tipos. Aparte de ello, en la Sección 5 se exponen los principales sistemas de tipos desarrollados para los lenguajes lógico-funcionales, haciendo mención especial al sistema de tipos de Damas-Milner y a otras extensiones de tipos relevantes en la programación funcional (que jugarán un papel importante en distintas partes de la tesis). La Parte III presenta los sistemas de tipos desarrollados en esta tesis: el sistema $\vdash^{\bullet}$ (Sección 6), el sistema de tipos liberal (Sección 7) y el sistema de tipos con soporte para variables extra y estrechamiento (Sección 8). Cada una de estas secciones contiene una parte de introducción y motivación, donde se introduce el sistema de tipos y se sitúa con respecto al estado del arte y el resto de sistemas propuestos; la presentación del sistema de tipos en sí, junto con sus propiedades y aplicaciones; y un apartado de conclusiones,

donde se resumen los objetivos conseguidos y las limitaciones del sistema de tipos, enlazándolo con los demás sistemas de tipos propuestos en la tesis. La Parte IV recoge, de manera unificada, las principales conclusiones de los distintos sistemas de tipos propuestos en la tesis (Sección 9). También incluye diferentes líneas de trabajo futuro (Sección 10). Por último, la Parte V contiene las publicaciones asociadas a la tesis en su formato y longitud original. La Sección A contiene las las publicaciones de primer nivel que forman parte de la tesis y que avalan la calidad de los resultados de la misma. Por otro lado, la Sección B contiene versiones extendidas de algunos artículos de la Sección A, para disponer, dentro de la propia tesis, de las demostraciones a todos los resultados presentados.

Las citas a referencias bibliográficas utilizan el formato ACM, por lo que están formadas por una sucesión de números entre corchetes. Para referirnos a partes concretas dentro de las publicaciones, incluiremos entre paréntesis la sección o enunciado específico. Por ejemplo:

- [n] — Cita a la publicación *n*.
- [n](§X) — Cita a la sección *X* de la publicación *n*.
- [n](Def. X) — Cita a la definición *X* de la publicación *n*. Además de definiciones, se pueden citar teoremas (*Th.*), lemas (*Lemma*), proposiciones (*Prop.*), corolarios (*Cor.*), ejemplos (*Ex.*) o figuras *(Fig.)*.

Cuando las citas traten publicaciones asociadas a la tesis, incluiremos entre los paréntesis el número del apéndice que lo contiene. De esta manera tendremos citas como:

- [n](A.m) — Cita a la publicación asociada *n* en el apéndice *A.m*.
- [n](B.m) — Cita a la versión extendida *n* en el apéndice *B.m*.
- [n](A.m, §X) — Cita a la sección *X* de la publicación asociada *n* en el apéndice *A.m*.
- [n](B.m, Def. X) — Cita a la definición *X* de la versión extendida *n* en el apéndice *B.m*.

Este tipo de citas se utilizará de manera intensiva en los títulos de los enunciados de la tesis como definiciones, teoremas, ejemplos, etc., con el fin de enlazarlos con la publicación asociada en la que aparecen.

Parte II
Estado del arte

En esta parte presentaremos el estado actual de los dos principales temas de esta tesis: la programación lógico-funcional y los sistemas de tipos (para lenguajes funcionales y lógico-funcionales). A la vez que exponemos ambos temas se introducirá gran parte de la notación y preliminares que usaremos en las siguientes secciones, que presentan e integran las aportaciones de los distintos artículos que componen esta tesis.

3. Programación lógico-funcional

La programación lógico funcional [8, 55, 130, 53] es un paradigma de programación que surge de la combinación de las principales clases de paradigmas declarativos. Estos paradigmas declarativos difieren del popular paradigma imperativo en que los programas describen *cuáles* son las propiedades del problema y de las soluciones válidas, en lugar de *cómo* hay que calcular la solución al problema paso a paso. Dentro del paradigma declarativo, se pueden distinguir tres clases principales:

- *Programación lógica*: se basa en un subconjunto de la lógica de predicados (cláusulas de Horn). Su método de evaluación es la solución de objetivos mediante el procedimiento de resolución, que para el mencionado subconjunto es eficiente. Tiene interesantes características como el cálculo con información parcial (utilizando para ello *variables lógicas*: variables libres que se van ligando a valores adecuados según avanza el cómputo) o la búsqueda indeterminista de soluciones por medio de un mecanismo de vuelta atrás. Entre los lenguajes de programación que adoptan este paradigma destaca Prolog [68, 34].
- *Programación funcional*: se basan en el λ-cálculo y en la reescritura de términos y grafos. En este paradigma las funciones se describen como ecuaciones que se utilizan de izquierda a derecha para evaluar las expresiones. Tiene características muy interesantes, como las funciones de orden superior (funciones que aceptan funciones como argumentos y pueden utilizarlas en sus cuerpos), un sistema de tipos estático que asegura que los cómputos no produzcan errores de tipos o la posibilidad de definir funciones polimórficas que funcionan para diversos tipos diferentes de manera uniforme. Ejemplos de lenguajes que adoptan este paradigma son Lisp [140], ML [104], Haskell [115, 65], Clean [21, 122] o F# [102, 101, 143].
- *Programación con restricciones*: en este paradigma las relaciones entre las variables se efectúan por medio de restricciones, y el mecanismo de cómputo es la búsqueda de soluciones por medio de un resolutor. Estas restricciones trabajan

sobre diferentes dominios específicos, como pueden ser los enteros, los reales o conjuntos finitos, que son manejados por resolutores especializados de manera muy eficiente. Aunque existen sistemas específicos que permiten modelizar y resolver problemas a partir de la colección de restricciones (por ejemplo las diferentes herramientas de IBM ILOG CPLEX [67]), este paradigma suele integrarse con otros como la programación imperativa u orientada a objetos (mediante bibliotecas externas que realizan la resolución de restricciones, como el citado IBM ILOG CPLEX [67] o Gecode [135, 136]) o la programación lógica [69] (por ejemplo en sistemas como SICStus Prolog[5] [26] o SWI Prolog[6] [153]).

Por todo ello, la combinación de estos paradigmas ha sido un tema que ha suscitado un gran interés durante las últimas décadas. Ahora bien, los distintos paradigmas tienen características cuya interacción es compleja, por lo que han surgido diferentes propuestas para realizar la combinación y diferentes lenguajes de programación que las implementan. Aunque la programación con restricciones es una componente interesante en la programación lógico-funcional, en el resto de la tesis omitiremos las referencias a ellas ya que no juega ningún papel en los trabajos que la componen.

Desde el punto de vista de la integración de los paradigmas lógico y funcional se pueden seguir dos aproximaciones: una es tomar como base un lenguaje lógico y extenderlo con características funcionales, mientras que la otra es tomar como base un lenguaje funcional y extenderlo con características lógicas. Dentro de la primera familia está incluido Ciao Prolog [22], que proporciona azúcar sintáctico para definir funciones que son traducidas a relaciones mediante un preprocesador. Mercury [139] también se incluye en esta primera familia, aunque su orientación hacia una arquitectura altamente eficiente le impide tener características típicamente lógicas como el cómputo con información parcial. Por otro lado, las características lógicas pueden ser integradas en un paradigma funcional mediante la combinación del mecanismo de resolución con la evaluación de funciones, intentando mantener la eficiencia de la estrategia de evaluación perezosa de los lenguajes funcionales. Dentro de esta familia podemos destacar a Escher [81], Curry [54] o Toy [93, 23]. En Escher, las llamadas a función son suspendidas si no están suficientemente instanciadas, por lo que las variables libres en estas llamadas no son ligadas a valores. Curry y Toy superan esta limitación, permitiendo que las funciones se apliquen a argumentos con variables que son instanciadas adecuadamente para poder aplicar una regla de programa. Este método, conocido como estrechamiento, combina el concepto funcional de reducción con el lógico de unificación y búsqueda no determinista.

Como se ha comentado, los lenguajes lógico-funcionales proporcionan una gran expresividad en comparación con los lenguajes imperativos, permitiendo a los programadores centrarse en el *qué* en lugar del *cómo*. Sin embargo, las características de la

[5] `http://www.sics.se/sicstus`
[6] `http://www.swi-prolog.org`

combinación hacen también que sea más expresivo que sus componentes lógica y funcional por separado, e incluso más eficiente en algunas ocasiones. Problemas donde los lenguajes lógico-funcionales son especialmente adecuados son los conocidos como de *generación y comprobación* (*generate and test*), donde se generan valores candidatos a solución paso a paso de manera indeterminista y se comprueba si cumplen las condiciones que caracterizan a las soluciones válidas. Un ejemplo de este tipo de problemas es la ordenación por permutación para listas de naturales (extraído de [43]):

Ejemplo 3 *Ordenación por permutación*

```
insert :: A -> [A] -> [A]
insert X Ys = [X|Ys]
insert X [Y|Ys] = [Y|insert X Ys]

permute :: [A] -> [A]
permute [] = []
permute [X|Xs] = insert X (permute Xs)

leq :: nat -> nat -> bool
leq zero Y = true
leq (succ X) zero = false
leq (succ X) (succ Y) = leq X Y

sorted :: [nat] -> bool
sorted [] = true
sorted [X] = true
sorted [X,X2|Xs] = if (leq X X2) /\ sorted [X2|Xs] then true

check, permutsort :: [nat] -> [nat]
check L = if (sorted L) then L
permutsort L = check (permute L)
```

Como se puede ver, la función $insert$ es una función que indeterministamente inserta un elemento en una lista. Basándose en esta función, $permute$ genera permutaciones de la lista insertando de manera indeterminista el elemento X en la cabeza de la lista dentro de la cola Xs de la lista. La función leq es la comparación de menor o igual *sobre números naturales de Peano (formados con las constructoras $zero$ y $succ$). La función $sorted$ comprueba que la lista pasada como argumento está ordenada —utilizando la conjunción booleana (`/\`)—, devolviendo $true$ en dicho caso. Nótese que, debido al uso de la función predefinida if_then, esta función no devuelve nada si la lista no está ordenada, ya que el caso de comprobar que una lista no está ordenada no es necesario en este esquema de programación. La función $check$ actúa como la identidad para listas ordenadas. Por último, $permutsort$ devuelve las permutaciones de lista original generadas con $permute$ que están ordenadas.*

Aunque en realidad se trata de un método de ordenación altamente ineficiente, el ejemplo sirve para mostrar cómo la combinación de indeterminismo (proveniente de la programación lógica) y la evaluación perezosa (proveniente de la programación funcional) mejoran la eficiencia de este tipo de esquemas de programación, llegando incluso a mejorar su orden de complejidad [55]. Si considerásemos un lenguaje lógico puro como Prolog, el predicado de ordenación por permutación quedaría como:

```
permutationSort(L,L2) :- permute(L,L2), sorted(L2).
```

En este caso cada lista candidata debe ser completamente generada antes de comprobar si está ordenada, por lo que para encontrar una solución debería generar completamente todas las permutaciones anteriores (según el orden de aplicación de las reglas de $permute$). En programación funcional el método clásico es el de *lista de éxitos*: generar una lista con todos los candidatos y filtrar aquellos que están ordenados. En este caso, encontrar una solución requeriría generar *todas* las posibles permutaciones. Aunque gracias a la evaluación perezosa cada permutación inválida sería generada solo hasta el punto que el filtro pueda rechazarla por no estar ordenada,

4. Semánticas para programación lógico-funcional

La semántica de un lenguaje de programación es una definición formal y rigurosa de cuál es el *significado* de las distintas construcciones del lenguaje. Disponer de una semántica es imprescindible a la hora de razonar sobre los programas y demostrar la corrección de transformaciones de programa o del sistema de tipos. Aunque en algunos lenguajes populares (léase C/C++/C# o Java) esta semántica no siempre está completamente detallada, en los lenguajes declarativos es común tener varias semánticas formales que detallan el modelo de cómputo desde diversos puntos de vista (principalmente operacional y denotacional). En esta sección presentaremos las semánticas más importantes que se han utilizado para los lenguajes lógico-funcionales.

Antes de presentar las distintas opciones semánticas, comentaremos dos aspectos importantes que estas deben tratar: uno es qué ocurre cuando se pasan argumentos indeterministas a las funciones, y otro es su grado de estrictez. Con respecto al primer aspecto existen dos alternativas fundamentalmente (aunque recientemente han surgido nuevas propuestas [127]): *call-time choice* y *run-time choice* [66][7]. El siguiente ejemplo sirve para comprender la diferencia entre las dos opciones:

Ejemplo 4 (Paso de parámetros indeterministas) *Consideremos el programa:*

```
coin :: nat              dup :: A -> (A,A)
coin = zero              dup X = (X,X)
coin = succ zero
```

[7] En esta tesis usaremos su nombre en inglés debido a que no existe una traducción al español ampliamente aceptada.

donde $coin$ es una función indeterminista que modela el lanzamiento de una moneda, y dup es una función de duplica el argumento que se le pasa.

A la hora de evaluar la expresión $dup\ coin$ se puede decidir que cada copia de $coin$ generada por la aplicación de dup pueda evolucionar de manera independiente, por lo que esta expresión tendría cuatro posibles valores: $(zero, zero)$, $(zero, succ\ zero)$, $(succ\ zero, zero)$ y $(succ\ zero, succ\ zero)$. Esta opción corresponde con *run-time choice*. Por otro lado, se puede decidir que todas las copias de expresiones indeterministas hechas durante el paso de parámetros deben evolucionar de la misma manera. De esta manera la evaluación de $dup\ coin$ solo podría alcanzar dos valores: $(zero, zero)$ y $(succ\ zero, succ\ zero)$. Esta opción, conocida como *call-time choice*, surge de forma natural al adoptar el mecanismo de compartición (*sharing*) de parámetros implementado en algunos lenguajes de programación. Aunque las dos opciones para el paso de parámetros son válidas para los lenguajes lógico-funcionales, *call-time choice* es la opción más natural y la que menos «asombro» puede causar al programador (ver [55] para más detalles).

Con respecto al aspecto de la estrictez, las semánticas se pueden clasificar en dos grupos. Se dice que una semántica es *estricta* cuando todos los argumentos de una función deben estar completamente evaluados para poder aplica dicha función. Por otro lado, una semántica es *no estricta* cuando la aplicación de funciones se puede llevar a cabo aun cuando algunos argumentos no estén completamente evaluados. La diferencia se puede observar en el siguiente ejemplo:

Ejemplo 5 (Funciones estrictas) *Consideremos el programa:*

```
loop :: A          f :: A -> bool       g :: bool -> A -> bool
loop = loop        f X = true           g true Y = false
```

$loop$ es una función cuya evaluación nunca termina, y f es una función constante que acepta cualquier elemento y devuelve $true$. Por otro lado, g acepta dos argumentos de los cuales el primero debe ser $true$ y el siguiente es ignorado, devolviendo $false$.

En una semántica estricta la expresión $f\ loop$ no estaría definida, ya que la evaluación del argumento $loop$ nunca termina. En cambio, bajo un semántica no estricta la evaluación de $f\ loop$ devolvería $true$ aun cuando el argumento $loop$ no se puede evaluar completamente. Dentro de las semánticas no estrictas, puede haber funciones que son estrictas en algunos de sus argumentos. Por ejemplo la función f anterior no es estricta en su primer argumento. Sin embargo, la función g sería estricta en su primer argumento (se requiere su evaluación para poder aplicar la regla) y no estricta en su segundo argumento.

Aparte de la opciones sobre el paso de parámetros y la estrictez, también existen otros aspectos a tratar como la visión de las elecciones indeterministas. El lector interesado puede encontrar una discusión en profundidad sobre estos aspectos en [144]. En

$$
\begin{array}{rcl}
Exp \ni e & ::= & \bot \mid X \mid c \mid f \mid e\ e \\
Pat \ni t & ::= & \bot \mid X \mid c\ t_1 \ldots t_n \text{ si } n \leq ar(c) \mid f\ t_1 \ldots t_n \text{ si } n < ar(f) \\
PSubst \ni \theta & ::= & [\overline{X_n \mapsto t_n}] \\
R & ::= & f\ \bar{t} \to e \text{ (no contiene } \bot, \bar{t} \text{ lineal)} \\
\mathcal{P} & ::= & \{R_1, \ldots, R_n\}
\end{array}
$$

Figura 2: Sintaxis de las expresiones y programas CRWL

esta tesis consideraremos un marco lógico-funcional como el contemplado por los lenguajes Toy y Curry, que adoptan una semántica no estricta y con *call-time choice*.

4.1. La lógica de reescritura CRWL

La lógica de reescritura condicional basada en constructoras (CRWL según las siglas de su nombre en inglés *Constructor-based conditional ReWriting Logic*) es un marco semántico ampliamente aceptado en la comunidad lógico-funcional [55]. CRWL proporciona una cálculo para computar los valores a los que se puede reducir una expresión, soportando *call-time choice*, indeterminismo y funciones no estrictas. Fue originariamente propuesto en [46, 43] para un lenguaje funcional indeterminista de primer orden, y posteriormente extendido para orden superior en [44]. En estas formalizaciones, las reglas de programa aparecen acompañadas con condiciones de *c-convergencia* $e \bowtie e'$ (*joinability* en inglés) que se satisfacen únicamente cuando e y e' pueden ser reducidas al mismo valor totalmente definido t. No obstante, salvo por ciertas cuestiones operacionales de importancia secundaria aquí, puede probarse [132] que dichas condiciones pueden reemplazarse por el uso de funciones ordinarias. Por ello, en esta tesis consideraremos solo reglas sin condiciones y sin sentencias de c-convergencia. De la misma manera, consideraremos solo el caso de CRWL de orden superior (también conocido como HO-CRWL) ya que los trabajos que componen esta tesis utilizan únicamente marcos de orden superior.

CRWL considera una signatura $\Sigma = CS \cup FS$ formada por símbolos de constructora CS[8] y función FS. Los símbolos de constructora y función se denotan con la letra $c \in CS$ y $f \in FS$ respectivamente, teniendo una aridad de programa asociada. Si un símbolo de constructora c tiene aridad n se expresa como $ar(c) = n$ o $c \in CS^n$, y de manera similar si un símbolo de función f tiene aridad m se expresa como $ar(f) = m$ o $f \in FS^m$. Para representar a un símbolo de Σ sin importar si es constructora o función utilizaremos la letra h. También se considerará un conjunto infinito numerable de variables de datos $X, Y, Z \ldots \in \mathcal{DV}$. Para poder manejar adecuadamente la no estrictez, se considera una constructora especial $\bot \in CS^0$ que representa el valor *indefinido*. Con los símbolos anteriores se pueden forman las *expresiones* $e, r \in Exp$ y los *patro-*

[8] En algunas de las publicaciones asociadas a esta tesis se utiliza DC para referirse al conjunto de símbolos de constructora.

nes $t, p \ldots \in Pat$, con la sintaxis que aparece en la Figura 2. Los patrones son la noción de valores, y como puede observarse en la figura se verifica $Pat \subseteq Exp$. Diremos que una expresión o patrón es *parcial* cuando contenga $\bot$, y *total* en otro caso. La notación $\overline{o_n}$ expresa la secuencia de n elementos sintácticos $o_1, \ldots, o_n$, siendo simplificada a $\overline{o}$ cuando el número exacto de elementos no importa.

Es interesante dividir el conjunto de patrones Pat en dos: los *patrones de primer orden*, definidos como $FOPat \ni fot ::= X \mid c\ fot_1 \ldots fot_n$ donde $c \in CS^n$; y los *patrones de orden superior* $HOPat = Pat \setminus FOPat$. A diferencia de lo que ocurre en los lenguajes funcionales clásicos, en este marco no sólo los patrones de primer orden sino también los de orden superior son tratados como verdaderos valores (*first class citizens*). Los patrones de orden superior representan funciones desde un punto de vista *intensional*, permitiendo distinguir distintas representaciones de la misma función *extensional*. El siguiente ejemplo (tomado de [45]) muestra un programa que explota este tipo de patrones para representar circuitos booleanos binarios:

Ejemplo 6 (Patrones de orden superior) *Consideremos el siguiente programa, donde* add *es la función de suma de naturales,* $circuit$ *es un sinónimo de* $bool \to bool \to bool$ *y (/\), (\/) son la conjunción y disyunción booleanas respectivamente:*

```
x1, x2 :: circuit
x1 X Y = X
x2 X Y = Y

notGate :: circuit -> circuit
notGate C X Y = not (C X Y)

andGate, orGate :: circuit -> circuit -> circuit
andGate C1 C2 X Y = (C1 X Y) /\ (C2 X Y)
orGate C1 C2 X Y = (C1 X Y) \/ (C2 X Y)

size :: circuit -> nat
size x1 = zero
size x2 = zero
size (notGate C) = succ (size C)
size (andGate C1 C2) = succ (add (size C1) (size C2))
size (orGate C1 C2) = succ (add (size C1) (size C2))
```

Como puede verse, $x1, x2 \in FS^2$, $notGate \in FS^3$ *y* $andGate, orGate \in FS^4$, *por lo que los patrones utilizados para definir* $size$ *son patrones de orden superior válidos (pues se trata de aplicaciones parciales). Aunque los patrones*

$$t_1 \equiv notGate\ (orGate\ x1\ x2)$$
$$t_2 \equiv andGate\ (notGate\ x1)\ (notGate\ x2)$$

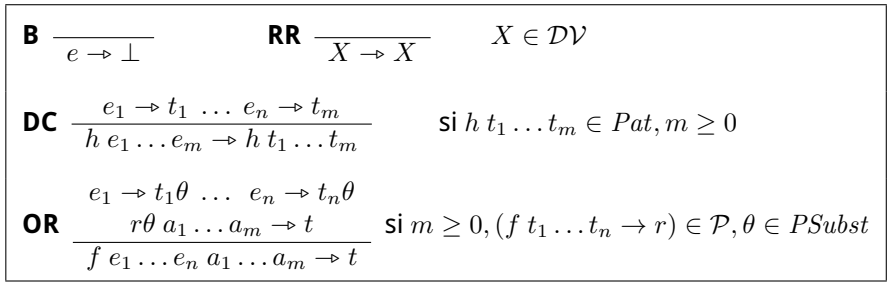


Figura 3: Reglas del cálculo CRWL

representan circuitos —funciones, por tanto— que se comportan igual para todas las entradas, son dos representaciones intensionales distintas que son diferenciados por la función $size$*. Por ello* $size\ t_1$ *se evaluaría a* $succ\ (succ\ zero)$*, mientras que* $size\ t_2$ *se evaluaría a* $succ\ (succ\ (succ\ zero))$*.*

Una *sustitución de patrones* $\theta \in PSubst$ es una aplicación finita de variables de datos a patrones $\mathcal{DV} \to Pat$, que se extiende de manera natural a una aplicación de expresiones a expresiones $Exp \to Exp$. La aplicación de sustituciones a expresiones se escribe como $e\theta$, y la composición de sustituciones $\theta_1\theta_2$ se define tal que $e(\theta_1\theta_2) = (e\theta_1)\theta_2$. Para detallar las sustituciones se utiliza la notación $\theta \equiv [X_1 \mapsto t_1, \ldots, X_n \mapsto t_n]$[9] que satisface que $X_i\theta \equiv t_i$ y $Z\theta \equiv Z$ para toda variable $Z \in \mathcal{DV} \setminus \{\overline{X_n}\}$. Nótese que en CRWL solo se consideran sustituciones donde el rango son patrones, no expresiones arbitrarias. Esto es importante a la hora de respetar el paso de parámetros por *call-time choice*, como se verá más adelante. Un programa $\mathcal{P}$ es un conjunto de reglas $R \equiv f\ t_1 \ldots t_n \to e$ que cumplen que $\overline{t_n}$ es lineal (no contiene múltiples apariciones de la misma variable) y que $\bot$ no aparece en R. Diremos que una variable del lado derecho $X \in var(e)$ es una *variable extra* de la regla $f\ t_1 \ldots t_n \to e$ si $X \notin var(f\ t_1 \ldots t_n)$. En principio CRWL no impone ninguna restricción sobre las variables de los lados derechos, permitiendo por tanto reglas con variables extra.

CRWL proporciona un cálculo para derivar reducciones del tipo $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$, que informalmente significa que t aproxima un posible valor para la evaluación de e usando $\mathcal{P}$. Cuando el programa $\mathcal{P}$ quede claro por el contexto, la notación se abreviará a $e \twoheadrightarrow t$. La Figura 3 muestra las reglas del cálculo CRWL. La regla B (*bottom*) evita la evaluación de una expresión, reduciéndola a $\bot$. Esta regla, en combinación con OR, es importante para conseguir una semántica no estricta. La regla RR (*restricted reflexivity*) permite la reducción de una variable a ella misma, y la regla DC (*decomposition*) descompone la evaluación de un patrón en la evaluación de sus componentes. Por último, la regla OR (*outer reduction*) realiza la aplicación de funciones respetando *call-time choice* y la no estrictez. Primero se reducen los argumentos $\overline{e_n}$ a patrones $\overline{t_n\theta}$ y luego

[9] En algunos artículos que componen esta tesis se usa la notación alternativa $[X_1/t_1, \ldots, X_n/t_n]$.

se aplica la instancia de la función $f\ t_1\theta \ldots t_n\theta \to r\theta$. Como $\theta \in PSubst$, las variables $\{\overline{X_m}\} = var(f\ t_1 \ldots t_n)$ de la función utilizada $f\ t_1 \ldots t_n \to r$ tomarán como valores patrones ($X_1\theta, \ldots, X_m\theta \in Pat$). De esta manera se respeta la opción de *call-time choice* ya que las distintas apariciones de las variables en r serán sustituidas por los mismos patrones, que son expresiones irreducibles y por tanto no podrán producir valores diferentes. Esto se puede observar en el siguiente ejemplo:

Ejemplo 7 (Derivaciones CRWL) *Consideremos los símbolos y reglas del Ejemplo 4 (página 16). Una posible reducción CRWL para la expresión $dup\ coin$ sería:*

$$\text{OR}\ \dfrac{\text{OR}\ \dfrac{\text{DC}\ \dfrac{}{zero \twoheadrightarrow zero}}{coin \twoheadrightarrow zero} \qquad \text{DC}\ \dfrac{\text{DC}\ \dfrac{\text{DC}\ \dfrac{}{zero \twoheadrightarrow zero}}{zero \twoheadrightarrow zero}}{(zero, zero) \twoheadrightarrow (zero, zero)}}{dup\ coin \twoheadrightarrow (zero, zero)}$$

Primero se reduce el argumento $coin$ al patrón $zero$ utilizando la primera regla de $coin$, y luego se realiza el paso de parámetros mediante la sustitución $[X \mapsto zero] \in PSubst$, dando lugar al resultado $(zero, zero)$. De manera similar se obtendría $dup\ coin \twoheadrightarrow (succ\ zero, succ\ zero)$, realizando la reducción $coin \twoheadrightarrow succ\ zero$ y utilizando la sustitución $[X \mapsto succ\ zero] \in PSubst$. Sin embargo, no sería posible obtener el resultado $(zero, succ\ zero)$, que viola la opción del call-time choice*, ya que $coin$ debe ser evaluado a un patrón antes de aplicar la función.*

El hecho de que los argumentos de una función se puedan evaluar a patrones parciales es imprescindible para conseguir una semántica no estricta. Un ejemplo de esto se puede encontrar en la siguiente derivación CRWL, que utiliza el programa del Ejemplo 5 (página 17):

$$\text{OR}\ \dfrac{\text{DC}\ \dfrac{\text{B}\ \dfrac{}{loop \twoheadrightarrow \bot}}{succ\ loop \twoheadrightarrow succ\ \bot} \qquad \text{DC}\ \dfrac{}{true \twoheadrightarrow true}}{f\ (succ\ loop) \twoheadrightarrow true}$$

Obsérvese que esta reducción solo es posible en semánticas no estrictas ya que el argumento $succ\ loop$ no está definido debido a la función no terminante $loop$. Para aplicar la regla OR primero se reduce el argumento $succ\ loop$ al patrón parcial $succ\ \bot$ utilizando la regla B, que evita la evaluación de la expresión $loop$. Luego se realiza el paso de parámetros con la sustitución $[X \mapsto succ\ \bot] \equiv \theta \in PSubst$, obteniendo como resultado $true\theta \equiv true$. Si solo se considerasen sustituciones de patrones totales en $PSubst$ esta derivación no sería posible, ya que no es posible reducir $succ\ loop$ a un patrón total.

Debido al indeterminismo y a que la lógica CRWL calcula aproximaciones al valor de las expresiones, una expresión se puede evaluar a varios patrones con respecto a CRWL. Por ejemplo la expresión $dup\ coin$ puede ser reducida a $\bot$, $(\bot, \bot)$, $(zero, \bot)$, etc. Al conjunto de todos los patrones a los que se puede reducir una expresión e con

respecto a un programa $\mathcal{P}$ se le llama la *denotación de* e, definida como

$$[\![e]\!]_{\mathcal{P}} = \{t \in Pat \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$$

El cálculo de la Figura 3 no puede entenderse como un mecanismo operacional para ejecutar programas, sino como una manera de describir el significado de programas y expresiones. Para llenar este vacío, la semántica de CRWL de orden superior presentada en [44] también propone un cálculo de estrechamiento perezoso (CLNC según las siglas en inglés de *Constructor-based Lazy Narrowing Calculus*) para la resolución de objetivos. Este cálculo de pequeño paso $G \Vdash G'$ opera sobre objetivos de la forma $G \equiv \exists\overline{U}.S \,\square\, P \,\square\, E$ formados por un conjunto de variables existenciales U, un conjunto de ecuaciones S (parte resuelta), un conjunto de condiciones de aproximación P y un conjunto de condiciones de c-convergencia E. Además, CLNC es correcto y completo con respecto al cálculo de soluciones (objetivos de la forma $\equiv \exists\overline{U}.S \,\square\, \square$ que representan sustituciones adecuadas con respecto a CRWL y el objetivo inicial). Sin embargo, CLNC es un cálculo complejo que no captura completamente la intuición de lo que realiza un cómputo lógico-funcional. Las semánticas de let-reescritura y let-estrechamiento surgen de esa necesidad de proporcionar una noción más sencilla de paso de cómputo.

4.2. Let-reescritura y let-estrechamiento

Las semánticas de *let-reescritura* y *let-estrechamiento* desarrolladas en [90, 91, 92] proporcionan una noción sencilla de paso de cómputo lógico-funcional para la reescritura y el estrechamiento respectivamente, a la vez que soportan funciones indeterministas no estrictas y respetan el paso de parámetros por *call-time choice*. Por todo ello han sido elegidas como marco semántico para demostrar la corrección de los sistemas de tipos que presentaremos en esta tesis. Estas semánticas se basan en las mismas expresiones soportadas por CRWL pero las extienden con construcciones *let* para expresar la compartición de subexpresiones. Gracias a estas construcciones de compartición, inspiradas en [10, 132], se respeta *call-time choice* a la vez que se consigue una semántica no estricta. La let-reescritura y el let-estrechamiento fueron presentados por primera vez para el marco de primer orden en [90] y [91] respectivamente, siendo extendidos a orden superior en [92]. En esta tesis consideraremos solamente el caso de orden superior ya que es el que se utiliza en los diferentes artículos que la componen.

En estas semánticas se considera una signatura Σ y un conjunto de variables de datos $\mathcal{DV}$ similar a los de CRWL. En este marco sólo consideraremos patrones y expresiones totales, ya que la constructora $\bot$ no es necesaria. En lugar de utilizar $\bot$ para descartar la evaluación de expresiones innecesarias y conseguir así funciones no estrictas, en este marco se utilizarán ligaduras para extraer dichas expresiones innecesarias y cambiarlas por variables, que serán desechadas posteriormente (ver Ejemplo 8, pági-

Variable de datos	$\mathcal{DV}$		$X, Y, Z, \ldots$
Constructora de datos	CS		c
Símbolo de función	FS		$f, g, \ldots$
Símbolo	s	::=	$X \mid c \mid f$
Símbolo no variable	h	::=	$c \mid f$
Expresión	$Exp \ni e, r$	::=	$X \mid c \mid f \mid e\ e \mid let\ X = e\ in\ e$
Patrón	$Pat \ni t$	::=	X
		$\mid$	$c\ t_1 \ldots t_n$ si $n \leq ar(c)$
		$\mid$	$f\ t_1 \ldots t_n$ si $n < ar(f)$
Contexto	$Cntxt \ni \mathcal{C}$	::=	$[\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$
		$\mid$	$let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$
Sustitución de patrones	$PSubst \ni \theta$	::=	$[\overline{X_n \mapsto t_n}]$
Regla de programa	R	::=	$f\ \bar{t} \to e$ ($\bar{t}$ lineal)
Programa	$\mathcal{P}$	::=	$\{R_1, \ldots, R_n\}$

Figura 4: Sintaxis de las expresiones y los programas

na 25, para más detalles). La sintaxis de los patrones Pat no cambia, a diferencia de las expresiones, que son extendidas con construcciones *let*. La Figura 4 muestra un resumen de la sintaxis de expresiones y programas en let-reescritura y let-estrechamiento, que será la sintaxis utilizada en el resto de la tesis. Distinguiremos distintas expresiones según su sintaxis. Las expresiones $c\ e_1 \ldots e_n$ se llaman *junk* (basura) si $n > ar(c)$, puesto que no podrán producir ningún valor útil. Las expresiones $f\ e_1 \ldots e_n$ se llaman *activas* si $n \geq ar(f)$, ya que se les ha proporcionado todos los argumentos que necesitan para ser aplicadas. Las expresiones $X\ e_1 \ldots e_n$ (con $n \geq 0$) se llaman *flexibles* (*aplicaciones de variable* si $n > 0$), pues el operador principal es una variable. Por último, las expresiones $let\ X = e_1\ in\ e_2$ se llaman let-expresiones, debido a que tienen una construcción *let* en la parte más externa. El conjunto $fv(e)$[10] de *variables libres* de una expresión e se define como el conjunto de variables en e que no están ligadas por ninguna construcción *let*. Las variables libres de las let-expresiones se definen como $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$, correspondiendo a que no se consideran let-expresiones recursivas. Esto no es una limitación ya que en este marco, a diferencia del marco funcional, las let-expresiones no se utilizan para definir funciones sino que solo realizan compartición de subexpresiones. Por su parte, el conjunto de *variables ligadas* $bv(e)$ de una expresión se define como:

$$
\begin{aligned}
&bv(s) = \emptyset \\
&bv(e_1\ e_2) = bv(e_1) \cup bve_2 \\
&bv(let\ X = e_1\ in\ e_2) = bv(e_1) \cup bv(e_2) \cup \{X\}
\end{aligned}
$$

[10] En algunos artículos de esta tesis este conjunto se denomina $FV(e)$.

(Fapp) $f\ t_1\theta \ldots t_n\theta \ \rightarrow^l \ r\theta$, si $(f\ t_1 \ldots t_n \rightarrow r) \in \mathcal{P}$

(LetIn) $e_1\ e_2 \rightarrow^l let\ X = e_2\ in\ e_1\ X$, si e_2 es una expresión *junk*, activa, una aplicación de variable o una let-expresión; para X fresca

(Bind) $let\ X = t\ in\ e\ \rightarrow^l\ e[X/t]$

(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^l e_2$, si $X \notin fv(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightarrow^l\ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, si $Y \notin fv(e_3)$

(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let\ X = e_1\ in\ e_2\ e_3$, si $X \notin fv(e_3)$

(Contx) $\mathcal{C}[e]\ \rightarrow^l\ \mathcal{C}[e']$, si $\mathcal{C} \neq [\ ]$, $e\ \rightarrow^l\ e'$ usando alguna de las reglas anteriores, y en caso de que $e \rightarrow^l e'$ use (Fapp) con la regla $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$ y $\theta \in PSubst$ entonces $vran(\theta|_{\setminus var(\overline{p})}) \cap bv(C) = \emptyset$

Figura 5: Relación $\rightarrow^l$ de let-reescritura

Para referirnos al conjunto de *variables* de una expresión, usaremos $var(e)$. Como los patrones no contienen construcciones let, se cumple que $var(t) = fv(t)$. Los *contextos* $\mathcal{C} \in Cntxt$ son expresiones con un solo «hueco». La aplicación de una expresión e a un contexto $\mathcal{C}$, escrito como $\mathcal{C}[e]$, significa colocar la expresión e en el hueco que tiene el contexto $\mathcal{C}$.

En este marco solo consideraremos sustituciones de variables de datos por patrones totales $PSubst \ni \theta \equiv [\overline{X_n \mapsto t_n}]$. Sobre estas haremos uso de dos funciones estándar: el *dominio* $dom(\theta) = \{X \in \mathcal{DV} \mid X\theta \not\equiv X\}$ y las *variables en el rango* $vran(\theta) = \bigcup_{X \in dom\theta} var(X\theta)$[11]. La *restricción de una sustitución* θ a un conjunto de variables $A \subseteq \mathcal{DV}$ se escribe como $\theta|_A$, utilizando la notación $\theta|_{\setminus A}$ como sinónimo de $\theta|_{(\mathcal{DV} \setminus A)}$. La sustitución vacía se denotará con ϵ[12]. Al aplicar sustituciones sobre expresiones supondremos que podemos renombrar libremente la expresión para asegurar que las variables ligadas de e no aparecen en θ: $bv(e) \cap (dom(\theta) \cup vran(\theta)) = \emptyset$. Una *regla de programa* tiene la forma $f\ \overline{t_n} \rightarrow e$ donde $\overline{t_n}$ es lineal, pudiendo contener variables extra[13]. Por último, un *programa* $\mathcal{P}$ es un conjunto de reglas de programa.

La Figura 5 contiene las reglas de la relación de let-reescritura. Utilizaremos $\mathcal{P} \vdash e \rightarrow^l_{(regla)} e'$ para expresar que e se reduce a e' en un paso de let-reescritura bajo el programa $\mathcal{P}$ utilizando la regla $(regla)$. Normalmente omitiremos el programa cuando quede implícito por el contexto y la regla cuando esta no importe, reduciéndose a

[11] En algunos trabajos de esta tesis se utiliza la notación $Dom(\theta)$ y $vRan(\theta)$ para el dominio y las variables en el rango de sustituciones.

[12] En la mayoría de los trabajos de esta tesis se utiliza id para referirse a dicha sustitución vacía.

[13] Aunque tanto let-reescritura como let-estrechamiento pueden soportar las variables extra, en buena parte de los sistemas de tipos propuestos en esta tesis impediremos su aparición para garantizar la corrección del sistema de tipos. En la Sección 8, no obstante, abordaremos específicamente el problema de las variables extra en conexión con los sistemas de tipos.

$e \to^l e'$. Para referirnos a cero o más pasos de let-reescritura utilizaremos $\to^{l*}$.

La regla (Fapp) utiliza una regla de programa para reducir una aplicación de función. Nótese que esta regla requiere que los argumentos de la aplicación sean patrones, si no la semántica de *call-time choice* no se respetaría. Las reglas (LetIn), (Bind), (Elim), (Flat) y (LetAp) no realizan reducción en sí, sino que solamente cambian la representación de la expresión. (LetIn) mueve argumentos de funciones que no son patrones a ligaduras locales, permitiendo así aplicar reglas de función sobre argumentos no evaluados (no estrictez) y consiguiendo también que los argumentos sean compartidos (*call-time choice*). (Bind) propaga una ligadura cuando su lado derecho se ha reducido a un patrón. Al requerir que el lado derecho sea un patrón se respeta *call-time choice*, ya que diferentes apariciones de la variable X en e serán sustituidas por el mismo patrón t, que es irreducible. La regla (Elim) sirve para eliminar ligaduras innecesarias. (Flat) y (LetAp) gestionan las ligaduras, y son necesarias para evitar que algunas reducciones se queden incorrectamente bloqueadas. Finalmente, la regla (Contx) permite aplicar cualquiera de las reglas anteriores en alguna subexpresión. Las condiciones de (Contx) son necesarias para evitar la captura de variables al aplicar (Fapp) con reglas que tienen variables extra, es decir, para evitar que las variables extra que se introducen queden ligadas por el contexto. El siguiente ejemplo muestra algunas reducciones de let-reescritura, donde se observa cómo se respeta *call-time choice* y se consigue la no estrictez.

Ejemplo 8 (Reducciones de let-reescritura) *Consideremos los símbolos y reglas del Ejemplo 4 (página 16). Una posible reducción de let-reescritura para* $dup\ coin$ *sería:*

$$\begin{aligned}
&\underline{dup\ coin} \to^l_{(LetIn)} let\ X = coin\ in\ \underline{dup\ X} \\
&\to^l_{(Fapp)} let\ X = \underline{coin}\ in\ (X, X) \\
&\to^l_{(Fapp)} \underline{let\ X = zero\ in\ (X, X)} \\
&\to^l_{(Bind)} (zero, zero)
\end{aligned}$$

En la primera expresión no se puede aplicar (Fapp) porque $coin$ *es no es un patrón, así que se utiliza (LetIn) para sacarlo a una ligadura local y poder continuar. Al compartir el argumento* $coin$ *en la ligadura de* X *se consigue respetar* call-time choice*. Después se continúa evaluando* $dup\ X$*, que da lugar a la tupla* (X, X)*. Finalmente se evalúa* $coin$ *al patrón* $zero$ *y se propaga para conseguir* $(zero, zero)$*. Adviértase que esta no es la única derivación que da lugar a este patrón. Se podría haber reducido primero* $coin$ *a* $zero$ *y luego propagarlo con (Bind), obteniendo* $dup\ zero$*, que produce igualmente* $(zero, zero)$*.*

Las ligaduras también sirven para permitir funciones no estrictas. Un claro ejemplo de esta situación se observa en la siguiente reducción, que utiliza el programa del

Ejemplo 5 (página 17):

$$
\begin{array}{l}
f\ (\underline{succ\ loop}) \rightarrow^{l}_{(LetIn)} \underline{f\ (let\ X = loop\ in\ succ\ X)} \\
\rightarrow^{l}_{(LetIn)} let\ Y = (let\ X = loop\ in\ succ\ X)\ in\ \underline{f\ Y} \\
\rightarrow^{l}_{(Fapp)} \underline{let\ Y = (let\ X = loop\ in\ succ\ X)\ in\ true} \\
\rightarrow^{l}_{(Elim)} true
\end{array}
$$

Como se puede ver, el argumento $succ\ loop$ cuya evaluación no termina se extrae de la aplicación mediante ligaduras, obteniendo $f\ Y$. Con esa expresión ya puede aplicar la regla de f, obteniendo $true$. Finalmente, como la variable Y no aparece en la expresión $true$, la ligadura se puede eliminar.

La propiedad más importante de la let-reescritura es su equivalencia con CRWL, que establece que let-reescritura es una noción de paso de cómputo adecuada para los cómputos lógico-funcionales. Aunque los detalles particulares pueden encontrarse en [92, 131], esta equivalencia puede resumirse en el siguiente resultado:

Teorema 1 (Equivalencia entre let-reescritura y CRWL)

$$\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t \Longleftrightarrow e \rightarrow^{l*} t,\ \textit{para cualquier}\ e \in Exp, t \in Pat$$

La let-reescritura no realiza un tratamiento plenamente satisfactoria de las expresiones con variables libres o de las reglas de programa con variables extra. Ciertamente, las reglas de la let-reescritura permiten que las expresiones a reducir contengan variables libres; sin embargo, ninguna de las reglas las liga a patrones, así que las variables libres de la expresión a reducir juegan un papel pasivo, y podrían asimilarse a nuevas constantes. Por otra parte, la let-reescritura también permite variables extra en las reglas. Estas variables extra sí que pueden ser ligadas a patrones pero esta ligadura no se realiza de acuerdo a las necesidades del cómputo, sino que deben ser «adivinadas mágicamente» por la sustitución utilizada al aplicar la regla usando (Fapp):

Ejemplo 9 (Variables extra en let-reescritura) *Consideremos el siguiente programa tomado de [131], que comprueba si un número es par:*

```
add :: nat -> nat -> nat          eqNat :: nat -> nat -> bool
add zero Y = Y                    eqNat zero zero = true
add (succ X) Y = succ (add X Y)   eqNat (succ X) (succ Y) = eqNat X Y

if_then :: bool -> A -> A        even :: nat -> bool
if_then true X = X               even X = if eqNat (add Y Y) X then true
```

(Por claridad, hemos tomado la licencia sintáctica de colocar el argumento de la función if_then entre las palabras if y $then$). Como se puede observar, la regla $even$ contiene la

(X) $e \rightsquigarrow^l_\epsilon e'$, si $e \rightarrow^l e'$ usando $X \in \{Elim, Bind, Flat, LetIn, LetAp\}$

(Narr) $f\ \overline{t_n} \rightsquigarrow^l_\theta r\theta$, para alguna variante fresca $(f\ \overline{p_n} \rightarrow r) \in \mathcal{P}$ y θ tal que $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$.

(VAct) $X\ \overline{t_k} \rightsquigarrow^l_\theta r\theta$, si $k > 0$, para alguna variante fresca $(f\ \overline{p} \rightarrow r) \in \mathcal{P}$ y θ tal que $(X\ \overline{t_k})\theta \equiv f\ \overline{p}\theta$

(VBind) $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\theta e_2\theta[X \mapsto e_1\theta]$, si $e_1 \notin Pat$, para alguna θ que hace que $e_1\theta \in Pat$, siempre que $X \notin (dom(\theta) \cap vran(\theta))$

(Contx) $\mathcal{C}[e] \rightsquigarrow^l_\theta \mathcal{C}\theta[e']$, para $\mathcal{C} \neq [\,]$, $e \rightsquigarrow^l_\theta e'$ usando alguna de las reglas anteriores, y:
i) $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$
ii) • si el paso es (Narr) o (VAct) usando $(f\ \overline{p_n} \rightarrow r) \in \mathcal{P}$ entonces $vran(\theta|_{\setminus var(\overline{p_n})}) \cap bv(\mathcal{C}) = \emptyset$
• si el paso es (VBind) entonces $vran(\theta) \cap bv(\mathcal{C}) = \emptyset$.

Figura 6: Relación $\rightsquigarrow^l$ de let-estrechamiento

variable extra Y. Una reducción de $even\ zero$ utilizando let-reescritura sería:

$$\begin{array}{l} \underline{even\ zero} \rightarrow^l_{(Fapp)} if\ eqNat\ \underline{(add\ zero\ zero)}\ zero\ then\ true \\ \rightarrow^l_{(Fapp)} if\ \underline{eqNat\ zero\ zero}\ then\ true \\ \rightarrow^l_{(Fapp)} \underline{if\ true\ then\ true} \\ \rightarrow^l_{(Fapp)} true \end{array}$$

En el primer paso de la reducción se ha utilizado la variante fresca de la regla

$$even\ X_1 \rightarrow if\ eqNat\ (add\ Y_1\ Y_1)\ X_1\ then\ true$$

y la sustitución $\theta \equiv [X_1 \mapsto zero, Y_1 \mapsto zero]$. Durante la aplicación de la función se debe dar valores a las variables extra, en este caso $zero$ para Y_1. Estas ligaduras son completamente especulativas, ya que no se sabe qué valores serán necesarios para esas variables según el cómputo avance. En este caso la ligadura $Y_1 \mapsto zero$ es la única válida, ya que cualquier otra resultaría en que la reducción se quedaría bloqueada al no tener reglas de $eqNat$ que encajen.

Para resolver este comportamiento poco natural de las variables extra y permitir un manejo de las variables libres en las expresiones surge la semántica de let-estrechamiento, como una elevación de la let-reescritura. Las reglas de la Figura 6 permiten realizar pasos de let-estrechamiento $e \rightsquigarrow^l_\theta e'$, que significa que e es estrechado a e' produciendo la sustitución $\theta \in PSubst$. La regla (X) colecciona los pasos de let-reescritura que corresponden a pasos de let-estrechamiento con la sustitución vacía. La regla (Narr) representa el paso de estrechamiento para aplicaciones de función. Nótese que la sustitución θ utilizada puede ser cualquier unificador, así que no está

restringido a *unificadores más generales* (*mgu's*). Las reglas (VAct) y (VBind) producen ligaduras de orden superior para expresiones o subexpresiones flexibles. Por último, la regla (Contx) permite aplicar pasos de let-estrechamiento en subexpresiones, evitando que la sustitución afecte a variables ligadas en el contexto (condición i) y la captura de variables ligadas por alguna let-expresión (condición ii).

Ejemplo 10 (Reducciones de let-estrechamiento) *Consideremos el programa aparecido en el Ejemplo 9 (página 26). Los siguientes pasos de let-estrechamiento representarían una posible reducción de* $even\ zero$*. En cada paso mostramos la sustitución producida restringida a las variables libres de la expresión:*

$$
\begin{array}{ll}
\underline{even\ zero} & \\
\leadsto^{l}_{\epsilon} \ if\ eqNat\ \underline{(add\ Y_1\ Y_1)}\ zero\ then\ true & \textit{(Narr)} \\
\leadsto^{l}_{[Y_1 \mapsto zero]} \ if\ \underline{eqNat\ zero\ zero}\ then\ true & \textit{(Narr)} \\
\leadsto^{l}_{\epsilon} \ \underline{if\ true\ then\ true} & \textit{(Narr)} \\
\leadsto^{l}_{\epsilon} \ true & \textit{(Narr)}
\end{array}
$$

En esta reducción, en el primer paso no se realiza ninguna ligadura «adivinatoria» sobre la variable extra Y_1 *al aplicar la regla de* $even$ *(aunque sería válido, ya que la regla (Narr) permite sustituciones arbitrarias) sino que dicha variable se introduce en la expresión resultante como una variable libre. Es en el siguiente paso cuando se realiza una ligadura de esta variable* Y_1 *a* $zero$ *para poder aplicar la primera regla de* add*. A partir de ese punto, la reducción avanza de manera similar a la let-reescritura.*

Otro ejemplo de reducción donde se generan ligaduras de orden superior sería la evaluación de $[F\ zero,\ let\ X = G\ zero\ in\ X]$*:*

$$
\begin{array}{ll}
[\underline{F\ zero},\ let\ X = G\ zero\ in\ X] & \\
\leadsto^{l}_{[F \mapsto add\ zero]} [zero,\ \underline{let\ X = G\ zero\ in\ X}] & \textit{(VAct)} \\
\leadsto^{l}_{[G \mapsto succ]} [zero, succ\ zero] & \textit{(VBind)}
\end{array}
$$

En este caso la sustitución generada durante la reducción (restringida a las variables libres de la expresión original) sería $\theta \equiv [F \mapsto add\ zero, G \mapsto succ]$*. En el primer paso se utiliza la regla (Vact) para estrechar la aplicación de variable* $F\ zero$*, generando la ligadura de orden superior* $F \mapsto add\ zero$*, que permite aplicar la primera regla de* add*. En el segundo paso se utiliza la regla (VBind) para conseguir, mediante la ligadura de orden superior* $G \mapsto succ$*, que la expresión* $G\ zero$ *—que no era un patrón— se convierta en el patrón* $succ\ zero$*, a la vez que elimina la let-expresión propagando su ligadura.*

El let-estrechamiento, al ser una elevación de la let-reescritura, posee las clásicas propiedades de corrección y completitud con respecto a esta última. Debido a la complejidad técnica de estos resultados hemos omitido su presentación, pero el lector interesado puede encontrar todos los detalles en [92, 131].

Para finalizar esta sección conviene insistir en que tanto la let-reescritura como el let-estrechamiento no pueden ser considerados como métodos *efectivos* de cómputo ya que carecen de una *estrategia* de elección de reglas y expresiones a reducir. Estas semánticas solamente establecen qué pasos son válidos, pero no indican qué expresiones son las mejores para reducir en cada caso, ni que regla aplicar si hay varias. En [133] se da unos primeros pasos para la definición de estrategias en let-reescritura, no obstante, es interesante incidir en que en algunas ocasiones tener en cuenta estrategias particulares da lugar a resultados menos generales. Un ejemplo de esto es la propiedad de *preservación de tipos*. Si se demuestra sin tener en cuenta ninguna estrategia, los tipos serán preservados para cualquier paso de cómputo, incluso para aquellos que no son necesarios o adecuados. Puesto que una estrategia restringirá las reglas o la expresión a reducir, el anterior resultado será válido para cualquier estrategia considerada. En esta tesis no nos hemos ceñido a ninguna estrategia de evaluación particular para la let-reescritura o el let-estrechamiento, por lo que nuestros resultados se benefician de la mencionada generalidad.

4.3. Otras semánticas para programación lógico-funcional

Aparte de las semánticas CRWL y let-reescritura/let-estrechamiento, que son las más influyentes en el marco de PLF considerado en esta tesis, existen otras semánticas operacionales de relevancia en PLF. Una de las más importantes son los *sistemas de reescritura de términos* [13, 145] (TRS según sus siglas en inglés). Este formalismo, ampliamente conocido y utilizado como semántica de lenguajes funcionales, no es completamente adecuado para PLF debido a que modeliza *run-time choice* en lugar de *call-time choice* como opción para el paso de parámetros. No obstante, esta semántica ha sido ampliamente utilizada en el ámbito lógico-funcional, en particular como base para el desarrollo de estrategias de estrechamiento con diversas propiedades de optimalidad [3, 6, 38, 39, 82]. Los *sistemas de reescritura de grafos* [15] pueden considerarse una generalización de los sistemas de reescritura de términos, donde las expresiones son representadas mediante grafos, permitiendo la compartición de subexpresiones comunes. Estos sistemas han sido utilizados principalmente como base para la implementación eficiente de lenguajes funcionales [21, 123, 121, 114], aunque también han sido aplicados como semántica operacional para PLF [36, 37]. Por último, en [1, 20, 19] se propone una familia de semánticas operacionales relevantes en el ámbito lógico-funcional. Estas semánticas —que siguen las ideas de la semántica operacional perezosa para PF de [80]— utilizan un *montón* (*heap*) para almacenar ligaduras de variables a expresiones, de manera similar a las let-expresiones en let-reescritura y let-estrechamiento. Sin embargo, son semánticas consideradas de más bajo nivel ya que requieren una transformación previa del programa, transformación que utiliza distintos análisis de demanda (como los relativos a las estrategias [6, 38]), con lo cual la propia estrategia queda codificada en el programa.

5. Sistemas de tipos en programación funcional y lógico-funcional

Los sistemas de tipos [24, 120], en su visión más aplicada, son formalismos que permiten el análisis de programas, con el fin primordial de garantizar que ciertos tipos de errores no aparecerán durante la ejecución de los programas. Para ello, clasifican las distintas construcciones del programa según la clase de valores que representan (su *tipo*), e impidiendo su uso en lugares incompatibles con esos valores. Este análisis y detección de problemas puede llevarse a cabo completamente durante la compilación del programa, dando lugar a los llamados *sistemas de tipos estáticos* y generando errores de compilación, o puede requerir ciertas comprobaciones en tiempo de ejecución, dando lugar a los *sistemas de tipos dinámicos* que generan excepciones al detectar los problemas de tipos. Además de la seguridad que proporcionan, evitando la aparición de errores durante la ejecución, los sistemas de tipos también aportan otros beneficios a los lenguajes de programación. Normalmente, los sistemas de tipos también imponen una cierta *disciplina* a los programas, exigiendo que las distintas partes del mismo aparezcan en un determinado orden (declaración de identificadores antes de su uso, por ejemplo), consiguiendo así un aspecto homogéneo de los programas y facilitando su lectura. También suelen soportar (y en algunos casos requerir) declaraciones de tipos sobre distintos elementos del programa, como las funciones. Estas declaraciones de tipos sirven como *documentación* de los programas, ya que suelen aclarar el significado de los distintos elementos. A diferencia de los comentarios incrustados en el código fuente, estas declaraciones de tipos son comprobadas en cada compilación, con lo que siempre están actualizadas. Aparte de esta función de documentación, las declaraciones de tipos pueden ayudar al programador en la detección temprana de errores. Durante la fase de la implementación, la declaración de tipos de una función puede entenderse como una aproximación del comportamiento esperado de la misma. Declarando el tipo de una función de manera previa a su implementación, el programador puede comprobar durante la compilación que no existen diferencias de tipos entre el código producido y el comportamiento esperado. Por último, los sistemas de tipos también pueden influir positivamente en la *eficiencia* del código producido. Debido a que clasifican las distintas construcciones con respecto al conjunto de valores que producen, pueden aportar información que permita generar código eficiente particular del tipo concreto de la construcción. Un ejemplo de esto es el lenguaje Fortran, que mejoraba la eficiencia de los cálculos numéricos distinguiendo entre expresiones aritméticas enteras y en coma flotante. Otros ejemplos son la propuesta de [147], donde un sistema de tipos con información de regiones de memoria es usado para reducir la recolección de basura durante la ejecución —mejorando así la eficiencia global del programa— o *HiPE* [11] (*High-Performance Erlang*), que genera código específico en lugar de código genérico para aquellas funciones cuya información de tipo es suficientemente concreta.

Los sistemas de tipos tienen una larga historia desde su aparición en la década de los 50 (ver [25][§1.3] para más información sobre la evolución de los sistemas de tipos en los lenguajes de programación). Sin embargo, ha sido en el campo de la programación funcional donde han tenido una mayor repercusión y desarrollo. Dentro de este paradigma es particularmente importante el sistema de tipos de Damas-Milner [103, 32], originalmente propuesto para el lenguaje ML [104]. Este sistema de tipos destaca por soportar *polimorfismo paramétrico* [142, 25] (una misma función puede ser aplicada a diversos tipos de manera uniforme), poseer *tipos principales* (toda expresión tiene un tipo que es más general que cualquier otro tipo derivable) y proporcionar un *algoritmo de inferencia* de tipos eficiente que permite comprobar e inferir los tipos de las distintas construcciones del programa. Basándose en este sistema de tipos, en el campo de la programación funcional se han propuesto una gran variedad de extensiones como la recursión polimórfica [108, 76, 57], los tipos existenciales [105, 112, 79], las clases de tipos [48, 12], el polimorfismo de rango arbitrario [111, 118], los tipos de datos algebraicos generalizados (GADTs) [28, 119, 134] o la programación genérica [60, 62, 64].

Debido a la gran importancia del sistema de tipos de Damas-Milner en programación funcional, las distintas aproximaciones a la programación lógico-funcional han adaptado de manera directa este sistema de tipos. Como se ha visto en la Sección 1, la adaptación directa de Damas-Milner no maneja adecuadamente la ligadura de variables libres de orden superior mediante estrechamiento ni los patrones de orden superior característicos de CRWL, con lo que la garantía de seguridad durante los cómputos se pierde. Como única excepción a esta situación, han surgido dos propuestas teóricas de sistemas de tipos para programación lógico-funcional: [45] basado en CRWL y CLNC, y [9] basado en estrechamiento sobre TRS; aparte de propuestas preliminares sobre la adaptación de clases de tipos a PLF [107, 95].

En esta sección introduciremos la situación actual de los sistemas de tipos en programación lógico-funcional. Primero presentaremos el sistema de tipos de Damas-Milner, base de los sistemas de tipos actuales en PF/PLF y de los sistemas de tipos que proponemos en esta tesis. Luego presentaremos los dos citados trabajos sobre sistemas de tipos en PLF, mostrando algunas de sus limitaciones. Por último, presentaremos con detalle algunas de las extensiones de sistemas de tipos para PF, debido a que sus ideas están relacionadas e influencian algunos de los sistemas de tipos que proponemos en esta tesis.

5.1. Sistema de tipos de Damas-Milner

El sistema de tipos de Damas-Milner fue presentado por primera vez en el artículo de Milner [103], siendo completado en los artículos de Milner y Damas [32, 31] con una formalización más sencilla y con la demostración de la completitud del algoritmo $\mathcal{W}$ de inferencia de tipos. Anteriormente y de manera independiente a [103], Hindley presen-

tó un trabajo para derivar *tipos principales* para términos de la lógica combinatoria [58] que utilizaba el algoritmo de unificación de Robinson [128] de manera similar al algoritmo $\mathcal{W}$. Es por ello que este sistema de tipos también es conocido como Hindley-Milner, aunque en esta tesis siempre será llamado como sistema de tipos de Damas-Milner o DM.

El sistema de tipos de DM considera un lenguaje funcional no explícitamente tipado con expresiones e definidas como $e ::= x \mid e\ e' \mid \lambda x.e \mid let\ x = e\ in\ e'$, donde x es un identificador. Para la sintaxis de los tipos consideramos un conjunto infinito numerable de *variables de tipo* $\alpha \in \mathcal{TV}$ y de *constructoras de tipos* $C \in \mathcal{TC}$, cada constructora de tipo C con una aridad asociada[14]. Los *tipos simples* τ se definen como $\tau ::= \alpha \mid C\ \tau_1 \ldots \tau_n$ (si $C \in \mathcal{TC}^n$) $\mid \tau \to \tau$, y los *esquemas de tipo* (*type-schemes*) σ se definen como $\sigma ::= \tau \mid \forall\alpha_1 \ldots \forall\alpha_n.\tau$. Normalmente los esquemas de tipo se simplificarán como $\forall\alpha_1 \ldots \alpha_n.\tau$ o $\forall\overline{\alpha_n}.\tau$. El conjunto de *variables de tipo libres* (ftv) de un tipo simple se define como $ftv(\tau) = var(\tau)$, y sobre esquemas de tipo como $ftv(\forall\overline{\alpha_n}.\tau) = ftv(\tau) \setminus \{\overline{\alpha_n}\}$. Una *sustitución de tipos* π es una aplicación finita de variables de tipos a tipos simples $\pi \equiv [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$, que se extiende de manera natural a tipos simples y esquemas de tipos (nótese que aplicar una sustitución π a un esquema de tipos σ solo afecta a sus variables libres). Un tipo σ' es una *instancia* de σ si $\sigma' = \sigma\pi$. Por otro lado, τ' es una *instancia genérica* de $\sigma \equiv \forall\overline{\alpha_n}.\tau$ (escrito $\sigma \succ \tau'$) si $\tau' = \tau[\overline{\alpha_n \mapsto \tau_n}]$ para algunos $\overline{\tau_n}$. Extendemos $\succ$ a una relación entre esquemas de tipo definiendo $\sigma \succ \sigma'$ si y solo si cada tipo simple que es instancia genérica de σ' lo es también de σ (alternativamente $\forall\overline{\alpha_n}.\tau \succ \forall\overline{\beta_m}.\tau[\overline{\alpha_n \mapsto \tau_n}]$ si y solo si $\{\overline{\beta_m}\} \cap ftv(\forall\overline{\alpha_n}.\tau) = \emptyset$ [31]). También decimos que τ' es una *variante* de $\sigma \equiv \forall\overline{\alpha_n}.\tau$ ($\sigma \succ_{var} \tau'$) si $\tau' = \tau[\overline{\alpha_n \mapsto \beta_n}]$ y $\overline{\beta_n}$ son variables frescas.

Un *conjunto de suposiciones de tipos* $\mathcal{A}$ es un conjunto de asociaciones *(identificador : esquema de tipos)* de la forma $\{\overline{x_n : \sigma_n}\}$, donde cada identificador es único[15]. La notación para acceder a la suposición asociada a un identificador x es $\mathcal{A}(x)$, verificando que $\mathcal{A}(x) = \sigma$ si $(x : \sigma) \in \mathcal{A}$. Las variables de tipo libres de un conjunto de suposiciones están definidas como $ftv(\{\overline{x_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. Añadir la suposición $(x : \sigma)$ a $\mathcal{A}$ se define como $\mathcal{A} \oplus \{x : \sigma\} = \mathcal{A}_x \cup \{x : \sigma\}$, donde $\mathcal{A}_x$ es el resultado de eliminar de $\mathcal{A}$ la posible aparición de una suposición sobre el identificador x. Esta notación se extiende a conjuntos de suposiciones como $\mathcal{A} \oplus \{\overline{x_n : \sigma_n}\} = \mathcal{A} \oplus \{x_1 : \sigma_1\} \oplus \ldots \{x_n : \sigma_n\}$. La aplicación de una sustitución de tipos sobre un conjunto de suposiciones $\mathcal{A}\pi$ se define como $\{\overline{x_n : \sigma_n}\}\pi = \{\overline{x_n : \sigma_n\pi}\}$. Por último, la *generalización de* τ *con respecto a* $\mathcal{A}$ se define como $Gen(\tau, \mathcal{A}) = \forall\overline{\alpha_n}.\tau$, donde $\{\alpha_n\} = ftv(\tau) \setminus ftv(\mathcal{A})$.

El sistema de tipos original de Damas-Milner [103, 32, 31] presenta una relación

[14] Aquí nos desviaremos ligeramente de la sintaxis de los tipos presentada en [103, 32, 31] utilizando constructoras de tipos C en lugar de los *tipos primitivos* ι para los booleanos, enteros, etc...

[15] Aunque esta es la definición original para DM, en los sistemas de tipos propuestos en esta tesis (Secciones 6–8) usaremos una definición ligeramente diferente de conjunto de suposiciones, asociando tipos a símbolos: $\mathcal{A} \equiv \{\overline{s_n : \sigma_n}\}$.

TAUT: $\dfrac{}{\mathcal{A} \vdash x : \sigma}$ si $\mathcal{A}(x) = \sigma$

INST: $\dfrac{\mathcal{A} \vdash e : \sigma}{\mathcal{A} \vdash e : \sigma'}$ si $\sigma \succ \sigma'$

GEN: $\dfrac{\mathcal{A} \vdash e : \sigma}{\mathcal{A} \vdash e : \forall\alpha.\sigma}$ si $\alpha \notin ftv(\mathcal{A})$

APP: $\dfrac{\mathcal{A} \vdash e : \tau' \to \tau \quad \mathcal{A} \vdash e' : \tau'}{\mathcal{A} \vdash e\ e' : \tau}$

ABS: $\dfrac{\mathcal{A} \oplus \{x : \tau'\} \vdash e : \tau}{\mathcal{A} \vdash (\lambda x.e) : \tau' \to \tau}$

LET: $\dfrac{\mathcal{A} \vdash e : \sigma \quad \mathcal{A} \oplus \{x : \sigma\} \vdash e' : \tau}{\mathcal{A} \vdash let\ x = e\ in\ e' : \tau}$

a) Versión original de DM

TAUT': $\dfrac{}{\mathcal{A} \vdash x : \tau}$ si $\mathcal{A}(x) = \sigma$ y $\sigma \succ \tau$

APP: $\dfrac{\mathcal{A} \vdash e : \tau' \to \tau \quad \mathcal{A} \vdash e' : \tau'}{\mathcal{A} \vdash e\ e' : \tau}$

ABS: $\dfrac{\mathcal{A} \oplus \{x : \tau'\} \vdash e : \tau}{\mathcal{A} \vdash (\lambda x.e) : \tau' \to \tau}$

LET': $\dfrac{\mathcal{A} \vdash e : \tau \quad \mathcal{A} \oplus \{x : \sigma\} \vdash e' : \tau}{\mathcal{A} \vdash let\ x = e\ in\ e' : \tau}$
si $\sigma = Gen(\tau, \mathcal{A})$

b) DM' dirigido por la sintaxis

Figura 7: Sistema de tipos de Damas-Milner

DM de tipado $\mathcal{A} \vdash e : \sigma$ con las reglas que aparecen en la Figura 7-a). Como se puede ver, esta relación no está dirigida por la sintaxis ya que las reglas INST y GEN pueden aplicarse en cualquier expresión. Para evitar esa situación, es posible modificar ligeramente las reglas combinando TAUT e INST en TAUT', y LET y GEN en LET', consiguiendo la relación DM' de tipado $\mathcal{A} \vdash e : \tau$ de la Figura 7-b) [30]. Aunque esta relación solo deriva tipos simples para las expresiones, ambos sistemas de tipos son equivalentes en el siguiente sentido:

Teorema 2 (Equivalencia de DM y DM', [30](Th. 2.1))

- $\forall \mathcal{A}, e, \tau \quad \mathcal{A} \vdash_{DM'} e : \tau \Longrightarrow \mathcal{A} \vdash_{DM} e : \tau$.
- $\forall \mathcal{A}, e\ \exists \tau \quad \mathcal{A} \vdash_{DM} e : \sigma \Longrightarrow \mathcal{A} \vdash_{DM'} e : \tau$ y $Gen(\tau, \mathcal{A}) \succ \sigma$.

Debido a este resultado, se suelen usar de manera intercambiable DM y DM', refiriéndose a ambos con el nombre de Damas-Milner. Para los sistemas de tipos propuestos en esta tesis nos hemos basado en DM' ya que es más sencillo y está dirigido por la sintaxis de la expresión.

Aparte del sistema de tipos, en [32, 31] se propone un algoritmo $\mathcal{W}$ de inferencia de tipos para expresiones. Dicho algoritmo se encuentra en la Figura 8, utilizando una presentación en pseudocódigo funcional similar a la de [32]. El algoritmo $\mathcal{W}$ toma como argumentos un conjunto de suposiciones y una expresión, y devuelve una pareja de sustitución de tipos y tipo simple: $\mathcal{W}(\mathcal{A}, e) = (\pi, \tau)$. De una manera intuitiva el tipo τ es el tipo más general que se puede derivar para e, mientras que π es la mínima sustitución que hace falta aplicar a $\mathcal{A}$ para poder derivar algún tipo para e (ver el Teorema

ALGORITMO $\mathcal{W}$

i) $\mathcal{W}(\mathcal{A}, x) = (\epsilon, \tau)$, si $\mathcal{A}(x) \succ_{var} \tau$.

ii) $\mathcal{W}(\mathcal{A}, e_1 e_2) =$ sea $(\pi_1, \tau_1) = \mathcal{W}(\mathcal{A}, e_1)$
y sea $(\pi_2, \tau_2) = \mathcal{W}(\mathcal{A}\pi_1, e_2)$
y sea $\pi_u = mgu(\tau_1\pi_2, \tau_2 \to \beta)$ donde β es fresca
en $(\pi_1\pi_2\pi_u, \beta\pi_u)$.

iii) $\mathcal{W}(\mathcal{A}, \lambda x.e_1) =$ sea $(\pi_1, \tau_1) = \mathcal{W}(\mathcal{A} \oplus \{x : \beta\}, e_1)$ donde β es fresca
en $(\pi_1, \beta\pi_1 \to \tau_1)$.

iv) $\mathcal{W}(\mathcal{A}, let\ x = e_1\ in\ e_2) =$ sea $(\pi_1, \tau_1) = \mathcal{W}(\mathcal{A}, e_1)$
y sea $(\pi_2, \tau_2) = \mathcal{W}(\mathcal{A}\pi_1 \oplus \{x : Gen(\tau_1, \mathcal{A}\pi_1)\}, e_2)$
en $(\pi_1\pi_2, \tau_2)$.

Nota: Cuando alguna de las condiciones anteriores no se cumple, $\mathcal{W}$ falla.

Figura 8: Algoritmo $\mathcal{W}$

3 más adelante). La idea de este algoritmo es introducir variables frescas en lugar de «inventarse» tipos, y posteriormente utilizar un unificador más general (*mgu*) [128, 98] para unificar aquellos tipos que deban ser iguales.

La propiedad más importante con respecto a este algoritmo es su adecuación con respecto a DM:

Teorema 3 (Adecuación de $\mathcal{W}$ con respecto a $\vdash_{\mathsf{DM}}$, [31](Th. 2 y 3))

- (Corrección) *Si* $\mathcal{W}(\mathcal{A}, e) = (\pi, \tau)$ *entonces* $\mathcal{A}\pi \vdash_{DM} e : \tau$.
- (Completitud) *Si* $\mathcal{A}\pi \vdash_{DM} e : \sigma$ *entonces:*
  - *i)* $\mathcal{W}(\mathcal{A}, e)$ *tiene éxito.*
  - *ii) Si* $\mathcal{W}(\mathcal{A}, e) = (\pi', e)$ *entonces, para alguna sustitución* π'', $\mathcal{A}\pi = \mathcal{A}\pi'\pi''$ *y* $Gen(\tau, \mathcal{A}\pi')\pi'' \succ \sigma$.

Como corolario de la completitud se obtiene que si $\mathcal{A} \vdash_{\mathsf{DM}} e : \sigma$ entonces existe un *tipo principal* de e bajo $\mathcal{A}$ [31](Cor. 1).

En el marco funcional en el que se basa DM no es necesario disponer de una noción de programa bien tipado, ya que los programas son en realidad expresiones: cadena de let-expresiones definiendo las funciones mediante λ-abstracciones acompañados de la expresión a evaluar. Por lo tanto, todos los resultados sobre expresiones se extienden de manera trivial a programas.

Para terminar esta sección comentaremos la corrección de Damas-Milner (versión DM'), resumida en el famoso eslogan *well-typed programs cannot go wrong*[16]. Este re-

[16]Para ello usaremos un estilo y notación más cercano a [154](§2), ya que su presentación es más clara que la original en [103, 31].

sultado considera como dominio semántico un *cpo* (*complete partial order*) con conjunto soporte definido por la ecuación recursiva de dominios:

$$V \simeq B_0 + \ldots + B_n + (V \to V) + W$$

donde $B_0 \ldots B_n$ son dominios básicos como booleanos y enteros, $+$ es la unión disjunta, $\to$ construye funciones continuas y W es el dominio conteniendo el elemento único $wrong$. La función semántica $\mathscr{E}[\![e]\!]\rho$ asigna un valor semántico en V a la expresión e, considerando un entorno ρ que da valores semánticos a las variables libres. Lo importante de esta función $\mathscr{E}$ es que devuelve $wrong$ en las ocasiones problemáticas, como cuando en una aplicación $e_1\ e_2$ la expresión e_1 no es una función. Para relacionar los tipos τ y los valores semánticos v se introduce la relación semántica $\vDash$ definida como

$$\vDash v : \tau \text{ si y solo si } v \in V^\tau$$

donde V^τ es un subconjunto (concretamente un *ideal* [97]) del dominio V que corresponden al fragmento de V representado por el tipo τ. También se dice que un entorno ρ satisface un conjunto de suposiciones $\mathcal{A}$ (escrito $\vDash \rho : \mathcal{A}$) si para toda variable x se cumple que $\vDash \rho(x) : \mathcal{A}(x)$. Basándose en estos conceptos se demuestra el siguiente resultado:

Teorema 4 (Corrección de Damas-Milner, [154](§2.1)) *Si* $\mathcal{A} \vdash_{DM'} e : \tau$ *y* $\vDash \rho : \mathcal{A}$ *entonces* $\vDash \mathscr{E}[\![e]\!]\rho : \tau$

Este teorema significa que si una expresión e tiene tipo τ, entonces su valor semántico $\mathscr{E}[\![e]\!]\rho$ estará en el conjunto V^τ de valores semánticos que representa el tipo τ, siempre que el entorno satisfaga el conjunto de suposiciones. Del anterior teorema se desprende de manera trivial el eslogan *well-typed programs cannot go wrong*, ya que $\mathscr{E}[\![e]\!]\rho \neq wrong$ debido a que $wrong$ no tiene ningún tipo, es decir, no existe τ tal que $\vDash wrong : \tau$.

5.2. Propuestas de sistemas de tipos para PLF

En la práctica, los sistemas de PLF como Toy o Curry adaptan de manera directa el sistema de tipos DM, adaptación que, como hemos visto, produce problemas de tipos. Sin embargo, sí que existen algunos trabajos teóricos en los que se proponen sistemas de tipos específicos para PLF. Entre ellos los que más importancia tienen en esta tesis son [45, 47][17] debido a que utilizan HO-CRWL como semántica, y por tanto deben abordar los problemas de tipos asociados a los patrones de orden superior.

En [45] solo se consideran tipos simples τ, llamando *tipos de datos* a aquellos en los que no aparece la flecha $\to$ del tipo funcional. Todas las constructoras $c \in CS^n$

[17] Normalmente citaremos solo [45], ya que es una versión extendida y revisada de [47]

vienen acompañadas de una declaración de tipos $c :: \overline{\tau_n} \to C\ \overline{\alpha_k}$, donde $n, k \geq 0$ y $\overline{\alpha_k}$ son variables distintas. Además, este sistema impone dos restricciones más a los tipos de las constructoras. La primera es que $\overline{\tau_n}$ deben ser tipos de datos, es decir, no se admiten constructoras que contengan tipos funcionales. La segunda restricción, llamada *propiedad de transparencia*, obliga a que $ftv(\overline{\tau_n}) \subseteq \{\overline{\alpha_k}\}$. Estas dos restricciones impiden la utilización de *constructoras existenciales* (ver la Sección 5.3.1 para más detalles) como $CS^2 \ni key :: \alpha \to (\alpha \to nat) \to key$ o $CS^1 \ni cont :: \alpha \to container$. La constructora especial $\bot$ viene con la declaración de tipos $\bot :: \alpha$, representando que el valor indefinido puede tener cualquier tipo. Por su parte, todas las funciones $f \in FS^n$ vienen acompañadas de una declaración de tipo $f :: \overline{\tau_n} \to \tau$, siendo $\overline{\tau_n}, \tau$ tipos simples arbitrarios. Continuando con la noción subyacente a la propiedad de transparencia de las constructoras, diremos que un tipo que se puede escribir como $\overline{\tau_m} \to \tau$ es *m-transparente* si $ftv(\overline{\tau_m}) \subseteq ftv(\tau)$, y un símbolo h es m-transparente si $(h :: \tau) \in \Sigma$ y τ es m-transparente. Las constructoras $c \in CS^n$ son siempre m-transparentes para todo $m \leq n$ debido a la propiedad de transparencia. Otra consecuencia importante de esta propiedad es que garantiza que los tipos de las variables de un patrón t construido *solo* por constructoras pueden ser deducidos directamente a partir del tipo de t. Esta característica, importante para la preservación de tipos, se tiene también para los llamados *patrones transparentes*:

$$t ::= X \mid \bot \mid c\ \overline{t_m}\ (c \in CS^n, m \leq n) \mid f\ \overline{t_m}\ (f \in FS^n, m < n)$$

donde los patrones $\overline{t_m}$ en $c\ \overline{t_m}$ y $f\ \overline{t_m}$ son patrones transparentes y f es m-transparente. Los patrones o tipos que no son transparentes se llamarán *opacos*. Un ejemplo representativo de patrón opaco es $snd\ X$, que se utiliza en el Ejemplo 1 (página 4) para crear el *casting* polimórfico. Este patrón es opaco ya que sabiendo que $snd\ X$ tiene un tipo concreto, por ejemplo $bool \to bool$, no puede deducirse el tipo que debe tener la variable X.

Como los símbolos $h \in \Sigma$ vienen con sus declaraciones de tipos, los *entornos de tipos* T solo han de contener suposiciones $X :: \tau$ para las variables de datos. Basándose en estos elementos, los juicios de tipos para expresiones (las mismas consideradas por CRWL) $T \vdash_{WT} e :: \tau$ se construyen con las siguientes tres reglas, basadas directamente en DM:

VR $T \vdash_{WT} X :: \tau$, si $T(X) = \tau$
ID $T \vdash_{WT} h :: \tau\pi$, si $(h :: \tau) \in \Sigma$ y π es cualquier sustitución de tipos
AP $T \vdash_{WT} (e\ e_1) :: \tau$, si $T \vdash_{WT} e :: (\tau_1 \to \tau)$ y $T \vdash_{WT} e_1 :: \tau_1$

Se puede observar que **ID** refleja la cuantificación implícita sobre las variables de tipos de las constructoras y funciones[18], mientras que **VR** deja fijados los tipos de las

[18] Esta regla permite el polimorfismo de los símbolos $h \in \Sigma$ sin la necesidad de utilizar esquemas de tipos, ya que la declaración de tipos $(h :: \tau) \in \Sigma$ puede entenderse como $(h :: \forall\overline{\alpha_n}.\tau)$,

variables de datos. Usando la relación de tipado $\vdash_{WT}$, en [45] se definen las *reglas bien tipadas* como reglas de la forma[19]:

$$f\ t_1 \ldots t_n \to r \,\square\, T$$

donde $(f :: \overline{\tau_n} \to \tau) \in \Sigma$ (posiblemente renombrando las variables), $\bot$ no aparece en la regla, $f\ t_1 \ldots t_n$ es lineal y T es un entorno de tipos cuyo dominio son las variables de datos de la regla, cumpliendo que:

i) Los patrones t_i son *transparentes*.

ii) No hay *variables extra*, es decir, $fv(f\ t_1 \ldots t_n) \subseteq fv(r)$.

iii) $T \vdash_{WT} t_i :: \tau_i$ para $1 \leq i \leq n$, y $T \vdash_{WT} r :: \tau$.

Un *programa bien tipado* es un conjunto de reglas bien tipadas. La condición i) excluye *por construcción* reglas con patrones opacos, aunque como veremos en la Sección 6 estos patrones no siempre generan problemas de tipos. La condición ii) excluye variables extra, que son una característica muy expresiva de la PLF. Por último, la condición iii) garantiza la *generalidad de tipo* de las reglas, cuyo tipo debe corresponder *exactamente* con el tipo declarado de la función. Este sistema requiere tanto la declaración de tipo de todas las funciones como entornos de tipos adecuados en cada regla para asignar tipos a las variables, sin proporcionar métodos efectivos para inferirlos.

Basándose en la noción de programa bien tipado, en [45] se proporciona un resultado de *preservación de tipos* para derivaciones CRWL:

Teorema 5 (Preservación de tipos para CRWL, [45](Th. 2)) *Consideremos un programa bien tipado $\mathcal{P}$. Si $T \vdash_{WT} e :: \tau$ y $\mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t$ entonces $T \vdash_{WT} t :: \tau$.*

Para conseguir este resultado es imprescindible la ausencia de variables extra, de otra manera los tipos pueden no preservarse. Un sencillo ejemplo es la función $wild \in FS^2$ de [45] con tipo $(wild :: \alpha \to \beta)$, definido con la regla bien tipada

$$wild\ X \to Y \,\square\, \{X :: \alpha, Y :: \beta\}$$

Con esta función tenemos que $\{X :: \alpha\} \vdash_{WT} wild\ X :: bool$ (en realidad $wild\ X$ puede tener cualquier tipo) y $wild\ X \twoheadrightarrow zero$ (debido a que en CRWL las variables extra se pueden instanciar a cualquier patrón durante la aplicación de la regla), pero no se cumple que $\{X :: \alpha\} \vdash_{WT} zero :: bool$.

Como el cálculo CRWL no liga las variables libres de las expresiones, en [45] también se presenta una extensión de CLNC que produce ligaduras de orden superior seguras

donde $\{\overline{\alpha_n}\} = ftv(\tau)$.

[19] En [45] consideran reglas con condiciones de c-convergencia, aunque nosotros las omitiremos ya que pueden sustituirse por construcciones *if_then* en los lados derechos (ver Sección 4.1).

con respecto a los tipos. Para ello, extiende los objetivos de CLNC original con un entorno de tipos T y una parte resuelta S_t de ecuaciones de tipo $\alpha \approx \tau$. Las reglas que generan ligaduras de orden superior son completadas con comprobaciones de tipos para garantizar que la ligadura respeta el tipo de la variable. Además, aunque no necesiten comprobaciones de tipos, el resto de reglas son completadas con operaciones de actualización y extensión del entorno de tipos, para dejarlo en un estado coherente tras aplicar la regla. Esta extensión de CLNC es correcta y completa con respecto al cálculo de soluciones *bien tipadas* del objetivo inicial. Obsérvese que este cálculo soluciona algunos de los problemas de tipos que produce el estrechamiento con las variables de orden superior mostrados en el Ejemplo 2 (página 6). En cambio, presenta algunas limitaciones. Primero, no permite que las constructoras tengan argumentos funcionales. Tampoco permite variables extra en las reglas, restricción que limita en cierta manera la utilización de variables libres (las variables extra se introducen como variables libres en los objetivos), que sí son soportadas. El *casting* polimórfico del Ejemplo 1 (página 4) se evita por definición, ya que las reglas no forman un programa válido porque contienen el patrón opaco $snd\ X$. Esta limitación es demasiado restrictiva, ya que no todos los patrones opacos generan problemas de tipos, como se verá en la Sección 6. Por último, los resultados de corrección y completitud del cálculo CLNC extendido sólo son válidos bajo la suposición de que no se producen pasos de descomposición opaca. Además, en [45] se demuestra que es indecidible saber si la evaluación de un objetivo CLNC producirá un paso de descomposición opaca, incluso para programas sencillos (reglas sin condiciones de c-convergencia y cuyos lados izquierdos contienen patrones formados solo por constructoras).

Aparte de [45, 47], otros trabajos consideran los sistemas de tipos en PLF. En [9] se considera un lenguaje lógico-funcional donde las variables están explícitamente tipadas, y cuyo método de evaluación es la relación estándar de estrechamiento sobre sistemas de reescritura de términos. Asimismo soporta patrones de orden superior en los lados izquierdos de las reglas, y también en los objetivos de igualdad estricta. A partir de este lenguaje, proponen una transformación de orden superior a primer orden (siguiendo el espíritu de la transformación clásica de Warren [151]) que produce programas de primer orden bien tipados. En este trabajo se demuestra la adecuación de la transformación, es decir, que el estrechamiento de primer orden sobre el programa transformado es correcto y completo con respecto al estrechamiento de orden superior sobre el programa original. Dicha transformación incorpora información de tipos en el programa de primer orden producido, gracias a la cual se consigue reducir el espacio de patrones sobre el que se realiza la búsqueda de ligaduras para las (originarias) variables de orden superior. Sin embargo, en [9] se utiliza un sistema de tipos monomórfico, dejando esbozada una posible extensión a tipos polimórficos mediante la generación de versiones monomórficas especializadas de las funciones polimórficas por cada tipo con el que son utilizados en el programa. Además, y aunque el lengua-

je original es tipado, los resultados de adecuación de la transformación consideran el conjunto de todas las soluciones, no solamente aquellas soluciones bien tipadas.

Basándose en las ideas de [50, 52] sobre resolución SLD tipada de primer orden, en [51] se propone un mecanismo de estrechamiento tipado de primer orden para calcular soluciones bien tipadas (considerando una transformación previa de orden superior a primer orden al estilo de la transformación de Warren [151]). Para ello requiere objetivos *completamente* anotados. En [51] se evitan las ligaduras mal tipadas durante la unificación, ya que esta fallará en esos casos debido a diferencias en el nivel de tipos. Un ejemplo sería la reducción de estrechamiento mal tipada $succ\ (F\ zero) \leadsto_{[F \mapsto and\ false]} succ\ false$ del Ejemplo 2 (página 6). La expresión completamente anotada sería $(succ^{nat \to nat}\ (F^{nat \to nat}\ zero^{nat})^{nat})^{nat}$, por lo que la solución mal tipada $[F \mapsto and\ false]$ no sería considerada porque $F^{nat \to nat}$ no unifica con $(and^{bool \to bool \to bool}\ false^{bool})^{bool \to bool}$ por discrepancias entre los tipos $nat \to nat$ y $bool \to bool$. Como se observa, [51] resuelve algunos de los problemas de la ligadura mal tipada de variables de orden superior debido a que realiza de manera implícita comprobaciones de tipo en tiempo de ejecución. En cambio, no soporta patrones de orden superior, por lo que no puede considerarse una solución a problemas como el *casting* polimórfico o la descomposición opaca (ver Ejemplo 1, página 4).

En otro orden de cosas, también han surgido varios trabajos tratando la integración de las clases de tipos de Haskell [150, 48] (método utilizado para definir funciones sobrecargadas, que presentaremos en la siguiente sección) en los sistemas lógico-funcionales. En [107] se estudian de manera preliminar algunas aplicaciones prometedoras que surgirían de una integración de las clases de tipos en el lenguaje Curry. Como ejemplos destacan la utilización de la búsqueda indeterminista para resolver la ambigüedad de tipo de las expresiones, la implementación de resolutores mediante una simulación de variables atribuidas (populares en algunas versiones de Prolog) por medio de la sobrecarga de la igualdad, o la mejora de la unificación de orden superior para algunos dominios donde es completa, como las funciones booleanas. Sin embargo, [107] es un trabajo mayoritariamente pragmático ya que carece de un enfoque formal, tanto desde el punto de vista semántico como desde el sistema de tipos, donde implícitamente se considera el de [150, 48]. Otro trabajo relativo a clases de tipos y Curry es [95], donde se transmite la experiencia de integrar las clases de tipos en una rama del sistema Münster Curry [94]. En este trabajo se abordan algunos problemas encontrados durante esta integración, como el manejo adecuado de las funciones sobrecargadas flexibles y rígidas (en el sentido de las funciones flexibles y rígidas de Curry) o el tratamiento de las restricciones de igualdad y los literales numéricos sobrecargados. Al igual que [107], [95] es un trabajo mayoritariamente pragmático que aporta soluciones a problemas concretos encontrados durante la integración, tomando como base el sistema de tipos usual con soporte para clases de tipos [150, 48].

5.3. Nociones y propuestas de tipos para PF

A diferencia de la PLF, los sistemas de tipos han recibido una gran atención por parte de la comunidad de PF durante las últimas décadas, surgiendo múltiples e interesantes propuestas. En esta sección presentaremos algunas de las más importantes, ya que algunos de sus formalismos o intuiciones están presentes en los sistemas de tipos propuestos en esta tesis.

5.3.1. Tipos existenciales

El sistema de tipos de DM considera esquemas de tipos donde algunas variables son universalmente cuantificadas. Esta cuantificación es indispensable para conseguir el polimorfismo paramétrico. No obstante, desde hace tiempo también se reconoce la utilidad de los *tipos existenciales* (tipos con cuantificación existencial) para conseguir ocultación de información efectiva y tipos abstractos de datos [25, 105]. Basado en esas ideas, en [112, 79] se propone una manera sencilla de integrar los tipos existenciales sin apenas ningún cambio sintáctico en el marco usual DM, mediante las llamadas *constructoras de datos existenciales*. En estas constructoras el tipo final no refleja el tipo de todos los elementos contenidos. Un ejemplo de constructora existencial es $CS^2 \ni mkKey : \forall\alpha.\alpha \to (\alpha \to nat) \to key$[20], ya que contiene dos elementos (de tipos α y $\alpha \to nat$ respectivamente) cuyos tipos no están reflejados en el tipo final key. Esta constructora fuerza la ocultación de información ya que en un patrón $mkKey\ X\ F$ sabremos que X tiene algún tipo τ, aunque no se conozca concretamente cuál es. Lo que sí que se conoce es que F es una función que acepta elementos de ese tipo τ y devuelve números naturales ($\tau \to nat$), por lo que la aplicación $F\ X$ será segura desde el punto de vista de los tipos. De esta manera, se puede utilizar $mkKey$ para construir el tipo de datos abstracto de las claves, donde X será la *representación interna desconocida*, y F será la *interfaz* mediante la cual se puede interactuar con la clave. Utilizando la interfaz, se pueden crear funciones que operan de manera uniforme sobre las claves, sin importar su representación interna (por claridad incluimos la declaración de la constructora existencial $mkKey$ utilizando una sintaxis similar a la de GADTs):

Ejemplo 11 (Constructoras existenciales y ocultación de información [79])

```
data key where
  mkKey :: A -> (A -> nat) -> key

getKey :: key -> nat
getKey (mkKey X F) = F X
```

[20] Este tipo es equivalente a $(\exists\alpha.\alpha \to (\alpha \to nat)) \to key$, de lo que proviene su nombre de constructora existencial.

Se pueden crear claves con distintas representaciones internas, como por ejemplo $(mkKey\ [1,2,3]\ length\_nat)$[21] o $(mkKey\ zero\ succ)$. Un aspecto muy potente de estos tipos de datos abstractos mediante constructoras existenciales es que, a diferencia de los sistemas de módulos disponibles en lenguajes como Standard ML [104], son simples valores. Por lo tanto, es posible pasarlos como argumentos de funciones y devolverlos como resultados de ellas. Esto no es posible con los módulos, ya que no son valores de primer nivel sino una metodología para separar la implementación de la interfaz.

Las constructoras existenciales son declaradas por el programador para fomentar la ocultación de información, por ello cualquier situación en la que esto se viole es rechazada por el sistema de tipos (aunque no genere errores de tipos). Esto ocurre cuando expresiones cuyo tipo contiene variables existenciales escapan al ámbito del encaje de patrones. Por ejemplo, una regla de función $f_1\ (mkKey\ X\ F) \to X$ sería rechazada ya que no conocemos el tipo de X, con lo que no se puede determinar el tipo de la función f_1. También se rechazaría una función si en el lado derecho se trata de concretar un tipo con variables cuantificadas existencialmente. Concretamente, una regla como $f_2\ (mkKey\ X\ F) \to not\ X$ sería rechazada ya que, aunque no conocemos el tipo de X a partir del encaje de patrones, el lado derecho requiere que tenga tipo booleano. Esta situación también ocurriría si el propio encaje de patrones hace suposiciones sobre el tipo que tendrá un argumento existencial: $f_3\ (mkKey\ true\ F) \to false$ sería rechazado ya que supone que el primer argumento será un booleano, hecho que se pretende ocultar ya que es un argumento cuantificado existencialmente en $mkKey$.

Para garantizar la ocultación de información, el sistema de tipos trata de manera especial las constructoras existenciales cuando aparecen en un encaje de patrones. En estos casos, en lugar de derivar una instancia genérica de su tipo declarado (como sería lo usual según DM), genera una instancia en la que las variables existencialmente cuantificadas son sustituidas por unas constantes frescas κ llamadas *constantes de Skolem*. Estas constantes son diferentes de cualquier otro tipo simple τ o constante de Skolem κ'. En las anteriores funciones f_1–f_3 el tipo asignado a la constructora $mkKey$ sería $\kappa \to (\kappa \to nat) \to key$, por lo que X y F tendrían tipos κ y $\kappa \to nat$ respectivamente. La función f_1 sería rechazada porque el tipo final de la función contiene una constante de Skolem, mientras que f_2 y f_3 serían rechazadas por incompatibilidad de tipos entre κ y $bool$: la primera porque el lado derecho $not\ X$ está mal tipado y la segunda porque el propio patrón del lado izquierdo $mkKey\ true\ F$ no admite ningún tipo.

La ocultación de información que se obtiene con el uso de constructoras existenciales es similar a la opacidad generada por los patrones de orden superior. Por ejemplo, tanto en $mkKey\ X\ F$ como en $snd\ X$, el tipo de la X es desconocido a partir del tipo del patrón. Sin embargo, a diferencia de las constructoras existenciales, la opacidad generada por los patrones de orden superior no es planeada (proviene del tipo declarado para una constructora) sino sobrevenida por el uso de aplicaciones parciales de

[21] Considerando que $length\_nat$ es la función que calcula la longitud de una lista y la devuelve como un natural, de tipo $\forall\alpha.[\alpha] \to nat$.

funciones. Esto nos inclina a pensar que en el marco de la PLF puede ser más adecuado realizar un tratamiento diferente de la ocultación de información, aceptando funciones que son rechazadas por los sistemas de tipos de constructoras existenciales [112, 79] siempre y cuando se garantice la seguridad de tipos.

Como comentario final, destacar que las constructoras existenciales son un recurso que ha sido ampliamente aceptado en los sistemas de PF como Haskell (GHC [41], Hugs [74], EHC [35]) o Clean [122], y también en sistemas de PLF como Mercury [139].

5.3.2. Clases de tipos

El sistema de tipos de DM permite polimorfismo paramétrico, es decir, funciones que operan *de manera uniforme* en varios tipos diferentes. Un ejemplo sería la función $reverse :: \forall\alpha.[\alpha] \to [\alpha]$ para invertir listas, que puede funcionar sobre diferentes listas ($[bool]$, $[nat]$, $[bool \to bool]$) pero cuya definición es la misma. En contraposición a este tipo de polimorfismo existe el polimorfismo *ad-hoc* o sobrecarga de funciones, que permite que una función opere sobre distintos tipos pero actuando de manera *diferente* para cada tipo. Un ejemplo sería la igualdad, que funciona sobre booleanos, naturales, listas ... pero cuya definición varía dependiendo del tipo que sea. Para integrar el polimorfismo *ad-hoc* en el sistema de tipos DM, Wadler y Blott propusieron las *clases de tipos* [150][22], que es un mecanismo para agrupar y declarar las funciones que tienen polimorfismo *ad-hoc* (*clases*) y definirlas para los diferentes tipos (*instancias*). Esta propuesta ha cosechado mucho éxito en la comunidad funcional, principalmente en el lenguaje Haskell, donde han surgido múltiples trabajos acerca de su implementación y sistema de tipos/inferencia [49, 17, 113, 109, 48].

Una clase de tipos K es una familia de tipos τ que tienen definidos un cierto conjunto de funciones. Mediante una declaración de clase de tipos se fija las funciones que deben estar definidas sobre un tipo para que este pertenezca a dicha clase. Por otro lado, una declaración de instancia asegura la pertenencia de un tipo particular a una clase de tipos mediante la definición de las funciones necesarias de la clase. Para soportar las clases e instancias de tipos, los esquemas de tipos se extienden con *contextos* $\forall\overline{\alpha_m}.\langle K_1\ \alpha_1, \ldots, K_n\ \alpha_n\rangle \Rightarrow \tau$, que representan que cada tipo α_i debe estar en la clase de tipos K_i. El siguiente ejemplo, utilizando sintaxis adaptada a Toy, muestra la definición de la clase eq de igualdad sobre enteros y booleanos, aparte de la definición de la función $member$ que comprueba si un elemento está en una lista mediante la igualdad:

Ejemplo 12 (Igualdad mediante clases de tipos)

```
class eq A where
  eq :: A -> A -> bool
```

[22] Una propuesta similar, aunque menos potente, fue propuesta con anterioridad y de manera independiente por Stefan Kaes [75].

```
instance eq bool where          instance eq nat where
  eq true true   = true           eq zero zero         = true
  eq true false  = false          eq zero (succ Y)     = false
  eq false true  = false          eq (succ X) zero     = false
  eq false false = true           eq (succ X) (succ Y) = eq X Y

member :: (eq A) => A -> [A] -> bool
member X []     = false
member X (Y:Ys) = (eq X Y) \/ member X Ys
```

La declaración de la clase eq solo contiene la función eq (podría contener varias funciones), cuyo tipo será $\forall\alpha.\langle eq\ \alpha\rangle \Rightarrow \alpha \to \alpha \to bool$. El significado es que la función eq tiene tipo $\forall\alpha.\alpha \to \alpha \to bool$, pero solo para aquellos α que estén en la clase eq. Las instancias $eq\ bool$ y $eq\ nat$ sirven para expresar que $bool$ y nat están en la clase eq, definiendo la función eq para esos tipos. Como puede verse, la definición de dicha función es completamente ad-hoc *para cada tipo. Por último, como la función $member$ requiere comprobaciones de igualdad sobre los elementos de la lista, eso queda reflejado en su tipo mediante el contexto correspondiente.*

La implementación estándar de clases de tipos no necesita ninguna extensión del lenguaje, sino que realiza una transformación de programa [150, 48]. Esta transformación acepta un programa bien tipado con clases de tipos y produce otro sin este tipo de construcciones que está bien tipado en un sistema DM estándar. Para ello se utilizan los *diccionarios*, que son tipos de datos que contienen las versiones especializadas de las funciones sobrecargadas. Cada clase de tipos da lugar a una declaración de constructora del diccionario asociado. Considerando el ejemplo anterior, la clase eq daría lugar a la constructora $dictEq$ de tipo $\forall\alpha.(\alpha \to \alpha \to bool) \to dictEq\ \alpha$, donde su único argumento sería la función de igualdad particular. De esta manera un diccionario $dictEq\ F$ de tipo $dictEq\ bool$ contendría la función $F : bool \to bool \to bool$ de igualdad de booleanos. A su vez, cada función sobrecargada que forma parte de una clase de tipos es transformada en una función selectora que extrae la función adecuada del diccionario de la clase. En el ejemplo anterior la función sobrecargada eq daría lugar a la función extractora $eq :: \forall\alpha.dictEq\ \alpha \to (\alpha \to \alpha \to bool)$, generando la regla $eq\ (dictEq\ F) = F$. Con respecto a las instancias, estas generan diccionarios específicos para cada tipo. Considerando el ejemplo anterior la instancia $eq\ bool$ generaría el diccionario $dictEqBool = dictEq\ eq_{bool}$, con tipo $dictEqBool : dictEq\ bool$, y la instancia $eq\ nat$ el diccionario $dictEqNat = dictEq\ eq_{nat}$, con tipo $dictEqNat : dictEq\ nat$. En ambos casos consideramos que $eq_{bool}, eq_{nat} \in FS^2$ son funciones generadas automáticamente utilizando las reglas de las instancias, con tipos $bool \to bool \to bool$ y $nat \to nat \to bool$ respectivamente. Finalmente, se introducen los diccionarios como argumentos de las funciones que los necesiten. En el ejemplo anterior, a partir del tipo de $member$ se deduce que necesita un diccionario de tipo $dictEq\ \alpha$ como argumento, que debe ser

pasado a la aplicación de eq y a la llamada recursiva. A la hora de evaluar expresiones que utilicen (directa o indirectamente) funciones sobrecargadas, se introducen los diccionarios de los tipos adecuados. Por ejemplo la expresión $member\ zero\ [zero]$ sería transformada a $member\ (dictEqNat)\ zero\ [zero]$, que se evaluaría a $true$. El siguiente ejemplo contiene la traducción completa del Ejemplo 12 utilizando diccionarios.

Ejemplo 13 (Traducción del Ejemplo 12 mediante diccionarios)

```
data dictEq A = dictEq (A -> A -> bool)

eq :: dictEq A -> (A -> A -> bool)
eq (dictEq F) = F

eq_bool :: bool -> bool -> bool    eq_nat :: nat -> nat -> nat
eq_bool true true   = true         eq_nat zero zero         = true
eq_bool true false  = false        eq_nat zero (succ Y)     = false
eq_bool false true  = false        eq_nat (succ X) zero     = false
eq_bool false false = true         eq_nat (succ X) (succ Y) = eq X Y

dictEqBool :: dictEq bool          dictEqNat :: dictEq nat
dictEqBool = dictEq eq_bool        dictEqNat = dictEq eq_nat

member :: dictEq A -> A -> [A] -> bool
member DEq X []     = false
member DEq X (Y:Ys) = (eq DEq X Y) \/ member DEq X Ys
```

La implementación de clases de tipos mediante diccionarios se comporta muy bien con respecto a la separación en módulos, permitiendo que las clases e instancias estén dispersas entre varios archivos sin complicar la compilación. Además produce programas transformados eficientes, en parte gracias a una serie de optimizaciones que se pueden aplicar [49, 12, 72]. Este, sin embargo, no es el único método de implementación posible. En [146] se propone una traducción alternativa en la que se pasan tipos como argumentos en lugar de pasar diccionarios, que sirven para seleccionar qué comportamiento se espera de la función sobrecargada. Por el contrario, esta traducción requiere que el lenguaje destino pueda realizar encaje de patrones sobre tipos o tenga algún mecanismo adecuado para representarlos.

Debido al gran éxito de las clases de tipos para proporcionar polimorfismo *ad-hoc*, se han desarrollado en la comunidad funcional multitud de extensiones y generalizaciones sobre las clases de tipos originales de [150, 48]: las clases de tipos multiparámetro [116], las clases de constructoras [71], las dependencias funcionales [73] o la combinación de tipos existenciales y clases de tipos [78] (todas ellas integradas en el sistema GHC [41]). Además de la comunidad funcional, las clases de tipos también han despertado interés en el ámbito lógico-funcional. En [107] se presentan nuevas posibilidades

surgidas de la integración de las clases de tipos en la PLF, como la resolución de la ambigüedad mediante la búsqueda indeterminista o la simulación de variables atribuidas mediante la sobrecarga del operador de igualdad. Por otro lado, en [95] se presentan soluciones a algunos de los problemas surgidos de la experiencia de integrar las clases de tipos en el sistema Münster Curry [94], como el tratamiento de las restricciones de igualdad, las funciones sobrecargadas flexibles y rígidas (al estilo de las funciones flexibles y rígidas de Curry) o los literales numéricos sobrecargados. Estos trabajos toman un enfoque fundamentalmente práctico, basándose de manera implícita en el sistema de tipos clásico de [150, 48] y en la traducción estándar mediante diccionarios. Aparte de estos trabajos teóricos, también han aparecido sistemas (como la mencionada rama del sistema Münster, o los sistemas Zinc [16] y Sloth [40]) integrando clases de tipos en Curry, aunque todos ellos están en una fase experimental, e incluso los dos últimos parecen haber sido abandonados desde 2006.

5.3.3. Tipos de datos algebraicos generalizados (GADTs)

Los tipos de datos algebraicos generalizados (GADTs) [119, 124, 134] —también llamados *guarded recursive data types* [155] o *first-class phantom types* [28]— son una generalización simple pero muy potente de los tipos de datos clásicos en lenguajes funcionales. En estos lenguajes funcionales los tipos de datos se definen mediante declaraciones `data`. Sin embargo, todas las constructoras así definidas deben tener el mismo tipo resultado. Por ejemplo, una declaración `data rep A = rNat | rBool` define las constructoras $rNat$ y $rBool$, pero ambas tienen tipo $\forall\alpha.rep\ \alpha$. Los GADTs superan esta restricción, permitiendo que cada constructora declarada tenga un tipo resultado diferente. Por ejemplo, se podrían definir constructoras para representar términos de un lenguaje sencillo:

```
data term A where
  tZero   :: term nat
  tSucc   :: term nat -> term nat
  tIsZero :: term nat -> term bool
  tIf     :: term bool -> term A -> term A
```

De esta manera, tendríamos que $tZero$ y $tSucc$ construyen datos de tipo $term\ nat$, $tIsZero$ datos de tipo $term\ bool$ y tIf datos de tipo $term\ \alpha$. Aparte de generalizar la definición de constructoras, también se liberaliza el sistema de tipos para aceptar estas constructoras con tipos concretos en argumentos polimórficos de funciones:

```
eval :: term A -> A
eval tZero = zero
eval (tSucc T) = succ (eval T)
eval (tIsZero T) = (eval T) == zero
eval (tIf B T) = if (eval B) then (eval T1)
```

El argumento de $eval$ tiene como tipo declarado $term\ \alpha$, no obstante, los patrones que aparecen en algunas reglas tienen tipos más concretos $term\ nat$ y $term\ bool$. Aunque esta situación no está permitida en DM ya que da lugar a la no preservación de tipos, en los sistemas con GADTs se relaja de manera segura en presencia de argumentos formados con estas constructoras. Esto da lugar a un sistema de tipos que carece de tipos principales, ya que algunas expresiones pueden tener varios tipos incomparables, y donde la inferencia de tipos alcanza una complejidad alta en comparación con la clásica de DM [119, 124, 134]. A pesar de ello, los GADTs son una extensión con múltiples aplicaciones como la programación genérica [61, 64].

5.3.4. Parametricidad

Los tipos de las funciones polimórficas (entendidas en el sentido de polimorfismo paramétrico al estilo de DM) proporcionan más información de la que a primera vista se puede pensar. Consideremos la función $f : \forall\alpha.[\alpha] \to [\alpha]$, que acepta listas de cualquier tipo y devuelve listas del mismo tipo. Aunque en principio parece que no podemos saber nada de ella sin conocer su definición, gracias al polimorfismo paramétrico disponemos de más información. Como las listas que acepta son de tipo $[\alpha]$ y el polimorfismo paramétrico se caracteriza por tratar de manera *uniforme* todos los tipos, sabemos que la función f no podrá inspeccionar los elementos de la lista. Lo único que podrá hacer es reordenarlos o descartarlos, basándose en criterios que no dependen del tipo de los elementos como su posición o la longitud de la propia lista. Tampoco podrá generar y añadir nuevos elementos a la lista, ya que se desconoce el tipo de los mismos. Basándose en este hecho, piedra angular del teorema de parametricidad [148] (también conocido como teorema de abstracción [126]), Wadler [148] formula los *teoremas gratis* (*theorems for free*) que surgen de manera automática a partir de los tipos de las funciones. De esta manera, a partir del tipo anterior de f y una función $g : \tau_1 \to \tau_2$ se puede deducir la igualdad:

$$map\ g\ (f\ X) = f\ (map\ g\ X)$$

La idea de esta igualdad es que como la función f solo puede reordenar o descartar elementos basándose en sus posiciones o en la longitud de la lista, da igual aplicar $map\ g$ antes o después de f ya que $map\ g$ solo modificará los elementos, que f no puede inspeccionar. Sin embargo, los teoremas gratis de [148] consideran como modelo de cómputo el λ-cálculo polimórfico [125] (también llamado sistema F [42]), que es *normalizante*, es decir, donde toda reducción de un término termina en forma normal. Por ello, su aplicación a lenguajes de programación reales que soporten no terminación o funciones parciales puede requerir debilitar el teorema, imponiendo condiciones sobre las funciones involucradas [148, 137]. La razón es que en estos lenguajes podemos generar nuevos valores de tipo polimórfico $\forall\alpha.\alpha$ como el fallo de encaje de patrones

$head\ [\,]$ o la expresión no terminante $loop$ (definida con la regla $loop \to loop$). De esta manera, la función f de tipo $\forall\alpha.[\alpha] \to [\alpha]$, que antes únicamente podía reordenar o descartar elementos, ahora puede añadir nuevos elementos a la lista devuelta. Por ejemplo la anterior igualdad sería inválida en un lenguaje perezoso para $f\ X \to [loop]$, $g\ X \to true$ y $X = [\,]$:

$$map\ g\ (f\ [\,]) = [true] \neq [\bot] = f\ (map\ g\ [\,])$$

Para recuperar la validez de la igualdad sería necesario exigir que g fuese estricta en su argumento (es decir, $g\ \bot\ =\ \bot$) para así contrarrestar la no terminación de $loop$. Este contraejemplo no funcionaría en un lenguaje impaciente, ya que tendríamos $map\ g\ (f\ [\,]) = \bot = f\ (map\ g\ [\,])$. En un lenguaje impaciente, en cambio, sí sería posible obtener un contraejemplo tomando $f\ (X : Xs) \to [X]$ y la función parcial $g\ 0 \to 0$:

$$map\ g\ (f\ [0,1]) = [0] \neq \bot = f\ (map\ g\ [0,1])$$

En este caso, para invalidar el contraejemplo sería necesario exigir que g fuese una función total.

En el marco lógico-funcional, es conocido que el indeterminismo invalida tanto los teoremas gratis como algunos razonamientos ecuacionales válidos en PF [29]. Un ejemplo es la siguiente igualdad relacionando las funciones predefinidas $filter$ y map:

$$filter\ p\ (map\ h\ As) = map\ h\ (filter\ (p \circ h)\ As)$$

Si consideramos $p\ X \to X$, la función indeterminista h definida con las reglas $\{h\ X \to true, h\ X \to false\}$ y $As = [zero]$ la igualdad no sería válida. Por un lado tendríamos que $filter\ p\ (map\ h\ [zero])$ solo se puede evaluar a $[true]$ y $[\,]$, mientras que $map\ h\ (filter\ (p \circ h)\ [zero])$ se podría evaluar a los valores $[true]$, $[false]$ y $[\,]$. Aparte del indeterminismo, las variables extra de las reglas y el estrechamiento también invalidan la aplicación de los teoremas gratis [29], aunque se pueden demostrar versiones debilitadas de algunos de ellos considerando distintas condiciones sobre las funciones involucradas.

Además de los teoremas gratis, la parametricidad es una propiedad útil en los sistemas de programación funcionales. Por ejemplo se utiliza para garantizar la corrección de optimizaciones como la *short-cut deforestation* de GHC [65], una versión ligera de la *deforestación* de [149], transformación utilizada para eliminar estructuras de datos intermedias en programas que utilizan listas. La parametricidad también sirve de ayuda a los propios compiladores a la hora de representar las constructoras de manera concisa. Como las funciones que aceptan argumentos polimórficos no podrán inspeccionar dichos elementos, no es ningún problema si constructoras de distintos tipos se representan internamente de la misma manera [114]. Si una función así es llamada con constructoras de tipos distintos, la representación no será problema ya que no será inspeccionada. En funciones con argumentos no polimórficos será el propio sistema

de tipos el que garantizará que las constructoras pasadas como argumento sean de un cierto tipo, por lo que el compilador solo necesita garantizar que la representación de las constructoras es diferente *dentro de cada tipo*. La parametricidad es clave para esta decisión de diseño, de otra forma se podrían confundir constructoras de distintos tipos. Esto se aprecia en la siguiente función:

```
f :: A -> A
f zero = zero
f true = true
```

Esta función no goza de parametricidad, ya que aunque el argumento es polimórfico sus reglas realizan encaje de patrones sobre el mismo. En esta situación, utilizar la misma representación interna para las constructoras de distinto tipo $zero$ y $true$ provocaría que tanto $f\ zero$ como $f\ true$ se evaluasen incorrectamente a $zero$, violando además la preservación de tipos en el segundo caso.

5.3.5. Programación genérica

Por *programación genérica* [14] entendemos cualquier método que permite utilizar un programa en distintas situaciones. Para ello se suele abstraer la parte del programa que es común a todas las situaciones, formando un programa genérico que puede ser especializado para cualquier situación. Sin embargo, dentro de la programación genérica se pueden entender diversos métodos. El polimorfismo paramétrico característico de DM puede ser considerado como programación genérica, ya que permite crear funciones que operan de manera uniforme sobre distintos tipos. Por ejemplo la función $reverse$ puede ser usada para invertir listas de naturales, de booleanos . . . incluso listas de tipos de datos que se definan en el futuro. Las clases de tipos [150, 48] también pueden considerarse como programación genérica ya que permiten sobrecargar funciones de manera particular para cada tipo, permitiendo su utilización en diversas situaciones. De manera similar, las *funciones indexadas por tipo* [64] (*type-indexed functions*), funciones que tienen diferentes definiciones para diferentes tipos, también son una forma de programación genérica. Por último, la *programación genérica de tipos de datos* (*datatype-generic programming*) o *programación polítípica* permite definir funciones que operan sobre cualquier tipo de datos. Este método, más potente que el polimorfismo paramétrico, se basa en una representación uniforme de las constructoras de datos (mediante *suma de productos* [14] o *spines* [64]). Al definir funciones sobre esta representación uniforme, se consigue que operen sobre cualquier tipo de datos, ya sea existente o que el programador pueda definir en el futuro. A diferencia del polimorfismo paramétrico, las funciones genéricas sí que pueden «meta-inspeccionar» los argumentos polimórficos representados de manera uniforme, y tener un comportamiento diferente dependiendo de la estructura de estos. Esta situación es parecida a lo que realiza el lenguaje Haskell mediante las derivaciones automáticas de instancias

de clases (utilizando la construcción $deriving$) para los tipos de datos declarados por el programador. De manera interna, el compilador tiene definidas de manera genérica las funciones de clases de tipos como Eq, Ord y $Show$, y genera versiones especializadas para los nuevos tipos de datos.

En Haskell, la programación genérica de tipos de datos ha sido objeto de gran interés por parte de la comunidad, surgiendo multitud de alternativas [63, 129]. Algunas de ellas proponen extensiones del lenguaje, como *PolyP* [70], *Generic Haskell* [59] o *Template Haskell* [138], aunque este último fue integrado en el sistema GHC a partir de la versión 6. Otras alternativas se ofrecen como librerías del sistema, como *PolyLib* [110], *Scrap Your Boilerplate* [77] o *RepLib* [152]. Por último, también existen propuestas *ligeras* que permiten programación genérica utilizando directamente recursos del lenguaje como pueden ser tipos existenciales [27], clases de tipos [62] o GADTs [64]. Aparte de Haskell, la programación genérica también ha sido integrada en otros lenguajes funcionales como Clean [2].

Parte III
Sistemas de tipos propuestos

En esta parte presentaremos, de manera unificada, los sistemas de tipos desarrollados en esta tesis y sus resultados relacionados. Estos sistemas están íntegramente contenidos en los artículos asociados a la tesis, que se pueden encontrar en la Parte V (página 130). Para facilitar la lectura indicaremos, en los enunciados presentados (teoremas, lemas, ejemplos . . .), el artículo en que aparecen y su numeración.

6. Sistema $\vdash^{\bullet}$

Este capítulo presenta el sistema de tipos del artículo *New results on type systems for functional logic programming* [84](A.1). A este sistema se le llamará *sistema* $\vdash^{\bullet}$, leído *sistema punto negro*. Las demostraciones se pueden encontrar en *Advances in Type Systems for Functional Logic Programming (Extended Version)* [86](B.1), además de en la tesis de máster del autor [99].

6.1. Motivación y objetivos

Como se ha visto en el Ejemplo 1 (página 4), los patrones de orden superior dan lugar a problemas de tipos (como el *casting* polimórfico y la descomposición opaca) incluso en un marco reducido donde solo se considera la reescritura de expresiones. Este comportamiento no deseado de los patrones de orden superior fue detectado origina-

riamente en [45]. En ese trabajo presentan la noción de patrón opaco para capturar aquellos patrones de orden superior en los cuales no se puede determinar de manera unívoca el tipo de sus componentes a partir del tipo del patrón entero. De esta manera, los problemas de tipos como el *casting* polimórfico se evitan de manera directa excluyendo de la definición de regla de programa todas aquellas que contengan patrones opacos en sus lados izquierdos. Esta solución, aunque segura desde el punto de vista de los tipos, es demasiado restrictiva ya que la mera presencia de patrones opacos no genera siempre problemas de tipos. Consideremos el siguiente programa, donde snd está definido como en el Ejemplo 1:

Ejemplo 14 (Patrones opacos seguros)

```
snd :: A -> B -> B          f :: (A -> A, B) -> B
snd X Y = Y                 f (snd X, Y) = Y

g :: (A -> A) -> bool       h :: (A -> A) -> bool
g (snd X) = true            h (snd true) = false
```

Los patrones de las reglas f, g y h son opacos ya que contienen la aplicación parcial de snd, función que no es 1-transparente, por lo que serían reglas inválidas en [45]. Sin embargo, ninguna de estas reglas genera problemas de tipos. En el caso de f, aunque la componente $snd\ X$ de la tupla es opaca, la regla solo utiliza la segunda componente, cuyo tipo sí que está unívocamente determinado por el tipo del patrón. En el caso de g, el tipo de X no puede ser determinado por el tipo del patrón $snd\ X$, sin embargo, esto no genera ningún problema de tipos ya que este tipo desconocido no es utilizado en el lado derecho (de hecho, X ni siquiera aparece en el lado derecho). El caso de h es similar al de g, con la salvedad de que no hay ninguna componente del patrón cuyo tipo sea desconocido. La única posibilidad sería el argumento de snd, pero su tipo queda fijado a $bool$ ya que se trata de la constructora $true$.

De manera independiente a los patrones opacos, y sin generar problemas de tipos, el tipado de let-expresiones con patrones en sus ligaduras es un aspecto que es tratado de manera diferente en los distintos sistemas de programación funcional y lógico-funcional. Esto puede observarse en las siguientes expresiones, tomadas de [84](A.1, Ex. 2):

$$\begin{aligned} e_1 \equiv &\quad let\ F = id\ in\ (F\ true, F\ zero) \\ e_2 \equiv &\quad let\ [F, G] = [id, id]\ in\ (F\ true, F\ zero, G\ zero, G\ true) \end{aligned}$$

Aunque ambas expresiones definen F y G como la identidad y la aplican a valores de distintos tipos, los distintos sistemas funcionales y lógico-funcionales las consideran de manera diferente. Algunos sistemas tratan la definición mediante let-expresiones de manera monomórfica, por lo que ambas expresiones están mal tipadas al utilizar F y G con distintos tipos ($bool \to bool$ y $nat \to nat$). Otros sistemas consideran que la definición mediante let-expresiones es polimórfica, por lo que ambas expresiones

Lenguaje de programación y versión	let_m	let_{pm}	let_p
GHC 6.8.2		×	
Hugs Sept. 2006			×
Standard ML of New Jersey 110.67		×	
Ocaml 3.10.2			×
F# Sept. 2008			×
Clean 2.0	×		
TOY 2.3.1*	×		
Curry PAKCS 1.9.1	×		
Curry Münster 0.9.11		×	
KICS 0.81893	×		

(*) utilizamos construcciones `where` en lugar de `let`, pues estas últimas no están soportadas en Toy.

Figura 9: Let-expresiones en diferentes lenguajes de programación

estarían bien tipadas. También hay sistemas que consideran que el grado de polimorfismo otorgado a las let-expresiones depende de si el lado izquierdo de la ligadura es una variable o un patrón compuesto. De esta manera, si es una variable lo tratan de manera polimórfica (aceptando e_1 como expresión bien tipada), mientras que si el lado izquierdo es un patrón compuesto lo tratan de manera monomórfica (rechazando por tanto e_2)[23]. La opción escogida por los distintos sistemas puede verse en la Figura 9 —utilizando let_m para representar las let-expresiones monomórficas, let_p para las polimórficas y let_{pm} para las mixtas—, donde se aprecia que hay poca uniformidad.

El objetivo del sistema $\vdash^\bullet$ es proporcionar un sistema de tipos adecuado para PLF en el marco de reescritura indeterminista con *call-time choice* (es decir, utilizando la semántica de let-reescritura) que trate los patrones de orden superior de manera más relajada que en [45]. También se persigue que sea el propio sistema de tipos el que rechace los patrones problemáticos, y no sea una restricción sobre los propios programas considerados. Con respecto a los distintos grados de polimorfismo para let-expresiones, se pretende formalizar las tres opciones consideradas anteriormente (distinguiéndolas sintácticamente mediante la notación let_m, let_p y let_{pm} mencionada anteriormente), y demostrar los resultados para todas ellas. De esta manera, estos resultados serán independientes de la opción escogida por el sistema final. Por último, se pretende desarrollar un algoritmo de inferencia de tipos para expresiones y programas, que permita su integración en el sistema Toy.

[23] La opción contraria, es decir, tratar monomórficamente las variables y polimórficamente los patrones compuestos, no es utilizada en ninguno de los sistemas probados ni parece tener mucho sentido en la práctica.

6.2. Sistema de tipos: derivación e inferencia

Para el sistema de tipos $\vdash^{\bullet}$ consideraremos la sintaxis de expresiones utilizada en la let-reescritura (ver Figura 4, página 23) pero extendida con λ-abstracciones y los distintos tipos de let-expresiones:

$$Exp \ni e ::= X \mid c \mid f \mid e\ e \mid \lambda t.e \mid let_m\ t = e\ in\ e \mid let_p\ t = e\ in\ e \mid let_{pm}\ t = e\ in\ e$$

donde $t \in Pat$. Para referirnos a cualquiera de las let-expresiones sin importar el polimorfismo utilizaremos la notación let_*. Las λ-abstracciones ($\lambda t.e$) no son tratadas por la semántica de la let-reescritura, por lo tanto no pueden aparecer en los programas ni en las expresiones a evaluar. En cambio, son consideradas como expresiones porque la noción de regla bien tipada se basará en la derivación de tipos para una λ-abstracción que asociaremos a la regla, como veremos más adelante. Los programas considerados son los mismos que en la Figura 4, es decir, conjuntos de reglas $f\ \overline{t_n} \to e$ cuyos lados izquierdos son lineales. De manera similar a [45], nos ceñiremos solamente a *reglas sin variables extra*. Esto es necesario porque en un marco de reescritura como el considerado aquí, las variables extra son instanciadas de manera libre al aplicar la regla, lo que produce fácilmente errores de tipos —ver [45](Ex. 5)—. Aunque en este sistema de tipos y en el sistema liberal de la próxima sección las variables extra estarán excluidas, serán tratadas en la Sección 8 (página 91), que utiliza estrechamiento como marco operacional.

El sistema $\vdash^{\bullet}$ se basa en una derivación de tipos básica $\vdash$, que luego se complementa con una fase de detección de situaciones problemáticas causadas por la opacidad. Las reglas de la relación $\vdash$ se pueden encontrar en la Figura 10. Como se puede observar, es un sistema de tipos basado en DM dirigido por la sintaxis (Figura 7-b, página 33) extendido con patrones en las λ-abstracciones y las tres opciones de polimorfismo en las let-expresiones. Para las λ-abstracciones, se asumen algunos tipos $\overline{\tau_n}$ para las variables del patrón t a la hora de derivar el tipo τ_t del patrón, y se deriva el tipo de e utilizando esas mismas suposiciones. En el caso de las reglas que tratan let-expresiones con patrones compuestos (LET$_m$, LET$^h_{pm}$ y LET$_p$), el método es similar: derivar un tipo τ_t para e_1, derivar el mismo tipo τ_t para t asumiendo algunos tipos $\overline{\tau_n}$ para las variables de t, y finalmente derivar un tipo para e_2 teniendo en cuenta los tipos asumidos. La única variación es si esos tipos asumidos se generalizan con $Gen(\tau_i, \mathcal{A})$, obteniendo un comportamiento polimórfico, o si se utilizan los tipos τ_i directamente, obteniendo el comportamiento monomórfico. A partir de este punto consideramos una definición de los *conjuntos de suposiciones* $\mathcal{A}$ ligeramente diferente a la presentada en la Sección 5.1. En lugar de sobre identificadores, contendrán suposiciones sobre símbolos, es decir, $\mathcal{A} \equiv \{\overline{s_n : \sigma_n}\}$ donde $s \in CS \cup FS \cup \mathcal{DV}$.

Para detectar las situaciones problemáticas utilizaremos la noción de *variable opaca*. Esta noción, basada en las mismas ideas que la de los patrones opacos de [45], sirve para identificar las variables de un patrón cuyo tipo no está unívocamente determinado

$$\text{(ID)}\frac{}{\mathcal{A} \vdash s : \tau} \text{ si } \mathcal{A}(s) \succ \tau$$

$$\text{(APP)}\frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$\text{(}\Lambda\text{)}\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau} \text{ si } \{\overline{X_n}\} = var(t)$$

$$\text{(LET}_m\text{)}\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash let_m\ t = e_1\ in\ e_2 : \tau_2} \text{ si } \{\overline{X_n}\} = var(t)$$

$$\text{(LET}^X_{pm}\text{)}\frac{\mathcal{A} \vdash e_1 : \tau_1 \quad \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A})\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau_2}$$

$$\text{(LET}^h_{pm}\text{)}\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash h\ \overline{t_m} : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash let_{pm}\ h\ \overline{t_m} = e_1\ in\ e_2 : \tau_2} \text{ si } \{\overline{X_n}\} = var(h\ \overline{t_m})$$

$$\text{(LET}_p\text{)}\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash let_p\ t = e_1\ in\ e_2 : \tau_2} \text{ si } \{\overline{X_n}\} = var(t)$$

Figura 10: Reglas del sistema de tipos básico $\vdash$

por el tipo del patrón. De esta manera se refina la localización de las situaciones problemáticas, moviéndola de los patrones a las variables dentro de ellos, permitiendo así una detección más concisa de las causas de los problemas de tipos. Para su definición formal utilizamos el sistema de tipos básico $\vdash$:

Definición 1 (Variable opaca de t con respecto a $\mathcal{A}$, [84](A.1, Def. 1)) *Sea un patrón t que admite algún tipo con respecto a $\mathcal{A}$. Decimos que $X_i \in \{\overline{X_n}\} = var(t)$ es una variable opaca de t con respecto a $\mathcal{A}$ si y solo si $\exists \overline{\tau_n}, \tau$ tal que $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$ y $ftv(\tau_i) \nsubseteq ftv(\tau)$. Al conjunto de todas las variables opacas de un patrón t con respecto a $\mathcal{A}$ lo denotaremos como $opaqueVar_{\mathcal{A}}(t)$. De manera similar a [45], a las variables que no son opacas en un patrón las llamaremos* variables transparentes.

$$\text{(P)}\frac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash^{\bullet} e : \tau} \text{ si } critVar_{\mathcal{A}}(e) = \emptyset$$

Figura 11: Regla del sistema $\vdash^{\bullet}$

Esta definición captura que el tipo de X_i no está unívocamente determinado por el tipo τ de t ya que $ftv(\tau_i) \not\subseteq ftv(\tau)$. De esta manera podríamos sustituir aquellas variables de tipo en $ftv(\tau_i)$ que no están en $ftv(\tau)$ por tipos diferentes, consiguiendo derivaciones de tipo de t para el mismo tipo τ en las cuales el tipo de X_i es diferente[24]. Por ejemplo, la variable X es opaca en $snd\ X$ porque podemos construir dos derivaciones $\mathcal{A} \oplus \{X : bool\} \vdash snd\ X : bool \to bool$ y $\mathcal{A} \oplus \{X : nat\} \vdash snd\ X : bool \to bool$ que asignan el mismo tipo $bool \to bool$ a $snd\ X$ pero donde la X toma tipos diferentes. Esto queda capturado por la anterior definición, ya que $\mathcal{A} \oplus \{X : \alpha\} \vdash snd\ X : bool \to bool$ y $ftv(\alpha) = \{\alpha\} \not\subseteq \emptyset = ftv(bool \to bool)$. Por otro lado, la variable X no es opaca en el patrón opaco $snd\ [X, true]$ ya que debido a la lista que contiene, cualquier derivación de tipos que asigne un tipo τ al patrón debe contener la suposición $\{X : bool\}$, por lo que el tipo de X está fijado por el patrón. Esto queda reflejado en la anterior definición, ya que $ftv(bool) = \emptyset \subseteq ftv(\tau)$ para cualquier τ.

La noción de variable opaca sirve para detectar variables problemáticas en los patrones, sin embargo, su presencia en los lados izquierdos de las reglas no genera automáticamente errores de tipos. Lo que produce problemas de tipos como el *casting* polimórfico (Ejemplo 1, página 4) es la aparición de dichas variables en los lados derechos, pues su tipo no es completamente conocido a partir del patrón. En el ejemplo del *casting* polimórfico, la variable X es opaca en el patrón $snd\ X$ de la regla de $unpack$ y además aparece en el lado derecho, por lo que dicha regla debe ser rechazada. Para caracterizar este tipo de variables que son opacas en un patrón de una λ-abstracción o let-expresión y que además aparecen en el resto de la expresión, es decir, aquellas que queremos evitar, utilizaremos la noción de *variable crítica*:

Definición 2 (Variables críticas de una expresión, [84](A.1, Def. 2))
Las variables críticas de una expresión e con respecto a $\mathcal{A}$, escrito $critVar_{\mathcal{A}}(e)$, se definen como:

$$\begin{aligned}
&critVar_{\mathcal{A}}(s) = \emptyset \\
&critVar_{\mathcal{A}}(e_1\ e_2) = critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2) \\
&critVar_{\mathcal{A}}(\lambda t.e) = (opaqueVar_{\mathcal{A}}(t) \cap fv(e)) \cup critVar_{\mathcal{A}}(e) \\
&critVar_{\mathcal{A}}(let_*\ t = e_1\ in\ e_2) = (opaqueVar_{\mathcal{A}}(t) \cap fv(e_2)) \cup\ critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)
\end{aligned}$$

Utilizando la definición anterior, la relación de tipado $\vdash^{\bullet}$ que evita los problemas de tipos generados por los patrones de orden superior queda definida por la única regla

[24] Esta explicación intuitiva se basa en la propiedad de cierre bajo sustituciones de tipos que posee la relación básica $\vdash$, como veremos más adelante.

(P) de la Figura 11. Esta regla expresa que $\mathcal{A} \vdash^{\bullet} e : \tau$ si se puede derivar el tipo τ para e mediante la relación de tipado básica ($\mathcal{A} \vdash e : \tau$) y además e no contiene ninguna variable crítica con respecto a $\mathcal{A}$.

Las dos relaciones de tipado presentadas ($\vdash$ y $\vdash^{\bullet}$) gozan de algunas propiedades clásicas de los sistemas de tipos. El siguiente teorema reúne algunas de ellas. Para referirnos a cualquiera de las relaciones $\vdash$ y $\vdash^{\bullet}$ utilizaremos $\vdash^{?}$:

Teorema 6 (Propiedades de las relaciones de tipado, [84](A.1, Th. 1))

a) Si $\mathcal{A} \vdash^{?} e : \tau$ entonces $\mathcal{A}\pi \vdash^{?} e : \tau\pi$, para cualquier π.

b) Sea $s \in DC \cup FS \cup \mathcal{DV}$ un símbolo que no aparece en e. Entonces $\mathcal{A} \vdash^{?} e : \tau \iff \mathcal{A} \oplus \{s : \sigma\} \vdash^{?} e : \tau$, para cualquier σ.

c) Si $\mathcal{A} \oplus \{X : \tau_x\} \vdash^{?} e : \tau$ y $\mathcal{A} \oplus \{X : \tau_x\} \vdash^{?} e' : \tau_x$ entonces $\mathcal{A} \oplus \{X : \tau_x\} \vdash^{?} e[X/e'] : \tau$.

d) Si $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ y $\sigma' \succ \sigma$ entonces $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$.

El apartado *a)* establece que las derivaciones de tipos son cerradas bajo sustituciones de tipos. El apartado *b)* muestra que las derivaciones de tipo para e solo dependen de las suposiciones de tipos sobre los símbolos que aparecen en e, por lo que añadir o quitar suposiciones sobre otros símbolos no las afecta. El apartado *c)* expresa que se pueden sustituir variables por expresiones que tengan el mismo tipo, y el resultado no cambia. Por último, el apartado *d)* establece que se pueden generalizar las suposiciones de tipos para algunos símbolos y derivar el mismo tipo para la expresión. Este último resultado sólo es válido para $\vdash$, ya que generalizar un tipo puede resultar en que una variable que antes era transparente se convierta en opaca y por tanto en crítica, invalidando una posible derivación $\vdash^{\bullet}$.

En PF, la relación de tipado sobre expresiones puede aplicarse a programas, ya que estos se pueden expresar como una cadena de let-expresiones definiendo las funciones acompañada de una expresión a evaluar. En nuestro marco esto no es posible, ya que nuestras let-expresiones no definen funciones sino que realizan encaje de patrones, además de que las λ-abstracciones no están soportadas por la semántica. Por ello es necesario definir explícitamente la noción de programa bien tipado con respecto a un conjunto de suposiciones. La siguiente definición establece cuándo una regla y programa están bien tipados. Para ello utiliza derivaciones de tipos $\vdash^{\bullet}$ sobre unas λ-abstracciones que asociamos a las reglas, como se ve en la definición siguiente, razón por la que este tipo de expresiones ha sido considerado en la sintaxis.

Definición 3 (Programa bien tipado, [84](A.1, Def. 3)) *Diremos que una regla de programa $f\ t_1 \ldots t_n \to e$ está bien tipada con respecto a $\mathcal{A}$ si $\mathcal{A} \vdash^{\bullet} \lambda t_1 \ldots \lambda t_n . e : \tau$ y τ es una variante de $\mathcal{A}(f)$. Por su parte, un programa $\mathcal{P}$ está bien tipado con respecto a $\mathcal{A}$, escrito $wt^{\bullet}_{\mathcal{A}}(\mathcal{P})$, si todas sus reglas está bien tipadas con respecto a $\mathcal{A}$.*

Al utilizar $\vdash^\bullet$ en la definición de regla bien tipada nos aseguramos que la regla no contiene variables críticas. Por otro lado, la condición de que el tipo derivado para la λ-abstracción asociada a la regla sea una variante del tipo de la función es imprescindible para garantizar la preservación de tipos. Si lo relajásemos a ser una instancia genérica los tipos no se preservarían durante la ejecución. Un ejemplo es el programa $\mathcal{P} \equiv \{not'\ true \to false, not'\ false \to true\}$ con las suposiciones $\mathcal{A} \equiv \{not' : \forall\alpha.bool \to \alpha\}$. Ambas reglas están bien tipadas ya que $bool \to bool$ es una instancia genérica de $\forall\alpha.bool \to \alpha$. Sin embargo, los tipos no se preservan: $add\ zero\ (not'\ true)$ está bien tipada (puesto que $not'\ true$ puede tener cualquier tipo, en particular nat), pero al reducirlo utilizando la regla de not' se obtiene la expresión $add\ zero\ false$, que está mal tipada.

Volviendo al Ejemplo 14 (página 50), las reglas de las funciones f, g y h, que son inválidas en [45] por utilizar patrones opacos, ahora son consideradas como bien tipadas. En el caso de f la variable Y es transparente en el patrón $(snd\ X, Y)$, por lo que puede ser utilizada en el lado derecho. La variable X sí que es opaca en el patrón $snd\ X$ de la función g, no obstante, la regla está bien tipada porque esta variable no aparece en el lado derecho, por lo que no es crítica[25]. La función h no contiene variables en el patrón opaco $snd\ true$, así que no puede existir ninguna variable crítica. Considerando el ejemplo del *casting* polimórfico (Ejemplo 1, página 4), el programa sería rechazado debido a la función $unpack$. Esta función tiene la variable opaca X en el patrón $snd\ X$, que se convierte en crítica al aparecer en el lado derecho. Esto hace que la regla de $unpack$ esté mal tipada, impidiendo por tanto definir la función $cast$.

La relación de tipado $\vdash^\bullet$ permite derivar tipos para expresiones, en cambio, no proporciona ningún método efectivo para inferir el tipo de una expresión al estilo del algoritmo $\mathcal{W}$ (Figura 8, página 34). Para salvar esa carencia proponemos los algoritmos de inferencia de tipos $\mathcal{A} \Vdash e : \tau|\pi$ y $\mathcal{A} \Vdash^\bullet e : \tau|\pi$ de la Figura 12, que infieren tipos válidos con respecto a $\vdash$ y $\vdash^\bullet$ respectivamente. Aunque son presentados como relaciones para mostrar su similaridad con $\vdash$ y $\vdash^\bullet$, en esencia son algoritmos que fallan cuando las reglas no se pueden aplicar. Los algoritmos $\Vdash$ y $\Vdash^\bullet$ aceptan un conjunto de suposiciones $\mathcal{A}$ y una expresión e, y devuelven un tipo simple τ y una sustitución de tipos π. De una manera intuitiva (que será formalizada más adelante) el tipo τ es el tipo más general que se puede derivar para e, mientras que π es la mínima sustitución que hace falta aplicar a $\mathcal{A}$ para poder derivar algún tipo para e. La idea utilizada en ambos algoritmos es la misma que en $\mathcal{W}$: introducir variantes de los tipos para los símbolos (que siempre serán más generales que sus instancias genéricas) y unificar [128, 98] aquellos tipos que deban ser iguales.

Aunque hemos presentado el algoritmo $\Vdash^\bullet$ como un método *efectivo* para calcular tipos, su definición en la Figura 12 utiliza el conjunto de variables críticas, que se basa

[25]Con la expresión de *variables críticas en una regla* nos referimos a la noción de variables críticas en la λ-abstracción asociada que se utiliza en la definición de regla bien tipada.

(iID) $\dfrac{}{\mathcal{A} \Vdash s : \tau | \epsilon}$ si $\mathcal{A}(s) \succ_{var} \tau$

(iAPP) $\dfrac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi | \pi_1\pi_2\pi}$ si α variable de tipos fresca y $\pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha)$

(iΛ) $\dfrac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t | \pi_t \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau | \pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau | \pi_t\pi}$ si $\{\overline{X_n}\} = var(t)$ y $\overline{\alpha_n}$ son variables de tipos frescas

(iLET$_m$) $\dfrac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash let_m\ t = e_1\ in\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$ si $\{\overline{X_n}\} = var(t), \overline{\alpha_n}$ son variables de tipo frescas y $\pi = mgu(\tau_t\pi_1, \tau_1)$

(iLET$^X_{pm}$) $\dfrac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash let_{pm}\ X = e_1\ in\ e_2 : \tau_2 | \pi_1\pi_2}$

(iLET$^h_{pm}$) $\dfrac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash h\ \overline{t_m} : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash let_{pm}\ h\ \overline{t_m} = e_1\ in\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$ si $\{\overline{X_n}\} = var(h\ \overline{t_m}), \overline{\alpha_n}$ son variables de tipo frescas y $\pi = mgu(\tau_t\pi_1, \tau_1)$

(iLET$_p$) $\dfrac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash let_p\ t = e_1\ in\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$
si $\{\overline{X_n}\} = var(t)$, $\overline{\alpha_n}$ son variables de tipos frescas y $\pi = mgu(\tau_t\pi_1, \tau_1)$

(iP) $\dfrac{\mathcal{A} \Vdash e : \tau | \pi}{\mathcal{A} \Vdash^{\bullet} e : \tau | \pi}$ si $critVar_{\mathcal{A}\pi}(e) = \emptyset$

Figura 12: Reglas de inferencia de tipos para $\Vdash$ y $\Vdash^{\bullet}$

en la definición de variables opacas. Estas están definidas en base a la existencia de derivaciones $\vdash$ que cumplan ciertas propiedades, por lo que no puede considerarse que la Definición 1 (página 53) proporcione un método efectivo para calcular las variables opacas. Sin embargo, gracias a los resultados de corrección y completitud de $\Vdash$ con respecto a $\vdash$ (que presentamos a continuación) es posible desarrollar una caracterización equivalente de las variables opacas basándose en el algoritmo $\Vdash$, consiguiendo

así un método efectivo de inferencia $\Vdash^\bullet$:

Proposición 1 ([84](A.1, Prop. 1 y Lemma 3)) *$X_i \in \overline{X_n} = var(t)$ es opaca con respecto a $\mathcal{A}$ si y solo si $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ y $ftv(\alpha_i\pi_g) \nsubseteq ftv(\tau_g)$.*

Un aspecto imprescindible acerca de los algoritmos de inferencia de tipos es que calculen tipos y sustituciones correctos con respecto al sistema de tipos. La corrección de $\Vdash$ y $\Vdash^\bullet$ queda capturada en el siguiente teorema, donde utilizamos $\Vdash^?$ para referirnos a cualquiera de los algoritmos $\Vdash$ o $\Vdash^\bullet$, de manera similar a $\vdash^?$:

Teorema 7 (Corrección de $\Vdash^?$, [84](A.1, Th. 4)) *Si $\mathcal{A} \Vdash^? e : \tau|\pi$ entonces $\mathcal{A}\pi \vdash^? e : \tau$*

Otra propiedad importante de los algoritmos de inferencia de tipos es que calculen tipos que sean, en cierta manera, lo más general posible para las expresiones. El siguiente teorema, que establece la completitud de $\Vdash$, expresa que si al aplicar alguna sustitución a $\mathcal{A}$ es posible derivar un tipo para e, entonces la inferencia tendrá éxito y encontrará un tipo y una sustitución más generales:

Teorema 8 (Completitud de $\Vdash$, [84](A.1, Th. 5)) *Si $\mathcal{A}\pi' \vdash e : \tau'$ entonces $\exists \tau, \pi, \pi''. \mathcal{A} \Vdash e : \tau|\pi$, $\mathcal{A}\pi\pi'' = \mathcal{A}\pi'$ y $\tau\pi'' = \tau'$.*

Un resultado similar al Teorema 8 es falso en general para $\Vdash^\bullet$, ya que en algunas situaciones no existe una sustitución más general que permita derivar tipos para una expresión. Esta situación se observa en el siguiente ejemplo:

Ejemplo 15 (Inexistencia de sustituciones más generales, [84](A.1, Ex. 4))
Consideremos el conjunto de suposiciones $\mathcal{A} \equiv \{snd' : \alpha \to bool \to bool\}$, que tiene la variable libre α. Con este conjunto $\mathcal{A}$ podemos construir las derivaciones válidas $\mathcal{A}[\alpha/bool] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to bool$ y $\mathcal{A}[\alpha/int] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to int$. La única sustitución más general que $[\alpha \mapsto bool]$ y $[\alpha \mapsto int]$ sería $[\alpha \mapsto \beta]$, sin embargo, no sería una sustitución correcta con respecto a $\vdash^\bullet$ ya que convierte a X en una variable crítica, impidiendo derivar cualquier tipo para $\lambda(snd'\ X).X$.

Aunque no siempre existan sustituciones más generales que permitan derivar tipos con respecto a $\vdash^\bullet$, siempre que esta exista el algoritmo $\Vdash^\bullet$ la encontrará, y viceversa. Para formalizar este resultado introduciremos la noción de *sustituciones tipantes* $\Pi^\bullet_{\mathcal{A},e}$ de una expresión e con respecto a $\mathcal{A}$:

Definición 4 (Sustituciones tipantes de e con respecto a $\mathcal{A}$, [84](A.1, Def. 4))
$\Pi^\bullet_{\mathcal{A},e} = \{\pi \mid \exists \tau.\ \mathcal{A}\pi \vdash^\bullet e : \tau\}$

Basándose en las sustituciones tipantes podemos enunciar el teorema de maximalidad de $\Vdash^\bullet$. Este teorema tiene dos partes. La primera expresa que si existe una

sustitución tipante más general (es decir, si $\Pi^{\bullet}_{\mathcal{A},e}$ tiene un elemento máximo), el algoritmo $\Vdash^{\bullet}$ la encuentra, y viceversa. Por otro lado, la segunda parte expresa que si al aplicar alguna sustitución a $\mathcal{A}$ es posible derivar un tipo para e y además la inferencia tiene éxito, entonces la sustitución y el tipo calculado por la inferencia serán los más generales:

Teorema 9 (Maximalidad de $\Vdash^{\bullet}$, [84](A.1, Th. 6))

a) $\Pi^{\bullet}_{\mathcal{A},e}$ *tiene un elemento máximo* $\Longleftrightarrow \exists \tau_g, \pi_g.\ \mathcal{A} \Vdash^{\bullet} e : \tau_g|\pi_g$.

b) *Si* $\mathcal{A}\pi' \vdash^{\bullet} e : \tau'$ *y* $\mathcal{A} \Vdash^{\bullet} e : \tau|\pi$ *entonces existe* π'' *tal que* $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ *y* $\tau' = \tau\pi''$.

Aunque no tengamos completitud de $\Vdash^{\bullet}$ en el caso general, se puede observar fácilmente que para $\mathcal{A}$ cerrados (es decir, $ftv(\mathcal{A}) = \emptyset$) sí que se tiene. Esto se desprende del hecho de que para este tipo de conjuntos de suposiciones $\mathcal{A} = \mathcal{A}\pi$ para toda sustitución π, por lo que $critVar_{\mathcal{A}}(e) = critVar_{\mathcal{A}\pi}(e)$. En este caso, la existencia de una derivación $\mathcal{A}\pi' \vdash^{\bullet} e : \tau'$ implica que $critVar_{\mathcal{A}\pi'}(e) = \emptyset = critVar_{\mathcal{A}}(e)$. Por ello la sustitución π calculada por $\Vdash$ (ver Teorema 8) cumplirá $critVar_{\mathcal{A}\pi}(e) = \emptyset$, de lo que se deduce que la inferencia $\mathcal{A} \Vdash^{\bullet} e : \tau|\pi$ existirá.

Corolario 1 (Completitud de $\Vdash^{\bullet}$) *Si* $ftv(\mathcal{A}) = \emptyset$ *y* $\mathcal{A}\pi' \vdash^{\bullet} e : \tau'$ *entonces* $\exists \tau, \pi, \pi''.\ \mathcal{A} \Vdash^{\bullet} e : \tau|\pi$, $\mathcal{A}\pi\pi'' = \mathcal{A}\pi'$ *y* $\tau\pi'' = \tau'$.

De la misma manera que es necesario proporcionar una definición de programa bien tipado, es necesario desarrollar un método de inferencia de tipos para programas, ya que la inferencia $\Vdash^{\bullet}$ únicamente es aplicable a expresiones. Para ello proponemos el método $\mathcal{B}$ de inferencia que acepta un conjunto de suposiciones $\mathcal{A}$ y un programa $\mathcal{P}$, devolviendo una sustitución π tal que el $\mathcal{P}$ está bien tipado con respecto a $\mathcal{A}\pi$. El conjunto $\mathcal{A}$ debe contener suposiciones para todos los símbolos del programa, incluidas las funciones. Esta suposiciones podrán ser esquemas de tipos cerrados, provenientes de declaraciones explícitas en el programa, o bien podrán ser variables frescas. En el primer caso los tipos proporcionados serán utilizados y comprobados por el método, mientras que en el segundo caso se inferirá el tipo de las funciones instanciando dichas variables libres mediante la sustitución π. Nótese además que de esta manera soportamos la *recursión polimórfica* [108, 76, 57] (la utilización de llamadas recursivas con tipos más concretos que el de la función), aunque solo para funciones cuyo tipo ha sido declarado explícitamente, ya que es conocido que en presencia de esta la inferencia de tipos es indecidible.

Definición 5 (Inferencia de tipos de un programa, [84](A.1, Def. 5))

$$\mathcal{B}(\mathcal{A}, \{rule_1, \ldots, rule_m\}) = \pi, \text{ si}$$

1. $\mathcal{A} \Vdash^{\bullet} (\varphi(rule_1), \ldots, \varphi(rule_m)) : (\tau_1, \ldots, \tau_m)|\pi$.
2. *Sean* $f^1 \ldots f^k$ *los símbolos de funciones de las reglas* $rule_i$ *tales que* $\mathcal{A}(f^i)$ *es un esquema de tipos cerrado, y sea* τ^i *el tipo obtenido para la regla* $rule_i$ *en el paso* 1*. Entonces* τ^i *debe ser variante de* $\mathcal{A}(f^i)$.

Consideraremos que $()$ *es el constructor usual de tuplas sobrecargado para cualquier aridad* m*—*$() : \forall\overline{\alpha_m}.\alpha_1 \to \ldots \alpha_m \to (\alpha_1, \ldots, \alpha_m)$*—,* φ *es una transformación de reglas a expresiones definida como*

$$\varphi(f\ t_1 \ldots t_n \to e) = pair\ \lambda t_1. \ldots \lambda t_n.e\ f$$

y $pair$ *es un constructor de parejas de elementos del mismo tipo (*pair $: \forall\alpha.\alpha \to \alpha \to \alpha$*).*

Como se puede observar, la inferencia para programas se basa en la inferencia mediante $\Vdash^{\bullet}$ de una tupla que contiene las λ-abstracciones asociadas a las reglas. La constructora $pair$ es importante ya que sirve para unificar los tipos inferidos para las reglas de la misma función. Basándose en los resultados de corrección y maximalidad de $\Vdash^{\bullet}$ con respecto a $\vdash^{\bullet}$, es sencillo obtener la corrección y maximalidad de $\mathcal{B}$:

Teorema 10 (Corrección de $\mathcal{B}$, [84](A.1, Th. 7)) *Si* $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ *entonces* $wt^{\bullet}_{\mathcal{A}\pi}(\mathcal{P})$.

Teorema 11 (Maximalidad de $\mathcal{B}$, [84](A.1, Th. 8)) *Si* $wt^{\bullet}_{\mathcal{A}\pi'}(\mathcal{P})$ *y* $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ *entonces* $\exists\pi''$ *tal que* $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.

Los tipos inferidos por $\mathcal{B}$ son tipos simples, por lo que para obtener esquemas de tipos sería necesario un paso de generalización final. Esto se debería combinar con un procesamiento por bloques de funciones mutuamente recursivas (componentes fuertemente conexas en el grafo de dependencias) para implementar la etapa de inferencia de tipos en un compilador. Se puede encontrar más información sobre este enfoque de inferencia de tipos estratificada para programas en [84](A.1, §5.1).

6.3. Preservación de tipos

La *preservación de tipos* (también conocida como *subject reduction*) es la propiedad más importante sobre la corrección de los sistemas de tipos [154] con respecto a semánticas operacionales, ya que expresa que las expresiones conservan su tipo tras cada paso de evaluación. Otra propiedad relevante con respecto a la corrección de los sistemas de tipos con respecto a estas semánticas es el *progreso* [120], que expresa que una expresión bien tipada es un valor o puede realizarse un paso de evaluación en ella. Aunque esta última propiedad es clásica en PF, no estamos seguros de cómo encaja en el marco de PLF donde se realizan búsquedas en un espacio posiblemente indeterminista. En PLF llegar a una expresión bien tipada que ni es un valor ni se puede reducir no debe verse como un error en sí, sino como un fallo en la rama de cómputo

$$
\begin{aligned}
\Psi(s) &= s \\
\Psi(e_1\ e_2) &= \Psi(e_1)\ \Psi(e_2) \\
\Psi(let_K\ X = e_1\ in\ e_2) &= let_K\ X = \Psi(e_1)\ in\ \Psi(e_2), \quad \text{con } K \in \{m, p\} \\
\Psi(let_{pm}\ X = e_1\ in\ e_2) &= let_p\ X = \Psi(e_1)\ in\ \Psi(e_2) \\
\Psi(let_m\ t = e_1\ in\ e_2) &= let_m\ Y = \Psi(e_1)\ in\ \overline{let_m\ X_n = f_{X_n}\ Y}\ in\ \Psi(e_2) \\
\Psi(let_{pm}\ t = e_1\ in\ e_2) &= let_m\ Y = \Psi(e_1)\ in\ \overline{let_m\ X_n = f_{X_n}\ Y}\ in\ \Psi(e_2) \\
\Psi(let_p\ t = e_1\ in\ e_2) &= let_p\ Y = \Psi(e_1)\ in\ \overline{let_p\ X_n = f_{X_n}\ Y}\ in\ \Psi(e_2)
\end{aligned}
$$

para $\{\overline{X_n}\} = var(t) \cap fv(e_2)$, $f_{X_i} \in FS^1$ funciones frescas definidas por las reglas $f_{X_i}\ t \to X_i$, $Y \in \mathcal{DV}$ variable fresca, y t un patrón compuesto.

Figura 13: Eliminación de patrones compuestos

actual que indica que debe llevarse a cabo el mecanismo de vuelta atrás y probar otras reglas para las funciones indeterministas. Por ello en esta tesis nos centraremos principalmente en la preservación de tipos como resultado de corrección de tipos (aunque en la siguiente sección consideraremos el progreso para el sistema de tipos liberal).

Para enunciar la preservación de tipos es necesaria una semántica que defina los pasos de cómputo posibles. Como ya hemos introducido, la semántica elegida es la let-reescritura (Sección 4.2, página 22) debido a que proporciona una noción básica de paso de cómputos con funciones indeterministas respetando *call-time choice*. Sin embargo, como se ve en la Figura 5 (página 24) las reglas de la let-reescritura consideran únicamente let-expresiones con variables, por lo que hace falta alguna extensión para adecuarlo a la sintaxis considerada por el sistema $\vdash^{\bullet}$. En lugar de extender las reglas originales de la let-reescritura para dar soporte a let-expresiones con patrones, hemos seguido un enfoque mixto: proporcionar una traducción de programas y expresiones para eliminar todos los patrones compuestos de las let-expresiones y extender las reglas de la let-reescritura para manejar adecuadamente las anotaciones de polimorfismo en las let-expresiones, que ya solo contendrán variables. Obsérvese además que las λ-abstracciones se utilizan únicamente en la definición de regla bien tipada, y no aparecen en los cómputos ni en las reglas de programa. Por tanto no es necesario extender la let-reescritura para que trate estas expresiones, para las que tampoco existe un consenso general sobre su significado semántico en la comunidad lógico-funcional.

Existen varias transformaciones posibles para eliminar patrones compuestos en las let-expresiones, que difieren en su estrictez. Nosotros hemos elegido la transformación Ψ[26] de la Figura 13, que está inspirada en [114] y no demanda el encaje del patrón si ninguna variable es usada, pero demanda el encaje del patrón entero si alguna variable es usada —como ocurre en Haskell—. Las let-expresiones resultantes de la transformación Ψ no contendrán let_{pm}, ya que este tipo de let-expresión es similar a let_p para variables. Intuitivamente esta transformación genera una ligadura $Y = e_1$

[26] En [84](A.1) utilizamos *TRL* en lugar de Ψ para referirnos a esta transformación.

y una cadena de ligaduras $X_i = f_{X_i}\ Y$ que extraen las diferentes componentes del patrón compuesto. Esto se observa claramente en el siguiente ejemplo:

Ejemplo 16 (Eliminación de patrones compuestos) *Consideremos la expresión* $e \equiv let_{pm}\ [F, G] = [id, id]\ in\ (F\ true, G\ false)$*. El resultado de eliminar los patrones compuestos sería* $\Psi(e) = let_m\ Y = [id, id]\ in\ let_m\ F = f_F\ Y\ in\ let_m\ G = f_G\ Y\ in\ (F\ true, G\ false)$*, donde las funciones extractoras estarían definidas mediante las reglas* $f_F\ [F, G] \to F$ *y* $f_G\ [F, G] \to G$*.*

La transformación que elimina patrones compuestos preserva el tipo de la expresión, tal y como queda patente en el siguiente teorema:

Teorema 12 (Preservación de tipos de Ψ, [84](A.1, Th. 2)) *Consideremos* $\mathcal{A} \vdash^\bullet e : \tau$ *y que* $\mathcal{P} \equiv \{\overline{f_{X_n}\ t_n \to X_n}\}$ *son las reglas de las funciones extractoras necesarias en la transformación de* $\Psi(e)$*. Consideremos también que* $\mathcal{A}'$ *es el conjunto de suposiciones sobre esas funciones, definido como* $\mathcal{A}' \equiv \{\overline{f_{X_n} : Gen(\tau_{X_n}, \mathcal{A})}\}$*, donde* $\mathcal{A} \Vdash^\bullet \lambda t_i.X_i : \tau_{X_i}|\pi_{X_i}$*. Entonces* $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet \Psi(e) : \tau$ *y* $wt^\bullet_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$*.*

El anterior teorema también establece que las funciones extractoras generadas estarán bien tipadas[27]. Por tanto, partiendo de un programa $\mathcal{P}$ y una expresión bien tipados, el programa resultante $\mathcal{P} \uplus \mathcal{P}'$ resultante de añadir las funciones extractoras al programa original estará bien tipado con respecto a $\mathcal{A} \oplus \mathcal{A}'$.

En relación a las reglas de la let-reescritura (Figura 5, página 24), es necesario extenderlas para que traten las let-expresiones let_m y let_p de manera segura desde el punto de vista de los tipos. El resultado puede verse en la Figura 14. Las reglas son muy similares a las originales, siendo la mayor diferencia la división de la regla (Flat) original en dos: (Flat$_m$) y (Flat$_p$). Aunque ambas se comportan igual desde el punto de vista de los valores, la división es necesaria para preservar tipos. Debido a que en este sistema no se consideran programas con variables extra, también se ha simplificado la regla (Contx) eliminando las condiciones que evitaban la captura de variables, pues esta no aparecerá.

Una vez definida la eliminación de patrones compuestos y la let-reescritura con soporte para let_m y let_p, podemos enunciar la preservación de tipos bajo pasos de evaluación:

Teorema 13 (Preservación de tipos, [84](A.1, Th. 3)) *Si* $\mathcal{A} \vdash^\bullet e : \tau$*,* $wt^\bullet_{\mathcal{A}}(\mathcal{P})$ *y* $\mathcal{P} \vdash e \to^{lp} e'$ *entonces* $\mathcal{A} \vdash^\bullet e' : \tau$*.*

Nótese que en el anterior teorema la expresión e a evaluar no puede contener λ-abstracciones (no soportadas por la let-reescritura), let-expresiones con patrones

[27] Debido a la derivación $\mathcal{A} \vdash^\bullet e : \tau$ las variables X_i extraídas serán transparentes en sus respectivos patrones t_i, así que la inferencia $\mathcal{A} \Vdash^\bullet \lambda t_i.X_i : \tau_{X_i}|\pi_{X_i}$ tendrá éxito (Teorema 8).

(Fapp) $f\ t_1\theta \ldots t_n\theta \rightarrow^{lp} r\theta$, si $(f\ t_1 \ldots t_n \rightarrow r) \in \mathcal{P}$

(LetIn) $e_1\ e_2 \rightarrow^{lp} let_m\ X = e_2\ in\ e_1\ X$, si e_2 es una expresión *junk*, activa, una aplicación de variable o una let-expresión; para X fresca

(Bind) $let_K\ X = t\ in\ e \rightarrow^{lp} e[X/t]$

(Elim) $let_K\ X = e_1\ in\ e_2 \rightarrow^{lp} e_2$, si $X \notin fv(e_2)$

(Flat$_m$) $let_m\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^{lp} let_K\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3)$, si $Y \notin fv(e_3)$

(Flat$_p$) $let_p\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^{lp} let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$, si $Y \notin fv(e_3)$

(LetAp) $(let_K\ X = e_1\ in\ e_2)\ e_3 \rightarrow^{lp} let_K\ X = e_1\ in\ e_2\ e_3$, si $X \notin fv(e_3)$

(Contx) $\mathcal{C}[e] \rightarrow^{lp} \mathcal{C}[e']$, si $\mathcal{C} \neq [\ ]$, $e \rightarrow^{lp} e'$ usando alguna de las reglas anteriores

donde $K \in \{m, p\}$

Figura 14: Let-reescritura $\rightarrow^{lp}$ con manejo de let_m y let_p

compuestos ni let_{pm}-expresiones de ningún tipo (habrán sido eliminados mediante la transformación Ψ). El Teorema 13 enuncia la preservación de tipos para un paso de let-reescritura, pero su extensión a cualquier número de pasos es trivial.

La ausencia de variables críticas en las reglas —garantizada por $wt^{\bullet}_{\mathcal{A}}(\mathcal{P})$— es imprescindible a la hora de conseguir preservación de tipos durante la aplicación de funciones, ya que garantiza que todas las variables que aparecen en los lados derechos serán variables transparentes de los patrones de los lados izquierdos. Las variables transparentes de los patrones tienen una importante propiedad con respecto a su instanciación, que explota la relación entre el tipo de estas variables y el tipo del patrón contenedor:

Lema 1 ([84](A.1, Lemma 1)) *Consideremos* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$, *donde* $var(t) \subseteq \{\overline{X_n}\}$. *Si* $\mathcal{A} \vdash t[\overline{X_n/s_n}] : \tau$ *y* X_i *es una variable transparente de* t *con respecto a* $\mathcal{A}$ *entonces* $\mathcal{A} \vdash s_i : \tau_i$.

El anterior lema expresa que si tenemos un patrón t con tipo τ y sustituimos sus variables por patrones, obteniendo un patrón sustituido $t[\overline{X_n/s_n}]$ con el mismo tipo τ, las variables transparentes habrán sido sustituidas por patrones de su mismo tipo. Esto no ocurre con las variable opacas, y por eso impedimos su aparición en los lados derechos de las reglas de programa.

6.4. Conclusiones

El sistema $\vdash^\bullet$ proporciona un tratamiento seguro de los patrones de orden superior en los lados izquierdos de las reglas, evitando problemas como el *casting* polimórfico del Ejemplo 1 (página 4). Para ello sigue un enfoque más relajado que en [45], donde se prohíben todos los patrones opacos en las reglas. El sistema $\vdash^\bullet$ clasifica las variables de los patrones en opacas, si su tipo no está unívocamente determinado por el tipo del patrón, o transparentes en caso contrario. Para garantizar la preservación de tipos, el sistema $\vdash^\bullet$ impide la aparición de variables opacas en los lados derechos de las reglas y en los cuerpos de las let-expresiones, las llamadas variables críticas. Con respecto a [45], el sistema $\vdash^\bullet$ presenta otras mejoras además del tratamiento más relajado de los patrones opacos. Por ejemplo, permite constructoras no transparentes, y constructoras cuyo tipo contenga funciones y no solo tipos de datos. También da soporte a distintos tipos de let-expresiones y a λ-abstracciones, aunque estas últimas no pueden aparecer en los programas. La preservación de tipos está demostrada utilizando let-reescritura como mecanismo operacional, una noción más cercana a los cómputos reales que la semántica de CRWL utilizada en [45]. Por último, proporciona un método de inferencia de tipos para programas, a diferencia de [45] donde se supone que los programas vienen acompañados de declaraciones de tipos explícitas. Este método de inferencia para programas ha sido implementado e integrado en una rama del sistema Toy[28].

Aunque el sistema $\vdash^\bullet$ soporta constructoras no transparentes —*constructoras existenciales* según la caracterización presentada en la Sección 5.3.1 (página 40)—, hay que insistir en que se trata de un sistema incomparable con respecto a los sistemas de tipos existenciales para PF [112, 79] en cuanto a los programas aceptados. Tomemos por caso el ejemplo clásico de constructoras existenciales del Ejemplo 11 (página 40), que sería rechazado por $\vdash^\bullet$ debido a que tanto X como F son variables opacas en $mkKey\ X\ F$ y por tanto variables críticas al aparecer en el lado derecho. Sin embargo, existen programas válidos para $\vdash^\bullet$ que son rechazados con el sistema de [112, 79]:

Ejemplo 17 *Consideremos la constructora existencial de* contenedor opaco $CS^1 \ni cont :: \forall\alpha.\alpha \to cont$. *Utilizando esta constructora podemos definir una función* $code$ *para codificar en listas de bits algunos valores encapsulados en contenedores opacos:*

```
data bit = i | o                code :: cont -> [bit]
data cont where                 code (cont true) = [o,o,o]
  cont :: A -> cont             code (cont zero) = [o,i,o]
```

La función $code$, aunque no genera ningún problema de tipos, sería rechazada por [112, 79] ya que la constante de Skolem proveniente del tipo de $cont$ sería incomparable con los tipos $bool$ y nat en la primera y segunda regla respectivamente. Las constructoras existenciales persiguen la ocultación de información, por lo que el rechazo de $code$

[28] Disponible en `http://gpd.sip.ucm.es/Toy2SafeOpaquePatterns`.

responde a esta filosofía: las reglas *inspeccionan* el contenido del contenedor opaco. En cambio, la función $code$ sería válida en $\vdash^\bullet$ debido a que no hay ninguna variable crítica (de hecho no hay ninguna variable). Obsérvese la diferencia entre las constructoras existenciales, que persiguen de manera expresa la ocultación de información y por ello son declaradas con tipos existenciales, y los patrones de orden superior, que representan de manera intensional funciones y cuya opacidad es sobrevenida. El sistema $\vdash^\bullet$ trata a ambos elementos de manera homogénea, desechando la ocultación de información pero conservando la preservación de tipos. Nótese además que la inspección que permite el sistema $\vdash^\bullet$ sobre los argumentos opacos viola la parametricidad (Sección 5.3.4, página 46), propiedad que es conservada en los sistemas de tipos existenciales de PF (Sección 5.3.1, página 40). Sin embargo, esto no un grave problema en nuestro marco ya que, como hemos visto en la Sección 5.3.4, los teoremas gratis están seriamente comprometidos en PLF debido al indeterminismo, el estrechamiento y las variables extra. A diferencia de la PF, la pérdida de parametricidad tampoco tiene un gran impacto en la representación de las constructoras ya que la mayoría de los sistemas de PLF realizan una traducción a Prolog, por lo que las constructoras están representadas internamente por átomos diferentes.

Aunque el sistema $\vdash^\bullet$ mejora el manejo de los patrones de orden superior, no resuelve todos los problemas generados por estos. En particular, $\vdash^\bullet$ no preserva tipos en presencia de descomposición opaca (Ejemplo 1, página 4), al igual que [45]. Esto es así porque $\vdash^\bullet$ únicamente maneja con seguridad los patrones de orden superior cuando aparecen en los *lados izquierdos de las reglas*, pero la descomposición opaca se produce en presencia de igualdad estructural, que no está definida mediante reglas (pues estaría mal tipada) sino que es una primitiva *ad-hoc* de los sistemas. Por tanto, la reducción problemática del Ejemplo 1 seguiría ocurriendo si se considera la igualdad estructural, que no es contemplada en la let-reescritura de la Figura 14. Además, el sistema $\vdash^\bullet$ solo tiene en cuenta la let-reescritura, dejando fuera todo cómputo que involucre ligadura de variables libres e incluso reglas con variables extra, aspectos muy importantes dentro de la PLF. En las siguientes secciones presentaremos sistemas que dan cabida a estos aspectos.

7. Sistema de tipos liberal

Este capítulo presenta el sistema de tipos liberal aparecido en los artículos *A Liberal Type System for Functional Logic Programs* [87](A.2) y *Liberal Typing for Functional Logic Programs* [83](A.3). Como el artículo [87] es una versión extendida y revisada de [83] que además incluye las demostraciones completas, todas las referencias a los artículos serán realizadas sobre [87].

7.1. Motivación y objetivos

En la anterior sección hemos presentado el sistema $\vdash^{\bullet}$ para manejar los patrones de orden superior en los lados izquierdos de las reglas de manera segura con respecto a los tipos. Este sistema, además, presenta unas posibilidades limitadas para la programación genérica que se pueden observan en el Ejemplo 17 (página 64). La función $code$ acepta, de manera encapsulada, valores de distintos tipos y los inspecciona, devolviendo una lista de bits. Una función similar que no utilizase los contenedores opacos sería rechazada tanto por $\vdash^{\bullet}$ como por DM, pues las reglas tendrían tipos incomparables: $bool \to [bit]$ y $nat \to [bit]$ respectivamente. No obstante, el hecho de que $\vdash^{\bullet}$ acepte $code$ nos indica de alguna manera que esta función de codificación preserva los tipos, aun inspeccionando elementos de tipos diversos. La generalidad proporcionada por $\vdash^{\bullet}$ y su inspección de argumentos encapsulados es bastante limitada ya que se reduce a valores que no contengan variables. Si los valores encapsulados contuviesen variables estas no podrían utilizarse en el lado derecho pues se convertirían en variables críticas. Esto impediría reglas como $code\ (cont\ (succ\ X)) \to [o, i, i]{+}{+}(code\ X)$, ya que la variable X sería crítica.

Un ejemplo similar a $code$ es una función que calcule el tamaño (número de símbolos de constructora) de su argumento:

Ejemplo 18 (Contar el número de constructoras, ([87](A.2), §1))

```
size true     = succ zero
size false    = succ zero
size zero     = succ zero
size (succ X) = succ (size X)
size []       = succ zero
size (X:Xs)   = succ (add (size X) (size Xs))
```

Esta función sería rechazada por $\vdash^{\bullet}$ y DM ya que los tipos de sus reglas serían incompatibles: $bool \to nat$, $nat \to nat$ y $[\alpha] \to nat$. Sin embargo, $size$ no produciría ningún problema desde el punto de vista de los tipos. Para definir esta función podríamos utilizar algunas de las extensiones de tipos presentadas en la Sección 5.3 (página 40). Por ejemplo, podríamos definir una clase de tipos $sizeable$ conteniendo la función $size$, y sobrecargar la función para los distintos tipos. También sería posible utilizar GADTs para representar los tipos, y definir una función indexada por tipo que aceptase como primer argumento esa representación. En cambio, nuestro objetivo en esta sección es desarrollar un sistema de tipos liberal para PLF que acepte funciones como $size$ simplemente añadiendo la declaración de tipos $size :: \forall\alpha.\alpha \to nat$. Esto significa que desde el origen decidimos perder la parametricidad del sistema de tipos, pues la función $size$ inspecciona su argumento polimórfico. Aunque así se elimina la posibilidad de obtener *teoremas gratis* [148] a partir de los tipos, como ya hemos visto en la Sección 5.3.4

(página 46) esta posibilidad ya está seriamente comprometida en PLF debido al indeterminismo, las variables extra y el estrechamiento [29].

El objetivo del sistema liberal de este capítulo es proporcionar un sistema de tipos adecuado para PLF en el marco de la reescritura indeterminista con *call-time choice* (utilizando la semántica de la let-reescritura) que soporte patrones de orden superior. El punto más importante es que este sistema de tipos debe ser lo más liberal posible (aceptar la mayor cantidad posible de programas) siempre y cuando se garantice la preservación de tipos. Esta liberalidad produce una gran diferencia con respecto al sistema de tipos DM y derivados (como $\vdash^{\bullet}$, el sistema de [45] y de los demás sistemas de PLF/PF) en lo relativo a programas bien tipados. Por ello, en ocasiones, los programas bien tipados por el sistema liberal pueden ser contrarios a la intuición de un programador acostumbrado a los sistemas de tipos habituales en PF, aunque como veremos la preservación de tipos está garantizada.

7.2. Sistema de tipos

Para el sistema de tipos liberal consideraremos la sintaxis de expresiones utilizada en la let-reescritura (ver Figura 4, página 23). A diferencia de $\vdash^{\bullet}$, y por concisión, eliminaremos las diferentes clases de let-expresiones, dejando solo una con comportamiento polimórfico. Por la misma razón tampoco consideraremos let-expresiones con patrones compuestos, basándonos en la transformación Ψ de eliminación de patrones compuestos presentada en la Figura 13 (página 61) para el sistema $\vdash^{\bullet}$. No consideraremos tampoco λ-abstracciones (no soportadas por la semántica) ya que, a diferencia de $\vdash^{\bullet}$, el sistema de tipos liberal no las utiliza en la definición de regla bien tipada. Por ello, la sintaxis de las expresiones queda como:

$$Exp \ni e ::= X \mid c \mid f \mid e\ e \mid let\ X = e\ in\ e$$

Para el sistema de tipos liberal abordaremos, además de la preservación de tipos, la propiedad de progreso y un enfoque de corrección sintáctica similar al de [154]. Por ello, consideraremos una constructora especial $fail \in CS^0$ para representar los fallos en el encaje de patrones, de manera similar a lo propuesto para GADTs en [28, 117]. De la misma manera que en $\vdash^{\bullet}$ los programas serán conjuntos de reglas $f\ \overline{t_n} \to e$ con lados izquierdos lineales y sin variables extra. Esta restricción sigue siendo necesaria para garantizar la preservación de tipos, ya que las variables extra son instanciadas de manera libre al aplicar las reglas, lo que produce fácilmente errores de tipos. Como $fail$ es un artificio ideado para poder enunciar las propiedades de progreso, supondremos que no aparece en las reglas ni en las expresiones a evaluar, sino que es generado por las reglas de la let-reescritura (como veremos más adelante). Con respecto a los conjuntos de suposiciones $\mathcal{A}$, todos deben contener la suposición $\{fail : \forall\alpha.\alpha\}$ y cumplir que para todo símbolo de constructora $c \in CS^n \setminus \{fail\}$, $\mathcal{A}(c) = \forall\overline{\alpha}.\tau_1 \to \ldots \to \tau_n \to C\ (\tau'_1\ \ldots\ \tau'_m)$

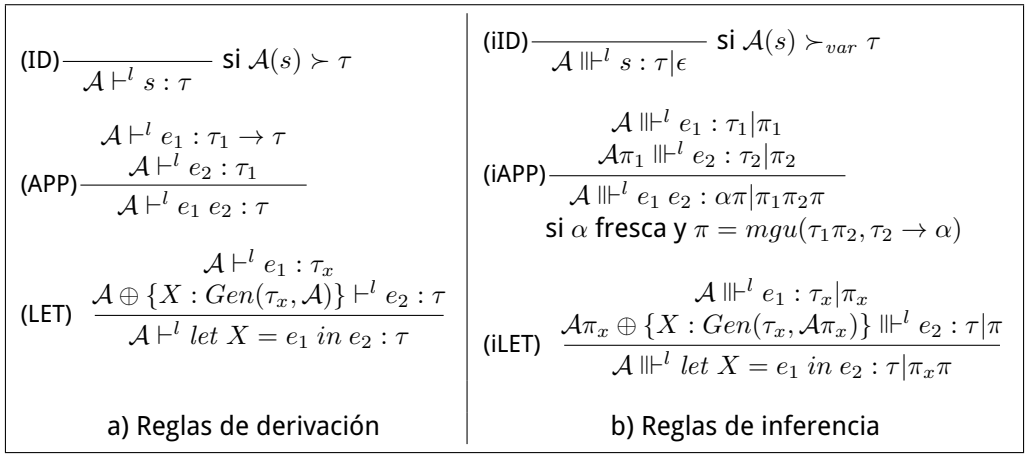


Figura 15: Sistema de tipos liberal para expresiones

para algún constructor de tipos C tal que $ar(C) = m$. Estas restricciones imponen que $fail$ puede tener cualquier tipo, y que las suposiciones de los demás símbolos de constructora corresponden con su aridad. Ambas restricciones, consideradas también en [28, 117], son necesarias para demostrar la corrección del sistema de tipos.

La derivación de tipos para expresiones $\vdash^l$ utiliza las mismas reglas que DM dirigido por la sintaxis (Figura 7-b, página 33) pero sin contar con una regla para λ-abstracciones pues estas no son soportadas. De manera similar, el algoritmo de inferencia de tipos $\Vdash^l$ considerado para expresiones es el algoritmo clásico $\mathcal{W}$ (Figura 8, página 34) eliminando la regla para λ-abstracciones. La Figura 15 contiene la relación de derivación y el algoritmo de inferencia de tipos considerados para expresiones. Si es posible derivar algún tipo para una expresión e (es decir, si existe τ tal que $\mathcal{A} \vdash^l e : \tau$) diremos que e es una *expresión bien tipada* con respecto a $\mathcal{A}$, escrito como $wt^l_{\mathcal{A}}(e)$.

Donde reside toda la liberalidad de nuestro sistema de tipos es en la noción de regla bien tipada. A diferencia del sistema $\vdash^\bullet$ de la sección anterior (y del sistema para manejar estrechamiento y variables extra que presentaremos en la próxima sección) no nos basamos en la derivación de tipos para la λ-abstracción asociada, sino que proponemos una definición directa. La intuición detrás de la definición de regla bien tipada es que *el lado derecho no restrinja el tipo de las variables más que el lado izquierdo*. Garantizando esta condición se consigue la preservación de tipos, a la vez que se consigue una noción muy general (de hecho lo más general posible, como veremos más adelante) que acepta como válidas reglas que son rechazadas por los demás sistemas de tipos de PLF y PF.

Definición 6 (Programa bien tipado, [87](A.2, Def. 3.1)) *Diremos que la regla de programa $f\ t_1 \ldots t_m \to e$ está* bien tipada *con respecto a un conjunto de suposiciones $\mathcal{A}$, escrito $wt^l_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$, si y solo si existen π_L, τ_L, π_R y τ_R tales que:*

i) π_L *es la sustitución más general tal que* $wt^l_{(\mathcal{A}\oplus\{\overline{X_n:\alpha_n}\})\pi_L}(f\ t_1 \ldots t_m)$*, y* τ_L *es el tipo más general que se puede derivar para* $f\ t_1 \ldots t_m$ *usando las suposiciones* $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_L$*.*

ii) π_R *es la sustitución más general tal que* $wt^l_{(\mathcal{A}\oplus\{\overline{X_n:\beta_n}\})\pi_R}(e)$*, y* τ_R *es el tipo más general que se puede derivar para* e *usando las suposiciones* $(\mathcal{A} \oplus \{\overline{X_n : \beta_n}\})\pi_R$*.*

iii) $\exists\pi.(\tau_L, \overline{\alpha_n\pi_L}) = (\tau_R, \overline{\beta_n\pi_R})\pi$*.*

iv) $\mathcal{A}\pi_L = \mathcal{A}$*,* $\mathcal{A}\pi_R = \mathcal{A}$*,* $\mathcal{A}\pi = \mathcal{A}$*.*

donde $\{\overline{X_n}\} = var(f\ t_1 \ldots t_m)$ *y* $\{\overline{\alpha_n}\}, \{\overline{\beta_n}\}$ *son variables frescas. Diremos que un programa* $\mathcal{P}$ *está* bien tipado *con respecto a un conjunto de suposiciones* $\mathcal{A}$*, escrito* $wt^l_{\mathcal{A}}(\mathcal{P})$*, si y solo si todas sus reglas están bien tipadas.*

Los primeros dos puntos comprueban que tanto el lado izquierdo como el derecho están bien tipados asignando algunos tipos para las variables. También obtiene los tipos más generales para esas variables en ambos lados, pero sin imponer ninguna relación entre ellos. Esto lo realiza el punto *iii)*, que comprueba que los tipos más generales obtenidos para el lado derecho y sus variables son más generales que los obtenidos para el lado izquierdo y sus variables. De esta manera se garantiza que el lado derecho no restringirá el tipo de las variables más que el lado izquierdo, garantizando que la aplicación de la regla preservará los tipos. Por último, el punto *iv)* es necesario para garantizar que las variables libres del conjunto de suposiciones no son modificadas por ninguna de las sustituciones consideradas en los anteriores puntos.

A diferencia del sistema $\vdash^\bullet$, donde proporcionamos un método $\mathcal{B}$ de inferencia de tipos para programas, en el sistema liberal no podemos desarrollar un método así. La razón principal es que, de manera similar a lo que ocurre con GADTs [28, 119, 134], no todas las reglas tienen un tipo más general. Consideremos por ejemplo la regla $f\ true \to false$. Esta regla estaría bien tipada con las suposiciones $f : \forall\alpha.\alpha \to \alpha$, $f : \forall\alpha.\alpha \to bool$ y $f : \forall\alpha.bool \to bool$, sin embargo, ninguna de las tres es más general que las demás. Como solución a este inconveniente adoptamos una solución similar a la utilizada por los GADTs: requerir las declaraciones de tipos para las funciones. A diferencia de estos, que únicamente necesitan las declaraciones de tipos para aquellas funciones que utilizan GADTs en sus reglas, en nuestro caso dichas declaraciones serán requeridas para todas las funciones. Esto es así ya que en nuestro sistema de tipos cualquier función puede tener un comportamiento liberal similar al que se consigue con GADTs. A pesar de carecer de inferencia de los tipos para programas al estilo de $\mathcal{B}$, sí es posible desarrollar un método efectivo para comprobar que un programa con declaraciones de tipos para todas sus funciones está bien tipado, gracias a la corrección y completitud del algoritmo de inferencia $\Vdash^l$ con respecto a $\vdash^l$ (Teorema 7 de la página 58 y Teorema 8 de la página 58, respectivamente). Los dos primeros puntos de la Definición 6 quedarían reducidos a inferir el tipo del lado izquierdo y derecho respectivamente. El punto *iii)* es simplemente un encaje de patrones. Por último, el punto *iv)*

se cumplirá de manera trivial en la práctica, ya que las suposiciones sobre las constructoras y funciones no contienen variables libres. Este método para comprobar que un programa está bien tipado es en realidad una formalización alternativa y equivalente de la Definición 6. Los detalles y la demostración de equivalencia se puede encontrar en [87](A.2, Def. 3.2 y Lemma 3.1).

Veamos cómo son considerados según la noción de programa bien tipado algunos de los ejemplos aparecidos anteriormente. El ejemplo motivador de contar constructoras (Ejemplo 18, página 66) sería considerado como bien tipado con la suposición $size : \forall\alpha.\alpha \to nat$. Las reglas sin variables están bien tipadas de manera trivial, ya que el tipo de su lado derecho es el mismo que el tipo de su lado izquierdo: nat. En el caso de la cuarta regla de $size$, el tipo del lado derecho y su variable es (nat, β), que es más general que el del lado izquierdo (nat, nat). Lo mismo ocurre con la última regla, donde (nat, β, γ) es más general que $(nat, \delta, [\delta])$. La función $code$ del Ejemplo 17 (página 64), aceptado por el sistema $\vdash^{\bullet}$, también sería considerado como bien tipado en este sistema liberal. La razón es que en ambas reglas el tipo del lado derecho e izquierdo coinciden: $[bit]$. El ejemplo de constructoras existenciales del Ejemplo 11 (página 40) también estaría bien tipado, ya que el tipo del lado derecho de la regla de $getKey$ y sus variables X y F es $(\beta, \alpha, \alpha \to \beta)$, mientras que el lado izquierdo tiene el tipo más concreto $(nat, \gamma, \gamma \to nat)$. En último lugar, el *casting* polimórfico del Ejemplo 1 (página 4) sería rechazado debido a la función $unpack$, en la que falla el punto *iii)*. La causa es que el tipo del lado derecho de su regla y la variable X es (α, α), que no es más general que el tipo obtenido para su lado izquierdo (β, γ). En la Sección 7.4 veremos más ejemplos de programas bien tipados en el sistema de tipos liberal.

Un aspecto interesante del sistema de tipos liberal es que es estrictamente más general que el sistema $\vdash^{\bullet}$ en lo referente a los programas que considera bien tipados. Esto queda reflejado en el siguiente teorema:

Teorema 14 ([87](A.2, Th. 3.1)) *Si* $wt^{\bullet}_{\mathcal{A}}(\mathcal{P})$ *entonces* $wt^{l}_{\mathcal{A}}(\mathcal{P})$.

Este resultado es un indicador favorable sobre la generalidad del sistema de tipos liberal. Sin embargo, en la siguiente sección enunciaremos esa generalidad con precisión, demostrando que en cierta manera la definición de regla bien tipada corresponde con la noción de regla cuya aplicación preserva tipos.

7.3. Propiedades del sistema de tipos

En esta sección abordaremos la corrección del sistema de tipos liberal desde dos enfoques: la combinación de preservación de tipos y progreso, y un enfoque sintáctico similar al de [154]. También demostraremos la máxima liberalidad del sistema de tipos, que acepta exactamente aquellas reglas que preservan tipos.

La semántica elegida para este sistema de tipos es, al igual que en el sistema $\vdash^{\bullet}$, la let-reescritura (Figura 5, página 24). En esta sección la extenderemos con dos reglas

(Fapp) $f\ t_1\theta \ldots t_n\theta \ \rightarrow^{lf}\ r\theta$, si $(f\ t_1 \ldots t_n \rightarrow r) \in \mathcal{P}$

(Ffail) $f\ t_1 \ldots t_n \ \rightarrow^{lf}\ fail$, si $n = ar(f)$ y $\nexists (f\ t'_1 \ldots t'_n \rightarrow r) \in \mathcal{P}$ tal que $f\ t'_1 \ldots t'_n$ y $f\ t_1 \ldots t_n$ son unificables.

(FailP) $fail\ e \ \rightarrow^{lf}\ fail$

(LetIn) $e_1\ e_2 \rightarrow^{lf} let\ X = e_2\ in\ e_1\ X$, si e_2 es una expresión *junk*, activa, una aplicación de variables o una let-expresión, para X fresca

(Bind) $let\ X = t\ in\ e \ \rightarrow^{lf}\ e[X/t]$

(Elim) $let\ X = e_1\ in\ e_2 \rightarrow^{lf} e_2$, si $X \notin fv(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \ \rightarrow^{lf}\ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, si $Y \notin fv(e_3)$

(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \rightarrow^{lf} let\ X = e_1\ in\ e_2\ e_3$, si $X \notin fv(e_3)$

(Contx) $\mathcal{C}[e] \ \rightarrow^{lf}\ \mathcal{C}[e']$, si $\mathcal{C} \neq [\ ]$, $e \ \rightarrow^{lf}\ e'$ usando alguna de las reglas anteriores

Figura 16: Let-reescritura con fallo de encaje de patrones

para manejar el fallo de encaje de patrones, de manera similar a las semánticas para GADTs [28, 117]. El resultado se encuentra en la Figura 16. La regla (Ffail) genera un fallo cuando no existe ninguna regla para reducir una aplicación de función. Usamos la unificación sintáctica en lugar del encaje con los patrones de las reglas para poder realizar la comprobación localmente, sin tener que consultar el contexto de la expresión. Por ejemplo, consideremos la conjunción lógica $\wedge$ (definida con las reglas $true \wedge X \rightarrow X$ y $false \wedge X \rightarrow false$) y la expresión a reducir $let\ Y = true\ in\ (Y \wedge true)$. La subexpresión $Y \wedge true$ unifica con los lados izquierdos de ambas reglas, por lo que no se genera ningún fallo utilizando la regla (Ffail). Si hubiésemos realizado la comprobación utilizando encaje de patrones sin contemplar el contexto habríamos generado incorrectamente un fallo, ya que ni $true \wedge X$ ni $false \wedge X$ encajan con $Y \wedge true$, mientras que la expresión $let\ Y = true\ in\ (Y \wedge true)$ se reduce a $true$ sin problemas. En caso de haber definido la regla (Ffail) utilizando encaje de patrones habríamos tenido que incluir condiciones adicionales en la regla (Contx) para que tuviese en cuenta las ligaduras actuales de las variables. Por ello consideramos preferible el enfoque basado en unificación debido a su sencillez y claridad. La regla (FailP) simplemente propaga los fallos una vez que aparecen. El conjunto de *formas normales* $nf_{\mathcal{P}}(e)$ alcanzables desde una expresión e utilizando $\rightarrow^{lf}$ y un programa $\mathcal{P}$ se define como $nf_{\mathcal{P}}(e) = \{e' \mid \mathcal{P} \vdash e \rightarrow^{lf*} e'$ y e' no es $\rightarrow^{lf}$ -reducible$\}$.

La inclusión de las reglas (Ffail) y (FailP) responde al deseo de distinguir dos clases de reducciones fallidas que pueden ocurrir:

- Reducciones que no pueden progresar porque existen funciones cuyos patrones no cubren todos los casos. Un ejemplo de esta situación es $head\ [\ ]$, que no puede

reducirse pues no existe ninguna regla que trate la lista vacía. En el ámbito funcional, dicha expresión daría un error en tiempo de ejecución. Sin embargo, en PLF una situación así no debe verse como un error sino como un fallo silencioso en un espacio de cómputo, que indica que debe realizarse vuelta a atrás y probar otras elecciones indeterministas.

- Reducciones que se quedan bloqueadas por un error de tipos genuino, como las expresiones *junk* (constructoras sobreaplicadas).

Las reglas (Ffail) y (FailP) han sido introducidas para manejar el primer tipo de reducciones fallidas. Las reducciones del segundo tipo siguen quedando bloqueadas, incluso con las reglas añadidas. Esto solo puede ocurrir con las expresiones mal tipadas, como muestra el teorema de progreso:

Teorema 15 (Progreso, [87](A.2, Th. 4.1)) *Si* $wt^l_{\mathcal{A}}(\mathcal{P})$, $wt^l_{\mathcal{A}}(e)$ *y* e *no contiene variables libres, entonces* e *es un patrón o* $\exists e'.\ \mathcal{P} \vdash e \rightarrow^{lf} e'$.

Este teorema de progreso enuncia que una expresión bien tipada sin variables libres o bien es un patrón (un valor) o bien se puede reescribir. Es necesario considerar expresiones sin variables libres, ya que son las únicas que tienen sentido en el marco de let-reescritura considerado en este sistema de tipos. De otra manera el progreso no se cumpliría en expresiones como $F\ true$, que no es un patrón y no puede reescribirse en otra expresión pues la let-reescritura no soporta la ligadura de variables. Nótese que las expresiones *junk*, que no son patrones ni se pueden sobreescribir (como $true\ zero$), son excluidas del anterior resultado al estar mal tipadas, gracias a la restricción impuesta sobre los conjuntos de suposiciones de que los tipos de las constructoras deben corresponder con su aridad.

Aparte del progreso, el sistema de tipos cumple también la preservación de tipos:

Teorema 16 (Preservación de tipos, [87](A.2, Th. 4.2)) *Si* $wt^l_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^l e : \tau$ *y* $\mathcal{P} \vdash e \rightarrow^{lf} e'$, *entonces* $\mathcal{A} \vdash^l e' : \tau$.

Este resultado muestra que la liberalidad proporcionada por nuestro sistema de tipos, que es claramente mayor que la de DM u otros sistemas de tipos, es lo suficientemente estricta como para garantizar la preservación de tipos durante la reducción. De hecho, el sistema de tipos es lo más relajado que es posible ser para garantizar la preservación de tipos, como refleja el siguiente teorema:

Teorema 17 (Máxima liberalidad de las condiciones de $wt^l_{\mathcal{A}}(\mathcal{P})$**, [87](A.2, Th. 4.3))** *Consideremos un conjunto de suposiciones cerrado* $\mathcal{A}$ *y un programa que no está bien tipado con respecto a* $\mathcal{A}$ *pero en el cual todas las reglas cumplen la condición* i) *de la Definición 6 (página 68) de regla bien tipada. Entonces existirán tipos* $\overline{\tau_n}$ *y* τ *tal que* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^l f\ t_1 \ldots t_m : \tau$ *y* $f\ t_1 \ldots t_m \rightarrow^{lf} e$ *pero* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \not\vdash^l e : \tau$.

La condición de que todas las reglas cumplan el punto *i)* de la Definición 6 evita considerar el caso trivial de programas cuyos lados izquierdos están mal tipados. Como en conjuntos de suposiciones cerrados el punto *iv)* se cumple trivialmente, el anterior teorema únicamente considera programas que están mal tipados por *ii)* o *iii)*, es decir, por una falta de correspondencia entre el tipo del lado izquierdo y derecho en alguna regla. Además, la demostración del Teorema 17 (ver [87](A.2, §A.5) es constructiva en el sentido de que, dado un programa cumpliendo las condiciones del teorema construye un paso de let-reescritura que viola la preservación de tipos.

Basándose en el Teorema 17, es posible demostrar que nuestra noción de regla bien tipada captura esencialmente la noción de regla que preserva los tipos cuando es aplicada. Para enunciar este resultado utilizaremos las siguientes definiciones:

Definición 7 (Regla que preserva tipos, [87](A.2, Def. 4.1)) *Dado un conjunto de suposiciones $\mathcal{A}$ decimos que una regla $f\ t_1 \ldots t_m \to e$* preserva tipos *si*

(i) su lado izquierdo admite algún tipo, es decir, $wt^l_{\mathcal{A}\oplus\{\overline{X_n:\tau_n}\}}(f\ t_1 \ldots t_m)$ para algunos tipos $\overline{\tau_n}$, donde $\overline{X_n}$ son las variables de la regla —$\{\overline{X_n}\} = fv(f\ t_1 \ldots t_m)$—.

(ii) $\mathcal{A} \vdash^l f\ t_1\theta \ldots t_m\theta : \tau \Longrightarrow \mathcal{A} \vdash^l e\theta : \tau$, para cualquier sustitución θ y tipo τ.

Definición 8 (Conjunto de suposiciones completo, [87](A.2, Def. 4.2)) *Diremos que un conjunto de suposiciones $\mathcal{A}$ es* completo *si para cada tipo τ existe un patrón t_τ tal que solamente admite ese tipo, es decir, tal que $\mathcal{A} \vdash^l t_\tau : \tau$ y $\mathcal{A} \nvdash^l t_\tau : \tau'$ para todo $\tau' \neq \tau$.*

La primera condición de la Definición 7 evita reglas que preserven tipos de manera trivial porque su lado izquierdo esté mal tipado, es decir, que $\mathcal{A} \nvdash^l f\ t_1\theta \ldots t_m\theta : \tau$ para todo τ. Utilizando las anteriores definiciones, podemos enunciar la equivalencia entre reglas bien tipadas y reglas que preservan tipos:

Proposición 2 ([87](A.2, Prop. 4.1)) *Consideremos un conjunto de suposiciones completo $\mathcal{A}$, y una regla R. Entonces R preserva tipos si y solo si $wt^l_{\mathcal{A}}(R)$.*

La consideración de conjuntos de suposiciones completos es necesaria para evitar situaciones donde la preservación de tipos está potencialmente comprometida pero no es violada con los símbolos de constructora y función que aparecen en el programa. Sin embargo, la preservación de tipos se invalidaría añadiendo nuevos símbolos al programa. Esto se puede observar en el programa $\mathcal{P} \equiv \{id\ X \to X, f\ F \to F\ true\}$ con tipos $\mathcal{A} \equiv \{id : \forall\alpha.\alpha \to \alpha, f : \forall\alpha.(\alpha \to \alpha) \to bool\}$. El único patrón que se puede pasar como argumento a f para que la aplicación esté bien tipada es id, que preservará los tipos. En cambio, añadiendo la función $\{inc\ N \to N + 1\}$ con tipo $int \to int$ conseguiríamos que la regla de f no preservase tipos: $\mathcal{A} \vdash^l f\ inc : bool$ pero $\mathcal{A} \nvdash^l inc\ true : bool$. Nótese que en ambas situaciones la regla de f estaría mal tipada con respecto a la Definición 6 (página 68) ya que el lado derecho restringe el tipo de la variable F más que

el lado izquierdo, aunque en el primer caso no hay suficientes símbolos para violar la preservación de tipos.

Siguiendo con la corrección del sistema de tipos, es posible aplicar un enfoque sintáctico similar al de [154] basándonos en los anteriores resultados de progreso y preservación de tipos (Teoremas 15 y 16 respectivamente). Para ello consideraremos las siguientes clases de expresiones:

Definición 9 (Expresiones bloqueadas e incorrectas, [87](A.2, Def. 4.3)) *Una expresión e está* bloqueada *(*stuck*) con respecto a un programa $\mathcal{P}$ si es una forma normal (irreducible) pero no es un patrón. Por otro lado, una expresión es* incorrecta *(*faulty*) si contiene una subexpresión que es* junk*.*

La *corrección sintáctica* establece que todas las reducciones terminadas que comienzan en expresiones bien tipadas sin variables libres no llegan a expresiones bloqueadas sino a patrones del mismo tipo que la expresión original:

Teorema 18 (Corrección sintáctica, [87](A.2, Th. 4.4)) *Si $wt^l_{\mathcal{A}}(\mathcal{P})$, e no tiene variables libres y $\mathcal{A} \vdash^l e : \tau$ entonces: para todo $e' \in nf_{\mathcal{P}}(e)$, e' es un patrón y $\mathcal{A} \vdash^l e' : \tau$.*

Otro resultado complementario, similar a la corrección débil (*weak soundness*) de [154], establece que la evaluación de expresiones bien tipadas no pasa por ninguna expresión incorrecta:

Teorema 19 ([87](A.2, Th. 4.5)) *Si $wt^l_{\mathcal{A}}(\mathcal{P})$, $wt^l_{\mathcal{A}}(e)$ y e no contiene variables libres, entonces no existe ninguna expresión incorrecta e' tal que $\mathcal{P} \vdash e \rightarrow^{lf*} e'$.*

Los resultados de corrección que hemos mostrado (tanto el progreso y preservación de tipos como la corrección sintáctica) son más débiles que los originales de DM. Por ejemplo, en DM la expresión $head\ true$ está bloqueada, mientras que según nuestra semántica $head\ true \rightarrow^{lf} fail$. Esto es así porque en DM se considera una compleción bien tipada de las funciones parciales para generar errores de encaje de patrones, que añadiría la regla $head\ [\,] \rightarrow error$. En nuestro marco esto es más complicado, ya que la presencia de patrones de orden superior puede necesitar un número infinito de reglas para tratar los casos de error de encaje de patrones. Por ello hemos delegado esta labor en la regla (Ffail), que no tiene en cuenta los tipos (solo la posible aplicación de reglas) y por tanto permite que tanto $head\ [\,]$ como $head\ true$ se evalúen a $fail$. Sin embargo, $head\ true$ está mal tipado, por lo que la preservación de tipos nos garantiza que dicha expresión nunca aparecerá durante la reducción de una expresión bien tipada. Por otro lado, comentar que debido a la liberalidad conseguida, el sistema de tipos no goza de parametricidad (ver Sección 5.3.4, página 46). Esto se ve claramente en el Ejemplo 18 (página 66), que está bien tipado con la suposición $size : \forall\alpha \rightarrow nat$: el argumento de la función es polimórfico, no obstante, las reglas lo inspeccionan. Por lo tanto no es

posible la extracción de *teoremas gratis* a partir de los tipos de las funciones, aunque como ya vimos en la Sección 5.3.4 esa posibilidad ya está bastante comprometida en PLF debido al indeterminismo, las variables extra y el estrechamiento.

7.4. Ejemplos

En esta sección veremos algunos ejemplos mostrando la flexibilidad del sistema de tipos liberal. Comenzaremos con las funciones indexadas por tipo. Como hemos visto en la Sección 5.3.5 (página 48) se trata de funciones que tienen una definición distinta para distintos tipos. Un ejemplo que ya ha aparecido es la función $size$ para contar símbolos de constructora del Ejemplo 18 (página 66), que está definida de manera distinta para booleanos, números naturales y listas. Una definición alternativa se puede realizar mediante clases de tipos, declarando una clase $sizeable$ y creando instancias para los tipos deseados, o mediante GADTs utilizando representaciones de los tipos como primer argumento (ver [87](A.2, §5.1 y Fig. 4) para más detalles). Sin embargo, en el sistema liberal la función $size$ está bien tipada simplemente añadiendo la suposición $size : \forall\alpha.\alpha \to nat$[29]. Otra función indexada por tipos que se podría definir en el sistema de tipos liberal sería la igualdad (considerando que `/\` es la conjunción booleana):

Ejemplo 19 (Igualdad en el sistema de tipos liberal, [87](A.2, Fig. 4-a))

```
eq :: A -> A -> bool
eq true true   = true          eq zero zero         = true
eq true false  = false         eq zero (succ Y)     = false
eq false true  = false         eq (succ X) zero     = false
eq false false = true          eq (succ X) (succ Y) = eq X Y

eq (X1,Y1) (X2,Y2) = (eq X1 X2) /\ (eq Y1 Y2)
```

Como ya hemos comentado, los sistemas de PLF proporcionan habitualmente una primitiva *ad-hoc* para la igualdad estructural debido a que su definición mediante reglas estaría mal tipada. Por el contrario, en el sistema liberal una función de igualdad con comportamiento estructural similar a la del ejemplo anterior estaría bien tipada. Esto abriría la posibilidad de que los programadores definieran la igualdad según sus necesidades, pudiendo omitir aquellos casos que no quieran tratar. Además, el propio sistema de tipos impediría las reglas que produjesen descomposición opaca. Por ejemplo la regla $eq\ (snd\ X)\ (snd\ Y) \to eq\ X\ Y$ que produciría el paso de descomposición opaca $eq\ (snd\ true)\ (snd\ [\,]) \to^{lf} eq\ true\ [\,]$ del Ejemplo 1 (página 4) estaría mal tipada ya que el tipo de las variables X e Y en el lado derecho (ambas deben tener el mismo

[29] El sistema de tipos liberal también consideraría como bien tipado el mencionado enfoque que utiliza representaciones mediante GADTs como primer argumento.

tipo α) es más concreto que en el lado izquierdo (las dos pueden tener un tipo posiblemente distinto β y γ). Otro ejemplo de funciones indexadas por tipo en el sistema liberal se verá en la traducción alternativa de clases de tipo de la Sección 7.5.

El sistema liberal acepta reglas con constructoras de tipo existencial (o constructoras no transparentes según [45]) como $getKey$ del Ejemplo 11 (página 40). Sin embargo, da un tratamiento más permisivo a estas constructoras que el que permite el enfoque tradicional presentado en la Sección 5.3.1 (página 40). En ese enfoque tradicional se prohíben reglas como $getKey\ (mkKey\ true\ F) \to zero$ ya que violaría la ocultación de información: el primer argumento de $mkKey$ está cuantificado existencialmente pero se pretende encajar con $true$. En el sistema de tipos liberal esta regla estaría bien tipada, ya que los tipos del lado derecho (nat, α) son más generales que los del lado izquierdo $(nat, bool \to nat)$. Además de soportar constructoras de tipo existencial, el sistema de tipos liberal maneja también la opacidad generada por los patrones de orden superior. Esto queda reflejado en el Teorema 14 (página 70), que expresa que todos los programas bien tipados por el sistema $\vdash^{\bullet}$ también lo estarán con respecto al sistema liberal. En cambio, el sistema liberal va más allá ya que acepta reglas con variables críticas (que por tanto son rechazadas por el sistema $\vdash^{\bullet}$) siempre que preserven tipos. Un ejemplo sería la función $f\ (snd\ (X : Xs)) \to length\_nat\ Xs$ con tipo $\forall\alpha.(\alpha \to \alpha) \to nat$ donde la variable Xs es crítica, pero que está bien tipada con respecto al sistema de tipos liberal porque los tipos para el lado derecho y sus variables X y Xs son $(nat, \alpha, [\beta])$, más más generales que los del lado izquierdo $(nat, \gamma, [\gamma])$.

Otro ejemplo interesante, utilizando patrones de orden superior, es la conocida traducción de programas de orden superior a programas de primer orden [151] que se utiliza en la compilación de programas lógico-funcionales [106, 9, 92]. Básicamente esta transformación introduce una función especial @ (leída *apply*) para representar las aplicaciones parciales, y añade reglas para reducir esas llamadas. Lo importante en este caso es que la función @ está mal tipada en los sistemas de tipos derivados de DM, mientras que el sistema de tipos liberal la considera bien tipada. Un ejemplo de esta traducción aparece en el siguiente ejemplo:

Ejemplo 20 (Traducción de orden superior a primer orden, [87](A.2, §5.2))
Consideremos un programa con las reglas de las funciones $length$, $append$ y snd con los tipos usuales, además de las constructoras de naturales y listas[30]*. Las reglas generadas para @ por la traducción de orden superior a primer orden son*

```
@ :: (A -> B) -> A -> B
@ succ X      = succ X          @ append X     = append X
@ cons X      = cons X          @ (append X) Y = append X Y
@ (cons X) Xs = cons X Xs       @ snd X        = snd X
@ length Xs   = length Xs       @ (snd X) Y    = snd X Y
```

[30] Por razones de claridad en la exposición, utilizaremos la constructora de listas $cons$ en lugar de su versión infija $(:)$.

Todas las reglas anteriores de @ estarían bien tipadas en el sistema de tipos liberal, con lo cual la traducción de orden superior a primer orden podría ser considerada como una transformación de programa fuente a fuente en lugar de un paso *ad-hoc* del compilador.

Por último, el sistema de tipos liberal permite definir funciones genéricas en el sentido de que una sola definición se pueda aplicar de manera «automática» a diferentes tipos. Para ello utilizamos un tipo de datos *universal* para representar de manera uniforme todos los tipos de datos. La función genérica se define en términos de esa función universal, y a la hora de aplicarla sobre cualquier tipo de dato se utiliza una función conversora. Una forma sencilla de definir el tipo de datos universal es mediante la declaración $data\ univ = c\ nat\ [univ]$, donde el primer argumento sirve para numerar las constructoras y el segundo es la lista de argumentos de la aplicación de constructora. Una función universal $usize$ de tipo $univ \to nat$ que persiga un contar constructoras como en el Ejemplo 18 (página 66) se definiría como $usize\ (c\ N\ L) \to succ\ (sum\ (map\ usize\ L))$[31]. La versión genérica de la función $size$ se definiría como $size\ X = usize\ (toU\ X)$, donde $toU : \forall\alpha.\alpha \to univ$ es una función indexada por tipo que para convertir patrones a su representación universal. Considerando que $succ^i$ representa la aplicación i veces de la constructora $succ$, las reglas de toU para los valores del Ejemplo 18 serían:

```
toU true     = c zero []
toU false    = c (succ zero) []
toU zero     = c (succ^2 zero) []
toU (succ X) = c (succ^3 zero) [toU X]
toU []       = c (succ^4 zero) []
toU (X:Xs)   = c (succ^5 zero) [toU X, toU Xs]
```

Todas estas reglas estarían bien tipadas en el sistema de tipos liberal. Aparte de este tipo universal, que es ciertamente primitivo, se podrían adaptar otras representaciones más complejas como *spines* [64] o sumas de productos [62]. El primer enfoque estaría bien tipado en el sistema liberal en su formulación original utilizando GADTs, mientras que el segundo requeriría transformar las funciones sobrecargadas de las clases de tipos por funciones indexadas por tipo, como explicamos a continuación.

7.5. Aplicación a la implementación de clases de tipos

Esta sección presenta la traducción alternativa para clases de tipos usando funciones indexadas por tipo recogida en el artículo *Type Classes in Functional Logic Programming* [100](A.4).

Como hemos comentado en la Sección 5.3.2 (página 42) las clases de tipos son el aspecto que más interés ha despertado en la comunidad lógico-funcional desde el punto

[31] Utilizando las funciones usuales sum y map del preludio de Haskell.

`class arb A where`
`  arb :: A`
 	
`instance arb bool where`
`  arb = true`
`  arb = false`
 	
`arbL2 :: arb A => [A]`
`arbL2 = [arb, arb]`
 	
a) Programa original
	`data dictArb A = dictArb A`
	
	`arb :: dictArb A -> A`
	`arb (dictArb F) = F`
	
	$\texttt{arb}_{bool}$ `:: bool`
	$\texttt{arb}_{bool}$ `= true`
	$\texttt{arb}_{bool}$ `= false`
	
	`dictArbBool :: dictArb bool`
	`dictArbBool = dictArb` $\texttt{arb}_{bool}$
	
	`arbL2 :: dictArb A -> [A]`
	`arbL2 DA = [arb DA, arb DA]`
	
	b) Traducción con diccionarios

Figura 17: Traducción de un programa con una función sobrecargada indeterminista sin argumentos

de vista de los tipos. Por ello han surgido algunos trabajos estudiando nuevas posibilidades expresivas y problemas que aparecen al integrar clases de tipos en PLF [107, 95], además de haber surgido algunas implementaciones experimentales del lenguaje Curry que las soportan (como una rama del compilador de Münster [94], o los sistemas Sloth [40] y Zinc [16]). Sin embargo, es conocido que la traducción clásica de las clases de tipos utilizando diccionarios [150, 48] (ver Sección 5.3.2 —página 42— para más detalles acerca de esta traducción) presenta el problema de soluciones perdidas cuando se combinan el indeterminismo con funciones sobrecargadas sin argumentos [96]. Este problema se puede observar en la Figura 17, tomada de [96]. La función sobrecargada arb es un generador indeterminista, que se instancia para los booleanos, y la función $arbL2$ devuelve una lista de dos elementos generados por la función arb. La Figura 17 también contiene la traducción del programa utilizando diccionarios siguiendo el procedimiento clásico presentado en la Sección 5.3.2 (página 42). A la hora de evaluar $arbL2 :: [bool]$ los resultados esperados serían $[true, true]$, $[true, false]$, $[false, true]$ y $[false, false]$. Por el contrario, la evaluación en el programa transformado devolvería únicamente los valores $[true, true]$ y $[false, false]$. La razón de la pérdida de soluciones esperadas es la combinación de los diccionarios y la semántica de *call-time choice*,

como muestra la siguiente reducción de la expresión traducida:

$$
\begin{array}{rl}
\underline{arbL2\ dictArbBool} \rightarrow^{lf}_{(LetIn)} & let\ X = dictArbBool\ in\ \underline{arbL2\ X} \\
\rightarrow^{lf}_{(Fapp)} & let\ X = \underline{dictArbBool}\ in\ [arb\ X, arb\ X] \\
\rightarrow^{lf}_{(Fapp)} & let\ X = dictArb\ \underline{arb_{bool}}\ in\ [arb\ X, arb\ X] \\
\rightarrow^{lf}_{(Fapp)} & \underline{let\ X = dictArb\ true\ in\ [arb\ X, arb\ X]} \\
\rightarrow^{lf}_{(Bind)} & [arb\ (dictArb\ true), arb\ (dictArb\ true)] \\
\rightarrow^{lf*}_{(Fapp)} & [true, true]
\end{array}
$$

Debido a la semántica de *call-time choice*, el diccionario $dictArbBool$ que se pasa como argumento a ambas apariciones de la función arb en la lista debe ser compartido, por lo que el valor de la función eq_{bool} que contienen debe ser el mismo. Esto se plasma en la imposibilidad de aplicar (Bind) con la ligadura $X = dictArb\ arb_{bool}$ ya que $dictArb\ arb_{bool}$ no es un patrón, puesto que arb_{bool} es de aridad 0. Por tanto primero se debe evaluar arb_{bool} a un patrón ($true$ o $false$), que luego es compartido.

En esta sección propondremos una traducción de las clases de tipos basada en el enfoque de pasar tipos en lugar de diccionarios [146]. Debido a la facilidad del sistema de tipos liberal para definir funciones indexadas por tipos, no será necesario construcciones adicionales en el lenguaje para realizar encaje de patrones sobre tipos. Básicamente, cada función sobrecargada de una clase de tipos se convertirá en una función indexada por tipos que acepta como primer argumento un *testigo de tipo*, que sirve para determinar qué reglas son aplicables. Gracias a la utilización de funciones indexadas por tipo y los testigos de tipo, los programas traducidos no tendrán el problema de soluciones perdidas en presencia de funciones sobrecargadas indeterministas sin argumentos. Además, se conseguirá una traducción más simple, que da lugar a programas más pequeños, y cuyos programas traducidos se ejecutan más rápido que los de la traducción mediante diccionarios (con una ganancia que varia entre 1,05 y 2,3). Esta traducción propuesta, que acepta programas bien tipados utilizando el sistema de tipos usual de clases de tipos [150, 48] y produce programas bien tipados en el sistema de tipos liberal, fue presentada en el artículo *Type Classes in Functional Logic Programming* [100](A.4). Se trata de un trabajo eminentemente práctico, por lo que no se proporciona un resultado técnico que demuestre que los programas transformados están bien tipados en el sistema liberal, aunque es una idea que se desprende con bastante claridad a partir de la traducción. Tampoco se proporciona ningún resultado garantizando que la traducción conserva la semántica de los programas, ya que, como es usual con las clases de tipos [48], se considera que es la propia traducción la que confiere significado a los programas originales.

Variable de tipos	$\alpha, \beta, \gamma \ldots$	
Constructor de tipos	C	
Nombre de clase	κ , $\kappa^\bullet$	
Tipo simple	τ	::= $\alpha \mid \tau \to \tau' \mid C\ \overline{\tau_n}$ con $n = ar(C)$, $n \geq 0$
Contexto	θ	::= $\langle \overline{\kappa_n\ \alpha_n} \rangle$ con $n \geq 0$
Contexto saturado	ϕ	::= $\langle \overline{\kappa_n\ \tau_n} \rangle$ con $n \geq 0$
Tipo sobrecargado	ρ	::= $\phi \Rightarrow \tau$

Figura 18: Sintaxis de los tipos utilizados por las clases de tipos

Símbolo de función	f	
Símbolo de constructora	c	
Variable de datos	X	
Programa	::=	$\overline{data}\ \overline{class}\ \overline{inst}\ \overline{type}\ \overline{rule}$
data	::=	$\texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \ldots \mid c_k\ \overline{\tau}$
class	::=	$\texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \overline{f :: \tau}$
inst	::=	$\texttt{instance}\ \theta \Rightarrow \kappa\ (C\ \overline{\alpha})\ \texttt{where}\ \overline{f\ \overline{t} \to e}$ con $\overline{t}$ lineal
type	::=	$f :: \theta \Rightarrow \tau$
rule r	::=	$(f :: \rho)\ \overline{t} \to e$ con $\overline{t}$ lineal
pattern t	::=	$X \mid c\ \overline{t_n}$ con $n \leq ar(c) \mid (f :: \langle\rangle \Rightarrow \tau)\ \overline{t_n}$ con $n < ar(f)$
expression e	::=	$X \mid c \mid f :: \rho \mid e\ e \mid let\ X = e\ in\ e$

Figura 19: Sintaxis de los programas con clases de tipos

7.5.1. Programas originales

Los tipos considerados para los programas originales (Figura 18) son similares a los que aparecen en los sistemas de clases de tipos de un solo parámetro [150, 48]. Utilizaremos la letra κ para referirnos a nombres de clase (como por ejemplo ord, eq ...), que pueden estar marcadas con $\bullet$ (como $ord^\bullet$ o $eq^\bullet$). Esta marca es importante a la hora de traducir los programas, ya que sirve para indicar los testigos de qué tipos será necesario pasar como argumentos de las funciones indexadas por tipo, como veremos más adelante. Con los nombres de clase se forman *restricciones de clase* $\kappa\ \tau$, que dan lugar a *contextos* θ[32], si las restricciones afectan solamente a variables, o a *contextos saturados* ϕ, si las restricciones afectan a tipos simples. Por último, un *tipo sobrecargado* ρ en un tipo simple acompañado por un contexto saturado. De manera general, en esta sección utilizaremos el término «función sobrecargada» para referirnos a toda aquella función cuyo tipo (inferido o declarado) tienen un contexto no vacío. A la hora de referirnos a una función sobrecargada que forma parte de una clase de tipos lo haremos de manera explícita.

[32]Se representan con la misma letra que las sustituciones de datos, pero siempre quedará claro por el contexto a qué noción nos referimos.

Los programas considerados (Figura 19) están compuestos por declaraciones de datos `data`, declaraciones de clases de tipos `class`, declaraciones de instancia de clase de tipo `inst`, declaraciones de tipos para las funciones `type` y reglas `rule`. A diferencia del enfoque seguido en el resto de la tesis, en esta sección consideraremos programas con declaraciones explícitas de las constructoras y los tipos de la función, ya que la transformación utilizará dichas declaraciones para añadir nuevas constructoras o declaraciones de tipos. Un aspecto particular de la sintaxis de los programas originales es que todos los símbolos de función en reglas y expresiones vendrán decorados con un contexto saturado. Sin embargo, no permitiremos patrones de orden superior formados por funciones sobrecargadas en los lados izquierdos de las reglas, ya que dan lugar a problemas sutiles durante la traducción, como se explica en [100](A.4, §5.3). Por ello, el contexto que acompañará a los símbolos de función que aparezcan en los patrones de los lados izquierdos de las reglas será el contexto vacío $\langle\rangle$.

Los contextos que acompañan a los símbolos de función del programa serán calculados por una fase previa de comprobación de tipos que utiliza el sistema de tipos usual para clases de tipos [150, 48], reflejando a qué tipos concretos se aplica la función. Por ejemplo, suponiendo que la función `eq`[33] tiene el tipo usual $\langle eq\ \alpha\rangle \Rightarrow \alpha \to \alpha \to bool$, la regla `g X` $\to$ `eq X` $[$`true`$]$ sería decorada como:

$$(\texttt{g} :: \langle\rangle \Rightarrow [bool] \to bool)\ \texttt{X} \to (\texttt{eq} :: \langle eq\ [bool]\rangle \Rightarrow [bool] \to [bool] \to bool)\ \texttt{X}\ [\texttt{true}]$$

El contexto saturado $eq\ [bool]$ en el lado derecho indica que la función sobrecargada `eq` está aplicada a elementos del tipo $[bool]$, así que será necesario pasarle un *testigo* de ese tipo como primer argumento, como explicaremos en la siguiente sección.

7.5.2. Traducción

La idea de la traducción propuesta es que las funciones sobrecargadas de las clases de tipos sean transformadas en funciones indexadas por tipo. En lugar de pasar diccionarios conteniendo la implementación concreta de la función sobrecargada, pasaremos testigos de tipo (patrones que representan tipos) que le indicarán a la función indexada por tipo qué reglas son aplicables. En el programa original, los contextos saturados que decoran los símbolos de función contienen la información sobre el tipo al que son aplicadas las funciones sobrecargadas. Por tanto, utilizaremos esos contextos para generar los testigos de tipos necesarios.

En lugar de utilizar una representación de los tipos mediante GADTs similar a la que se utiliza en [64] para conseguir funciones indexadas por tipo, utilizaremos un enfoque diferente: extenderemos cada declaración de tipos con una constructora para

[33] Como en la traducción se mezclarán trozos de los programas con anotaciones de tipos, en esta sección seguiremos el criterio de distinguir los fragmentos de programa con una fuente `monoespaciada` y los tipos con $cursiva$.

representar el tipo declarado. Por ejemplo, la declaración los números naturales sería extendida con la constructora $\#\texttt{nat}$, resultando en $\texttt{data nat} = \texttt{zero} \mid \texttt{succ nat} \mid \#\texttt{nat}$; mientras que la declaración de las listas sería extendida con la constructora $\#\texttt{list A}$, resultando en la declaración $\texttt{data list A} = \texttt{nil} \mid \texttt{cons A (list A)} \mid \#\texttt{list A}$[34]. Lo importante de los testigos de tipo así construidos es que tienen *el mismo tipo* que representan. Por ejemplo, $\#\texttt{nat}$ tiene tipo nat, y $\#\texttt{list}\ \#\texttt{nat}$ tiene tipo $list\ nat$. Este vínculo entre tipos y testigos permite que sea muy sencillo generar testigos a partir los tipos simples:

Definición 10 (Generación de testigos de tipos, [100](A.4, Def. 2))

- *testify(α) = X_α.*
- *testify($C\ \tau_1 \ldots \tau_n$) = #C testify(τ_1) . . . testify(τ_n).*

La función *testify* devuelve la misma variable de datos X_α para la misma variable de tipos α. Además, no está definida para tipos funcionales, aunque eso no es una limitación ya que consideramos un lenguaje origen en el que no se permiten instancias sobre tipos funcionales. De esta manera, nunca necesitaremos generar testigos para estos tipos, aunque se podría solventar con una constructora *ad-hoc* $\#\texttt{arrow}$ de tipo $\alpha \to \beta \to (\alpha \to \beta)$.

A la hora de generar los testigos de tipo para pasarlos como argumentos a las funciones sobrecargadas, es necesario generar testigos *para todos los tipos diferentes* que aparezcan en su contexto saturado. Sin embargo, no es necesario generar testigos *para todas las restricciones de clase* que aparezcan, sino solo una por cada tipo distinto. Esto es diferente a la traducción clásica, que necesita pasar un diccionario por cada restricción de clase, aunque varias afecten al mismo tipo. Por ejemplo, consideremos la función de Fibonacci

$$\texttt{fib N} = \texttt{if N} < 2\ \texttt{then (succ zero) else add (fib (N} - 1\texttt{)) (fib (N} - 2\texttt{))}$$

que devuelve un número natural. Teniendo en cuenta las definiciones de clases de tipos estándar en Haskell, el tipo inferido para `fib` es $\langle num\ \alpha, ord\ \alpha \rangle \Rightarrow \alpha \to nat$ (obsérvese que las clases num y ord no son subclases la una de la otra). La traducción clásica necesitaría pasar dos diccionarios, uno para la clase num y otro para la clase ord, pues ambos contendrían versiones especializadas de funciones sobrecargadas distintas. Sin embargo, en la traducción propuesta no es necesario pasar dos testigos duplicados del tipo α a la regla, sino uno que sirva para indicar a la función indexada por tipo qué comportamiento específico se espera. Para tratar esta situación es necesario que, tras la fase de inferencia de tipos clásica que decora los símbolos de función, se realice un proceso para marcar con $\bullet$ una restricción por variable en los contextos de los

[34]Para facilitar la presentación, en este apartado utilizaremos de manera indistinta la sintaxis de listas mediante las constructoras prefijas $nil/cons$ o infijas $[\,]/(:)$, representando al tipo de las listas como $list\ A$ o $[A]$.

tipos inferidos (o declarados) para las funciones del programa. Este proceso es muy sencillo, ya que únicamente debe recorrer los contextos de los tipos de las funciones del programa y marcar (por ejemplo, de izquierda a derecha) aquellas restricciones que afecten a variables de tipo aún no marcadas. Nótese que debido al mecanismo de reducción de contexto [116] (*context reduction*) incorporado en la inferencia clásica —que sirve para simplificar los contextos eliminando restricciones redundantes—, las restricciones de clase que aparecen en los contextos de los tipos de las funciones del programa afectarán solo a variables de tipo, y no a otros tipos simples[35]. Además, el proceso de marcado se propagará a todas las aplicaciones de dichas funciones que aparezcan en el programa, marcando sus contextos saturados de la misma manera que ha marcado el contexto de la función. De esta manera, el tipo final para `fib` será $\langle num^{\bullet}\ \alpha, ord\ \alpha\rangle \Rightarrow \alpha \to nat$, indicándonos que solo es necesario generar el testigo para el tipo de la restricción $num^{\bullet}\ \alpha$. Además, en todos los lugares donde se aplique `fib` se marcará con $\bullet$ la primera restricción de su contexto saturado. Por ejemplo, una aplicación `fib zero` sería decorada con `fib` $:: \langle num^{\bullet}\ nat, ord\ nat\rangle \Rightarrow nat \to nat$ `zero`, indicando que se ha de añadir un solo testigo de nat.

La traducción de las clases de tipos se define con un conjunto de funciones que transforman las diferentes construcciones que aparecen en un programa: declaraciones de datos, declaraciones de clases e instancias, declaraciones de tipos, reglas y expresiones.

Definición 11 (Funciones de traducción, [100](A.4, Def. 3))

- $trans_{prog}(\overline{data}\ \overline{class}\ \overline{inst}\ \overline{type}\ \overline{rule}) =$
  $\overline{trans_{data}(data)}\ \overline{trans_{class}(class)}\ \overline{trans_{inst}(inst)}\ \overline{trans_{type}(type)}\ \overline{trans_{rule}(rule)}$
- $trans_{data}(\texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \ldots \mid c_k\ \overline{\tau}) = \texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \ldots \mid c_k\ \overline{\tau} \mid \#C\ \overline{\alpha}$
- $trans_{class}(\texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \overline{f :: \tau}) = \overline{f :: \alpha \to \tau}$
- $trans_{inst}(\texttt{instance}\ \theta \Rightarrow \kappa\ (C\ \overline{\alpha})\ \texttt{where}\ \overline{f\ \overline{t} \to e}) =$
  $\overline{f\ testify(C\ \overline{\alpha})\ \overline{trans_{expr}(t)} \to trans_{expr}(e)}$
- $trans_{type}(f :: \theta \Rightarrow \tau) = f :: \alpha_1 \to \ldots \to \alpha_n \to \tau$,
  donde $\alpha_1 \ldots \alpha_n$ *aparecen en restricciones de clase de* θ *marcadas con* $\bullet$
- $trans_{rule}((f :: \rho)\ \overline{t} \to e) = trans_{expr}(f :: \rho)\ \overline{trans_{expr}(t)} \to trans_{expr}(e)$
- $trans_{expr}(X) = X$
  $trans_{expr}(c) = c$
  $trans_{expr}(f :: \rho) = f\ testify(\tau_1) \ldots testify(\tau_n)$, *donde* $\rho \equiv \phi \Rightarrow \tau$ *y* $\tau_1 \ldots \tau_n$ *aparecen en restricciones de clase de* ϕ *marcadas con* $\bullet$
  $trans_{expr}(e\ e') = trans_{expr}(e)\ trans_{expr}(e')$
  $trans_{expr}(let\ X = e\ in\ e') = let\ X = trans_{expr}(e)\ in\ trans_{expr}(e')$

[35] Se puede encontrar más información en [100](A.4, §3.3).

La traducción $trans_{prog}$ de un programa es la traducción de sus componentes. Las declaraciones de datos son extendidas por $trans_{data}$ con los testigos del tipo declarado, como se explicó anteriormente. Las declaraciones de clases de tipos dan lugar mediante la función $trans_{class}$ a declaraciones de tipos para las funciones indexadas por tipo, añadiendo un primer argumento que será el testigo de tipo. Por ejemplo, la declaración de la clase `foo`

`class foo A where`

`foo` $:: A \to bool$

daría lugar a la declaración del tipo de la función `foo` $:: A \to A \to bool$. Las declaraciones de tipo de funciones son traducidas por $trans_{type}$ de manera similar a $trans_{class}$, aunque en este caso solo se añaden argumentos extra a la función si su contexto contiene restricciones de clase marcadas con $\bullet$. Por ejemplo, una declaración de tipo `f` $:: \langle eq^{\bullet}\ A, show\ A, eq^{\bullet}\ B\rangle \Rightarrow A \to B \to bool$ sería traducido en `f` $:: A \to B \to A \to B \to bool$, extendiendo la declaración con los argumentos A y B, que son la variables afectadas por restricciones de clase marcadas con $\bullet$. Para las instancias, $trans_{inst}$ traduce sus reglas una a una. Para ello se introduce como primer argumento de cada regla el testigo del tipo de la instancia, que servirá para distinguir las reglas de la función indexada por tipo que afectan al mismo tipo. Por ejemplo, la declaración de instancia de la clase `foo` para $list\ A$

`instance` $foo\ (list\ A)$ `where`

`foo X` $\to$ `false`

daría lugar a la regla `foo` $(\#$`list` $\mathtt{X}_{\mathtt{A}})$ `X` $\to$ `false`, en la que se ha introducido el testigo $\#$`list` $\mathtt{X}_{\mathtt{A}}$ del tipo $list\ A$ como primer argumento. Obsérvese que en las reglas de las instancias no sería necesario añadir más testigos que el propio de la instancia, ya que los tipos de las funciones sobrecargadas de las clases están restringidos a ser tipos simples (ver Figura 19, página 80), con lo que no pueden contener ningún contexto. Para traducir una regla, $trans_{rule}$ traduce todos sus componentes. Como hemos explicado anteriormente, los patrones del lado izquierdo no contendrán funciones sobrecargadas, con lo que la traducción de los patrones no introducirá ningún testigo de tipos sino que se limitará a eliminar las decoraciones de tipos. Lo más importante a la hora de traducir una regla es la traducción del símbolo de la función. Cuando esta se trate de una función sobrecargada, su contexto será no vacío y deberemos proporcionar los testigos de tipos que necesite. Para ello se inspecciona el contexto saturado ϕ, añadiendo los testigos para los tipos afectados por restricciones de clase marcadas con $\bullet$. El orden en el que se añaden estos testigos es importante, y debe ser el mismo en todas las apariciones de la misma función sobrecargada. Por ejemplo, una aparición de la anterior función `f` aplicada a tipos concretos quedaría como `f` $::\langle eq^{\bullet}\ bool, show\ bool, eq^{\bullet}\ (list\ nat)\rangle \Rightarrow bool \to (list\ nat) \to bool$, por lo que sería transformada en `f` $\#$`bool` $(\#$`list` $\#$`nat`$)$.

Como se puede observar, la traducción de una expresión sin funciones sobrecargadas es la expresión original eliminando las decoraciones de tipos en los símbolos

de función. Lo mismo ocurre con los programas. Por tanto, en esos casos la traducción no penaliza en ningún modo la eficiencia. Adviértase también que en realidad la traducción no utiliza completamente las anotaciones de tipos de las funciones, sino únicamente su contexto. Sin embargo, hemos elegido incluir las decoraciones completas para recalcar la relación entre la traducción y la etapa de inferencia de tipos previa.

En la Figura 20 vemos un programa origen completamente decorado, y en la Figura 21 su traducción. En este programa consideramos las funciones booleanas `and` y `or` de tipo $\langle\rangle \Rightarrow bool \to bool \to bool$, y la función condicional `if_then` de tipo $\langle\rangle \Rightarrow bool \to A \to A$. También consideramos que existen funciones para comparar la igualdad y el orden de booleanos y naturales: `eqBool`, `eqNat`, `gtBool` y `gtNat`. Las decoraciones de tipos en los símbolos de función habrán sido añadidas por la etapa de inferencia de tipos, así que el programador no tendrá que haberlas añadido manualmente. Las marcas $\bullet$ en las restricciones de clase de los contextos también habrán sido añadidas automáticamente durante la inferencia, de la manera que hemos comentado antes. Hemos definido las funciones `eq` y `gt` con dos argumentos, en lugar de definirlas de manera más concisa como $\texttt{eq} = \texttt{eqBool}$ o $\texttt{gt} = \texttt{gtNat}$, para que las reglas tengan aridad 2 y se puedan formar patrones de orden superior con esos símbolos. Aunque en PF es posible definir las reglas de una misma función sobrecargada de una clase con distintas aridades en distintas instancias, en PLF es necesario que todas ellas tengan la misma aridad, como se explica en [100](A.4, §5.3). Por último, hacer notar cómo en la etapa de inferencia de tipos se ha decorado cada símbolo de función con el tipo correspondiente instanciado al tipo usado en la aplicación. Esto se puede ver en la última regla de la instancia de igualdad para listas. En el lado derecho de esa regla el símbolo aparece decorado con $\langle eq^{\bullet}\ A\rangle \Rightarrow A \to A \to bool$, cuando aparece aplicado a los elementos `X` e `Y`, y con la decoración $\langle eq^{\bullet}\ (list\ A)\rangle \Rightarrow (list\ A) \to (list\ A) \to bool$ cuando aparece aplicado a las listas `Xs` e `Ys`. De esta manera, en el primer caso se pasará el testigo $\texttt{X}_{\texttt{A}}$ del tipo A de los elementos de la lista, mientras que en el segundo se pasará el testigo $\#\texttt{list}\ \texttt{X}_{\texttt{A}}$ del tipo $list\ A$.

7.5.3. Ventajas de la traducción

Una de las ventajas de la traducción propuesta es que resuelve el problema de soluciones perdidas que presenta la traducción clásica utilizando diccionarios cuando se aplica a programas indeterministas con semántica de *call-time choice*. Volviendo al programa de la Figura 17-a (página 78), el programa traducido utilizando funciones indexadas por tipo y testigos de tipos sería:

$$\begin{array}{ll} \texttt{arb} :: A \to A & \texttt{arbL2} :: A \to list\ A \\ \texttt{arb}\ \#\texttt{bool} = \texttt{true} & \texttt{arbL2}\ \texttt{X}_{\texttt{A}} = [\texttt{arb}\ \texttt{X}_{\texttt{A}}, \texttt{arb}\ \texttt{X}_{\texttt{A}}] \\ \texttt{arb}\ \#\texttt{bool} = \texttt{false} & \end{array}$$

```
class eq A where
 eq :: A → A → bool

instance eq bool where
 eq X Y = eqBool :: ⟨⟩ ⇒ bool → bool → bool X Y

instance eq nat where
 eq X Y = eqNat :: ⟨⟩ ⇒ nat → nat → bool X Y

instance ⟨eq A⟩ ⇒ eq (list A) where
 eq [ ] [ ] = true
 eq [ ] (Y : Ys) = false
 eq (X : Xs) [ ] = false
 eq (X : Xs) (Y : Ys) = and :: ⟨⟩ ⇒ bool → bool → bool
  (eq :: ⟨eq• A⟩ ⇒ A → A → bool X Y)
  (eq :: ⟨eq• (list A)⟩ ⇒ (list A) → (list A) → bool Xs Ys)

member :: ⟨eq• A⟩ ⇒ (list A) → A → bool
member :: ⟨eq• A⟩ ⇒ (list A) → A → bool [ ] Y = false
member :: ⟨eq• A⟩ ⇒ (list A) → A → bool (X : Xs) Y =
 or :: ⟨⟩ ⇒ bool → bool → bool
  (eq :: ⟨eq•A⟩ ⇒ A → A → bool X Y)
  (member :: ⟨eq• A⟩ ⇒ (list A) → A → bool Xs Y)

class ⟨eq A⟩ ⇒ ord A where
 gt :: A → A → bool

instance ord bool where
 gt X Y = gtBool :: ⟨⟩ ⇒ bool → bool → bool X Y

instance ord nat where
 gt X Y = gtNat :: ⟨⟩ ⇒ nat → nat → bool X Y

memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool
memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool [ ] Y = false
memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool (X : Xs) Y =
 if_then :: ⟨⟩ ⇒ bool → bool → bool
  (gt :: ⟨ord• A⟩ ⇒ A → A → bool X Y) false
memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool (X : Xs) Y =
 if_then :: ⟨⟩ ⇒ bool → bool → bool
  (eq :: ⟨eq• A⟩ ⇒ A → A → bool X Y) true
memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool (X : Xs) Y =
 if_then :: ⟨⟩ ⇒ bool → bool → bool
  (gt :: ⟨ord• A⟩ ⇒ A → A → bool Y X)
  (memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool Xs Y)
```

Figura 20: Programa con clases de tipos decorado

```
eq :: A → A → A → bool
eq #bool X Y = eqBool X Y
eq #nat X Y = eqNat X Y
eq (#list X_A ) [ ] [ ] = true
eq (#list X_A) [ ] (Y : Ys) = false
eq (#list X_A) (X : Xs) [ ] = false
eq (#list X_A) (X : Xs) (Y : Ys) = and (eq X_A X Y) (eq (#list X_A) Xs Ys)

member :: A → (list A) → A → bool
member X_A [ ] Y = false
member X_A (X : Xs) Y = or (eq X_A X Y) (member X_A Xs Y)

gt :: A → A → A → bool
gt #bool X Y = gtBool X Y
gt #nat X Y = gtNat X Y

memberOrd :: A → (list A) → A → bool
memberOrd X_A [ ] Y = false
memberOrd X_A (X : Xs) Y = if_then (gt X_A X Y) false
memberOrd X_A (X : Xs) Y = if_then (eq X_A X Y) true
memberOrd X_A (X : Xs) Y = if_then (gt X_A Y X) (memberOrd X_A Xs Y)
```

Figura 21: Programa traducido utilizando funciones indexadas por tipo

Considerando los contextos marcados necesarios para la traducción, el tipo de la función `arbL2` original de la Figura 17-a sería $\langle arb^{\bullet}\ A\rangle \Rightarrow list\ A$. Por ello la expresión `arbL2` de tipo $[bool]$ resultaría decorada como `arbL2` $:: \langle arb^{\bullet}\ bool\rangle \Rightarrow list\ bool$, que sería traducida a `arbL2 #bool`. A partir de esta expresión traducida sí que es posible alcanzar las soluciones que se perdían con los diccionarios:

$$\begin{aligned}
\underline{arbL2\ \#bool} &\rightarrow^{lf}_{(Fapp)} [\underline{arb\ \#bool}, arb\ \#bool] \\
&\rightarrow^{lf}_{(Fapp)} [true, \underline{arb\ \#bool}] \\
&\rightarrow^{lf}_{(Fapp)} [true, false]
\end{aligned}$$

De la misma manera se podría obtener $[\texttt{false}, \texttt{true}]$, la otra solución perdida. El problema con los diccionarios era que la semántica de *call-time choice* hacía que se tuviesen que compartir el valor de las funciones sobrecargadas de aridad 0 contenidas en ellos, imposibilitando que distintas apariciones tomasen distintos valores indeterministas. En cambio, los testigos son patrones que sirven a la función indexada por tipo únicamente para elegir qué reglas son aplicables. De esta manera, la compartición de los testigos no hará perder soluciones, ya que el indeterminismo quedará «protegido» por las propias reglas de la función indexada por tipo.

Aparte de resolver el problema de las soluciones perdidas, la traducción da lugar a programas más eficientes que los programas obtenidos utilizando la traducción de diccionarios. Para ello hemos realizado pruebas sobre distintas funciones que utilizan

Programa	Ganancia	Ganancia con optimizaciones
eqlist	1,6414	1,3627
fib	2,3063	2,3777
galeprimes	1,4885	1,0016
memberord	2,2802	2,2386
mergesort	1,0476	1,0453
permutsort	1,7186	1,7259
quicksort	1,0743	1,0005

Figura 22: Ganancia en tiempo de ejecución de la traducción propuesta sobre la traducción clásica utilizando diccionarios

las clases de tipos eq, ord y num, adaptando algunas de estas funciones de la *suite* de pruebas de eficiencia *nobench* [141] para Haskell. Estas funciones cubren la igualdad de listas de enteros (*eqlist*), el cálculo del número de Fibonacci (*fib*), la criba de números primos (*galeprimes*), la búsqueda de elementos en listas ordenadas (*memberord*) y la ordenación de listas (*mergesort*, *permutsort*, *quicksort*). Para cada una de las funciones hemos realizado ambas traducciones a mano, produciendo programas Toy, y hemos medido su tiempo de ejecución para la evaluación de 100 expresiones aleatorias[36]. Los resultados de ganancia (tiempo del programa traducido que utiliza diccionarios dividido entre el tiempo del programa traducido que utiliza funciones indexadas por tipo) pueden encontrarse en la columna «Ganancia» de la Figura 22. Aunque en algunos casos la ganancia es casi inapreciable, en la mayoría de los casos se obtiene una ganancia considerable. Esta se puede explicar teniendo en cuenta que cuando intervienen clases de tipos relacionadas los diccionarios pueden llegar a ser construcciones bastante complejas, puesto que los diccionarios de las subclases deben contener los diccionarios de todas sus superclases (ver [150, 48]). En estas ocasiones, acceder a una función sobrecargada de una clase de tipos puede requerir varias extracciones de diccionarios intermedios, penalizando el tiempo de ejecución. Sin embargo, en la traducción propuesta no existe penalización debido a la complejidad de la jerarquía de clases, pues siempre se pasa el testigo del tipo a las funciones sobrecargadas independientemente de la complejidad de la jerarquía de clases considerada. Otro aspecto que explica esta ganancia es el marcado que realizamos con $\bullet$ de las restricciones de clase. Cada restricción de clase en el contexto de una función indica que será necesario pasar un diccionario diferente. Por el contrario, en nuestra traducción solo pasaremos un testigo por cada tipo, aunque ese tipo esté presente en varias restricciones de clase. Por ello, la traducción usando funciones indexadas por tipo puede ahorrar argumentos en funciones que utilicen funciones sobrecargadas en su cuerpo.

Existen algunas optimizaciones conocidas que se pueden aplicar a la traducción de

[36] Para su ejecución hemos utilizado el sistema Toy 2.3.2 sobre Ubuntu 10.04 LTS, en un máquina con procesador Intel® Core™2 Quad Q9550 y 2 GB de memoria.

diccionarios [12, 49], como el *aplanamiento de diccionarios* o la *especialización de funciones*. Sin embargo, también hay lugar para optimizaciones similares en la traducción utilizando funciones indexadas por tipo. Para conseguir unos resultados de ganancia que tuvieran en cuenta el estado actual de la traducción de diccionarios, hemos repetido las pruebas considerando algunas optimizaciones en ambas traducciones. El resultado puede encontrarse en la columna «Ganancia con optimizaciones» de la Figura 22. Aunque los resultados de ganancia obtenidos son menores que sin considerar optimizaciones, se observa que en algunos casos la ganancia sigue siendo bastante importante, y en ninguno de ellos se producen programas más lentos que con diccionarios. Para ampliar la información sobre las pruebas y sus resultados remitimos al lector a [100](A.4, §4.1).

Por último, mencionar que los programas obtenidos con la traducción propuesta son más simples que los obtenidos mediante la traducción de diccionarios. Estos programas obtenidos suelen ser más cortos, puesto que declaran menos tipos de datos y funciones. Además los testigos de tipos son datos de primer orden, al contrario que los diccionarios, que son datos de orden superior porque contienen funciones en su interior.

7.6. Conclusiones

El sistema de tipos liberal proporciona seguridad desde el punto de vista de los tipos (preservación de tipo, progreso y corrección sintáctica) a la vez que acepta programas que usualmente son rechazados por los sistemas de tipos de PLF e incluso PF. En particular es más general que el sistema $\vdash^{\bullet}$, como indica el Teorema 14 (página 70). La noción de programa bien tipado es sencilla, y se basa en la bien conocida relación de tipado DM para expresiones: los lados derechos de las reglas no pueden restringir el tipo de las variables de la regla más que los lados izquierdos. La liberalidad conseguida con esta noción provoca que algunas expresiones no tengan un tipo principal, lo que impide la existencia de un algoritmo de inferencia de tipos al estilo del algoritmo $\mathcal{W}$ o del algoritmo $\Vdash^{\bullet}$ de la sección anterior. En cambio, sí que es posible desarrollar un método efectivo para comprobar si un programa está o no bien tipado a partir de las declaraciones de tipo de las funciones, como se demuestra en [87](A.2, Def. 3.2 y Lemma 3.1). Utilizando ese método efectivo de comprobación de tipos hemos desarrollado dos sistemas que lo integran. El primero de ellos es una interfaz web del sistema de tipos[37] que sirve para comprobar de manera sencilla si un programa está o no bien tipado. El segundo sistema es una rama del sistema Toy[38] en la cual el sistema de tipos tradicional ha sido reemplazado por el sistema de tipos liberal. Aunque soporta únicamente la sintaxis clásica para declaración de tipos de datos (por lo que no es posible utilizar constructoras existenciales ni GADTs), proporciona un sistema Toy completo y

[37] Disponible en `http://gpd.sip.ucm.es/LiberalTyping`.
[38] Disponible en `http://gpd.sip.ucm.es/Toy2Liberal`.

funcional en el que compilar programas liberales y evaluarlos.

El sistema de tipos liberal proporciona una gran flexibilidad y expresividad al programador, permitiéndoles definir una amplia variedad de funciones, algunas de ellas prohibidas por los sistemas de tipos utilizados en PLF o PF, basados en DM. Entre ellas encontramos las funciones genéricas, que se podrían definir en el sistema de tipos liberal basándose en representaciones universales de los tipos de datos como *spines* o sumas de productos. También es interesante el caso de la función @ (*apply*) generada durante la traducción de orden superior a primer orden que forma parte de la compilación de programas lógico-funcionales. En el sistema de tipos liberal esta función estaría bien tipada, por lo que dicha traducción podría considerarse como una transformación de programas en lugar de una fase específica de la compilación. El sistema de tipos liberal también da soporte de manera directa a las constructoras existenciales, los patrones opacos y a declaraciones de tipos de datos al estilo de los GADTs, sin necesidad de ninguna extensión.

Una mención aparte merecen las funciones indexadas por tipo, es decir, aquellas que tienen un comportamiento distinto para cada tipo. Entre estas funciones destaca el caso de la función de igualdad estructural, que en la práctica está incrustada en los sistemas como una primitiva *ad-hoc* debido a que sus reglas estarían mal tipadas. Esto no es así con el sistema liberal, donde la igualdad sería una función bien tipada cuyas reglas podría aparecer en un preludio o ser definidas por el usuario. Siguiendo este enfoque, el sistema de tipos rechazaría cualquier regla que produzca errores de tipo en ejecución, solucionando por tanto el problema de la *descomposición opaca*. Además, basándonos en estas funciones indexadas por tipo y en testigos de tipos (patrones que representan tipos) es posible definir una traducción de programas con clases de tipos alternativa a la traducción clásica que se utiliza diccionarios. Esta nueva traducción resuelve el problema de soluciones perdidas que aparece en PLF en presencia de funciones sobrecargadas indeterministas y sin argumentos cuando se aplica la traducción de diccionarios. Además de recuperar las soluciones perdidas, la traducción alternativa presenta una interesante ganancia en tiempo (entre 1 y 2,3) frente a la traducción usando diccionarios en las pruebas que hemos realizado. Esta ganancia sigue siendo observable incluso cuando hemos aplicado conocidas optimizaciones en ambas traducciones. Aunque la ganancia obtenida en nuestras pruebas es muy esperanzadora, sería necesario implementar ambas traducciones en un sistema lógico-funcional como Toy y realizar pruebas más exhaustivas sobre un conjunto de programas reales para constatar la ganancia real.

Como hemos comentado, el sistema de tipos liberal es correcto desde el punto de vista de los tipos con respecto a una semántica de reescritura indeterminista con *call-time choice* (let-reescritura). Por tanto, sus resultados no son aplicables en un marco lógico-funcional donde se realicen reducciones de estrechamiento para ligar las variables libres de las expresiones. Además, las variables extra, otra característica muy potente de la PLF, han sido excluidas explícitamente de las reglas porque su presencia

invalidaría la preservación de tipos. Para salvar estas carencias, la siguiente y última sección presenta un sistema de tipos para manejo adecuado del estrechamiento y las variables extra.

8. Variables extra y estrechamiento

En este capítulo presentamos el sistema de tipos con soporte para variables extra y estrechamiento aparecido en el artículo *Well-typed Narrowing with Extra Variables in Functional-Logic Programming* [89](A.5). Las demostraciones de los resultados se encuentran en su versión extendida [85](B.2).

8.1. Motivación y objetivos

Como hemos visto en el Ejemplo 2 (página 6), el estrechamiento viola la preservación de tipos, no solo en casos de ligaduras de variables de orden superior sino en escenarios sencillos sin presencia de patrones de orden superior en los cuales se ligan variables de primer orden. En todos estos casos se puede ver que el origen del problema es que el paso de estrechamiento utiliza una sustitución que reemplaza variables de un tipo por patrones de tipos incompatibles. Consideremos los pasos del Ejemplo 2 utilizando la semántica de let-estrechamiento de la Figura 6 (página 27). En el paso $succ\ (F\ zero) \leadsto^{l}_{[F \mapsto and\ false]}\ succ\ false$ se reemplaza la variable F de tipo $nat \to nat$ por el patrón $and\ false$ de tipo $bool \to bool$. De la misma manera, en $and\ true\ X \leadsto^{l}_{[X \mapsto zero, X_1 \mapsto zero]}\ zero$, que utiliza una instancia fresca de la primera regla de and —$and\ true\ X_1 \to X_1$—, se sustituye la variable X de tipo $bool$ por el patrón $zero$, de tipo nat. Por otro lado, las variables extra también dan lugar a la pérdida de la preservación de tipos. Tomemos como caso la función $f \to and\ true\ X$ de tipo $bool$, que tiene la variable extra X de tipo booleano. Podemos realizar el paso de let-estrechamiento $f \leadsto^{l}_{[X_1 \mapsto zero]}\ and\ true\ zero$ (donde X_1 es la variable fresca proveniente de la variante fresca de f) cuya expresión resultante $and\ true\ zero$ está mal tipada. El problema, como ha ocurrido antes, es que se reemplaza una variables booleana por el patrón natural $zero$.

Estas situaciones ya eran detectadas y manejadas en [45], como hemos comentado en la Sección 5.2 (página 35). Para ello prohibían la aparición de variables extra en las reglas. Además, incluían información de tipos en los objetivos CLNC a evaluar y realizaban comprobaciones de tipos cada vez que se producía ligadura de variables libres de orden superior. No era necesario realizar comprobaciones de tipos al ligar variables libres de primer orden ya que las propias reglas del cálculo CLNC codifican el cómputo de unificadores más general, por lo que el paso incorrecto $and\ true\ X \leadsto^{l}_{[X \mapsto zero]}\ zero$ no se podría llevar a cabo en ese cálculo. Sin embargo, la información de tipos incluida en los objetivos debía actualizarse para mantenerse coherente tras cada paso CLNC.

En esta sección proponemos un sistema de tipos adecuado para reducciones en las que se producen ligaduras de variables libres. A diferencia de [45], utilizaremos la semántica de let-estrechamiento, más sencilla y cercana a los cómputos lógico-funcionales que CLNC. Basándonos en este sistema de tipos, y en la noción de *sustituciones bien tipadas*, desarrollaremos la relación $\leadsto^{lwt}$ de *let-estrechamiento bien tipado* que preserva tipos para sustituciones arbitrarias (no restringidas a unificadores más generales) a la vez que soporta variables extra.

Como queda patente en [45], realizar pasos de estrechamiento de manera que se preserven los tipos requiere comprobaciones de tipos en los pasos, y nuestra relación $\leadsto^{lwt}$ no es una excepción. Por tanto desarrollaremos una restricción $\leadsto^{lmgu}$ del cálculo de let-estrechamiento bien tipado que preserva tipos sin necesidad de comprobaciones, cuya demostración se basa en la preservación de tipos de $\leadsto^{lwt}$. Utilizando este let-estrechamiento reducido $\leadsto^{lmgu}$, demostraremos además la preservación de tipos para programas Curry simplificados simulando reducciones de *estrechamiento necesario* [6]. Por último, identificaremos una restricción de programas para los cuales el let-estrechamiento bien tipado $\leadsto^{lwt}$ se comporta igual que el let-estrechamiento reducido $\leadsto^{lmgu}$.

8.2. Sistema de tipos

En esta sección consideraremos una sintaxis de las expresiones y programas similar a la presentada para el let-estrechamiento (ver Figura 4, página 23), e igual por tanto a la considerada en el sistema de tipos liberal de la sección anterior. La única diferencia es que la reglas podrán contener variables extra en sus lados derechos. Con respecto a los tipos, reutilizaremos la noción de *transparencia* presentada en [45] (ver Sección 5.2, página 35), adaptándola a esquemas de tipos. Diremos que un esquema de tipos $\forall\overline{\alpha}.\overline{\tau_m} \to \tau$ es *m-transparente* si $var(\overline{\tau_m}) \subseteq var(\tau)$[39]. También diremos que un patrón t es *transparente* con respecto a un conjunto de suposiciones $\mathcal{A}$ si $t \in \mathcal{DV}$ o $t \equiv h\ \overline{t_n}$, donde $\mathcal{A}(h)$ es n-transparente y $\overline{t_n}$ son patrones transparentes con respecto a $\mathcal{A}$. Por último, una constructora c es *transparente* con respecto a $\mathcal{A}$ si $ar(c) = n$ y $\mathcal{A}(c)$ es n-transparente. También consideraremos una ligera restricción sobre los conjuntos de suposiciones: consideraremos que las suposiciones que acompañan a las variables de datos son siempre tipos simples, es decir, $\mathcal{A}(X) = \tau$. Esta restricción no limitará la expresividad del sistema, ya que consideraremos let-expresiones monomórficas, a la vez que simplifica la presentación. Por último, diremos que un tipo simple τ es básico si no contiene variables de tipos, es decir, si $var(\tau) = \emptyset$.

La Figura 23 contiene las reglas del sistema de tipos $\vdash^e$ con soporte para variables extra. Como se puede ver, contiene una regla para derivar tipos para λ-abstracciones, aunque estas no han sido consideradas en la sintaxis para expresiones. Esto responde

[39] Al tratarse de tipos simples tenemos que el conjunto de variables (var) coincide con el conjunto de variables de tipo libres (ftv), por lo que los usaremos indistintamente.

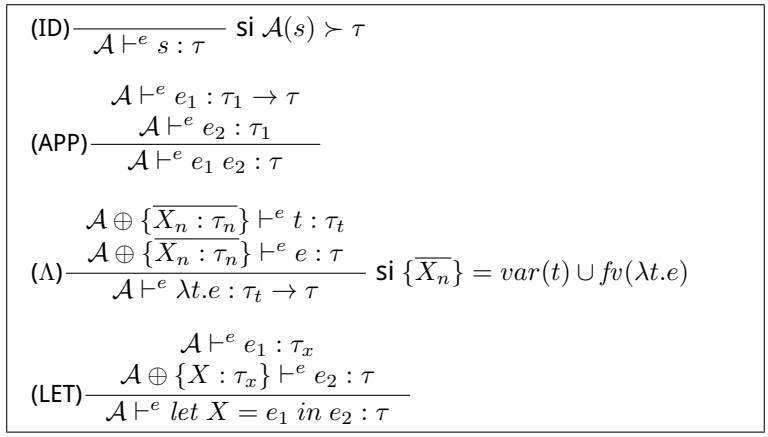


Figura 23: Sistema de tipos con soporte para variables extra

a que, de manera similar al sistema $\vdash^\bullet$ de la Sección 6 (página 49) utilizaremos la derivación de tipos sobre λ-abstracciones en la definición de regla y programa bien tipado. A diferencia de la Sección 6, en esta sección (y en su publicación asociada [89]) decidimos excluirla de la sintaxis de las expresiones e incluirla únicamente en las reglas del sistema de tipos para resaltar que es una construcción que no puede aparecer en los programas ni en las expresiones a evaluar. Las reglas de la Figura 23 son muy similares a las de DM dirigido por la sintaxis (Figura 7-b, página 33), salvo dos diferencias. La primera es el carácter monomórfico de las let-expresiones, pues el tipo inferido para la variable no es generalizado. Esto únicamente persigue simplificar el sistema de tipos, para así poder centrarnos más fácilmente en la cuestión principal a tratar en este capítulo: la problemática del estrechamiento y las variables extra. La segunda diferencia, más importante, es la extensión para cubrir variables extra en la regla (Λ) que trata λ-abstracciones. Para ello, genera suposiciones no solo sobre las variables que aparecen en los patrones —$var(t)$— sino también sobre las variables extra de la λ-abstracción —$ftv(\lambda t.e)$—. Diremos que una expresión e está bien tipada con respecto a $\mathcal{A}$, escrito como $wt^e_{\mathcal{A}}(e)$, si $\mathcal{A} \vdash^e e : \tau$ para algún τ. Además utilizaremos la metavariable $\mathcal{D}$ para referirnos a derivaciones concretas $\mathcal{A} \vdash^e e : \tau$.

De manera similar al resto de sistemas de tipos presentados en esta tesis, es necesario proporcionar una noción explícita de regla y programa bien tipado. Para ello utilizaremos la derivación de tipos de las λ-abstracciones asociadas a las reglas:

Definición 12 (Programa bien tipado, [89](A.5, Def. 3.1)) *Una regla de programa sin argumentos $f \to e$ está bien tipada con respecto a $\mathcal{A}$ si y solo si $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^e e : \tau$, donde $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ y $\overline{\tau_n}$ son algunos tipos simples. Por otra parte, una regla de programa $f\ \overline{p_n} \to e$ (con $n > 0$) está bien tipada con respecto a $\mathcal{A}$ si y solo si $\mathcal{A} \vdash^e \lambda p_1 \ldots \lambda p_n.e : \tau$, con $\mathcal{A}(f) \succ_{var} \tau$. Diremos que un programa está bien tipado con respecto a $\mathcal{A}$, escrito $wt^e_{\mathcal{A}}(\mathcal{P})$, si todas sus reglas están bien tipadas con respecto a $\mathcal{A}$.*

Nótese cómo, a diferencia del sistema $\vdash^{\bullet}$, debemos manejar explícitamente el caso de reglas sin argumentos $f \to e$. Esto es debido a que en esos casos la λ-abstracción asociada a la regla es simplemente el lado derecho e, lo que daría lugar a que las variables extra que apareciese en e no fuesen consideradas. Por ello es necesario manejarlo explícitamente, añadiendo suposiciones para las variables libres $\{\overline{X_n}\} = fv(e)$ que aparecen en el lado derecho.

Antes de presentar la relación de let-estrechamiento bien tipado $\rightsquigarrow^{lwt}$, necesitamos introducir dos nociones nuevas. Como hemos visto, los pasos de let-estrechamiento que no preservaban tipos tenían la característica en común de utilizar sustituciones que no respetaban los tipos, es decir, que reemplazaban variables de un tipo por patrones de otro tipo. Para capturar la idea de sustitución que reemplaza variables por patrones del mismo tipo utilizaremos la noción de *sustitución bien tipada*:

Definición 13 (Sustitución bien tipada, [89](A.5, Def. 3.4)) *Una sustitución de datos θ está bien tipada con respecto a $\mathcal{A}$, escrito $wt^e_{\mathcal{A}}(\theta)$, si $\mathcal{A} \vdash^e X\theta : \mathcal{A}(X)$ para todo $X \in dom(\theta)$.*

Debido a la mencionada restricción impuesta sobre los conjuntos de suposiciones, podemos incorporar $\mathcal{A}(X)$ en la derivación de la definición anterior pues será un tipo simple y no un esquema de tipo. Este tipo de sustituciones es importante ya que, como veremos más adelante, los pasos de let-estrechamiento que utilicen sustituciones bien tipadas preservarán los tipos. Sin embargo, los pasos de let-estrechamiento pueden introducir nuevas variables en la expresión, provenientes tanto de variables extra como variantes frescas de las reglas —introducidas por las reglas (Narr) o (VAct)— o de patrones «inventados» —introducidos por (VBind)—. En consecuencia, será necesario considerar suposiciones de tipos adecuadas sobre estas nuevas variables. Estas suposiciones no son siempre arbitrarias, sino que en muchas ocasiones están unívocamente determinadas por el paso realizado:

Ejemplo 21 (Conjunto de suposiciones asociadas a (Narr), [89](A.5, Ex. 3.5)) *Consideremos la función $f \in FS^1$ con tipo $\forall\alpha.\alpha \to [\alpha]$ definida con la regla $f\ X \to [X, Y]$. Podemos realizar el paso de let-estrechamiento $f\ true \rightsquigarrow^l_{[X_1 \mapsto true]} [true, Y_1]$ usando (Narr) con la variante fresca de la regla $f\ X_1 \to [X_1, Y_1]$. Como la expresión original es $f\ true$, queda claro que X_1 debe tener tipo $bool$ en el nuevo conjunto de suposiciones. Además, Y_1 debe tener el mismo tipo, ya que aparece en la misma lista que X_1. Por lo tanto, el conjunto de suposiciones asociado a este paso concreto será $\{X_1 : bool, Y_1 : bool\}$.*

La siguiente definición establece cuándo un conjunto de suposiciones está asociado a un paso de let-estrechamiento. Obsérvese que en algunos casos puede no haber ningún conjunto de suposiciones asociado a un paso de let-estrechamiento, o haber varios.

Definición 14 (Conjunto de suposiciones asociadas a pasos $\rightsquigarrow^l$, [89](A.5, Def. 3.6)) *Consideremos una derivación de tipos $\mathcal{D}$ para $\mathcal{A} \vdash^e e : \tau$ y un programa bien tipado $\mathcal{P}$ —$wt^e_{\mathcal{A}}(\mathcal{P})$—. Diremos que un conjunto de suposiciones $\mathcal{A}'$ está asociado al paso de let-estrechamiento $e \rightsquigarrow^l_\theta e'$ si y solo si:*

- *$\mathcal{A}' \equiv \emptyset$ y la regla de let-estrechamiento utilizada es (LetIn), (Bind), (Elim), (Flat) o (LetAp).*
- *Si la regla de let-estrechamiento utilizada es (Narr) entonces tenemos que $f\ \overline{t_n} \rightsquigarrow^l_\theta r\theta$ usando una variante fresca de la regla $(f\ \overline{p_n} \to r) \in \mathcal{P}$ y una sustitución θ tal que $(f\ \overline{p_n})\theta \equiv (f\ \overline{t_n})\theta$. Como $\mathcal{D}$ es una derivación de tipos para $\mathcal{A} \vdash^e f\ \overline{t_n} : \tau$, contendrá una derivación $\mathcal{A} \vdash^e f : \overline{\tau_n} \to \tau$ para algunos tipos $\overline{\tau_n}$. Por otro lado la regla $f\ \overline{p_n} \to r$ está bien tipada debido a que $wt^e_{\mathcal{A}}(\mathcal{P})$, por lo que sabemos que existe la derivación:*

$$(\Lambda)\ \frac{\mathcal{A} \oplus \mathcal{A}_1 \vdash^e p_1 : \tau'_1 \quad \cdots \quad (\Lambda)\ \dfrac{\mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash^e p_n : \tau'_n \quad \mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash^e r : \tau'}{\vdots}}{\mathcal{A} \vdash^e \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'}$$

  donde $\overline{\mathcal{A}_n}$ son conjuntos de suposiciones sobre variables introducidos por la regla (Λ) y $\overline{\tau'_n} \to \tau'$ es una variante de $\mathcal{A}(f)$ —el caso para reglas sin argumentos es similar—. Por lo tanto $(\overline{\tau'_n} \to \tau')\pi \equiv \overline{\tau_n} \to \tau$ para alguna sustitución π cuyo dominio son variables frescas de la variante. En este caso el conjunto de suposiciones $\mathcal{A}'$ está asociado al paso (Narr) si $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$.
- *Si el paso de let-estrechamiento usa la regla (VAct) tenemos que $X\ \overline{t_k} \rightsquigarrow^l_\theta r\theta$ utilizando una variante de regla fresca $(f\ \overline{p_n} \to r) \in \mathcal{P}$ y una sustitución θ tal que $(X\ \overline{t_k})\theta \equiv f\ \overline{p_n}\theta$. Como $\mathcal{D}$ es una derivación de tipos $\mathcal{A} \vdash^e X\ \overline{t_k} : \tau$, contendrá una derivación $\mathcal{A} \vdash^e X : \overline{\tau_k} \to \tau$. La regla $f\ \overline{p_n} \to r$ está bien tipada debido a $wt^e_{\mathcal{A}}(\mathcal{P})$, por lo que tenemos una derivación $\mathcal{A} \vdash^e \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'$ como en el caso anterior (de manera similar si la regla no tiene argumentos). Sea $\overline{\tau''_k} \equiv \tau'_{n-k+1} \to \tau'_{n-k+2} \ldots \to \tau'_n$, es decir, los últimos k tipos que aparecen en $\overline{\tau'_n}$. Si $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$ para alguna sustitución π tal que $(\overline{\tau''_k} \to \tau')\pi \equiv \overline{\tau_k} \to \tau$ y $ftv(\mathcal{A}) \cap dom(\pi) = \emptyset$, entonces $\mathcal{A}'$ es un conjunto de suposiciones asociado al paso (VAct). La condición $ftv(\mathcal{A}) \cap dom(\pi) = \emptyset$ es necesaria para evitar que en el conjunto de suposiciones $\mathcal{A}'$ asociado al paso se instancien variables libres del conjunto $\mathcal{A}$ original.*
- *Cualquier $\mathcal{A}' \equiv \{\overline{X_n : \tau_n}\}$ es un conjunto de suposiciones asociado a un paso (VBind) si $\overline{X_n}$ son las variables introducidas por $vran(\theta)$ —que no aparecen en $\mathcal{A}$— y $\overline{\tau_n}$ son tipos simples.*
- *$\mathcal{A}'$ es un conjunto de suposiciones asociado a un paso de let-estrechamiento (Contx) si está asociado al paso interno aplicado.*

Un conjunto de suposiciones $\mathcal{A}'$ está asociado a n pasos de let-estrechamiento ($e_1 \leadsto^l e_2 \ldots \leadsto^l e_{n+1}$) si $\mathcal{A}' \equiv \mathcal{A}'_1 \oplus \mathcal{A}'_2 \ldots \oplus \mathcal{A}'_n$, donde $\mathcal{A}'_i$ es el paso asociado al paso $e_i \leadsto^l e_{i+1}$ y la derivación de tipos $\mathcal{D}_i$ para e_i usando $\mathcal{A} \oplus \mathcal{A}'_1 \ldots \oplus \mathcal{A}'_{i-1}$ ($\mathcal{A}' \equiv \emptyset$ si $n = 0$).

Utilizando las nociones anteriores, se puede definir el let-estrechamiento bien tipado $\leadsto^{lwt}$, que solo utiliza sustituciones bien tipadas:

Definición 15 (Let-estrechamiento bien tipado $\leadsto^{lwt}$, [89](A.5, Def. 3.7))
Consideremos una expresión e, un programa $\mathcal{P}$ y un conjunto de suposiciones $\mathcal{A}$ tal que $wt^e_{\mathcal{A}}(e)$ con una derivación de tipos $\mathcal{D}$ y $wt^e_{\mathcal{A}}(\mathcal{P})$. Entonces $e \leadsto^{lwt}_\theta e'$ es un paso de let-estrechamiento bien tipado si y solo si $e \leadsto^l_\theta e'$ y $wt^e_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, donde $\mathcal{A}'$ es un conjunto de suposiciones asociado $e \leadsto^l_\theta e'$ y $\mathcal{D}$.

Las premisas $wt^e_{\mathcal{A}}(e)$ y $wt^e_{\mathcal{A}}(\mathcal{P})$ son imprescindibles, pues los conjuntos de suposiciones asociados al paso solo están definidos en esos casos. Además, el paso $\leadsto^{lwt}$ no está definido si no existe ningún conjunto de suposiciones asociado. La relación $\leadsto^{lwt}$ es más pequeña que el let-estrechamiento original $\leadsto^l$, ya que impone restricciones sobre las sustituciones obtenidas. No obstante, y a diferencia de $\leadsto^l$, sí goza de preservación de tipos:

Teorema 20 (Preservación de tipos de $\leadsto^{lwt}$, [89](A.5, Th. 3.8)) *Si $wt^e_{\mathcal{A}}(\mathcal{P})$, $e \leadsto^{lwt^*}_\theta e'$ y $\mathcal{A} \vdash^e e : \tau$ entonces $\mathcal{A} \oplus \mathcal{A}' \vdash^e e' : \tau$ y $wt^e_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, donde $\mathcal{A}'$ es un conjunto de suposiciones asociado a la reducción.*

Este teorema es el principal resultado de esta sección, ya que establece de manera clara que una reducción de let-estrechamiento preserva tipos siempre que las sustituciones obtenidas paso a paso estén bien tipadas. Como resultado adicional establece que la sustitución global obtenida estará bien tipada con respecto al conjunto de suposiciones asociado a la reducción. Además este resultado es general en el sentido de que no asume ninguna restricción sobre los patrones en el programa, la transparencia de la constructoras o las variables extra. Gracias a la generalidad de este resultado, es posible utilizarlo para demostrar que relaciones de let-estrechamiento más pequeñas que $\leadsto^{lwt}$ preservan tipos, simplemente demostrando que las sustituciones que generan sus pasos están bien tipadas.

La relación $\leadsto^{lwt}$ preserva tipos, sin embargo, su implementación efectiva requeriría realizar comprobaciones de tipos en cada paso (encerradas en la condición de sustitución bien tipada), algo que en general se pretende evitar en lenguajes con tipado estático del estilo de DM, y en particular en las implementaciones de lenguajes lógico-funcionales considerados en esta tesis. Es por tanto interesante encontrar relaciones más pequeñas que $\leadsto^{lwt}$ que preserven tipos, pero sin necesidad de comprobaciones. Para ello vamos a fijarnos en los casos problemáticos del Ejemplo 2 (página 6).

La reducción $succ\ (F\ zero) \leadsto_{[F \mapsto and\ false]} succ\ false$ nos indica que los pasos que ligan variables libres de orden superior son una fuente de problemas, ya que realizan una búsqueda sobre todos los posibles patrones que permiten aplicar reglas, de los cuales muchos de ellos no preservarán tipos. Esta situación también se detecta en [45]. Por ello, en la relación reducida evitaremos cualquier ligadura de variables de orden superior, eliminando las reglas de let-estrechamiento (VAct) y (VBind). Aunque esto produce un relación más pequeña, sigue teniendo sentido: las expresiones que necesitan utilizar (VAct) o (VBind) para progresar pueden entenderse como expresiones *congeladas* hasta que otro paso ligue esas variables libres de orden superior. Esto es similar al mecanismo de *residuación* [55] utilizado en algunos lenguajes lógico-funcionales como Curry. Un ejemplo de esta situación sería la búsqueda en un espacio de estados. Supongamos que los estados están identificados como números naturales, y que una estrategia es una función que dado un estado devuelve el siguiente en la búsqueda. Los programadores pueden definir diferentes estrategias, aunque para determinados problemas solo algunas de ellas pueden ser admisibles. Para distinguir entre ellas, usan una función $admissible$ que acepta una estrategia y devuelve $true$ si es admisible. Entonces podemos definir una función $next$ que devuelve el siguiente estado siguiendo una estrategia admisible como $next\ St \to if\_then\ (admissible\ F)\ (F\ St)$. Consideremos un programa donde la única estrategia admisible es st_2 (es decir, $admissible\ st_2 \to true$), una estrategia definida como $st_2\ N \to succ\ N$. Entonces el cómputo del siguiente estado admisible a partir de $zero$ sería (por motivos de claridad en la reducción anotamos únicamente la sustitución sobre las variables libres de la expresión):

$$
\begin{array}{l}
next\ zero \\
\leadsto^{l}_{\epsilon}\ if\_then\ (\underline{admissible\ F})\ (F\ zero) \\
\leadsto^{l}_{[F \mapsto st_2]}\ if\_then\ true\ (st_2\ zero) \\
\leadsto^{l}_{\epsilon}\ st_2\ zero \leadsto^{l}_{\epsilon}\ succ\ zero
\end{array}
$$

En la segunda expresión, $F\ zero$ podría haber sido reducida utilizando la regla (VAct) y reemplazando F por cualquier patrón que permita aplicar una regla de programa. Sin embargo, consideramos esta expresión congelada hasta que la reducción instancia F. Por ello la reducción continúa evaluando $admissible\ F$ a $true$, lo que liga F a st_2. Una vez tenemos esta ligadura, la expresión congelada $F\ zero$ se convierte en $st_2\ zero$, que puede ser reducida utilizando (Narr).

Aparte del peligro de las reglas (VAct) y (VBind), utilizar unificadores demasiado específicos en la regla (Narr) también puede romper la preservación de tipos. Esto se observa en el paso $and\ true\ X \leadsto_{[X \mapsto zero]}\ zero$ del Ejemplo 2 (página 6). Por ello en la relación de let-estrechamiento reducida restringiremos los unificadores utilizados por la regla (Narr) a unificadores más generales. Finalmente, el Ejemplo 2 también nos muestra que aun usando unificadores más generales, la preservación de tipos se puede romper si aparecen patrones opacos en los lados izquierdos de las reglas. En conse-

cuencia, restringiremos los patrones que pueden aparecer en los lados izquierdos de las reglas a patrones transparentes.

Teniendo todas estas cuestiones en mente, definiremos la relación $\leadsto^{lmgu}$ de let-estrechamiento reducido como:

Definición 16 (Let-estrechamiento reducido $\leadsto^{lmgu}$, [89](A.5, Def. 3.9)) $e \leadsto_\theta^{lmgu} e'$ *si y solo si* $e \leadsto_\theta^{l} e'$ *usando una regla del let-estrechamiento (Figura 6, página 27) excepto (VAct) y (VBind), y si el paso es del tipo* $f\ \overline{t_n} \leadsto_\theta^{l} r\theta$ *usando (Narr) con la variante fresca de regla* $(f\ \overline{p_n} \to r)$ *entonces* $\theta = mgu(f\ \overline{t_n}, f\ \overline{p_n})$.

Dado que todas las reglas de $\leadsto^{lmgu}$ salvo la de (Narr) generan sustituciones vacías (que trivialmente están bien tipadas), para demostrar que $\leadsto^{lmgu}$ preserva tipos solo sería necesario demostrar que las sustituciones generadas por (Narr) están bien tipadas, pues en ese caso todo paso $\leadsto^{lmgu}$ será un paso $\leadsto^{lwt}$. Para ello necesitaremos considerar la restricción a patrones transparentes en los lados izquierdos de las reglas. Como la regla (Narr) unifica el lado izquierdo de una variante fresca (que contendrá patrones lineales, frescos y transparentes) con una expresión, basta con tener un resultado como el siguiente:

Lema 2 (Unificadores más generales bien tipados, [89](A.5, Lemma 3.10))
Consideremos unos patrones $\overline{p_n}$ *lineales, frescos y transparentes con respecto a* $\mathcal{A}$ *y otros patrones arbitrarios* $\overline{t_n}$ *tales que* $\mathcal{A} \vdash^e p_i : \tau_i$ *y* $\mathcal{A} \vdash^e t_i : \tau_i$ *para algunos* τ_i. *Si* $\theta = mgu(f\ \overline{p_n}, f\ \overline{t_n})$ *entonces* $wt^e_{\mathcal{A}}(\theta)$.

La necesidad de la transparencia queda evidente en la reducción del Ejemplo 2 (página 6) $[f\ (snd\ X), X] \leadsto_{[X \mapsto zero]} [true, zero]$, que no preserva tipos utilizando el unificador más general debido a que la regla $f\ (snd\ zero) \to true$ tiene el patrón opaco $snd\ zero$. Por otro lado, la linealidad de los patrones $\overline{p_n}$ (garantizada por la sintaxis de los programas) también es necesaria. Consideremos el patrón transparente pero no lineal $p \equiv (Y, Y)$, el patrón arbitrario $t \equiv (snd\ X, snd\ true)$ y un conjunto de suposiciones $\mathcal{A}$ conteniendo $\{Y : bool \to bool, X : nat\}$. Es sencillo ver que tanto p como t tienen el mismo tipo $(bool \to bool, bool \to bool)$. El unificador más general de $f\ p$ y $f\ t$ es $\theta \equiv [Y \mapsto snd\ true, X \mapsto true]$, sin embargo, no es una sustitución bien tipada con respecto a $\mathcal{A}$ ya que $\mathcal{A} \not\vdash^e X\theta : \mathcal{A}(X)$, es decir, $\mathcal{A} \not\vdash^e true : nat$.

Del anterior lema se desprende que un paso (Narr) que utilice un unificador más general en un programa con patrones transparentes en los lados izquierdos generará una sustitución que estará bien tipada, por lo tanto tenemos que todo paso $\leadsto^{lmgu}$ es un paso $\leadsto^{lwt}$. En consecuencia podemos demostrar que $\leadsto^{lmgu}$ preserva tipos gracias a la propia preservación de tipos de $\leadsto^{lwt}$ (Teorema 20, página 96):

Teorema 21 (Preservación de tipos de $\leadsto^{lmgu}$, [89](A.5, Th. 3.11)) *Sea* $\mathcal{P}$ *un programa tal que los lados izquierdos de sus reglas contienen solo patrones transparentes. Si* $wt^e_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^e e : \tau$ *y* $e \leadsto_\theta^{lmgu^*} e'$ *entonces* $\mathcal{A} \oplus \mathcal{A}' \vdash^e e' : \tau$ *y* $wt^e_{\mathcal{A} \oplus \mathcal{A}'}(\theta)$, *donde* $\mathcal{A}'$ *es un conjunto de suposiciones asociado a la reducción.*

La relación $\leadsto^{lwt}$ es más general que el cálculo CLNC extendido con tipos de [45], ya que soporta patrones opacos en las reglas y variables extra. Además permite el estrechamiento con sustituciones arbitrarias, en lugar de estar reducido a unificadores más generales. El cálculo CLNC de [45] soporta sentencias de c-convergencia, lo que puede dar lugar a descomposición opaca. En cambio, sus resultados de preservación de tipos únicamente son válidos para reducciones en las que no se realice ningún paso de descomposición opaca, propiedad que es indecidible. En el caso de $\leadsto^{lwt}$ la descomposición opaca no puede aparecer ya que no existen reglas del let-estrechamiento que calculen la igualdad de patrones. Además, a diferencia del sistema de tipos liberal de la Sección 7 (página 65), en el sistema de tipos presentado en esta sección no sería posible definir la igualdad por reglas, ya que estaría mal tipada.

El cálculo $\leadsto^{lmgu}$, a diferencia del cálculo CLNC de [45] y $\leadsto^{lwt}$, no realiza pasos en los que se «inventan» patrones para las variables de orden superior —pues omite las reglas del let-estrechamiento (Vact) y (VBind)— aunque sí que realiza ligadura de variables de orden superior, como se ha visto en el anterior ejemplo de la búsqueda en un espacio de estados. Al igual que el cálculo CLNC de [45] no soporta patrones opacos en las reglas y considera únicamente unificadores más generales al aplicar pasos de estrechamiento con reglas de programa, aunque sí que soporta variables extra. Sin embargo, la característica que más le diferencia de $\leadsto^{lwt}$ y el cálculo CLNC de [45] es que $\leadsto^{lmgu}$ preserva tipos de manera automática, sin necesidad de realizar comprobaciones de tipos en cada paso. Esto lo hace una relación muy interesante para ser usada en sistemas estáticamente tipados, que carecen de información de tipos durante la ejecución. En los próximos dos apartados veremos la utilidad de $\leadsto^{lmgu}$ para demostrar la preservación de tipos de una simulación de estrechamiento necesario sobre un lenguaje Curry simplificado y propondremos algunas restricciones sobre los programas bajo las cuales $\leadsto^{lmgu}$ es completa con respecto a $\leadsto^{lwt}$ utilizando unificadores más generales.

8.3. Preservación de tipos para estrechamiento necesario

En esta sección consideraremos programas Curry simplificados, omitiendo características de ese lenguaje como las restricciones o la entrada/salida. Por ello, como es usual en Curry [54], consideramos únicamente reglas cuyos patrones son de primer orden y constructoras transparentes (lo que implica que todos los patrones formados serán transparentes). Estos programas se evalúan utilizando la estrategia de estrechamiento necesario [6] (una de las más usadas en el ámbito lógico-funcional), realizando residuación para las aplicaciones de variables —lo que se simula omitiendo las reglas del let-estrechamiento (VAct) y (VBind)—.

Para demostrar que la simulación de evaluación utilizando estrechamiento necesario sobre programas Curry simplificados preserva los tipos utilizaremos un enfoque transformacional. Para ello haremos uso de dos transformaciones muy conocidas. La

primera [5] sirve para transformar un programa Curry simplificado en un programa *inductivamente secuencial con solapamiento* [4] (OIS según sus siglas en inglés). Para este tipo de programas, existe un árbol definicional con solapamiento para cada función, que es una estructura de datos arborescente que codifica la demanda de los distintos argumentos de la función que se desprende de los lados izquierdos de las reglas. Los árboles definicionales con solapamiento son como los *árboles definicionales* originales [3], con la diferencia de que cada hoja puede tener asociadas varias reglas o, dicho de otro modo, tiene asociada una sola regla cuyo lado derecho está formado por alternativas indeterministas. Estos árboles definicionales son los que dirigen la estrategia, consiguiendo que se realicen únicamente las reducciones «necesarias» para avanzar. La segunda transformación [156] toma un programa OIS y lo transforma a *formato uniforme*. En este tipo de programas los lados izquierdos de las reglas son del tipo $f\ \overline{X}$ o $f\ \overline{X}\ (c\ \overline{Y})\ \overline{Z}$, es decir, contienen a lo sumo un símbolo de constructora. Intuitivamente, lo que hace esta transformación es procesar las reglas para que la demanda expresada en los árboles definicionales con solapamiento quede patente directamente en los lados izquierdos de las reglas.

Aunque la prueba de que ambas transformaciones preservan la semántica de los programas [5, 156] se ha dado en el contexto de la reescritura de términos, no es difícil convencerse de que dicha preservación también se extiende al contexto de *call-time choice* de HO-CRWL. Además, el estrechamiento usando unificadores más generales sobre el programa en formato uniforme es completo con respecto al estrechamiento necesario sobre el programa original[156], hecho del que también estamos bastante seguros de que se extienda al marco del let-estrechamiento. Por tanto, consideramos que las reducciones $\leadsto^{lmgu}$ sobre los programas transformados serán adecuadas con respecto a una evaluación utilizando estrechamiento necesario sobre los programas Curry simplificados originales. Para demostrar que dichas reducciones preservan los tipos necesitaremos demostrar que las dos transformaciones producen programas bien tipados, permitiéndonos aplicar el Teorema 21 de preservación de tipos de $\leadsto^{lmgu}$.

Como hemos comentado, para realizar la transformación de programas Curry simplificados arbitrarios a programas OIS utilizaremos una transformación similar a la que se encuentra en [5]. Existen otras transformaciones para realizar este objetivo, como por ejemplo [33], pero nos hemos basado en la de [5] debido a su simplicidad. La transformación de la Definición 17 procesa las diferentes funciones de manera independiente: toma el conjunto $\mathcal{P}_f$ de reglas de la función f y devuelve una pareja formada por las reglas transformadas y el conjunto de suposiciones sobre las funciones introducidas.

Definición 17 (Transformación a programa OIS, [89](A.5, Def. 5.1))
Sea $\mathcal{P}_f \equiv \{f\ \overline{t_n^1} \to e^1, \ldots, f\ \overline{t_n^m} \to e^m\}$ *el conjunto de* m *reglas de programa para la función* f *tal que* $wt^e_{\mathcal{A}}(\mathcal{P}_f)$*. Si* f *es una función que ya es inductivamente secuencial con solapamiento,* $OIS(\mathcal{P}_f) = (\mathcal{P}_f, \emptyset)$*. En otro caso* $OIS(\mathcal{P}_f) = (\{f_1\ \overline{t_n^1} \to e^1, \ldots, f_m\ \overline{t_n^m} \to e^m, f\ \overline{X_n} \to f_1\ \overline{X_n}?\ldots?f_m\ \overline{X_n}\}, \{\overline{f_m : \mathcal{A}(f)}\})$*, donde* $?$ *es la función de elección indeter-*

minista definida con las reglas $\{X?Y \to X, X?Y \to Y\}$.

Intuitivamente la transformación lo que hace es generar una nueva función f cuyo único lado derecho es la alternativa indeterminista de todas las reglas, cambiando su nombre a f_i. Tanto la nueva función f como las antiguas f_i tienen el mismo tipo que la función f original: $\mathcal{A}(f)$. Veamos un ejemplo de esta traducción.

Ejemplo 22 (Ejemplo de la transformación OIS) *Consideremos una función que no es inductivamente secuencial con solapamiento como la función* $insert$ *del Ejemplo 3 (página 15), que inserta de manera indeterminista un elemento en una lista. Por lo tanto* $\mathcal{P} \equiv \{insert\ X\ Ys \to (X : Ys), insert\ X\ (Y : Ys) \to (Y : insert\ X\ Ys)\}$ *y* $\mathcal{A}(insert) = \forall\alpha.\alpha \to [\alpha] \to [\alpha]$. *Entonces la transformación produciría* $OIS(\mathcal{P}) = (\mathcal{P}', \mathcal{A}')$, *donde:*

- $\mathcal{P}' \equiv \{insert\ X_1\ X_2 \to insert_1\ X_1\ X_2\ ?\ insert_2\ X_1\ X_2,$
  $insert_1\ X\ Ys \to (X : Ys),$
  $insert_2\ X\ (Y : Ys) \to (Y : insert\ X\ Ys)\}$
- $\mathcal{A}' \equiv \{insert_1 : \forall\alpha.\alpha \to [\alpha] \to [\alpha], insert_2 : \forall\alpha.\alpha \to [\alpha] \to [\alpha]\}$

Lo importante de la transformación OIS es que preserva los tipos, es decir, el programa resultante de la transformación está bien tipado si el original lo estaba.

Teorema 22 (Preservación de tipos de OIS, [89](A.5, Th. 5.2))
Sea $\mathcal{P}_f$ *un conjunto de reglas de programa para la misma función* f *tales que* $wt^e_{\mathcal{A}}(\mathcal{P}_f)$. *Si* $OIS(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ *entonces* $wt^e_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$.

Es sencillo observar que el resultado anterior también sirve para programas con varias funciones. Nótese que para nuestros propósitos cualquier otra transformación a programas inductivamente secuenciales con solapamiento sería válida, siempre que preservase los tipos.

Para la transformación de programas inductivamente secuenciales con solapamiento a formato uniforme utilizaremos una transformación similar a la de [156] pero extendida para que genere suposiciones de tipo sobre las nuevas funciones creadas. Como antes, la transformación procede función a función, tomando todas las reglas de cada una de ellas.

Definición 18 (Transformación a formato uniforme, [89](A.5, Def. 5.3))
Consideremos el programa inductivamente secuencial con solapamiento $\mathcal{P}_f \equiv \{f\ \overline{t^1_n} \to e^1, \ldots, f\ \overline{t^m_n} \to e^m\}$ *compuesto por* m *reglas de programa para la función* f *tal que* $wt^e_{\mathcal{A}}(\mathcal{P}_f)$. *Si* $\mathcal{P}_f$ *ya está en formato uniforme entonces* $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \emptyset)$. *En otro caso, tomemos la* posición uniformemente demandada[40] o *y partamos* $\mathcal{P}_f$ *en* r *conjuntos*

[40] Una posición es uniformemente demandada si todas las reglas de $\mathcal{P}_f$ tienen una constructora en esa posición. Esta posición siempre existirá ya que $\mathcal{P}_f$ es un programa inductivamente secuencial con solapamiento [4].

$\overline{\mathcal{P}_r}$ *conteniendo las reglas de* $\mathcal{P}_f$ *con la misma constructora en la posición* o*. Entonces* $\mathcal{U}(\mathcal{P}_f) = (\bigcup_{i=1}^{r} \mathcal{P}'_i \cup \mathcal{P}'', \bigcup_{i=1}^{r} \mathcal{A}'_i \cup \mathcal{A}'')$ *donde:*

- $\mathcal{U}(\mathcal{P}_i^o) = (\mathcal{P}'_i, \mathcal{A}'_i)$*.*
- c_i *es la constructora en la posición* o *de las reglas de* $\mathcal{P}_i$*, con* $ar(c_i) = k_i$*.*
- $\mathcal{P}_i^o$ *es el resultado de sustituir el símbolo de función* f *en* $\mathcal{P}_i$ *por* $f_{(c_i,o)}$ *y aplanar los patrones de la posición* o *de las reglas, es decir,* $f\ \overline{t_j}\ (c_i\ \overline{t'_{k_i}})\ \overline{t''_l} \to e$ *se reemplaza por* $f_{(c_i,o)}\ \overline{t_j}\ \overline{t'_{k_i}}\ \overline{t''_l} \to e$*.*
- $\mathcal{P}'' \equiv \{f\ \overline{X_j}\ (c_1\ \overline{Y_{k_1}})\ \overline{Z_l} \to f_{(c_1,o)}\ \overline{X_j}\ \overline{Y_{k_1}}\ \overline{Z_l}, f\ \overline{X_j}\ (c_2\ \overline{Y_{k_2}})\ \overline{Z_l} \to f_{(c_2,o)}\ \overline{X_j}\ \overline{Y_{k_2}}\ \overline{Z_l}, \ldots, f\ \overline{X_j}\ (c_r\ \overline{Y_{k_r}})\ \overline{Z_l} \to f_{(c_r,o)}\ \overline{X_j}\ \overline{Y_{k_r}}\ \overline{Z_l}\}$*, con* $\overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l}$ *variables frescas distintas tales que* $j + l + 1 = n$*.*
- $\mathcal{A}'' \equiv \{f_{(c_1,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau'_{k_1}} \to \overline{\tau_l} \to \tau, \ldots, f_{(c_r,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau'_{k_r}} \to \overline{\tau_l} \to \tau\}$ *donde* $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$ *y* $\mathcal{A} \oplus \{\overline{Y_{k_i} : \tau'_{k_i}}\} \vdash^e c_i\ \overline{Y_{k_i}} : \tau'$*. Como las constructoras* c_i *son transparentes, estos* $\overline{\tau'_{k_i}}$ *existen y son únicos.*

La idea subyacente a esta traducción es generar reglas cuyos lados izquierdos solo contengan una constructora c situada en la posición uniformemente demandada o. El lado derecho de esas reglas contendrá una llamada a la función $f_{(c,o)}$, que solo considera aquellas reglas que tenían la constructora c en la posición o y en la que se habrá «consumido» dicha constructora. Esto se puede ver mejor en el siguiente ejemplo:

Ejemplo 23 (Ejemplo de la traducción $\mathcal{U}$) *Consideremos una función inductivamente secuencial con solapamiento como la función* leq *del Ejemplo 3 (página 15), que compara números naturales. Por lo tanto* $\mathcal{P} \equiv \{leq\ zero\ Y \to true, leq\ (succ\ X)\ Y \to false, leq\ (succ\ X)\ (succ\ Y) \to leq\ X\ Y\}$ *y* $\mathcal{A}(leq) = nat \to nat \to nat$*. Entonces la transformación a formato uniforme producirá* $\mathcal{U}(\mathcal{P}) = (\mathcal{P}', \mathcal{A}')$*, donde:*

- $\mathcal{P}' \equiv \{leq\ zero\ Y \to leq_{(zero,1)}\ Y,$
  $leq\ (succ\ X)\ Y \to leq_{(succ,1)}\ X\ Y,$
  $leq_{(zero,1)}\ Y \to true,$
  $leq_{(succ,1)}\ X\ zero \to false,$
  $leq_{(succ,1)}\ X\ (succ\ Y) \to leq\ X\ Y\}$
- $\mathcal{A}' \equiv \{leq_{(zero,1)} : nat \to nat, leq_{(succ,1)} : nat \to nat \to nat\}$

Como se ha comentado antes con la transformación OIS, la transformación $\mathcal{U}$ también es válida para programas con varias funciones, solo es necesario aplicarla a todas ellas. Además, la transformación $\mathcal{U}$ preserva tipos:

Teorema 23 (Preservación de tipos de $\mathcal{U}(\mathcal{P}_f)$, [89](A.5, Th. 5.4)) *Sea* $\mathcal{P}_f$ *el programa formado por reglas de la misma función inductivamente secuencial con solapamiento* f *tal que* $wt^e_{\mathcal{A}}(\mathcal{P}_f)$*. Si* $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ *entonces* $wt^e_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$*.*

Dado que tanto OIS como $\mathcal{U}$ preservan tipos, tenemos que si aplicamos ambas a un programa Curry simplificado $\mathcal{P}$ bien tipado —$wt^e_{\mathcal{A}}(\mathcal{P})$— obtendremos un programa en formato uniforme $\mathcal{P}'$ bien tipado —$wt^e_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$, donde $\mathcal{A}'$ son las suposiciones de tipos generadas por las transformaciones—. Si consideramos una expresión bien tipada $\mathcal{A} \vdash^e e : \tau$, una reducción $e \leadsto^{lmgu^*}_{\theta} e'$ simulará una evaluación mediante la estrategia de estrechamiento necesario y utilizando residuación. Además, gracias al Teorema 21 (página 98) los tipos serán preservados, es decir, $\mathcal{A} \oplus \mathcal{A}' \oplus \mathcal{A}'' \vdash^e e' : \tau$ y $wt^e_{\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}''}(\theta)$, siendo $\mathcal{A}''$ el conjunto de suposiciones asociado a la reducción.

8.4. Reducciones de estrechamiento sin aplicaciones de variables

Como hemos visto, la relación $\leadsto^{lmgu}$ es menos general que $\leadsto^{lwt}$ porque omite las reglas (VAct) y (VBind) para generar ligaduras de orden superior y porque solo considera unificadores más generales. No obstante, en el *Lifting Lemma* de [92] se prueba que la restricción del let-estrechamiento $\leadsto^{l}$ que solo utiliza unificadores más generales es completa con respecto a HO-CRWL. Por tanto, tenemos la firme creencia de que la restricción de $\leadsto^{lwt}$ que solo utiliza unificadores más generales también será completa con respecto al cálculo de soluciones bien tipadas. Por ello en esta sección nos centraremos en encontrar las condiciones para las cuales $\leadsto^{lmgu}$ es completo con respecto a $\leadsto^{lwt}$ restringido a unificadores más generales, ya que en esos programas se preservará tipos sin necesidad de comprobaciones en cada paso.

Para conseguir esto será necesario encontrar una clase de programas y de expresiones a evaluar en la que se asegure que ni (VAct) ni (VBind) son usados. Sin embargo, la caracterización de esta clase de programas es más complicada de lo que puede parecer a primera vista. Una primera idea es que en expresiones que no contienen variables libres de primer orden (es decir, con un tipo funcional $\tau \to \tau'$) no se pueden usar ni (VAct) ni (VBind). Esto es cierto, como prueba el siguiente lema:

Lema 3 (Ausencia de variables de orden superior, [89](A.5, Lemma 4.1)) *Sea e una expresión tal que $wt^e_{\mathcal{A}}(e)$ y para cada variable $X_i \in fv(e)$, $\mathcal{A}(X_i)$ no es de tipo funcional. Entonces ningún paso $e \leadsto^{l}_{\theta} e'$ puede usar (VAct) o (VBind).*

El problema con el anterior resultado es que la evaluación de este tipo de expresiones puede introducir variables libres de orden superior, incluso cuando esas variables no aparecen como variables extra en las reglas.

Ejemplo 24 (Aparición de variables de orden superior, [89](A.5, Ex 4.2))
Consideremos la constructora bfc de tipo $bfc : (bool \to bool) \to BoolFunctContainer$ y la función f con tipo $f : BoolFunctContainer \to bool$ definida como $\{f\ (bfc\ F) \to F\ true\}$. Podemos realizar el paso de let-estrechamiento utilizando la regla (Narr)

$$\underline{f\ X} \leadsto^{lmgu}_{\theta} F_1\ true$$

donde $\theta \equiv [X \mapsto bfc\ F_1] = mgu(f\ X, f\ (bfc\ F_1))$*. La variable libre* F_1 *que se ha introducido es de orden superior, en cambio, la única variable de la expresión original tenía el tipo* $BoolFunctContainer$*. Además el programa no contenía ninguna variable extra.*

Este ejemplo nos muestra que no solo hay que evitar las variables libres de orden superior sino que también hay que evitar las variables libres de tipo *inseguro* como $BoolFunctContainer$. La razón es que los patrones de tipo inseguro pueden contener variables de orden superior, y un paso utilizando (Narr) puede unificar un patrón de tipo inseguro con una variable de tipo inseguro, introduciendo variables libres de orden superior. Para formalizar esta idea definiremos el conjunto de *tipos inseguros* como aquellos para los cuales se puede formar un patrón que contenga una variable de orden superior:

Definición 19 (Tipos inseguros, [89](A.5, Def 4.3)) *El conjunto de tipos inseguros con respecto a un conjunto de suposiciones* $\mathcal{A}$*, escrito* $UTypes_{\mathcal{A}}$*, se define como el mínimo conjunto de tipos simples tales que:*

1. *Los tipos funcionales (*$\tau \to \tau'$*) están en* $UTypes_{\mathcal{A}}$*.*
2. *Un tipo simple* τ *está en* $UTypes_{\mathcal{A}}$ *si existe algún patrón* $t \in Pat$ *con* $\{\overline{X_n}\} = var(t)$ *tal que:*
   a) $t \equiv \mathcal{C}[X_i]$ *con* $\mathcal{C} \neq [\,]$*.*
   b) $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$*, para algunos tipos* $\overline{\tau_n}$*.*
   c) $\tau_i \in UTypes_{\mathcal{A}}$*.*

Si un tipo τ *no está en* $UTypes_{\mathcal{A}}$ *diremos que es un* tipo seguro *con respecto a* $\mathcal{A}$*.*

Por definición una variable libre de orden superior es de tipo inseguro. Sin embargo, impedir la aparición de variables libres de tipo inseguro tampoco es suficiente:

Ejemplo 25 (Aparición de variables de tipo inseguro, [89](A.5, Ex 4.4))
Consideremos los símbolos del Ejemplo 24 y una nueva función $g \to X$ *de tipo* $g : \forall\alpha.\alpha$*. La variable extra tiene el tipo polimórfico* α*, así que es segura. La expresión* $(f\ g)$ *no contienen variables inseguras, sin embargo, estas aparecen:*

$$f\ \underline{g} \leadsto^{lmgu}_{\epsilon} \underline{f\ X_1} \leadsto^{lmgu}_{[X_1 \mapsto bfc\ F_1]} F_1\ true$$

La variable libre X_1 *introducida tiene tipo* $BoolFunctContainer$*, que es un tipo inseguro que da lugar a la aparición de una aplicación de variable.*

El anterior ejemplo muestra que no solo las variables de tipo inseguro debe ser evitadas, sino cualquier expresión que pueda ser reducida a una expresión de tipo inseguro (como g).

$$(\Lambda^r)\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^e t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \{\overline{Y_k : \tau'_k}\} \vdash^e e : \tau}{\mathcal{A} \vdash^e \lambda^r t.e : \tau_t \to \tau}$$

donde $\{\overline{X_n}\} = var(t)$, $\{\overline{Y_k}\} = fv(\lambda^r t.e)$ tal que $\overline{\tau'_k}$ son tipos básicos y seguros con respecto a $\mathcal{A}$.

Figura 24: Regla de tipado para las λ-abstracciones restringidas

Basándonos en las anteriores ideas vamos a desarrollar una noción de programa restringido en el que se evite la aparición de variables libres de tipo inseguro, y por tanto la aplicación de variables que permite aplicar (Vact) o (VBind). Para ello obligaremos a las expresiones a evaluar a no contener variables libres de tipo inseguro. Con respecto a los programas obligaremos a que las variables extra de sus reglas sean de tipo básico (sin variables de tipos) y seguro. Al forzar que las variables extra tengan tipo básico solventamos el problema de las variables extra polimórficas, que como se ha visto en el Ejemplo 25 pueden tomar tipos inseguros.

Para imponer las citadas restricciones sobre los programas definiremos una nueva noción de *programa bien tipado restringido*. Esta noción está definida de manera similar a la de la Definición 12 (página 93) pero utilizando λ-abstracciones restringidas $\lambda^r t.e$ cuyo tipo se deriva con la regla de la Figura 24.

Definición 20 (Programa bien tipado restringido, [89](A.5, Def. 4.5))
Una regla de programa $f \to e$ está bien tipada restringida *con respecto a $\mathcal{A}$ si y solo si $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$, donde $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ y $\overline{\tau_n}$ son tipos simples básicos y seguros con respecto a $\mathcal{A}$. Una regla de programa $(f\ \overline{p_n} \to e)$ (con $n > 0$) está bien tipada restringida con respecto a $\mathcal{A}$ si y solo si $\mathcal{A} \vdash \lambda^r p_1 \ldots \lambda^r p_n.e : \tau$ con $\mathcal{A}(f) \succ_{var} \tau$. Un programa $\mathcal{P}$ está* bien tipado restringido *con respecto a $\mathcal{A}$, escrito $wt^r_{\mathcal{A}}(\mathcal{P})$, si todas sus reglas están bien tipadas restringidas con respecto a $\mathcal{A}$.*

Nótese que $wt^r_{\mathcal{A}}(\mathcal{P})$ implica $wt^e_{\mathcal{A}}(\mathcal{P})$. La noción de paso $\leadsto^{lwt}$ solo tiene sentido para programas bien tipados $wt^e_{\mathcal{A}}(\mathcal{P})$. Al considerar la noción más restrictiva de programas bien tipados restringidos $wt^r_{\mathcal{A}}(\mathcal{P})$ estamos utilizando implícitamente una variante $\leadsto^{lwt}$ que es ligeramente más pequeña que la que aparece en la Definición 12 (página 93). Esta variante también preserva los tipos. Aunque los resultados que siguen son solo válidos para esta variante siguen siendo relevantes.

La propiedad clave de los programas bien tipados restringidos es que no generan variables libres de tipo inseguro si comienzan con una expresión que no las tenía.

Lema 4 (Ausencia de variables inseguras, [89](A.5, Lemma 4.6))
Sea e una expresión sin contener variables libres de tipo inseguro con respecto a $\mathcal{A}$, y sea $\mathcal{P}$ un programa tal que $wt^r_{\mathcal{A}}(\mathcal{P})$. Si $e \leadsto^{lwt^*}_{\theta} e'$ entonces e' no contendrá variables

libres de tipos inseguro con respecto a $\mathcal{A} \oplus \mathcal{A}'$, donde $\mathcal{A}'$ es un conjunto de suposiciones asociado a la reducción.

El uso de unificadores más generales en $\leadsto^{lwt}$ no es necesario en el lema anterior, ya que la ausencia de variables libres de tipo inseguro se garantiza por la sustitución bien tipada implícita en la definición de $\leadsto^{lwt}$. Basándonos en el Lema 4 es sencillo probar que $\leadsto^{lmgu}$ es completo con respecto a la restricción de $\leadsto^{lwt}$ a unificadores más generales, ya que la ausencia de aplicación de variables libres de orden superior evita que las reducciones $\leadsto^{lwt}$ puedan usar (VAct) o (VBind).

Teorema 24 (Completitud de $\leadsto^{lmgu}$ con respecto a $\leadsto^{lwt}$, [89](A.5, Th. 4.7))
Sea e una expresión que no contiene variables libres de tipo inseguro con respecto a $\mathcal{A}$, y sea $\mathcal{P}$ un programa tal que $wt^r_{\mathcal{A}}(\mathcal{P})$. Si $e \leadsto^{lwt^*}_{\theta} e'$ usando unificadores más generales en cada paso entonces $e \leadsto^{lmgu^*}_{\theta} e'$.

La relación $\leadsto^{lmgu}$ es completa con respecto a $\leadsto^{lwt}$ considerando programas bien tipados restringidos. Por lo tanto, para los programas bien tipados restringidos se preservan los tipos sin necesidad de comprobaciones (pues se usa $\leadsto^{lmgu}$) a la vez que no se pierde ninguna solución con respecto a $\leadsto^{lwt}$. Sin embargo, esta familia de programas deja fuera cualquier función polimórfica que utilice variables extra, como

$$last\ X \to if\_then\ (Xs == Zs\ ++\ [E])\ E$$
$$sublist\ Xs\ Ys \to if\_then\ (Us\ ++\ Xs\ ++\ Zs == Ys)\ true$$

que calcula el último elemento de una lista y comprueba si Xs es una sublista de Ys, respectivamente. Aunque no todas las funciones utilizando variables extra quedan fuera (consideremos por ejemplo la función $even$ del Ejemplo 9, página 26), la noción de programa bien tipado restringida es demasiado restrictiva y debería ser relajada para considerarse una noción útil en el ámbito lógico-funcional, lo que constituye sin duda una interesante cuestión de trabajo futuro.

8.5. Conclusiones

En esta sección hemos abordado el problema de preservación de tipos en reducciones de let-estrechamiento, considerando variables extra. Hemos demostrado que los tipos se preservan durante este tipo de reducciones siempre que las sustituciones generadas estén bien tipadas (englobado en la relación $\leadsto^{lwt}$). Este resultado no impone ninguna restricción adicional sobre las sustituciones generadas, que pueden ser unificadores más generales o sustituciones arbitrarias. La relación $\leadsto^{lwt}$ es más general que el cálculo CLNC de [45], que aunque también preserva tipos considera únicamente unificadores más generales y no soporta variables extra. También es más general que [45] en el sentido que no impone restricciones sobre los programas, tales como contener únicamente patrones transparentes o constructoras que no contengan argumentos

funcionales. Debido a que el sistema de tipos considerado soporta funciones polimórficas, también constituye una interesante mejora sobre [9], que únicamente considera funciones monomórficas.

A diferencia de [9], la relación $\leadsto^{lwt}$ necesita comprobaciones de tipo en cada paso debido a que exige sustituciones bien tipadas. Esto es similar a lo que realiza el cálculo CLNC de [45], que realiza comprobaciones de tipos en aquellos pasos que ligan variables de orden superior, además de actualizar y mantener consistente el entorno de tipos del objetivo tras cada paso. Para evitar este tipo de comprobaciones hemos desarrollado una relación de let-estrechamiento $\leadsto^{lmgu}$ más pequeña que $\leadsto^{lwt}$ que no necesita comprobaciones de tipo en cada paso para preservar tipos. Esta relación omite las reglas (VAct) y (VBind), que generan ligaduras para variables de orden superior, y considera únicamente unificadores más generales. De manera similar a [45] exige patrones transparentes en las reglas para preservar tipos, aunque sigue soportando variables extra.

Basándonos en la preservación de tipos de la relación $\leadsto^{lmgu}$, hemos demostrado que una evaluación de programas Curry simplificados utilizando estrechamiento necesario y residuación preserva los tipos. Para ello hemos utilizado dos transformaciones: la primera transforma programas Curry simplificados a programas inductivamente secuenciales con solapamiento [4], y la segunda transforma este tipo de programas en programas en formato uniforme [156]. Sobre los programas obtenidos, las reducciones $\leadsto^{lmgu}$ simulan la evaluación utilizando estrechamiento necesario y residuación sobre los programas originales. Como ambas transformaciones preservan tipos el programa transformado estará bien tipado, por lo que $\leadsto^{lmgu}$ preservará los tipos.

Aparte, hemos definido una clase de programas bajo los cuales ninguna evaluación utilizará las reglas (VAct) o (VBind). Para estos programas, $\leadsto^{lmgu}$ será completo con respecto a $\leadsto^{lwt}$ restringido a usar unificadores más generales. Para la definición de esta clase hemos usado la noción de tipos inseguros, aquellos tipos para los cuales existe un patrón de ese tipo conteniendo variables de orden superior. Utilizando esta noción, hemos definido la clase de programas como aquellos cuyas variables extra tienen tipo seguro y básico. Bajo esta clase de programas se garantiza que no se utilizarán (VAct) ni (VBind), ya que nunca podrá aparecer una aplicación de variable. Sin embargo, esta clase excluye todas aquellas funciones que utilicen variables extra polimórficas, aunque no generen problemas de tipos, por lo que consideramos que es una clase de problemas demasiado restrictiva para la programación lógico-funcional que debería ser relajada en un futuro.

Como último comentario, advertir que el sistema de tipos de esta sección no solo soporta de manera segura las variables extra y el estrechamiento, sino que también evita problemas como el *casting* polimórfico. A diferencia del sistema $\vdash^{\bullet}$, donde la seguridad de los patrones de orden superior se consigue por medio de las variables opacas y críticas, en este sistema la seguridad se desprende directamente de la utilización de sustituciones bien tipadas. Esto se ve claramente en la reducción del Ejemplo 1

(página 4). Un paso $\leadsto^{lwt}$ de let-estrechamiento bien tipado

$$not\ (\underline{unpack\ (snd\ [\,])}) \leadsto^{lwt}_{[X_1 \mapsto [\,]\,]}\ not\ [\,]$$

que utiliza la variante fresca de la regla $unpack\ (snd\ X_1) \to X_1$ no sería válido. La razón es que el único conjunto de suposiciones asociado al paso es $\mathcal{A}' \equiv \{X_1 : bool\}$, con respecto al cual la sustitución $[X_1 \mapsto [\,]\,]$ estaría mal tipada. Por la misma razón, cualquier paso de $\leadsto^{lwt}$ que ponga en peligro la preservación de tipos debido a la opacidad de los patrones de orden superior será un paso inválido. Estas situaciones ni siquiera están contempladas en $\leadsto^{lmgu}$ ya que su resultado de preservación de tipos supone la ausencia de patrones opacos. Con respecto a la descomposición opaca, esta no está soportada por las reglas del let-estrechamiento, ni se podría definir mediante reglas y resultar en un programa bien tipado. Por tanto, la descomposición opaca no afecta a este sistema de tipos porque no se puede llevar a cabo ningún cómputo de igualdad estructural. Sin embargo, la descomposición opaca podría producirse en sistemas lógico-funcionales que adoptaran este sistema de tipos y que además utilizasen una primitiva *ad-hoc* para la igualdad estructural, pues el sistema de tipos no la detectaría.

Parte IV
Conclusiones y trabajo futuro

9. Conclusiones

En esta tesis se han presentado tres sistemas de tipos que manejan de manera segura desde el punto de vista de los tipos diversos aspectos de los programas lógico funcionales que no estaban contemplados (o no de manera plenamente satisfactoria) en los sistemas actuales. Como ya introdujimos en la Sección 1 (página 3), los principales aspectos que causan errores de tipos son:

- El incorrecto tratamiento de la opacidad creada por los patrones de orden superior, que conduce a situaciones indeseables de pérdida de preservación de tipos, como en el ejemplo del *casting* polimórfico.
- La descomposición opaca, que se produce en presencia de igualdad estructural y patrones de orden superior o constructoras existenciales.
- Las variables extra, que a pesar de ser un recurso altamente expresivo de la programación lógico-funcional suelen ser omitidas de los resultados sobre sistemas de tipos.

Aparte de tratar estas situaciones problemáticas, también hemos explorado la posibilidad de relajar la noción clásica de programa bien tipado (heredada de DM) para

considerar más programas bien tipados, asegurando a la vez la corrección del sistema de tipos con respecto a la evaluación.

En la Sección 6 —[84](A.1), [86](B.1)— hemos presentado el sistema $\vdash^{\bullet}$, que proporciona un tratamiento seguro de los patrones de orden superior en los lados izquierdos de las reglas, evitando el *casting* polimórfico. Para ello hemos seguido una aproximación basada en variables opacas y críticas. Una variable de datos es opaca en un patrón si su tipo no se puede conocer a partir del tipo del patrón. Por otro lado, una variable es crítica en una regla si es opaca en un patrón del lado izquierdo y además aparece en el lado derecho. El sistema $\vdash^{\bullet}$ prohíbe la aparición de las variables críticas en las reglas, por lo que el *casting* polimórfico (y errores similares) no pueden darse ya que el tipo de todas las variables es conocido a la hora de aplicar reglas. Este enfoque admite mejoras, ya que no siempre las variables críticas generan problemas (por ejemplo pueden aparecer en el lado derecho en lugares donde su tipo no importa). No obstante, es un enfoque más relajado que [45], donde se prohíben todos los patrones opacos en las reglas, tengan o no variables críticas. También considera constructoras existenciales y constructoras con argumentos funcionales, excluidas en [45]. Otro aspecto destacable del sistema $\vdash^{\bullet}$ es que soporta let-expresiones con distintos grados de polimorfismo, característica utilizada por los distintos sistemas funcionales y lógico-funcionales que en algunos casos no queda completamente documentada. La preservación de tipos es válida sobre reescritura con *call-time choice* (let-reescritura) sin variables extra, por lo que no cubre completamente los cómputos lógico-funcionales, que usualmente hacen uso de variables libres (que se van ligando a valores) y variables extra. Tampoco da solución al problema de la descomposición opaca, que aunque no se da en el formalismo utilizado (pues no soporta igualdad estructural) ocurriría en los sistemas que sí proporcionen una primitiva *ad-hoc* para este fin.

Aparte del sistema de tipos, se proporciona un método de inferencia correcto y maximal para expresiones y programas, que permite inferir y comprobar los tipos para las distintas funciones de los programas. Esto mejora el caso de [45], donde se supone que los programas han de venir acompañados de declaraciones de tipos explícitas.

Partiendo de ideas para relajar las variables opacas del sistema $\vdash^{\bullet}$, en la Sección 7 —[87](A.2), [83](A.3), [100](A.4)— proponemos un sistema de tipos liberal para programas lógico-funcionales. Esto da lugar a un sistema de tipos seguro desde el punto de vista de los tipos (garantizando preservación de tipos y progreso, aparte de corrección sintáctica) a la vez que acepta multitud de programas que son rechazados en PF, PLF o en [45]. Esto proporciona una gran flexibilidad a los programadores, permitiéndoles definir funciones indexadas por tipos, funciones genéricas, utilizar constructoras de datos al estilo de los GADTs o definir una función *apply* similar a la que se usa en las traducciones de orden superior a primer orden. Este sistema de tipos liberal maneja adecuadamente los patrones de orden superior en la reglas, evitando el problema del *casting* polimórfico. Además, y gracias a su liberalidad, permite que una función de

igualdad estructural pueda ser definida mediante reglas. De esta manera se soluciona el problema de la descomposición opaca, ya que las reglas que lo desencadenan no estarían bien tipadas en el sistema liberal.

La noción de regla bien tipada es muy sencilla: el lado derecho no debe restringir los tipos más que el lado izquierdo. Esta noción de regla bien tipada es lo más general posible para preservar tipos, en otras palabras, esta noción es equivalente (bajo ciertas suposiciones razonables) a la noción de regla que preserva tipos. Debido a la liberalidad obtenida no existen tipos principales. Por ello el sistema liberal no proporciona un mecanismo de inferencia de tipos, sino un método de comprobación de tipos a partir de programas con declaraciones de tipos explícitas (como el considerado en [45]).

Uno de los usos más interesantes que proponemos del sistema de tipos liberal es una nueva traducción para las clases de tipos basada en funciones indexadas por tipo y testigos de tipos. La traducción de tipos clásica utiliza diccionarios (una estructura de datos que contiene las versiones específicas de las funciones sobrecargadas) y es la que se ha utilizado en las versiones experimentales de Curry soportando clases de tipos. Sin embargo, dicha traducción adolece de soluciones perdidas en presencia de funciones sobrecargadas indeterministas sin argumentos. Basándose en un sistema de tipos con soporte para clases de tipos clásico, la traducción que proponemos convierte las funciones sobrecargadas en funciones indexadas por tipo y les pasa como argumentos testigos de tipos (patrones que representan tipos) para elegir el comportamiento deseado. Con esta traducción alternativa desaparece el problema de soluciones perdidas. Además, según pruebas realizadas sobre funciones que utilizan clases de tipos, observamos que la traducción alternativa genera programas más simples y que se ejecutan más rápido que la traducción que utiliza diccionarios (con una ganancia entre que varia entre 1,05 y 2,3) incluso considerando optimizaciones.

El sistema de tipos liberal es correcto con respecto a reescritura con *call-time choice* (let-reescritura), pero no cubre cómputos lógico-funcionales en los que se liguen variables libres. De hecho, de manera similar a [45] excluye explícitamente variables libres en las reglas, ya que este tipo de variables violaría la preservación de tipos. Por tanto, aunque da solución a los problemas generados por la opacidad de los patrones de orden superior en las reglas (como el *casting* polimórfico) y a la descomposición opaca, no maneja adecuadamente todos los aspectos lógico-funcionales que perseguíamos.

En la Sección 8 —[89](A.5), [85](B.2)— presentamos un sistema de tipos para dar soporte a cómputos de estrechamiento con variables extra. La derivación de tipos para expresiones es una ligera extensión de DM con soporte para variables extra en las λ-abstracciones. Sin embargo, la aportación más importante es la relación $\leadsto^{lwt}$, que clarifica que cualquier reducción de let-estrechamiento preserva tipos siempre que las sustituciones generadas en cada paso estén bien tipadas. Basándonos en este resultado podemos desarrollar otras relaciones más pequeñas que $\leadsto^{lwt}$, que preservarán tipos si utilizan sustituciones bien tipadas. Un ejemplo de ello es $\leadsto^{lmgu}$ —definido

como una restricción del let-estrechamiento eliminando las reglas que ligan variables libres de tipo funcional y utilizando unificadores más generales al aplicar (Narr)— que preserva tipos cuando no aparecen patrones opacos en el programa.

La relación $\leadsto^{lwt}$ es más general que el cálculo CLNC de [45], ya que soporta variables extra y sustituciones arbitrarias (en lugar de unificadores más generales). Por otro lado, la relación $\leadsto^{lmgu}$ no es más general que dicho cálculo CLNC pues no utiliza las reglas que ligan variables libres de tipo funcional. La característica más importante de $\leadsto^{lmgu}$ es que únicamente genera sustituciones bien tipadas, por lo que preserva tipos sin necesidad de realizar comprobaciones de tipos en cada paso.

También demostramos que la evaluación de programas Curry simplificados utilizando estrechamiento necesario y residuación preserva los tipos. Para ello hemos seguido un enfoque transformacional: primero transformamos el programa Curry simplificado en un programa inductivamente secuencial con solapamiento, y posteriormente transformamos este programa en otro en formato uniforme. El estrechamiento utilizando unificadores más generales sobre el programa transformado se comporta de manera equivalente al estrechamiento necesario sobre el programa original. Además la residuación se simula omitiendo las reglas que ligan variables libres de tipo funcional. Por tanto, la reducción utilizando $\leadsto^{lmgu}$ sobre los programas transformados simula reducciones de estrechamiento necesario con residuación sobre el programa original. Como los programas Curry simplificados no contienen patrones opacos, gracias a $\leadsto^{lmgu}$ tenemos que se preservan los tipos.

Aparte, hemos estudiado las situaciones en las que $\leadsto^{lmgu}$ y $\leadsto^{lwt}$ se comportan de manera similar. Para ello hemos definido una clase de programas bajo los cuales ninguna evaluación utilizará las reglas (VAct) o (VBind). La definición de esta clase de programas utiliza la noción de tipo inseguro: un tipo para el cual existe algún patrón de dicho tipo que contiene alguna variable de tipo funcional. Utilizando esta noción podemos caracterizar dicha clase de programas como aquellos cuyas variables extra tienen tipo seguro y básico (sin variables de tipo). En esta clase de programas nunca podrá aparecer una aplicación de variable, por lo que no se utilizarán las reglas (VAct) ni (VBind). No obstante, esta clase excluye todas aquellas funciones que utilicen variables extra polimórficas, aunque no generen problemas de tipos. Por tanto consideramos que se trata de una clase de programas demasiado restrictiva, que excluye bastantes programas lógico-funcionales interesantes que no introducen aplicaciones de variable, por lo que en el futuro debería ser relajada para ser considerada plenamente satisfactoria.

En este sistema de tipos solo nos hemos centrado en las condiciones necesarias para garantizar la preservación de tipos durante las reducciones de estrechamiento con variables extra. Por ello no hemos incluido ningún método de inferencia para expresiones o programas, considerando que estos vendrán acompañados de declaraciones para todas sus funciones. Sin embargo, debido a la gran similitud del sistema de tipos con el sistema DM, parece sencillo extender el algoritmo de DM para inferir el tipo de

λ-abstracciones con variables extra, dando lugar a un método de inferencia correcto y completo. Posteriormente este método de inferencia para expresiones podría ser utilizado de manera similar al procedimiento para inferir tipos para programas completos o de manera estratificada, como se ha presentado en la Sección 6.2 (página 52).

El sistema de tipos de esta sección soporta variables extra de manera segura. También impide el *casting* polimórfico y problemas similares generados por la opacidad de los patrones opacos, ya que para producirse deben utilizarse sustituciones mal tipadas. Por el contrario, no proporciona ninguna seguridad sobre la descomposición opaca ya que el propio formalismo impide el cómputo de la igualdad estructural: no existen reglas del let-estrechamiento que lo realicen ni se puede definir mediante reglas de programa dando lugar a un programa bien tipado.

Aparte del desarrollo teórico de los sistemas de tipos y la demostración de sus propiedades, también hemos realizado la implementación de algunos de ellos y los hemos integrado como fase de comprobación de tipos en el sistema Toy[41]. El sistema *Toy con patrones opacos seguros*[42] utiliza el sistema $\vdash^{\bullet}$ de la Sección 6, concretamente la inferencia de tipos por bloques de funciones mutuamente recursivas —ver Sección 6.2 y [84](A.1, §5.1)—. Este sistema supone una mejora sobre Toy 2.3.2 oficial ya que, aparte de eliminar los problemas de tipos en presencia patrones opacos como el *casting* polimórfico, soporta recursión polimórfica para funciones cuyo tipo haya sido declarado. Este tipo de recursión no era soportada en Toy 2.3.2 debido a que las declaraciones de tipos proporcionadas por el usuario no eran utilizadas durante la inferencia de tipos, sino que únicamente se comprobaban al final con los tipos inferidos. Con respecto al sistema de tipos de la Sección 7, el sistema *Toy liberal*[43] comprueba que los programas compilados son correctos mediante la noción liberal de programa bien tipado (Definición 6, página 68), concretamente con su versión efectiva que utiliza inferencia para expresiones. La sintaxis de los programas es la misma que la de Toy 2.3.2 oficial, por lo que no acepta constructoras existenciales o GADTs. En cambio, sí que permite explotar el resto de características expresivas, como las funciones indexadas por tipo, las funciones genéricas, la igualdad estructural, etc. Para el sistema de tipos liberal también hemos desarrollado una interfaz web[44] que permite comprobar, sin necesidad de descargar e instala ningún sistema, si un un programa está bien tipado. Esta interfaz web soporta la declaración de GADTs con una sintaxis similar a la de Haskell, por lo que este tipo de constructoras, además de las existenciales, están permitidas. También soporta let-expresiones de variables con tratamiento polimórfico, ausentes en el sistema Toy liberal.

[41] Estos sistemas de tipos no forman parte de la versión oficial de Toy, sino que se trata de ramas independientes.

[42] `http://gpd.sip.ucm.es/Toy2SafeOpaquePatterns`

[43] `http://gpd.sip.ucm.es/Toy2Liberal`

[44] `http://gpd.sip.ucm.es/LiberalTyping`

Antes de esta tesis no existían demasiados trabajos acerca de los aspectos de tipos en programación lógico-funcional, aunque sí que existían algunos que los trataban con rigurosidad. En esta tesis hemos realizado avances interesantes, clarificando el comportamiento con respecto a los tipos de distintos elementos de la programación lógico-funcional y proponiendo soluciones para ellos. Sin embargo, aún queda trabajo por realizar para poder afirmar que los cómputos lógico-funcionales utilizando toda la potencia del paradigma son seguros con respecto a los tipos.

10. Trabajo futuro

En esta siguiente sección analizamos algunas líneas abiertas para el trabajo futuro que nos parecen interesantes:

- Como hemos comentado durante la presentación del sistema $\vdash^{\bullet}$ (Sección 6, página 49), la opacidad generada por los patrones de orden superior es similar a la ocultación de información de las constructoras existenciales. No obstante, esta opacidad tiene orígenes distintos. Los patrones de orden superior representan funciones de manera intensional, y la opacidad que crean es sobrevenida debido a las aplicaciones parciales. En el caso de las constructoras existenciales, su tipo es declarado por el usuario para ocultar información, permitiendo así la implementación de tipos abstractos de datos. Aunque el sistema $\vdash^{\bullet}$ sigue un enfoque basado en variables opacas y críticas, tendría también interés, como vía alternativa, utilizar un enfoque similar al de tipos existenciales [79] y tratar la opacidad generada como ocultación de información. De esta manera podremos reutilizado la idea de las constantes de Skolem, dando lugar a un sistema de tipos con un método de inferencia correcto y completo, con tipos principales [79], y más cercano a lo que se utiliza en lenguajes funcionales como Haskell. Con este enfoque se pierden las posibilidades de genericidad limitada que se aprecian en el Ejemplo 17 (página 64), aunque se aceptarían programas que actualmente son rechazados como el del Ejemplo 11 (página 40). Un enfoque como este requeriría posiblemente que los tipos de todas las funciones que aparezcan en patrones de orden superior hayan sido declarados explícitamente por el usuario. Además esta adaptación debe hacerse con cuidado, ya que a diferencia las constructoras existenciales de [79], que generan ocultación de información siempre que aparecen, en el caso de los patrones de orden superior la opacidad puede aparecer o no dependiendo del número de patrones a los que se aplica un símbolo.
- En el sistema $\vdash^{\bullet}$ hemos utilizado la noción de variable crítica para evitar que variables opacas aparezcan en el lado derecho de las reglas. Sin embargo, la presencia de variables críticas no genera problemas de tipos en todas las situaciones. Consideremos por ejemplo la función $f\ (snd\ X) \to snd\ X$. Esta regla contiene la variable crítica X, en cambio, no generará problemas de tipos

ya que esta aparece en una posición donde su tipo no es *demandado* (el primer argumento de snd puede ser de cualquier tipo, y además no se propaga al tipo del resultado de la función). Una situación similar ocurre con la regla $g\ (snd\ (X : Xs)) \to length\_nat\ Xs$, donde Xs es crítica. Aunque Xs es una variable opaca en el patrón $snd\ (X : Xs)$, ya que su tipo no está completamente determinado por él, sí que conocemos algo de su tipo: es una lista. Por eso podemos asegurar que $length\_nat\ Xs$ no generará ningún error de tipos, ya que $length\_nat$ se puede aplicar a listas conteniendo cualquier tipo de elementos. Estos ejemplos nos indican que el sistema $\vdash^\bullet$ se podría refinar, moviendo el foco de atención de las variables de datos a las variables de tipos. De esta manera habría que asegurar que los fragmentos del tipo de una variable que la hacen opaca (aquellos que no quedan fijados unívocamente por el tipo del patrón) no son «demandados» en el lado derecho. Para ello habría que comprobar que no son forzados a ser más concretos en el lado derecho —como sucede en $h_1\ (snd\ X) \to length\_nat\ X$—, y que además no se reflejan en el tipo de este —como ocurre en $h_2\ (snd\ X) \to [X, X]$—. Ya hemos dado algunos pasos en la dirección de esta línea de trabajo futuro [88], incorporando las mencionadas condiciones en una nueva noción de derivación de tipos $\vdash^\circ$.

- Debido a la gran expresividad que permite el sistema de tipos liberal (Sección 7, página 65), en algunos casos no existen tipos principales para las funciones. Esto dificulta el desarrollo de un método de inferencia de tipos, ya que en esos casos habría que elegir alguno de los tipos válidos en base a algún criterio. Para mejorar la implantación real del sistema de tipos liberal en un sistema lógico-funcional sería deseable desarrollar un método de inferencia para programas que calculase el tipo principal de las funciones para las que exista, y para el resto que calcule alguno válido o falle, indicando al usuario que debe declarar el tipo para esas funciones. Este método de inferencia podría utilizar algunas ideas aparecidas en los últimos trabajos sobre inferencia de tipos para GADTs, como [134]. También sería interesante considerar un sistema de tipos para PLF combinado, que integre el sistema de tipos $\vdash^\bullet$, el sistema de tipos liberal y clases de tipos (implementadas posiblemente mediante la traducción alternativa que utiliza funciones indexadas por tipo y testigos de tipo). De esta manera el usuario podría elegir exactamente qué funciones quiere que sean tratadas de manera liberal, proporcionando además su tipo, dejando que el resto de tipos sean inferidos. Además tendría a su disposición las clases de tipos, para que pudiera elegir la técnica (clases de tipos, funciones genéricas, GADTs o funciones indexadas por tipo) que mejor se ajusta a sus necesidades. Esta combinación necesitaría un estudio formal de las propiedades de seguridad de tipos que se obtienen al mezclar las distintas opciones.
- Con respecto a la traducción alternativa de clases de tipos (Sección 7.5, página

77), sería muy interesante implementarla e integrarla completamente en Toy. De esta manera podríamos comprobar si los esperanzadores resultados de ganancia se mantienen al realizar pruebas exhaustivas sobre un conjunto más grande de programas reales. También sería interesante estudiar la facilidad con la que diversas extensiones bien conocidas en PF, como las *clases de tipos multiparámetro* [116] o las *clases de constructoras* [71], encajan en la traducción propuesta. De acuerdo con [146] estas extensiones son fácilmente integrables en una traducción como la nuestra, que en lugar de diccionarios utiliza tipos como argumentos. Un estudio más detallado de los patrones de orden superior formados por funciones sobrecargadas también sería deseable, para encontrar una solución más permisiva que prohibirlos.

- Con respecto al sistema de tipos que soporta variables extra y estrechamiento (Sección 8, página 91), sería interesante encontrar una clase de programas más relajada en la que no se utilicen las reglas (VAct) ni (VBind), ya que la actualmente propuesta es demasiado restrictiva. Como hemos visto, $\leadsto^{lmgu}$ preserva tipos en programas cuyos patrones son transparentes. Otra línea interesante de trabajo futuro sería afinar el sistema de tipos (restringiendo así la noción de programa bien tipado) para conseguir que $\leadsto^{lmgu}$ preserve tipos aun en presencia de patrones opacos. Una posibilidad sería utilizar un enfoque similar a los tipos existenciales [112, 79], que prohíben el encaje de patrones en posiciones opacas. De esta manera se evitaría que aparezcan patrones compuestos en posiciones opacas de los lados izquierdos de las reglas —únicamente podrían aparecer variables—, con lo que el unificador más general utilizado utilizado por (Narr) no ligaría variables incorrectamente. Esto se aprecia en la reducción

$$[\underline{f\ (snd\ X)}, X] \leadsto^{lmgu}_{[X \mapsto zero]} [true, zero]$$

del Ejemplo 2 (página 6), donde la variable X es ligada a $zero$ al aplicar la regla $f\ (snd\ zero) \to true$, que contiene el patrón opaco $snd\ true$. Con el enfoque propuesto esta regla estaría mal tipada, pues contiene el patrón compuesto $zero$ en una posición opaca. Sin embargo, una regla como $f'\ (snd\ X) \to true$ sí que estaría bien tipada, considerando que f' tiene tipo $\forall\alpha.(\alpha \to \alpha) \to bool$. La razón es que la posición opaca está ahora ocupada por una variable, aunque el patrón sigue siendo opaco. Como se puede ver, la aplicación de $\leadsto^{lmgu}$ —utilizando la regla (Narr) con la variante de regla fresca $f'\ (snd\ X_1) \to true$— no produce ahora una expresión mal tipada:

$$[\underline{f'\ (snd\ X)}, X] \leadsto^{lmgu}_{[X_1 \mapsto X]} [true, X]$$

La justificación de este comportamiento parece estar ligada a la parametricidad del sistema de tipos (como ya se comenta en [45] con respecto a la generalidad de

tipos), pues al recuperarla se dejan de producir sustituciones de estrechamiento mal tipadas incluso en presencia de patrones opacos.

- En esta tesis, el único sistema de tipos que ha propuesto una solución a la descomposición opaca ha sido el sistema liberal, que permitía definir mediante reglas bien tipadas la igualdad estructural. Sin embargo, sería interesante estudiar posibles extensiones del sistema $\vdash^{\bullet}$ que la manejen de manera segura aun tratándose de una primitiva *ad-hoc*. Para ello, sería necesario introducir reglas en la let-reescritura que realicen los cómputos de igualdad estructural. Una posibilidad puede ser extender la propia representación de los tipos para que los patrones opacos reflejen exactamente el tipo de los datos que contienen, de tal manera que $snd\ true$ refleje que contiene un booleano y $snd\ zero$ refleje que contiene un natural. De esta manera una igualdad $snd\ true = snd\ zero$ debería ser considerada mal tipada (al igual que $true = zero$), puesto que los patrones tienen tipos distintos.

Referencias

[1] ALBERT, E., HANUS, M., HUCH, F., OLIVER, J., AND VIDAL, G. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation 40*, 1 (2005), 795–829.

[2] ALIMARINE, A., AND PLASMEIJER, R. A generic programming extension for Clean. In *13th International Workshop on Implementation of Functional Languages (IFL '01), Revised Papers* (2002), vol. 2312 of *Lecture Notes in Computer Science*, Springer, pp. 168–185.

[3] ANTOY, S. Definitional trees. In *Proceedings of the Third International Conference on Algebraic and Logic Programming (ALP '92)* (1992), Springer, pp. 143–157.

[4] ANTOY, S. Optimal non-deterministic functional logic computations. In *Proceedings of the 6th International Conference on Algebraic and Logic Programming (ALP '97)* (1997), vol. 1298 of *Lecture Notes in Computer Science*, Springer, pp. 16–30.

[5] ANTOY, S. Constructor based conditional narrowing. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '01)* (2001), ACM, pp. 199–206.

[6] ANTOY, S., ECHAHED, R., AND HANUS, M. A needed narrowing strategy. *Journal of the ACM 47* (July 2000), 776–822.

[7] ANTOY, S., AND HANUS, M. Declarative programming with function patterns. In *Proceedings of the 15th International Conference on Logic Based Program Synthesis and Transformation (LOPSTR '05)* (2006), Springer, pp. 6–22.

[8] ANTOY, S., AND HANUS, M. Functional logic programming. *Communications of the ACM 53*, 4 (2010), 74–85.

[9] ANTOY, S., AND TOLMACH, A. Typed higher-order narrowing without higher-order strategies. In *Proceedings of the 4th International Symposium on Functional and Logic Programming (FLOPS '99)* (1999), vol. 1722 of *Lecture Notes in Computer Science*, Springer, pp. 335–352.

[10] ARIOLA, Z., FELLEISEN, M., MARAIST, J., ODERSKY, M., AND WADLER, P. A call-by-need lambda calculus. In *Proceedings of the 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '95)* (1995), ACM, pp. 233–246.

[11] ARMSTRONG, J. A history of erlang. In *Proceedings of the 3rd ACM SIGPLAN conference on History of programming languages (HOPL III)* (2007), ACM, pp. 6–1–6–26.

[12] AUGUSTSSON, L. Implementing Haskell overloading. In *Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture (FPCA '93)* (1993), ACM, pp. 65–73.

[13] BAADER, F., AND NIPKOW, T. *Term Rewriting and All That*. Cambridge University Press, 1998.

[14] BACKHOUSE, R., JANSSON, P., JEURING, J., AND MEERTENS, L. Generic programming — an introduction. In *Advanced Functional Programming, Third International School (AFP '98)* (1999), vol. 1608 of *Lecture Notes in Computer Science*, Springer, pp. 28–115.

[15] BARENDREGT, H. P., VAN EEKELEN, M. C. J. D., GLAUERT, J. R. W., KENNAWAY, R., PLASMEIJER, M. J., AND SLEEP, M. R. Term graph rewriting. In *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages* (1987), vol. 259 of *Lecture Notes in Computer Science*, Springer, pp. 141–158.

[16] BERRUETA, D. Zinc project. `http://zinc-project.sourceforge.net/`.

[17] BLOTT, S. Type inference and type classes. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming* (1990), Springer, pp. 254–265.

[18] BRASSEL, B. Two to three ways to write an unsafe type cast without importing unsafe - post to the Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0705.html`, May 2008.

[19] BRASSEL, B., HANUS, M., AND HUCH, F. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming 2004*, 6 (2004).

[20] BRASSEL, B., AND HUCH, F. On a tighter integration of functional and logic programming. In *Proceedings of the 5th Asian Symposium on Programming Languages and Systems (APLAS '07)* (2007), vol. 4807 of *Lecture Notes in Computer Science*, Springer, pp. 122–138.

[21] BRUS, T. H., VAN EEKELEN, C. J. D., VAN LEER, M. O., AND PLASMEIJER, M. J. CLEAN: A language for functional graph rewriting. In *Proceedings of the 3rd International Conference on Functional Programming Languages and Computer Architecture (FPCA '87)* (1987), vol. 274 of *Lecture Notes in Computer Science*, Springer, pp. 364–384.

[22] BUENO, F., CARRO, M., HAEMMERLÉ, R., HEMENEGILDO, M., LÓPEZ, P., MERA, E., MORALES, J. F., AND PUEBLA, G. The Ciao system: A new generation, multi-paradigm programming language and environment. reference manual. Tech. Rep. CLIP 3/97-1.14#2, Universidad Politécnica de Madrid, August 2011. Available at `http://www.ciaohome.org/manuals.html`.

[23] CABALLERO, R., SÁNCHEZ, J., SÁNCHEZ, P. A., LEIVA, A. J. F., LUEZAS, A. G., FRAGUAS, F. L., ARTALEJO, M. R., AND PÉREZ, F. S. Toy, a multiparadigm declarative language. version 2.3.2, October 2011. Available at `http://toy.sourceforge.net`.

[24] CARDELLI, L. Type systems. In *CRC Handbook of Computer Science and Engineering Handbook*, 2nd ed. CRC Press, 2004, ch. 97.

[25] CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys 17* (1985), 471–522.

[26] CARLSSON, M., AND MILDNER, P. Sicstus prolog — the first 25 years. *Theory and Practice of Logic Programming* (2010). To appear. arXiv:1011.5640.

[27] CHENEY, J., AND HINZE, R. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell '02)* (October 2002), M. M. Chakravarty, Ed., ACM-Press, pp. 90–104.

[28] CHENEY, J., AND HINZE, R. First-class phantom types. Tech. Rep. TR2003-1901, Cornell University, July 2003.

[29] CHRISTIANSEN, J., SEIDEL, D., AND VOIGTLÄNDER, J. Free theorems for functional logic programs. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV '10)* (2010), ACM, pp. 39–48.

[30] CLÉMENT, D., DESPEYROUX, T., KAHN, G., AND DESPEYROUX, J. A simple applicative language: mini-ML. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming (LFP '86)* (1986), ACM, pp. 13–27.

[31] DAMAS, L. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985. Also appeared as Technical report CST-33-85.

[32] DAMAS, L., AND MILNER, R. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)* (1982), ACM, pp. 207–212.

[33] DEL VADO VÍRSEDA, R. Estrategias de estrechamiento perezoso. Master's thesis, Universidad Compluetense de Madrid, 2002.

[34] DERANSART, P., ED-DBALI, A., AND CERVONI, L. *Prolog: The Standard. Reference Manual*. Springer, 1996.

[35] DIJKSTRA, A., FOKKER, J., AND SWIERSTRA, S. D. The structure of the essential Haskell compiler, or coping with compiler complexity. In *Implementation and Application of Functional Languages*. Springer, 2008, pp. 57–74.

[36] ECHAHED, R., AND JANODET, J.-C. On constructor-based graph rewriting systems. Research Report 985-I, IMAG, 1997.

[37] ECHAHED, R., AND JANODET, J.-C. Admissible graph rewriting and narrowing. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming (JICSLP '98)* (1998), MIT Press, pp. 325–340.

[38] ESCOBAR, S. Refining weakly outermost-needed rewriting and narrowing. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '03)* (2003), ACM, pp. 113–123.

[39] ESCOBAR, S., MESEGUER, J., AND THATI, P. Natural narrowing for general term rewriting systems. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA '05)* (2005), vol. 3467 of *Lecture Notes in Computer Science*, Springer, pp. 279–293.

[40] GALLEGO ARIAS, E. J. Sloth Curry compiler. `http://babel.ls.fi.upm.es/research/Sloth/`.

[41] GHC-TEAM. The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/latest/html/users_guide`, 2011.

[42] GIRARD, J.-Y. The system F of variable types, fifteen years later. *Theoretical Computer Science 45* (1986), 159–192.

[43] GONZÁLEZ-MORENO, J., HORTALÁ-GONZÁLEZ, T., LÓPEZ-FRAGUAS, F., AND RODRÍGUEZ-ARTALEJO, M. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming 40*, 1 (1999), 47–87.

[44] GONZÁLEZ-MORENO, J., HORTALÁ-GONZÁLEZ, T., AND RODRÍGUEZ-ARTALEJO, M. A higher order rewriting logic for functional logic programming. In *Proceedings of the 14th International Conference on Logic Programming (ICLP '97)* (1997), MIT Press, pp. 153–167.

[45] GONZÁLEZ-MORENO, J., HORTALÁ-GONZÁLEZ, T., AND RODRÍGUEZ-ARTALEJO, M. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming 2001*, 1 (July 2001).

[46] GONZÁLEZ-MORENO, J. C., HORTALÁ-GONZÁLEZ, M. T., LÓPEZ-FRAGUAS, F. J., AND RODRÍGUEZ-ARTALEJO, M. A rewriting logic for declarative programming. In *Proceedings of the 6th European Symposium on Programming Languages and Systems (ESOP '96)* (1996), Springer, pp. 156–172.

[47] GONZÁLEZ-MORENO, J. C., HORTALÁ-GONZÁLEZ, M. T., AND RODRÍGUEZ-ARTALEJO, M. Semantics and types in functional logic programming. In *Proceedings of the 4th International Symposium on Functional and Logic Programming (FLOPS '99)* (1999), Springer, pp. 1–20.

[48] HALL, C. V., HAMMOND, K., PEYTON JONES, S., AND WADLER, P. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems 18*, 2 (1996), 109–138.

[49] HAMMOND, K., AND BLOTT, S. Implementing Haskell type classes. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming* (1990), Springer, pp. 266–286.

[50] HANUS, M. Polymorphic higher-order programming in prolog. In *Proceedings of the 6th International Conference on Logic Programming (ICLP '89)* (1989), MIT Press, pp. 382–397.

[51] HANUS, M. A functional and logic language with polymorphic types. In *Proceeding of the International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO '90)* (1990), vol. 429 of *Lecture Notes in Computer Science*, Springer, pp. 215–224.

[52] HANUS, M. Horn clause programs with polymorphic types: Semantics and resolution. *Theoretical Computer Science 89* (1991), 63–106.

[53] HANUS, M. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programing 19/20* (1994), 583–628.

[54] HANUS, M. Curry: An integrated functional logic language (version 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, March 2006.

[55] HANUS, M. Multi-paradigm declarative languages. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP '07)* (2007), vol. 4670 of *Lecture Notes in Computer Science*, Springer, pp. 45–75.

[56] HANUS, M. *PAKCS 1.10.0, The Portland Aachen Kiel Curry System, User manual*, November 2011. Available at `http://www.informatik.uni-kiel.de/~pakcs/Manual.pdf`.

[57] HENGLEIN, F. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems 15*, 2 (1993), 253–289.

[58] HINDLEY, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (1969), 29–60.

[59] HINZE, R. A generic programming extension for Haskell. In *Proceedings of the 1999 Haskell Workshop* (1999). The proceedings appeared as a technical report of Universiteit Utrecht, UU-CS-1999-28.

[60] HINZE, R. A new approach to generic functional programming. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)* (2000), ACM, pp. 119–132.

[61] HINZE, R. Fun with phantom types. In *The Fun of Programming*. Palgrave Macmillan, 2003, pp. 245–262.

[62] HINZE, R. Generics for the masses. *Journal of Functional Programming 16*, 4-5 (2006), 451–483.

[63] HINZE, R., JEURING, J., AND LÖH, A. Comparing approaches to generic programming in Haskell. In *Datatype-Generic Programming*, vol. 4719 of *Lecture Notes in Computer Science*. Springer, 2007, pp. 72–149.

[64] HINZE, R., AND LÖH, A. Generic programming, now! In *Datatype-Generic Programming*, vol. 4719 of *Lecture Notes in Computer Science*. Springer, 2007, pp. 150–208.

[65] HUDAK, P., HUGHES, J., PEYTON JONES, S., AND WADLER, P. A history of Haskell: Being lazy with class. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III)* (2007), ACM, pp. 12–1–12–55.

[66] HUSSMANN, H. *Nondeterminism in Algebraic Specifications and Algebraic Programs*. Birkhauser Verlag, 1993.

[67] IBM. IBM ILOG CPLEX Optimization Studio, 2012. `http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/`.

[68] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO/IEC 13211-1:1995 and ISO/IEC 13211-2:2000: Information technology – Programming languages – Prolog – Part 1: General core and Part 2: Modules*. International Organization for Standardization (ISO), 1995.

[69] JAFFAR, J., AND MAHER, M. J. Constraint logic programming: A survey. *Journal of Logic Programing 19/20* (1994), 503–581.

[70] JANSSON, P., AND JEURING, J. Polyp — a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)* (1997), ACM, pp. 470–482.

[71] JONES, M. P. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture (FPCA '93)* (1993), ACM, pp. 52–61.

[72] JONES, M. P. Dictionary-free overloading by partial evaluation. *Lisp and Symbolic Computation 8* (1995), 229–248.

[73] JONES, M. P. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00)* (2000), Springer, pp. 230–244.

[74] JONES, M. P., AND PETERSON, J. The Hugs 98 user manual. `http://cvs.haskell.org/Hugs/pages/hugsman/index.html`, 1999.

[75] KAES, S. Parametric overloading in polymorphic programming languages. In *Proceedings of the 2nd European Symposium on Programming (ESOP '88)* (1988), vol. 300 of *Lecture Notes in Computer Science*, Springer, pp. 131–144.

[76] KFOURY, A. J., TIURYN, J., AND URZYCZYN, P. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems 15*, 2 (1993), 290–311.

[77] LÄMMEL, R., AND JONES, S. P. Scrap your boilerplate: A practical design pattern for generic programming. *SIGPLAN Notices 38* (January 2003), 26–37.

[78] LÄUFER, K. Type classes with existential types. *Journal of Functional Programming 6*, 3 (1996), 485–517.

[79] LÄUFER, K., AND ODERSKY, M. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems 16* (1994), 1411–1430.

[80] LAUNCHBURY, J. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '93)* (1993), ACM, pp. 144–154.

[81] LLOYD, J. W. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming 3* (1999).

[82] LOOGEN, R., LÓPEZ-FRAGUAS, F., AND RODRÍGUEZ-ARTALEJO, M. A demand driven computation strategy for lazy narrowing. In *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming (PLILP '93)* (1993), Springer, pp. 184–200.

[83] LÓPEZ-FRAGUAS, F., MARTIN-MARTIN, E., AND RODRÍGUEZ-HORTALÁ, J. Liberal typing for functional logic programs. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10)* (2010), vol. 6461 of *Lecture Notes in Computer Science*, Springer, pp. 80–96.

[84] LÓPEZ-FRAGUAS, F., MARTIN-MARTIN, E., AND RODRÍGUEZ-HORTALÁ, J. New results on type systems for functional logic programming. In *18th International Workshop on Functional and (Constraint) Logic Programming (WFLP '09), Revised Selected Papers* (2010), vol. 5979 of *Lecture Notes in Computer Science*, Springer, pp. 128–144.

[85] LÓPEZ-FRAGUAS, F., MARTIN-MARTIN, E., AND RODRÍGUEZ-HORTALÁ, J. Well-typed narrowing with extra variables in functional-logic programming (extended version). Tech. Rep. SIC-11-11, Universidad Complutense de Madrid, November 2011.

Available at http://gpd.sip.ucm.es/enrique/publications/pepm12/SIC-11-11.pdf.

[86] López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. Advances in type systems for functional logic programming (extended version). Tech. Rep. SIC-05-12, Universidad Complutense de Madrid, March 2012. Available at http://gpd.sip.ucm.es/enrique/publications/wflp09/SIC-05-12.pdf.

[87] López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. A liberal type system for functional logic programs. *Mathematical Structures in Computer Science* (2012). Accepted for publication.

[88] López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. Safe typing of functional logic programs with opaque patterns and local bindings. *Information and Computation* (2012). Under consideration for publication.

[89] López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. Well-typed narrowing with extra variables in functional-logic programming. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '12)* (2012), ACM, pp. 83–92.

[90] López-Fraguas, F. J., Rodríguez-Hortalá, J., and Sánchez-Hernández, J. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '07)* (2007), ACM, pp. 197–208.

[91] López-Fraguas, F. J., Rodríguez-Hortalá, J., and Sánchez-Hernández, J. Narrowing for first order functional logic programs with call-time choice semantics. In *Proceedings of the 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP '07), and 21st Workshop on Logic Programming (WLP '07), Revised Selected Papers* (2009), vol. 5437 of *Lecture Notes in Computer Science*, Springer, pp. 206–222.

[92] López-Fraguas, F., Rodríguez-Hortalá, J., and Sánchez-Hernández, J. Rewriting and call-time choice: the HO case. In *Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS '08)* (2008), vol. 4989 of *Lecture Notes in Computer Science*, Springer, pp. 147–162.

[93] López-Fraguas, F., and Sánchez-Hernández, J. Toy: A multiparadigm declarative system. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA '99)* (1999), vol. 1631 of *Lecture Notes in Computer Science*, Springer, pp. 244–247.

[94] Lux, W. The münster Curry compiler. http://danae.uni-muenster.de/~lux/curry/.

[95] Lux, W. Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI* (2008), pp. 67–76.

[96] LUX, W. Type-classes and call-time choice vs. run-time choice - post to the Curry mailing list. http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html, August 2009.

[97] MACQUEEN, D., PLOTKIN, G., AND SETHI, R. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)* (1984), ACM, pp. 165–174.

[98] MARTELLI, A., AND MONTANARI, U. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems 4*, 2 (1982), 258–282.

[99] MARTIN-MARTIN, E. Advances in type systems for functional logic programming. Master's thesis, Universidad Complutense de Madrid, July 2009. Available at http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf.

[100] MARTIN-MARTIN, E. Type classes in functional logic programming. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)* (2011), ACM, pp. 121–130.

[101] MICROSOFT. *The F# 2.0 Language Specification*, April 2010. Available at http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.pdf.

[102] MICROSOFT RESEARCH. F# at microsoft research. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp.

[103] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[104] MILNER, R., TOFTE, M., AND HARPER, R. *The Definition of Standard ML*. MIT Press, 1990.

[105] MITCHELL, J. C., AND PLOTKIN, G. D. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems 10*, 3 (1988), 470–502.

[106] MORENO, J. C. G. A correctness proof for Warren's HO into FO translation. In *Proceedings of the 8th Italian Conference on Logic Programming (GULP '93)* (1993), D. Saccà, Ed., pp. 569–584.

[107] MORENO-NAVARRO, J. J., MARIÑO, J., DEL POZO-PIETRO, A., HERRANZ-NIEVA, Á., AND GARCÍA-MARTÍN, J. Adding type classes to functional-logic languages. In *1996 Joint Conference on Declarative Programming (APPIA-GULP-PRODE '96)* (1996), pp. 427–438.

[108] MYCROFT, A. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming* (1984), Springer, pp. 217–228.

[109] NIPKOW, T., AND PREHOFER, C. Type reconstruction for type classes. *Journal of Functional Programming 5*, 2 (1995), 201–224.

[110] NORELL, U., AND JANSSON, P. Polytypic programming in Haskell. In *15th International Workshop on Implementation of Functional Languages (IFL '03), Revised Papers* (2004), vol. 3145 of *Lecture Notes in Computer Science*, Springer, pp. 168–184.

[111] ODERSKY, M., AND LÄUFER, K. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)* (1996), ACM, pp. 54–67.

[112] PERRY, N. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College, 1991.

[113] PETERSON, J., AND JONES, M. Implementing type classes. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '93)* (1993), ACM, pp. 227–236.

[114] PEYTON JONES, S. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science, 1987.

[115] PEYTON JONES, S., Ed. *Haskell 98 Language and Libraries — The Revised Report*. Cambridge University Press, 2003.

[116] PEYTON JONES, S., JONES, M., AND MEIJER, E. Type classes: An exploration of the design space. In *Proceedings of the 1997 ACM SIGPLAN Haskell Workshop (Haskell '97)* (1997).

[117] PEYTON JONES, S., VYTINIOTIS, D., AND WEIRICH, S. Simple unification-based type inference for GADTs (technical appendix). Tech. Rep. MS-CIS-05-22, University of Pennsylvania, May 2006.

[118] PEYTON JONES, S., VYTINIOTIS, D., WEIRICH, S., AND SHIELDS, M. Practical type inference for arbitrary-rank types. *Journal of Functional Programming 17*, 1 (2007), 1–82.

[119] PEYTON JONES, S., VYTINIOTIS, D., WEIRICH, S., AND WASHBURN, G. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP '06)* (2006), ACM, pp. 50–61.

[120] PIERCE, B. C. *Types and Programming Languages*. The MIT Press, 2002.

[121] PLASMEIJER, M. J. CLEAN: A programming environment based on term graph rewriting. *Electronic Notes in Theoretical Computer Science 194*, 2 (1998), 246–255.

[122] PLASMEIJER, R., VAN EEKELEN, M., AND VAN GRONINGEN, J. *Clean version 2.2 Language Report*, December 2011. Available at `http://wiki.clean.cs.ru.nl/Documentation`.

[123] PLASMEIJER, R., AND VAN EEKELEN, M. C. J. D. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[124] POTTIER, F., AND RÉGIS-GIANAS, Y. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)* (2006), ACM, pp. 232–244.

[125] REYNOLDS, J. C. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation* (1974), Springer, pp. 408–423.

[126] REYNOLDS, J. C. Types, abstraction and parametric polymorphism. *Information Processing*, 83 (1983), 513–523.

[127] RIESCO, A., AND RODRÍGUEZ-HORTALÁ, J. Singular and plural functions for functional logic programming. *Theory and Practice of Logic Programming* (2012). To appear, arXiv:1203.2431v1 [cs.PL].

[128] ROBINSON, J. A. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12*, 1 (1965), 23–41.

[129] RODRIGUEZ, A., JEURING, J., JANSSON, P., GERDES, A., KISELYOV, O., AND OLIVEIRA, B. C. D. S. Comparing libraries for generic programming in Haskell. *SIGPLAN Notices 44* (2008), 111–122.

[130] RODRÍGUEZ-ARTALEJO, M. Functional and constraint logic programming. In *Constraints in Computational Logics: Theory and Applications. International Summer School (CCL'99)*, vol. 2002 of *Lecture Notes in Computer Science*. Springer, 2001, pp. 202–270.

[131] RODRÍGUEZ-HORTALÁ, J. *Programming with Non-Determinism: a Rewriting Based Approach*. PhD thesis, Universidad Complutense de Madrid, June 2010.

[132] SÁNCHEZ-HERNÁNDEZ, J. *Una aproximación al fallo constructivo en programación declarativa multiparadigma*. PhD thesis, Universidad Complutense de Madrid, 2004.

[133] SÁNCHEZ-HERNÁNDEZ, J. Reduction strategies for rewriting with call-time choice (work in progress). In *Actas de las XI Jornadas sobre Programación y Lenguajes (PROLE '11)* (2011), pp. 47–61.

[134] SCHRIJVERS, T., PEYTON JONES, S., SULZMANN, M., AND VYTINIOTIS, D. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)* (2009), ACM, pp. 341–352.

[135] SCHULTE, C., TACK, G., AND LAGERKVIST, M. Z. Gecode 3.7.3: Generic Constraint Development Environment, 2012. `http://www.gecode.org`.

[136] SCHULTE, C., TACK, G., AND LAGERKVIST, M. Z. Modeling and Programming with Gecode, 2012. http://www.gecode.org/doc-latest/MPG.pdf.

[137] SEIDEL, D., AND VOIGTLÄNDER, J. Automatically generating counterexamples to naive free theorems. In *Proceedings of 10th International Symposium on Functional and Logic Programming (FLOPS '10)* (2010), vol. 6009 of *Lecture Notes in Computer Science*, Springer, pp. 175–190.

[138] SHEARD, T., AND JONES, S. P. Template meta-programming for Haskell. *SIGPLAN Notices 37* (December 2002), 60–75.

[139] SOMOGYI, Z., HENDERSON, F. J., AND CONWAY, T. C. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming 29* (1996).

[140] STEELE, JR., G. L., AND GABRIEL, R. P. The evolution of Lisp. In *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages (HOPL II)* (1993), ACM, pp. 231–270.

[141] STEWART, D. nobench: Benchmarking Haskell implementations. http://code.haskell.org/nobench/.

[142] STRACHEY, C. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation 13*, 1-2 (2000), 11–49.

[143] SYME, D., GRANICZ, A., AND CISTERNINO, A. *Expert F# 2.0*. Apress, June 2010.

[144] SØNDERGAARD, H., AND SESTOFT, P. Non-determinism in functional languages. *The Computer Journal 35*, 5 (1992), 514–523.

[145] TERESE. *Term Rewriting Systems*, vol. No. 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[146] THATTÉ, S. R. Semantics of type classes revisited. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming (LFP '94)* (1994), ACM, pp. 208–219.

[147] TOFTE, M., AND TALPIN, J.-P. Region-based memory management. *Information and Computation 132*, 2 (1997), 109–176.

[148] WADLER, P. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)* (1989), ACM, pp. 347–359.

[149] WADLER, P. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science 73* (1990), 231–248.

[150] WADLER, P., AND BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)* (1989), ACM, pp. 60–76.

[151] WARREN, D. H. Higher-order extensions to Prolog: Are they needed? In *Machine Intelligence 10*, J. Hayes, D. Michie, and Y.-H. Pao, Eds. Ellis Horwood Ltd., 1982, pp. 441–454.

[152] WEIRICH, S. Replib: A library for derivable type classes. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell (Haskell '06)* (2006), ACM, pp. 1–12.

[153] WIELEMAKER, J. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments (WLPE '03)* (2003), Katholieke Universiteit Leuven, pp. 1–16. CW 371.

[154] WRIGHT, A. K., AND FELLEISEN, M. A syntactic approach to type soundness. *Information and Computation 115* (1992), 38–94.

[155] XI, H., CHEN, C., AND CHEN, G. Guarded recursive datatype constructors. *SIGPLAN Notices 38*, 1 (2003), 224–235.

[156] ZARTMANN, F. Denotational abstract interpretation of functional logic programs. In *Proceedings of the 4th International Symposium on Static Analysis (SAS '97)* (1997), vol. 1302 of *Lecture Notes in Computer Science*, Springer, pp. 141–159.

Parte V
Publicaciones asociadas a la tesis

A. Publicaciones principales

(A.1) **New Results on Type Systems for Functional Logic Programming**
Francisco López-Fraguas, Enrique Martin-Martin y Juan Rodríguez-Hortalá
En Proceedings 18th International Workshop on Functional and (Constraint) Logic Programming (WFLP'09), Revised Selected Papers, páginas 128–144. Springer LNCS 5979, 2010.

→ Página 131

(A.2) **A Liberal Type System for Functional Logic Programs**
Francisco López-Fraguas, Enrique Martin-Martin y Juan Rodríguez-Hortalá
Aceptado para publicación en *Mathematical Structures in Computer Science*. Cambridge University Press, 2013.

→ Página 148

(A.3) **Liberal Typing for Functional Logic Programs**
Francisco López-Fraguas, Enrique Martin-Martin y Juan Rodríguez-Hortalá
En Proceedings 8th Asian Symposium on Programming Languages and Systems (APLAS'10), páginas 80–96. Springer LNCS 6461, 2010.

→ Página 184

(A.4) **Type Classes in Functional Logic Programming**
Enrique Martin-Martin
En Proceedings of the ACM SIGPLAN 2011 Workshop on Partial Evaluation and Program Manipulation (PEPM'11), páginas 121–130. ACM, 2011.

→ Página 201

(A.5) **Well-typed Narrowing with Extra Variables in Functional-Logic Programming**
Francisco López-Fraguas, Enrique Martin-Martin y Juan Rodríguez-Hortalá
En Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation (PEPM'12), páginas 83–92. ACM, 2012.

→ Página 211

New Results on Type Systems for Functional Logic Programming* **

Francisco J. López-Fraguas
Enrique Martin-Martin
Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, emartinm@fdi.ucm.es, juanrh@fdi.ucm.es

Abstract. Type systems are widely used in programming languages as a powerful tool providing safety to programs, and forcing the programmers to write code in a clearer way. Functional logic languages have inherited Damas & Milner type system from their functional part due to its simplicity and popularity. In this paper we address a couple of aspects that can be subject of improvement. One is related to a problematic feature of functional logic languages not taken under consideration by standard systems: it is known that the use of *opaque* HO patterns in left-hand sides of program rules may produce undesirable effects from the point of view of types. We re-examine the problem, and propose a Damas & Milner-like type system where certain uses of HO patterns (even opaque) are permitted while preserving type safety, as proved by a subject reduction result that uses *HO-let-rewriting*, a recently proposed reduction mechanism for HO functional logic programs. The other aspect is the different ways in which polymorphism of local definitions can be handled. At the same time that we formalize the type system, we have made the effort of technically clarifying the overall process of type inference in a whole program.

1 Introduction

Type systems for programming languages are an active area of research [18], no matter which paradigm one considers. In the case of functional programming, most type systems have arisen as extensions of Damas & Milner's [4], for its remarkable simplicity and good properties (decidability, existence of principal

* This work has been partially supported by the Spanish projects Merit-Forms-UCM (TIN2005-09207-C03-03), STAMP (TIN2008-06622-C03-01), Promesas-CAM (S-0505/TIC/0407) and GPD-UCM (UCM-BSCH-GR58/08-910502)

** This is the authors' version of the work. The definitive version was published in FUNCTIONAL AND CONSTRAINT LOGIC PROGRAMMING, Lecture Notes in Computer Science, 2010, Volume 5979/2010, 128-144, DOI: 10.1007/978-3-642-11999-6_9, http://www.springerlink.com/content/r0410hp00182h247/. The original publication is available at www.springerlink.com

types, possibility of type inference). Functional logic languages [12, 8, 7], in their practical side, have inherited more or less directly Damas & Milner's types. In principle, most of the type extensions proposed for functional programming could be also incorporated to functional logic languages (this has been done, for instance, for type classes in [15]). However, if types are not only decoration but are to provide safety, one should be sure that the adopted system has indeed good properties. In this paper we tackle a couple of orthogonal aspects of existing FLP systems that are problematic or not well covered by standard Damas & Milner systems. One is the presence of so called *HO patterns* in programs, an expressive feature allowed in some systems and for which a sensible semantics exists [5]; however, it is known that unrestricted use of HO patterns leads to type unsafety, as recalled below. The second is the degree of polymorphism assumed for local pattern bindings, a matter with respect to which existing FP or FLP systems vary greatly.

The rest of the paper is organized as follows. The next two subsections further discuss the two mentioned aspects. Sect. 2 contains some preliminaries about FL programs and types. In Sect. 3 we expose the type system and prove its soundness wrt. the *let rewriting* semantics of [11]. Sect. 4 contains a type inference relation, which let us find the most general type of expressions. Sect. 5 presents a method to infer types for programs. Finally, Sect. 6 contains some conclusions and future work. Omitted proofs can be found in [13].

1.1 Higher order patterns

In our formalism patterns appear in the left-hand side of rules and in lambda or let expressions. Some of these patterns can be HO patterns, if they contain partial applications of function or constructor symbols. HO patterns can be a source of problems from the point of view of the types. In particular, it was shown in [6] that unrestricted use of HO patterns leads to loss of *subject reduction*, an essential property for a type system expressing that evaluation does not change types. The following is a crisp example of the problem.

Example 1 (Polymorphic Casting [2]). Consider the program consisting of the rules $snd\ X\ Y \to Y$, $and\ true\ X \to X$, $and\ false\ X \to false$, with the usual types inferred by a classical Damas & Milner algorithm. Then we can write the functions $unpack\ (snd\ X) \to X$ and $cast\ X \to unpack\ (snd\ X)$, whose inferred types will be $\forall\alpha.\forall\beta.(\alpha \to \alpha) \to \beta$ and $\forall\alpha.\forall\beta.\alpha \to \beta$ respectively. It is clear that the expression $and\ (cast\ 0)\ true$ is well-typed, because $cast\ 0$ has type $bool$ (in fact it has any type), but if we reduce that expression using the rules of $cast$ and $unpack$ the resulting expression $and\ 0\ true$ is ill-typed.

The problem arises when dealing with HO patterns, because unlike FO patterns, knowing the type of a HO pattern does not always permit us to know the type of its subpatterns. In the previous example the cause is function co, because its pattern $snd\ X$ is *opaque* and shadows the type of its subpattern X. Usual inference algorithms treat this opacity as polymorphism, and that is the reason

why it is inferred a completely polymorphic type for the result of the function *co*.

In [6] the appearance of any opaque pattern in the left-hand side of the rules is prohibited, but we will see that it is possible to be less restrictive. The key is making a distinction between **transparent** and **opaque** variables of a pattern: a variable is transparent if its type is univocally fixed by the type of the pattern, and is opaque otherwise. We call a variable of a pattern **critical** if it is opaque in the pattern and also appears elsewhere in the expression. The formal definition of opaque and critical variables will be given in Sect. 3. With these notions we can relax the situation in [6], prohibiting only those patterns having critical variables.

1.2 Local definitions

Functional and functional logic languages provide syntax to introduce local definitions inside an expression. But in spite of the popularity of let-expressions, different implementations treat them differently because of the polymorphism they give to bound variables. This difference can be observed in Ex. 2, being $(e_1, \ldots, e_n)$ and $[e_1, \ldots, e_n]$ the usual tuple and list notation respectively.

Example 2 (let expressions). Let e_1 be $let\ F = id\ in\ (F\ true, F\ 0)$, and e_2 be $let\ [F, G] = [id, id]\ in\ (F\ true, F\ 0, G\ 0, G\ false)$

Intuitively, e_1 gives a new name to the identity function and uses it twice with arguments of different types. Surprisingly, not all implementations consider this expression as well-typed, and the reason is that F is used with different types in each appearance: $bool \to bool$ and $int \to int$. Some implementations as Clean 2.2, PAKCS 1.9.1 or KICS 0.81893 consider that a variable bound by a let-expression must be used with the same type in all the appearances in the body of the expression. In this situation we say that lets are completely monomorphic, and write let_m for it.

On the other hand, we can consider that all the variables bound by the let-expression may have different but coherent types, i.e., are treated polymorphically. Then expressions like e_1 or e_2 would be well-typed. This is the decision adopted by Hugs Sept. 2006, OCaml 3.10.2 or F# Sept. 2008. In this case, we will say that lets are completely polymorphic, and write let_p.

Finally, we can treat the bound variables monomorphically or polymorphically depending on the form of the pattern. If the pattern is a variable, the let treats it polymorphically, but if it is compound the let treats all the variables monomorphically. This is the case of GHC 6.8.2, SML of New Jersey v110.67 or Curry Münster 0.9.11. In this implementations e_1 is well-typed, while e_2 not. We call this kind of let-expression let_{pm}.

Fig. 1 summarizes the decisions of various implementations of functional and functional logic languages. The exact behavior wrt. types of local definitions is usually not well documented, not to say formalized, in those systems. One of our contributions is this paper is to technically clarify this question by adopting a

Programming language and version	let_m	let_{pm}	let_p
GHC 6.8.2		×	
Hugs Sept. 2006			×
Standard ML of New Jersey 110.67		×	
Ocaml 3.10.2			×
F# Sept. 2008			×
Clean 2.0	×		
$\mathcal{TOY}$ **2.3.1***	×		
Curry PAKCS 1.9.1	×		
Curry Münster 0.9.11		×	
KICS 0.81893	×		

(*) we use `where` instead of `let`, not supported by $\mathcal{TOY}$

Fig. 1. Let expressions in different programming languages.

neutral position, and formalizing the different possibilities for the polymorphism of local definitions.

2 Preliminaries

We assume a signature $\Sigma = DC \cup FS$, where DC and FS are two disjoint sets of *data constructor* and *function* symbols resp., all them with associated arity. We write DC^n (resp FS^n) for the set of constructor (function) symbols of arity n. We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. We define the set of *patterns* $Pat \ni t ::= X \mid c\ t_1 \dots t_n\ (n \leq k) \mid f\ t_1 \dots t_n\ (n < k)$, where $c \in DC^k$ and $f \in FS^k$; and the set of *expressions* $Exp \ni e ::= X \mid c \mid f \mid e_1\ e_2 \mid \lambda t.e \mid let_m\ t = e_1\ in\ e_2 \mid let_{pm}\ t = e_1\ in\ e_2 \mid let_p\ t = e_1\ in\ e_2$ where $c \in DC$, $f \in FS$ and t is a linear pattern. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ t_1 \dots t_n$ where $c \in DC^n$, and *Higher order patterns* $HOPat = Pat \setminus FOPat$. Expressions $h\ e_1 \dots e_n$ are called *junk* if $h \in CS^k$ and $n > k$, and *active* if $h \in FS^k$ and $n \geq k$. $FV(e)$ is the set of variables in e which are not bound by any lambda or let expression and is defined in the usual way (notice that since our let expressions do not support recursive definitions the bindings of the pattern only affect e_2: $FV(let_*\ t = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \setminus var(t))$. A *one-hole context* $\mathcal{C}$ is an expression with exactly one hole. A *data substitution* $\theta \in \mathcal{PS}ubst$ is a finite mapping from data variables to patterns: $[\overline{X_i/t_i}]$. Substitution application over data variables and expressions is defined in the usual way. A *program rule* is defined as $PRule \ni r ::= f\ t_1 \dots t_n \to e\ (n \geq 0)$ where the set of patterns $\overline{t_i}$ is linear and $FV(e) \subseteq \bigcup_i var(t_i)$. Therefore, extra variables are not considered in this paper. A program is a set of program rules $Prog \ni \mathcal{P} ::= \{r_1; \dots; r_n\} (n \geq 0)$.

For the types we assume a denumerable set $\mathcal{TV}$ of *type variables* α and a countable alphabet $\mathcal{TC} = \bigcup_{n \in \mathbb{N}} \mathcal{TC}^n$ of *type constructors* C. The set of *simple types* is defined as $SType \ni \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid C\ \tau_1 \dots \tau_n\ (C \in \mathcal{TC}^n)$. Based on simple types we define the set of *type-schemes* as $TScheme \ni \sigma ::= \tau \mid \forall \alpha.\sigma$. The

set of *free type variables* (FTV) of a simple type τ is $var(\tau)$, and for type-schemes $FTV(\forall\overline{\alpha_i}.\tau) = FTV(\tau) \setminus \{\overline{\alpha_i}\}$. A type-scheme $\forall\overline{\alpha_i}.\overline{\tau_n} \to \tau$ is *transparent* if $FTV(\overline{\tau_n}) \subseteq FTV(\tau)$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_i : \sigma_i}\}$, where $s_i \in DC \cup FS \cup \mathcal{DV}$. Notice that the transparency of type-schemes for data constructors is not required in our setting, although that hypothesis is usually assumed in classical Damas & Milner type systems. If $(s_i : \sigma_i) \in \mathcal{A}$ we write $\mathcal{A}(s_i) = \sigma_i$. A *type substitution* $\pi \in \mathcal{TS}ubst$ is a finite mapping from type variables to simple types $[\overline{\alpha_i/\tau_i}]$. For sets of assumptions $FTV(\{\overline{s_i : \sigma_i}\}) = \bigcup_i FTV(\sigma_i)$. We will say a type-scheme σ is *closed* if $FTV(\sigma) = \emptyset$. Application of type substitutions to simple types is defined in the natural way, and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions in the obvious way. We will say σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. τ' is a *generic instance* of $\sigma \equiv \forall\overline{\alpha_i}.\tau$ if $\tau' = \tau[\overline{\alpha_i/\tau_i}]$ for some $\overline{\tau_i}$, and we write it $\sigma \succ \tau'$. We extend $\succ$ to a relation between type-schemes by saying that $\sigma \succ \sigma'$ iff every simple type such that is a generic instance of σ' is also a generic instance of σ. Then $\forall\overline{\alpha_i}.\tau \succ \forall\overline{\beta_i}.\tau[\overline{\alpha_i/\tau_i}]$ iff $\{\overline{\beta_i}\} \cap FTV(\forall\overline{\alpha_i}.\tau) = \emptyset$ [3]. Finally, τ' is a *variant* of $\sigma \equiv \forall\overline{\alpha_i}.\tau$ ($\sigma \succ_{var} \tau'$) if $\tau' = \tau[\overline{\alpha_i/\beta_i}]$ and $\overline{\beta_i}$ are fresh type variables.

3 Type derivation

We propose a modification of Damas & Milner type system [4] with some differences. We have found convenient to separate the task of giving a regular Damas & Milner type and the task of checking critical variables. To do that we have defined two different type relations: $\vdash$ and $\vdash^\bullet$.

The basic typing relation $\vdash$ in the upper part of Fig. 2 is like the classical Damas & Milner's system but extended to handle the three different kinds of let expressions and the occurrence of patterns instead of variables in lambda and let expressions. We have also made the rules more syntax-directed so that the form of type derivations depends only on the form of the expression to be typed. $Gen(\tau, \mathcal{A})$ is the closure or generalization of τ wrt. $\mathcal{A}$ [4, 3, 19], which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall\overline{\alpha_i}.\tau$ *where* $\{\overline{\alpha_i}\} = FTV(\tau) \setminus FTV(\mathcal{A})$. As can be seen, [LET$_m$] and [LET$^h_{pm}$] behave the same, and do not generalize any of the types τ_i for the variables X_i to give a type for the body. On the contrary, [LET$^X_{pm}$] and [LET$_p$] generalize the types given to the variables. Notice that if two variables share the same type in the set of assumptions $\mathcal{A}$, generalization will lose the connection between them. This fact can be seen with e_2 in Ex. 2. Although the type for both F and G can be $\alpha \to \alpha$ (with α a variable not appearing in $\mathcal{A}$) the generalization step will assign both the type-scheme $\forall\alpha.\alpha \to \alpha$, losing the connection between them. Fig. 3 shows a type derivation for the expression $\lambda(snd\ X).X$.

The $\vdash^\bullet$ relation (lower part of Fig. 2) uses $\vdash$ but enforces also the absence of critical variables. A variable X_i is *opaque* in t when it is possible to build a type derivation for t where the type assumed for X_i contains type variables which do not occur in the type derived for the pattern. The formal definition is as follows.

$$[\mathbf{ID}]\quad \frac{}{\mathcal{A} \vdash s : \tau} \quad \text{if } \begin{array}{l} s \in DC \cup FS \cup \mathcal{DV} \\ \wedge\ (s : \sigma) \in \mathcal{A} \wedge\ \sigma \succ \tau \end{array}$$

$$[\mathbf{APP}]\quad \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$[\Lambda]\quad \frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau} \quad \text{if } \{\overline{X_i}\} = var(t)$$

$$[\mathbf{LET}_m]\quad \frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad \text{if } \{\overline{X_i}\} = var(t)$$

$$[\mathbf{LET}^X_{pm}]\quad \frac{\mathcal{A} \vdash e_1 : \tau_1 \quad \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A})\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_{pm}\ X = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

$$[\mathbf{LET}^h_{pm}]\quad \frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash h\ t_1 \ldots t_n : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad \text{if } \begin{array}{l} \{\overline{X_i}\} = var(t_1 \ldots t_n) \\ \wedge\ h \in DC \cup FS \end{array}$$

$$[\mathbf{LET}_p]\quad \frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : Gen(\tau_i, \mathcal{A})}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad \text{if } \{\overline{X_i}\} = var(t)$$

$$[\mathbf{P}]\quad \frac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash^{\bullet} e : \tau} \quad \text{if } critVar_{\mathcal{A}}(e) = \emptyset$$

Fig. 2. Rules of type system

Assuming $\mathcal{A} \equiv \{snd : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta\}$ and $\mathcal{A}' \equiv \mathcal{A} \oplus \{X : \gamma\}$

$$[\Lambda]\ \frac{[\mathbf{APP}]\ \dfrac{(*)}{\mathcal{A} \oplus \{X : \gamma\} \vdash snd\ X : bool \to bool} \qquad [\mathbf{ID}]\ \dfrac{}{\mathcal{A}' \vdash X : \gamma}}{\mathcal{A} \vdash \lambda(snd\ X).X : (bool \to bool) \to \gamma}$$

where the type derivation for $(*)$ is:

$$[\mathbf{APP}]\ \frac{[\mathbf{ID}]\ \dfrac{}{\mathcal{A}' \vdash snd : \gamma \to bool \to bool} \qquad [\mathbf{ID}]\ \dfrac{}{\mathcal{A}' \vdash X : \gamma}}{\mathcal{A}' \vdash snd\ X : bool \to bool}$$

Fig. 3. Example of type derivation using $\vdash$

Definition 1 (Opaque variable of t wrt. $\mathcal{A}$). *Let t be a pattern that admits type wrt. a given set of assumptions $\mathcal{A}$. We say that $X_i \in \overline{X_i} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\exists \overline{\tau_i}, \tau$ s.t. $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$ and $FTV(\tau_i) \not\subseteq FTV(\tau)$.*

Example 3 (Opaque variables of t wrt. $\mathcal{A}$).

- We will see that X is an opaque variable in *snd* X wrt. any set of assumptions $\mathcal{A}_1$ containing the usual type-scheme for *snd* ($snd : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta$) and any type assumption for X. It is clear that *snd* X admits a type wrt. that $\mathcal{A}_1$, e.g. $bool \to bool$ (see Fig. 3). However we can build the type derivation $\mathcal{A}_1 \oplus \{X : \gamma\} \vdash snd\ X : bool \to bool$ such that $FTV(\gamma) = \{\gamma\} \not\subseteq \emptyset = FTV(bool \to bool)$.
- On the other hand we can see that X is not opaque in *snd* $[X, true]$. It corresponds to the intuition, since in this case the pattern itself fixes univocally the type of the variable X. Consider a set of assumptions $\mathcal{A}_2$ containing the usual type-schemes for *snd* and the list constructors, and the assumption $\{X : bool\}$. Clearly *snd* $[X, true]$ admits type wrt. $\mathcal{A}_2$. The only assumption for X that we can add to $\mathcal{A}_2$ in order to derive a type for *snd* $[X, true]$ is $\{X : bool\}$, otherwise the subpattern $[X, true]$ would not admit any type. Therefore any type derivation has to be of the shape $\mathcal{A}_2 \oplus \{X : bool\} \vdash snd\ [X, true] : \tau$, and obviously $FTV(bool) = \emptyset \subseteq FTV(\tau)$, for any τ.

Def. 1 is based on the existence of a certain type derivation, and therefore cannot be used as an effective check for the opacity of variables. Prop. 1 provides a more operational characterization of opacity that exploits the close relationship between $\vdash$ an type inference $\Vdash$ presented in Sect. 4.

Proposition 1. *$X_i \in \overline{X_i} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_g | \pi_g$ and $FTV(\alpha_i \pi_g) \not\subseteq FTV(\tau_g)$.*

We write $opaqueVar_{\mathcal{A}}(t)$ for set of opaque variables of t wrt. $\mathcal{A}$. Now, we can define the *critical variables* of an expression e wrt. $\mathcal{A}$ as those variables that, being opaque in a let or lambda pattern of e, are indeed used in e. Formally:

Definition 2 (Critical variables).

$$critVar_{\mathcal{A}}(s) = \emptyset \quad if\ s \in DC \cup FS \cup \mathcal{DV}$$
$$critVar_{\mathcal{A}}(e_1\ e_2) = critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$$
$$critVar_{\mathcal{A}}(\lambda t.e) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e)) \cup critVar_{\mathcal{A}}(e)$$
$$critVar_{\mathcal{A}}(let_*\ t = e_1\ in\ e_2) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e_2)) \cup\ critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$$

Notice that the if we write the function *unpack* of Ex. 1 as $\lambda(snd\ X).X$, it is well-typed wrt. $\vdash$ using the usual type for *snd*. However it is ill-typed wrt. $\vdash^\bullet$ since X is a critical variable, i.e., it is an opaque variable in *snd* X and it occurs in the body of the λ-abstraction.

The typing relation $\vdash^\bullet$ has been defined in a modular way in the sense that the opacity check is kept separated from the regular Damas & Milner typing. Therefore it is easy to see that if every constructor and function symbol in

program has a transparent assumption, then all the variables in patterns will be transparent, and so $\vdash^\bullet$ will be equivalent to $\vdash$. This happens in particular for those programs using only first order patterns and whose constructor symbols come from a Haskell (or Toy, Curry)-like `data` declaration.

3.1 Properties of the typing relations

The typing relations fulfill a set of useful properties. Here we use $\vdash^?$ for any of the two typing relations: $\vdash$ or $\vdash^\bullet$.

Theorem 1 (Properties of the typing relations).

a) If $\mathcal{A} \vdash^? e : \tau$ *then* $\mathcal{A}\pi \vdash^? e : \tau\pi$*, for any* $\pi \in \mathcal{TS}ubst$*.*

b) Let $s \in DC \cup FS \cup \mathcal{DV}$ *be a symbol not occurring in* e*. Then* $\mathcal{A} \vdash^? e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash^? e : \tau$*.*

c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e[X/e'] : \tau$*.*

d) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$*, then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$*.*

Part *a)* states that type derivations are closed under type substitutions. *b)* shows that type derivations for e depend only on the assumptions for the symbols in e. *c)* is a substitution lemma stating that in a type derivation we can replace a variable by an expression with the same type. Finally, *d)* establishes that from a valid type derivation we can change the assumption of a symbol for a more general type-scheme, and we still have a correct type derivation for the same type. Notice that this is not true wrt. the typing relation $\vdash^\bullet$ because a more general type can introduce opacity. For example the variable X is opaque in *snd X* with the usual type for *snd*, but with a more specific type such as *bool* $\to$ *bool* $\to$ *bool* it is no longer opaque.

3.2 Subject Reduction

Subject reduction is a key property for type systems, meaning that evaluation does not change the type of an expression. This ensures that run-time type errors will not occur. Subject reduction is only guaranteed for *well-typed* programs, a notion that we formally define now.

Definition 3 (Well-typed program). *A program rule* $f\ t_1 \ldots t_n \to e$ *is well-typed wrt.* $\mathcal{A}$ *if* $\mathcal{A} \vdash^\bullet \lambda t_1 \ldots \lambda t_n . e : \tau$ *and* τ *is a variant of* $\mathcal{A}(f)$*. A program* $\mathcal{P}$ *is well-typed wrt.* $\mathcal{A}$ *if all its rules are well-typed wrt.* $\mathcal{A}$*. If* $\mathcal{P}$ *is well-typed wrt.* $\mathcal{A}$ *we write* $wt_{\mathcal{A}}(\mathcal{P})$*.*

Notice the use of the extended typing relation $\vdash^\bullet$ in the previous definition. This is essential, as we will explain later. Returning to Ex. 1, we can see that the program will not be well-typed because of the rule *unpack* (*snd X*) $\to X$, since λ(*snd X*).X will be ill-typed wrt. the usual type for *snd*, as we explained before.

$$
\begin{array}{rl}
TRL(s) = s, & \text{if } s \in DC \cup FS \cup \mathcal{DV} \\
TRL(e_1\ e_2) = TRL(e_1)\ TRL(e_2) & \\
TRL(let_K\ X = e_1\ in\ e_2) = let_K\ X = TRL(e_1)\ in\ TRL(e_2), & \text{with } K \in \{m, p\} \\
TRL(let_{pm}\ X = e_1\ in\ e_2) = let_p\ X = TRL(e_1)\ in\ TRL(e_2) & \\
TRL(let_m\ t = e_1\ in\ e_2) = let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) & \\
TRL(let_{pm}\ t = e_1\ in\ e_2) = let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) & \\
TRL(let_p\ t = e_1\ in\ e_2) = let_p\ Y = TRL(e_1)\ in\ \overline{let_p\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) &
\end{array}
$$

for $\{\overline{X_i}\} = var(t) \cap FV(e_2)$, $f_{X_i} \in FS^1$ fresh defined by the rule $f_{X_i}\ t \to X_i$, $Y \in \mathcal{DV}$ fresh, t a non variable pattern.

Fig. 4. Transformation rules of let expressions with patterns

Although the restriction that the type of the lambda abstraction associated to a rule must be a variant of the type of the function symbol (and not an instance) might seem strange, it is necessary. Otherwise, the fact that a program is well-typed will not give us important information about the functions like the type of their arguments, and will make us to consider as well-typed undesirable programs like $\mathcal{P} \equiv \{f\ true \to true; f\ 2 \to false\}$ with the assumptions $\mathcal{A} \equiv \{f :: \forall\alpha.\alpha \to bool\}$. Besides, this restriction is implicitly considered in [6].

For subject reduction to be meaningful, a notion of evaluation is needed. In this paper we consider the *let-rewriting* relation of [11]. As can be seen, *let-rewriting* does not support let expressions with compound patterns. Instead of extending the semantics with this feature we propose a transformation from let-expressions with patterns to let-expressions with only variables (Fig. 4). There are various ways to perform this transformation, which differ in the strictness of the pattern matching. We have chosen the alternative explained in [17] that does not demand the matching if no variable of the pattern is needed, but otherwise forces the matching of the whole pattern. This transformation has been enriched with the different kinds of let expressions in order to preserve the types, as is stated in Th. 2. Notice that the result of the transformation and the expressions accepted by *let-rewriting* only has let_m or let_p expressions, since without compound patterns let_{pm} is the same as let_p. Finally, we have added polymorphism annotations to let expressions (Fig. 5). Original **(Flat)** rule has been split into two, one for each kind of polymorphism. Although both behave the same from the point of view of values, the splitting is needed to guarantee type preservation. λ-abstractions have been omitted, since they are not supported by *let-rewriting*.

Theorem 2 (Type preservation of the let transformation). *Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_i}\ t_i \to X_i}\}$ be the rules of the projection functions needed in the transformation of e according to Fig. 4. Let also $\mathcal{A}'$ be the set of assumptions over that functions, defined as $\mathcal{A}' \equiv \{\overline{f_{X_i} : Gen(\tau_{X_i}, \mathcal{A})}\}$, where $\mathcal{A} \Vdash^\bullet \lambda t_i.X_i : \tau_{X_i}|\pi_{X_i}$. Then $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet TRL(e) : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.*

Th. 2 also states that the projection functions are well-typed. Then if we start from a well-typed program $\mathcal{P}$ wrt. $\mathcal{A}$ and apply the transformation to all its rules, the program extended with the projections rules will be well-typed

(Fapp) $f\ t_1\theta \ldots t_n\theta \ \rightarrow^l\ r\theta, \quad$ if $(f\ t_1 \ldots t_n \rightarrow r) \in \mathcal{P}$ and $\theta \in \mathcal{PS}ubst$

(LetIn) $e_1\ e_2 \rightarrow^l let_m\ X = e_2\ in\ e_1\ X, \quad$ if e_2 is an active expression, variable application, junk or *let* rooted expression, for X fresh.

(Bind) $let_K\ X = t\ in\ e\ \rightarrow^l\ e[X/t], \quad$ if $t \in Pat$

(Elim) $let_K\ X = e_1\ in\ e_2 \rightarrow^l e_2, \quad$ if $X \notin FV(e_2)$

(Flat$_m$) $let_m\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightarrow^l\ let_K\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3), \quad$ if $Y \notin FV(e_3)$

(Flat$_p$) $let_p\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightarrow^l\ let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(LetAp) $(let_K\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let_K\ X = e_1\ in\ e_2\ e_3, \quad$ if $X \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \rightarrow^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \rightarrow^l e'$ using any of the previous rules

where $K \in \{m, p\}$

Fig. 5. Higher order *let*-rewriting relation $\rightarrow^l$

wrt. the extended assumptions: $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P} \uplus \mathcal{P}')$. This result is straightforward, because $\mathcal{A}'$ does not contain any assumption for the symbols in $\mathcal{P}$, so $wt_{\mathcal{A}}(\mathcal{P})$ implies $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

Th. 3 states the subject reduction property for a *let-rewriting* step, but its extension to any number of steps is trivial.

Theorem 3 (Subject Reduction). *If $\mathcal{A} \vdash^\bullet e : \tau$ and $wt_{\mathcal{A}}(\mathcal{P})$ and $\mathcal{P} \vdash e \rightarrow^l e'$ then $\mathcal{A} \vdash^\bullet e' : \tau$.*

For this result to hold it is essential that the definition of well-typed program relies on $\vdash^\bullet$. A counterexample can be found in Ex. 1, where the program would be well-typed wrt. $\vdash$ but the subject reduction property fails for *and* (*cast* 0) *true*.

The proof of the subject reduction property is based on the following lemma, an important auxiliary result about the instantiation of transparent variables. Intuitively it states that if we have a pattern t with type τ and we change its variables by other expressions, the only way to obtain the same type τ for the substituted pattern is by changing the transparent variables for expressions with the same type. This is not guaranteed with opaque variables, and that is why we forbid their use in expressions.

Lemma 1. *Assume $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$, where $var(t) \subseteq \{\overline{X_i}\}$. If $\mathcal{A} \vdash t[\overline{X_i/s_i}] : \tau$ and X_j is a transparent variable of t wrt. $\mathcal{A}$ then $\mathcal{A} \vdash s_j : \tau_j$.*

4 Type inference for expressions

The typing relation $\vdash^\bullet$ lacks some properties that prevent its usage as a type-checker mechanism in a compiler for a functional logic language. First, in spite

[iID]
$$\frac{}{\mathcal{A} \Vdash s : \tau | id} \quad if \begin{array}{l} s \in DC \cup FS \cup \mathcal{DV} \\ \wedge\ (s : \sigma) \in \mathcal{A} \wedge \sigma \succ_{var} \tau \end{array}$$

[iAPP]
$$\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi | \pi_1\pi_2\pi} \quad if \begin{array}{l} \alpha\ fresh\ type\ variable \\ \wedge\ \pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha) \end{array}$$

[iΛ]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t \Vdash e : \tau | \pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau | \pi_t\pi} \quad if \begin{array}{l} \{\overline{X_i}\} = var(t) \\ \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \end{array}$$

[iLET$_m$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$^X_{pm}$]
$$\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ X = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_1\pi_2}$$

[iLET$^h_{pm}$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash h\ t_1 \ldots t_n : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ h \in DC \cup FS \wedge\ \{\overline{X_i}\} = var(h\ t_1 \ldots t_n) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$_p$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_i : Gen(\alpha_i\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iP]
$$\frac{\mathcal{A} \Vdash e : \tau | \pi}{\mathcal{A} \Vdash^{\bullet} e : \tau | \pi} \quad if\ critVar_{\mathcal{A}\pi}(e) = \emptyset$$

Fig. 6. Inference rules

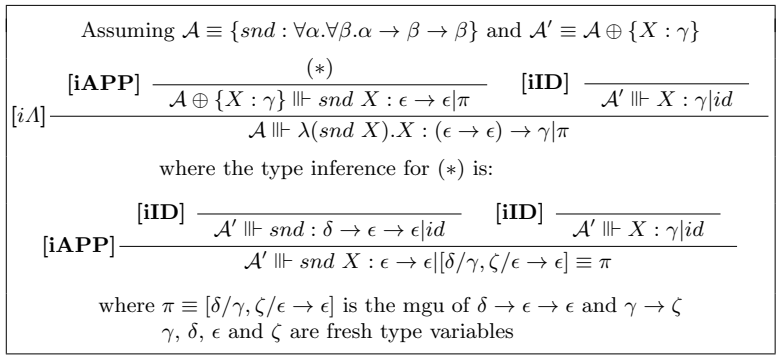


Fig. 7. Example of type inference using $\Vdash$

of the syntax-directed style, the rules for $\vdash$ and $\vdash^\bullet$ have a bad operational behavior: at some steps they need to guess a type. Second, the types related to an expression can be infinite due to polymorphism. Finally, the typing relation needs all the assumptions for the symbols in order to work. To overcome these problems, type systems usually are accompanied with a type inference algorithm which returns a valid type for an expression and also establishes the types for some symbols in the expression.

In this work we have given the type inference in Fig. 6 a relational style to show the similarities with the typing relation. But in essence, the inference rules represent an algorithm (similar to algorithm $\mathcal{W}$ [4, 3]) which fails if any of the rules cannot be applied. This algorithm accepts a set of assumptions $\mathcal{A}$ and an expression e, and returns a simple type τ and a type substitution π. Intuitively, τ will be the "most general" type which can be given to e, and π the "minimum" substitution we have to apply to $\mathcal{A}$ in order to able to derive a type for e. Fig. 7 contains an example of type inference for the expression $\lambda(snd\ X).X$.

Th. 4 shows that the type and substitution found by the inference are correct, i.e., we can build a type derivation for the same type if we apply the substitution to the assumptions.

Theorem 4 (Soundness of $\Vdash^?$). $\mathcal{A} \Vdash^? e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash^? e : \tau$

Th. 5 expresses the completeness of the inference process. If we can derive a type for an expression applying a substitution to the assumptions, then inference will succeed and will find a type and a substitution which are the most general ones.

Theorem 5 (Completeness of $\Vdash$ wrt $\vdash$). *If* $\mathcal{A}\pi' \vdash e : \tau'$ *then* $\exists\tau, \pi, \pi''.\ \mathcal{A} \Vdash e : \tau|\pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$.

A result similar to Th. 5 cannot be obtained for $\Vdash^\bullet$ because of critical variables, as the following example 4 shows.

Example 4 (Inexistence of a most general typing substitution). Let $\mathcal{A} \equiv \{snd' : \alpha \to bool \to bool\}$ and consider the following two valid derivations $\mathcal{D}_1 \equiv \mathcal{A}[\alpha/bool] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to bool$ and $\mathcal{D}_2 \equiv \mathcal{A}[\alpha/int] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to int$. It is clear that there is not a substitution more general than $[\alpha/bool]$ and $[\alpha/int]$ which makes possible a type derivation for $\lambda(snd'\ X).X$. The only substitution more general than these two will be $[\alpha/\beta]$ (for some β), converting X in a critical variable.

In spite of this, we will see that $\Vdash^\bullet$ is still able to find the most general substitution when it exists. To formalize that, we will use the notion of $\Pi^\bullet_{\mathcal{A},e}$, which denotes the set collecting all type substitution π such that $\mathcal{A}\pi$ gives some type to e.

Definition 4 (Typing substitutions of e).
$\Pi^\bullet_{\mathcal{A},e} = \{\pi \in \mathcal{TS}ubst \mid \exists \tau \in SType.\ \mathcal{A}\pi \vdash^\bullet e : \tau\}$

Now we are ready to formulate our result regarding the maximality of $\Vdash^\bullet$.

Theorem 6 (Maximality of $\Vdash^\bullet$).

a) *$\Pi^\bullet_{\mathcal{A},e}$ has a maximum element* $\iff \exists \tau_g \in SType, \pi_g \in \mathcal{TS}ubst.\ \mathcal{A} \Vdash^\bullet e : \tau_g|\pi_g$.

b) *If $\mathcal{A}\pi' \vdash^\bullet e : \tau'$ and $\mathcal{A} \Vdash^\bullet e : \tau|\pi$ then exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$.*

5 Type inference for programs

In the functional programming setting, type inference does not need to distinguish between programs and expressions, because the program can be incorporated in the expression by means of let expressions and λ-abstractions. This way, the results given for expressions are also valid for programs. But in our framework it is different, because our semantics (*let-rewriting*) does not support λ-abstractions and our let expressions do not define new functions but only perform pattern matching. Thereby in our case we need to provide an explicit method for inferring the types of a whole program. By doing so, we will also provide a specification closer to implementation.

The type inference procedure for a program takes a set of assumptions $\mathcal{A}$ and a program $\mathcal{P}$ and returns a type substitution π. The set $\mathcal{A}$ must contain assumptions for all the symbols in the program, even for the functions defined in $\mathcal{P}$. We want to reflect the fact that in practice some defined functions may come with an explicit type declaration. Indeed this is a frequent way of documenting a program. Furthermore, type declarations are sometimes a real need, for instance if we want the language to support *polymorphic recursion* [16, 10]. Therefore, for some of the functions –those for which we want to infer types– the assumption will be simply a fresh type variable, to be instantiated by the inference process. For the rest, the assumption will be a closed type-scheme, to be checked by the procedure.

Definition 5 (Type Inference of a Program). *The procedure $\mathcal{B}$ for type inference of a program $\{rule_1, \ldots, rule_m\}$ is defined as:*

$$\mathcal{B}(\mathcal{A}, \{rule_1, \ldots, rule_m\}) = \pi, \textit{if}$$

1. $\mathcal{A} \Vdash^{\bullet} (\varphi(rule_1), \ldots, \varphi(rule_m)) : (\tau_1, \ldots, \tau_m)|\pi$.
2. *Let $f^1 \ldots f^k$ be the function symbols of the rules $rule_i$ in $\mathcal{P}$ such that $\mathcal{A}(f^i)$ is a closed type-scheme, and τ^i the type obtained for $rule_i$ in step 1. Then τ^i must be a variant of $\mathcal{A}(f^i)$.*

φ is a transformation from rules to expressions defined as:

$$\varphi(f\ t_1 \ldots t_n \to e) = pair\ \lambda t_1 \ldots . \lambda t_n . e\ f$$

*where $()$ is the usual tuple constructor, with type $() : \forall \overline{\alpha_i}.\alpha_1 \to \ldots \alpha_m \to (\alpha_1, \ldots, \alpha_m)$; and **pair** is a special constructor of tuples of two elements of the same type, with type **pair** $: \forall \alpha.\alpha \to \alpha \to \alpha$.*

Example 5 (Type Inference of Programs).

- Consider the program $\mathcal{P}$ consisting in the rules $\{ugly\ true \to true, ugly\ 0 \to true\}$ and the set of assumptions $\mathcal{A} \equiv \{ugly : \forall \alpha.\alpha \to bool\}$. Our intuition advises us to reject this program because the type of *ugly* expresses parametric polymorphism, and the rules are not parametric but defined for arguments whose types are not compatible. Using procedure $\mathcal{B}$ we will first infer the type for the expression associated to the program, getting

  $\mathcal{A} \Vdash^{\bullet} (pair\ \lambda true.true\ ugly, pair\ \lambda 0.true\ ugly) : (bool \to bool, int \to bool)|\pi$

  for some π that affects only type variables generated during the inference. Since *ugly* has a closed type-scheme in $\mathcal{A}$ then we will check that the types $bool \to bool$ and $int \to bool$ inferred for its rules are variants of $\forall \alpha.\alpha \to bool$. This check will fail, therefore the procedure $\mathcal{B}$ will reject the program.
- Consider the program $\mathcal{P} \equiv \{and\ true\ X \to X, and\ false\ X \to false, id\ X \to X\}$ and the set of assumptions $\mathcal{A} \equiv \{and : \beta, id : \forall \alpha.\alpha \to \alpha\}$. In this case we want to infer the type for *and* (instantiating type variable β) and check that the type for *id* is correct. Using procedure $\mathcal{B}$, in the first step we infer the type for the expression associated to the program:

  $\mathcal{A} \Vdash^{\bullet} (pair\ \lambda true.\lambda X.X\ and, pair\ \lambda false.\lambda X.false\ and, pair\ \lambda X.X\ id) : (bool \to bool \to bool, bool \to bool \to bool, \gamma \to \gamma) : [\beta / bool \to bool \to bool]^1$

  Therefore the type inferred for *and* would be the expected one: $bool \to bool \to bool$. Since *id* has a closed type-scheme in $\mathcal{A}$ then the second step will check the type inferred $\gamma \to \gamma$ is a variant of $\forall \alpha.\alpha \to \alpha$. The check is correct, therefore $\mathcal{B}$ succeeds with the substitution $[\beta / bool \to bool \to bool]$.

The procedure $\mathcal{B}$ has two important properties. It is sound: if the procedure $\mathcal{B}$ finds a substitution π then the program $\mathcal{P}$ is well-typed with respect to the

[1] Note that the bindings for type variables which are not free in $\mathcal{A}$ have been omitted here for the sake of conciseness.

assumptions $\mathcal{A}\pi$ (Th. 7). And second, if the procedure $\mathcal{B}$ succeeds it finds the most general typing substitution (Th. 8). It is not true in general that the existence of a well-typing substitution π' implies the existence of a most general one. A counterexample of this fact is very similar to Ex. 4.

Theorem 7 (Soundness of $\mathcal{B}$). *If $\mathcal{B}(\mathcal{A},\mathcal{P}) = \pi$ then $wt_{\mathcal{A}\pi}(\mathcal{P})$.*

Theorem 8 (Maximality of $\mathcal{B}$). *If $wt_{\mathcal{A}\pi'}(\mathcal{P})$ and $\mathcal{B}(\mathcal{A},\mathcal{P}) = \pi$ then $\exists\pi''$ such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.*

Notice that types inferred for the functions are simple types. In order to obtain type-schemes we need and extra step of generalization, as discussed in the next section.

5.1 Stratified Type Inference of a Program

It is known that splitting a program into blocks of mutually recursive functions and inferring the types in order may reduce the need of providing explicit type-schemes. This situation is shown in the next example.

Example 6 (Program Inference vs Stratified Inference).
$\mathcal{A} \equiv \{true : bool, 0 : int, id : \alpha, f : \beta, g : \gamma\}$
$\mathcal{P} \equiv \{id\ X \to X; f \to id\ true; g \to id\ 0\}$
$\mathcal{P}_1 \equiv \{id\ X \to X\}$, $\mathcal{P}_2 \equiv \{f \to id\ true\}$, $\mathcal{P}_3 \equiv \{g \to id\ 0\}$

An attempt to apply the procedure $\mathcal{B}$ to infer types for the whole program fails because it is not possible for id to have types $bool \to bool$ and $int \to int$ at the same time. We will need to provide explicitly the type-scheme for $id : \forall\alpha.\alpha \to \alpha$ in order to the type inference to succeed, yielding types $f : bool \to bool$ and $g : int \to int$. But this is not necessary if we first infer types for $\mathcal{P}_1$, obtaining $\delta \to \delta$ for id which will be generalized to $\forall\delta.\delta \to \delta$. With this assumption the type inference for both programs $\mathcal{P}_2$ and $\mathcal{P}_3$ will succeed with the expected types.

A general *stratified inference* procedure can be defined in terms of the basic inference $\mathcal{B}$. First, it calculates the graph of strongly connected components from the dependency graph of the program, using e.g. Kosaraju or Tarjan's algorithm [20]. Each strongly connected component will contain mutually dependent functions. Then it will infer types for every component (using $\mathcal{B}$) in topological order, generalizing the obtained types before following with the next component.

Although stratified inference needs less explicit type-schemes, programs involving polymorphic recursion still require explicit type-schemes in order to infer their types.

6 Conclusions and Future Work

In this paper we have proposed a type system for functional logic languages based on Damas & Milner type system. As far as we know, prior to our work only [6] treats with technical detail a type system for functional logic programming. Our paper makes clear contributions when compared to [6]:

– By introducing the notion critical variables, we are more liberal in the treatment of opaque variables, but still preserving the essential property of subject reduction; moreover, this liberality extends also to data constructors, dropping the traditional restriction of transparency required to them. This is somehow similar to what happens with *existential types* [14] or *generalized abstract datatypes* [9], a connection that we plan to further investigate in the future.
– Our type system considers local pattern bindings and λ-abstractions (also with patterns), that were missing in [6]. In addition to that, we have made a rather exhaustive analysis and formalization of different possibilities for polymorphism in local bindings.
– Subject reduction was proved in [6] wrt. a narrowing calculus. Here we do it wrt. an small-step operational semantics closer to real computations.
– In [6] programs came with explicit type declarations. Here we provide algorithms for inferring types for programs without such declarations that can became part of the type stage of a FL compiler.

We have in mind several lines for future work. As an immediate task we plan to implement and integrate the stratified type inference into the $\mathcal{TOY}$ [12] compiler. Apart from the relation to existential types mentioned above, we are interested in other known extensions of type system, like type classes or generic programming. We also want to generalize the subject reduction property to narrowing, using *let narrowing* reductions of [11], and taking into account known problems [6, 1] in the interaction of HO narrowing and types. Handling extra variables (variables occurring only in right hand sides of rules) is another challenge from the viewpoint of types.

References

1. S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Proc. International Symposium on Functional and Logic Programming (FLOPS 1999)*, pages 335–353, 1999.
2. B. Brassel. Two to three ways to write an unsafe type cast without importing unsafe - Post to the Curry mailing list. http://www.informatik.uni-kiel.de/~curry/listarchive/0705.html, May 2008.
3. L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985.
4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. Symposium on Principles of Programming Languages (POPL 1982)*, pages 207–212, 1982.
5. J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP 1997)*, pages 153–167. MIT Press, 1997.
6. J. González-Moreno, T. Hortalá-González, and Rodríguez-Artalejo, M. Polymorphic types in functional logic programming. In *Journal of Functional and Logic Programming*, volume 2001/S01, pages 1–71, 2001.

7. M. Hanus. Multi-paradigm declarative languages. In *Proc. of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
8. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
9. S. L. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proc. 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 50–61. ACM, 2006.
10. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.
11. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
12. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
13. E. Martin-Martin. Advances in type systems for functional-logic programming. Master's thesis, Universidad Complutense de Madrid. http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf, July 2009.
14. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.
15. J. J. Moreno-Navarro, J. Mariño, A. del Pozo-Pietro, Á. Herranz-Nieva, and J. García-Martín. Adding type classes to functional-logic languages. In *1996 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'96*, pages 427–438, 1996.
16. A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. 6th Colloquium on International Symposium on Programming*, pages 217–228, London, UK, 1984. Springer-Verlag.
17. S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.
18. B. P. Pierce. *Advanced topics in types and programming languages*. MIT Press, Cambridge, MA, USA, 2005.
19. C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.
20. R. Sedgewick. *Algorithms in C++, Part 5: Graph Algorithms*, pages 205–216. Addison-Wesley Professional, 2002.

Under consideration for publication in Math. Struct. in Comp. Science

A Liberal Type System for Functional Logic Programs[†‡]

FRANCISCO JAVIER LÓPEZ-FRAGUAS[1],

ENRIQUE MARTIN-MARTIN[1] *and*

JUAN RODRÍGUEZ-HORTALÁ[1]

[1] *Dpto. de Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain.*

Received 30 January 2012

We propose a new type system for functional logic programming which is more liberal than the classical Damas-Milner usually adopted, but it is also restrictive enough to ensure type soundness. Starting from Damas-Milner typing of expressions we propose a new notion of well-typed program that adds support for type-indexed functions, a particular form of existential types, opaque higher-order patterns and generic functions—as shown by an extensive collection of examples that illustrate the possibilities of our proposal. In the negative side, the types of functions must be declared, and therefore types are checked but not inferred. Another consequence is that parametricity is lost, although the impact of this flaw is limited as "free theorems" were already compromised in functional logic programming because of non-determinism.

1. Introduction

Functional logic programming. Functional logic languages (Hanus, 2007) like Toy (López-Fraguas and Sánchez-Hernández, 1999) or Curry (Hanus (ed.), 2006) have a strong resemblance to lazy functional languages like Haskell (Hudak et al., 2007). A remarkable difference is that functional logic programs (FLP) can be non-confluent, giving raise to so-called *non-deterministic functions*, for which a *call-time choice* semantics (González-Moreno et al., 1999) is adopted. The following program is a simple example, using natural numbers given by the constructors z and s—we follow syntactic conventions of some functional logic languages where function and constructor names are lowercased, and variables are uppercased—and assuming a natural definition for *add*:

$$f\ X \to X \qquad f\ X \to s\ X \qquad double\ X \to add\ X\ X$$

[†] This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465) and GPD (UCM-BSCH-GR35/10-A-910502)

Here, f is non-deterministic ($f\ z$ evaluates both to z and $s\ z$) and, according to call-time choice, *double (f z)* evaluates to z and s *(s z)* but not to $s\ z$. Operationally, call-time choice means that all copies of a non-deterministic subexpression ($f\ z$ in the example) created during reduction share the same value.

In the HO-CRWL[†] approach to FLP (González-Moreno et al., 1997), followed by the Toy system, programs can use *HO-patterns* (essentially, partial applications of function or constructor symbols to other patterns) in left hand sides of function definitions. These patterns are treated in a purely syntactic way, so problems of HO unification are avoided. HO patterns correspond to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics. This is not an exoticism: it is known (López-Fraguas et al., 2008) that extensionality is not a valid principle within the combination of HO, non-determinism and call-time choice. It is also known that *HO-patterns* cause some bad interferences with types: (González-Moreno et al., 2001) and (López-Fraguas et al., 2010) considered that problem, and this paper makes also some contributions in this sense.

All those aspects of FLP play a role in the paper, and Section 3 uses a formal setting according to that. However, most of the paper can be read from a functional programming perspective leaving aside the specificities of FLP. For example, our operational semantics (Section 3.1) supports evaluation of open expressions, i.e., expressions containing free variables, which are forbidden in functional programming. However this feature does not play any relevant role in this paper, so readers can assume that all expressions to reduce are closed.

Types, FLP and genericity. FLP languages are typed languages adopting classical Damas-Milner types (Damas and Milner, 1982). However, their treatment of types is very simple, far away from the impressive set of possibilities offered by functional languages like Haskell: type and constructor classes, existential types, GADTs, generic programming, arbitrary-rank polymorphism ... (Hudak et al., 2007) Some exceptions to this fact are some preliminary proposals for type classes in FLP (Moreno-Navarro et al., 1996; Lux, 2008), where in particular a technical treatment of the type system is absent.

By the term *generic programming* we refer generically to any situation in which a program piece serves for a family of types instead of a single concrete type. Parametric polymorphism as provided the by Damas-Milner system is probably the main contribution to genericity in the functional programming setting. However, in a sense it is 'too generic' and leaves out many functions which are generic by nature, like equality. Type classes (Wadler and Blott, 1989) were invented to deal with those situations. Some further developments of the idea of generic programming (Hinze, 2006) are based on type classes, while others (Hinze and Löh, 2007) have preferred to use simpler extensions of Damas-Milner system, such as GADTs (Cheney and Hinze, 2003; Schrijvers et al., 2009). We propose a modification of Damas-Milner type system that accepts natural definitions of intrinsically generic functions like equality. The following example illustrates the main points of our approach.

[†] CRWL (González-Moreno et al., 1999) stands for *Constructor Based Rewriting Logic*; HO-CRWL is a higher order extension of it.

An introductory example. Consider a program that manipulates Peano natural numbers, booleans and polymorphic lists. Programming a function *size* to compute the number of constructor occurrences in its argument is an easy task in a type-free language with functional syntax:

```
size true → s z        size false → s z
size z → s z           size (s X) → s (size X)
size [ ] → s z         size (X:Xs) → s (add (size X) (size Xs))
```

However, as far as *bool*, *nat* and $[\alpha]$ are different types, this program would be rejected as ill-typed in a language using Damas-Milner system, since we obtain contradictory types for different rules of *size*. This is a typical case where one wants some support for genericity. Type classes certainly solve the problem if you define a class *Sizeable* and declare *bool, nat* and $[\alpha]$ as instances of it. GADT-based solutions would add an explicit representation of types to the encoding of *size* converting it into a so-called *type-indexed* function (Hinze and Löh, 2007). This kind of encoding is also supported by our system (see the *show* function in Example 3.1 and *eq* in Figure 4-b later), but the interesting point is that our approach allows also a simpler solution: the program above becomes well-typed in our system simply by declaring *size* to have the type $\forall\alpha.\alpha \to nat$, of which each rule of *size* gives a more concrete instance. A detailed discussion of the advantages and disadvantages of such liberal declarations appears in Sections 4 and 6.

The proposed well-typedness criterion for programs proceeds rule by rule and requires only a quite simple additional check over usual Damas-Milner type inference performed over both sides of each rule. Here, 'simple' does not mean 'naive'. For example, imposing the type of each function rule to be an instance of the declared type is a too weak requirement, leading easily to type unsafety. To illustrate this, consider the rule $f\ X \to not\ X$ with the assumptions $f : \forall\alpha.\alpha \to bool$, $not : bool \to bool$. The type of the rule is $bool \to bool$, which is an instance of the type declared for f. However, that rule does not preserve the type: the expression $f\ z$ is well-typed according to f's declared type, but reduces to the ill-typed expression $not\ z$. Our notion of well-typedness, roughly explained, requires also that right-hand sides of rules do not restrict the types of variables more than left-hand sides, a condition that is violated in the rule for f above. Definition 3.1 in Section 3.3 states that point with precision, and allows us to prove type soundness for our system. As we will also see in Section 4, our conditions are in some technical sense the most liberal suitable conditions under which reduction preserve types.

Contributions. We give now a list of the main contributions of our work, presenting the structure of the paper at the same time:

— After some preliminaries, in Section 3 we present a novel notion of well-typed program for FLP that induces a simple and direct way of programming type-indexed and generic functions. The approach supports also a particular form of existential types and GADT-like encodings, not available in current FLP systems. Moreover, the use of HO-patterns is ensured to be type-safe, while in current FLP systems it is either unrestricted (and therefore unsafe) or forbidden because of those type-safety problems.

— Section 4 is devoted to the properties of our type system. We prove that well-typed

programs enjoy *type preservation*, an essential property for a type system, and we give a result of maximal liberality while keeping type preservation; then by introducing *failure* rules to the formal operational calculus, we are also able to ensure the *progress* property of well-typed expressions. Based on those results we also state *syntactic soundness* of the type system, in the sense of (Wright and Felleisen, 1992).

— In Section 5 we give a significant collection of examples showing the interest of the proposal. These examples cover type-indexed functions (with an application to the implementation of type classes), existential types, opaque higher-order patterns and generic functions. None of them is supported by existing FLP systems.

— The well-typedness criterion given in this paper provides a valuable alternative to (López-Fraguas et al., 2010) in the management of type-unsoundness problems due to the use of *HO-patterns* in function definitions. Both works, which are technically compared at the end of Section 3.3, improve largely the solutions given previously in (González-Moreno et al., 2001). As concrete advantages of the proposal in this paper, we can type equality, solving known problems of *opaque decomposition* (González-Moreno et al., 2001) (Section 5.1) and, most remarkably, we can type the *apply* function appearing in the HO-to-FO translation used in standard FLP implementations (Section 5.2).

— Finally, we further discuss in Section 6 the strengths and weaknesses of our proposal, and we end up with some conclusions in Section 7.

This is a revised and extended version of a previous conference paper (López-Fraguas et al., 2010).

2. Preliminaries

We assume a signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of *data constructor* and *function* symbols resp., all of them with associated arity. We write CS^n (resp. FS^n) for the set of constructor (function) symbols of arity n, and if a symbol h is in CS^n or FS^n we write $ar(h) = n$. We consider a special constructor $fail \in CS^0$ to represent pattern matching failure in programs as it is also proposed for GADTs (Cheney and Hinze, 2003; Peyton Jones et al., 2006). We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. The notation $\overline{o_n}$ stands for a sequence of n objects $o_1, \ldots, o_n$, where o_i is the i^{th} element in the sequence. Figure 1 shows the syntax of *patterns* $\in Pat$—our notion of values—and *expressions* $\in Exp$. The role of let-bindings is to express sharing of subexpressions, as corresponds to call-time choice semantics. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ fot_1 \ldots fot_n$ where $ar(c) = n$, and *higher-order patterns* $HOPat = Pat \setminus FOPat$, i.e., patterns containing some partial application of a symbol of the signature. Expressions $c\, e_1 \ldots e_n$ are called *junk* if $n > ar(c)$ and $c \neq fail$, and expressions $f\ e_1 \ldots e_n$ are called *active* if $n \geq ar(f)$. The set $fv(e)$ of *free variables* of an expression e is defined in the usual way as the set of variables in e which are not bound by any let construction; notice that free variables in let-bindings are defined as $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$, corresponding to the fact that we do not consider recursive let-bindings. We say that an expression e is *ground* if $fv(e) = \emptyset$. A *one-hole context* is defined as $\mathcal{C} ::= [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$. A *data*

Data variables			$X, Y, Z, \ldots$
Type variables			$\alpha, \beta, \gamma, \ldots$
Data constructors			c
Type constructors			C
Function symbols			f
Symbol	s	$::=$	$X \mid c \mid f$
Non variable symbol	h	$::=$	$c \mid f$
Expressions	e	$::=$	$X \mid c \mid f \mid e\ e$
		$\mid$	$let\ X = e\ in\ e$
Patterns	t	$::=$	X
		$\mid$	$c\ t_1 \ldots t_n$ if $n \leq ar(c)$
		$\mid$	$f\ t_1 \ldots t_n$ if $n < ar(f)$
Data substitution	θ	$::=$	$[\overline{X_n/t_n}]$
Program rule	R	$::=$	$f\ \overline{t} \to e$ ($\overline{t}$ linear)
Program	$\mathcal{P}$	$::=$	$\{R_1, \ldots, R_n\}$
Simple Types	τ	$::=$	α
		$\mid$	$C\ \tau_1 \ldots \tau_n$ if $ar(C) = n$
		$\mid$	$\tau_1 \to \tau_2$
Type Schemes	σ	$::=$	$\forall\overline{\alpha_n}.\tau$
Type substitution	π	$::=$	$[\overline{\alpha_n/\tau_n}]$
Assumptions	$\mathcal{A}$	$::=$	$\{s_1 : \sigma_1, \ldots, s_n : \sigma_n\}$

Fig. 1. Syntax of expressions, programs and types.

substitution θ is a finite mapping from data variables to patterns: $[\overline{X_n/t_n}]$. Substitution application over data variables and expressions is defined in the usual way. The empty substitution is written as id. A *program rule* R is defined as $f\ \overline{t_n} \to e$ (we also refer to rules as $f\ \overline{t_n} \to r$ or $l \to r$) where the set of patterns $\overline{t_n}$ is linear (there is not repetition of variables), $ar(f) = n$ and $fv(e) \subseteq \bigcup_{i=1}^{n} var(t_i)$. Therefore, extra variables are not considered in this paper. Since the constructor *fail* is an artifact conceived to deal properly with *progress* properties of the type system in Section 4, *fail* is not supposed to occur in program rules, although it would not produce any technical problem. A program $\mathcal{P}$ is a set of program rules: $\{R_1, \ldots, R_n\} (n \geq 0)$.

For the types we assume a denumerable set $\mathcal{TV}$ of *type variables* α and a countable alphabet $\mathcal{TC} = \bigcup_{n \in \mathbb{N}} \mathcal{TC}^n$ of *type constructors* C. As before, if $C \in \mathcal{TC}^n$ then we write $ar(C) = n$. Figure 1 shows the syntax of *simple types* τ and *type-schemes* σ. The set of *free type variables (ftv)* of a simple type τ is $var(\tau)$, and for type-schemes $ftv(\forall\overline{\alpha_n}.\tau) = ftv(\tau) \setminus \{\overline{\alpha_n}\}$. A type-scheme σ is *closed* if $ftv(\sigma) = \emptyset$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_n : \sigma_n}\}$ fulfilling that $\mathcal{A}(fail) = \forall\alpha.\alpha$ and for every c in $CS^n \setminus \{fail\}$, $\mathcal{A}(c) = \forall\overline{\alpha}.\tau_1 \to \ldots \to \tau_n \to (C\ \tau'_1 \ldots \tau'_m)$ for some type constructor C with $ar(C) = m$. Therefore the type assumptions for constructors must correspond to their arity and, as in (Cheney and Hinze, 2003; Peyton Jones et al., 2006), the constructor *fail* can have any type. $\mathcal{A}(s)$

(Fapp) $f\ t_1\theta \ldots t_n\theta \rightarrowtail r\theta$, if $(f\ t_1 \ldots t_n \to r) \in \mathcal{P}$

(Ffail) $f\ t_1 \ldots t_n \rightarrowtail \mathit{fail}$, if $n = ar(f)$ and $\nexists (f\ t'_1 \ldots t'_n \to r) \in \mathcal{P}$ such that $f\ t'_1 \ldots t'_n$ and $f\ t_1 \ldots t_n$ are unifiable

(FailP) $\mathit{fail}\ e \rightarrowtail \mathit{fail}$

(LetIn) $e_1\ e_2 \rightarrowtail \mathit{let}\ X = e_2\ \mathit{in}\ e_1\ X$, if e_2 is junk, active, variable application or *let* rooted, for X fresh

(Bind) $\mathit{let}\ X = t\ \mathit{in}\ e \rightarrowtail e[X/t]$

(Elim) $\mathit{let}\ X = e_1\ \mathit{in}\ e_2 \rightarrowtail e_2$, if $X \notin fv(e_2)$

(Flat) $\mathit{let}\ X = (\mathit{let}\ Y = e_1\ \mathit{in}\ e_2)\ \mathit{in}\ e_3 \rightarrowtail \mathit{let}\ Y = e_1\ \mathit{in}\ (\mathit{let}\ X = e_2\ \mathit{in}\ e_3)$, if $Y \notin fv(e_3)$

(LetAp) $(\mathit{let}\ X = e_1\ \mathit{in}\ e_2)\ e_3 \rightarrowtail \mathit{let}\ X = e_1\ \mathit{in}\ e_2\ e_3$, if $X \notin fv(e_3)$

(Contx) $\mathcal{C}[e] \rightarrowtail \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \rightarrowtail e'$ using any of the previous rules

Fig. 2. **Higher order *let*-rewriting relation with pattern matching failure $\rightarrowtail$**

denotes the type-scheme associated to symbol s, and the union of sets of assumptions is denoted by $\oplus$: $\mathcal{A} \oplus \mathcal{A}'$ contains all the assumptions in $\mathcal{A}'$ and the assumptions in $\mathcal{A}$ over symbols not appearing in $\mathcal{A}'$ (notice that $\oplus$ is not commutative). For sets of assumptions, free type variables are defined as $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. Notice that type-schemes for data constructors may be existential, i.e., they can be of the form $\forall\overline{\alpha_n}.\overline{\tau_m} \to \tau$ where $(\bigcup_{i=1}^{m} ftv(\tau_i)) \setminus ftv(\tau) \neq \emptyset$. A *type substitution* π is a finite mapping from type variables to simple types $[\overline{\alpha_n/\tau_n}]$. Application of type substitutions to simple types is defined in the natural way and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions in the obvious way. We say that σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. A simple type τ' is a *generic instance* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ \tau'$, if $\tau' = \tau[\overline{\alpha_n/\tau_n}]$ for some $\overline{\tau_n}$. Finally, τ' is a *variant* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ_{var} \tau'$, if $\tau' = \tau[\overline{\alpha_n/\beta_n}]$ and $\overline{\beta_n}$ are fresh type variables.

3. Formal setup

3.1. *Operational semantics*

The operational semantics of our programs is based on *let*-rewriting (López-Fraguas et al., 2008), a high level notion of reduction step devised to express call-time choice through the use of let-bindings that represent subexpression sharing. For this paper, we have extended *let*-rewriting with two rules for managing failure of pattern matching (Figure 2), playing a role similar to the rules for pattern matching failure in GADTs (Cheney and Hinze, 2003; Peyton Jones et al., 2006). We write $\rightarrowtail$ for the extended relation and $\mathcal{P} \vdash e \rightarrowtail e'$ ($\mathcal{P} \vdash e \rightarrowtail^* e'$ resp.) to express one step (zero or more steps resp.) of $\rightarrowtail$ using the program $\mathcal{P}$. By $nf_{\mathcal{P}}(e)$ we denote the set of *normal forms* reachable from e, i.e., $nf_{\mathcal{P}}(e) = \{e' \mid \mathcal{P} \vdash e \rightarrowtail^* e' \text{and } e' \text{ is not } \rightarrowtail\text{-reducible}\}$. Notice that *let*-rewriting can reduce expressions with free variables (open expressions), although it does not bind them

to values. However this support for open expressions does not play any relevant role in this paper, which can be understood as if all expressions to reduce were closed.

The new rule (Ffail) generates a failure when no program rule can be used to reduce a function application. Notice the use of syntactic unification[‡] instead of simple pattern matching to check that the variables of the expression will not be able to match the patterns in the rule. This allows us to perform this failure test locally without having to consider the possible bindings for the free variables in the expression caused by the surrounding context. Otherwise, these should be checked in an additional condition for (Contx). To see that, consider for instance the program

$$true \wedge X \to X \qquad false \wedge X \to false$$

and the expression $let\ Y = true\ in\ (Y \wedge true)$. The subexpression $Y \wedge true$ unifies with the function rule left-hand side $true \wedge X$, so no failure is generated. If we use pattern matching as condition without considering the binding $Y = true$, a failure is incorrectly generated since none of the left-hand sides $true \wedge X$ and $false \wedge X$ matches the subexpression $Y \wedge true$. Besides, using unification in (Ffail) also contributes to early detection of proper failures. Consider the program $\mathcal{P}_2 = \{f\ true\ false \to true, loop \to loop\}$ and the expression $let\ Y = loop\ in\ f\ Y\ Y$. Since $f\ Y\ Y$ does not unify with $f\ true\ false$, (Ffail) detects a failure, while other operational approaches to failure in FLP (Sánchez-Hernández, 2006) would lead to divergence.

Finally, rule (FailP) is used to propagate the pattern matching failure when *fail* is applied to another expression.

Extending the *let*-rewriting relation of (López-Fraguas et al., 2008) has been motivated by the desire of distinguishing two kinds of failing reductions that occur in an untyped setting:

— Reductions that cannot progress because of an incomplete function definition, in the sense that the patterns of the function rules do not cover all possible cases for data constructors. A prototypical example is given by the definition *head (x:xs)* $\to$ *x*, where the case *head [ ]* is (intentionally) missing. Similar to what happens in FP systems like Haskell, we expect *(head [ ])* to give raise to a failing reduction, but not to a type error. A difference is that in FP an attempt to evaluate *(head [ ])* will result in a run-time error, while in FLP systems rather than an error this is a silent failure in a possible space of non-deterministic computations that is managed by backtracking. That justifies our choice of the word *fail* instead of *error*.

— Reductions that cannot progress (get *stuck*) because of a genuine type error, as happens for *junk expressions* that apply a non-functional value to some arguments (e.g. *true false*).

Our failure rules (Ffail) and (FailP) try to accomplish with the first kind of reductions. Reductions of the second kind remain stuck even with the added failure rules. As we will

[‡] As mentioned in Section 1, patterns in our setting (both first and higher order patterns) are treated in a purely syntactic way, so syntactic unification is used instead of more complex HO unification procedures.

$$[\mathbf{ID}]\ \frac{}{\mathcal{A} \vdash s : \tau}\ \text{if } \mathcal{A}(s) \succ \tau$$

$$[\mathbf{APP}]\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$[\mathbf{LET}]\ \frac{\mathcal{A} \vdash e_1 : \tau_x \quad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash \mathbf{let}\ X = e_1\ \mathbf{in}\ e_2 : \tau}$$

a) Type derivation rules

$$[\mathbf{iID}]\ \frac{}{\mathcal{A} \Vdash s : \tau | id}\ \text{if } \mathcal{A}(s) \succ_{var} \tau$$

$$[\mathbf{iAPP}]\ \frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi | \pi_1\pi_2\pi}$$
$$\text{if } \alpha \text{ fresh and } \pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha)$$

$$[\mathbf{iLET}]\ \frac{\mathcal{A} \Vdash e_1 : \tau_x | \pi_x \quad \mathcal{A}\pi_x \oplus \{X : Gen(\tau_x, \mathcal{A}\pi_x)\} \Vdash e_2 : \tau | \pi}{\mathcal{A} \Vdash \mathbf{let}\ X = e_1\ \mathbf{in}\ e_2 : \tau | \pi_x\pi}$$

b) Type inference rules

Fig. 3. Type system

see in Section 4, this can only happen to ill-typed expressions. At the end of that section, once the type system and its formal properties have been presented, we further discuss the issues of *fail*-ended and stuck reductions.

3.2. *Type derivation and inference for expressions*

Both derivation and inference rules are based on those presented in (López-Fraguas et al., 2010). Our type derivation rules for expressions (Figure 3-a) correspond to the well-known variation of Damas-Milner's (Damas and Milner, 1982) type system with syntax-directed rules, so there is nothing essentially new here—the novelty will come from the notion of well-typed program given in Definition 3.1 below. $Gen(\tau, \mathcal{A})$ is the closure or generalization of τ wrt. $\mathcal{A}$, which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall\overline{\alpha_n}.\tau\ \ where\ \{\overline{\alpha_n}\} = ftv(\tau) \setminus ftv(\mathcal{A})$. We say that e is well-typed under $\mathcal{A}$, written $wt_{\mathcal{A}}(e)$, if there exists some τ such that $\mathcal{A} \vdash e : \tau$; otherwise it is ill-typed.

The type inference algorithm $\Vdash$ (Figure 3-b) follows the same ideas as the algorithm $\mathcal{W}$ (Damas and Milner, 1982). We have given a relational style to type inference to show the similarities with the typing rules. Nevertheless, the inference rules represent an algorithm that fails if no rule can be applied. This algorithm accepts as inputs a set of assumptions $\mathcal{A}$ and an expression e, and returns a simple type τ and a type substitution π. Intuitively, τ is the "most general" type which can be given to e, and π is the "most general" substitution we have to apply to $\mathcal{A}$ for deriving any type for e.

3.3. *Well-typed programs*

The next definition—the most important in the paper—establishes the conditions that a program must fulfil to be well-typed in our proposal. This definition formalizes in terms of type derivations and substitutions the intuitive well-typedness idea explained in Section

1: right-hand sides of program rules must not restrict the types of variables more than left-hand sides.

Definition 3.1 (Well-typed program wrt. $\mathcal{A}$). The program rule $f\ t_1 \dots t_m \to e$ is *well-typed* wrt. a set of assumptions $\mathcal{A}$, written $wt_{\mathcal{A}}(f\ t_1 \dots t_m \to e)$, iff there exist π_L, τ_L, π_R and τ_R such that:

i) π_L is the most general substitution such that $wt_{(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_L}(f\ t_1 \dots t_m)$, and τ_L is the most general type derivable for $f\ t_1 \dots t_m$ under the assumptions $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_L$.
ii) π_R is the most general substitution such that $wt_{(\mathcal{A} \oplus \{\overline{X_n : \beta_n}\})\pi_R}(e)$, and τ_R is the most general type derivable for e under the assumptions $(\mathcal{A} \oplus \{\overline{X_n : \beta_n}\})\pi_R$.
iii) $\exists \pi.(\tau_L, \overline{\alpha_n \pi_L}) = (\tau_R, \overline{\beta_n \pi_R})\pi$
iv) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$, $\mathcal{A}\pi = \mathcal{A}$

where $\{\overline{X_n}\} = var(f\ t_1 \dots t_m)$ and $\{\overline{\alpha_n}\}, \{\overline{\beta_n}\}$ are fresh type variables. A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$, written $wt_{\mathcal{A}}(\mathcal{P})$, iff all its rules are well-typed.

The first two points check that both right and left-hand sides of the rule can independently have valid types by assigning *some* types to variables, obtaining the most general ones for them in both sides, but not imposing any relationship between them. This is left to the third point, which is the most important one. It checks that the obtained most general types for the right-hand side and the variables appearing in it are more general than the obtained ones for the left-hand side. This point, which avoids that right-hand sides restrict the types of variables more than left-hand sides, guarantees the *type preservation* property (i.e., that the expression resulting after a reduction step has the same type as the original one) when applying a program rule. Moreover, this point ensures a correct management of *opaque variables* (López-Fraguas et al., 2010)—either introduced by the presence of existentially quantified constructors or HO-patterns—which results in the support of a particular variant of existential types (Läufer and Odersky, 1994)—see Section 5.2 for more details. Finally, the last point guarantees that free variables in the set of assumptions are not modified by neither the most general typing substitutions of both sides nor the matching substitution. In practice, this point holds trivially if type assumptions for program functions are closed, as it is usual. Points i) and ii) in the previous definition have are very declarative formulation, but are not particularly well suited to the effective implementation of the well-typedness check. Thanks to the close relationship between type derivation and inference for expressions—soundness and completeness, Theorems A.1 and A.2 in page 27—we can recast points i) and ii) of Definition 3.1 in a more operational and oriented to implementation style.

Definition 3.2 (Well-typed program wrt. $\mathcal{A}$; alternative formulation).
The program rule $f\ t_1 \dots t_m \to e$ is *well-typed* wrt. a set of assumptions $\mathcal{A}$, written $wt_{\mathcal{A}}(f\ t_1 \dots t_m \to e)$, iff there exist π_L, τ_L, π_R and τ_R such that:

i) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t_1 \dots t_m : \tau_L | \pi_L$
ii) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R | \pi_R$
iii) $\exists \pi.(\tau_L, \overline{\alpha_n \pi_L}) = (\tau_R, \overline{\beta_n \pi_R})\pi$
iv) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$, $\mathcal{A}\pi = \mathcal{A}$

where $\{\overline{X_n}\} = var(f\ t_1 \ldots t_m)$ and $\{\overline{\alpha_n}\}, \{\overline{\beta_n}\}$ are fresh type variables. A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$, written $wt_{\mathcal{A}}(\mathcal{P})$, iff all its rules are well-typed.

Now, conditions i) and ii) use the algorithm of type inference for expressions, iii) is just matching, and iv) holds trivially in practice, as we have noticed before; so the implementation is straightforward. The equivalence between both definitions of well-typed rule follows easily from the following result about type derivation and inference:

Lemma 3.1. π is the most general substitution that enables to derive a type for the expression e under the assumptions $\mathcal{A}$, and τ is the most general derivable type for e ($\mathcal{A}\pi \vdash e : \tau) \Longleftrightarrow \exists \pi', \tau'$ such that $\mathcal{A} \Vdash e : \tau' | \pi'$, where π, π' (τ, τ' respectively) are equal up to variable renaming.

Proof. Straightforward based on soundness and completeness of the inference relation wrt. to type derivation (Theorem A.1 and Theorem A.2 in Appendix A). □

Both definitions of well-typed rule present some similarities with the notion of *typeable rewrite rule* for Curryfied Term Rewriting Systems in (van Bakel and Fernández, 1997). In that paper the key condition is that the *principal type* for the left-hand side allows to derive the same type for the right-hand side. This condition is similar to points 1–3 of our definition, which force the most general types obtained for the right-hand side to be more general than those inferred for the right-hand side. However, Definition 3.2 provides a more effective procedure to check well-typedness than the notion of typeable rewrite rule. On the other hand (van Bakel and Fernández, 1997) considers a different setting that includes intersection types, not addressed in our work.

Example 3.1 (Well and ill-typed rules and expressions). Let us consider the following assumptions and program:

$$\mathcal{A} \equiv \{\ \mathbf{z} : nat,\ \mathbf{s} : nat \to nat, \mathbf{true} : bool, \mathbf{false} : bool, (:) : \forall\alpha.\alpha \to [\alpha] \to [\alpha],$$
$$[\,] : \forall\alpha.[\alpha],\ \mathbf{rnat} : repr\ nat,\ \mathbf{id} : \forall\alpha.\alpha \to \alpha,\ \mathbf{snd} : \forall\alpha,\beta.\alpha \to \beta \to \beta,$$
$$\mathbf{unpack} : \forall\alpha,\beta.(\alpha \to \alpha) \to \beta, \mathbf{eq} : \forall\alpha.\alpha \to \alpha \to bool, \mathbf{showNat} : nat \to [char],$$
$$\mathbf{show} : \forall\alpha.repr\ \alpha \to \alpha \to [char],\ \mathbf{f} : \forall\alpha.bool \to \alpha,\ \mathbf{flist} : \forall\alpha.[\alpha] \to \alpha\ \}$$

$$\mathcal{P} \equiv \{\ id\ X \to X,\ snd\ X\ Y \to Y,\ unpack\ (snd\ X) \to X,\ eq\ (s\ X)\ z \to false,$$
$$show\ rnat\ X \to showNat\ X,\ f\ true \to z,\ f\ true \to false,$$
$$flist\ [z] \to s\ z,\ flist\ [true] \to false\ \}$$

It is easy to see that the rules for the functions *id* and *snd* are well-typed. The function *unpack* is taken from (González-Moreno et al., 2001) as a typical example of the type problems that HO-patterns can produce. According to Definition 3.2 the rule of *unpack* is not well-typed since the tuple $(\tau_L, \overline{\alpha_n \pi_L})$ inferred for the left-hand side is (γ, δ), which is not matched by the tuple (η, η) inferred as $(\tau_R, \overline{\beta_n \pi_R})$ for the right-hand side. This shows the problem of existential type variables that "escape" from the scope. If that rule was well-typed then type preservation could not be granted anymore—e.g. consider the step $unpack\ (snd\ true) \rightarrowtail true$, where the type *nat* can be assigned to $unpack\ (snd\ true)$ but *true* can only have type *bool*. The rule for *eq* is well-typed because the tuple inferred for the right-hand side, $(bool, \gamma)$, matches the one inferred for the left-hand side, $(bool, nat)$.

In the rule for *show* the inference obtains $([char], nat)$ for both sides of the rule, so it is well-typed.

The functions f and $flist$ show that our type system cannot be forced to accept an arbitrary function definition by generalizing its type assumption. For instance, the first rule for f is not well-typed since the type nat inferred for the right-hand side does not match γ, the type inferred for the left-hand side. The second rule for f is also ill-typed for a similar reason. If these rules were well-typed, type preservation would not hold: consider the step $f\ true \rightarrowtail z$; $f\ true$ can have any type, in particular *bool*, but z can only have type nat. Both rules of function $flist$ are well-typed, however its type assumption cannot be made more general for its first argument: it can be seen that there is no τ such that the rules for $flist$ remain well-typed under the assumption $flist : \forall\alpha.\alpha \rightarrow \tau$.

With the previous assumptions, expressions like $id\ z\ true$ or $snd\ z\ z\ true$ that lead to *junk* are ill-typed, since the symbols *id* and *snd* are applied to more expressions than the arity of their types. Notice also that although our type system accepts more expressions that may produce pattern matching failures than classical Damas-Milner, it still rejects many such expressions, that typically correspond to programming errors. Examples of this are $flist\ z$ and $eq\ z\ true$, which are ill-typed since the type of the function prevents the existence of program rules that can be used to rewrite these expressions: $flist$ can only have rules treating lists as argument and *eq* can only have rules handling both arguments of the same type.

In (López-Fraguas et al., 2010) we extended Damas-Milner types with some extra control over HO-patterns, leading to another definition of well-typed programs, written $wt^{old}_{\mathcal{A}}(\mathcal{P})$ here. All valid programs in (López-Fraguas et al., 2010) are still valid:

Theorem 3.1. If $wt^{old}_{\mathcal{A}}(\mathcal{P})$ then $wt_{\mathcal{A}}(\mathcal{P})$.

Proof. See page 27 in Appendix A. □

To further appreciate the difference between the two approaches, notice that all the examples in Section 5 are rejected as ill-typed by (López-Fraguas et al., 2010). The purpose of the two systems is different: in this paper we attempt deliberately to go beyond Damas-Milner, while (López-Fraguas et al., 2010) only aims to deal safely with programs using HO-patterns in rules, but keeping the behavior of Damas-Milner otherwise. In correspondence to that, in (López-Fraguas et al., 2010) the types of program functions can be inferred, while in the present work they must be explicitly declared.

4. Properties of the type system

We will follow two alternative approaches for proving type soundness of our system. First, we prove the theorems of *progress* and *type preservation* similar to those that play the main role in the type soundness proof for GADTs (Cheney and Hinze, 2003; Peyton Jones et al., 2006). After that, we follow a syntactic approach similar to (Wright and Felleisen, 1992). The first result, *progress*, states that well-typed ground expressions are either patterns or expressions reducible by *let*-rewriting.

Theorem 4.1 (Progress). If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then either e is a pattern or $\exists e'.\ \mathcal{P} \vdash e \rightarrowtail e'$.

Proof. By induction over the structure of e, see page 29 in Appendix A for the complete proof. □

In order to relate well-typed expressions and evaluation we need a *type preservation*—or *subject reduction*—result, stating that in well-typed programs reduction does not change types.

Theorem 4.2 (Type Preservation). If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $\mathcal{P} \vdash e \rightarrowtail e'$, then $\mathcal{A} \vdash e' : \tau$.

Proof. By case distinction over the rule of the let-rewriting relation $\rightarrowtail$ used to reduce e to e'. The detailed proof can be found in page 31 in Appendix A. □

This result shows that the degree of liberality given to our type system is not arbitrary: types are certainly more liberal than in the usual Damas-Milner system, but they are also restricted enough as to ensure that types are not lost during reduction. In Example 3.1 we saw examples of ill-typed programs for which type preservation fails. At this point, an interesting question arises: could the type system be even more relaxed but still keep type preservation? The following results shows that in a certain sense the answer is *'no'*, and therefore our well-typedness conditions are as liberal as possible without compromising type preservation.

Theorem 4.3 (Maximal liberality of well-typedness conditions).
Let $\mathcal{A}$ be a closed set of assumptions, and assume that $\mathcal{P}$ is a program which is not well-typed wrt. $\mathcal{A}$, but such that every rule $R \in \mathcal{P}$ verifies the condition i) of well-typedness in Definition 3.2. Then there exists a rule $(f\ t_1 \ldots t_m \rightarrow e) \in \mathcal{P}$ with variables $\overline{X_n}$ and there exist types $\overline{\tau_n}$, τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ and $f\ t_1 \ldots t_m \rightarrowtail e$ but $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \nvdash e : \tau$.

Proof. By case distinction on the condition of $wt_{\mathcal{A}}(\mathcal{P})$ that fails. The complete proof can be found in page 32 in Appendix A. □

By requiring the condition that all rules in the program verify condition i) of program well-typedness, we ensure that ill-typedness of the program is not due to a badly typed left-hand side of a rule—an uninteresting case from the point of view of type preservation under reduction—but must be due to a failure of conditions ii) or iii)—as condition iv) does not fail for closed assumptions—that is, due to a lack of right correspondence between some left-hand side and its companion right-hand side. We remark that the proof of Theorem 4.3 is constructive in the sense that, for a program in the hypothesis of the theorem, it provides explicitly a reduction step and types which witness the failure of type preservation.

Theorem 4.3 also indicates that, in a sense, our notion of well-typed rule captures essentially the intuitive idea that a rule preserves types when applied to reduce an expression.

That intuition becomes indeed a provable technical result by giving a declarative definition of type-preserving rule and proving that, under certain reasonable conditions, this notion is equivalent to well-typedness.

Definition 4.1 (Type-preserving rule). Given a set of assumptions $\mathcal{A}$, we say that a rule $f\ t_1 \ldots t_m \to e$ *preserves types* if

(i) its left-hand side admits some type, i.e., $wt_{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}}(f\ t_1 \ldots t_m)$ for some $\overline{\tau_n}$, where $\overline{X_n}$ are the variables appearing in the rule—$\{\overline{X_n}\} = fv(f\ t_1 \ldots t_m)$.
(ii) $\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau \Longrightarrow \mathcal{A} \vdash e\theta : \tau$, for any substitution θ and type τ.

We impose the first condition to avoid the case of rules which do not break type preservation trivially because their left-hand sides are not well-typed, so that $\mathcal{A} \not\vdash f\ t_1\theta \ldots t_m\theta : \tau$ for any τ.

The notions of well-typed rules and type-preserving rules are equivalent, but only for a certain kind of assumptions which are rich enough to build monomorphic terms of any given type, as formalized in the following definition.

Definition 4.2 (Type-complete set of assumptions). A set of assumptions $\mathcal{A}$ is called *type-complete* if for each simple type τ there exists a pattern t_τ which can only have that type, i.e., $\mathcal{A} \vdash t_\tau : \tau$ and $\mathcal{A} \not\vdash t_\tau : \tau'$ for all $\tau' \neq \tau$.

Now, we can prove the announced equivalence result, showing that the definition of well-typed rule capture algorithmically the precise declarative notion of type preservation in function applications.

Proposition 4.1. Consider a type-complete set of assumptions $\mathcal{A}$, and a program rule R. Then R preserves types iff $wt_{\mathcal{A}}(R)$.

The condition of type-completeness is imposed to avoid cases when type preservation in a function application is potentially compromised but not actually broken *with the data constructors and functions currently in the program*. However, if the program is extended with new symbols, it would be possible to call the function breaking type preservation. The following example shows this situation:

Example 4.1. Consider the program $\mathcal{P} \equiv \{id\ X \to X, f\ F \to F\ true\}$ with types $\mathcal{A} \equiv \{id : \forall\alpha.\alpha \to \alpha, f : \forall\alpha.(\alpha \to \alpha) \to bool\}$. It is easy to check that, with the current data constructor and functions symbols, the only pattern that can be passed as argument of f making the application well-typed is id, which preserves types. However types are not preserved for any pattern whose only type was $\tau \to \tau$ (for any τ). If we add to the program the function $\{inc\ N \to N + 1\}$ with type $int \to int$ then the rule for f break type preservation: $\mathcal{A} \vdash f\ inc : bool$ but $\mathcal{A} \not\vdash inc\ true : bool$.

Notice that according to the definition of well-typed rule (Definitions 3.1 or 3.2) the rule for f is ill-typed in both situations, as the right-hand side restricts the type of F more than its left-hand side—although in the first case there is not enough symbols to cause the loss of type preservation.

We now turn to a syntactic approach to type safety similar to (Wright and Felleisen, 1992). Before that we need to define some properties about expressions:

Definition 4.3. An expression e is *stuck* wrt. a program $\mathcal{P}$ if it is a normal form but not a pattern, and is *faulty* if it contains a *junk* subexpression.

Faulty is a pure syntactic property that tries to overapproximate *stuck*. Not all faulty expressions are stuck. For example, $snd\ (z\ z)\ true$ is *faulty* but $snd\ (z\ z)\ true \rightarrowtail true$. However all faulty expressions are ill-typed:

Lemma 4.1 (Faulty expressions are ill-typed). If e is faulty then there is no $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$.

Proof. By contradiction, using the fact that *junk* expressions cannot have a valid type wrt. any set of assumptions $\mathcal{A}$. See page 34 in Appendix A for a complete proof. □

The next theorem states that all finished reductions of well-typed ground expressions do not get stuck but end up in patterns of the same type as the original expression.

Theorem 4.4 (Syntactic Soundness). If $wt_{\mathcal{A}}(\mathcal{P})$, e is ground and $\mathcal{A} \vdash e : \tau$ then: for all $e' \in nf_{\mathcal{P}}(e)$, e' is a pattern and $\mathcal{A} \vdash e' : \tau$.

Proof. See page 35 in Appendix A for a complete proof. □

The following complementary result states that the evaluation of well-typed expressions does not pass through any faulty expression.

Theorem 4.5. If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then there is no e' such that $\mathcal{P} \vdash e \rightarrowtail^* e'$ and e' is faulty.

Proof. By contradiction. Suppose that $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$, e is ground and there exists some e' such that $\mathcal{P} \vdash e \rightarrowtail^* e'$ and e' is faulty. By Type Preservation (Theorem 4.2) we know that $\mathcal{A} \vdash e' : \tau$, but by Lemma 4.1 faulty expressions are ill-typed, reaching a contradiction. □

4.1. *Discussion of the properties*

We discuss now the strength of our results considering some interdependent factors: the rules for failure in Section 3, the liberality of our well-typedness condition, and our notion of faulty expression.

Progress and type preservation. In (Milner, 1978) Milner considered *'a value 'wrong', which corresponds to the detection of a failure at run-time'* to reach his famous lemma *'well-typed programs don't go wrong'*. For this to be true in languages with pattern matching, like Haskell or ours, not all run-time failures should be seen as wrong, as happens with definitions like $head\ (x{:}xs) \rightarrow x$, where there is no rule for *(head [ ])*. Otherwise, progress does not hold and some well-typed expressions become stuck. A solution is considering a 'well-typed completion' of the program, adding a rule like $head\ [\ ] \rightarrow error$ where *error* is a value accepting any type. With it, *(head [ ])* reduces to *error* and is not wrong, but *(head true)*, which is ill-typed, is wrong and its reduction gets stuck. In

our setting, completing definitions would be more complex because of HO-patterns that could lead to an infinite number of 'missing' cases. To cope with this problem, our failure rules in Section 3 are used to replace the 'well-typed completion'. We prefer the word *fail* instead of *error* because, in contrast to FP systems where an attempt to evaluate *(head [ ])* results in a run-time error, in FLP systems rather than an error this is a silent failure in a possible space of non-deterministic computations managed by backtracking. Admittedly, in our system the difference between 'wrong' and 'fail' is weaker from the point of view of reduction. Certainly, junk expressions are stuck but, for instance, *(head [ ])* and *(head true)* both reduce to *fail*, instead of the ill-typed *(head true)* getting stuck. Since *fail* accepts all types, this might seem a point where ill-typedness comes in hiddenly and then magically disappear by the effect of reduction to *fail*. This cannot happen, however, because *type preservation* holds step-by-step, and then no reduction $e \rightarrow^* fail$ starting with a well-typed e can pass through the ill-typed *(head true)* as intermediate (sub)-expression.

Liberality. In our system the risk of accepting as well-typed some expressions that one might prefer to reject at compile time is higher than in more restrictive type systems. Consider the function *size* of Section 1, page 3. For any well-typed expression e, *size* e is also well-typed, even if e's type is not considered in the definition of *size*; for instance, *size (true,false)* is a well-typed expression reducing to *fail*. This is consistent with the liberality of our system, since the definition of *size* could perfectly have included a rule for computing sizes of pairs. Hence, for our system, this is a pattern matching failure similar to the case of *(head [ ])*. This can be appreciated as a weakness, and is further discussed in Section 6 in connection to type classes and GADTs.

Syntactic soundness and faulty expressions. Theorems 4.4 and 4.5 are easy consequences of progress and type preservation. Theorem 4.5 is indeed a weaker safety criterion, because our faulty expressions only capture the presence of junk, which by no means is the only source of ill-typedness. For instance, the expressions *(head true)* or *(eq true z)* are ill-typed but not faulty. Theorem 4.5 says nothing about them; it is type preservation who ensures that those expressions will not occur in any reduction starting in a well-typed expression. Still, Theorem 4.5 contains no trivial information. Although checking the presence of junk is trivial (counting arguments suffices for it), the fact that a given expression will not become faulty during reduction is a typically undecidable property approximated by our type system. For example, consider g with type $\forall\alpha,\beta.(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$, defined as $g\ H\ X \rightarrow H\ X$. The expression *(g true false)* is not faulty but reduces to the faulty *(true false)*. Our type system avoids that because the non-faulty expression *(g true false)* is detected as ill-typed.

5. Examples

In this section we present some examples showing the flexibility achieved by our type system. They are written in two parts: a set of assumptions $\mathcal{A}$ over constructors and functions and a set of program rules $\mathcal{P}$. We consider the following initial set of assumptions,

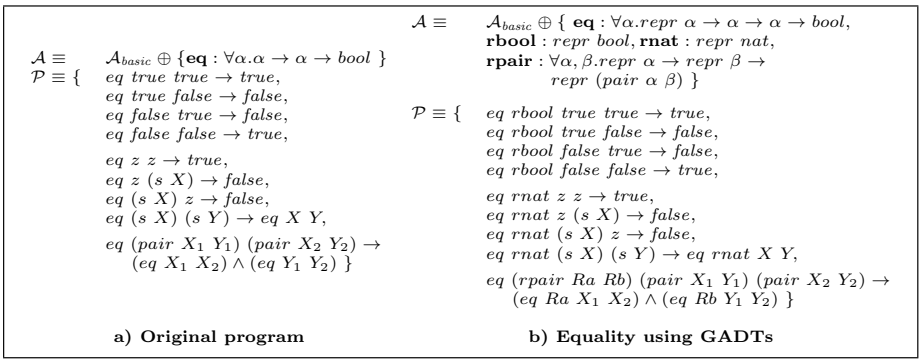


Fig. 4. Type-indexed equality

common to all examples:

$$\mathcal{A}_{basic} \equiv \{\mathbf{true}, \mathbf{false} : bool,\ \mathbf{z} : nat,\ \mathbf{s} : nat \to nat,\ (:) : \forall\alpha.\alpha \to [\alpha] \to [\alpha],$$
$$[\,] : \forall\alpha.[\alpha],\ \mathbf{pair} : \forall\alpha,\beta.\alpha \to \beta \to pair\ \alpha\ \beta,\ \mathbf{key} : \forall\alpha.\alpha \to (\alpha \to nat) \to key,$$
$$\wedge, \vee : bool \to bool \to bool,\ \mathbf{snd} : \forall\alpha,\beta.\alpha \to \beta \to \beta,\ \mathbf{length} : \forall\alpha.[\alpha] \to int\}$$

5.1. *Type-indexed functions*

Type-indexed functions—in the sense appeared in (Hinze and Löh, 2007)—are functions that have a particular definition for each type in a certain family. The function *size* of Section 1—page 3—is an example of such a function. A similar example is given in Figure 4-a, containing the code for an equality function which operates only with booleans, natural numbers and pairs.

An interesting point is that we do not need a type representation as an extra argument of this function as we would need in a system using GADTs (Cheney and Hinze, 2003; Hinze and Löh, 2007). In these systems the pattern matching on the GADT induces a type refinement, allowing the rule to have a more specific type than the type of the function. In our case this flexibility resides in the notion of well-typed rule. Then a type representation is not necessary because the arguments of each rule of *eq* already force the type of the left-hand side and its variables to be more specific (or the same) than those inferred for the right-hand side. The absence of type representations provides simplicity to rules and programs, since extra arguments imply that all functions using *eq* direct or indirectly must be extended to accept and pass these type representations. In contrast, our rules for *eq* (extended to cover all constructed types) are the standard rules defining strict equality that one can find in FLP papers—see e.g. (Hanus, 2007)—but that cannot be written directly in existing systems like Toy or Curry, because they are ill-typed according to Damas-Milner types.

We stress also the fact that the program of Figure 4-a would be rejected by systems supporting GADTs (Cheney and Hinze, 2003; Schrijvers et al., 2009), while the encoding of equality using GADTs as type representations in Figure 4-b is also accepted by our type system.

Another interesting point is that we can handle equality in a quite fine way, much more flexible than in Toy or Curry, where equality is a *built-in* that proceeds structurally as in Figure 4-a. With our proposed type system programmers can define structural equality as in Figure 4-a for some types, choose another behavior for others, and omitting the rules for the cases they do not want to handle. Moreover, the type system protects against unsafe definitions, as we explain now: it is known (González-Moreno et al., 2001) that in the presence of HO-patterns[§] structural equality can lead to the problem of *opaque decomposition*. For example, consider the expression $eq\ (snd\ z)\ (snd\ true)$. It is well-typed, but after a decomposition step using the structural equality we obtain $eq\ z\ true$, which is ill-typed. Different solutions have been proposed (González-Moreno et al., 2001), but all of them need fully type-annotated expressions at run time, which penalizes efficiency. With our proposed type system that overloading at run time is not necessary since this problem of opaque decomposition is handled statically at compile time: we simply cannot write equality rules leading to opaque decomposition, because they are rejected by the type system. This happens with the rule $eq\ (snd\ X)\ (snd\ Y) \to eq\ X\ Y$, which will produce the previous problematic step. It is rejected because the inferred type for the right-hand side and its variables X and Y is $(bool, \gamma, \gamma)$, which is more specific than the inferred in the left-hand side $(bool, \alpha, \beta)$.

Finally, type-indexed functions in our type system have a very interesting application. It is well known that type classes (Wadler and Blott, 1989; Hall et al., 1996) provide a clean, modular and elegant way of writing overloaded functions in functional languages as Haskell. Type classes are usually implemented by means of a source-to-source transformation that introduces extra parameters—called dictionaries—to overloaded functions (Wadler and Blott, 1989; Hall et al., 1996). However, this classical translation produces a problem of missing answers when applied to FLP due to a bad interaction between non-determinism and the call-time choice semantics (Lux, 2009; Martin-Martin, 2011). Using type-indexed functions and *type witnesses*—a representation of types as values—it is possible to develop a type-passing translation for type classes similar to (Thatté, 1994) that solves this problem and whose translated programs are well-typed in the proposed liberal type system. Figure 5 shows the translation of a program with type classes using the equality class and function. As can be seen, the `eq` function is translated into a type-indexed function whose first argument is a type witness. These type witnesses—which are new constructors generated for the data types in program, with types `#bool`:: *bool* and `#list`:: $A \to [A]$—are used to determine which rules of the type-indexed function `eq` can be used. Proper type witnesses are passed to overloaded functions, as in the case of the `member` function. These witnesses are determined by a type analysis over the expressions in source programs, just as it is done in the classical dictionary-based translation of type classes.

Apart from solving the problem of missing answers, this type-passing translation also produces faster and simpler programs than the classical translation. A complete discus-

[§] This situation also appears with first order patterns containing data constructors with existential types.

```
eqBool :: bool → bool → bool
eqBool true true = true
eqBool true false = false
eqBool false true = false
eqBool false false = true

class eq A where
  eq :: A → A → bool

instance eq bool where
  eq X Y = eqBool X Y

instance ⟨eq A⟩ ⇒ eq [A] where
  eq [] [] = true
  eq [] (Y:Ys) = false
  eq (X:Xs) [] = false
  eq (X:Xs) (Y:Ys) =
    and (eq X Y) (eq Xs Ys)

member :: ⟨eq A⟩ ⇒ [A] → A → bool
member [] Y = false
member (X:Xs) Y =
  or (eq X Y) (member Xs Y)
```

a) Source program

```
eqBool :: bool → bool → bool
eqBool true true = true
eqBool true false = false
eqBool false true = false
eqBool false false = true

eq :: A → A → A → bool
eq #bool X Y = eqBool X Y
eq (#list W_A) [] [] = true
eq (#list W_A) [] (Y:Ys) = false
eq (#list W_A) (X:Xs) [] = false
eq (#list W_A) (X:Xs) (Y:Ys) = and
  (eq W_A X Y)
  (eq (#list W_A) Xs Ys)

member :: A → [A] → A → bool
member W_A [] Y = false
member W_A (X:Xs) Y =
  or (eq W_A X Y) (member W_A Xs Y)
```

b) Translated program

Fig. 5. Translation of a program using equality

sion of these points, the formalization of the translation and further examples can be found in (Martin-Martin, 2011).

5.2. *Existential types, opacity and HO-patterns*

Existential types (Mitchell and Plotkin, 1988; Perry, 1991; Läufer and Odersky, 1994) appear when type variables in the type of a constructor do not occur in the final type. For example the constructor $key : \forall\alpha.\alpha \to (\alpha \to nat) \to key$ has an existential type, since α does not appear in the final type key, i.e., it has the equivalent type $(\exists\alpha.\alpha \to (\alpha \to nat)) \to key$. This type means that the first argument of key is an expression of some *unknown* type α, and the second one is a function from that unknown type to natural numbers $(\alpha \to nat)$. Systems supporting existential types treat differently constructors with existential type (in the sequel *existential constructors*) depending on their place in the rule. If they appear in the right-hand side, they are treated as any other polymorphic symbol, allowing any instance of their type. However, if they appear in the left-hand side, new distinct constant types—called *Skolem constants*—are introduced for each existentially quantified variable. For example in $key\ X\ F$ the constructor key is assigned the type $\kappa \to (\kappa \to nat) \to key$—where κ is a fresh Skolem constant—so X and F have types κ and $\kappa \to nat$ respectively. Therefore, any occurrence of these data variables in the right-hand side that needs a more concrete type as $(not\ X)$ or $(F\ true)$

will be considered ill-typed. This situation also happens in the left-hand side of the rule, if *key* contains arguments of more concrete types as in (*key z s*).

The type system presented in this paper accepts classical functions dealing with existential constructors, like *getKey*:

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{getKey} : key \to nat\ \} \qquad \mathcal{P} \equiv \{\ getKey\ (key\ X\ F) \to F\ X\ \}$$

Notice that this rule is well-typed because the right-hand side does not force the types of the variables X and F (α and $\alpha \to \beta$ resp.) more than the left-hand side does (α and $\alpha \to nat$ resp.). However, the type system presented here gives a more permissive treatment to existential constructors than usual approaches (Mitchell and Plotkin, 1988; Perry, 1991; Läufer and Odersky, 1994). As a consequence, rules containing existential constructors with arguments of concrete types—as $getKey\ (key\ z\ s) \to z$ or $getKey\ (key\ (s\ X)\ F) \to s\ (F\ X)$—are allowed provided right-hand sides does not restrict the types of the variables more than left-hand sides. Notice that our more permissive behavior comes directly from the definition of well-typed rule and no specific treatment of existential constructors is needed¶, in the same way that the *size* function from Section 1—page 3—has rules whose argument have a more specific type (*bool*, *nat* and $[\alpha]$) than the type for them that comes from the declared type of the function (α).

Apart from existential constructors, in functional logic languages HO-patterns can introduce a similar opacity than existential types. A prototypical example is $snd\ X$: we know that X has some type, but we cannot know anything about it from the type $\beta \to \beta$ of the expression. This opacity problem, originally identified in (González-Moreno et al., 2001), is solved in (López-Fraguas et al., 2010) by means of *opaque variables*. Briefly explained, a data variable is opaque in a pattern if the type of the whole pattern does not univocally fix the type of the variable. That is the case of X in the pattern $snd\ X$: from the type $\beta \to \beta$ of the pattern we cannot know univocally the type of X, which indeed can have any type (*bool*, *int*, [*bool*] ...). The problems that opaque variables generate for type preservation are solved in (López-Fraguas et al., 2010) by forbidding *critical variables* in program rules (data variables appearing in the righ-hand side which are opaque in a pattern of the left-hand side). However, it is known that this solution rejects functions that do not compromise type preservation although they contain critical variables. The program below shows how the system presented here generalizes that from (López-Fraguas et al., 2010), accepting functions containing critical variables:

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{idSnd} : \forall\alpha,\beta.(\alpha \to \alpha) \to (\beta \to \beta),\ \mathbf{f} : \forall\alpha.(\alpha \to \alpha) \to int\ \}$$
$$\mathcal{P} \equiv \{idSnd\ (snd\ X) \to snd\ X,\ f\ (snd\ X) \to length\ [X],\ f\ (snd\ (X : Xs)) \to length\ Xs\}$$

Variables X and Xs are critical in all the rules, so they are rejected by the type system in (López-Fraguas et al., 2010). However, the type system presented here accepts all the rules because they verify the well-typedness criterion: right-hand sides do not restrict the types of the variables more than left-hand sides.

Another remarkable example using HO patterns is given by the well-known translation of higher-order programs to first-order programs (Warren, 1982) often used as a stage of

¶ In contrast to the explicit treatment of existentially quantified variables using Skolem constants.

the compilation of functional logic programs—see e.g. (Antoy and Tolmach, 1999; López-Fraguas et al., 2008). In short, this translation introduces a new function symbol @ (to be read as '*apply*'), and then adds calls to @ in some points in the program and appropriate rules for evaluating it. This latter aspect is interesting here, since those @-rules are not Damas-Milner typeable. The following program contains the @-rules (written in infix notation) for a concrete example with the constructors z, s, $[\,]$, $(:)$ and the functions *length*, *append* and *snd* with the usual types.

$$\begin{array}{rl} \mathcal{A} \equiv & \mathcal{A}_{basic} \oplus \{\ \mathbf{length} : \forall\alpha.[\alpha] \to nat, \mathbf{append} : \forall\alpha.[\alpha] \to [\alpha] \to [\alpha], \\ & \mathbf{add} : nat \to nat \to nat, @ : \forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta\ \} \\ \mathcal{P} \equiv \{ & s\ @\ X \to s\ X, (:)\ @\ X \to (:)\ X, ((:)\ X)\ @\ Y \to (X : Y), \\ & append\ @\ X \to append\ X, (append\ X)\ @\ Y \to append\ X\ Y, \\ & snd\ @\ X \to snd\ X, (snd\ X)\ @\ Y \to snd\ X\ Y, length\ @\ X \to length\ X\ \} \end{array}$$

These rules use HO-patterns, which is a cause of rejection in many systems. Even if HO-patterns were allowed, the rules for @ would be rejected by a Damas-Milner-like type system. Because of all this, the @-introduction stage of the FLP compilation process can be considered as a source to source transformation, instead of a hard-wired step.

5.3. *Generic functions*

According to a strict view of genericity, the functions *size* and *eq* in Section 1 and 5.1 resp. are not truly generic. We have a definition for each type, instead of one 'canonical' definition to be used by each concrete type. However we can achieve this by introducing a 'universal' data type over which we define the function and then use it for concrete types via a conversion function. We develop the idea for the *size* example.

This can be done by using GADTs to represent uniformly the applicative structure of expressions—for instance, the *spines* of (Hinze and Löh, 2007)—then defining *size* over that uniform representations, and finally applying it to concrete types via conversion functions. Again, we can also offer a similar but simpler alternative. A uniform representation of constructed data can be achieved with a data type *data univ = c nat [univ]* where the first argument of c is used for numbering constructors, and the second one is the list of arguments of a constructor application. A universal *size* can be defined as *usize (c _ Xs) → s (sum (map usize Xs))* using some functions of Haskell's prelude. Now, a generic *size* can be defined as $size \to usize \cdot toU$, where toU is a conversion function with declared type $toU : \forall\alpha.\alpha \to univ$

$$\begin{array}{ll} toU\ true \to c\ z\ [\,] & toU\ false \to c\ (s\ z)\ [\,] \\ toU\ z \to c\ (s^2\ z)\ [\,] & toU\ (s\ X) \to c\ (s^3\ z)\ [toU\ X] \\ toU\ [\,] \to c\ (s^4\ z)\ [\,] & toU\ (X{:}Xs) \to c\ (s^5\ z)\ [toU\ X, toU\ Xs] \end{array}$$

(s^i abbreviates iterated s's). This toU function uses the specific features of our system. It is interesting also to remark that in our system the truly generic rule $size \to usize \cdot toU$ can coexist with the type-indexed rules for *size* of Section 1. This might be useful in

practice: one can give specific, more efficient definitions for some concrete types, and a generic default case via *toU* conversion for other types[‖].

Admittedly, the type *univ* has less representation power than the spines of (Hinze and Löh, 2007), which could be a better option in more complex situations. Nevertheless, notice that the GADT-based encoding of spines is also valid in our system.

6. Discussion

We further discuss here some positive and negative aspects of our type system.

Simplicity. Our well-typedness condition, which adds only one simple check for each program rule to standard Damas-Milner inference, is much easier to integrate in existing FLP systems than, for instance, type classes—see (Lux, 2008) for some known problems for the latter—or GADTs, which have a specific type system more complex than Damas-Milner.

Liberality (continued from Section 4). We recall the example of *size*, where our system accepts the expression *size e* as well-typed, for any well-typed *e*. Type classes impose more control: *size e* is only accepted if *e* has a type in the class *Sizeable*. There is a burden here: you need a class for each generic function, or at least for each range of types for which a generic function exists; therefore, the number of class instance declarations for a given type can be very high. GADTs are in the middle way. At a first sight, it seems that the types to which *size* can be applied are perfectly controlled because only *representable* types are permitted. The problem, as with classes, comes when considering other functions that are generic but for other ranges of types. Now, there are two options: either you enlarge the family of representable types, facing up again the possibility of applying *size* to unwanted arguments, or you introduce a new family of representation types, which is a programming overhead, somehow against genericity.

Need of type declarations. In contrast to Damas-Milner system, where principal types exist and can be inferred, our definition of well-typed program (Definition 3.1) assumes an explicit type declaration for each function. This happens also with other well-known type features, like polymorphic recursion, arbitrary-rank polymorphism or GADTs (Cheney and Hinze, 2003; Schrijvers et al., 2009). Moreover, programmers usually declare the types of functions as a way of documenting programs. Notice also that type inference for functions would be a difficult task since functions, unlike expressions, do not have *principal types*. Consider for instance the rule *not true* $\rightarrow$ *false*. All the possible types for the *not* function are $\forall\alpha.\alpha \rightarrow \alpha$, $\forall\alpha.\alpha \rightarrow bool$ and $bool \rightarrow bool$ but none of them is most general.

Loss of parametricity. In (Wadler, 1989) one of the most remarkable applications of type systems was developed. The main idea there is to derive "free theorems" about the equivalence of functional expressions by just using the types of some of its constituent functions. These equivalences express different distribution properties, based on

[‖] For this to be really practical in FLP systems, where there is not a 'first-fit' policy for pattern matching in case of overlapping rules, a specific syntactic construction for 'default rule' would be needed.

Reynold's abstraction theorem there recast as "the parametricity theorem", which basically exploits the fact that the polymorphic type variables in the types of function symbols cannot be instantiated in the left-hand side of program rules. Parametricity was originally developed for the polymorphic λ-calculus, which in particular enjoys the strong normalisation property, so its application to actual languages with practical features like unbounded recursion or partial functions has to be done with care. This can be easily understood by considering the first example in (Wadler, 1989), stating that for any function $f : \forall\alpha.[\alpha] \to [\alpha]$ and any function g with some (irrelevant) type then $(map\ g) \circ f \equiv f \circ (map\ g)$. The intuition is that, as by parametricity f cannot inspect the polymorphic elements of its input list—to do so it should instantiate the type variable α into a more concrete type in the left-hand side of some program rule for f—then it may only return a rearrangement of that list, maybe dropping or duplicating some of its elements but never introducing new elements. This is not the case for a practical language like Haskell, for example, as we can define the functions $\{loop \to loop, fail \to head\ [\,]\}$, both with type $\forall\alpha.\alpha$, that can be used to introduce new elements in the resulting list for f thus breaking that free theorem (Seidel and Voigtländer, 2010). Similarly an impure feature like Haskell's *seq* operator weakens parametricity because it essentially inspects its polymorphic first argument in order to force its evaluation (Hudak et al., 2007). Nevertheless free theorems can be weakened with several additional conditions so they actually hold for Haskell (Wadler, 1989; Johann and Voigtländer, 2004). These efforts are motivated by the fact that parametricity is used to justify the soundness of some important compiler optimizations, like the "short-cut deforestation" of GHC (GHC-Team, 2011)—although it is admitted that *seq* still makes this particular transformation unsound (Hudak et al., 2007).

Regarding FLP, it is known that non-determinism not only breaks free theorems but also equational rules for concrete functions that hold for Haskell, like $(filter\ p) \circ (map\ h) \equiv (map\ h) \circ (filter\ (p \circ h))$ (Christiansen et al., 2010). The situation gets even worse when considering extra variables and narrowing—not treated in the present work but standard in FLP systems—because then the function f above could also introduce a free variable in its resulting list, thus breaking the equivalence from a new side wrt. Haskell, as in FLP free variables may produce interesting values in contrast to $loop$ and $fail$.

With our type system, not only those free theorems derived from parametricity are broken, but it is the more fundamental notion of parametricity they rely on that is lost, because functions are allowed to inspect any argument subexpression, as seen in the *size* function from page 3. This has a limited impact in the FLP setting, as free theorems were already heavily compromised by non-determinism and free variables, but it could limit the applicability of our type system to pure FP. For example, working without the hypothesis of parametricity would be a problem for GHC because of its representation of datatypes, which results in an unpredictable behaviour when matching two expressions with different types—as can be seen by using the polymorphic casting function from (Hudak et al., 2007). Fortunately, state-of-the-art FLP systems are based on a compilation to Prolog for which those heterogeneous matchings pose no problem. In fact ours would not be the first type system for FP that allows that kind of liberalized inspections, i.e. it is possible to do that by using GADTs, as seen in Figure 4-b. Nevertheless GADTs—at

least those implemented by GHC—are only able to inspect "liberalized" arguments whose type has been already sufficiently refined in the left-to-right Haskell matching process. For example if we interchange the first and third argument of *eq* in Figure 4-b then the program would be rejected by GHC—while it is still accepted by our type system. The reason is that GHC's matching process proceeds from left to right and, as GADT arguments fix their polymorphic types when matched thus fixing the types of the arguments they liberalize, that ensures the absence of dangerous matchings in GHC. Similarly, classical existential types use skolem constants to forbid liberalized inspections that would threaten parametricity and turn GHC style matching and datatypes representation into an unsound procedure. However, that liberalized inspections just result from the kind of matchings exploited by our liberal functions, therefore the possible application of our type system to concrete Haskell implementations remains an open problem. Maybe a modification of our proposed type system, that would restrict liberal typing of functions to some fragments of the program only, would still enjoy some relevant parametricity property. We consider this an interesting subject of future work.

7. Conclusions

Starting from a simple type system, essentially Damas-Milners's one, we have proposed a new notion of well-typed functional logic program that exhibits interesting properties: simplicity; enough expressivity to achieve a variety of existential types or GADT-like encodings, and to open new possibilities to genericity; good formal properties (type soundness, protection against unsafe use of HO-patterns, maximal liberality while fulfilling the previous conditions). Regarding the practical interest of our work, we stress the fact that no existing FLP system supports any of the examples in Section 5, in particular the examples of the *equality*—where known problems of *opaque decomposition* (González-Moreno et al., 2001) can be addressed—and *apply* functions, which play important roles in the FLP setting. Moreover, our work provides a valuable alternative to our previous results (González-Moreno et al., 2001; López-Fraguas et al., 2010) about safe uses of HO-patterns. However, considering also the weaknesses discussed in Section 6 suggests that a good option in practice could be a partial adoption of our system, not attempting to replace standard type inference, type classes or GADTs, but rather complementing them.

We find suggestive to think of the following future scenario for our system Toy: a typical program will use standard type inference except for some concrete definitions where it is annotated that our new liberal system is adopted instead. In addition, adding type classes to the languages is highly desirable; then programmers can choose the feature—ordinary types, classes, GADTs or our more direct generic functions—that best fits their needs of genericity and/or control in each specific situation.

Some steps to achieve this scenario have been already performed. The first one is a web interface (`http://gpd.sip.ucm.es/LiberalTyping`) of the type system which checks program well-typedness. This web interface supports GADT syntax for data declarations, so all the examples in this paper can be checked. Another performed step is the development of a branch of Toy using the type system proposed in this paper, which

can be downloaded at `http://gpd.sip.ucm.es/Toy2Liberal`. This branch lacks syntax for GADT data declaration, however it provides the users a complete and functional Toy system where programs can be compiled and evaluated.

Apart from further implementation work, we consider several lines of future work:

— A precise specification of how to mix different typing conditions in the same program and how to translate type classes into our generic functions. A first step towards the specification of the translation of type classes has been already developed in (Martin-Martin, 2011).
— Despite of the lack of principal types, some work on type inference can be done, in the spirit of (Schrijvers et al., 2009).
— Combining our genericity with the existence of modules could require adopting *open* types and functions (Löh and Hinze, 2006).
— Narrowing, which poses specific problems to types, should be also considered.

Acknowledgments We are grateful to the anonymous reviewers for suggesting us the idea of an alternative declarative formulation of well-typedness, that lead us to include Definition 4.1 and Proposition 4.1. We also thank Philip Wadler and the rest of reviewers of the previous conference version of the paper, for their stimulating criticisms and comments.

References

Antoy, S. and Tolmach, A. P. (1999). Typed higher-order narrowing without higher-order strategies. In *4th International Symposium on Functional and Logic Programming (FLOPS '09)*, pages 335–353. Springer *LNCS* 1722.

Cheney, J. and Hinze, R. (2003). First-class phantom types. Technical Report TR2003-1901, Cornell University.

Christiansen, J., Seidel, D., and Voigtländer, J. (2010). Free theorems for functional logic programs. In *4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV '10)*, pages 39–48. ACM.

Damas, L. and Milner, R. (1982). Principal type-schemes for functional programs. In *9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212. ACM.

GHC-Team (2011). The Glorious Glasgow Haskell Compilation System User's Guide. `http://www.haskell.org/ghc/docs/latest/html/users_guide`.

González-Moreno, J., Hortalá-González, T., López-Fraguas, F., and Rodríguez-Artalejo, M. (1999). An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87.

González-Moreno, J., Hortalá-González, T., and Rodríguez-Artalejo, M. (1997). A higher order rewriting logic for functional logic programming. In *14th International Conference on Logic Programming (ICLP '97)*, pages 153–167. MIT Press.

González-Moreno, J., Hortalá-González, T., and Rodríguez-Artalejo, M. (2001). Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1).

Hall, C. V., Hammond, K., Peyton Jones, S., and Wadler, P. (1996). Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138.

Hanus, M. (2007). Multi-paradigm declarative languages. In *23rd International Conference on Logic Programming (ICLP '07)*, pages 45–75. Springer *LNCS* 4670.

Hanus (ed.), M. (2006). Curry: An integrated functional logic language. `http://www.informatik.uni-kiel.de/~curry/report.html`.

Hinze, R. (2006). Generics for the masses. *Journal of Functional Programming*, 16(4-5):451–483.

Hinze, R. and Löh, A. (2007). Generic programming, now! In *Datatype-Generic Programming 2006*, pages 150–208. Springer *LNCS* 4719.

Hudak, P., Hughes, J., Peyton Jones, S., and Wadler, P. (2007). A history of Haskell: being lazy with class. In *3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL '07)*, pages 12–1–12–55. ACM.

Johann, P. and Voigtländer, J. (2004). Free theorems in the presence of *seq*. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 99–110. ACM.

Läufer, K. and Odersky, M. (1994). Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16:1411–1430.

Löh, A. and Hinze, R. (2006). Open data types and open functions. In *8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '06)*, pages 133–144. ACM.

López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. (2010). Liberal Typing for Functional Logic Programs. In *8th Asian Symposium on Programming Languages and Systems (APLAS '10)*, pages 80–96. Springer *LNCS* 6461.

López-Fraguas, F., Martin-Martin, E., and Rodríguez-Hortalá, J. (2010). New results on type systems for functional logic programming. In *18th International Workshop on Functional and Constraint Logic Programming (WFLP '09)*, pages 128–144. Springer *LNCS* 5979.

López-Fraguas, F., Rodríguez-Hortalá, J., and Sánchez-Hernández, J. (2008). Rewriting and call-time choice: the HO case. In *9th International Symposium on Functional and Logic Programming (FLOPS '08)*, pages 147–162. Springer *LNCS* 4989.

López-Fraguas, F. and Sánchez-Hernández, J. (1999). $\mathcal{TOY}$: A multiparadigm declarative system. In *10th Rewriting Techniques and Applications (RTA '99)*, pages 244–247. Springer *LNCS* 1631.

Lux, W. (2008). Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, pages 67–76.

Lux, W. (2009). Type-classes and call-time choice vs. run-time choice - Post to the Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html`.

Martin-Martin, E. (2009). Advances in type systems for functional logic programming. Master's thesis, Universidad Complutense de Madrid. `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`.

Martin-Martin, E. (2011). Type classes in functional logic programming. In *20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*, pages 121–130. ACM.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375.

Mitchell, J. C. and Plotkin, G. D. (1988). Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502.

Moreno-Navarro, J., Mariño, J., del Pozo-Pietro, A., Herranz-Nieva, A., and García-Martín, J. (1996). Adding type classes to functional-logic languages. In *Joint Conference on Declarative Programming, APPIA-GULP-PRODE'96*, pages 427–438.

Perry, N. (1991). *The Implementation of Practical Functional Programming Languages*. Ph.D. thesis, Imperial College, London.

Peyton Jones, S., Vytiniotis, D., and Weirich, S. (2006). Simple unification-based type inference for GADTs (Technical Appendix). Technical Report MS-CIS-05-22, University of Pennsylvania.

Sánchez-Hernández, J. (2006). Constructive failure in functional-logic programming: From theory to implementation. *Journal of Universal Computer Science*, 12(11):1574–1593.

Schrijvers, T., Peyton Jones, S., Sulzmann, M., and Vytiniotis, D. (2009). Complete and decidable type inference for GADTs. In *14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*, pages 341–352. ACM.

Seidel, D. and Voigtländer, J. (2010). Automatically generating counterexamples to naive free theorems. In *10th International Symposium on Functional and Logic Programming (FLOPS '10)*, pages 175–190. Springer LNCS 6009.

Thatté, S. R. (1994). Semantics of type classes revisited. In *8th ACM Conference on LISP and Functional Programming (LFP '94)*, pages 208–219. ACM.

van Bakel, S. and Fernández, M. (1997). Normalization results for typeable rewrite systems. *Information and Computation*, 133(2):73 – 116.

Wadler, P. (1989). Theorems for free! In *4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 347–359. ACM.

Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*, pages 60–76. ACM.

Warren, D. H. (1982). Higher-order extensions to prolog: are they needed? In *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd.

Wright, A. K. and Felleisen, M. (1992). A Syntactic Approach to Type Soundness. *Information and Computation*, 115:38–94.

Appendix A. Proofs and auxiliary results

This appendix contains complete proofs for all the results in the paper. We first present some notions used in the proofs:

a) For any type substitution π its *domain* is defined as $dom(\pi) = \{\alpha \mid \alpha\pi \neq \alpha\}$; and the *variable range* of π is $vran(\pi) = \bigcup_{\alpha \in dom(\pi)} ftv(\alpha\pi)$

b) Provided the domains of two type substitutions π_1 and π_2 are disjoint, the *simultaneous composition* $(\pi_1 + \pi_2)$ is defined as:

$$\alpha(\pi_1 + \pi_2) = \begin{cases} \alpha\pi_1 & \text{if } \alpha \in dom(\pi_1) \\ \alpha\pi_2 & \text{otherwise} \end{cases}$$

c) If A is a set of type variables, the *restriction* of a substitution π to A $(\pi|_A)$ is defined as:

$$\alpha(\pi|_A) = \begin{cases} \alpha\pi & \text{if } \alpha \in A \\ \alpha & \text{otherwise} \end{cases}$$

We use $\pi|_{\setminus A}$ as an abbreviation of $\pi|_{\mathcal{TV} \setminus A}$

A.1. *Auxiliary results*

Theorem A.1 shows that the type and substitution found by the inference are correct, i.e., we can build a type derivation for the same type if we apply the substitution to the assumptions.

Theorem A.1 (Soundness of $\Vdash\!\!\vdash$). $\mathcal{A} \Vdash\!\!\vdash e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash e : \tau$

Theorem A.2 expresses the completeness of the inference process. If we can derive a type for an expression applying a substitution to the assumptions, then inference will succeed and will find a type and a substitution which are more general.

Theorem A.2 (Completeness of $\Vdash\!\!\vdash$ wrt. $\vdash$). If $\mathcal{A}\pi' \vdash e : \tau'$ then $\exists \tau, \pi, \pi''$. $\mathcal{A} \Vdash\!\!\vdash e : \tau|\pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$.

The following theorem shows some useful properties of the typing relation $\vdash$, used in the proofs.

Theorem A.3 (Properties of the typing relation).

a) If $\mathcal{A} \vdash e : \tau$ then $\mathcal{A}\pi \vdash e : \tau\pi$, for any π
b) Let s be a symbol not appearing in e. Then $\mathcal{A} \vdash e : \tau \Longleftrightarrow \mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$, for any σ.
c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$ and $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ then $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.

Proof. The proof of Theorems A.1, A.2 and A.3 appears in Enrique Martin-Martin's master thesis (Martin-Martin, 2009). □

Remark A.1. If $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ and $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash\!\!\vdash e : \tau'|\pi$ with $\{\overline{\alpha_n}\} \cap ftv(\mathcal{A}) = \emptyset$ then we can assume that $\mathcal{A}\pi = \mathcal{A}$.

Explanation. If it is possible to derive a type for e with the assumptions $\mathcal{A}$, then the inference will not need to instantiate $\mathcal{A}$. Since $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})[\overline{\alpha_n/\tau_n}] \vdash e : \tau$ then by Theorem A.2 we know that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash\!\!\vdash e : \tau'|\pi$ and $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi\pi'' = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})[\overline{\alpha_n/\tau_n}]$ for some substitution π''. Therefore $\mathcal{A}\pi\pi'' = \mathcal{A}[\overline{\alpha_n/\tau_n}] = \mathcal{A}$, so π only replace variables in $\mathcal{A}$ which are restored by π''. These replacements are generated by unification steps that substitute free type variables in $\mathcal{A}$ for fresh type variables created during inference. Then we can assume that in these cases unification only replaces fresh variables, obtaining that $\mathcal{A}\pi = \mathcal{A}$. □

A.2. *Proof of Theorem 3.1*

Theorem 3.1
If $wt^{old}_{\mathcal{A}}(\mathcal{P})$ then $wt_{\mathcal{A}}(\mathcal{P})$.

Proof. In (López-Fraguas et al., 2010) and also in this paper the definition of well-typed program proceeds rule by rule, so we only have to prove that if $wt^{old}_{\mathcal{A}}(f\ t_1 \ldots t_n \to e)$ then $wt_{\mathcal{A}}(f\ t_1 \ldots t_n \to e)$. For the sake of conciseness we will consider functions with just one argument: $f\ t \to e$. Since patterns are linear (all the variables are different) the proof for functions with more arguments follows the same ideas.

From $wt^{old}_{\mathcal{A}}(f\ t \to e)$ we know that $\mathcal{A} \vdash^{\bullet} \lambda t.e : \tau'_t \to \tau'_e$, being $\tau'_t \to \tau'_e$ a variant of $\mathcal{A}(f)$. Then we have a type derivation of the form:

$$[\Lambda]\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau'_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau'_e}{\mathcal{A} \vdash \lambda t.e : \tau'_t \to \tau'_e}$$

and $critVar_{\mathcal{A}}(\lambda t.e) = \emptyset$, i.e., $opaqueVar_{\mathcal{A}}(t) \cap fv(e) = \emptyset$. We want to prove that:

a) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t : \tau_L|\pi_L$
b) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R|\pi_R$
c) $\exists\pi.(\tau_R, \overline{\beta_n\pi_R})\pi = (\tau_L, \overline{\alpha_n\pi_L})$
d) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$, $\mathcal{A}\pi = \mathcal{A}$

By the type derivation of t and Theorem A.2 we obtain the type inference

$$\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t$$

and there exists a type substitution π''_t such that $\tau_t\pi''_t = \tau'_t$ and $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t = \mathcal{A} \oplus \{\overline{X_n : \tau_n}\}$, i.e., $\mathcal{A}\pi_t\pi''_t = \mathcal{A}$ and $\alpha_i\pi_t\pi''_t = \tau_i$. Moreover, from $critVar_{\mathcal{A}}(\lambda t.e) = \emptyset$ we know that for every data variable $X_i \in fv(e)$ then $ftv(\alpha_i\pi_t) \subseteq ftv(\tau_t)$. Then we can build the type inference for the application $f\ t$:

$$[\mathbf{i}\Lambda]\frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f : \tau'_t \to \tau'_e|id \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})id \Vdash t : \tau_t|\pi_t}{\mathbf{a)}\ \mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t : \gamma\pi_g|\pi_t\pi_g}$$

By Remark A.1 we are sure that $\mathcal{A}\pi_t = \mathcal{A}$. Since $\tau'_t \to \tau'_e$ is a variant of $\mathcal{A}(f)$ we know that it contains only free type variables in $\mathcal{A}$ or fresh variables, so $(\tau'_t \to \tau'_e)\pi_t = \tau'_t \to \tau'_e$. In order to complete the type inference we need to create a unifier π_u for $(\tau'_t \to \tau'_e)\pi_t$ and $\tau_t \to \gamma$, being γ a fresh type variable. Notice that we had $\mathcal{A}\pi_t\pi''_t = \mathcal{A}$ and by Remark A.1 $\mathcal{A}\pi_t = \mathcal{A}$, so $\mathcal{A}\pi''_t = \mathcal{A}$. Since $\tau'_t \to \tau'_e$ is a variant of $\mathcal{A}(f)$ it contains only type variables which are free in $\mathcal{A}$ or fresh type variables, so π''_t will not affect it. Defining π_u as $\pi''_t|_{ftv(\tau_t)} + [\gamma/\tau'_e]$ we have an unifier, since:

$$\begin{array}{lll}
 & \underline{(\tau'_t \to \tau'_e)\pi_t\pi_u} & \pi_t \text{ does not affect } \tau'_t \to \tau'_e \\
= & \underline{(\tau'_t \to \tau'_e)\pi_u} & \gamma \notin ftv(\tau'_t \to \tau'_e) \\
= & (\tau'_t \to \tau'_e)\underline{\pi''_t|_{ftv(\tau_t)}} & \pi''_t|_{ftv(\tau_t)} \text{ does not affect } \tau'_t \to \tau'_e \\
= & \tau'_t \to \underline{\tau'_e} & \text{definition of } \pi_u \\
= & \underline{\tau'_t} \to \gamma\pi_u & \text{Theorem A.2: } \tau_t\pi''_t = \tau'_t \\
= & \underline{\tau_t\pi''_t|_{ftv(\tau_t)}} \to \gamma\pi_u & \gamma \notin ftv(\tau_t) \\
= & \underline{\tau_t\pi_u \to \gamma\pi_u} & \text{application of substitution} \\
= & (\tau_t \to \gamma)\pi_u &
\end{array}$$

Moreover, it is clear that π_u is a *most general unifier* of $(\tau'_t \to \tau'_e)\pi_t$ and $\tau_t \to \gamma$, so $\pi_g \equiv \pi''_t|_{ftv(\tau_t)} + [\gamma/\tau'_e]$.

By Theorem A.2 and the type derivation for e we obtain the type inference:

$$\mathbf{b)}\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_e|\pi_e$$

and there exists a type substitution π''_e such that $\tau_e\pi''_e = \tau'_e$ and $(\mathcal{A} \oplus \{\overline{X_n : \beta_n}\})\pi_e\pi''_e =$

$\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}$, i.e., $\mathcal{A}\pi_e\pi''_e = \mathcal{A}$ and $\beta_i\pi_e\pi''_e = \tau_i$. By Remark A.1 we also know that $\mathcal{A}\pi_e = \mathcal{A}$, so $\mathcal{A}\pi''_e = \mathcal{A}$.

To prove **c)** we need to find a type substitution π such that $(\tau_e, \overline{\beta_n\pi_e})\pi = (\gamma\pi_g, \overline{\alpha_n\pi_t\pi_g})$. Let I be the set containing the indexes of the data variables in t which appear in $fv(e)$ and N its complement. We can define the substitution π as the simultaneous composition:

$$\pi \equiv \pi''_e|_{\setminus\{\beta_i|i\in N\}} + [\overline{\beta_i/\alpha_i\pi_t\pi_g}]|_{\{\beta_i|i\in N\}}$$

This substitution is well defined because the domains of the two substitutions are disjoint. The first component is the substitution π''_e restricted to the variables which appear in its domain but not in $\{\beta_i|i \in N\}$, while the domain of the second component contains only the variables $\{\beta_i|i \in N\}$. Notice that the data variables in $\{X_i|i \in N\}$ do not occur in $fv(e)$ so they are not involved in the type inference for e. Therefore the type variables in $\{\beta_i|i \in N\}$ do not appear in $ftv(\tau_e)$, $dom(\pi_e)$ or $vran(\pi_e)$. With this substitution π the equality $(\tau_e, \overline{\beta_n\pi_e})\pi = (\gamma\pi_g, \overline{\alpha_n\pi_t\pi_g})$ holds because:

— Since $\tau_e\pi''_e = \tau'_e$ and the type variables in $\{\beta_i|i \in N\}$ do not occur in $ftv(\tau_e)$ we know that $\tau_e\pi = \tau_e\pi''_e|_{\setminus\{\beta_i|i\in N\}} = \tau_e\pi''_e = \tau'_e = \gamma\pi_g$.
— We know that the variables in $\{X_i|i \in I\}$ cannot be opaque in t, so $ftv(\alpha_i\pi_t) \subseteq ftv(\tau_t)$ for every $i \in I$ and $\alpha_i\pi_t\pi_g = \alpha_i\pi_t\pi''_t|_{ftv(\tau_t)} = \tau_i$ for those variables. Since the type variables $\{\beta_i|i \in N\}$ do not occur in $vran(\pi_e)$ then $\beta_i\pi_e\pi = \beta_i\pi_e\pi''_e|_{\setminus\{\beta_i|i\in N\}} = \beta_i\pi_e\pi''_e = \tau_i = \alpha_i\pi_t\pi_g$ for every $i \in I$.
— Since the type variables $\{\beta_i|i \in N\}$ do not occur in $dom(\pi_e)$ then $\beta_i\pi_e\pi = \beta_i\pi = \alpha_i\pi_t\pi_g$ for every $i \in N$.

Finally, we have to prove that **d)** $\mathcal{A}\pi_t\pi_g = \mathcal{A}$, $\mathcal{A}\pi_e = \mathcal{A}$ and $\mathcal{A}\pi = \mathcal{A}$. For the first case we already know that $\mathcal{A}\pi_t = \mathcal{A}$ and $\mathcal{A}\pi''_t = \mathcal{A}$. Since π_g is defined as $\pi''_t|_{ftv(\tau_t)} + [\gamma/\tau'_e]$ and γ is a fresh type variable not appearing in $ftv(\mathcal{A})$ then $\mathcal{A}\pi_t\pi_g = \mathcal{A}\pi_g = \mathcal{A}\pi''_t|_{ftv(\tau_t)} = \mathcal{A}$. For the second case, $\mathcal{A}\pi_e = \mathcal{A}$ holds using Remark A.1. For the last case we know that $\mathcal{A}\pi''_e = \mathcal{A}$. Since π is defined as $\pi''_e|_{\setminus\{\beta_i|i\in N\}} + \{\beta_i/\alpha_i\pi_t\pi_g|i \in N\}$ and no type variable β_i appears in $ftv(\mathcal{A})$ (they are fresh type variables) then $\mathcal{A}\pi = \mathcal{A}\pi''_e = \mathcal{A}$.

□

A.3. *Proof of Theorem 4.1: Progress*

Theorem 4.1 (Progress)
If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then either e is a pattern or $\exists e'.\ \mathcal{P} \vdash e \rightarrowtail e'$.

Proof. By induction over the structure of e
Base case

X)This cannot happen because e is ground.
$c \in CS^n$)Then c is a pattern, regardless of its arity n. This case covers $e \equiv fail$.
$f \in FS^n$)Depending on n there are two cases:

— $n > 0$) Then f is a partially applied function symbols, so it is a pattern.
— $n = 0$) If there is a rule $(f \rightarrow e) \in \mathcal{P}$ then we can apply rule (Fapp), so $\mathcal{P} \vdash s \rightarrowtail e$.

Otherwise there is not any rule $(l \to e') \in \mathcal{P}$ such that l and f unify, so we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash f \rightarrowtail \mathit{fail}$.

Inductive Step

$e_1\ e_2$)From the premises we know that there is a type derivation:

$$[\mathbf{APP}]\frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

Both e_1 and e_2 are well-typed and ground. If e_1 is not a pattern, by the Induction Hypothesis we have $\mathcal{P} \vdash e_1 \rightarrowtail e_1'$ and using the (Contx) rule we obtain $\mathcal{P} \vdash e_1\ e_2 \rightarrowtail e_1'\ e_2$. If e_2 is not a pattern we can apply the same reasoning. Therefore we only have to treat the case when both e_1 and e_2 are patterns. We make a distinction over the structure of the pattern e_1:

— X) This cannot happen because e_1 is ground.
— $c\ t_1 \dots t_n$ with $c \in CS^m$ and $n \leq m$) Depending on m and n we distinguish two cases:
- $n < m$) Then $e_1\ e_2$ is $c\ t_1 \dots t_n\ e_2$ with $n+1 \leq m$, which is a pattern.
- $n = m$)
  - If $c = \mathit{fail}$ then $m = n = 0$, so we have the expression $\mathit{fail}\ e_2$. In this case we can apply rule (FailP), so $\mathcal{P} \vdash \mathit{fail}\ e_2 \rightarrowtail \mathit{fail}$.
  - Otherwise $e_1\ e_2$ is $c\ t_1 \dots t_n\ e_2$ with $n+1 > m$, which is *junk*. This cannot happen because $\mathcal{A} \vdash e_1\ e_2 : \tau$, and Lemma A.2 states that *junk* expressions cannot be well-typed wrt. any set of assumptions.

— $f\ t_1 \dots t_n$ with $c \in FS^m$ and $n < m$) Depending on m and n we distinguish two cases:
- $n+1 < m$) Then $e_1\ e_2$ is $f\ t_1 \dots t_n\ e_2$ which is a partially applied function symbol, i.e., a pattern.
- $n + 1 = m$) Then $e_1\ e_2$ is $f\ t_1 \dots t_n\ e_2$. If there is a rule $(l \to r) \in \mathcal{P}$ such that $l\theta = f\ t_1 \dots t_n\ e_2$ then we can apply rule (Fapp), so $\mathcal{P} \vdash e_1\ e_2 \rightarrowtail r\theta$. If such a rule does not exist, then there is not any rule $(l' \to r') \in \mathcal{P}$ such that l' and $f\ t_1 \dots t_n\ e_2$ unify. Therefore we can apply the rule for the matching failure (Ffail) obtaining $\mathcal{P} \vdash e_1\ e_2 \rightarrowtail \mathit{fail}$.

$\mathit{let}\ X = e_1\ \mathit{in}\ e_2$)From the premises we know that there is a type derivation:

$$[\mathbf{LET}]\frac{\mathcal{A} \vdash e_1 : \tau_X \quad \mathcal{A} \oplus \{X : Gen(\tau_X, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash \mathit{let}\ X = e_1\ \mathit{in}\ e_2 : \tau}$$

There are two cases depending on whether e_1 is a pattern or not:

— e_1 is a pattern) Then we can use the (Bind) rule, obtaining $\mathcal{P} \vdash \mathit{let}\ X = e_1\ \mathit{in}\ e_2 \rightarrowtail e_2[X/e_1]$.
— e_1 is not a pattern) Since $\mathit{let}\ X = e_1\ \mathit{in}\ e_2$ is ground we know that e_1 is ground (notice that this does not force e_2 to be ground). Moreover, $\mathcal{A} \vdash e_1 : \tau_t$, so by the Induction Hypothesis we can rewrite e_1 to some e_1': $\mathcal{P} \vdash e_1 \rightarrowtail e_1'$. Using the

(Contx) rule we can transform this local step into a step in the whole expression: $\mathcal{P} \vdash let\ X = e_1\ in\ e_2 \rightarrowtail let\ X = e_1'\ in\ e_2$.

□

A.4. *Proof of Theorem 4.2: Type Preservation*

Theorem 4.2 (Type Preservation)
If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ *and* $\mathcal{P} \vdash e \rightarrowtail e'$, *then* $\mathcal{A} \vdash e' : \tau$.

Proof. We proceed by case distinction over the rule of the *let*-rewriting relation $\rightarrowtail$ (Figure 2) used to reduce e to e'.

(Fapp) If we reduce an expression e using the (Fapp) rule is because e has the form $f\ t_1\theta \ldots t_m\theta$ (being $f\ t_1 \ldots t_m \rightarrow r$ a rule in $\mathcal{P}$) and e' is $r\theta$. In this case we want to prove that $\mathcal{A} \vdash r\theta : \tau$. Since $wt_{\mathcal{A}}(\mathcal{P})$ then $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \rightarrow r)$, and by the definition of well-typed rule (Definition 3.2) we have:

(A) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t_1 \ldots t_m : \tau_L | \pi_L$

(B) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash r : \tau_R | \pi_R$

(C) $\exists \pi.\ (\tau_R, \overline{\beta_n \pi_R})\pi = (\tau_L, \overline{\alpha_n \pi_L})$

(D) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$ and $\mathcal{A}\pi = \mathcal{A}$.

By the premises we have the derivation

$$(E)\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau$$

where $\theta = [\overline{X_n / t_n'}]$. Since the type derivation *(E)* exists, then there exists also a type derivation for each pattern t_i': *(F)* $\mathcal{A} \vdash t_i' : \tau_i$. Notice that these $\overline{\tau_n}$ are unique as the left-hand side of the rule is linear, so each t_i' will appear once.

If we replace every pattern t_i' in the type derivation *(E)* by their associated variable X_i and we add the assumptions $\{\overline{X_n : \tau_n}\}$ to $\mathcal{A}$, we obtain the type derivation:

$$(G)\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$$

By *(A)* and *(G)* and Theorem A.2 we have *(H)* $\exists \pi_1.\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_L\pi_1 = \mathcal{A} \oplus \{\overline{X_n : \tau_n}\}$ and $\tau_L\pi_1 = \tau$. Therefore $\mathcal{A}\pi_L\pi_1 = \mathcal{A}$ and $\alpha_i\pi_L\pi_1 = \tau_i$ for each i.

By *(B)* and the soundness of the inference (Theorem A.1):

$$(I)\mathcal{A}\pi_R \oplus \{\overline{X_n : \beta_n\pi_R}\} \vdash r : \tau_R$$

Using the fact that type derivations are closed under substitutions (Theorem A.3-a) we can add the substitution π of *(C)* to *(I)*, obtaining:

$$(J)\mathcal{A}\pi_R\pi \oplus \{\overline{X_n : \beta_n\pi_R\pi}\} \vdash r : \tau_R\pi$$

By *(J)* y *(C)* we have that *(K)* $\mathcal{A}\pi_R\pi \oplus \{\overline{X_n : \alpha_n\pi_L}\} \vdash r : \tau_L$

Using the closure under substitutions of type derivations (Theorem A.3-a) we can add

the substitution π_1 of *(H)* to *(K)*:

$$(L)\mathcal{A}\pi_R\pi\pi_1 \oplus \{\overline{X_n : \alpha_n\pi_L\pi_1}\} \vdash r : \tau_L\pi_1$$

By *(L)* and *(H)* we have *(M)* $\mathcal{A}\pi_R\pi\pi_1 \oplus \{\overline{X_n : \tau_n}\} \vdash r : \tau$

By $\mathcal{A}\pi_L = \mathcal{A}$ *(D)* and $\mathcal{A}\pi_L\pi_1 = \mathcal{A}$ *(H)* we know that *(N)* $\mathcal{A}\pi_1 = \mathcal{A}$.

From *(D)* and *(N)* follows *(O)* $\mathcal{A}\pi_R\pi\pi_1 = \mathcal{A}\pi\pi_1 = \mathcal{A}\pi_1 = \mathcal{A}$.

By *(O)* and *(M)* we have *(P)* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash r : \tau$

Using Theorem A.3-b) we can add the type assumptions $\{\overline{X_n : \tau_n}\}$ to the type derivations in *(F)*, obtaining *(Q)* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t'_i : \tau_i$. Notice that we assume that $\overline{X_n}$ do not appear in $t'_i \equiv X_i\theta$, as $\overline{X_n}$ are the variables of the rule.

By Theorem A.3-c) we can replace the data variables $\overline{X_n}$ in *(P)* by expressions of the same type. We use the patterns $\overline{t'_n}$ in *(Q)*:

$$(R)\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash r\theta : \tau$$

Finally, the data variables $\overline{X_n}$ do not appear in $r\theta$, so by Theorem A.3-b) we can erase that assumptions in *(R)*:

$$(S)\mathcal{A} \vdash r\theta : \tau$$

(Ffail) and (FailP) Straightforward since in both cases e' is *fail*. A type derivation $\mathcal{A} \vdash \mathit{fail} : \tau$ is possible for any τ since $\mathcal{A}$ contains the assumption $\mathit{fail} : \forall\alpha.\alpha$.

The rest of the cases are the same as the proof in Enrique Martin-Martin's master thesis (Martin-Martin, 2009).

□

A.5. *Proof of Theorem 4.3: Maximal liberality of well-typedness conditions*

In order to prove Theorem 4.3 we will use an auxiliary result relating the types involved in type derivations to the types inferred by a type inference:

Lemma A.1. Given a closed set of assumptions $\mathcal{A}$, if $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash e : \tau_g|\pi_g$ and $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ (for some $\overline{\alpha_n}$ fresh) then there exists some π such that $\tau_g\pi = \tau$ and $\alpha_i\pi_g\pi = \tau_i$ for every $i \in [1..n]$.

Proof. Straightforward by Theorem A.2 with $\pi' \equiv [\overline{\alpha_n/\tau_n}]$. □

Theorem 4.3 (Maximal liberality of well-typedness conditions)
Let $\mathcal{A}$ be a closed set of assumptions, and assume that $\mathcal{P}$ is a program which is not well-typed wrt. $\mathcal{A}$, but such that every rule $R \in \mathcal{P}$ verifies the condition i) of well-typedness in Definition 3.2. Then there exists a rule $(f\ t_1 \ldots t_m \to e) \in \mathcal{P}$ with variables $\overline{X_n}$ and there exist types $\overline{\tau_n}$, τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ and $f\ t_1 \ldots t_m \rightarrowtail e$ but $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \nvdash e : \tau$.

Proof. For every rule, *i*) holds by hypothesis and *iv*) holds trivially as $\mathcal{A}$ is closed. Therefore either condition *ii*) or *iii*) must fail for some rule $R \equiv (f\ t_1 \ldots t_m \to e) \in \mathcal{P}$. The condition *i*) says that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t_1 \ldots t_m : \tau_L|\pi_L$, for some τ_L, π_L. Then, by the soundness of $\Vdash$ (Theorem A.1) we have

$$(1) \quad \mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_L}\} \vdash f\ t_1 \ldots t_m : \tau_L$$

Moreover, using (Fapp) and the rule R it is possible to perform the rewrite step

$$(2) \quad f\ t_1 \ldots t_m \rightarrowtail_{(Fapp)} e$$

We will now see that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_L}\} \nvdash e : \tau_L$, which will finish the proof by taking $\overline{\tau_n} = \overline{\alpha_n\pi_L}$ and $\tau = \tau_L$. We distinguish two cases depending on which of the conditions *ii*) or *iii*) in Definition 3.2 fails for the rule R.

a) If *ii*) does not hold for R then by the completeness of $\Vdash$ (Theorem A.2) there are not any types $\overline{\tau_n}$, τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$, so in particular $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_L}\} \nvdash e : \tau_L$ as desired.

b) If *ii*) holds but *iii*) does not, then we have that there exist some τ_R, π_R such that

$$(3) \quad \mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R|\pi_R \quad \text{by } ii)$$

$$(4) \quad \neg\exists\pi.(\tau_L, \overline{\alpha_n\pi_L}) = (\tau_R, \overline{\beta_n\pi_R})\pi \quad \text{by failure of } iii)$$

Condition (4) is equivalent to say that

$$(5) \quad \forall\pi.(\tau_L = \tau_R\pi \Longrightarrow \exists i \in [1..n].\alpha_i\pi_L \neq \beta_i\pi_R\pi)$$

We reason now by contradiction, assuming that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_L}\} \vdash e : \tau_L$ (we want to prove the contrary). Then by (3) and Lemma A.1 we have that there is some π such that $\tau_R\pi = \tau_L$ and $\beta_i\pi_R\pi = \alpha_i\pi_L$ for every $i \in [1..n]$, which contradicts (5).

□

The previous proof is constructive since it shows that given a rule $(f\ t_1 \ldots t_m \to e) \in \mathcal{P}$ not holding *ii*) or *iii*), the evaluation step $f\ t_1 \ldots t_m \rightarrowtail_{(Fapp)} e$ never preserves types using $\overline{\tau_n} = \overline{\alpha_n\pi_L}$ and $\tau = \tau_L$.

The following examples illustrates the lost of type preservation in the different cases considered in the proof. The rule $f_1 \to not\ [\,]$ with assumption $f_1 : bool$ does not verify point *ii*) since the right-hand side is ill-typed. In this case it is easy to check that $\mathcal{A} \vdash f_1 : bool$ and $f_1 \rightarrowtail not\ [\,]$, but $\mathcal{A} \nvdash not\ [\,] : bool$—indeed, $not\ [\,]$ does not have any type. The rule $f_2 \to true$ with assumption $f_2 : nat$ verifies point *ii*) but not *iii*) because *bool* does not match *nat*, which corresponds to the case when (5) holds because the antecedent in the implication always fails. Trivially $\mathcal{A} \vdash f_2 : nat$ and $f_2 \rightarrowtail true$, but $\mathcal{A} \nvdash true : nat$. Finally, the rule $f_3\ X \to not\ X$ with assumption $f_3 : \forall\alpha.\alpha \to bool$ illustrates the case when point *ii*) holds but *iii*) does not, although in this case the antecedent $\tau_L = \tau_R\pi$ of (5) holds for some π (for any π indeed, since $\tau_L = \tau_R = bool$). What happens here is that the type *bool* inferred for the variable X in the right-hand side does not match the type α inferred in the left-hand side. In this case it is clear that $\mathcal{A} \oplus \{X : \alpha\} \vdash f_3\ X : bool$ and $f_3\ X \rightarrowtail not\ X$, but $\mathcal{A} \oplus \{X : \alpha\} \nvdash not\ X : bool$.

A.6. *Proof of Proposition 4.1*

Proposition 4.1
Consider a type-complete set of assumptions $\mathcal{A}$, and a program rule $R \equiv f\ t_1 \ldots t_m \to e$. Then R preserves types iff $wt_{\mathcal{A}}(R)$.

Proof.

$\Longrightarrow$) We proceed proving the contrapositive

$$\neg wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e) \Longrightarrow f\ t_1 \ldots t_m \to e \text{ is not type-preserving}$$

If $f\ t_1 \ldots t_m \to e$ is not well-typed because it does not verify point i) of Definition 3.2 then by completeness of type inference (Theorem A.2) the left-hand side of the rule does not admit any type, so the rule is not type-preserving.
If $f\ t_1 \ldots t_m \to e$ is not well-typed but it verifies the point point i) of Definition 3.2 then by soundness of type inference (Theorem A.1) its left-hand side admits some—point i) of Definition 4.1 of type-preserving rule. In this case we have to prove that

$$\neg wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e) \Longrightarrow \exists\theta.(\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau \wedge \mathcal{A} \nvdash e\theta : \tau)$$

As the the point i) of the definition of well-typed rule is verified, by Theorem 4.3 (maximal liberality of well-typedness conditions) we know that there are types $\overline{\tau_n}$ and τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ and $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \nvdash e : \tau$. The set of assumptions $\mathcal{A}$ is type-complete, so there are patterns $\overline{t_{\tau_n}}$ which can only have those types, i.e., $\mathcal{A} \vdash t_{\tau_i} : \tau_i$. As the variables $\overline{X_n}$ are the variables of the rule we can assume that they do not appear in the patterns $\overline{t_{\tau_n}}$, so by Theorem A.3-b) we have that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t_{\tau_i} : \tau_i$. Using Theorem A.3-c) we can replace the variables $\overline{X_n}$ in $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1 \ldots t_m : \tau$ by the patterns of the same type with the substitution $\theta \equiv [X_n/t_{\tau_n}]$, obtaining $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash f\ t_1\theta \ldots t_m\theta : \tau$. Again by Theorem A.3-b) we can remove the variables $\overline{X_n}$ from the set of assumptions as they do not ocur in $f\ t_1\theta \ldots t_m\theta$, obtaining $\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau$. On the other hand, it is easy to check that $\mathcal{A} \nvdash e\theta : \tau$ because $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \nvdash e : \tau$ and we replace the variables $\overline{X_n}$ by patterns $\overline{t_{\tau_n}}$ which can only have those types $\overline{\tau_n}$.
$\Longleftarrow$) If $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$ then from point i) of Definition 3.2 (well-typed rule) and Theorem A.1 (soundness of type inference), the left-hand side of the rule admits some type—point i) of Definition 4.1 (type-preserving rule). Regarding the point ii) of Definition 4.1, consider an arbitrary θ and τ such that $\mathcal{A} \vdash f\ t_1\theta \ldots t_m\theta : \tau$. Following the same reasoning as in the proof for the (Fapp) rule in Theorem 4.2 (type preservation) we conclude that $\mathcal{A} \vdash e\theta : \tau$.

□

A.7. *Proof of Lemma 4.1: Faulty Expressions are ill-typed*

In order to prove Lemma 4.1 we use an auxiliary result stating that *junk* expressions cannot have a valid type wrt. any set of assumptions $\mathcal{A}$:

Lemma A.2. If e is a *junk* expression then there is no $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$.

Proof. By contradiction. Assume there is $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$. If e is *junk* then it has the form $c\ t_1 \ldots t_n$ with $c \in CS^m$ and $n > m$, i.e., $(c\ t_1 \ldots t_m)\ t_{m+1} \ldots t_n$. The type derivation for e must contain a subderivation of the form:

$$[\mathbf{APP}]\frac{\mathcal{A} \vdash (c\ t_1 \ldots t_m) : \tau_1 \to \tau \quad \mathcal{A} \vdash t_{m+1} : \tau_1}{\mathcal{A} \vdash (c\ t_1 \ldots t_m)\ t_{m+1} : \tau}$$

Any possible type derived for the symbol c has the form $\tau'_1 \to \ldots \to \tau'_m \to (C\ \tau''_1 \ldots \tau''_k)$. Then after m applications of the **[APP]** rule the type derived for $c\ t_1 \ldots t_m$ is $C\ \tau''_1 \ldots \tau''_k$. This is not a functional type $(\tau_1 \to \tau)$, so we have found a contradiction. □

Using the previous result, we can prove Lemma 4.1:

Lemma 4.1 (Faulty Expressions are ill-typed)
If e is faulty then there is no $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$.

Proof. We prove it by contradiction. Suppose that e has a junk subexpression e' and $\mathcal{A} \vdash e : \tau$. Therefore, in that derivation we have a subderivation $\mathcal{A}' \vdash e' : \tau'$ (for some $\mathcal{A}'$ and τ'). By Lemma A.2 those $\mathcal{A}'$ and τ' cannot exist, so we have found a contradiction. □

A.8. *Proof of Theorem 4.4: Syntactic Soundness*

We need some auxiliary results:

Lemma A.3 (Well-typed normal forms are patterns). If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$, e is ground and e is a normal form then e is a pattern.

Proof. Straightforward from progress (Theorem 4.1).

□

Lemma A.4. If $\mathcal{P} \vdash e \rightarrowtail e'$ and $\mathcal{P}$ does not contains extra variables in its rules, then $fv(e') \subseteq fv(e)$.

Proof. Easily by case distinction over the rule applied in the step $\mathcal{P} \vdash e \rightarrowtail e'$.

□

From the previous lemma follows an useful corollary:

Corollary A.1. If e is ground, $\mathcal{P} \vdash e \rightarrowtail^* e'$ and $\mathcal{P}$ does not contains extra variables in its rules, then e' is ground.

Using the previous results, the proof of Theorem 4.4 is straightforward:

Theorem 4.4 (Syntactic Soundness) *If $wt_{\mathcal{A}}(\mathcal{P})$, e is ground and $\mathcal{A} \vdash e : \tau$ then: for all $e' \in nf_{\mathcal{P}}(e)$, e' is a pattern and $\mathcal{A} \vdash e' : \tau$.*

Proof. Let e' be an arbitrary expression in $nf_{\mathcal{P}}(e)$. Since e is ground, by Corollary A.1 e' is also ground. Applying Type Preservation (Theorem 4.2) in all the reduction steps we have $\mathcal{A} \vdash e' : \tau$. Since e' is a well-typed normal form, by Lemma A.3 e' is a pattern. □

Liberal Typing for Functional Logic Programs* **

Francisco López-Fraguas, Enrique Martin-Martin, and Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, emartinm@fdi.ucm.es, juanrh@fdi.ucm.es

Abstract. We propose a new type system for functional logic programming which is more liberal than the classical Damas-Milner usually adopted, but it is also restrictive enough to ensure type soundness. Starting from Damas-Milner typing of expressions we propose a new notion of well-typed program that adds support for type-indexed functions, existential types, opaque higher-order patterns and generic functions—as shown by an extensive collection of examples that illustrate the possibilities of our proposal. In the negative side, the types of functions must be declared, and therefore types are checked but not inferred. Another consequence is that parametricity is lost, although the impact of this flaw is limited as "free theorems" were already compromised in functional logic programming because of non-determinism.

Keywords: Type systems, functional logic programming, generic functions, type-indexed functions, existential types, higher-order patterns.

1 Introduction

Functional logic programming. Functional logic languages [9] like TOY [19] or Curry [10] have a strong resemblance to lazy functional languages like Haskell [13]. A remarkable difference is that functional logic programs (FLP) can be non-confluent, giving raise to so-called *non-deterministic functions*, for which a *call-time choice* semantics [6] is adopted. The following program is a simple example, using natural numbers given by the constructors z and s—we follow syntactic conventions of some functional logic languages where function and constructor names are lowercased, and variables are uppercased—and assuming a natural definition for *add*: $\{\ f\ X \to X,\ f\ X \to s\ X,\ double\ X \to add\ X\ X\ \}$. Here, f is non-deterministic ($f\ z$ evaluates both to z and $s\ z$) and, according to call-time choice, *double (f z)* evaluates to z and *s (s z)* but not to $s\ z$. Operationally,

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S2009TIC-1465 and UCM-BSCH-GR58/08-910502.

** This is the authors' version of the work. The definitive version was published in PROGRAMMING LANGUAGES AND SYSTEMS, Lecture Notes in Computer Science, 2010, Volume 6461/2010, 80-96, DOI: 10.1007/978-3-642-17164-2_7, http://www.springerlink.com/content/u5q8016158402754/. The original publication is available at www.springerlink.com

call-time choice means that all copies of a non-deterministic subexpression ($f\ z$ in the example) created during reduction share the same value.

In the HO-CRWL[1] approach to FLP [7], followed by the TOY system, programs can use *HO-patterns* (essentially, partial applications of symbols to other patterns) in left hand sides of function definitions. This corresponds to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics. This is not an exoticism: it is known [18] that extensionality is not a valid principle within the combination of HO, non-determinism and call-time choice. It is also known that *HO-patterns* cause some bad interferences with types: [8] and [17] considered that problem, and this paper improves on those results.

All those aspects of FLP play a role in the paper, and Sect. 3 uses a formal setting according to that. However, most of the paper can be read from a functional programming perspective leaving aside the specificities of FLP.

Types, FLP and genericity. FLP languages are typed languages adopting classical Damas-Milner types [5]. However, their treatment of types is very simple, far away from the impressive set of possibilities offered by functional languages like Haskell: type and constructor classes, existential types, GADTs, generic programming, arbitrary-rank polymorphism ... Some exceptions to this fact are some preliminary proposals for type classes in FLP [23,20], where in particular a technical treatment of the type system is absent.

By the term *generic programming* we refer generically to any situation in which a program piece serves for a family of types instead of a single concrete type. Parametric polymorphism as provided by Damas-Milner system is probably the main contribution to genericity in the functional programming setting. However, in a sense it is 'too generic' and leaves out many functions which are generic by nature, like equality. Type classes [26] were invented to deal with those situations. Some further developments of the idea of generic programming [11] are based on type classes, while others [12] have preferred to use simpler extensions of Damas-Milner system, such as GADTs [3,25]. We propose a modification of Damas-Milner type system that accepts natural definitions of intrinsically generic functions like equality. The following example illustrates the main points of our approach.

An introductory example. Consider a program that manipulates Peano natural numbers, booleans and polymorphic lists. Programming a function *size* to compute the number of constructor occurrences in its argument is an easy task in a type-free language with functional syntax:

size true $\rightarrow$ *s z* *size false* $\rightarrow$ *s z*
size z $\rightarrow$ *s z* *size (s X)* $\rightarrow$ *s (size X)*
size nil $\rightarrow$ *s z* *size (cons X Xs)* $\rightarrow$ *s (add (size X) (size Xs))*

However, as far as *bool*, *nat* and $[\alpha]$ are different types, this program would be rejected as ill-typed in a language using Damas-Milner system, since we obtain contradictory types for different rules of *size*. This is a typical case where

[1] CRWL [6] stands for *Constructor Based Rewriting Logic*; HO-CRWL is a higher order extension of it.

one wants some support for genericity. Type classes certainly solve the problem if you define a class *Sizeable* and declare *bool, nat* and $[\alpha]$ as instances of it. GADT-based solutions would add an explicit representation of types to the encoding of *size* converting it into a so-called *type-indexed* function [12]. This kind of encoding is also supported by our system (see the *show* function in Ex. 1 and *eq* in Fig 4-b later), but the interesting point is that our approach allows also a simpler solution: the program above becomes well-typed in our system simply by declaring *size* to have the type $\forall\alpha.\alpha \to nat$, of which each rule of *size* gives a more concrete instance. A detailed discussion of the advantages and disadvantages of such liberal declarations appears in Sect. 6 (see also Sect. 4).

The proposed well-typedness criterion requires only a quite simple additional check over usual type inference for expressions, but here 'simple' does not mean 'naive'. Imposing the type of each function rule to be an instance of the declared type is a too weak requirement, leading easily to type unsafety. As an example, consider the rule $f\ X \to not\ X$ with the assumptions $f : \forall\alpha.\alpha \to bool$, $not : bool \to bool$. The type of the rule is $bool \to bool$, which is an instance of the type declared for f. However, that rule does not preserve the type: the expression $f\ z$ is well-typed according to f's declared type, but reduces to the ill-typed expression $not\ z$. Our notion of well-typedness, roughly explained, requires also that right-hand sides of rules do not restrict the types of variables more than left-hand sides, a condition that is violated in the rule for f above. Def. 1 in Sect. 3.3 states that point with precision, and allows us to prove type soundness for our system.

Contributions. We give now a list of the main contributions of our work, presenting the structure of the paper at the same time:

• After some preliminaries, in Sect. 3 we present a novel notion of well-typed program for FLP that induces a simple and direct way of programming type-indexed and generic functions. The approach supports also existential types, opaque HO-patterns and GADT-like encodings, not available in current FLP systems.

• Sect. 4 is devoted to the properties of our type system. We prove that well-typed programs enjoy *type preservation*, an essential property for a type system; then by introducing *failure* rules to the formal operational calculus, we also are able to ensure the *progress* property of well-typed expressions. Based on those results we state type soundness. Complete proofs can be found in [16].

• In Sect. 5 we give a significant collection of examples showing the interest of the proposal. These examples cover type-indexed functions, existential types, opaque higher-order patterns and generic functions. None of them is supported by existing FLP systems.

• Our well-typedness criterion goes far beyond the solutions given in previous works [8,17] to type-unsoundness problems of the use of *HO-patterns* in function definitions. We can type equality, solving known problems of *opaque decomposition* [8] (Sect. 5.1) and, most remarkably, we can type the *apply* function appearing in the HO-to-FO translation used in standard FLP implementations (Sect. 5.2).

• Finally we discuss in Sect. 6 the strengths and weaknesses of our proposal, and we end up with some conclusions in Sect. 7.

2 Preliminaries

We assume a signature $\Sigma = CS \cup FS$, where CS and FS are two disjoint sets of *data constructor* and *function* symbols resp., all of them with associated arity. We write CS^n (resp. FS^n) for the set of constructor (function) symbols of arity n, and if a symbol h is in CS^n or FS^n we write $ar(h) = n$. We consider a special constructor $fail \in CS^0$ to represent pattern matching failure in programs as it is proposed for GADTs [3,24]. We also assume a denumerable set $\mathcal{DV}$ of *data variables* X. Fig. 1 shows the syntax of *patterns* $\in Pat$—our notion of values—and *expressions* $\in Exp$. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ fot_1 \ldots fot_n$ where $ar(c) = n$, and *higher order patterns* $HOPat = Pat \setminus FOPat$, i.e., patterns containing some partial application of a symbol of the signature. Expressions $c\ e_1 \ldots e_n$ are called *junk* if $n > ar(c)$ and $c \neq fail$, and expressions $f\ e_1 \ldots e_n$ are called *active* if $n \geq ar(f)$. The set of *free variables* of an expression—$fv(e)$—is defined in the usual way. Notice that since our let expressions do not support recursive definitions the binding of the variable only affect e_2: $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$. We say that an expression e is *ground* if $fv(e) = \emptyset$. A *one-hole context* is defined as $\mathcal{C} ::= [] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid let\ X = \mathcal{C}\ in\ e \mid let\ X = e\ in\ \mathcal{C}$. A *data substitution* θ is a finite mapping from data variables to patterns: $[\overline{X_n/t_n}]$. Substitution application over data variables and expressions is defined in the usual way. The empty substitution is written as id. A *program rule* r is defined as $f\ \overline{t_n} \to e$ where the set of patterns $\overline{t_n}$ is linear (there is not repetition of variables), $ar(f) = n$ and $fv(e) \subseteq \bigcup_{i=1}^{n} var(t_i)$. Therefore, extra variables are not considered in this paper. The constructor *fail* is not supposed to occur in the rules, although it does not produce any technical problem. A program $\mathcal{P}$ is a set of program rules: $\{r_1, \ldots, r_n\} (n \geq 0)$.

For the types we assume a denumerable set $\mathcal{TV}$ of *type variables* α and a countable alphabet $\mathcal{TC} = \bigcup_{n \in \mathbb{N}} \mathcal{TC}^n$ of *type constructors* C. As before, if $C \in \mathcal{TC}^n$ then we write $ar(C) = n$. Fig. 1 shows the syntax of *simple types* and *type-schemes*. The set of *free type variables (ftv)* of a simple type τ is $var(\tau)$, and for type-schemes $ftv(\forall \overline{\alpha_n}.\tau) = ftv(\tau) \setminus \{\overline{\alpha_n}\}$. We say a type-scheme σ is *closed* if $ftv(\sigma) = \emptyset$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_n : \sigma_n}\}$, where $s_i \in CS \cup FS \cup \mathcal{DV}$. We require set of assumptions to be *coherent* wrt. CS, i.e., $\mathcal{A}(fail) = \forall \alpha.\alpha$ and for every c in $CS^n \setminus \{fail\}$, $\mathcal{A}(c) = \forall \overline{\alpha}.\tau_1 \to \ldots \to \tau_n \to (C\ \tau'_1 \ldots \tau'_m)$ for some type constructor C with $ar(C) = m$. Therefore the assumptions for constructors must correspond to their arity and, as in [3,24], the constructor *fail* can have any type. The union of sets of assumptions is denoted by $\oplus$: $\mathcal{A} \oplus \mathcal{A}'$ contains all the assumptions in $\mathcal{A}'$ and the assumptions in $\mathcal{A}$ over symbols not appearing in $\mathcal{A}'$. For sets of assumptions $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. Notice that type-schemes for data constructors may be existential, i.e., they can be of the form $\forall \overline{\alpha_n}.\overline{\tau} \to \tau'$ where $(\bigcup_{\tau_i \in \overline{\tau}} ftv(\tau_i)) \setminus ftv(\tau') \neq \emptyset$. If $(s : \sigma) \in \mathcal{A}$ we write $\mathcal{A}(s) = \sigma$. A *type*

Data variables	$X, Y, Z, \ldots$	Patterns	$t ::= X$
Type variables	$\alpha, \beta, \gamma, \ldots$		$\mid\ c\ t_1 \ldots t_n$ if $n \leq ar(c)$
Data constructors	c		$\mid\ f\ t_1 \ldots t_n$ if $n < ar(f)$
Type constructors	C	Simple Types	$\tau ::= \alpha$
Function symbols	f		$\mid\ C\ \tau_1 \ldots \tau_n$ if $ar(C) = n$
			$\mid\ \tau \to \tau$
Expressions	$e ::= X \mid c \mid f \mid e\ e$	Type Schemes	$\sigma ::= \forall\overline{\alpha_n}.\tau$
	$\mid\ let\ X = e\ in\ e$	Assumptions	$\mathcal{A} ::= \{s_1 : \sigma_1, \ldots, s_n : \sigma_n\}$
Symbol	$s ::= X \mid c \mid f$	Program rule	$r ::= f\ \bar{t} \to e\ \ (\bar{t}$ linear$)$
Non variable symbol	$h ::= c \mid f$	Program	$\mathcal{P} ::= \{r_1, \ldots, r_n\}$
Data substitution	$\theta ::= [\overline{X_n/t_n}]$	Type substitution	$\pi ::= [\overline{\alpha_n/\tau_n}]$

Fig. 1. Syntax of expressions and programs

(Fapp) $f\ t_1\theta \ldots t_n\theta \ \to^{lf}\ r\theta$, if $(f\ t_1 \ldots t_n \to r) \in \mathcal{P}$

(Ffail) $f\ t_1 \ldots t_n \ \to^{lf}\ fail$, if $n = ar(f)$ and $\nexists (f\ t'_1 \ldots t'_n \to r) \in \mathcal{P}$ such that $f\ t'_1 \ldots t'_n$ and $f\ t_1 \ldots t_n$ unify

(FailP) $fail\ e \ \to^{lf}\ fail$

(LetIn) $e_1\ e_2 \to^{lf} let\ X = e_2\ in\ e_1\ X$, if e_2 is junk, active, variable application or *let* rooted, for X fresh.

(Bind) $let\ X = t\ in\ e \ \to^{lf}\ e[X/t]$

(Elim) $let\ X = e_1\ in\ e_2 \to^{lf} e_2$, if $X \notin fv(e_2)$

(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \ \to^{lf}\ let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, if $Y \notin fv(e_3)$

(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \to^{lf} let\ X = e_1\ in\ e_2\ e_3$, if $X \notin fv(e_3)$

(Contx) $\mathcal{C}[e] \to^{lf} \mathcal{C}[e']$, if $\mathcal{C} \neq [\,]$, $e \to^{lf} e'$ using any of the previous rules

Fig. 2. Higher order *let*-rewriting relation with pattern matching failure $\to^{lf}$

substitution π is a finite mapping from type variables to simple types $[\overline{\alpha_n/\tau_n}]$. Application of type substitutions to simple types is defined in the natural way and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions in the obvious way. We say σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. A simple type τ' is a *generic instance* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ \tau'$, if $\tau' = \tau[\overline{\alpha_n/\tau_n}]$ for some $\overline{\tau_n}$. Finally, τ' is a *variant* of $\sigma = \forall\overline{\alpha_n}.\tau$, written $\sigma \succ_{var} \tau'$, if $\tau' = \tau[\alpha_n/\beta_n]$ and $\overline{\beta_n}$ are fresh type variables.

3 Formal setup

3.1 Semantics

The operational semantics of our programs is based on *let*-rewriting [18], a high level notion of reduction step devised to express call-time choice. For this paper, we have extended *let*-rewriting with two rules for managing failure of pattern matching (Fig. 2), playing a role similar to the rules for pattern matching failures

in GADTs [3,24]. We write $\rightarrow^{lf}$ for the extended relation and $\mathcal{P} \vdash e \rightarrow^{lf} e'$ ($\mathcal{P} \vdash e \twoheadrightarrow^{lf} e'$ resp.) to express one step (zero or more steps resp.) of $\rightarrow^{lf}$ using the program $\mathcal{P}$. By $nf_{\mathcal{P}}(e)$ we denote the set of *normal forms* reachable from e, i.e., $nf_{\mathcal{P}}(e) = \{e' \mid \mathcal{P} \vdash e \twoheadrightarrow^{lf} e' \text{and } e' \text{ is not } \rightarrow^{lf}\text{-reducible}\}$.

The new rule (Ffail) generates a failure when no program rule can be used to reduce a function application. Notice the use of unification instead of simple pattern matching to check that the variables of the expression will not be able to match the patterns in the rule. This allows us to perform this failure test locally without having to consider the possible bindings for the free variables in the expression caused by the surrounding context. Otherwise, these should be checked in an additional condition for (Contx). Consider for instance the program $\mathcal{P}_1 = \{true \wedge X \rightarrow X, false \wedge X \rightarrow false\}$ and the expression $let\ Y = true\ in\ (Y \wedge true)$. The application $Y \wedge true$ unifies with the function rule left-hand side $true \wedge X$, so no failure is generated. If we use pattern matching as condition, a failure is incorrectly generated since neither $true \wedge X$ nor $false \wedge X$ match with $Y \wedge true$.

Finally, rule (FailP) is used to propagate the pattern matching failure when *fail* is applied to another expression.

Notice that with the new rules (Ffail) and (FailP) there are still some expressions whose evaluation can get stuck, as happens with *junk expressions* like $true\ z$. As we will see in Sect. 4, this can only happen to ill-typed expressions. We will further discuss there the issues of *fail*-ended and stuck reductions.

3.2 Type derivation and inference for expressions

Both derivation and inference rules are based on those presented in [17]. Our type derivation rules for expressions (Fig. 3-a) correspond to the well-known variation of Damas-Milner's [5] type system with syntax-directed rules, so there is nothing essentially new here—the novelty will come from the notion of well-typed program. $Gen(\tau, \mathcal{A})$ is the closure or generalization of τ wrt. $\mathcal{A}$, which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall \overline{\alpha_n}.\tau\ where\ \{\overline{\alpha_n}\} = ftv(\tau) \setminus ftv(\mathcal{A})$. We say that e is well-typed under $\mathcal{A}$, written $wt_{\mathcal{A}}(e)$, if there exists some τ such that $\mathcal{A} \vdash e : \tau$; otherwise it is ill-typed.

The type inference algorithm $\Vdash$ (Fig. 3-b) follows the same ideas as the algorithm $\mathcal{W}$ [5]. We have given the type inference a relational style to show the similarities with the typing rules. Nevertheless, the inference rules represent an algorithm that fails if no rule can be applied. This algorithm accepts a set of assumptions $\mathcal{A}$ and an expression e, and returns a simple type τ and a type substitution π. Intuitively, τ is the "most general" type which can be given to e, and π is the "most general" substitution we have to apply to $\mathcal{A}$ for deriving any type for e.

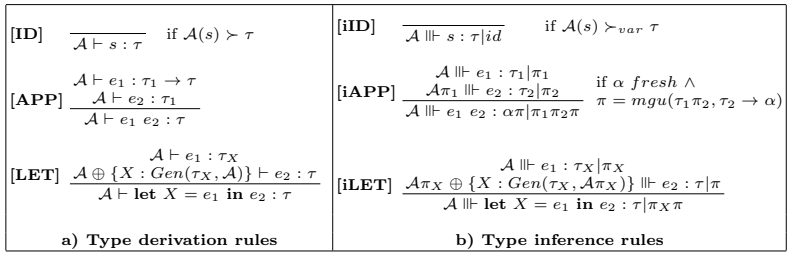


Fig. 3. Type system

3.3 Well-typed programs

The next definition—the most important in the paper—establishes the conditions that a program must fulfil to be well-typed in our proposal:

Definition 1 (Well-typed program wrt. $\mathcal{A}$). *The program rule $f\ t_1 \ldots t_m \to e$ is* well-typed *wrt. a set of assumptions $\mathcal{A}$, written $wt_{\mathcal{A}}(f\ t_1 \ldots t_m \to e)$, iff:*

i) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ t_1 \ldots t_m : \tau_L | \pi_L$
ii) $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R | \pi_R$
iii) $\exists \pi . (\tau_L, \overline{\alpha_n \pi_L}) = (\tau_R, \overline{\beta_n \pi_R})\pi$
iv) $\mathcal{A}\pi_L = \mathcal{A}$, $\mathcal{A}\pi_R = \mathcal{A}$, $\mathcal{A}\pi = \mathcal{A}$

where $\{\overline{X_n}\} = var(f\ t_1 \ldots t_m)$ and $\{\overline{\alpha_n}\}, \{\overline{\beta_n}\}$ are fresh type variables. A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$, written $wt_{\mathcal{A}}(\mathcal{P})$, iff all its rules are well-typed.

The first two points check that both right and left hand sides of the rule can have a valid type assigning *some* types for the variables. Furthermore, it obtains the most general types for those variables in both sides. The third point is the most important. It checks that the obtained most general types for the right-hand side and the variables appearing in it are more general than the ones for the left-hand side. This fact guarantees the *type preservation* property (i.e., the expression resulting after a reduction step has the same type as the original one) when applying a program rule. Moreover, this point ensures a correct management of both *skolem* constructors [14] and *opaque variables* [17], either introduced by the presence of existentially quantified constructors or higher order patterns. Finally, the last point guarantees that the set of assumptions is not modified by neither the type inference nor the matching substitution. In practice, this point holds trivially if type assumptions for program functions are closed, as it is usual.

The previous definition presents some similarities with the notion of *typeable* rewrite rule for Curryfied Term Rewriting Systems in [2]. In that paper the key condition is that the *principal type* for the left-hand side allows to derive the same type for the right-hand side. Besides, [2] considers intersection types and it does not provide an effective procedure to check well-typedness.

Example 1 (Well and ill-typed rules and expressions). Let us consider the following assumptions and program:

$\mathcal{A} \equiv \{\ \mathbf{z} : nat,\ \mathbf{s} : nat \to nat, \mathbf{true} : bool, \mathbf{false} : bool, \mathbf{cons} : \forall\alpha.\alpha \to [\alpha] \to [\alpha],$
$\mathbf{nil} : \forall\alpha.[\alpha],\ \mathbf{rnat} : repr\ nat,\ \mathbf{id} : \forall\alpha.\alpha \to \alpha,\ \mathbf{snd} : \forall\alpha,\beta.\alpha \to \beta \to \beta,$
$\mathbf{unpack} : \forall\alpha,\beta.(\alpha \to \alpha) \to \beta, \mathbf{eq} : \forall\alpha.\alpha \to \alpha \to bool, \mathbf{showNat} : nat \to [char],$
$\mathbf{show} : \forall\alpha.repr\ \alpha \to \alpha \to [char],\ \mathbf{f} : \forall\alpha.bool \to \alpha,\ \mathbf{flist} : \forall\alpha.[\alpha] \to \alpha\ \}$

$\mathcal{P} \equiv \{\ id\ X \to X,\ snd\ X\ Y \to Y,\ unpack\ (snd\ X) \to X,\ eq\ (s\ X)\ z \to false,$
$show\ rnat\ X \to showNat\ X,\ f\ true \to z,\ f\ true \to false,$
$flist\ (cons\ z\ nil) \to s\ z,\ flist\ (cons\ true\ nil) \to false\ \}$

The rules for the functions *id* and *snd* are well-typed. The function *unpack* is taken from [8] as a typical example of the type problems that HO-patterns can produce. According to Def. 1 the rule of *unpack* is not well-typed since the tuple $(\tau_L, \overline{\alpha_n \pi_L})$ inferred for the left-hand side is (γ, δ), which is not matched by the tuple (η, η) inferred as $(\tau_R, \overline{\beta_n \pi_R})$ for the right-hand side. This shows the problem of existential type variables that "escape" from the scope. If that rule was well-typed then type preservation could not be granted anymore—e.g. consider the step $unpack\ (snd\ true) \to^{lf} true$, where the type *nat* can be assigned to $unpack\ (snd\ true)$ but *true* can only have type *bool*. The rule for *eq* is well-typed because the tuple inferred for the right-hand side, $(bool, \gamma)$, matches the one inferred for the left-hand side, $(bool, nat)$. In the rule for *show* the inference obtains $([char], nat)$ for both sides of the rule, so it is well-typed.

The functions *f* and *flist* show that our type system cannot be forced to accept an arbitrary function definition by generalizing its type assumption. For instance, the first rule for *f* is not well-typed since the type *nat* inferred for the right-hand side does not match γ, the type inferred for the left-hand side. The second rule for *f* is also ill-typed for a similar reason. If these rules were well-typed, type preservation would not hold: consider the step $f\ true \to^{lf} z$; $f\ true$ can have any type, in particular *bool*, but *z* can only have type *nat*. Concerning *flist*, its type assumption cannot be made more general for its first argument: it can be seen that there is no τ such that the rules for *flist* remain well-typed under the assumption $flist : \forall\alpha.\alpha \to \tau$.

With the previous assumptions, expressions like $id\ z\ true$ or $snd\ z\ z\ true$ that lead to *junk* are ill-typed, since the symbols *id* and *snd* are applied to more expressions than the arity of their types. Notice also that although our type system accepts more expressions that may produce pattern matching failures than classical Damas-Milner, it still rejects some expressions presenting those situations. Examples of this are $flist\ z$ and $eq\ z\ true$, which are ill-typed since the type of the function prevents the existence of program rules that can be used to rewrite these expressions: *flist* can only have rules treating lists as argument and *eq* can only have rules handling both arguments of the same type.

Def. 1 is based on the notion of type inference of expressions to stress the fact that it can be implemented easily. For each program rule, conditions *i*) and *ii*) use the algorithm of type inference for expressions, *iii*) is just matching, and

iv) holds trivially in practice, as we have noticed before. A more declarative alternative to Def. 1 based on type derivations can be found in [16].

We encourage the reader to play with the implementation, made available as a web interface at `http://gpd.sip.ucm.es/LiberalTyping`.

In [17] we extended Damas-Milner types with some extra control over HO-patterns, leading to another definition of well-typed programs (we write $wt^{old}_{\mathcal{A}}(\mathcal{P})$ for that). All valid programs in [17] are still valid:

Theorem 1. *If* $wt^{old}_{\mathcal{A}}(\mathcal{P})$ *then* $wt_{\mathcal{A}}(\mathcal{P})$.

To further appreciate the usefulness of the new notion with respect the old one, notice that all the examples in Sect. 5 are rejected as ill-typed by [17].

4 Properties of the type system

We will follow two alternative approaches for proving type soundness of our system. First, we prove the theorems of *progress* and *type preservation* similar to those that play the main role in the type soundness proof for GADTs [3,24]. After that, we follow a syntactic approach similar to [28].

Theorem 2 (Progress). *If* $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ *and* e *is ground, then either* e *is a pattern or* $\exists e'.\ \mathcal{P} \vdash e \rightarrow^{lf} e'$.

The *type preservation* result states that in well-typed programs reduction does not change types.

Theorem 3 (Type Preservation). *If* $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ *and* $\mathcal{P} \vdash e \rightarrow^{lf} e'$, *then* $\mathcal{A} \vdash e' : \tau$.

In order to follow a syntactic approach similar to [28] we need to define some properties about expressions:

Definition 2. *An expression* e *is* **stuck** *wrt. a program* $\mathcal{P}$ *if it is a normal form but not a pattern, and is* **faulty** *if it contains a* junk *subexpression.*

Faulty is a pure syntactic property that tries to overapproximate *stuck*. Not all faulty expressions are stuck. For example, $snd\ (z\ z)\ true \rightarrow^{lf} true$. However all faulty expressions are ill-typed:

Lemma 1 (Faulty Expressions are ill-typed). *If* e *is faulty then there is no* $\mathcal{A}$ *such that* $wt_{\mathcal{A}}(e)$.

The next theorem states that all finished reductions of well-typed ground expressions do not get stuck but end up in patterns of the same type as the original expression.

Theorem 4 (Syntactic Soundness). *If* $wt_{\mathcal{A}}(\mathcal{P})$, e *is ground and* $\mathcal{A} \vdash e : \tau$ *then: for all* $e' \in nf_{\mathcal{P}}(e)$, e' *is a pattern and* $\mathcal{A} \vdash e' : \tau$.

The following complementary result states that the evaluation of well-typed expressions does not pass through any faulty expression.

Theorem 5. *If $wt_{\mathcal{A}}(\mathcal{P})$, $wt_{\mathcal{A}}(e)$ and e is ground, then there is no e' such that $\mathcal{P} \vdash e \twoheadrightarrow^{lf} e'$ and e' is $faulty$.*

We discuss now the strength of our results.

• **Progress and type preservation** In [22] Milner considered *'a value 'wrong', which corresponds to the detection of a failure at run-time'* to reach his famous lemma *'well-typed programs don't go wrong'*. For this to be true in languages with patterns, like Haskell or ours, not all run-time failures should be seen as wrong, as happens with definitions like *head (cons x xs)* $\rightarrow x$, where there is no rule for *(head nil)*. Otherwise, progress does not hold and some well-typed expressions become stuck. A solution is considering a 'well-typed completion' of the program, adding a rule like *head nil* $\rightarrow$ *error* where *error* is a value accepting any type. With it, *(head nil)* reduces to *error* and is not wrong, but *(head true)*, which is ill-typed, is wrong and its reduction gets stuck. In our setting, completing definitions would be more complex because of HO-patterns that could lead to an infinite number of 'missing' cases. Our *failure* rules in Sect. 2 try to play a similar role. We prefer the word *fail* instead of *error* because, in contrast to FP systems where an attempt to evaluate *(head nil)* results in a run-time error, in FLP systems rather than an error this is a silent failure in a possible space of non-deterministic computations managed by backtracking. Admittedly, in our system the difference between 'wrong' and 'fail' is weaker from the point of view of reduction. Certainly, junk expressions are stuck but, for instance, *(head nil)* and *(head true)* both reduce to *fail*, instead of the ill-typed *(head true)* getting stuck. Since *fail* accepts all types, this might seem a point where ill-typedness comes in hiddenly and then magically disappear by the effect of reduction to *fail*. This cannot happen, however, because *type preservation* holds step-by-step, and then no reduction $e \rightarrow^* fail$ starting with a well-typed e can pass through the ill-typed *(head true)* as intermediate (sub)-expression.

• **Liberality:** In our system the risk of accepting as well-typed some expressions that one might prefer to reject at compile time is higher than in more restrictive languages. Consider the function *size* of Sect. 1. For any well-typed e, *size* e is also well-typed, even if e's type is not considered in the definition of *size*; for instance, *size (true,false)* is a well-typed expression reducing to *fail*. This is consistent with the liberality of our system, since the definition of *size* could perfectly have included a rule for computing sizes of pairs. Hence, for our system, this is a pattern matching failure similar to the case of *(head nil)*. This can be appreciated as a weakness, and is further discussed in Sect. 6 in connection to type classes and GADT's.

• **Syntactic soundness and faulty expressions:** Th. 4 and 5 are easy consequences of progress and type preservation. Th. 5 is indeed a weaker safety criterion, because our faulty expressions only capture the presence of junk, which by no means is the only source of ill-typedness. For instance, the expressions *(head true)* or *(eq true z)* are ill-typed but not faulty. Th. 5 says nothing about them; it is type preservation who ensures that those expressions will not occur

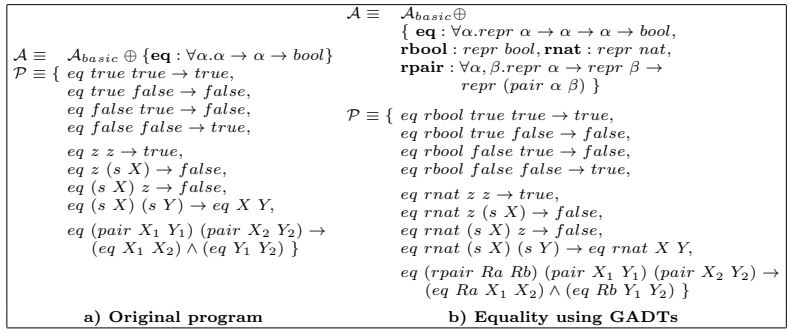


Fig. 4. Type-indexed equality

in any reduction starting in a well-typed expression. Still, Th. 5 contains no trivial information. Although checking the presence of junk is trivial (counting arguments suffices for it), the fact that a given expression will not become faulty during reduction is a typically undecidable property approximated by our type system. For example, consider g with type $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta$, defined as $g\ H\ X \to H\ X$. The expression *(g true false)* is not faulty but reduces to the faulty *(true false)*. Our type system avoids that because the non-faulty expression *(g true false)* is detected as ill-typed.

5 Examples

In this section we present some examples showing the flexibility achieved by our type system. They are written in two parts: a set of assumptions $\mathcal{A}$ over constructors and functions and a set of program rules $\mathcal{P}$. In the examples we consider the following initial set of assumptions:

$\mathcal{A}_{basic} \equiv \{\mathbf{true}, \mathbf{false} : bool,\ \mathbf{z} : nat,\ \mathbf{s} : nat \to nat,\ \mathbf{cons} : \forall\alpha.\alpha \to [\alpha] \to [\alpha],$
$\mathbf{nil} : \forall\alpha.[\alpha],\ \mathbf{pair} : \forall\alpha,\beta.\alpha \to \beta \to pair\ \alpha\ \beta,\ \mathbf{key} : \forall\alpha.\alpha \to (\alpha \to nat) \to key,$
$\wedge, \vee : bool \to bool \to bool,\ \mathbf{snd} : \forall\alpha,\beta.\alpha \to \beta \to \beta, \}$

5.1 Type-indexed functions

Type-indexed functions (in the sense appeared in [12]) are functions that have a particular definition for each type in a certain family. The function *size* of Sect. 1 is an example of such a function. A similar example is given in Fig. 4-a, containing the code for an equality function which only operates with booleans, natural numbers and pairs.

An interesting point is that we do not need a type representation as an extra argument of this function as we would need in a system using GADTs [3,12]. In

these systems the pattern matching on the GADT induces a type refinement, allowing the rule to have a more specific type than the type of the function. In our case this flexibility resides in the notion of well-typed rule. Then a type representation is not necessary because the arguments of each rule of *eq* already force the type of the left-hand side and its variables to be more specific (or the same) than the inferred type for the right-hand side. The absence of type representations provides simplicity to rules and programs, since extra arguments imply that all functions using *eq* direct or indirectly must be extended to accept and pass these type representations. In contrast, our rules for *eq* (extended to cover all constructed types) are the standard rules defining strict equality that one can find in FLP papers (see e.g. [9]), but that cannot be written directly in existing systems like TOY or Curry, because they are ill-typed according to Damas-Milner types.

We stress also the fact that the program of Fig. 4-a would be rejected by systems supporting GADTs [3,25], while the encoding of equality using GADTs as type representations in Fig. 4-b is also accepted by our type system.

Another interesting point is that we can handle equality in a quite fine way, much more flexible than in TOY or Curry, where equality is a *built-in* that proceeds structurally as in Fig. 4-a. With our proposed type system programmers can define structural equality as in Fig. 4-a for some types, choose another behavior for others, and omitting the rules for the cases they do not want to handle. Moreover, the type system protects against unsafe definitions, as we explain now: it is known [8] that in the presence of HO-patterns[2] structural equality can lead to the problem of *opaque decomposition*. For example, consider the expression $eq\ (snd\ z)\ (snd\ true)$. It is well-typed, but after a decomposition step using the structural equality we obtain $eq\ z\ true$, which is ill-typed. Different solutions have been proposed [8], but all of them need fully type-annotated expressions at run time, which penalizes efficiency. With the proposed type system that overloading at run time is not necessary since this problem of opaque decomposition is handled statically at compile time: we simply cannot write equality rules leading to opaque decomposition, because they are rejected by the type system. This happens with the rule $eq\ (snd\ X)\ (snd\ Y) \to eq\ X\ Y$, which will produce the previous problematic step. It is rejected because the inferred type for the right-hand side and its variables X and Y is $(bool, \gamma, \gamma)$, which is more specific than the inferred in the left-hand side $(bool, \alpha, \beta)$.

5.2 Existential types, opacity and HO patterns

Existential types [14] appear when type variables in the type of a constructor do not occur in the final type. For example the constructor $key : \forall\alpha.\alpha \to (\alpha \to nat) \to key$ has an existential type, since α does not appear in the final type *key*. In functional logic languages, however, HO-patterns can introduce the same opacity as constructors with existential type. A prototypical example is $snd\ X$:

[2] This situation also appears with first order patterns containing data constructors with existential types.

we know that X has some type, but we cannot know anything about it from the type $\beta \to \beta$ of the expression. In [17] a type system managing the opacity of HO-patterns is proposed. The program below shows how the system presented here generalizes [17], accepting functions that were rejected there (e.g. $idSnd$) and also supporting constructors with existential type (e.g. $getKey$):

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{getKey} : key \to nat,\ \mathbf{idSnd} : \forall\alpha,\beta.(\alpha \to \alpha) \to (\beta \to \beta)\ \}$$
$$\mathcal{P} \equiv \{\ getKey\ (key\ X\ F) \to F\ X,\ idSnd\ (snd\ X) \to snd\ X\ \}$$

Another remarkable example is given by the well-known translation of higher-order programs to first-order programs often used as a stage of the compilation of functional logic programs (see e.g. [18,1]). In short, this translation introduces a new function symbol @ ('*apply*'), adds calls to @ in some points in the program and appropriate rules for evaluating it. This latter aspect is interesting here, since the rules are not Damas-Milner typeable. The following program contains the @-rules (written in infix notation) for a concrete example with the constructors z, s, nil, $cons$ and the functions $length$, $append$ and snd with the usual types.

$$\mathcal{A} \equiv \mathcal{A}_{basic} \oplus \{\ \mathbf{length} : \forall\alpha.[\alpha] \to nat, \mathbf{append} : \forall\alpha.[\alpha] \to [\alpha] \to [\alpha],$$
$$\mathbf{add} : nat \to nat \to nat, @ : \forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta\ \}$$
$$\mathcal{P} \equiv \{\ s\ @\ X \to s\ X, cons\ @\ X \to cons\ X, (cons\ X)\ @\ Y \to cons\ X\ Y,$$
$$append\ @\ X \to append\ X, (append\ X)\ @\ Y \to append\ X\ Y,$$
$$snd\ @\ X \to snd\ X, (snd\ X)\ @\ Y \to snd\ X\ Y, length\ @\ X \to length\ X\ \}$$

These rules use HO-patterns, which is a cause of rejection in most systems. Even if HO patterns were allowed, the rules for @ would be rejected by a Damas-Milner type system, no matter if extended to support existential types or GADTs. However using Def. 3.1 they are all well-typed, provided we declare @ to have the type $@ : \forall\alpha,\beta.(\alpha \to \beta) \to \alpha \to \beta$. Because of all this, the @-introduction stage of the FLP compilation process can be considered as a source to source transformation, instead of a hard-wired step.

5.3 Generic functions

According to a strict view of genericity, the functions *size* and *eq* in Sect. 1 and 5.1 resp. are not truly generic. We have a definition for each type, instead of one 'canonical' definition to be used by each concrete type. However we can achieve this by introducing a 'universal' data type over which we define the function (we develop the idea for *size*), and then use it for concrete types via a conversion function.

This can be done by using GADTs to represent uniformly the applicative structure of expressions (for instance, the *spines* of [12]), by defining *size* over that uniform representations, and then applying it to concrete types via conversion functions. Again, we can also offer a similar but simpler alternative. A uniform representation of constructed data can be achieved with a data type

data univ = c nat [univ] where the first argument of c is for numbering constructors, and the second one is the list of arguments of a constructor application. A universal *size* can be defined as *usize (c _ Xs) → s (sum (map usize Xs))* using some functions of Haskell's prelude. Now, a generic *size* can be defined as $size \to usize \cdot toU$, where toU is a conversion function with declared type $toU : \forall\alpha.\alpha \to univ$

$toU\ true \to c\ z\ []$ $\quad$ $toU\ false \to c\ (s\ z)\ []$
$toU\ z \to c\ (s^2\ z)\ []$ $\quad$ $toU\ (s\ X) \to c\ (s^3\ z)\ [toU\ X]$
$toU\ [] \to c\ (s^4\ z)\ []$ $\quad$ $toU\ (X{:}Xs) \to c\ (s^5\ z)\ [toU\ X, toU\ Xs]$

(s^i abbreviates iterated s's). This toU function uses the specific features of our system. It is interesting also to remark that in our system the truly generic rule $size \to usize \cdot toU$ can coexist with the type-indexed rules for *size* of Sect. 1. This might be useful in practice: one can give specific, more efficient definitions for some concrete types, and a generic default case via toU conversion for other types[3].

Admittedly, the type *univ* has less representation power than the spines of [12], which could be a better option in more complex situations. Nevertheless, notice that the GADT-based encoding of spines is also valid in our system.

6 Discussion

We further discuss here some positive and negative aspects of our type system.
Simplicity. Our well-typedness condition, which adds only one simple check for each program rule to standard Damas-Milner inference, is much easier to integrate in existing FLP systems than, for instance, type classes (see [20] for some known problems for the latter).
Liberality (continued from Sect. 4): we recall the example of *size*, where our system accepts as well-typed *(size e)* for any well-typed e. Type classes impose more control: *size e* is only accepted if e has a type in the class *Sizeable*. There is a burden here: you need a class for each generic function, or at least for each range of types for which a generic function exists; therefore, the number of class instance declarations for a given type can be very high. GADTs are in the middle way. At a first sight, it seems that the types to which *size* can be applied are perfectly controlled because only *representable* types are permitted. The problem, as with classes, comes when considering other functions that are generic but for other ranges of types. Now, there are two options: either you enlarge the family of representable functions, facing up again the possibility of applying *size* to unwanted arguments, or you introduce a new family of representation types, which is a programming overhead, somehow against genericity.
Need of type declarations. In contrast to Damas & Milner system, where principal types exist and can be inferred, our definition of well-typed program

[3] For this to be really practical in FLP systems, where there is not a 'first-fit' policy for pattern matching in case of overlapping rules, a specific syntactic construction for 'default rule' would be needed.

(Def. 1) assumes an explicit type declaration for each function. This happens also with other well-known type features, like polymorphic recursion, arbitrary-rank polymorphism or GADTs [3,25]. Moreover, programmers usually declare the types of functions as a way of documenting programs. Notice also that type inference for functions would be a difficult task since functions, unlike expressions, do not have *principal types*. Consider for instance the rule $not\ true \to false$. All the possible types for the *not* function are $\forall\alpha.\alpha \to \alpha$, $\forall\alpha.\alpha \to bool$ and $bool \to bool$ but none of them is most general.

Loss of parametricity. In [27] one of the most remarkable applications of type systems was developed. The main idea there is to derive "free theorems" about the equivalence of functional expressions by just using the types of some of its constituent functions. These equivalences express different distribution properties, based on Reynold's abstraction theorem there recasted as "the parametricity theorem", which basically exploits the fact that a function cannot inspect the values of argument subexpressions with a polymorphic variable as type. Parametricity was originally developed for the polymorphic λ-calculus, so free theorems have to be weakened with additional conditions in order to accomodate them to practical languages like Haskell, as their original formulations are false in the presence of unbounded recursion, partial functions or impure features like seq [27,13].

With our type system parametricity is lost, because functions are allowed to inspect any argument subexpression, as seen in the *size* function from page 2. This has a limited impact in the FLP setting, since it is known that non-determinism and narrowing—not treated in the present work but standard in FLP systems—not only breaks free theorems but also equational rules for concrete functions that hold for Haskell, like $(filter\ p) \circ (map\ h) \equiv (map\ h) \circ (filter\ (p \circ h))$ [4].

7 Conclusions

Starting from a simple type system, essentially Damas-Milners's one, we have proposed a new notion of well-typed functional logic program that exhibits interesting properties: simplicity; enough expressivity to achieve existential types or GADT-like encodings, and to open new possibilities to genericity; good formal properties (type soundness, protection against unsafe use of HO patterns). Regarding the practical interest of our work, we stress the fact that no existing FLP system supports any of the examples in Sect. 5, in particular the examples of the *equality*—where known problems of *opaque decomposition* [8] can be addressed—and *apply* functions, which play important roles in the FLP setting. Moreover, our work greatly improves our previous results [17] about safe uses of HO patterns. However, considering also the weaknesses discussed in Sect. 6 suggests that a good option in practice could be a partial adoption of our system, not attempting to replace standard type inference, type classes or GADTs, but rather complementing them.

We find suggestive to think of the following future scenario for our system TOY: a typical program will use standard type inference except for some concrete

definitions where it is annotated that our new liberal system is adopted instead. In addition, adding type classes to the languages is highly desirable; then the programmer can choose the feature—ordinary types, classes, GADTs or our more direct generic functions—that best fits his needs of genericity and/or control in each specific situation. We have some preliminary work [21] exploring the use of our type-indexed functions to implement type classes in FLP, with some advantages over the classical dictionary-based technology.

Apart from the implementation work, to realize that vision will require further developments of our present work:

• A precise specification of how to mix different typing conditions in the same program and how to translate type classes into our generic functions.

• Despite of the lack of principal types, some work on type inference can be done, in the spirit of [25].

• Combining our genericity with the existence of modules could require adopting *open* types and functions [15].

• Narrowing, which poses specific problems to types, should be also considered.

Acknowledgments We thank Philip Wadler and the rest of reviewers for their stimulating criticisms and comments.

References

1. Antoy, S., Tolmach, A.P.: Typed higher-order narrowing without higher-order strategies. In: Proc. FLOPS'99, Springer LNCS 1722, pp. 335–353, 1999.
2. van Bakel, S., Fernández, M.: Normalization Results for Typeable Rewrite Systems. Information and Computation 133(2), pp. 73–116, 1997.
3. Cheney, J., Hinze, R.: First-class phantom types. Tech. Rep. TR2003-1901, Cornell University, 2003.
4. Christiansen, J., Seidel, D., Voigtländer, J.: Free theorems for functional logic programs. In: Proc. PLPV '10, pp. 39–48. ACM, 2010.
5. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proc. POPL'82, pp. 207–212. ACM, 1982.
6. González-Moreno, J.C., Hortalá-González, T., López-Fraguas, F., Rodríguez-Artalejo, M.: An approach to declarative programming based on a rewriting logic. Journal of Logic Programming 40(1), pp. 47–87, 1999.
7. González-Moreno, J., Hortalá-González, M., Rodríguez-Artalejo, M.: A higher order rewriting logic for functional logic programming. In: Proc. ICLP'97, pp. 153–167. MIT Press, 1997.
8. Gonzalez-Moreno, J.C., Hortala-Gonzalez, M.T., Rodriguez-Artalejo, M.: Polymorphic types in functional logic programming. Journal of Functional and Logic Programming 2001(1), 2001.
9. Hanus, M.: Multi-paradigm declarative languages. In: Proc. ICLP'07, Springer LNCS 4670, pp. 45–75, 2007.
10. Hanus (ed.), M.: Curry: An integrated functional logic language (version 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2006.
11. Hinze, R.: Generics for the masses. J. Funct. Program. 16(4-5), pp. 451–483, 2006.
12. Hinze, R., Löh, A.: Generic programming, now!. In: Revised Lectures SSDGP'06, Springer LNCS 4719, pp. 150–208, 2007.

13. Hudak, P., Hughes, J., Jones, S.P., Wadler, P.: A History of Haskell: being lazy with class. In: Proc. HOPL III, pp. 12-1–12-55. ACM, 2007.
14. Läufer, K., Odersky, M.: Polymorphic type inference and abstract data types. ACM Transactions on Programming Languages and Systems 16. ACM, 1994.
15. Löh, A., Hinze, R.: Open data types and open functions. In: Proc. PPDP '06, pp. 133–144. ACM, 2006.
16. López-Fraguas, F.J., Martin-Martin, E., Rodríguez-Hortalá, J.: Liberal Typing for Functional Logic Programs (long version). Tech. Rep. SIC-UCM, Universidad Complutense de Madrid (August 2010), available at `http://gpd.sip.ucm.es/enrique/publications/liberalTypingFLP/long.pdf`
17. López-Fraguas, F.J., Martin-Martin, E., Rodríguez-Hortalá, J.: New results on type systems for functional logic programming. In: WFLP'09 Revised Selected Papers, Springer LNCS 5979, pp. 128–144, 2010.
18. López-Fraguas, F., Rodríguez-Hortalá, J., Sánchez-Hernández, J.: Rewriting and call-time choice: the HO case. In: Proc. FLOPS'08, Springer LNCS 4989, pp. 147–162, 2008.
19. López-Fraguas, F., Sánchez-Hernández, J.: $\mathcal{TOY}$: A multiparadigm declarative system. In: Proc. RTA'99, Springer LNCS 1631, pp. 244–247, 1999.
20. Lux, W.: Adding haskell-style overloading to curry. In: Workshop of Working Group 2.1.4 of the German Computing Science Association GI. pp. 67–76, 2008.
21. Martin-Martin, E.: Implementing type classes using type-indexed functions. To appear in TPF'10, available at `http://gpd.sip.ucm.es/enrique/publications/implementingTypeClasses/implementingTypeClasses.pdf`.
22. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348–375 (1978)
23. Moreno-Navarro, J.J., Mariño, J., del Pozo-Pietro, A., Herranz-Nieva, Á., García-Martín, J.: Adding type classes to functional-logic languages. In: Proc. APPIA-GULP-PRODE'96. pp. 427–438, 1996.
24. Peyton Jones, S., Vytiniotis, D., Weirich, S.: Simple unification-based type inference for GADTs. Tech. Rep. MS-CIS-05-22, Univ. Pennsylvania, 2006.
25. Schrijvers, T., Peyton Jones, S., Sulzmann, M., Vytiniotis, D.: Complete and decidable type inference for GADTs. In: Proc. ICFP '09, pp. 341–352. ACM, 2009.
26. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proc. POPL'89, pp. 60–76. ACM, 1989.
27. Wadler, P.: Theorems for free! In: Proc. FPCA'89, pp. 347–359. ACM, 1989.
28. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Information and Computation 115, pp. 38–94, 1992.

Type Classes in Functional Logic Programming

Enrique Martin-Martin
Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain
emartinm@fdi.ucm.es

Abstract

Type classes provide a clean, modular and elegant way of writing overloaded functions. Functional logic programming languages (FLP in short) like Toy or Curry have adopted the Damas-Milner type system, so it seems natural to adopt also type classes in FLP. However, type classes has been barely introduced in FLP. A reason for this lack of success is that the usual translation of type classes using *dictionaries* presents some problems in FLP like the absence of expected answers due to a bad interaction of dictionaries with the call-time choice semantics for non-determinism adopted in FLP systems.

In this paper we present a *type-passing* translation of type classes based on *type-indexed functions* and *type witnesses* that is well-typed with respect to a new liberal type system recently proposed for FLP. We argue the suitability of this translation for FLP because it improves the dictionary-based one in three aspects. First, it obtains programs which run as fast or faster—with an speedup from 1.05 to 2.30 in our experiments. Second, it solves the mentioned problem of missing answers. Finally, the proposed translation generates shorter and simpler programs.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism; D.3.2 [*Language Classifications*]: Multiparadigm languages

General Terms Languages, Design, Performance.

Keywords Type Classes, Functional Logic Programming, Type-indexed functions.

1. Introduction

Type classes [10, 30] are one of the most successful features in Haskell. They provide an easy syntax to define overloaded functions—*classes*—and the implementation of those functions for different types—*instances*. Type classes are usually implemented by means of a source-to-source transformation that introduces extra parameters—called *dictionaries*—to overloaded functions [10, 30], generating Damas-Milner [7] correct programs. Dictionaries are data structures containing the implementation of overloaded functions for specific types and dictionaries for the superclasses. The efficiency of translated programs—using several optimizations [4, 11]—and the fact that the translation handles correctly multiple modules and separate compilation, have resulted in that nowadays it is the most used technique for implementing type classes in functional programming (FP). Another scheme for translating type classes is passing type information as extra arguments to overloaded functions [29]. In this scheme, overloaded functions use a *typecase* construction in order to pattern-match types and decide which concrete behavior—*instance*—to use. Although it is possible to encode it using *generalized algebraic data types* (GADTs) [6, 14] or Guarded Recursive Datatype Constructors [31], this translation scheme has not succeeded in the FP community.

Functional logic programming (FLP) [12] aims to combine the best of declarative paradigms (functional, logic and constraint languages) in a single model. FLP languages like Toy [22] or Curry [13] have a strong resemblance to lazy functional languages like Haskell [15]. However, a remarkable difference is that functional logic programs can be non-confluent, giving raise to so-called *non-deterministic functions*, for which a *call-time choice* semantics [8] is adopted. The following program is a simple example, using Peano natural numbers given by the constructors *z* and *s*[1]: *coin* → *z*, *coin* → *s z*, *dup X* → *pair X X*—where *pair* is the constructor symbol for pairs. Here, *coin* is a non-deterministic function (*coin* evaluates to *z* and *s z*) and, according to call-time choice, *dup coin* evaluates to *pair z z* and *pair (s z) (s z)* but not to *pair z (s z)* or *pair (s z) z*. Operationally, call-time choice means that all copies of a non-deterministic subexpression (*coin* in the example) created during reduction share the same value.

Functional logic languages have adopted the Damas-Milner type system, although it presents some problems when applied directly [9, 21]. However, with the exception of some preliminary proposals as [26]—presenting some ideas about type classes and FLP not further developed—and [23]—showing some problems that the dictionary approach produces when applied to FLP systems—type classes have not been incorporated in FLP. From the point of view of the systems, only an experimental branch of [1] and the experimental systems [2, 3] have tried to adopt type classes. One reason for this limited success is the problems presented in [23]. In addition to them, another important issue to address is the lack of expected answers when combining non-determinism and *nullary*[2] overloaded functions [24]. This problem is shown in the program in Fig. 1, taken from [24]. We use a syntax of type classes and instances similar to Haskell but following the mentioned syntactic convention adopted in the Toy system. The program contains an overloaded function `arb` which is a non-deterministic generator, and its instance for booleans. It also contains a function `arbL2` which returns a list of two elements of the same instance of `arb`. Fig. 1-b) contains the translated program following the standard translation using dictionaries [10, 30]. The `arb` type class generates a data declaration for `arb` dictionaries—`dictArb`—and a projecting function `arb` to extract the concrete implementation from the dictionary. The instance `arb bool` gen-

http://doi.acm.org/10.1145/1929501.1929524.

[1] We follow the syntactic conventions of Toy where identifiers are lowercased and variables are uppercased.

[2] i.e. of arity 0.

```
class arb A where
  arb :: A

instance arb bool where
  arb → false
  arb → true

arbL2 :: arb A => list A
arbL2 → [arb, arb]
```

a) Original program

```
data dictArb A = dictArb A

arb :: dictArb A -> A
arb (dictArb F) → F

arbBool :: bool
arbBool → false
arbBool → true

dictArbBool :: dictArb bool
dictArbBool → dictArb arbBool

arbL2 :: dictArb A -> list A
arbL2 DA → [arb DA, arb DA]
```

b) Translated program using dictionaries

Figure 1. Program containing a type class with a constant non-deterministic overloaded function

erates a concrete dictionary—`dictArbBool`—and the `arbL2` function is transformed to accept an `arb` dictionary as first argument and pass it to the `arb` functions in its right-hand side. Expected results for the expression `arbL2::(list bool)` are `[true, true]`, `[true, false]`, `[false, true]` and `[false, false]`, however its evaluation in the translated program only produces `[true, true]` and `[false, false]`. The reason is the call-time choice semantics. The translated expression `arbL2 dictArbBool` reduces to `[arb dictArbBool, arb dictArbBool]`, but both copies of `dictArbBool` must share their value. Therefore they cannot be reduced to `dictArb true` and `dictArb false` in the different occurrences of the right-hand side, losing two expected solutions.

In this paper we propose and evaluate a type-passing translation of type classes for FLP based on type-indexed functions—functions with a different behavior for different types [14]—and type witnesses—representations of types as data values—that is well-typed in a new liberal type system recently proposed for FLP [20]. The proposed translation is not integrated in the type checking phase as in [10, 30], but it is a separated phase after type checking. This previous type checking phase is assumed to use a standard type system supporting type classes [5, 27], and decorates the function symbols with the inferred types.

We show that the proposed translation is a suitable option for FLP compared to the classical dictionary-based translation because of three reasons. First, it obtains programs which run as fast or faster—with and speedup ranging from 1.05 to 2.30 in our experiments. When we apply optimizations to both translated programs the speedup still remains favorable to the proposed translation. Second, it solves the mentioned problem of missing answers when combining non-determinism and nullary overloaded functions. Finally, the proposed translation has a similar complexity to the dictionary-based one, but generates shorter and simpler programs.

The following list summarizes the main contributions of the paper and at the same time presents the structure of the paper.

- We formalize a type-passing translation for type classes in FLP in Sect. 3. Although the broad idea of using such kind of translation is not a novelty [29], its concrete realization and the application to FLP, relying in a new type system [20], are new. In particular, the liberality of the type system avoids the need of a *typecase* construction in the target language, resulting in that translated programs do not need to enhance the syntax of FLP systems with that construction.
- We have measured the execution time of a collection of different programs involving overloaded functions that can be part of bigger real FLP programs—see Sect 4.1. Some of these programs have been adapted from the *nobench* suite of benchmark programs for Haskell. The speedup results—from 1.05 to 2.30—show that when no optimizations are applied, programs translated using the proposed type-passing scheme perform faster than those translated using the dictionary-based translation.
- There are several well-known optimizations than can be applied to translated programs using the dictionary-based scheme [4, 11]. In Sect. 4.1 we present some optimizations to the proposed type-passing translation. We have repeated the execution time measurements to the optimized programs, and we have checked that the proposed translation still obtains faster programs even when optimizations are applied.
- We study how the proposed translation solves the problem of missing answers that appears when combining non-determinism and nullary overloaded functions—see Sect. 4.2.
- In Sect. 5 we discuss some additional aspects—including some problems—that arise with the translations of type classes in FLP.

Type variable	$\alpha, \beta, \gamma \ldots$	
Type constructor	C	
Class name	κ , $\kappa^\bullet$	
Simple type	τ	$::= \alpha \mid \tau \to \tau'$
		$\mid C\ \overline{\tau_n}$ with n = arity(C), n ≥ 0
Context	θ	$::= \langle \overline{\kappa_n\ \alpha_n} \rangle$ with n ≥ 0
Saturated context	ϕ	$::= \langle \overline{\kappa_n\ \tau_n} \rangle$ with n ≥ 0
Overloaded type	ρ	$::= \phi \Rightarrow \tau$
Type scheme	σ	$::= \forall \overline{\alpha_n}.\tau$ with n ≥ 0

Figure 2. Syntax of types

2. Preliminaries

This section introduces the syntax of types, the source language and the target language of the proposed translation. It also introduces the liberal type system in which the translated programs are well-typed.

2.1 Syntax

Fig. 2 gives the syntax of types, which are the usual ones when using type classes [10]. The only difference is that class names can have a mark $^\bullet$. We use this mark in the translation to distinguish between which class constraints generate a type information to pass to overloaded functions, as we will explain in Sect. 3. Overloaded types are simple types enclosed with a *saturated* context. Notice that in a saturated context class restrictions not only affect type variables but they can affect simple types as $list\ bool$ or $pair\ int\ (list\ nat)$. Contexts, which express class constraints over type variables, will be used in class and instance declarations. Type schemes are the same as in the Damas-Milner type system [7], and play the usual role to handle *parametric* polymorphism.

The syntax of source programs of the translation is shown in Fig. 3. It is the usual syntax for programs with type classes of one argument [10] adapted to Toy's syntax. We assume a denumerable set of data variables (X), and a set of function symbols

function symbol	f	
constructor symbol	c	
data variable	X	
program ::=	$\overline{data}\ \overline{class}\ \overline{inst}\ \overline{type}\ \overline{rule}$	
data ::=	$\texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \ldots \mid \overline{c_k\ \overline{\tau}}$	
class ::=	$\texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \overline{f :: \tau}$	
inst ::=	$\texttt{instance}\ \theta \Rightarrow \kappa\ (C\ \overline{\alpha})\ \texttt{where}$	
	$\quad \overline{f\ \overline{t} \to e}$ with $\overline{t}$ linear	
type ::=	$f :: \theta \Rightarrow \tau$	
rule r ::=	$(f :: \rho)\ \overline{t} \to e$ with $\overline{t}$ linear	
pattern t ::=	$X \mid c\ \overline{t_n}$ with n $\leq$ arity(c)	
	$\mid f\ \overline{t_n}$ with n $<$ arity(f)	
expression e ::=	$X \mid c \mid f :: \rho \mid e\ e \mid let\ X = e\ in\ e$	

Figure 3. Syntax of source programs

(f) and constructor symbols (c), all them with associated arity. We say that a function is a *member* of a type class if it is declared inside that type class declaration, and it is an *overloaded function* if its inferred type has class constraints in the context. Notice that member function are overloaded functions, since they have exactly one class constraint in the context of its type. Patterns—our notion of values—are a subset of expressions. Notice that constructor and function symbols partially applied to patterns—called HO-patterns—are considered as patterns in our setting, the HO Constructor-based conditional ReWriting Logic (HO-CRWL) approach to FLP [25] followed by the Toy system. This corresponds to an intensional view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics. In program rules (r) the set of patterns $\overline{t}$ is linear (there is not repetition of variables) and there are not extra variables in the right-hand side. However we do not support HO-patterns made with overloaded function symbols in the left-hand side of rules, due to some complications that arise during translation—see Sect 5.3. A particularity of the syntax is that function symbols in rules and expressions are always decorated with an overloaded type. We assume that this decoration comes from a previous type checking phase, and reflects to which types are functions applied. In the type checking stage the type checker decorates function symbols with a variant of its type, and instantiate it with the proper type of the application. For example if `eq` has the usual type $\langle eq\ A\rangle \Rightarrow A \to A \to bool$, a rule for a function `g`:

```
g X → eq X [true]
```

will have the decoration

```
g::⟨⟩ ⇒ (list bool) → bool  X →
  eq::⟨eq (list bool)⟩ ⇒ (list bool) → (list bool) → bool
  X [true]
```

In the right-hand side of `g`, the saturated context $\langle eq\ (list\ bool)\rangle$ indicates that the overloaded `eq` function is applied to elements of type $list\ bool$, so it needs that type information. The function `g` in the left hand side does not have any context because its context is *reduced* during type checking—see Sect. 3.3—and became empty, so it does not appear in the inferred type for `g`.

The syntax of target programs is similar to source programs, except that there are not class or instance declarations, function symbols in rules and expressions are not decorated with type information and type declarations for functions are only simple types.

2.2 Liberal type system for FLP

The type system considered for the target language is a new simple extension of the Damas-Milner type system recently proposed for FLP [20]. The typing rules for expressions correspond to the well-known variation of Damas-Milner type system [7] with syntax-directed rules. The type inference algorithm $\Vdash$ follows the same ideas that algorithm $\mathcal{W}$ [7], however we have given the type inference a relational style $\mathcal{A} \Vdash e : \tau|\pi$. This algorithm accepts a set of type scheme assumptions $\mathcal{A}$ over symbols s_i which can be variables or constructor/function symbols—$\{\overline{s_n : \sigma_n}\}$—and an expression e, returning a simple type τ and a type substitution π—$[\overline{\alpha_n/\tau_n}]$. Intuitively, τ is the "most general" type which can be given to e, and π the "most general" substitution we have to apply to $\mathcal{A}$ in order to be able to derive any type for e. The difference is that, unlike FP, we cannot write programs as expressions—we do not have λ-abstractions—so we need an explicit method for checking whether a program is well-typed. We will say that a program is well-typed wrt. a set of assumptions if all the rules are well-typed:

```
size :: A -> nat
size false →s z
size true →s z
size z →s z
size (s X) →s (size X)

eq :: A -> A -> bool
eq true true →true
eq false false →true
eq z z →true
eq (s X) (s Y) →eq X Y
```

Figure 4. Examples of type-indexed functions

DEFINITION 1. *A rule $f\ \overline{t} \to e$ is well-typed wrt. to a set of assumptions $\mathcal{A}$ iff:*

- $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash f\ \overline{t} : \tau_L|\pi_L$
- $\mathcal{A} \oplus \{\overline{X_n : \beta_n}\} \Vdash e : \tau_R|\pi_R$
- $\exists\pi.(\tau_L, \overline{\alpha_n\pi_L}) = (\tau_R, \overline{\beta_n\pi_R})\pi$

where $\overline{X_n}$ are the variables in $\overline{t}$, $\oplus$ is the symbol for the usual union of sets of assumptions and $\overline{\alpha_n}, \overline{\beta_n}$ are fresh type variables.

Intuitively, a rule is well-typed if the types $(\tau_R, \overline{\beta_n\pi_R})$ inferred for the right-hand side and its variables are more general than the types $(\tau_L, \overline{\alpha_n\pi_L})$ inferred for its left-hand side and its variables. Notice that programmers must provide an explicit type for every function symbol, otherwise the first point of the definition fails to infer the type for the expression $f\ \overline{t}$. Therefore Def. 1 cannot be used to infer the types of the functions, but to check that the types provided for the functions are correct.

The most remarkable feature of this new system is its liberality, that allows the programmer to define type-indexed functions in a very easy way, but still assuring essential safety properties like *type preservation* and *progress*—see [20] for more details. Consider the type-indexed functions `size` and `eq` defined over natural and booleans that appear in Fig. 4. The first three rules for `size` are well-typed because the type inferred for the right-hand side (nat) is more general than the inferred in the left-hand side (nat again). In the fourth rule the types inferred for the left-hand side and the variable `X` are both nat, and in the right-hand side the inferred types are nat and β resp., so the rule is well typed since (nat, β) is more general than (nat, nat). The same happens in the fourth rule of `eq`, where $(bool, \beta, \beta)$ inferred for the right-hand side is more general than $(bool, nat, nat)$ inferred for the left-hand side. The rest of rules for `eq` are well-typed for similar reasons.

3. Translation

As we have said in Sect. 1, the translation follows a type-passing scheme [29] and uses type-indexed functions and type witnesses. Instead of passing dictionaries containing the concrete implementation of the overloaded functions to use, in this scheme we pass data values—type witnesses—representing the types to which overloaded functions are applied. In the source program, saturated con-

texts that decorate function symbols show what types are they applied to, so we use that information to generate the concrete type witnesses. Member functions are translated into type-indexed functions that pattern-match on the type witness and decide which instance of the overloaded function to use. Due to the liberality of the type system, these type-indexed functions are encoded with type witnesses without the need of a special *typecase* constructions as in [29], so translated programs are usual FL programs.

3.1 Type witnesses

Type witnesses are data values that represent types. In [6, 14] these *type representations* are encoded using a GADT containing all the type representations. We follow a slightly different approach: we extend every data declaration with a new constructor in order to represent the type of the declared data. For example, a data declaration for Peano naturals `data nat = z | s nat` is extended with the constructor `#nat`, resulting in `data nat = z | s nat | #nat`; and a data declaration for lists `data list A = nil | cons A` is extended to `data list A = nil | cons A | #list A`. This extension of data declarations can be easily performed by the system. An interesting point of type witnesses defined this way is that they have exactly the same type they represent. In the previous example, `#nat` has type nat, and `#list (#list #nat)` has type *list (list A)*. This link between types and type witnesses allows us to generate automatically the type witness of a given simple type, fact that is used during translation.

DEFINITION 2 (Generation of type witnesses).
- *testify*$(\alpha) = X_\alpha$
- *testify*$(C\ \tau_1 \ldots \tau_n)$ = *#C testify*$(\tau_1) \ldots$ *testify*(τ_n)

The function testify returns the same data variable X_α for the same type variable α. Notice that the testify function is not defined for functional types $\tau \to \tau'$. This is because we consider a source language where instances over functional types are not possible, so in the translation we will not need to generate type witnesses for that types. However, in our liberal type system it would be simple to create type witnesses for those types using a special data constructor `#arrow` of type $\alpha \to \beta \to (\alpha \to \beta)$.

3.2 Translation

In the classical dictionary-based scheme [10, 30], the translation is integrated in the type checking phase so that it uses the inferred type information. In this paper we follow a different approach, supposing that the translation from type classes to type-indexed functions comes after a type checking phase that has inferred the types to the whole program [5, 27]. Since the inferred type information is needed for the translation, we assume that the type checking phase has decorated the function symbols with their corresponding types. The idea of the translation is simple: we inspect the context of the types that decorate function symbols and extract from them the concrete type witnesses that we need to pass to the functions. We define a set of translation functions for the different constructions (whole programs, data declarations, classes, instances, type declarations, rules and expressions):

DEFINITION 3 (Translation functions).

$trans_{prog}(\overline{data}\ \overline{class}\ \overline{inst}\ \overline{type}\ \overline{rule}) =$
$\overline{trans_{data}(data)}\ \overline{trans_{class}(class)}\ \overline{trans_{inst}(inst)}$
$\overline{trans_{type}(type)}\ \overline{trans_{rule}(rule)}$

$trans_{data}(\texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \ldots \mid c_k\ \overline{\tau}) =$
$\texttt{data}\ C\ \overline{\alpha} = c_1\ \overline{\tau} \mid \ldots \mid c_k\ \overline{\tau} \mid \#C\ \overline{\alpha}$

$trans_{class}(\texttt{class}\ \theta \Rightarrow \kappa\ \alpha\ \texttt{where}\ \overline{f :: \tau}) = \overline{f :: \alpha \to \tau}$

$trans_{inst}(\texttt{instance}\ \theta \Rightarrow \kappa\ (C\ \overline{\alpha})\ \texttt{where}\ \overline{f\ \overline{t} \to e}) =$
$\overline{f\ testify(C\ \overline{\alpha})\ \overline{trans_{expr}(t)} \to trans_{expr}(e)}$

$trans_{type}(f :: \theta \Rightarrow \tau) = f :: \alpha_1 \to \ldots \to \alpha_n \to \tau$
where $\alpha_1 \ldots \alpha_n$ *appear in* θ *constrained by a class marked with* $^\bullet$

$trans_{rule}((f :: \rho)\ \overline{t} \to e) =$
$trans_{expr}(f :: \rho)\ \overline{trans_{expr}(t)} \to trans_{expr}(e)$

$trans_{expr}(X) = X$
$trans_{expr}(c) = c$
$trans_{expr}(f :: \rho) = f\ testify(\tau_1) \ldots testify(\tau_n)$
where $\rho \equiv \phi \Rightarrow \tau$ *and* $\tau_1 \ldots \tau_n$ *appear in* ϕ *constrained by a class marked with* $^\bullet$
$trans_{expr}(e\ e') = trans_{expr}(e)\ trans_{expr}(e')$
$trans_{expr}(let\ X = e\ in\ e') =$
$let X = trans_{expr}(e)\ in\ trans_{expr}(e')$

The translation of a program is simply the translation of its components. Data declarations are extended with the constructor of its type witness as explained in Sect. 3.1. Class declarations generate type declarations for the type-indexed functions. The generated type is the same as the one declared in the class but it has an extra first argument for the type witness. Consider the class declaration for the class `foo`:

```
class foo A where
  foo :: A → bool
```

This declaration generates a type declaration for the type-indexed function `foo` adding an extra first argument A to the type of the member function. This argument A is the type variable of the type class:

```
foo :: A → A → bool
```

Type declarations are treated in a similar way, with the difference that we only add new arguments to the translated type if they are constrained by a class with a $^\bullet$ mark, i.e., if the corresponding type witnesses are needed. Consider the type declaration for `f`:

```
f :: ⟨eq•  A, ord  A, eq•  B⟩ ⇒ A → B → bool
```

This declaration generates a type declaration with the extra arguments A and B—and in that order—which are the type variables constrained by marked class names in the context:

```
f :: A → B → A → B → bool
```

Rules in an instance declaration are translated one by one. These rules generate the rules of type-indexed functions, so we add a type witness of the concrete instance as the first argument so they dispatch on it. Notice that a rule generated from an instance do not need any extra type-witness, since the type declared in the class declaration is a simple type and does not have a context. Consider the instance declaration `foo` for *list* A:

```
instance foo (list A) where
  foo X → false
```

This declaration generates a rule for the type-indexed function `foo` whose first argument is the type witness (`#list` X_A), the result of the testify function for the type *list* A of the instance declaration:

```
foo (#list X_A) X → false
```

To translate a rule, we translate all its components. Notice that according to our source syntax, patterns $\overline{t}$ do not contain overloaded function symbols, so they are decorated with types with empty contexts $\langle\rangle$. Therefore type witnesses will not be added to patterns, and the translation function $trans_{expr}$ will only erase the type decorations. The most important case of $trans_{expr}$ is the translation of a function symbol. When we have an overloaded function, we have to provide the type witnesses it needs. In this case we inspect the saturated context ϕ, collecting those types constrained by a marked class name and adding their associated type witnesses. The

order in which these type witnesses are supplied is important, and must be the same for all the occurrences of the same overloaded function. Consider a possible occurrence of the previous function `f` applied to concrete types:

`f` $:: \langle eq^{\bullet}\ bool, ord\ bool, eq^{\bullet}\ (list\ int)\rangle \Rightarrow bool \rightarrow (list\ int) \rightarrow bool$

The translation of this decorated function symbol adds type witnesses for booleans and lists of integers, which are the types constrained by marked class names in the context:

```
f #bool (#list #int)
```

Notice that in expressions not containing overloaded functions, the result of the translation is the original expression without type decorations in functions symbols. The same happens with programs no containing overloaded functions. Therefore in these cases the translation does not introduce any overhead in the program.

As the reader can notice, the translation does not need the complete decoration of function symbols but only the types marked with a $^{\bullet}$ in the context. We have decided to use the complete inferred decorations to make more notable the close link between the translation and the type checking phase.

3.3 Important issues for the translation

The type checking phase is very important for this translation, since the information it provides in the contexts of the types that decorates function symbols directs the translation. There are two important issues that the type checker must address: context reduction and the marking of class names in contexts.

Context reduction

When performing the type checking of functions, the type checker infers a type τ and a context of class constraints. Consider the nondeterministic function `f`, where `gt` is the *greater* function with type $\langle ord\ A\rangle \Rightarrow A \rightarrow A \rightarrow bool$ and `eq` the equality function with type $\langle eq\ A\rangle \Rightarrow A \rightarrow A \rightarrow bool$:

```
f (X:Xs) Z → gt X Z
f (X:Xs) Z → and (eq X Z) (eq Xs [Z])
```

For these rules, the inferred type is $(list\ A) \rightarrow A \rightarrow bool$ and the context is $\langle ord\ A, eq\ A, eq\ (list\ A)\rangle$. The constraint $ord\ A$ comes from the order comparison in the first rule `gt X Z`, the constraint $eq\ A$ from the equality comparison between Z and the head of the list X, and the constraint $eq\ (list\ A)$ from the equality comparison `eq Xs [Z]`. However, this context contains some redundant information and could be reduced. There are three rules for context reduction:

- *Eliminating duplicate constraints*. We can reduce the context $\langle eq\ A, eq\ A\rangle$ to $\langle eq\ A\rangle$ and no information is lost.
- *Using instance declarations*. The usual instance declaration for equality on lists is `instance eq A` $\Rightarrow$ `eq (list A) where (...)`, specifying how to use the equality on values A to define an equality on $list\ A$. Therefore, we can reduce the context $\langle eq\ A, eq\ (list\ A)\rangle$ to $\langle eq\ A\rangle$. This reduction is not a problem from the point of view of type witnesses, because given a type witness for A we can generate a type witness for $list\ A$.
- *Using class declarations*. The class declaration for `ord` is `class eq A` $\Rightarrow$ `ord A where (...)`, specifying that any instance of `ord` is also an instance of `eq`. Therefore we can reduce the context $\langle ord\ A, eq\ A\rangle$ to $\langle ord\ A\rangle$. From the point of view of type witnesses this is not a problem, because we still know that we need a type witness of A.

Therefore, the previous context for function `f` would be reduced to $\langle ord\ A\rangle$ using all the previous rules. In [17] they explore different choices about how much context reduction to apply. Haskell's choice is to reduce the context completely before generalization, and this choice is necessary in our translation. Otherwise, the translation could generate rules that violate the restriction of linear left-hand sides. Consider the instance declaration for equality on pairs `instance` $\langle eq\ A, eq\ B\rangle \Rightarrow$ `eq (pair A B) where (...)`, and the rule `g P1 P2` $\rightarrow$ `([fst P1, snd P2], eq P1 P2)`—where `fst` and `snd` project the first and second component of a pair respectively. If we do not use the instance declaration to reduce the context, the type decoration obtained for `g` is $\langle eq^{\bullet}\ (pair\ A\ A)\rangle \Rightarrow (pair\ A\ A) \rightarrow (pair\ A\ A) \rightarrow (pair\ (list\ A)\ bool)$. Then the left-hand side of the translated rule would be `g (#pair` X_A X_A`) P1 P2`. This is not syntactically valid in our target language as the data variable X_A appears twice. Applying two steps of context reduction using the instance and eliminating duplicates we obtain $\langle eq\ A\rangle$. With this new context the left-hand side of the translated rule is `g` X_A `P1 P2`, which now is valid in the target language.

Marking of class names

We have used marked class names in contexts to know which type witness to pass to functions. The task of marking class names is an easy task that must be done after type checking, when the types of all the functions are inferred. At this point, contexts will have only constraints on type variables due to context reduction. There can be more than one class constraint over the same type variable, however we do not want to pass duplicate type witnesses for the same type. That is the reason why we mark with a $^{\bullet}$ only one constraint per type variable, defining the order in which type witnesses must be passed. Consider a Fibonacci function that accepts any numeric argument and returns an integer:

```
fib N = if N<2 then 1 else fib (N-1) + fib (N-2)
```

Its inferred type is $\langle num\ A, ord\ A\rangle \Rightarrow A \rightarrow int$. However, we do not need to pass two identical type witnesses to the rule. Therefore we mark one of the constraints over A, obtaining the type $\langle num^{\bullet}\ A, ord\ A\rangle \Rightarrow A \rightarrow int$. Then in every call of the `fib` function we will only pass one type witness. Moreover, if we do not use the $^{\bullet}$ marks the left-hand side of the `fib` rule would be translated into `fib` X_A X_A `N`, with two occurrences of the data variable X_A, violating the syntactic constraint that patterns in a left-hand side of a rule are linear.

3.4 Case study: equality and order

Fig. 5 contains the translation of a complete program using equality and order. Fig. 5-a) shows the source program with type declarations in the function symbols. These decorations are introduced by the type checker so the user does not need to write them in the source program. We suppose that usual booleans functions `and`, `or`$::\langle\rangle \Rightarrow bool \rightarrow bool \rightarrow bool$ and the conditional function `ifthen`$::\langle\rangle \Rightarrow bool \rightarrow A \rightarrow A \rightarrow bool$ are defined. We also assume that functions for equality and ordering are defined for booleans and integers: `eqBool`, `eqInt`, `gtBool` and `gtInt`. Notice that the type checker has marked with a $^{\bullet}$ the classes *eq* and *ord* in the types of `eq` and `gt` respectively, as can be seen in the decorations of the different occurrences of these functions. We have defined the `eq` and `gt` functions for booleans and integers using two variables `X` and `Y` as arguments so that the rules have arity 2, instead of defining them as `eq = eqBool`, `eq = eqInt`, etc. The reason for this is that because of HO-patterns, we need that all the rules for overloaded functions have the same arity, as we will discuss in Sect. 5.3. Notice how the type checker decorates function symbols with the corresponding type instantiated to the concrete type used in the application. This is the case of the second occurrence of `eq` in the last rule of the instance *eq (list A)*, which has the decoration $\langle eq^{\bullet}\ (list\ A)\rangle \Rightarrow (list\ A) \rightarrow (list\ A) \rightarrow bool$ since `eq` is applied to lists. Fig. 5-b) shows the result of applying the translation of Def. 3 to the source program. Notice how the same

```
class eq A where
  eq :: A → A → bool

instance eq bool where
  eq X Y = eqBool::⟨⟩ ⇒ bool → bool → bool X Y

instance eq int where
  eq X Y = eqInt::⟨⟩ ⇒ int → int → bool X Y

instance ⟨eq A, eq B⟩ ⇒ eq (pair A B) where
  eq (U,V) (X,Y) = and::⟨⟩ ⇒ bool → bool → bool
    (eq::⟨eq• A⟩ ⇒ A → A → bool U X)
    (eq::⟨eq• B⟩ ⇒ B → B → bool V Y)

instance ⟨eq A⟩ ⇒ eq (list A) where
  eq [] [] = true
  eq [] (Y:Ys) = false
  eq (X:Xs) [] = false
  eq (X:Xs) (Y:Ys) = and::⟨⟩ ⇒ bool → bool → bool
    (eq::⟨eq• A⟩ ⇒ A → A → bool X Y)
    (eq::⟨eq• (list A)⟩ ⇒ (list A) → (list A) → bool Xs Ys)

member :: ⟨eq• A⟩ ⇒ (list A) → A → bool
member::⟨eq• A⟩ ⇒ (list A) → A → bool [] Y = false
member::⟨eq• A⟩ ⇒ (list A) → A → bool (X:Xs) Y =
    or::⟨⟩ ⇒ bool → bool → bool
    (eq::⟨eq•A⟩ ⇒ A → A → bool X Y)
    (member::⟨eq• A⟩ ⇒ (list A) → A → bool Xs Y)

class ⟨eq A⟩ ⇒ ord A where
  gt :: A → A → bool

instance ord bool where
  gt X Y = gtBool::⟨⟩ ⇒ bool → bool → bool X Y

instance ord int where
  gt X Y = gtInt::⟨⟩ ⇒ int → int → bool X Y

memberOrd :: ⟨ord• A⟩ ⇒ (list A) → A → bool
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool [] Y = false
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool (X:Xs) Y = ifthen
  (gt::⟨ord• A⟩ ⇒ A → A → bool X Y) false
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool (X:Xs) Y = ifthen
  (eq::⟨eq• A⟩ ⇒ A → A → bool X Y) true
memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool (X:Xs) Y = ifthen
  (gt::⟨ord• A⟩ ⇒ A → A → bool Y X)
  (memberOrd::⟨ord• A⟩ ⇒ (list A) → A → bool Xs Y)
```

a) Source program with type decorations

```
eq :: A → A → A → bool
eq #bool X Y = eqBool X Y
eq #int X Y = eqInt X Y
eq (#pair X_A X_B) (U,V) (X,Y) = and
  (eq X_A U X)
  (eq X_B V Y)
eq (#list X_A) [] [] = true
eq (#list X_A) [] (Y:Ys) = false
eq (#list X_A) (X:Xs) [] = false
eq (#list X_A) (X:Xs) (Y:Ys) = and
  (eq X_A X Y)
  (eq (#list X_A) Xs Ys)

member :: A → (list A) → A → bool
member X_A [] Y = false
member X_A (X:Xs) Y = or
    (eq X_A X Y)
    (member X_A Xs Y)

gt :: A → A → A → bool
gt #bool X Y = gtBool X Y
gt #int X Y = gtInt X Y

memberOrd :: A → (list A) → A → bool
memberOrd X_A [] Y = false
memberOrd X_A (X:Xs) Y = ifthen
  (gt X_A X Y) false
memberOrd X_A (X:Xs) Y = ifthen
  (eq X_A X Y) true
memberOrd X_A (X:Xs) Y = ifthen
  (gt X_A Y X)
  (memberOrd X_A Xs Y)
```

b) Translated program

Figure 5. Translation of a program using equality and order

type variable A in the decorations generates the same data variable X_A in the translated program—see for example the second rule for `member`. This is important since all these occurrences represent the same type witness that is passed as an argument.

4. Advantages of the Translation

In this section we show some of the benefits of the proposed translation compared to the classical dictionary-based one in FLP.

4.1 Efficiency

To test the efficiency of the proposed translation against the classical translation using dictionaries [10, 30], we have elaborated 7 different programs using type classes. We have chosen programs that can be part of real functional-logic programs and use the standard type classes *eq, ord* and *num*:

- *eqlist*: equality comparison between lists of integers.
- *fib*: Fibonacci function that accepts numeric arguments.
- *galeprimes*: sieve of prime numbers using a function of difference of sorted lists.
- *memberord*: member function in sorted lists.
- *mergesort*: John von Neumann's sorting algorithm.
- *permutsort*: sorting by selecting a sorted permutation of the original list.
- *quicksort*: C.A.R. Hoare's sorting algorithm.

The programs *fib, galeprimes, mergesort* and *quicksort* have been adapted from the suite of benchmark programs for Haskell implementations *nobench* [28]. Although *permutsort* is an inefficient sorting algorithm, we have included it in the set of tests because it is an example of the *generate-and-test* scheme, a kind of programs combining non-determinism and lazy evaluation, for which FLP obtains better results than functional or logic programs [8].

For each program we have measured in Toy the elapsed time in the evaluation of 100 random expressions in both translations. Translated programs using dictionaries are valid programs in Toy since it has a Damas-Milner type system. However, Toy has not integrated the liberal type system for FLP presented in [20]. In order to compile and execute the translated programs with type-indexed functions and type witnesses—which are not correct with respect to a Damas-Milner type system—we have used a especial version of Toy without the type checking phase. This does not distort the

Program	Speedup	Speedup (Optimized)
eqlist	1.6414	1.3627
fib	2.3063	2.3777
galeprimes	1.4885	1.0016
memberord	2.2802	2.2386
mergesort	1.0476	1.0453
permutsort	1.7186	1.7259
quicksort	1.0743	1.0005

Figure 6. Speedup of the proposed translation over the classical translation using dictionaries

measures since once compiled Toy programs do not carry any type information at run time, so compiled programs are the same regardless the type system. For each expression we have calculated the speedup: the elapsed time in the translated program with dictionaries divided by the elapsed time in the translated program using type-indexed functions and type witnesses, and we have computed the mean speedup of the 100 tests. The results appear in the second column of Fig. 6. The biggest speedups are obtained in *fib* and *memberord*. The reason for the speed gain in *fib* is that the function `fib` needs two dictionaries—*ord* and *num*—but only one type witness, which means one extra matching each time `fib` is called. In *memberord* the reason is that it uses the overloaded function `eq` with every element. This function is contained in the *eq* dictionary which is inside the *ord* dictionary, so before apply it we have to extract the *eq* dictionary. This projection is not needed with type witnesses. The programs *permutsort*, *eqlist* and *galeprimes* also obtain a good speedup. In the case of *eqlist*, the reason of the speedup is that the `eq` function builds the dictionary of equality on lists in each recursive call. However, the same type witness argument for lists is passed to the recursive call. The rest of programs—*mergesort* and *quicksort*—do not obtain any improvement and run as fast as with dictionaries.

There are some well-known optimizations that can be applied to the translation using dictionaries [4, 11]. However, in the translation using type-indexed functions and type witnesses there is also room for optimizations. Therefore we have measured the speedup of the same programs when optimizations are applied to both translations. For the dictionary-based translation we have considered those optimizations from [4] applicable to our set of tests. For each test program, the following optimizations have been applied in sequence:

- *Flattening of dictionaries*: expand class dictionaries to contain both the methods of the class and all its superclasses. The dictionary of the superclasses is kept as well as flatting it, because it is sometimes needed.
- *Constant folding*: eliminate the method projection from a dictionary when the concrete dictionary is known. For example, `arb dictArbBool` is replaced by `arbBool`—see Fig. 1-b).
- *Automatic function specialization*: generate an specialized version of a function when it is applied to a concrete dictionary. This optimization has been only applied to `galeprimes`, since it is the only tested program whose code contains a function that is applied to a concrete dictionary.

The rest of optimizations presented in [4] have not been considered because they are dependent on the underlying architecture, which is different between Haskell and Toy, or because they address specific problems which do not appear in our test programs—as programming with complex numbers.

For the proposed translation using type-indexed functions and type witnesses the considered optimizations are:

- *Specialized version from instances*: Apart from the generated rules for the type-indexed functions, instances also generate specialized versions of the overloaded functions. For example, the instance `instance` $\langle eq\ \ A\rangle \Rightarrow eq\ \ (list\ \ A)$ from Fig. 5-a) generates the function `eq_list`:

  ```
  eq_list :: A → (list A) → (list A) → bool
  eq_list X_A [] [] = true
  eq_list X_A [] (Y:Ys) = false
  eq_list X_A (X:Xs) [] = false
  eq_list X_A (X:Xs) (Y:Ys) =
     and (eq X_A X Y) (eq_list X_A Xs Ys)
  ```

  Any occurrence of an overloaded symbol applied to a concrete type witness is replaced by the specialized version: `eq (#list bool)` is replaced by `eq_list #bool`, `ord #nat` by `ord_nat`, etc.
- *Automatic function specialization*: The same optimization explained before, but used when a function is applied to a concrete type witness. This optimization has been only applied to `galeprimes` for the same reasons as before.

The speedup results of the optimized versions appear in the third column of Fig. 6. For the programs *fib*, *memberord*, *mergesort*, *permutsort* and *quicksort*, the speedup does not change substantially. The reason is that dictionary optimizations do not affect the target program—with the exception of a constant folding in the definition of the `ord` dictionaries that is used once per test—and the specialized version of the type-indexed functions are not used. For the program *eqlist* the optimizations avoid the creation of the equality dictionary for lists—in the dictionary-based translation—and make use of the specialized version of equality for list—in the type-passing translation. The speedup decreases but the program with type-indexed functions and type witnesses still runs faster. For the *galeprimes* program there is no speedup since after applying the optimization to both translations the resulting code is similar because of the automatic function specialization.

The code of the tested programs and detailed results of the tests can be found in `http://gpd.sip.ucm.es/enrique/publications/pepm11/testPrograms.zip`.

4.2 Adequacy to call-time choice

Apart from the improvement in efficiency, the proposed translation also solves the problem of missing answers when combining non-determinism and overloading presented in Sect. 1. The problem is that dictionaries are shared, and non-deterministic nullary member functions inside them are evaluated to the same value in all the copies. With the proposed translation this problem does not arise because member function are not projecting functions that extracts from dictionaries but type-indexed functions that accepts a type witness as an argument. This type witness is shared as dictionaries, but each occurrence of the member function is a different application so they can generate different values.

The translation using type-indexed functions and type witnesses of the program containing the `arb` class appeared in Fig. 1-a) is:

```
arb :: A → A
arb #bool → false
arb #bool → true

arbL2 :: A → (list A)
arbL2 X_A → [arb X_A, arb X_A]
```

The class and instance declaration have generated the type-indexed `arb` function with two rules for booleans, and `arbL2` is translated to accept a type witness and pass it to the `arb` functions in its right-hand side. In this case the translation of the expression `arbL2::(list bool)` is `arbL2 #bool`, which can be reduced to `[arb #bool, arb #bool]` using the rule for `arbL2`. Here the

first occurrence of `arb #bool` in the list can be reduced to `false` and the second to `true` using the different rules for `arb`, so it produces the answer `[false, true]` that was missing. In a similar way `arbL2 #bool` can be reduced to `[true, false]`.

The problem with non-deterministic nullary member functions and the dictionary-based translation could be solved if they are automatically replaced by functions of arity 1. This way, dictionaries do not contain functions that can be evaluated but HO-patterns—functions partially applied—that are values and can be shared without problem. However this solution presents some problems that are further discussed in Sect. 5.2.

4.3 Simplicity

From the point of view of difficulty, both translations—the dictionary-based and the proposed one—have a similar complexity: a type checking phase and a translation that uses the obtained type information. However, translated programs using the proposed translation are simpler than those obtained using the dictionary-based one. They are shorter, since they declare less data types and functions. Besides, type witnesses are first-order data, unlike dictionaries which are higher-order data containing functions. Finally, type witnesses have in most cases a simpler structure and are smaller than dictionaries.

With the two translations, obtained programs are the result of an automatic procedure integrated in the compiler, so the simplicity of obtained programs is not so important from the point of view of the user. However, it might be useful for later analyses or manipulations of translated programs. Furthermore, as we have seen in Sect. 4.1 and Sect. 4.2, this simplicity comes with an improvement of the efficiency and a better adequacy to call-time choice.

5. Discussion

In this section we discuss some additional aspects, including some problems, that arise with the translations of type classes in FLP.

5.1 Multiple modules and separate compilation

The dictionary-based translation combines well with multiple modules and separate compilation. A class declaration defines a datatype and some projecting functions, and instances define concrete values of the dictionary type. Therefore different instances can be compiled separately and joined later. With the proposed translation using type-indexed functions and type witnesses this seems more difficult. The problem is that generated type-indexed functions are *open* functions [18]: there is *one* type-indexed function per member function, but the rules can be spread in several modules. However, this is not a problem in Toy due to its code generation method and the demand of the type-indexed functions generated from member functions of classes. Toy programs use a demand driven strategy [19] for evaluating function applications. Consider a `leq` function on Peano natural numbers defined as:

```
leq z Y         = true
leq (s X) z     = false
leq (s X) (s Y) = leq X Y
```

In this case, the first argument is demanded in all the rules, and the second argument is demanded only in the second and third rules. Then the strategy is to evaluate the first argument to *head-normal form*. If it is the constructor `z`, then we apply the first rule. If it is the constructor `s` we evaluate the second argument of the rule. If the evaluation of that argument is the constructor `z` we apply the second rule. Otherwise if it is the constructor `s` we apply the third rule. The Prolog code generated for this function is[3]:

```
leq(A,B,H) :- hnf(A,HA), leq_1(HA,B,H).

leq_1(z,B,true).
leq_1(s(X),B,H) :- hnf(B,HB), leq_1_2(s(X),HB,H).

leq_1_2(s(X),z,false).
leq_1_2(s(X),s(Y),H) :- leq(X,Y,H).
```

The predicate `hnf` is a built-in predicate that computes *head normal forms*. The predicate `leq` is the main predicate to evaluate the `leq` function. It uses the predicates `leq_1` and `leq_1_2`, where the numbers represent in which positions a head normal form has been previously obtained. Notice that the last argument of the predicates represents the result. It is easy to see that these predicates follow the demand driven strategy explained before.

The peculiarity of translated member functions is that they always have a constructor in their first argument: the type-witness. Therefore their first argument is always demanded in all the rules translated from the instances, so the strategy is to evaluate it to head normal form. Consider the `eq` function in Fig. 5-b). Since the first argument is demanded in all the rules, we generate the predicate to evaluate the type witness to head normal form:

```
eq(W,A,B,H) :- hnf(W,HW), eq_1(HW,A,B,H).
```

We also generate the predicate `eq_1` with clauses for the different instances:

```
eq_1(#bool,A,B,H) :- eqBool(A,B,H).
eq_1(#int,A,B,H) :- eqInt(A,B,H).
eq_1(#pair(WA,WB),A,B,H) :- (...)
eq_1(#list(WA),A,B,H) :- (...)
```

If each instance of `eq` is in a different module, we compile them separately. However, in each translated module the first argument of `eq` is uniformly demanded, so we generate the predicate `eq/4` as before and the corresponding clauses for `eq_1/4` and the rest of predicates. Notice that in the translated rules for equality on pairs and list, the three arguments are uniformly demanded. In these cases we chose from left to right, so we always generate the same clause for `eq/4` that computes the head-normal form of the first argument and calls to `eq_1/4`. In the compilation of a program that imports the different modules with the instances, the code for the `eq` function is obtained by simply joining the predicates `eq/4`, `eq_1/4` ... from the compiled modules. Each compiled module contains a clause for `eq/4`, so it is important to remove those duplicates in the final compiled program.

Notice that this solution is not valid for arbitrary open functions, since the demand of the arguments is unknown and the code generation would require an analysis with the rules from all the modules.

5.2 Possible solution for non-deterministic nullary member functions in the dictionary-based translation

The loss of expected answers that arises in the dictionary-based translation when non-deterministic nullary member functions are used could be solved if they are automatically replaced by unary functions. Fig. 7 shows the program translated with dictionaries from Fig. 1-a) where `arb` has been extended to an unary function accepting unit as argument. The translation of `arbL2::(list bool)` is `arbL2 dictArbBool` as in the original case, but now it reduces to `[arb dictArbBool (), arb dictArbBool ()]`. Although both copies of the dictionary are shared, now they can only be reduced to `dictArb arbBool`. It is now a value—notice that `arbBool` is a HO-pattern—so it cannot be reduced further. After the extraction of the `arbBool` function from the dictionary the expression is `[arbBool (), arbBool ()]`, which can be reduced to `[false, true]` or `[true, false]` applying the rule for `arbBool` for twice.

Since being non-deterministic is a typically undecidable property, the technique of adding the unit argument should be applied to every nullary member function, even if it is indeed determinis-

[3] This is not the exact code generated by the Toy compiler. We have simplified it for the sake of conciseness.

```
data dictArb A = dictArb (unit → A)

arb :: dictArb A -> (unit → A)
arb (dictArb F) → F

arbBool :: unit → bool
arbBool () → false
arbBool () → true

dictArbBool :: dictArb bool
dictArbBool → dictArb arbBool

arbL2 :: dictArb A -> list A
arbL2 DA → [arb DA (), arb DA ()]
```

Figure 7. Translation of the program in Fig. 1-a) extending `arb` to have one argument

tic. This will introduce an unnecessary overhead—apart from the inevitable overhead caused by dictionaries—to nullary deterministic member functions. We could consider an analysis to detect (in some cases) if the definition of a nullary member function in a concrete instance is deterministic. In those cases the extra unit argument could in principle be avoided. However this solution makes difficult separate compilation. The reason is that a later inclusion of a new module with an instance where the considered nullary member function is non-deterministic will force the recompilation of all the related modules: it will be necessary to change the dictionary declaration—now it contains a member function whose first argument is of type *unit*—and add the unit argument to the rules in the previous instances.

The translation using type-indexed functions and type witnesses proposed in this paper treats non-deterministic nullary member functions and the rest of member functions in a homogeneous way. Furthermore, it does not require recompilation and it does not add any extra overhead to deterministic nullary member functions—apart from the type-witness. Therefore, we believe that the proposed translation is a better option than the dictionary-based translation when dealing with the combination of non-determinism and nullary member functions.

5.3 Problems with arities and HO-patterns

In our FLP setting the arity of function symbols plays an important role to identify whether a function application forms a HO-pattern or it is totally applied and can be reduced. Therefore all the rules of the same function must have the same arity, and this property must be ensured in the target program. In FP the compiler checks that all the rules of a function have the same number of arguments, but this is not checked for the rules of member functions in different instances. However, this property must be checked if the proposed translation is used. The reason is that the rules of the same member function in different instances are translated to be the rules of the same type-indexed function. If the original rules from the instances have different arities, then the rules for the type-indexed functions will have different arities and the translated program will not be a valid FL program. To solve this problem we propose to annotate the arity of member functions in the class declaration. For example the class declaration for `eq` in Fig. 5-a) is changed to:

`class` $eq\ A$ `where`
  `eq/2` :: $A \to A \to bool$

Using this arity declaration the compiler will be able to check if all the rules for `eq` have the same arity even if they belong to instances in different modules. Notice that this problem with arities does not appear in the dictionary-based translation since the rules of a member function in an instance generates a specialized function—see `arbBool` in Fig. 1-b)—and the member function itself is transformed into a function which projects from the dictionary.

Another problem to address is the occurrence of HO-patterns containing overloaded functions in the patterns of the left-hand side of rules. If this kind of functions appear in the patterns, the type checking stage will decorate them with an overloaded type. Besides, class constraints coming from the overloaded function could remain after context reduction, so the defined function symbol will have an overloaded type containing them. In this situation the proposed translation will generate non-linear functions. Consider the program from Fig. 5-a) and the rule that uses the HO-pattern `eq`:

`f eq` → `true`

After the type checking stage the rule is decorated as:

`f`::$\langle eq^{\bullet}\ A\rangle \Rightarrow (A \to A \to bool) \to bool$
  `eq`::$\langle eq^{\bullet}\ A\rangle \Rightarrow A \to A \to bool$ → `true`

so the translated rule would be:

`f` X_A `(eq` X_A`)` → `true`

This rule is invalid in our setting, since the variable X_A appears twice in the left-hand side so the patterns are non-linear. Notice that this problem also appears in the dictionary-based translation since the same variable representing the dictionary would be passed as the extra argument of `f` and `eq`.

A possible solution to this problem might be not to translate the patterns in the left-hand sides of the rules, so no type witnesses would be added to the overloaded functions in patterns. Since the class constraints from these functions remain in the context of the defined function, they will generate the type witnesses as the first arguments of the defined function. However, this solution leads to a loss of expected answers. Consider the same function rule for `f`. If we do not translate the patterns, the translated rule would be:

`f` X_A `eq` → `true`

which now is linear. The value `true` is an expected answer of the evaluation of `f eq`::$bool \to bool \to bool$—we have added the type decoration to `eq` to avoid ambiguity. The type checker would extend this expression with complete type decorations:

`f`::$\langle eq^{\bullet}\ bool\rangle \Rightarrow (bool \to bool \to bool) \to bool$
  `eq`::$\langle eq^{\bullet}\ bool\rangle \Rightarrow bool \to bool \to bool$

and the translation of this expression would be:

```
f #bool (eq #bool)
```

However this translated expression does not match with the head of the rule `f` X_A `eq`, so it cannot be reduced to `true`. Notice that it also happens with the dictionary-based translation. The translation of the rule would be the same, as `f` needs an extra argument containing the dictionary of equality. The translation of the expression would add two dictionaries for the equality on booleans:

```
f dictEqBool (eq dicEqBool)
```

This translated expression cannot be reduced to the value `true` either. It does not match with the head of the rule for `f`, but the subexpression `eq dicEqBool` can be reduced to `eqBool`—assuming that `eqBool` is the function inside the dictionary of equality for booleans. However the resulting expression `f dictEqBool eqBool` cannot be reduced to `true` using the rule `f` X_A `eq` → `true` because it does not match with its head.

Considering the problems that HO-patterns containing overloaded functions in the left-hand side of rules cause in both translations, it seems a good design choice to prohibit the occurrence of overloaded functions in the patterns in the left-hand side of rules. However HO-patterns are a very expressive feature of FLP, so this problem must be further studied in order to find a solution.

6. Concluding Remarks and Future Work

In this paper we have proposed a translation for type classes in FLP following a type-passing scheme [29]. The translation uses type-indexed functions and type witnesses, and translated programs are well-typed wrt. a new liberal type system for FLP [20]. We argue

that the proposed translation is a good design choice to implement type classes in FLP because it improves on the standard dictionary-based translation in some points:

- Our tests show that translated programs using type-indexed functions and type witnesses perform faster—in general—than those using the dictionary-based translation [10, 30]. The tests also show that if we apply optimizations to both translated programs, those using type-indexed functions and type witnesses still perform faster, although the difference in this case is smaller.
- It does not present the problem of missing answers which appears with the dictionary-based translation in programs that use non-deterministic nullary member functions [24].
- The proposed translation consists in simple steps that make use of type decorations for function symbols obtained by usual type checking algorithms supporting type classes [5, 27], so it does not add extra complications over the standard dictionary-based translation. Besides, translated programs using the proposed translation are shorter and simpler than those generated using the dictionary-based translation.
- Although it needs some special treatment, the proposed translation supports multiple modules and separate compilation in an easy way.

We consider some lines of future work. The first is the implementation of the complete translation into the Toy system. Since the translation rules are pretty simple, the hard step is implementing the standard type checker supporting type classes and place the type decorations in the function symbols. Once the translation is implemented, we will be able to test the efficiency results with a larger set of programs. We also want to study if the proposed translation supports easily well-known extensions of type classes like *multi-parameter type classes* [17] or *constructor classes* [16] for FLP. According to [29], these extensions fit easily in a type-passing translation scheme. Finally, we intend to study in further detail the problematic of HO-patterns using overloaded functions in the left-hand sides of rules, so that we can find better solutions than prohibit them.

Acknowledgments

This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S2009TIC-1465, UCM-BSCH-GR58/08-910502. We also want to acknowledge to Francisco López-Fraguas and Juan Rodríguez-Hortalá for their useful comments and ideas.

References

[1] Münster Curry compiler. `http://danae.uni-muenster.de/~lux/curry/`.

[2] Sloth Curry compiler. `http://babel.ls.fi.upm.es/research/Sloth/`.

[3] Zinc compiler. `http://zinc-project.sourceforge.net/`.

[4] L. Augustsson. Implementing Haskell overloading. In Proc. *FPCA '93*, pages 65–73, 1993.

[5] S. Blott. Type inference and type classes. In *Proc. of the 1989 Glasgow FP Workshop*, pages 254–265, 1990.

[6] J. Cheney and R. Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, July 2003.

[7] L. Damas and R. Milner. Principal type-schemes for functional programs. In Proc. *POPL '82*, pages 207–212, 1982.

[8] J. C. González-Moreno, M. T. Hortalá-González, F. J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[9] J. C. González-Moreno, M. T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), July 2001.

[10] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[11] K. Hammond and S. Blott. Implementing Haskell type classes. In *Proc. of the 1989 Glasgow FP Workshop*, pages 266–286, 1990.

[12] M. Hanus. Multi-paradigm declarative languages. In Proc. *ICLP 2007*, volume 4670 of *LNCS*, pages 45–75. Springer, 2007.

[13] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, March 2006.

[14] R. Hinze and A. Löh. Generic programming, now! In *Datatype-Generic Programming 2006*, volume 4719 of *LNCS*, pages 150–208. Springer, 2007.

[15] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In Proc. *HOPL III*, pages 12–1–12–55, 2007.

[16] M. P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In Proc. *FPCA '93*, pages 52–61, 1993.

[17] S. P. Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In *Haskell Workshop*, 1997.

[18] A. Löh and R. Hinze. Open data types and open functions. In Proc. *PPDP '06*, pages 133–144, 2006.

[19] R. Loogen, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In Proc. *PLILP '93*, pages 184–200, 1993.

[20] F. J. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Liberal Typing for Functional Logic Programs. *To appear APLAS 2010*. Available at `http://gpd.sip.ucm.es/enrique/publications/liberalTypingFLP/aplas2010.pdf`.

[21] F. J. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. New results on type systems for functional logic programming. Volume 5979 of *LNCS*, pages 128–144. Springer, 2010.

[22] F. J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multi-paradigm declarative system. In Proc. *RTA'99*, volume 1631 of *LNCS*, pages 244–247. Springer, 1999.

[23] W. Lux. Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, pages 67–76, 2008.

[24] W. Lux. Type-classes and call-time choice vs. run-time choice - Post to the Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0790.html`, 2009.

[25] J. C. González-Moreno, M. T. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In Proc. *ICLP'97*, pages 153–167, 1997.

[26] J. J. Moreno-Navarro, J. Mariño, A. del Pozo-Pietro, Á. Herranz-Nieva, and J. García-Martín. Adding type classes to functional-logic languages. In *1996 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'96*, pages 427–438, 1996.

[27] T. Nipkow and C. Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[28] D. Stewart. nobench: Benchmarking Haskell implementations. `http://www.cse.unsw.edu.au/~dons/nobench.html`.

[29] S. R. Thatté. Semantics of type classes revisited. In Proc. *LFP '94*, pages 208–219, 1994.

[30] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In Proc. *POPL '89*, pages 60–76, 1989.

[31] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. *SIGPLAN Not.*, 38(1):224–235, 2003. ISSN 0362-1340.

Well-typed Narrowing with Extra Variables in Functional-Logic Programming *

Francisco López-Fraguas Enrique Martin-Martin Juan Rodríguez-Hortalá
Dpto. de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es emartinm@fdi.ucm.es juanrh@fdi.ucm.es

Abstract

Narrowing is the usual computation mechanism in functional-logic programming (FLP), where bindings for free variables are found at the same time that expressions are reduced. These free variables may be already present in the goal expression, but they can also be introduced during computations by the use of program rules with extra variables. However, it is known that narrowing in FLP generates problems from the point of view of types, problems that can only be avoided using type information at run-time. Nevertheless, most FLP systems use static typing based on Damas-Milner type system and they do not carry any type information in execution, thus ill-typed reductions may be performed in these systems. In this paper we prove, using the let-narrowing relation as the operational mechanism, that types are preserved in narrowing reductions provided the substitutions used preserve types. Based on this result, we prove that types are also preserved in narrowing reductions without type checks at run-time when higher order (HO) variable bindings are not performed and most general unifiers are used in unifications, for programs with transparent patterns. Then we characterize a restricted class of programs for which no binding of HO variables happens in reductions, identifying some problems encountered in the definition of this class. To conclude, we use the previous results to show that a simulation of needed narrowing via program transformation also preserves types.

Categories and Subject Descriptors F.3.3 [*Logics and meanings of programs*]: Studies of Program Constructs—Type Structure; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Theory, Languages, Design

Keywords Functional-logic programming, narrowing, extra variables, type systems

1. Introduction

Functional-logic programming (FLP). Functional logic languages [3, 15, 30] like Toy [24] or Curry [16] can be described as an extension of a lazy purely-functional language similar to Haskell [18], that has been enhanced with logical features, in particular logical variables and non-deterministic functions. Disregarding some syntactic conventions, the following program defining standard list concatenation is valid in all the three mentioned languages:

$$[\,] + + Ys = Ys \qquad [X \mid Xs] + + Ys = [X \mid Xs + + Ys]$$

Logical variables are just free variables that get bound during the computation in a way similar to what it is done in logic programming languages like Prolog [11]. This way FLP shares with logic programming the ability of computing with partially unkown data. For instance, assuming a suitable definition and implementation of equality $==$, the following is a natural FLP definition of a predicate (a *true*-valued function) *sublist* stating that a given list Xs is a sublist of Ys:

$$sublist\ Xs\ Ys = cond\ (Us + + Xs + + Vs == Ys)\ true$$
$$cond\ true\ X = X$$

Notice that the rule for *sublist* is not valid in a functional language due to the presence of the variables Us and Vs, which do not occur in the left hand side of the program rule. They are called *extra variables*. Using *cond* and extra variables makes easy translating pure logic programs into functional logic ones[1]. For instance, the logic program using Peano's natural numbers z (zero) and s (successor)

$$add(z, X, X).$$
$$add(s(X), Y, s(Z)) :- add(X, Y, Z).$$
$$even(X) :- add(Y, Y, X).$$

can be transformed into the following functional logic one:

$$add\ z\ X\ Y = cond\ (X == Y)\ true$$
$$add\ (s\ X)\ Y\ (s\ Z) = add\ X\ Y\ Z$$
$$even\ X = add\ Y\ Y\ X$$

Notice that the rule for *even* is another example of FLP rule with an extra variable Y. The previous examples show that, contrary to the usual practice in functional programming, free variables may appear freely during the computation, even when starting from an expression without free variables. Despite these connections with logic programming, owing to the functional characteristics of FLP languages—like the nesting of function applications instead of SLD resolution—several variants and formulations of narrowing [19] have been adopted as the computation mechanism in FLP. There are several operational semantics for computing with logical

* This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S2009TIC-1465 and UCM-BSCH-GR35/10-A-910502.

http://doi.acm.org/10.1145/2103746.2103763.

[1] As a secondary question here, notice that using *cond* is needed if $==$, as usual, is a two-valued function returning *true* or *false*. Defining directly $sublist\ Xs\ Ys = (Us + + Xs + + Vs == Ys)$ would compute wrong answers: evaluating $sublist\ [1]\ [1, 2]$ produces $true$ but also the wrong value $false$, because there are values of the extra variables Us and Vs such that $Us + + [1] + + Vs == [1, 2]$ evaluates to *false*.

and extra variables [15, 25, 30], and this kind of variables are supported in every modern FLP system.

As FLP languages were already non-deterministic due to the different possible instantiations of logical variables—these are handled by means of a backtracking mechanism similar to that of Prolog—it was natural that these languages eventually evolved to include so-called *non-deterministic functions*, which are functions that may return more than one result for the same input. These functions are expressed by means of program rules whose left hand sides overlap, and that are tried in order by backtracking during the computation, instead of taking a first fit or best fit approach like in pure functional languages. The combination of lazy evaluation and non-deterministic functions gives rise to several semantic options, being *call-time choice* semantics [13] the option adopted by the majority of modern FLP implementations. This point can be easily understood by means of the following program example:

$$coin \to z \quad coin \to s\ z \quad dup\ X \to (X, X)$$

In this example *coin* is a non-deterministic expression, as it can be reduced both to the values z and $s\ z$. But the point is that, according to call-time choice the expression *dup coin* evaluates to *(z, z)* and *(s z, s z)* but not to *(z, s z)* nor *(s z, z)*. Operationally, call-time choice means that all copies of a non-deterministic subexpression, like *coin* in the example, created during the computation reduction share the same value. In Section 2.2 we will see a simple formulation of narrowing for programs with extra variables, that also respects call-time choice, which will be used as the operational procedure for this paper.

Apart from these features, in the Toy system left hand sides of program rules can use not only first order patterns like those available in Haskell programs, but also higher order patterns (*HO-patterns*), which essentially are partial applications of function or constructor symbols to other patterns. This corresponds to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics, and it is formalized and semantically characterized with detail in the HO-CRWL[2] logic for FLP [12]. This is not an exoticism: it is known [25] that extensionality is not a valid principle within the combination of higher order functions, non-determinism and call-time choice. HO-patterns are a great expressive feature [30], however they may have some bad interferences with types, as we will see later in the paper.

Because of all the presented features, FLP languages can be employed to write concise and expressive programs, specially for search problems, as it was explored in [3, 15, 30].

FLP and types. Current FLP languages are strongly typed. Apart from programming purposes, types play a key role in some program analysis or transformations for FLP, as detecting deterministic computations [17], translation of higher order into first order programs [4], or transformation into Haskell [8]. From the point of view of types FLP has not evolved much from Damas-Milner type system [9], so current FLP systems use an almost direct adaptation of that classic type system. However, that approach lacks type preservation during evaluation, even for the restricted case where we drop logical and extra variables. It is known from afar [14] that, even in that simplified scenario, HO-patterns break the type preservation property. In particular, they allows us to create polymorphic casting functions [7]—functions with type $\forall\alpha, \beta.\alpha \to \beta$, but that behave like the identity wrt. the reduction of expressions. This has motivated the development of some recent works dealing with *opaque HO-patterns* [22], or liberal type systems for FLP [21]. There are also some preliminary works concerning the incorporation of *type classes* to FLP languages [26, 29], but this feature is still in an experimental phase in current systems.

Regardless of the expressiveness of extra variables, these are usually out the scope of the works dealing with types and FLP, in particular in all the aforementioned. But these variables are a distinctive feature of FLP systems, hence in this work our *main goal* is to investigate the properties of a variation of the Damas-Milner type system that is able to handle extra variables, giving an abstract characterization of the problematic issues—most of them were already identified in the seminal work [14]—and then determining sufficient conditions under which type preservation is recovered for programs with extra variables evaluated with narrowing. In particular, we are interested in preserving types without having to use type information at run-time, in contrast to what it is done in previous proposals [14].

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about programs and expressions, and the formulation of the let-narrowing relation $\rightsquigarrow^{l}$, which will be used as the operational mechanism for this paper. In Section 3 we present our type system and study those interactions with let-narrowing that lead to the loss of type preservation. Then we define the well-typed let-narrowing relation $\rightsquigarrow^{lwt}$, a restriction of $\rightsquigarrow^{l}$ that preserves types relying on the abstract notion of well-typed substitution. To conclude that section we present $\rightsquigarrow^{lmgu}$, another restriction of $\rightsquigarrow^{l}$ that is able to preserve types without using type information—in contrast to $\rightsquigarrow^{lwt}$, which uses types at each step to determine that the narrowing substitution is well-typed—at the price of losing some completeness. To cope with this lack of completeness, in Section 4 we look for sufficient conditions under which the narrowing relation $\rightsquigarrow^{lmgu}$ is complete wrt. the computation of well-typed solutions, thus identifying a class of programs for which completeness is recovered, and whose expressiveness is then investigated. In Section 5 we propose a simulation of needed narrowing with $\rightsquigarrow^{lmgu}$ via two well-known program transformations, and show that it also preserves types. The class of programs supported in that section is specially relevant, as it corresponds to a simplified version of the Curry language. Finally Section 6 summarizes some conclusions and future work. Fully detailed proofs, including some auxiliary results, can be found in the extended version of this paper [23].

2. Preliminaries

2.1 Expressions and programs

We consider a set of *functions* symbols $f, g, \ldots \in FS$ and *constructor* symbols $c, d, \ldots \in CS$, each $h \in FS \cup CS$ with an associated arity $ar(h)$. We also consider a denumerable set of *data variables* $X, Y, \ldots \in \mathcal{V}$. The notation $\overline{o_n}$ stands for a sequence $o_1, \ldots, o_n$ of n syntactic elements o, being o_i the i^{th} element. Figure 1 shows the syntax of patterns $t \in Pat$ and expressions $e \in Exp$. We split the set of patterns into two: *first order patterns* $FOPat \ni fot ::= X \mid c\ \overline{fot_n}$ where $ar(c) = n$, and *higher-order patterns* $HOPat = Pat \setminus FOPat$, i.e., patterns containing some partial application of a symbol of the signature. Expressions $X\ \overline{e_n}$ are called *variable application* when $n > 0$, and expressions with the form $h\ \overline{e_n}$ are called *junk* if $h \in CS$ and $n > ar(h)$ or *active* if $h \in FS$ and $n \geq ar(h)$. The set of *free* and *bound* variables of an expression e—$fv(e)$ and $bv(e)$ resp.—are defined in the usual way. Notice that let-expressions are not recursive, so $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$. The set $var(e)$ is the set containing all the variables in e, both free and bound. Notice that for patterns $var(t) = fv(t)$.

Contexts $\mathcal{C} \in Cntxt$ are expressions with one hole, and the application of $\mathcal{C}$ to e—written $\mathcal{C}[e]$—is the standard. The notion of free and bound variables are extended in the natural way to

[2] CRWL [13] stands for *Constructor Based Rewriting Logic*; HO-CRWL is a higher order extension of it.

Data variable			$X,Y\ldots$
Function symbol			$f,g\ldots$
Constructor symbol			$c,d\ldots$
Non-variable symbol	h	::=	$c \mid f$
Symbol	s	::=	$X \mid c \mid f$
Pat	t,p	::=	X
			$\mid c\ \overline{t_n}$ if n $\leq$ ar(c)
			$\mid f\ \overline{t_n}$ if n $<$ ar(f)
FOPat	fot	::=	$X \mid c\ \overline{fot_n}$ if n = ar(c)
Exp	e,r	::=	$X \mid c \mid f \mid e_1\ e_2$
			$\mid let\ X = e_1\ in\ e_2$
PSubst	θ	::=	$[\overline{X_n \mapsto t_n}]$
Cntxt	$\mathcal{C}$	::=	$[\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$
			$\mid let\ X = \mathcal{C}\ in\ e$
			$\mid let\ X = e\ in\ \mathcal{C}$
Program rule	R	::=	$f\ \overline{t_n} \to e$ if ar(f) = n
Program	$\mathcal{P}$	::=	$\{\overline{R_n}\}$
Type variable			$\alpha,\beta\ldots$
Type constructor			C
Simple type	τ	::=	$\alpha \mid \tau_1 \to \tau_2$
			$\mid C\ \overline{\tau_n}$ if n = ar(C)
Type-scheme	σ	::=	$\forall\overline{\alpha_n}.\tau$
Set of assumptions	$\mathcal{A}$	::=	$\{\overline{s_n : \sigma_n}\}$
TSubst	π	::=	$[\overline{\alpha_n \mapsto \tau_n}]$

Figure 1. Syntax of programs and types

contexts: $fv(\mathcal{C}) = fv(\mathcal{C}[h])$ for any $h \in FS \cup CS$ with $ar(h) = 0$, and $bv(\mathcal{C})$ is defined as $bv([\,]) = \emptyset$, $bv(\mathcal{C}\ e) = bv(\mathcal{C})$, $bv(e\ \mathcal{C}) = bv(\mathcal{C})$, $bv(let\ X = \mathcal{C}\ in\ e) = bv(\mathcal{C})$, $bv(let\ X = e\ in\ \mathcal{C}) = \{X\} \cup bv(\mathcal{C})$.

Data substitution $\theta \in PSubst$ are finite maps from data variables to patterns $[\overline{X_n \mapsto t_n}]$. We write ϵ for the empty substitution, $dom(\theta)$ for the domain of θ and $vran(\theta) = \bigcup_{X \in dom(\theta)} fv(X\theta)$. Given $A \subseteq \mathcal{V}$, the notation $\theta|_A$ represents the restriction of θ to D, and $\theta|_{\setminus A}$ is a shortcut for $\theta|_{\mathcal{V} \setminus A}$. Substitution application over data variables and expressions is defined in the usual way.

Program rules R have the form $f\ \overline{t_n} \to e$, where $ar(f) = n$ and $\overline{t_n}$ is *linear*, i.e., there is no repetition of variables. Notice that we allow extra variables, so it could be the case that e contains variables which do not appear in $\overline{t_n}$. A program $\mathcal{P}$ is a set of program rules.

2.2 Let-narrowing

Let-narrowing [25] is a narrowing relation devised to effectively deal with logical and extra variables, that is also sound and complete wrt. HO-CRWL [12], a standard logic for higher order FLP with call-time choice. Figure 2 contains the rules of the let-narrowing relation $\rightsquigarrow^l$. The first five rules (LetIn)–(LetAp) do not use the program and just change the textual representation of the term graph implied by the let-bindings in order to enable the application of program rules, but keeping the implied term graph untouched. The (Narr) rule performs function application, finding the bindings for the free variables needed to be able to apply the rule, and possibly introducing new variables if the program rule contains some extra variables. Notice that it does not require the use of a most general unifier (mgu) so any unifier can be used. As we will see in Section 3, this later point should be refined in order to ensure type preservation. Rules (VAct) and (VBind) produce HO bindings for variable applications, and are needed for let-narrowing to be complete. These rules are particularly problematic because they have to generate speculative bindings that may involve any function of the program, contrary to (Narr) where the computation of bindings is directed by the program rules for f. Later on we will see how this "wild" nature of the bindings generated by these rules poses especially hard problems to type preservation. Finally, (Contx) allows to apply a narrowing rule in any part of the expression, protecting bound variables from narrowing and avoiding variable capture.

3. Type Preservation

In this section we first present the type system we will use in this work, which is a simple variation of Damas-Milner typing enhanced with support for extra variables. Then we show some examples of $\rightsquigarrow^l$-reductions not preserving types (Section 3.2). Based on the ideas that emerge from these examples, in Section 3.3 we develop a new let-narrowing relation $\rightsquigarrow^{lwt}$ that preserves types. This new relation uses only *well-typed substitutions* in each step, which gives an abstract and general characterization of the requirements a narrowing relation must fulfil in order to preserve types, but it still needs to perform type checks at run-time. To solve this problem, in Section 3.4 we present a restricted let-narrowing $\rightsquigarrow^{lmgu}$ which only uses mgu's as unifiers and drops the problematic rules (VAct) and (VBind). The main advantage of this relation is that if the patterns that can appear in program rules are limited then mgu's are always well-typed, thus obtaining type preservation without using type information at run-time. Sadly this comes at a price, as $\rightsquigarrow^{lmgu}$ loses some completeness wrt. HO-CRWL.

3.1 A type system for extra variables

In Figure 1 we can find the usual syntax for *simple types* τ and *type-schemes* σ. For a simple type τ, the set of *free type variables*—denoted $ftv(\tau)$—is $var(\tau)$, and for type-schemes $ftv(\forall\overline{\alpha_n}.\tau) = var(\tau) \setminus \{\overline{\alpha_n}\}$. A type-scheme is *closed* if $ftv(\sigma) = \emptyset$. We say that a type-scheme is *k-transparent* if it can be written as $\forall\overline{\alpha_n}.\overline{\tau_k} \to \tau$ such that $var(\overline{\tau_k}) \subseteq var(\tau)$.

A set of assumptions $\mathcal{A}$ is a set of the form $\{\overline{s_n : \sigma_n}\}$ such that the assumption for variables are simple types. If $(s_i : \sigma_i) \in \mathcal{A}$ we write $\mathcal{A}(s_i) = \sigma_i$. For sets of assumptions we define $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. The union of set of assumptions is denoted by $\oplus$ with the usual meaning: $\mathcal{A} \oplus \mathcal{A}'$ contains all the assumptions in $\mathcal{A}'$ as well as the assumptions in $\mathcal{A}$ for those symbols not appearing in $\mathcal{A}'$. Based on the previous notion of k-transparency, we say a pattern t is *transparent* wrt. $\mathcal{A}$ if $t \in \mathcal{V}$ or $t \equiv h\ \overline{t_n}$ where $\mathcal{A}(h)$ is n-transparent and $\overline{t_n}$ are transparent patterns. We also say a constructor symbol c is transparent wrt. $\mathcal{A}$ if $\mathcal{A}(c)$ is n-transparent, where $ar(c) = n$.

Type substitutions $\pi \in TSubst$ are mappings from type variables to simple types, where dom and $vran$ are defined similarly to data substitutions. Application of type substitutions to simple types is defined in the natural way, and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions: $\{\overline{s_n : \sigma_n}\}\pi = \{\overline{s_n : \sigma_n\pi}\}$. We say τ is a *generic instance* of $\sigma \equiv \forall\overline{\alpha_n}.\tau'$ if $\tau = \tau'[\overline{\alpha_n \mapsto \tau_n}]$ for some $\overline{\tau_n}$, written $\sigma \succ \tau$. Finally, τ is a *variant* of $\sigma \equiv \forall\overline{\alpha_n}.\tau'$ (denoted by $\sigma \succ_{var} \tau$) if $\tau = \tau'[\overline{\alpha_n \mapsto \beta_n}]$ where $\overline{\beta_n}$ are fresh type variables.

Figure 3 contains the typing rules for expressions considered in this work, which constitute a variation of Damas-Milner typing that now is able to handle extra variables. The main novelty wrt. a regular formulation of Damas-Milner typing with support for pattern matching is that now the (Λ) rule considers extra variables in λ-abstractions: in addition to guessing types for the variables in the pattern t, it also guesses types for the free variables of $\lambda t.e$, which correspond to extra variables. Although λ-abstractions are expressions not included in the syntax of programs showed in Fig-

(LetIn) $e_1\ e_2 \rightsquigarrow^l_\epsilon let\ X = e_2\ in\ e_1\ X$, if e_2 is an active expression, variable application, junk or let-rooted expression, for X fresh.
(Bind) $let\ X = t\ in\ e \rightsquigarrow^l_\epsilon e[X \mapsto t]$, if $t \in Pat$
(Elim) $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\epsilon e_2$, if $X \notin fv(e_2)$
(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightsquigarrow^l_\epsilon let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, if $Y \notin fv(e_3)$
(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \rightsquigarrow^l_\epsilon let\ X = e_1\ in\ e_2\ e_3$, if $X \notin fv(e_3)$
(Narr) $f\ \overline{t_n} \rightsquigarrow^l_\theta r\theta$, for any fresh variant $(f\ \overline{p_n} \to r) \in \mathcal{P}$ and θ such that $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$.
(VAct) $X\ \overline{t_k} \rightsquigarrow^l_\theta r\theta$, if $k > 0$, for any fresh variant $(f\ \overline{p} \to r) \in \mathcal{P}$ and θ such that $(X\ \overline{t_k})\theta \equiv f\ \overline{p}\theta$
(VBind) $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\theta e_2\theta[X \mapsto e_1\theta]$, if $e_1 \notin Pat$, for any θ that makes $e_1\theta \in Pat$, provided that $X \notin (dom(\theta) \cap vran(\theta))$
(Contx) $\mathcal{C}[e] \rightsquigarrow^l_\theta \mathcal{C}\theta[e']$, for $\mathcal{C} \neq [\,]$, $e \rightsquigarrow^l_\theta e'$ using any of the previous rules, and:
i) $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$
ii) • if the step is (Narr) or (VAct) using $(f\ \overline{p_n} \to r) \in \mathcal{P}$ then $vran(\theta|_{\setminus var(\overline{p_n})}) \cap bv(\mathcal{C}) = \emptyset$
• if the step is (VBind) then $vran(\theta) \cap bv(\mathcal{C}) = \emptyset$.

Figure 2. Let-narrowing relation $\rightsquigarrow^l$

ure 1 and thus they cannot appear in the expressions to reduce[3], we use them as the basis for the notions of well-typed rule and program. Essentially, for each program rule we construct an associated λ-abstraction so the rule is well-typed iff the corresponding λ-abstraction is well-typed. This is reflected in the following definition of *program well-typedness*, an important property assuring that assumptions over functions are related to their rules:

DEFINITION 3.1 (Well-typed program wrt. $\mathcal{A}$). *A program rule $f \to e$ is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ where $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ and $\overline{\tau_n}$ are some simple types. A program rule $(f\ \overline{p_n} \to e)$ (with $n > 0$) is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.e : \tau$ with $\mathcal{A}(f) \succ_{var} \tau$. A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ if all its rules are well-typed wrt. $\mathcal{A}$.*

This definition is the same as the one from [22] but it has a different meaning, as it is based on a different definition for the (Λ) rule. Notice that the case $f \to e$ must be handled independently because it does not have any argument. In this case the (Λ) rule is not used to derive the type for e, so the types for the extra variables would not be guessed.

An expression e is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash e : \tau$ for some type τ, written as $wt_{\mathcal{A}}(e)$. We will use the metavariable $\mathcal{D}$ to denote particular type derivations $\mathcal{A} \vdash e : \tau$. If $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ we write $wt_{\mathcal{A}}(\mathcal{P})$.

3.2 Let-narrowing does not preserve types

Now we will see how let-narrowing interacts with types. It is easy to see that let-narrowing steps $\rightsquigarrow^l$ which do not generate bindings for the logical variables—i.e., those using the rules (LetIn), (Bind), (Elim), (Flat) and (LetAp)—preserve types trivially. This is not very surprising because, as we showed in Section 2.2, those steps just change the textual representation of the implied term graph. However, steps generating non trivial bindings can break type preservation easily:

EXAMPLE 3.2. *Consider the function and defined by the rules $\{and\ true\ X \to X, and\ false\ X \to false\}$ with type $(bool \to bool \to bool)$ and the constructor symbols for Peano's natural numbers z and s, with types (nat) and $(nat \to nat)$ respectively. Starting from the expression $and\ true\ Y$—which has type bool when Y has type bool—we can perform the let-narrowing step:*

$$and\ true\ Y \rightsquigarrow^l_{[X_1 \mapsto z, Y \mapsto z]} z$$

This (Narr) step uses the fresh program rule $(and\ true\ X_1 \to X_1)$, but the resulting expression z does not have type bool.

The cause of the loss of type preservation is that the unifier $\theta_1 = [X_1 \mapsto z, Y \mapsto z]$ used in the (Narr) step is ill-typed, because it replaces the boolean variables X_1 and Y by the natural z. The problem with θ_1 is that it instantiates the variables too much, and without using any criterion that ensures that the types of the expressions in its range are adequate.

We have just seen that using the (Narr) rule with an ill-typed unifier may lead to breaking type preservation because of the instantiation of logical variables, like the variable Y above. We may reproduce the same problem easily with extra variables, just consider the function f with type bool defined by the rule $(f \to and\ true\ X)$ for which we can perform the following let-narrowing step:

$$f \rightsquigarrow^l_{[X_2 \mapsto z]} and\ true\ z$$

using (Narr) with the fresh rule $(f \to and\ true\ X_2)$. The resulting expression is obviously ill-typed, and so type preservation is broken again because the substitution used in (Narr) instantiates variables too much and without assuring that the expression in its range have the correct types. The interested reader may easily check that this is also a valid let-rewriting step [25], thus showing that extra variables break type preservation even in the restricted scenario where we drop logical variables. Hence, the type systems in the

$$\text{(ID)}\ \frac{}{\mathcal{A} \vdash s : \tau}\ \text{if } \mathcal{A}(s) \succ \tau$$

$$\text{(APP)}\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$\text{(}\Lambda\text{)}\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau}\ \text{if } \{\overline{X_n}\} = var(t) \cup fv(\lambda t.e)$$

$$\text{(LET)}\ \frac{\mathcal{A} \vdash e_1 : \tau_x \quad \mathcal{A} \oplus \{X : \tau_x\} \vdash e_2 : \tau}{\mathcal{A} \vdash let\ X = e_1\ in\ e_2 : \tau}$$

Figure 3. Type System

[3] As there is no general consensus about the semantics of λ-abstractions in the FLP community, due to their interactions with non-determinism and logical variables, we have decided to leave λ-abstractions out of programs and evaluating expressions, thus following the usual applicative programming style of the HO-CRWL logic.

papers mentioned at the end of Section 1 lose type preservation if we allow extra variables in the programs.

However, the (Narr) rule is not the only one which can break type preservation. The rules (VAct) and (VBind) also lead to problematic situations:

EXAMPLE 3.3. *Consider the functions and symbols from Example 3.2. Using the rule (VAct) it is possible to perform the step*

$$s\ (F\ z) \leadsto^{l}_{[F \mapsto and\ false, X_3 \mapsto z]} s\ false$$

with the fresh rule $(and\ false\ X_3 \to false)$*. Clearly* $s\ (F\ z)$ *has type* nat *and* F *has type* $(nat \to nat)$*, but the resulting expression is ill-typed. As before, the reason is an ill-typed binding for* F*, which binds* F *with a pattern of type* $(bool \to bool)$*.*

On the other hand, we can perform the step

$$let\ X = F\ z\ in\ s\ X \leadsto^{l}_{[F \mapsto and]} s\ (and\ z)$$

using the rule (VBind). The expression $let\ X = F\ z\ in\ s\ X$ *has type* nat *when* F *has type* $(nat \to nat)$*, but the resulting expression is ill-typed. The cause of the loss of type preservation is again an ill-typed substitution binding, in this case the one for* F *which assigns a pattern of type* $(bool \to bool \to bool)$ *to a variable of type* $(nat \to nat)$*.*

Notice that ill-typed substitutions do not break type preservation necessarily. For example the step $and\ false\ X \leadsto^{l}_{\theta_5} false$ using (Narr) with the fresh rule $(and\ false\ X_5 \to false)$ preserves types, although it can use the ill-typed unifier $\theta_5 \equiv [X \mapsto z, X_5 \mapsto z]$. However, avoiding ill-typed substitutions is a sufficient condition which guarantees type preservation, as we will see soon. Besides, it is important to remark that the bindings for the free variables of the starting expression that are computed in a narrowing derivation are as important as the final value reached at the end of the derivation, because these bindings constitute a solution for the starting expression if we consider it as a goal to be solved, just like the goal expressions used in logic programming. That allows us to use predicate functions like the function *sublists* in Section 1 with some variables as their arguments, i.e., using some arguments in Prolog-like output mode. Therefore, well-typedness of the substitutions computed in narrowing reductions is also important and the restriction to well-typed substitutions is not only reasonable but also desirable, as it ensures that the solutions computed by narrowing respect types.

3.3 Well-typed let-narrowing $\leadsto^{lwt}$

In this section we present a narrowing relation $\leadsto^{lwt}$ which is smaller than $\leadsto^{l}$ in Figure 2 but that preserves types. The idea behind $\leadsto^{lwt}$ is that it only considers steps $e \leadsto^{l}_{\theta} e'$ using well-typed programs where the substitution θ is also well-typed. We say a substitution is well-typed when it replaces data variables by patterns of the same type. Formally:

DEFINITION 3.4 (Well-typed substitution). *A data substitution* θ *is well-typed wrt.* $\mathcal{A}$*, written* $wt_{\mathcal{A}}(\theta)$*, if* $\mathcal{A} \vdash X\theta : \mathcal{A}(X)$ *for every* $X \in dom(\theta)$*.*

Notice that according to the definition of set of assumptions, $\mathcal{A}(X)$ is always a simple type.

As it is usual in narrowing relations, let-narrowing steps can introduce new variables that do not occur in the original expression. Moreover, this new variables do not come only from extra variables but from fresh variants of program rules—using (Narr) and (VAct)—or from invented patterns—using (VBind). Therefore, we need to consider some suitable assumptions over these new variables. However, that set of assumptions over the new variables is not arbitrary but it is closely related to the step used:

EXAMPLE 3.5 ($\mathcal{A}$ associated to a (Narr) step). *Consider the function* f *with type* $\forall\alpha.\alpha \to [\alpha]$ *defined with the rule* $f\ X \to [X, Y]$*. We can perform the narrowing step* $f\ true \leadsto^{l}_{\theta} [true, Y_1]$ *using (Narr) with the fresh variant* $f\ X_1 \to [X_1, Y_1]$ *and* $\theta \equiv [X_1 \mapsto true]$*. Since the original expression is* $f\ true$*, it is clear that* X_1 *must have type* $bool$ *in the new set of assumptions. Moreover,* Y_1 *must have the same type since it appears in a list with* X_1*. Therefore in this concrete step the associated set of assumptions is* $\{X_1 : bool, Y_1 : bool\}$*.*

The following definition establishes when a set of assumptions is associated to a step. Notice that due to the particularities of the rules (VAct) and (VBind), in some cases there is not such set or there are several associated sets.

DEFINITION 3.6 ($\mathcal{A}$ associated to $\leadsto^{l}$ steps). *Given a type derivation* $\mathcal{D}$ *for* $\mathcal{A} \vdash e : \tau$ *and* $wt_{\mathcal{A}}(\mathcal{P})$*, a set of assumptions* $\mathcal{A}'$ *is associated to the step* $e \leadsto^{l}_{\theta} e'$ *iff:*

- $\mathcal{A}' \equiv \emptyset$ *and the step is (LetIn), (Bind), (Elim), (Flat) or (LetAp).*
- *If the step is (Narr) then* $f\ \overline{t_n} \leadsto^{l}_{\theta} r\theta$ *using a fresh variant* $(f\ \overline{p_n} \to r) \in \mathcal{P}$ *and substitution* θ *such that* $(f\ \overline{p_n})\theta \equiv (f\ \overline{t_n})\theta$*. Since* $\mathcal{D}$ *is a type derivation for* $\mathcal{A} \vdash f\ \overline{t_n} : \tau$*, it will contain a derivation* $\mathcal{A} \vdash f : \overline{\tau_n} \to \tau$*. The rule* $f\ \overline{p_n} \to r$ *is well-typed by* $wt_{\mathcal{A}}(\mathcal{P})$*, so we also have (when the rule is* $f \to e$ *it is similar):*

$$(\Lambda)\ \dfrac{\mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash p_n : \tau'_n \quad \mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash r : \tau'}{\vdots}\ (\Lambda)\ \dfrac{\mathcal{A} \oplus \mathcal{A}_1 \vdash p_1 : \tau'_1 \quad \vdots}{\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'}$$

  where $\overline{\mathcal{A}_n}$ *are the set of assumptions over variables introduced by* (Λ) *and* $\overline{\tau'_n} \to \tau'$ *is a variant of* $\mathcal{A}(f)$*. Therefore* $(\overline{\tau'_n} \to \tau')\pi \equiv \overline{\tau_n} \to \tau$ *for some type substitution* π *whose domain are fresh type variables from the variant. In this case* $\mathcal{A}'$ *is associated to the (Narr) step if* $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$*.*
- *If the step is (VAct) then we have* $X\ \overline{t_k} \leadsto^{l}_{\theta} r\theta$ *for a fresh variant* $(f\ \overline{p_n} \to r) \in \mathcal{P}$ *and substitution* θ *such that* $(X\ \overline{t_k})\theta \equiv f\ \overline{p_n}\theta$*. Since* $\mathcal{D}$ *is a type derivation for* $\mathcal{A} \vdash X\ \overline{t_k} : \tau$*, it will contain a derivation* $\mathcal{A} \vdash X : \overline{\tau_k} \to \tau$*. The rule* $f\ \overline{p_n} \to r$ *is well-typed by* $wt_{\mathcal{A}}(\mathcal{P})$*, so we have a type derivation* $\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'$ *as in the (Narr) case (similarly when the rule is* $f \to e$*). Let* $\overline{\tau''_k}$ *be* $\tau'_{n-k+1} \to \tau'_{n-k+2} \ldots \to \tau'_n$*, i.e., the last* k *types in* $\overline{\tau'_n}$*. If* $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$ *for some substitution* π *such that* $(\overline{\tau''_k} \to \tau')\pi \equiv \overline{\tau_k} \to \tau$ *and* $fv(\mathcal{A}) \cap dom(\pi) = \emptyset$*, then* $\mathcal{A}'$ *is associated to the (VAct) step.*
- *Any* $\mathcal{A}' \equiv \{\overline{X_n : \tau_n}\}$ *is associated to a (VBind) step, if* $\overline{X_n}$ *are those data variables introduced by* $vran(\theta)$*—they do not appear in* $\mathcal{A}$*—and* $\overline{\tau_n}$ *are simple types.*
- $\mathcal{A}'$ *is associated to a (Contx) step if it is associated to its inner step.*

A set of assumptions $\mathcal{A}'$ *is associated to* n $\leadsto^{l}$ *steps* $(e_1 \leadsto^{l} e_2 \ldots \leadsto^{l} e_{n+1})$ *if* $\mathcal{A}' \equiv \mathcal{A}'_1 \oplus \mathcal{A}'_2 \ldots \oplus \mathcal{A}'_n$*, where* $\mathcal{A}'_i$ *is associated to the step* $e_i \leadsto^{l} e_{i+1}$ *and the type derivation* $\mathcal{D}_i$ *for* e_i *using* $\mathcal{A} \oplus \mathcal{A}'_1 \ldots \oplus \mathcal{A}'_{i-1}$ *(*$\mathcal{A}' \equiv \emptyset$ *if* $n = 0$*).*

Based on the previously introduced notions we can define a restriction of let-narrowing that only employs well-typed substitutions, that we will denote by $\leadsto^{lwt}$:

DEFINITION 3.7 ($\leadsto^{lwt}$ let-narrowing). *Consider an expression* e*, a program* $\mathcal{P}$ *and set of assumptions* $\mathcal{A}$ *such that* $wt_{\mathcal{A}}(e)$ *with a derivation* $\mathcal{D}$ *and* $wt_{\mathcal{A}}(\mathcal{P})$*. Then* $e \leadsto^{lwt}_{\theta} e'$ *iff* $e \leadsto^{l}_{\theta} e'$ *and* $wt_{\mathcal{A} \oplus \mathcal{A}'}(\theta)$*, where* $\mathcal{A}'$ *is a set of assumptions associated to* $e \leadsto^{l}_{\theta} e'$*,* $\mathcal{D}$*.*

The premises $wt_{\mathcal{A}}(e)$ and $wt_{\mathcal{A}}(\mathcal{P})$ are essential, since the associated set of assumptions wrt. $e \leadsto^{l}_{\theta} e'$ is only well defined in those cases. Note that the step $\leadsto^{lwt}$ cannot be performed if no set of associated assumptions $\mathcal{A}'$ exists. Although $\leadsto^{lwt}$ is strictly smaller than $\leadsto^{l}$—the steps in Examples 3.2 and 3.3 are not valid $\leadsto^{lwt}$-steps—it enjoys the intended type preservation property:

THEOREM 3.8 (Type preservation of $\leadsto^{lwt}$). *If $wt_{\mathcal{A}}(\mathcal{P})$, $e \leadsto^{lwt^*}_{\theta} e'$ and $\mathcal{A} \vdash e : \tau$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

The previous result is the main contribution of this paper. It states clearly that, provided that the substitutions used are well-typed, let-narrowing steps preserve types. Moreover, type preservation is guaranteed for general programs, i.e., programs containing extra variables, non-transparent constructor symbols, opaque HO-patterns ... This result is very relevant because it clearly isolates a sufficient and reasonable property that, once imposed to the unifiers, ensures type preservation. Besides, this condition is based upon the abstract notion of well-typed substitution, which is parameterized by the type system and independent of the concrete narrowing or reduction notion employed. Thus the problem of type preservation in let-narrowing reductions is clarified. New let-narrowing subrelations can be proposed for restricted classes of programs or using particular unifiers and, provided the generated substitutions are well-typed, they will preserve types. We will see an example of that in Section 3.4.

This is an important advance wrt. previous proposals like [14], where the computation of the mgu was interleaved with and inseparable from the rest of the evaluation process in the narrowing derivations. Besides, although the identification of three kinds of problematic situations for the type preservation made in that work was very valuable—especially taking into account it was one of the first studies of the subject in FLP with HO-patterns—having a more general and abstract result is also valuable for the reasons stated above.

3.4 Restricted narrowing using mgu's $\leadsto^{lmgu}$

The $\leadsto^{lwt}$ relation has the good property of preserving types, however it presents a drawback if used as the reduction mechanism of a FLP system: it requires the substitutions generated in each $\leadsto^{lwt}$ step to be well-typed. Since these substitutions are generated just by using the syntactic criteria expressed in the rules of the let-narrowing relation $\leadsto^{l}$, the only way to guarantee this is to perform type checks at run-time, discarding ill-typed substitutions. But, as we mentioned in Section 1, we are interested in preserving types without having to use type information at run-time. Hence, in this section we propose a new let-narrowing relation $\leadsto^{lmgu}$ which preserves types without need of type checks at run-time. The let-narrowing relation $\leadsto^{lmgu}$ is defined as:

DEFINITION 3.9 (Restricted narrowing $\leadsto^{lmgu}$). *$e \leadsto^{lmgu}_{\theta} e'$ iff $e \leadsto^{l}_{\theta} e'$ using any rule from Figure 2 except (VAct) and (VBind), and if the step is $f\ \overline{t_n} \leadsto^{l}_{\theta} r\theta$ using (Narr) with the fresh variant $(f\ \overline{p_n} \to r)$ then $\theta = mgu(f\ \overline{t_n}, f\ \overline{p_n})$.*

As explained in Section 3.2, the rules that break type preservation are (Narr), (VAct) and (VBind). The rules (VAct) and (VBind) present harder problems to preserve types since they replace HO variables by patterns. These patterns are searched in the entire space of possible patterns, producing possible ill-typed substitutions. Since we want to avoid type checks at run-time, and we have not found any syntactic criterion to forbid the generation of ill-typed substitutions by those rules, (VAct) and (VBind) have been omitted from $\leadsto^{lmgu}$. Although this makes $\leadsto^{lmgu}$ a relation strictly smaller than $\leadsto^{lwt}$, it is still meaningful: expressions needing (VAct) or (VBind) to proceed can be considered as *frozen* until other let-narrowing step instantiates the HO variable. This is somehow similar to the operational principle of *residuation* used in some FLP languages such as Curry [15, 16]. Regarding the rule (Narr), Example 3.2 shows the cause of the break of type preservation. In that example, the unifier of $and\ true\ Y$ and $and\ true\ X_1$ is $\theta_1 = [X_1 \mapsto z, Y \mapsto z]$. Although θ_1 is a valid unifier, it instantiates variables unnecessarily in an ill-typed way. In other words, it does not use just the information from the program and the expression, which are well-typed, but it "invents" the pattern z. We can solve this situation easily using the mgu $\theta'_1 = [X_1 \mapsto Y]$, which is well-typed, so by Theorem 3.8 we can conclude that the step preserves types.

Moreover, this solution applies to any (Narr) step (under certain conditions that will be specified later): if we chose mgu's in the (Narr) rule and both the rule and the original expression are well-typed, then the mgu's will also be well-typed. This fact is based in the following result:

LEMMA 3.10 (Mgu well-typedness). *Let $\overline{p_n}$ be fresh linear transparent patterns wrt. $\mathcal{A}$ and let $\overline{t_n}$ be any patterns such that $\mathcal{A} \vdash p_i : \tau_i$ and $\mathcal{A} \vdash t_i : \tau_i$ for some type τ_i. If $\theta \equiv mgu(f\ \overline{p_n}, f\ \overline{t_n})$ then $wt_{\mathcal{A}}(\theta)$.*

The restriction to fresh linear transparent patterns $\overline{p_n}$ is essential, otherwise the mgu may not be well-typed. Consider for example the constructor $cont : \forall\alpha.\alpha \to container$ and a set of assumptions $\mathcal{A}$ containing $(X : nat)$. It is clear that $p \equiv cont\ X$ is linear but non-transparent, because $cont$ is not 1-transparent. Both p and $t \equiv cont\ true$ patterns have type $container$ and $mgu(f\ p, f\ t) = [X \mapsto true] \equiv \theta$ for any function symbol f. However the unifier θ is ill-typed as $\mathcal{A} \not\vdash X\theta : \mathcal{A}(X)$, i.e., $\mathcal{A} \not\vdash true : nat$. Similarly, consider the patterns $p' \equiv (Y, Y)$ and $t' \equiv (cont\ X, cont\ true)$ and a set of assumptions $\mathcal{A}$ containing $(Y : container, X : nat)$. It is easy to see that p' and t' have type $(container, container)$, and p' is transparent but non-linear. The mgu of $f\ p'$ and $f\ t'$ is $[Y \mapsto cont\ true, X \mapsto true]$, which is ill-typed by the same reasons as before.

Due to the previous result, type preservation is only guaranteed for $\leadsto^{lmgu}$-reductions for programs such that left-hand sides of rules contain only transparent patterns. This is not a severe limitation, as it is considered in other works [14], and as we will see in the next section.

THEOREM 3.11 (Type preservation of $\leadsto^{lmgu}$). *Let $\mathcal{P}$ be a program such that left-hand sides of rules contain only transparent patterns. If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $e \leadsto^{lmgu^*}_{\theta} e'$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

So finally, with $\leadsto^{lmgu}$ we have obtained a narrowing relation that is able to ensure type preservation without using any type information at run-time. However, as we mentioned before, this comes at the price of losing completeness wrt. HO-CRWL, not only because we are restricted to using mgu's—which is not a severe restriction, as we will see later—but mainly because we are not able to use the rules (VAct) and (VBind) any more, which are essential for generating binding for variable applications like those in Example 3.3. We will try to mitigate that problem in Section 4.

4. Reductions without Variable Applications

In this section we want to identify a class of programs in which $\leadsto^{lmgu}$ is sufficiently complete so it can perform well-typed narrowing derivations without losing well-typed solutions. As can be

seen in the Lifting Lemma from [25], the restriction of the let-narrowing relation $\leadsto^{l}$ that only uses mgu's in each step is complete wrt. HO-CRWL. Therefore, we strongly believe that the restriction of $\leadsto^{lwt}$ using only mgu's is complete wrt. to the computation of well-typed solutions, although proving it is an interesting matter of future work. For this reason, in this section we are only concerned about determining under which conditions $\leadsto^{lmgu}$ is complete wrt. the restriction of $\leadsto^{lwt}$ to mgu's.

Our experience shows that although we only have to assure that neither (VAct) nor (VBind) are used, the characterization of such a family of programs is harder than expected. In Section 4.1 we show the different approaches tried, explaining their lacks, that led us to a restrictive condition—Section 4.2. This condition limits the expressiveness of the programs, hence we explore the possibilities of that class of programs in Section 4.3.

4.1 Naive approaches

Our first attempt follows the idea that if an expression does not contain any free HO variable (free variable with a functional type of the shape $\tau \to \tau'$) then neither (VAct) nor (VBind) can be used in a narrowing step. This result is stated in the following easy Lemma:

LEMMA 4.1 (Absence of HO variables). *Let e be an expression such that $wt_{\mathcal{A}}(e)$ and for every $X_i \in fv(e)$, $\mathcal{A}(X_i)$ is not a functional type. Then no step $e \leadsto^{l}_{\theta} e'$ can use (VAct) or (VBind).*

Our belief was that if an expression does not contain free HO variables and the program does not have extra HO variables, the resulting expression after a $\leadsto^{lmgu}$ step does not have free HO variables either. This is false, as the following example shows:

EXAMPLE 4.2. *Consider a constructor symbol bfc with type $bfc : (bool \to bool) \to BoolFunctContainer$ and the function f with type $f : BoolFunctContainer \to bool$ defined as $\{f\ (bfc\ F) \to F\ true\}$. We can perform the narrowing reduction*

$$f\ X \leadsto^{lmgu}_{\theta} F_1\ true$$

where $\theta \equiv [X \mapsto bfc\ F_1] = mgu(f\ X, f\ (bfc\ F_1))$. The free variable F_1 introduced has a functional type, however the original expression has not any free HO variable—X has the ground type $BoolFunctContainer$. Moreover, the program does not contain extra variables at all.

The previous example shows that not only free HO variables must be avoided in expressions, but also free variables with *unsafe* types as *BoolFunctContainer*. The reason is that patterns with unsafe types may contain HO variables. Those patterns can appear in left-hand sides of rules, so a narrowing step can unify a free variable with one of these patterns, thereby introducing free HO variables—notice that the unification of X and $bfc\ F_1$ introduces the free HO variable F_1 in the previous example. To formalize these intuitions we define the set of *unsafe* types as those for which problematic patterns can be formed:

DEFINITION 4.3 (Unsafe types). *The set of unsafe types wrt. a set of assumptions $\mathcal{A}$ ($UTypes_{\mathcal{A}}$) is defined as the least set of simple types verifying:*

1. *Functional types ($\tau \to \tau'$) are in $UTypes_{\mathcal{A}}$.*
2. *A simple type τ is in $UTypes_{\mathcal{A}}$ if there exists some pattern $t \in Pat$ with $\{\overline{X_n}\} = var(t)$ such that:*
   a) $t \equiv \mathcal{C}[X_i]$ with $\mathcal{C} \neq [\,]$
   b) $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$, for some $\overline{\tau_n}$
   c) $\tau_i \in UTypes_{\mathcal{A}}$.

For brevity we say a variable X is *unsafe* wrt. $\mathcal{A}$ if $\mathcal{A}(X)$ is unsafe wrt. $\mathcal{A}$.

$$(\Lambda^r)\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \{\overline{Y_k : \tau'_k}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda^r t.e : \tau_t \to \tau}$$

where $\{\overline{X_n}\} = var(t)$, $\{\overline{Y_k}\} = fv(\lambda^r t.e)$ such that $\overline{\tau'_k}$ are ground and safe wrt. $\mathcal{A}$.

Figure 4. Typing rule for restricted λ-abstractions

Clearly, if an expression does not contain free unsafe variables it does not contain free HO variables either, so by Lemma 4.1 neither (VAct) nor (VBind) could be used in a narrowing step. However, the absence of unsafe variables is not preserved after $\leadsto^{lmgu}$ steps even if the rules do not contain unsafe extra variables:

EXAMPLE 4.4. *Consider the symbols in Example 4.2 and a new function g defined as $\{g \to X\}$ with type $g : \forall\alpha.\alpha$. The extra variable X has the polymorphic type α in the rule for g, so it is safe. The expression $(f\ g)$ does not contain any unsafe variable, however we can make the reduction:*

$$f\ g \leadsto^{lmgu}_{\epsilon} f\ X_1 \leadsto^{lmgu}_{[X_1 \mapsto bfc\ F_1]} F_1\ true$$

The new variable X_1 introduced has type $BoolFunctContainer$, which is unsafe.

Example 4.4 shows that not only unsafe free variables must be avoided, but any expression of unsafe type which can be reduced to a free variable. In this case the problematic expression is g, which has type *BoolFunctContainer* and produces a free variable. Example 4.4 also shows that polymorphic extra variables are a source of problems, since they can take unsafe types depending on each particular use.

4.2 Restricted programs

Based on the problems detected in the previous section, we characterize a restricted class of programs and expressions to evaluate in which $\leadsto^{lwt}$ steps do not apply (VAct) and (VBind). First, we need that the expression to evaluate does not contain unsafe variables. Second, we forbid rules whose extra variables have unsafe types. Finally, we must also avoid polymorphic extra variables, since they can take different types, in particular unsafe ones. The restriction over programs is somehow tight: any program with functions using polymorphic extra variables are out of this family of programs, in particular the function *sublist* in Section 1 and other common functions using extra variables—see Section 4.3 for a detailed discussion.

In order to define formally this family of programs, we propose a restricted notion of well-typed programs. This notion is very similar to that in Definition 3.1, but using the restricted typing rule (Λ^r) for λ-abstractions in Figure 4, which avoids extra variables with polymorphic or unsafe types.

DEFINITION 4.5 (Well-typed restricted program). *A program rule $f \to e$ is well-typed restricted wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ where $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ and $\overline{\tau_n}$ are some ground and safe simple types wrt. $\mathcal{A}$. A program rule $(f\ \overline{p_n} \to e)$ (with $n > 0$) is well-typed restricted wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash \lambda^r p_1 \ldots \lambda^r p_n.e : \tau$ with $\mathcal{A}(f) \succ_{var} \tau$. A program $\mathcal{P}$ is well-typed restricted wrt. $\mathcal{A}$ if all its rules are well-typed restricted wrt. $\mathcal{A}$.*

If a program $\mathcal{P}$ is well-typed restricted wrt. $\mathcal{A}$ we write $wt^r_{\mathcal{A}}(\mathcal{P})$. Notice that for any $\mathcal{P}$ and $\mathcal{A}$ we have that $wt^r_{\mathcal{A}}(\mathcal{P})$ implies $wt_{\mathcal{A}}(\mathcal{P})$. For the rest of the section we will implicitly use this notion of well-typed restricted programs. Since the notion of well-typed substitution, and as a consequence the notion of $\leadsto^{lwt}$

step, is parameterized by the type system, then further mentions to $\leadsto^{lwt}$ in this section will refer to a relation slightly smaller than the one presented in Section 3.3: a variant of $\leadsto^{lwt}$ based on the type system from Definition 4.5. It is easy to see that this variant also preserves types in derivations. Therefore, although the following results are limited to this variant, they are still relevant.

The key property of well-typed restricted programs is that, starting from an expression without unsafe variables, the resulting expression of a $\leadsto^{lwt}$ reduction do not contain such variables either:

LEMMA 4.6 (Absence of unsafe variables). *Let e be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \leadsto^{lwt^*}_{\theta} e'$ then e' does not contain unsafe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

Notice that the use of mgu's in the $\leadsto^{lwt}$ steps is not necessary in the previous lemma, as the absence of unsafe variables is guaranteed by the well-typed substitution implicit in the definition of the $\leadsto^{lwt}$. Based on Lemma 4.6, it is easy to prove that $\leadsto^{lmgu}$ is complete to the restriction of $\leadsto^{lwt}$ to mgu's:

THEOREM 4.7 (Completeness of $\leadsto^{lmgu}$ wrt. $\leadsto^{lwt}$). *Let e be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \leadsto^{lwt^*}_{\theta} e'$ using mgu's in each step then $e \leadsto^{lmgu^*}_{\theta} e'$.*

Notice that completeness is assured even for programs having non transparent left-hand sides, as well-typedness of substitutions is guaranteed by $\leadsto^{lwt}$.

4.3 Expressiveness of the restricted programs

The previous section states the completeness of $\leadsto^{lmgu}$ wrt. $\leadsto^{lwt}$ for the class of well-typed restricted programs, when only mgu's are used in (Narr) steps. However this class leaves outside a number of interesting functions containing extra variables. For example, the *sublist* function in Section 1 is discarded. The reason is that extra variables of the rule—*Us* and *Vs*—must have type $[\alpha]$, which is not ground. A similar situation happens with other well-known polymorphic functions using extra variables, as the *last* function to compute the last element of a list—$last\ Xs \to cond\ (Ys +\!\!+ [E] == Xs)\ E$ [15]—or the function to compute the inverse of a function at some point—$inv\ F\ X \to cond\ (F\ Y == X)\ Y$. A consequence is that the class of well-typed restricted programs excludes many polymorphic functions using extra variables, since they usually have extra variables with polymorphic types.

However, not all functions using extra variables are excluded from the family of well-typed restricted programs. An example is the *even* function from Section 1 that checks whether a natural number is even or not. The whole rule has type $nat \to nat$ and it contains the extra variable Y of type *nat*, which is ground and safe, making the rule valid. Other functions handling natural numbers and using extra variables as $compound\ X \to cond\ (times\ M\ N == X)\ true$—where *times* computes the product of natural numbers—are also valid, since both M and N have type *nat*. Moreover, versions of the rejected polymorphic functions adapted to concrete ground types are also in the family of well-typed restricted programs. For example, functions as *sublistNat* or *lastBool* with types $[nat] \to [nat] \to bool$ and $[bool] \to bool$ and the same rules as their polymorphic versions are accepted. However, this is not a satisfactory solution: the generation of versions for the different types used implies duplication of code, which is clearly contrary to the degree of code reuse and generality offered by declarative languages—specially by means of polymorphic functions and the different input/output modes of function arguments.

The class of well-typed restricted programs is tighter than desired, and leaves out several interesting functions. Furthermore, for some of those functions—as *subslist* or *last*—we have not discovered any example where unsafe variables were introduced during reduction[4]. Therefore, we plan to further investigate the characterization of such a family in order to widen the number of programs accepted, while leaving out the problematic ones.

5. Type Preservation for Needed Narrowing

In this section we consider the type preservation problem for a simplified version of the Curry language, where features irrelevant to the scope of this paper are ignored, like constraints, encapsulated search, i/o, etc. Therefore we restrict ourselves to *simple Curry programs*, i.e., programs using only first-order patterns and transparent constructor symbols—which implies that all the patterns in left-hand sides are transparent. Besides, programs will be evaluated using the *needed narrowing* strategy [5] and performing residuation for variable applications—which is simulated by dropping the rules (VAct) and (VBind). We have decided to focus on needed narrowing because it is the most popular on-demand evaluation strategy, and it is at the core of the majority of modern FLP systems.

We use a transformational approach to employ $\leadsto^{lmgu}$ to simulate an adaptation of the needed narrowing strategy for let-narrowing. We rely on two program transformations well-known in the literature. In the first one, we start with an arbitrary simple Curry program and transform it into an *overlapping inductively sequential* (OIS) program [1]. For programs in this class, an *overlapping definitional tree* is available for every function, that encodes the demand structure implied by the left-hand sides of its rules. Then we proceed with the second transformation, which takes an OIS program and transforms it into *uniform format* [32]: programs in which the left-hand sides of the rules for every function f have either the shape $f\ \overline{X}$ or $f\ \overline{X}\ (c\ \overline{Y})\ \overline{Z}$.

There are other well-known transformations from general programs to OIS programs—for example [10]—but we have chosen the transformation in Definition 5.1—which is similar to the transformation in [2], but now extended to generate type assumptions—because of its simplicity. The transformation processes each function independently: it takes the set of rules $\mathcal{P}_f$ for each function f and returns a pair composed by the transformed rules and a set of assumptions for the auxiliary fresh functions introduced by the transformation.

DEFINITION 5.1 (Transformation to OIS). *Let $\mathcal{P}_f \equiv \{f\ \overline{t^1_n} \to e^1, \ldots, f\ \overline{t^m_n} \to e^m\}$ be a set of m program rules for the function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If f is an OIS function, $OIS(\mathcal{P}_f) = (\mathcal{P}_f, \emptyset)$. Otherwise $OIS(\mathcal{P}_f) = (\{f_1\ \overline{t^1_n} \to e^1, \ldots, f_m\ \overline{t^m_n} \to e^m, f\ \overline{X_n} \to f_1\ \overline{X_n}?\ldots?f_m\ \overline{X_n}\}, \{f_m : \mathcal{A}(f)\})$, where $?$ is the non-determistic choice function defined with the rules $\{X?Y \to X, X?Y \to Y\}$.*

The following result states that the transformation OIS preserves types. Notice that any other transformation to OIS format that also preserves types could be used instead.

THEOREM 5.2 ($OIS(\mathcal{P}_f)$ well-typedness). *Let $\mathcal{P}_f$ be a set of program rules for the same function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $OIS(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$.*

After the transformation the assumption for f remains the same and the new assumptions refer to fresh function symbols. There-

[4] The function *inv* can introduce HO variables when combined with a constant function as $zero\ X \to z$ with type $\forall\alpha.\alpha \to nat$: $(inv\ zero\ z)\ true \leadsto^{lwt^*}_{\theta} Y_1\ true$, where Y_1 is clearly unsafe.

fore, it is easy to see that the previous result is also valid for programs with several functions.

Now, to transform the program from OIS into uniform format we use the following transformation, which is a slightly variant of the transformation in [32]. Like in the previous transformation, we treat each function independently, returning the translated rules together with the extra assumptions for the auxiliary functions.

DEFINITION 5.3 (Transformation to uniform format). *Let $\mathcal{P}_f \equiv \{f\ \overline{t_n^1} \to e^1, \ldots, f\ \overline{t_n^m} \to e^m\}$ be an OIS program of m program rules for a function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{P}_f$ is already in uniform format, then $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \emptyset)$. Otherwise, we take the uniformly demanded position[5] o and split $\mathcal{P}_f$ into r sets $\overline{\mathcal{P}_r}$ containing the rules in $\mathcal{P}_f$ with the same constructor symbol in position o. Then $\mathcal{U}(\mathcal{P}_f) = (\bigcup_{i=1}^{r} \mathcal{P}_i' \cup \mathcal{P}'', \bigcup_{i=1}^{r} \mathcal{A}_i' \cup \mathcal{A}'')$ where:*

- $\mathcal{U}(\mathcal{P}_i^o) = (\mathcal{P}_i', \mathcal{A}_i')$
- *c_i is the constructor symbol in position o in the rules of $\mathcal{P}_i$, with $ar(c_i) = k_i$*
- *$\mathcal{P}_i^o$ is the result of replacing the function symbol f in $\mathcal{P}_i$ by $f_{(c_i,o)}$ and flattening the patterns in position o in the rules, i.e., $f\ \overline{t_j}\ (c_i\ \overline{t_{k_i}'})\ \overline{t_l''} \to e$ is replaced by $f_{(c_i,o)}\ \overline{t_j}\ \overline{t_{k_i}'}\ \overline{t_l''} \to e$*
- *$\mathcal{P}'' \equiv \{f\ \overline{X_j}\ (c_1\ \overline{Y_{k_1}})\ \overline{Z_l} \to f_{(c_1,o)}\ \overline{X_j}\ \overline{Y_{k_1}}\ \overline{Z_l}, \ldots, f\ \overline{X_j}\ (c_r\ \overline{Y_{k_r}})\ \overline{Z_l} \to f_{(c_r,o)}\ \overline{X_j}\ \overline{Y_{k_r}}\ \overline{Z_l}\}$, with $\overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l}$ pairwise distinct fresh variables such that $j + l + 1 = n$*
- *$\mathcal{A}'' \equiv \{f_{(c_1,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau_{k_1}'} \to \overline{\tau_l} \to \tau, \ldots, f_{(c_r,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau_{k_r}'} \to \overline{\tau_l} \to \tau\}$ where $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$ and $\mathcal{A} \oplus \{\overline{Y_{k_i} : \tau_{k_i}'}\} \vdash c_i\ \overline{Y_{k_i}} : \tau'$. Notice that since constructor symbols c_i are transparent, these $\overline{\tau_{k_i}'}$ do exist and are univocally fixed.*

This transformation also preserves types. For the same reasons as before, the following result is also valid for programs with several functions.

THEOREM 5.4 ($\mathcal{U}(\mathcal{P}_f)$ well-typedness). *Let $\mathcal{P}_f$ be a set of program rules for the same overlapping inductive sequential function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$.*

We have just seen that we can transform an arbitrary program into uniform format while preserving types. The preservation of the semantics is also stated in [2, 32]. Although these results have been proved in the context of term rewriting, we strongly believe that they remain valid for the call-time choice semantics of the HO-CRWL framework. Similarly, we are strongly confident that the completeness of narrowing with mgu's over a uniform program wrt. needed narrowing over the original program [32] is also valid in the framework of let-narrowing. Combining those results with the type preservation results for $\leadsto^{lmgu}$ and the program transformations—Theorems 3.11, 5.2 and 5.4—we can conclude that a simulation of the evaluation of simple Curry programs using $\leadsto^{lmgu}$ based on the transformations above, is safe wrt. types.

6. Conclusions and Future Work

In this paper we have tackled the problem of type preservation for FLP programs with extra variables. As extra variables lead to the introduction of fresh free variables during the computations, we have decided to use the let-narrowing relation $\leadsto^{l}$—which is sound and complete wrt. HO-CRWL, a standard semantics for FLP—as the operational mechanism for this paper. This is also a natural choice because let-narrowing reflects the behaviour of current FLP systems like Toy or Curry, that provide support for extra and logical variables instead of reducing expressions by rewriting only.

The other main technical ingredient of the paper is a novel variation of Damas-Milner type system that has been enhanced with support for extra variables. Based on this type system we have defined the well-typed let-narrowing relation $\leadsto^{lwt}$, which is a restriction of let-narrowing that preserves types. To the best of our knowledge, this is the first paper proposing a polymorphic type system for FLP programs with logical and extra variables such that type preservation is formally proved. As we have seen in Example 3.2 from Section 3 the type systems from [21, 22] lose type preservation when extra variables are introduced. In [4], another remarkable previous work, the proposed type system only supports monomorphic functions and extra variables are not allowed. In [14] only programs with transparent patterns and without extra variables are considered, and functional arguments in data constructors are forbidden. Nevertheless, any of those programs is supported by our $\leadsto^{lwt}$ relation, which has to carry type information at run-time, but just like the extension of the Constructor-based Lazy Narrowing Calculus proposed in [14].

The relevance of Theorem 3.8, which states that $\leadsto^{lwt}$ preserves types, lies in the clarification it makes of the problem of type preservation on narrowing reductions with programs with extra variables. Relying on the abstract notion of well-typed substitution, which is parametrized by the type system and independent of any concrete operational mechanism, we have isolated a sufficient condition that ensures type preservation when imposed to the unifiers used in narrowing derivations. This contrasts with previous works like [14]—the closest to the present paper—in which a most general unifier was implicitly computed. Moreover, $\leadsto^{lwt}$ preserves types for arbitrary programs, something novel in the field of type systems in FLP—to the best of our knowledge. Hence, $\leadsto^{lwt}$ is an intended ideal narrowing relation that always preserves types, but that can only be directly realized by using type checks at run-time. Therefore, $\leadsto^{lwt}$ is most useful when used as a reference to define some imperfect but more practical materializations of it—subrelations of $\leadsto^{lwt}$—that only work for certain program classes but also preserve types while avoiding run-time type checks. An example of this is the relation $\leadsto^{lmgu}$, whose applicability is restricted to programs with transparent patterns, and that also lacks some completeness. This relation is based on two conditions imposed over $\leadsto^{l}$ steps: mgu's are used in every (Narr) step; and the rules (VAct) and (VBind) are avoided. While the former is not a severe restriction—as $\leadsto^{l}$ is complete wrt. HO-CRWL even if only mgu's are allowed as unifiers [25]—the latter is more problematic, because then $\leadsto^{lmgu}$ is not able to generate bindings for variable applications. To mitigate this weakness we have investigated how to prevent the use of (VAct) and (VBind) in $\leadsto^{lwt}$ derivations. After some preliminary attempts that witness the difficulty of the task, and also give valuable insights about the problem, we have finally characterized a class of programs in which these bindings for variable applications are not needed, and studied their expressiveness. Then we have applied the results obtained so far for proving the type preservation for a simplified version of the Curry language. HO-patterns are not supported in Curry, which treats functions as black boxes [4]. Therefore Curry programs do not intend to generate solutions that include bindings for variable applications, and so the rules (VAct) and (VBind) will not be used to evaluate these programs. Besides, in Curry all the constructors are transparent, and the needed narrowing on-demand strategy is employed in most implementations of Curry. We have used two well-known program transformations to simulate the evaluation of Curry programs with an adaptation of needed narrowing for let-narrowing. Then we have proved that both transformations preserve types which, combined

[5] A position in which all the rules in $\mathcal{P}_f$ have a constructor symbol. Notice that this position will always exist because $\mathcal{P}_f$ is an OIS program [1].

with the type preservation of $\leadsto^{lmgu}$, implies that our proposed simulation of needed narrowing also preserves types.

Regarding future work, we would like to look for new program classes more general than the one presented in Section 4 because, as we pointed out at the end of that section, the proposed class is quite restrictive and it forbids several functions that we think are not dangerous for the types.

Another interesting line of future work would deal with the problems generated by opaque pattens, as we did in [22] for the restricted case where we drop logical and extra variables. We think that an approach in the line of existential types [20] that, contrary to [22], forbids pattern matching over existential arguments, is promising. This has to do with the parametricy property of types systems [31], which is broken in [22] as we allowed matching on existential arguments, and which is completely abandoned from the very beginning in [21]. In fact it was already detected in [14] that the loss of parametricity leads to the loss of type preservation in narrowing derivations—in that paper instead of parametricity the more restrictive property of type generality is considered. All that suggests that our first task regarding this subject should be modifying our type system from [22] to recover parametricity by following an approach to opacity closer to standard existential types.

References

[1] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. 6th Int. Conf. on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

[2] S. Antoy. Constructor based conditional narrowing. In *Proc. 3rd Int. Conf. on Principles and Practice of Declarative Programming (PPDP'01)*, pages 199–206. ACM, 2001.

[3] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.

[4] S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Proc. 4th Int. Symp. on Functional and Logic Programming (FLOPS'99)*, pages 335–352. Springer LNCS 1722, 1999.

[5] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47:776–822, July 2000.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] B. Brassel. Two to three ways to write an unsafe type cast without importing unsafe - Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0705.html`, May 2008.

[8] B. Brassel, S. Fischer, M. Hanus and F. Reck Transforming Functional Logic Programs into Monadic Functional Programs In *Proc. 19th Int. Work. on Functional and (Constraint) Logic Programming (WFLP'10)*, Springer LNCS 6559, pages 30–47, 2011.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM, 1982.

[10] R. del Vado Vírseda. Estrategias de estrechamiento perezoso. Master's thesis, Universidad Compluetense de Madrid, 2002.

[11] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard. Reference Manual*. Springer, 1996.

[12] J. González-Moreno, T. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. 14th Int. Conf. on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.

[13] J. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1): 47–87, 1999.

[14] J. González-Moreno, T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), July 2001.

[15] M. Hanus. Multi-paradigm declarative languages. In *Proc. 23rd Int. Conf. on Logic Programming (ICLP'07)*, pages 45–75. Springer LNCS 4670, 2007.

[16] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). `http://www.informatik.uni-kiel.de/~curry/report.html`, March 2006.

[17] M. Hanus and F. Steiner. Type-based nondeterminism checking in functional logic programs. In *Proc. 2nd. Inf. Conf. Principles and Practice of Declarative Programming. (PDP 2000)*, pages 202–213, ACM, 2000.

[18] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. 3rd ACM SIGPLAN Conf. on History of Programming Languages (HOPL III)*, pages 12–1–12–55. ACM, 2007.

[19] J.-M. Hullot. Canonical forms and unification. In *Proc. 5th Conf. on Automated Deduction (CADE-5)*, pages 318–334. Springer LNCS 87, 1980.

[20] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16:1411–1430, 1994.

[21] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Liberal typing for functional logic programs. In *Proc. 8th Asian Symp. on Programming Languages and Systems (APLAS'10)*, pages 80–96. Springer LNCS 6461, 2010.

[22] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. New results on type systems for functional logic programming. In *Proc. 18th Int. Workshop on Functional and (Constraint) Logic Programming (WFLP'09), Revised Selected Papers*, pages 128–144. Springer LNCS 5979, 2010.

[23] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Well-typed narrowing with extra variables in functional-logic programming (extended version). Technical Report SIC-11-11, Universidad Complutense de Madrid, November 2011. `http://gpd.sip.ucm.es/enrique/publications/pepm12/SIC-11-11.pdf`.

[24] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. 10th Int. Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[25] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th Int. Symp. on Functional and Logic Programming (FLOPS'08)*, pages 147–162. Springer LNCS 4989, 2008.

[26] W. Lux. Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, pages 67–76, 2008.

[27] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[28] E. Martin-Martin. Advances in type systems for functional logic programming. Master's thesis, Universidad Complutense de Madrid, July 2009. `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`.

[29] E. Martin-Martin. Type classes in functional logic programming. In *Proc. 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 121–130. ACM, 2011.

[30] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Constraints in Computational Logics*, pages 202–270. Springer LNCS 2002, 2001.

[31] P. Wadler. Theorems for free! In *Proc. 4th Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359. ACM, 1989.

[32] F. Zartmann. Denotational abstract interpretation of functional logic programs. In *Proc. 4th Int. Symp. on Static Analysis (SAS'97)*, pages 141–159. Springer LNCS 1302, 1997.

B. Versiones extendidas

(B.1) **Advances in Type Systems for Functional Logic Programming (Extended Version)**

Francisco López-Fraguas, Enrique Martin-Martin y Juan Rodríguez-Hortalá

Technical Report SIC-05-12, Universidad Complutense de Madrid, 2012.

→ Página 222

(B.2) **Well-typed Narrowing with Extra Variables in Functional-Logic Programming (Extended Version)**

Francisco López-Fraguas, Enrique Martin-Martin y Juan Rodríguez-Hortalá

Technical Report SIC-11-11, Universidad Complutense de Madrid, 2011.

→ Página 271

Advances in Type Systems for Functional Logic Programming (Extended Version)*

Technical Report SIC-05-12

Francisco J. López-Fraguas
Enrique Martin-Martin
Juan Rodríguez-Hortalá

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, emartinm@fdi.ucm.es, juanrh@fdi.ucm.es

Abstract. Type systems are widely used in programming languages as a powerful tool providing safety to programs, and forcing the programmers to write code in a clearer way. Functional-logic languages have inherited Damas & Milner type system from their functional part due to its simplicity and popularity. In this paper we address a couple of aspects that can be subject of improvement. One is related to a problematic feature of functional logic languages not taken under consideration by standard systems: it is known that the use of *opaque* HO patterns in left-hand sides of program rules may produce undesirable effects from the point of view of types. We re-examine the problem, and propose a Damas & Milner-like type system where certain uses of HO patterns (even opaque) are permitted while preserving type safety, as proved by a subject reduction result that uses *HO-let-rewriting*, a recently proposed operational semantics for HO functional logic programs. At the same time that we formalize the type system, we have made the effort of technically clarifying additional issues: one is the different ways in which polymorphism of local definitions can be handled, and the other is the overall process of type inference in a whole program.

1 Introduction

Type systems for programming languages are an active area of research [17], no matters which paradigm one considers. In the case of functional programming, most type systems have arisen as extensions of Damas & Milner's [3], for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference). Functional logic languages [11,7,6], in their practical side, have inherited more or less directly Damas & Milner's types.

* This paper is the extended version of "**Advances in Type Systems for Functional Logic Programming**" appeared in *Pre-proceedings of the 18th International Workshop on Functional and (Constraint) Logic Programming (WFLP'09)*, June 28, 2009, Brasília, Brazil.

In principle, most of the type extensions proposed for functional programming could be also incorporated to functional logic languages (this has been done, for instance, for type classes in [14]). However, if types are not only decoration but are to provide safety, one should be sure that the adopted system has indeed good properties. In this paper we tackle a couple of aspects of existing FLP systems that are problematic or not well covered by standard Damas & Milner systems. One is the presence of so called *HO patterns* in programs, an expressive feature for which a sensible semantics exists [4]; however, it is known that unrestricted use of HO patterns leads to type unsafety, as recalled below. The second is the degree of polymorphism assumed for local pattern bindings, a matter with respect to which existing FP or FLP systems vary greatly.

The rest of the paper is organized as follows. The next two subsections further discuss the two mentioned aspects. Section 2 contains some preliminaries about FL programs and types. In Section 3 we expose the type system and prove its soundness wrt. the *let rewriting* semantics of [10]. Section 4 contains a type inference relation, which let us find the most general type of expressions. Section 5 present a method to infer types for programs. Finally, Section 6 contains some conclusions and future work.

1.1 Higher order patterns

In our formalism patterns appear in the left-hand side of rules and in lambda or let expressions. Some of these patterns can be HO patterns, if they contain partial applications of function or constructor symbols. HO patterns can be a source of problems from the point of view of the types. In particular, it was shown in [5] that unrestricted use of HO patterns leads to loss of expected property of *subject reduction* (i.e., evaluation does not change types), an essential property for a type system. The following is a crisp example of the problem.

Example 1 (Polymorphic Casting [2]). Consider the program consisting of the rules $snd\ X\ Y \to Y$, $and\ true\ X \to X$, $and\ false\ X \to false$, $id\ X \to X$, with the usual types inferred by a classical Damas & Milner algorithm. Then we can write the functions $co\ (snd\ X) \to X$ and $cast\ X \to co\ (snd\ X)$, whose inferred types will be $\forall\alpha.\forall\beta.(\alpha \to \alpha) \to \beta$ and $\forall\alpha.\forall\beta.\alpha \to \beta$ respectively. It is clear that $and\ (cast\ 0)\ true$ is well-typed, because $cast\ 0$ has type $bool$ (in fact it has any type), but if we reduce the expression to $and\ 0\ true$ using the rule of $cast$ the resulting expression is bad-typed.

The problem arises when dealing with HO patterns, because unlike FO patterns, knowing the type of a pattern does not always permit us to know the type of its subpatterns. In the previous example the cause is function co, because its pattern $snd\ X$ is *opaque* and shadows the type of its subpattern X. Usual inference algorithms treat this opacity as polymorphism, and that is the reason why it is inferred a completely polymorphic type for the the result of the function co.

In [5] the appearance of any opaque pattern in the left-hand side of the rules is prohibited, but we will see that it is possible to be less restrictive. The key is

making a distinction between **opaque** and **transparent** variables of a pattern: a variable is opaque if its type is not univocally fixed by the type of the pattern, and is transparent otherwise. We call a variable of a pattern **critical** if it is opaque in the pattern and also appears elsewhere in the expression. The formal definition of opaque and critical variables will be given in Sect. 3. With these notions we can relax the situation in [5], prohibiting only those patterns having critical variables.

1.2 Local definitions

Functional and functional logic languages provide syntax to introduce local definitions inside an expression. But in spite of the popularity of let-expressions, different implementations treat them differently because of the polymorphism they give to bound variables. This differences can be observed in Example 2, being $(e_1, \ldots, e_n)$ and $[e_1, \ldots, e_n]$ the usual tuple and list notation respectively.

Example 2 (let expressions). Let e_1 be $let\ F = id\ in\ (F\ true, F\ 0)$, and e_2 be $let\ [F, G] = [id, id]\ in\ (F\ true, F\ 0, G\ 0, G\ false)$

Intuitively, e_1 gives a new name to the identity function and uses it twice with arguments of different types. Surprisingly, not all the implementations consider this expression as well-typed, and the reason is that F is used with different types in each appearance: $bool \to bool$ and $int \to int$. Some implementations as Clean 2.2, PAKCS 1.9.1 or KICS 0.81893 consider that a variable bound by a let-expression must be used with the same type in all the appearances in the body of the expression. In this situation we say that lets are completely monomorphic, and write let_m for it.

On the other hand, we can consider that all the variables bound by the let-expression may have different but coherent types, i.e., are treated polymorphically. Then expressions like e_1 or e_2 would be well-typed. This is the decision adopted by Hugs Sept 2006 or OCaml 3.10.2. In this case, we will say that lets are completely polymorphic, and write let_p.

Finally, we can treat the bound variables monomorphically or polymorphically depending on the form of the pattern. If the pattern is a variable, the let treats it polymorphically, but if it is compound the let treats all the variables monomorphically. This is the case of GHC 6.8.2, SML of New Jersey v110.67 or Curry Münster 0.9.11. In this implementations e_1 is well-typed, while e_2 not. We call this kind of let-expression let_{pm}.

Fig. 1 summarizes the decisions of various implementations of functional and functional logic languages. The exact behavior wrt. types of local definitions is usually not well documented, not to say formalized, in those system. One of our contributions is this paper is to technically clarify this question by adopting a neutral position, and formalizing the different possibilities for the polymorphism of local definitions.

Programming language and version	let_m	let_{pm}	let_p
GHC 6.8.2		×	
Hugs Sept. 2006			×
Standard ML of New Jersey 110.67		×	
Ocaml 3.10.2			×
Clean 2.0	×		
$\mathcal{TOY}$ 2.3.1*	×		
Curry PAKCS 1.9.1	×		
Curry Münster 0.9.11		×	
KICS 0.81893	×		

(*) we use `where` instead of `let`, not supported by $\mathcal{TOY}$

Fig. 1. Let expressions in different programming languages.

2 Preliminaries

We assume a signature $\Sigma = DC \cup FS$, where DC and FS are two disjoint set of *data constructor* and *function* symbols resp., all them with associated arity. We write DC^n (resp FS^n) for the set of constructor (function) symbols of arity n. We also assume a numerable set $\mathcal{DV}$ of *data variables* X. We define the set of *patterns* $Pat \ni t ::= X \mid f \mid c\ t_1 \dots t_n\ (n \leq m) \mid f\ t_1 \dots t_n (n < m)$ and the set of *expressions* $Exp \ni e ::= X \mid c \mid f \mid e_1\ e_2 \mid \lambda t.e \mid let_m\ t = e_1\ in\ e_2 \mid let_{pm}\ t = e_1\ in\ e_2 \mid let_p\ t = e_1\ in\ e_2$ where $c \in DC^m$ and $f \in FS^m$. We split the set of patterns in two: *first order patterns* $FOPat \ni fot ::= X \mid c\ t_1 \dots c_n$ where $c \in DC^n$, and *Higher order patterns* $HOPat = Pat \setminus FOPat$. Expressions $h\ e_1 \dots e_m$ are called *junk* if $h \in CS^n$ and $m > n$, and *active* if $h \in FS^n$ and $m \geq n$. $FV(e)$ is the set of variables in e which are not bound by any lambda or let expression and is defined in the usual way (notice that since our let expressions do not support recursive definitions the bindings of the pattern only affect e_2: $FV(let_*\ t = e_1\ in\ e_2) = FV(e_1) \cup (FV(e_2) \setminus var(t))$. A *one-hole context* $\mathcal{C}$ is an expression with exactly one hole. A *data substitution* $\theta \in \mathcal{PS}ubst$ is a finite mapping from data variables to patterns: $[\overline{X_i/t_i}]$. Substitution application over data variables and expressions is defined in the usual way. A *program rule* is defined as $PRule \ni r ::= f\ t_1 \dots t_n \to e\ (n \geq 0)$ where the set of patterns $\overline{t_i}$ is linear and $FV(e) \subseteq \bigcup_i FV(t_i)$. Therefore, extra variables are not considered in this paper. A program is a set of program rules $Prog \ni \mathcal{P} ::= \{r_1; \dots; r_n\} (n \geq 0)$.

For the types we assume a numerable set $\mathcal{TV}$ of *type variables* α and a countable alphabet $\mathcal{TC} = \bigcup_{n \in \mathbb{N}} \mathcal{TC}^n$ of *type constructors* C. The set of *simple types* is defined as $SType \ni \tau ::= \alpha \mid \tau_1 \to \tau_2 \mid C\ \tau_1 \dots \tau_n\ (C \in \mathcal{TC}^n)$. Based on simple types we can define the set of *type-schemes* as $TScheme \ni \sigma ::= \tau \mid \forall \alpha.\sigma$. The set of *free type variables* (FTV) of a simple type τ is $var(\tau)$, and for type-schemes $FTV(\forall \overline{\alpha_i}.\tau) = FTV(\tau) \setminus \{\overline{\alpha_i}\}$. A type-scheme $\forall \overline{\alpha_i}.\overline{\tau_n} \to \tau$ is *transparent* if $FTV(\overline{\tau_n}) \subseteq FTV(\tau)$. A *set of assumptions* $\mathcal{A}$ is $\{\overline{s_i : \sigma_i}\}$, where $s_i \in DC \cup FS \cup \mathcal{DV}$. Notice that the transparency of type-schemes for data constructors is not required in our setting, although that hypothesis is usually assumed in classical Damas & Milner type systems. If $(s_i : \sigma_i) \in \mathcal{A}$

we write $\mathcal{A}(s_i) = \sigma_i$. A *type substitution* $\pi \in \mathcal{TS}ubst$ is a finite mapping from type variables to simple types $[\overline{\alpha_i/\tau_i}]$. For sets of assumptions $FTV(\{\overline{s_i : \sigma_i}\}) = \bigcup_i FTV(\sigma_i)$. We will say a type-scheme σ is *closed* if $FTV(\sigma) = \emptyset$. The notion of applying a type substitution to a type variable or simple type is the natural, and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions in the obvious way. We will say σ is an *instance* of σ' if $\sigma = \sigma'\pi$ for some π. τ' is a *generic instance* of $\sigma \equiv \forall\overline{\alpha_i}.\tau$ if $\tau' = \tau[\overline{\alpha_i/\tau_i}]$ for some $\overline{\tau_i}$, and we write it $\sigma \succ \tau'$. We extend $\succ$ to a relation between type-schemes by saying that $\sigma \succ \sigma'$ iff every simple type such that is a generic instance of σ' is also a generic instance of σ. Then $\forall\overline{\alpha_i}.\tau \succ \forall\overline{\beta_i}.\tau[\overline{\alpha_i/\tau_i}]$ iff $\{\overline{\beta_i}\} \cap FTV(\forall\overline{\alpha_i}.\tau) = \emptyset$ [12]. Finally, τ' is a *variant* of $\sigma \equiv \forall\overline{\alpha_i}.\tau$ ($\sigma \succ_{var} \tau'$) if $\tau' = \tau[\overline{\alpha_i/\beta_i}]$ and $\overline{\beta_i}$ are fresh type variables.

3 Type derivation

We propose a modification of Damas & Milner type system [3] with some differences. We have found convenient to separate the task of giving a regular Damas & Milner type and the task of checking critical variables. To do that we have defined two different type relations: $\vdash$ and $\vdash^\bullet$.

The basic typing relation $\vdash$ in the upper part of Fig. 2 is like the classical Damas & Milner's system but extended to handle the three different kinds of let expressions and the occurrence of patterns instead of variables in lambda and let expressions. We have also made the rules more syntax-directed so that the form of type derivations depends only on the form of the expression to be typed. $Gen(\tau, \mathcal{A})$ is the clausure or generalization of τ wrt. $\mathcal{A}$ [3,12,18], which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall\overline{\alpha_i}.\tau$ *where* $\{\overline{\alpha_i}\} = FTV(\tau) \setminus FTV(\mathcal{A})$. As can be seen, $[\mathrm{LET}_m]$ and $[\mathrm{LET}^h_{pm}]$ behave the same, and do not generalize any of the types τ_i for the variables X_i to give a type for the body. On the contrary, $[\mathrm{LET}^X_{pm}]$ and $[\mathrm{LET}_p]$ generalize the types given to the variables. Notice that if two variables share the same type in the set of assumptions $\mathcal{A}$, generalization will lose the connection between them. This fact can be seen with e_2 in Ex. 2. Although the type for both F and G can be $\alpha \to \alpha$ (with α a variable not appearing in $\mathcal{A}$) the generalization step will assign both the type-scheme $\forall\alpha.\alpha \to \alpha$, losing the connection between them.

The $\vdash^\bullet$ relation (lower part of Fig. 2) uses $\vdash$ but enforces also the absence of critical variables. The characterization of an *opaque variable* is defined as follows. It states that a variable X_i is opaque in t when it is possible to build a type derivation for t where the type assumed for X_i contains type variables which do not occur in the type derived for the pattern.

Definition 1 (Opaque variable of t wrt. $\mathcal{A}$). *Let t be a pattern that admits type wrt. a given set of assumptions $\mathcal{A}$. We say that $X_i \in \overline{X_i} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\exists\overline{\tau_i}, \tau$ s.t. $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$ and $FTV(\tau_i) \not\subseteq FTV(\tau)$.*

The previous definition is based on the existence of a certain type derivation, and therefore cannot be used as an effective check for the opacity of variables. An equivalent characterization can be formulated exploiting the close relationship between $\vdash$ an type inference $\Vdash$ that will be presented in Sect. 4. Since $\Vdash$ can be viewed as an algorithm, Prop. 1 provides a more operational definition which is useful when implementing the type system.

$$
\begin{array}{lcl}
\textbf{[ID]} & \dfrac{}{\mathcal{A} \vdash s : \tau} & \text{if } \begin{array}{l} s \in DC \cup FS \cup \mathcal{DV} \\ \wedge\ (s : \sigma) \in \mathcal{A} \wedge\ \sigma \succ \tau \end{array} \\[2ex]
\textbf{[APP]} & \dfrac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1 e_2 : \tau} & \\[2ex]
[\Lambda] & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau} & \text{if } \{\overline{X_i}\} = var(t) \\[2ex]
\textbf{[LET}_m\textbf{]} & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_m\ t = e_1\ \textbf{in}\ e_2 : \tau_2} & \text{if } \{\overline{X_i}\} = var(t) \\[2ex]
\textbf{[LET}^X_{pm}\textbf{]} & \dfrac{\mathcal{A} \vdash e_1 : \tau_1 \quad \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A})\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_{pm}\ X = e_1\ \textbf{in}\ e_2 : \tau_2} & \\[2ex]
\textbf{[LET}^h_{pm}\textbf{]} & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash h\ t_1 \ldots t_n : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \textbf{in}\ e_2 : \tau_2} & \text{if } \{\overline{X_i}\} = var(t_1 \ldots t_n) \\[2ex]
\textbf{[LET}_p\textbf{]} & \dfrac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_i : Gen(\tau_i, \mathcal{A})}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_p\ t = e_1\ \textbf{in}\ e_2 : \tau_2} & \text{if } \{\overline{X_i}\} = var(t) \\[2ex]
\hline
\textbf{[P]} & \dfrac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash^{\bullet} e : \tau} & \text{if } critVar_{\mathcal{A}}(e) = \emptyset
\end{array}
$$

Fig. 2. Rules of type system

Proposition 1. *$X_i \in \overline{X_i} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_g | \pi_g$ and $FTV(\alpha_i \pi_g) \nsubseteq FTV(\tau_g)$.*

We write $opaqueVar_{\mathcal{A}}(t)$ for set of opaque variables of t wrt. $\mathcal{A}$. Now, we can define the *critical variables* of an expression e wrt. $\mathcal{A}$ as those variables that, being opaque in a let or lambda pattern of e, are indeed used in e. Formally:

Definition 2 (Critical variables).

$$critVar_{\mathcal{A}}(s) = \emptyset \qquad critVar_{\mathcal{A}}(e_1 e_2) = critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$$
$$critVar_{\mathcal{A}}(\lambda t.e) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e)) \cup critVar_{\mathcal{A}}(e)$$
$$critVar_{\mathcal{A}}(let_* \ t = e_1 \ in \ e_2)$$
$$= (opaqueVar_{\mathcal{A}}(t) \cap FV(e_2)) \cup \ critVar_{\mathcal{A}}(e_1) \cup critVar_{\mathcal{A}}(e_2)$$

The typing relation $\vdash^\bullet$ has been defined in a modular way in the sense that the opacity check is kept separated from the regular Damas & Milner typing. Therefore it is easy to see that if every constructor and function symbol in program has a transparent assumption, then all the variables in patterns will be transparent, and so $\vdash^\bullet$ will be equivalent to $\vdash$. This happens in particular for those programs using only first order patterns and whose constructor symbols come from a Haskell (or Toy, Curry)-like `data` declaration.

3.1 Properties of the typing relations

The typing relations fulfill a set of useful properties. Here we use $\vdash^?$ for any of the two typing relations: $\vdash$ or $\vdash^\bullet$.

Theorem 1 (Properties of the typing relations).

a) If $\mathcal{A} \vdash^? e : \tau$ *then* $\mathcal{A}\pi \vdash^? e : \tau\pi$

b) Let s *be a symbol which does not appear in* e*. Then* $\mathcal{A} \vdash^? e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash^? e : \tau$.

c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e[X/e'] : \tau$.

d) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$*, then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$.

Part *a)* states that type derivations are closed under type substitutions. *b)* shows that type derivations for e depend only on the assumptions for the symbols in e. *c)* is a substitution lemma stating that in a type derivation we can replace a variable by an expression with the same type. Finally, *d)* establishes that from a valid type derivation we can change the assumption of a symbol for a more general type-scheme, and we still have a correct type derivation for the same type. Notice that this is not true wrt. the typing relation $\vdash^\bullet$ because a more general type can introduce opacity. For example the variable X is opaque in *snd* X with the usual type for *snd*, but with a more specific type such as $bool \to bool \to bool$ it is no longer opaque.

3.2 Subject Reduction

Subject reduction is a key property for type systems, meaning that evaluation does not change the type of an expression. This ensures that run-time type errors will not occur. Subject reduction is only guaranteed for *well-typed* programs, a notion that we formally define now.

Definition 3 (Well-typed program). *A program rule $f\ t_1 \ldots t_n \to e$ is well-typed wrt. $\mathcal{A}$ if $\mathcal{A} \vdash^{\bullet} \lambda t_1 \ldots \lambda t_n.e : \tau$ and τ is a variant of $\mathcal{A}(f)$. A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ if all its rules are well-typed wrt. $\mathcal{A}$. If $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ we write $wt_{\mathcal{A}}(\mathcal{P})$.*

Notice the use of the extended typing relation $\vdash^{\bullet}$ in the previous definition. This is essential, as we will explain later.

Although the restriction that the type of the lambda abstraction associated to a rule must be a variant of the type of the function symbol (and not an instance) may be strange, it is necessary. If not, the fact that a program is well-typed will not give us important information about the functions like the type of their arguments, and will make us to consider as well-typed undesirable programs like $\mathcal{P} \equiv \{f\ true \to true; f\ 2 \to false\}$ with the assumptions $\mathcal{A} \equiv \{f :: \forall\alpha.\alpha \to bool\}$. Besides, this restriction is implicitly considered in [5].

$$
\begin{array}{rl}
TRL(s) = s, & \text{if } s \in DC \cup FS \cup \mathcal{DV} \\
TRL(e_1\ e_2) = & TRL(e_1)\ TRL(e_2) \\
TRL(let_K\ X = e_1\ in\ e_2) = & let_K\ X = TRL(e_1)\ in\ TRL(e_2), \text{ with } K \in \{m, p\} \\
TRL(let_{pm}\ X = e_1\ in\ e_2) = & let_p\ X = TRL(e_1)\ in\ TRL(e_2) \\
TRL(let_m\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) \\
TRL(let_{pm}\ t = e_1\ in\ e_2) = & let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) \\
TRL(let_p\ t = e_1\ in\ e_2) = & let_p\ Y = TRL(e_1)\ in\ \overline{let_p\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2)
\end{array}
$$

for $\{\overline{X_i}\} = var(t) \cap var(e_2)$, $f_{X_i} \in FS^1$ fresh defined by the rule $f_{X_i}\ t \to X_i$, $Y \in \mathcal{DV}$ fresh, t a non variable pattern and t' any pattern.

Fig. 3. Transformation rules of let expressions with patterns

For subject reduction to be meaningful, a notion of evaluation is needed. In this paper we consider the *let-rewriting* relation of [10]. As can be seen, *let-rewriting* does not support let expressions with compound patterns. Instead of extending the semantics with this feature we propose a transformation from let-expressions with patterns to let-expressions with only variables (Fig. 3). There are various ways to perform this transformation, which differ in the strictness of the pattern matching. We have chosen the alternative explained in [16] that does not demand the matching if no variable of the pattern is needed, but otherwise forces the matching of the whole pattern. This transformation has been enriched with the different kinds of let expressions in order to preserve the types, as is stated in Th. 2. Notice that the result of the transformation and the expressions accepted by *let-rewriting* only has let_m or let_p expressions, since without compound patterns let_{pm} is the same as let_p. Finally, we have added polymorphism annotations to let expressions (Fig. 4). Original **(Flat)** rule has been split into two, one for each kind of polymorphism. Although both behave the same from the point of view of values, the splitting is needed to guarantee type preservation. λ-abstractions have been omitted, since they are not supported by *let-rewriting*.

(Fapp) $f\ t_1\theta \ldots t_n\theta \ \rightarrow^l \ r\theta, \quad$ if $(f\ t_1 \ldots t_n \rightarrow r) \in \mathcal{P}$ and $\theta \in \mathcal{PS}ubst$

(LetIn) $e_1\ e_2 \rightarrow^l let_m\ X = e_2\ in\ e_1\ X, \quad$ if e_2 is an active expression, variable application, junk or *let* rooted expression, for X fresh.

(Bind) $let_K\ X = t\ in\ e\ \rightarrow^l\ e[X/t], \quad$ if $t \in Pat$

(Elim) $let_K\ X = e_1\ in\ e_2 \rightarrow^l e_2, \quad$ if $X \notin FV(e_2)$

(Flat$_m$) $let_m\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightarrow^l\ let_K\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3), \quad$ if $Y \notin FV(e_3)$

(Flat$_p$) $let_p\ X = (let_K\ Y = e_1\ in\ e_2)\ in\ e_3\ \rightarrow^l\ let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$ if $Y \notin FV(e_3)$

(LetAp) $(let_K\ X = e_1\ in\ e_2)\ e_3 \rightarrow^l let_K\ X = e_1\ in\ e_2\ e_3, \quad$ if $X \notin FV(e_3)$

(Contx) $\mathcal{C}[e] \rightarrow^l \mathcal{C}[e']$, if $\mathcal{C} \neq [\ ]$, $e \rightarrow^l e'$ using any of the previous rules

where $K \in \{m, p\}$

Fig. 4. Higher order *let*-rewriting relation $\rightarrow^l$

Theorem 2 (Type preservation of the let transformation). *Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_i}\ t_i \rightarrow X_i}\}$ be the rules of the projection functions needed in the transformation of e according to Fig. 3. Let also $\mathcal{A}'$ be the set of assumptions over that functions, defined as $\mathcal{A}' \equiv \{\overline{f_{X_i} : Gen(\tau_{X_i}, \mathcal{A})}\}$, where $\mathcal{A} \Vdash^\bullet \lambda t_i.X_i : \tau_{X_i}|\pi_{X_i}$. Then $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet TRL(e) : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.*

Th. 2 also states that the projection functions are well-typed. Then if we start from a well-typed program $\mathcal{P}$ wrt. $\mathcal{A}$ and apply the transformation to all its rules, the program extended with the projections rules will be well-typed wrt. the extended assumptions: $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P} \uplus \mathcal{P}')$. This result is straightforward, because $\mathcal{A}'$ does not contain any assumption for the symbols in $\mathcal{P}$, so $wt_{\mathcal{A}}(\mathcal{P})$ implies $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

Th. 3 states the subject reduction property for a *let-rewriting* step, but its extension to any number of steps is trivial.

Theorem 3 (Subject Reduction). *If $\mathcal{A} \vdash^\bullet e : \tau$ and $wt_{\mathcal{A}}(\mathcal{P})$ and $\mathcal{P} \vdash e \rightarrow^l e'$ then $\mathcal{A} \vdash^\bullet e' : \tau$.*

For this result to hold it is essential that the definition of well-typed program relies on $\vdash^\bullet$. A counterexample can be found in Ex. 1, where the program would be well-typed wrt. $\vdash$ but the subject reduction property fails for *and* (*cast* 0) *true* because of the rule for *co*.

The proof of the subject reduction property is based on the following Lemma, an important auxiliary result about the instantiation of transparent variables. Intuitively it states that if we have a pattern t with type τ and we change its variables by other expressions, the only way to obtain the same type τ for the substituted pattern is by changing the transparent variables for expressions with the same type. This is not guaranteed with opaque variables, and that is why we forbid their use in expressions.

Lemma 1. *Assume* $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$*, where* $var(t) \subseteq \{\overline{X_i}\}$*. If* $\mathcal{A} \vdash t[\overline{X_i/s_i}] : \tau$ *and* X_j *is a transparent variable of* t *wrt.* $\mathcal{A}$ *then* $\mathcal{A} \vdash s_j : \tau_j$*.*

4 Type inference for expressions

The typing relation $\vdash^\bullet$ lacks some properties that prevent its usage as a type-checker mechanism in a compiler for a functional logic language. First, in spite of the syntax-directed style, the rules for $\vdash$ and $\vdash^\bullet$ have a bad operational behavior: at some steps they need to guess a type. Second, the types related to an expression can be infinite due to polymorphism. Finally, the typing relation needs all the assumptions for the symbols in order to work. To overcome this problems, type systems usually are accompanied with a type inference algorithm which returns a valid type for an expression and also establish the types for some symbols in the expression.

In this work we have given the type inference in Fig. 5 a relational style to show the similarities with the typing relation. But in essence, the inference rules represent an algorithm (similar to algorithm $\mathcal{W}$ [3,12]) which fails if any of the rules cannot be applied. This algorithm accepts a set of assumptions $\mathcal{A}$ and an expression e, and returns a simple type τ and a type substitution π. Intuitively, τ will be the "most general" type which can be given to e, and π the "minimum" substitution we have to apply to $\mathcal{A}$ in order to able to derive a type for e.

Th. 4 shows that the type and substitution found by the inference are correct, i.e., we can build a type derivation for the same type if we apply the substitution to the assumptions.

Theorem 4 (Soundness of $\Vdash^?$). $\mathcal{A} \Vdash^? e : \tau | \pi \Longrightarrow \mathcal{A}\pi \vdash^? e : \tau$

Th. 5 expresses the completeness of the inference process. If we can derive a type for an expression applying a substitution to the assumptions, then inference will succeed and will find a type and a substitution which are more general.

Theorem 5 (Completeness of $\Vdash$ wrt $\vdash$). *If* $\mathcal{A}\pi' \vdash e : \tau'$ *then* $\exists \tau, \pi, \pi''.\ \mathcal{A} \Vdash e : \tau | \pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$*.*

A result similar to Th. 5 cannot be obtained for $\Vdash^\bullet$ because of critical variables, as Example 3 shows.

Example 3 (Inexistence of a more general typing substitution). Let $\mathcal{A} \equiv \{snd' :: \alpha \to bool \to bool\}$ and consider the following two valid derivations $\mathcal{D}_1 \equiv \mathcal{A}[\alpha/bool] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to bool$ and $\mathcal{D}_2 \equiv \mathcal{A}[\alpha/int] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to int$. It is clear that there is not a substitution more general than $[\alpha/bool]$ and $[\alpha/int]$ which makes possible a type derivation for $\lambda(snd'\ X).X$. The only substitution more general than these two will be $[\alpha/\beta]$ (for some β), converting X in a critical variable.

In spite of this, we will see that $\Vdash^\bullet$ is still able to find a more general substitution when it exists. To formalize that, we will use the notion of $\Pi^\bullet_{\mathcal{A},e}$, which denotes the set collecting all type substitution π such that $\mathcal{A}\pi$ gives some type to e.

[iID]
$$\frac{}{\mathcal{A} \Vdash s : \tau | id} \quad if \begin{array}{l} s \in DC \cup FS \cup \mathcal{DV} \\ \wedge\ (s:\sigma) \in \mathcal{A} \wedge \sigma \succ_{var} \tau \end{array}$$

[iAPP]
$$\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi | \pi_1\pi_2\pi} \quad if \begin{array}{l} \alpha\ fresh\ type\ variable \\ \wedge\ \pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha) \end{array}$$

[iΛ]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t \Vdash e : \tau | \pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau | \pi_t\pi} \quad if \begin{array}{l} \{\overline{X_i}\} = var(t) \\ \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \end{array}$$

[iLET$_m$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$^X_{pm}$]
$$\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ X = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_1\pi_2}$$

[iLET$^h_{pm}$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash h\ t_1 \ldots t_n : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_n = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ h \in DC \cup FS \wedge\ \{\overline{X_i}\} = var(h\ t_1 \ldots t_n) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iLET$_p$]
$$\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_i : Gen(\alpha_i\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
$$if\ \{\overline{X_i}\} = var(t) \wedge\ \overline{\alpha_i}\ fresh\ type\ variables \wedge\ \pi = mgu(\tau_t\pi_1, \tau_1)$$

[iP]
$$\frac{\mathcal{A} \Vdash e : \tau | \pi}{\mathcal{A} \Vdash^{\bullet} e : \tau | \pi} \qquad if\ critVar_{\mathcal{A}\pi}(e) = \emptyset$$

Fig. 5. Inference rules

Definition 4 (Typing substitutions of e).
$\Pi^{\bullet}_{\mathcal{A},e} = \{\pi \in \mathcal{TS}ubst \mid \exists \tau \in SType.\ \mathcal{A}\pi \vdash^{\bullet} e : \tau\}$

Now we are ready to formulate our result regarding the maximality of $\Vdash^{\bullet}$.

Theorem 6 (Maximality of $\Vdash^{\bullet}$).

a) *$\Pi^{\bullet}_{\mathcal{A},e}$ has a maximum element* $\iff \exists \tau_g \in SType, \pi_g \in \mathcal{TS}ubst.\mathcal{A} \Vdash^{\bullet} e : \tau_g|\pi_g$.

b) *If $\mathcal{A}\pi' \vdash^{\bullet} e : \tau'$ and $\mathcal{A} \Vdash^{\bullet} e : \tau|\pi$ then exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$.*

5 Type inference for programs

In the functional programming setting, type inference does not need to distinguish between programs and expressions, because the program can be incorporated in the expression by means of let expressions and λ-abstractions. This way, the results given for expressions are also valid for programs. But in our framework it is different, because our semantics (*let-rewriting*) does not support λ-abstractions and our let expressions do not define new functions but only perform pattern matching. Thereby in our case we need to provide an explicit method for inferring the types of a whole program. By doing so, we will also provide a specification closer to implementations.

The type inference procedure for a program takes a set of assumptions $\mathcal{A}$ and a program $\mathcal{P}$ and returns a type substitution π. The set $\mathcal{A}$ must contain assumptions for all the symbols in the program, even for the functions defined in $\mathcal{P}$. We want to reflect the fact that in practice some defined functions may come with an explicit type declaration. Indeed this is a frequent way of documenting a program. Furthermore, type declarations are sometimes a real need, for instance if we want the language to support *polymorphic recursion* [15,9]. Therefore, for some of the functions –those for which we want to infer types– the assumption will be simply a fresh type variable, to be instantiated by the inference process. For the rest, the assumption will be a closed type-scheme, to be checked by the procedure.

Definition 5 (Type Inference of a Program). *The procedure $\mathcal{B}$ for type inference of a program $\{rule_1, \ldots, rule_m\}$ is defined as:*

$$\mathcal{B}(\mathcal{A}, \{rule_1, \ldots, rule_m\}) = \pi, \textit{if}$$

1. $\mathcal{A} \Vdash^{\bullet} (\varphi(rule_1), \ldots, \varphi(rule_m)) : (\tau_1, \ldots, \tau_m)|\pi$.
2. *Let $f^1 \ldots f^k$ be the function symbols of the rules $rule_i$ in $\mathcal{P}$ such that $\mathcal{A}(f^i)$ is a closed type-scheme, and τ^i the type obtained for $rule_i$ in step 1. Then τ^i must be a variant of $\mathcal{A}(f^i)$.*

φ is a transformation from rules to expressions defined as:

$$\varphi(f\ t_1 \ldots t_n \to e) = pair\ \lambda t_1 \ldots . \lambda t_n . e\ f$$

where $()$ *is the usual tuple constructor, with type* $() : \forall\overline{\alpha_i}.\alpha_1 \to \ldots \alpha_m \to (\alpha_1, \ldots, \alpha_m)$*; and* ***pair*** *is a special constructor of tuples of two elements of the same type, with type* $\mathbf{pair} :: \forall\alpha.\alpha \to \alpha \to \alpha$*.*

The procedure $\mathcal{B}$ has two important properties. It is sound: if the procedure $\mathcal{B}$ finds a substitution π then the program $\mathcal{P}$ is well typed with respect to the assumptions $\mathcal{A}\pi$ (Th. 7). And second, if the procedure $\mathcal{B}$ succeeds it finds a more general typing substitution (Th. 8). It is not true in general that the existence of a well-typing substitution π' implies the existence of a more general one. A counterexample of this fact is very similar to Ex. 3.

Theorem 7 (Soundness of $\mathcal{B}$). *If* $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ *then* $wt_{\mathcal{A}\pi}(\mathcal{P})$*.*

Theorem 8 (Maximality of $\mathcal{B}$). *If* $wt_{\mathcal{A}\pi'}(\mathcal{P})$ *and* $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ *then* $\exists\pi''$ *such that* $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$*.*

Notice that types inferred for the functions are simple types. In order to obtain type-schemes we need and extra step of generalization, as discussed in the next section.

5.1 Stratified Inference of a Program

It is known that splitting a program into blocks of mutually recursive functions and inferring the types in order may reduce the need of providing explicit type-schemes. This situation is shown in Example 4.

Example 4 (Program Inference vs Stratified Inference).
$\mathcal{A} \equiv \{true : bool, 0 : int, id : \alpha, f : \beta, g : \gamma\}$
$\mathcal{P} \equiv \{id\ X \to X; f \to id\ true; g \to id\ 0\}$
$\mathcal{P}_1 \equiv \{id\ X \to X\}$, $\mathcal{P}_2 \equiv \{f \to id\ true\}$, $\mathcal{P}_3 \equiv \{g \to id\ 0\}$

An attempt to apply the procedure $\mathcal{B}$ to infer types for the whole program fails because it is not possible for id to have types $bool \to bool$ and $int \to int$ at the same time. We will need to provide explicitly the type-scheme for $id : \forall\alpha.\alpha \to \alpha$ in order to the type inference to succeed, yielding types $f : bool \to bool$ and $g : int \to int$. But this is not necessary if we first infer types for $\mathcal{P}_1$, obtaining $\delta \to \delta$ for id which will be generalized to $\forall\delta.\delta \to \delta$. With this assumption the type inference for both programs $\mathcal{P}_2$ and $\mathcal{P}_3$ will succeed with the expected types.

A general *stratified inference* procedure can be defined in terms of the basic inference $\mathcal{B}$. First, it calculates the graph of strongly connected components from the dependency graph of the program, using e.g. Kosaraju or Tarjan's algorithm [20]. Each strongly connected component will contain mutually dependent functions. Then it will infer types for every component (using $\mathcal{B}$) in topological order, generalizing the obtained types before following with the next component.

Although stratified inference needs less explicit type-schemes, programs involving polymorphic recursion still require explicit type-schemes in order to infer their types.

6 Conclusions and Future Work

In this paper we have proposed a type system for functional logic languages based on Damas & Milner type system. As far as we know, prior to our work only [5] treats with technical detail a type system for functional logic programming. Our paper makes clear contributions when compared to [5]:

- By introducing the notion critical variables, we are more liberal in the treatment of opaque variables, but still preserving the essential property of subject reduction; moreover, this liberality extends also to data constructors, dropping the traditional restriction of transparency required to them. This is somehow similar to what happens with *existential types* [13] or *generalized abstract datatypes* [8], a connection that we plan to further investigate in the future.
- Our type system considers local pattern bindings and λ-abstractions (also with patterns), that were missing in [5]. In addition to that, we have made a rather exhaustive analysis and formalization of different possibilities for polymorphism in local bindings.
- Subject reduction was proved in [5] wrt. a narrowing calculus. Here we do it wrt. an small-step operational semantics closer to real computations.
- In [5] programs came with explicit type declarations. Here we provide type inference algorithms where type declarations are optional.

We have in mind several lines for future work: apart from the relation to existential types mentioned above, we are interested in other known extensions of type system, like type classes or generic programming. We also want to generalize the subject reduction property to narrowing, using *let narrowing* reductions of [10], and taking into account known problems [5,1] in the interaction of HO narrowing and types. Handling extra variables (variables occurring only in right hand sides of rules) is another challenge from the viewpoint of types.

References

1. S. Antoy and A. P. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Fuji International Symposium on Functional and Logic Programming*, pages 335–353, 1999.
2. B. Brassel. Post to the curry mailing list. `http://www.informatik.uni-kiel.de/~{}curry/listarchive/0706.html`, May 2008.
3. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. Symposium on Principles of Programming Languages*, pages 207–212, 1982.
4. J. González-Moreno, M. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. International Conference on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.
5. J. González-Moreno, T. Hortalá-González, and Rodríguez-Artalejo, M. Polymorphic types in functional logic programming. In *Journal of Functional and Logic Programming*, volume 2001/S01, pages 1–71, 2001. Special issue of selected papers contributed to the International Symposium on Functional and Logic Programming (FLOPS'99).

6. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
7. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
8. S. L. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 50–61. ACM, 2006.
9. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.
10. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
11. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
12. L. M. Martins Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, April 1985. Also appeared as Technical report CST-33-85.
13. J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.
14. J. J. Moreno-Navarro, J. Mariño, A. del Pozo-Pietro, Á. Herranz-Nieva, and J. García-Martín. Adding type classes to functional-logic languages. In *1996 Joint Conf. on Declarative Programming, APPIA-GULP-PRODE'96*, pages 427–438, 1996.
15. A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228, London, UK, 1984. Springer-Verlag.
16. S. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.
17. B. P. Pierce. *Advanced topics in types and programming languages.* MIT Press, Cambridge, MA, USA, 2005.
18. C. Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.
19. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
20. R. Sedgewick. *Algorithms in C++, Part 5: Graph Algorithms*, chapter 19.8. Strong Components in Digraphs, pages 205–216. Addison-Wesley Professional, 2002.

A Proofs

Definition 6.
$\Pi_{\mathcal{A},e} = \{\pi \in \mathcal{TS}ubst \mid \exists \tau \in SType.\ \mathcal{A}\pi \vdash e : \tau\}$

Observation 1
Note that $\forall\overline{\alpha_i}.\tau = \forall\overline{\beta_i}.\tau[\overline{\alpha_i/\beta_i}]$ *if* $\{\overline{\beta_i}\} \cap FTV(\tau) = \emptyset$. *In other words, two different type-schemes are the same if we change the bounded variables for other variables which do not appear free in* τ. *For example,* $\forall\alpha,\beta.(\alpha,\beta) \to \alpha$ *is equal to* $\forall\gamma,\delta.(\gamma,\delta) \to \gamma$.

Observation 2
If $\sigma \succ \sigma'$ then $FTV(\sigma) \subseteq FTV(\sigma')$. It is clear from the definition of $\succ$. If α is a type variable in $FTV(\sigma)$ then it will not be affected by the substitution. Besides, α will be different from the generalized variables in σ'. Therefore $\alpha \in FTV(\sigma) \Longrightarrow \alpha \in FTV(\sigma')$, so $FTV(\sigma) \subseteq FTV(\sigma')$.

Observation 3
If $s \neq s'$ then $\mathcal{A} \oplus \{s : \sigma\} \oplus \{s' : \sigma'\}$ is the same as $\mathcal{A} \oplus \{s' : \sigma'\} \oplus \{s : \sigma\}$. This observation can be extended to sets of assumptions, in the sense that $\mathcal{A} \oplus \{\overline{X_i : \sigma_i}\} \oplus \{\overline{X'_j : \sigma'_j}\} = \mathcal{A} \oplus \{\overline{X'_j : \sigma'_j}\} \oplus \{\overline{X_i : \sigma_i}\}$ if $X_i \neq X'_j$ for all i and j.

Observation 4
If $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau$ then we can assume that $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash e : \tau' | \pi$ such that $\mathcal{A}\pi = \mathcal{A}$.

Proof (Explanation). Intuitively, the inference finds a type which is more general than all the possible types for an expression, and also a type substitution which is necessary applying to the set of assumptions in order to derive a type for the expression. In this case it is possible from the original set of assumptions $\mathcal{A}$ to derive a type, so we do not need to change $\mathcal{A}$. Therefore the type substitution π from the inference would not need to affect $\mathcal{A}$, just only $\overline{\alpha_i}$ and the fresh variables generated during inference.

By Theorem 5 we know that there exists a type substitution π'' such that $\mathcal{A}\pi\pi'' = \mathcal{A}$ and $\tau'\pi'' = \tau$. This means that $\mathcal{A}\pi$ is just a renaming of some free type variables of $\mathcal{A}$, which are restored with the type substitution π''. Being $\mathcal{A}\pi$ a renaming of $\mathcal{A}$ is a consequence of the *mgu* algorithm used. In this case, during inference there will be some unifying steps between a free type variable α from $\mathcal{A}$ and a fresh one β. Clearly, both $[\alpha/\beta]$ and $[\beta/\alpha]$ are more general unifiers. In this cases if we choose the first, we will compute a substitution which will make $\mathcal{A}\pi$ a renaming of $\mathcal{A}$; but if we choose always to substitute the fresh type variables the set of assumption $\mathcal{A}\pi$ will remain the same as $\mathcal{A}$.

Observation 5
If $FTV(\mathcal{A}) = FTV(\mathcal{A}')$ then $Gen(\tau, \mathcal{A}) = Gen(\tau, \mathcal{A}')$

Observation 6 (Uniqueness of the type inference)
The result of a type inference is unique upon renaming of fresh type variables. In a type inference $\mathcal{A} \Vdash e : \tau | \pi$ the variables in $FTV(\tau)$, $Dom(\pi)$ or $Rng(\pi)$ which do not occur in $FTV(\mathcal{A})$ are fresh variables generated by the inference process, so the result will remain valid if we replace them with different fresh types variables.

Observation 7
In a type derivation $\mathcal{A} \vdash e : \tau$ will appear a type derivation for every subexpression e' of e. That is, the derivation will have a part of the tree rooted by $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e' : \tau'$, being τ' a suitable type for e', and being $\{\overline{X_i : \tau_i}\}$ a set

of assumptions over variables of the expression e which have been introduced by the rules $[\Lambda]$, $[LET_m], [LET^X_{pm}], [LET^h_{pm}]$ *or* $[LET_p]$.
If the expression is a pattern, the set of assumptions $\{\overline{X_i : \tau_i}\}$ *will be empty because the only rules used to type a pattern are* $[ID]$ *and* $[APP]$.

Observation 8
If $wt_{\mathcal{A}}(\mathcal{P})$ *and* $\mathcal{A}'$ *is a set of assumptions for variables, then* $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.
The reason is that $\mathcal{A}'$ *does not change the assumptions for the function and constructor symbols in* $\mathcal{A}$. *Since there are not extra variables in the right hand sides, for every function rule in* $\mathcal{P}$ *the typing rule for the lambda expression will add assumptions for all the variables, shadowing the provided ones.*

Lemma 1
Assume $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$, *where* $var(t) \subseteq \{\overline{X_i}\}$. *If* $\mathcal{A} \vdash t[\overline{X_i/s_i}] : \tau$ *and* X_j *is a transparent variable of* t *wrt.* $\mathcal{A}$ *then* $\mathcal{A} \vdash s_j : \tau_j$.

Proof. According to Observation 7, in the derivation of $\mathcal{A} \vdash t[\overline{X_i/s_i}] : \tau$ appear derivations for every subpattern s_i, and they have the form $\mathcal{A} \vdash s_i : \tau'_i$ for some τ'_i. We will prove that if X_j is a particular transparent variable of t, then $\tau_j = \tau'_j$. It is easy to see that taking the types τ'_i as assumptions for the original variables X_i we can construct a derivation of $\mathcal{A} \oplus \{X_i : \tau'_i\} \vdash t : \tau$, simply replacing the derivations for the subpatterns $\mathcal{A} \vdash s_i : \tau'_i$ with derivations for the variables $\mathcal{A} \oplus \{X_i : \tau'_i\} \vdash X_i : \tau'_i$ in the original derivation for $\mathcal{A} \vdash t[\overline{X_i/s_i}] : \tau$. Since X_j is a transparent variable of t wrt $\mathcal{A}$, by definition $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_g|\pi_g$ and $FTV(\alpha_j\pi_g) \subseteq FTV(\tau_g)$. By Theorem 5, if any type for t can be derived from $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\}\pi_s$ then π_g must be more general than π_s. We know that there are (at least) two substitutions π^1 and π^2 which can type t: $\pi^1 \equiv \{\overline{\alpha_i \mapsto \tau_i}\}$ and $\pi^2 \equiv \{\overline{\alpha_i \mapsto \tau'_i}\}$, so they must be more specific than π_g (i.e. there exist π, π' such that $\pi^1 = \pi_g\pi$ and $\pi^2 = \pi_g\pi'$. We also know (by Theorem 4) that $\mathcal{A} \oplus \{X_i : \alpha_i\} \Vdash t : \tau_g|\pi_g$ implies $(\mathcal{A} \oplus \{X_i : \alpha_i\})\pi_g \vdash t : \tau_g$, and by Theorem 1-a this implies that $(\mathcal{A} \oplus \{X_i : \alpha_i\})\pi_g\pi \vdash t : \tau_g\pi$; so $\tau_g\pi = \tau$ (the same thing happens with π': $\tau_g\pi' = \tau$).

At this point we can distinguish two cases:

A) X_j is transparent because of $FTV(\alpha_j\pi_g) = \emptyset$. Then $\tau_j = (\alpha_j\pi_g)\pi = \alpha_j\pi_g = (\alpha_j\pi_g)\pi' = \tau'_j$, because if $\alpha_j\pi_g$ does not have any free variable, it cannot be affected by any substitution.
B) X_j is transparent because of $FTV(\alpha_j\pi_g) \subseteq FTV(\tau_g)$. As $\tau_g\pi = \tau$ and $\tau_g\pi' = \tau$, then for every type variable β in $FTV(\tau_g)$ then $\beta\pi = \beta\pi'$. As every type variable β in $FTV(\alpha_j\pi_g)$ is also in $FTV(\tau_g)$ then as $\tau_j = (\alpha_j\pi_g)\pi = (\alpha_j\pi_g)\pi' = \tau'_j$.

□

Lemma 2.
If $\mathcal{A}\pi \Vdash e : \tau_1|\pi_1$ *then* $\exists \tau_2 \in SType, \pi_2\ \pi'' \in \mathcal{TS}ubst$ *s.t.* $\mathcal{A} \Vdash e : \tau_2|\pi_2$ *and* $\tau_2\pi'' = \tau_1$ *and* $\mathcal{A}\pi_2\pi'' = \mathcal{A}\pi\pi_1$.

Proof. By Theorem 4 $\mathcal{A}(\pi\pi_1) \Vdash e : \tau_1$. Then applying Theorem 5 $\mathcal{A} \Vdash e : \tau_2|\pi_2$ and there exists a type substitution $\pi'' \in \mathcal{TS}ubst$ such that $\tau_2\pi'' = \tau_1$ and $\mathcal{A}\pi_2\pi'' = \mathcal{A}\pi\pi_1$.

Lemma 3 (Equivalence of the two characterizations of opaque variable).
Let t be a pattern that admits type wrt. a given set of assumptions $\mathcal{A}$. Then

$$\exists \overline{\tau_i}, \tau \text{ s.t. } \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau \text{ and } FTV(\tau_i) \not\subseteq FTV(\tau)$$
$$\Longleftrightarrow$$
$$\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_g|\pi_g \text{ and } FTV(\alpha_i\pi_g) \not\subseteq FTV(\tau_g)$$

Proof.

– $\Longrightarrow$) The type derivation can be written as $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})[\overline{\alpha_i/\tau_i}] \vdash t : \tau$, so by Theorem 5 $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_g|\pi_g$ and there exists some $\pi'' \in \mathcal{TS}ubst$ s.t. $\tau_g\pi'' = \tau$, $\mathcal{A}\pi_g\pi'' = \mathcal{A}$ and $\alpha_i\pi_g\pi'' = \tau_i$. We only need to prove that

$$FTV(\tau_i) \not\subseteq FTV(\tau) \Longrightarrow FTV(\alpha_i\pi_g) \not\subseteq FTV(\tau_g)$$

It is equivalent to prove

$$FTV(\alpha_i\pi_g) \subseteq FTV(\tau_g) \Longrightarrow FTV(\tau_i) \subseteq FTV(\tau)$$

which is trivial since $\alpha_i\pi_g\pi'' = \tau_i$ and $\tau_g\pi'' = \tau$, so

$$FTV(\alpha_i\pi_g) \subseteq FTV(\tau_g) \Longrightarrow FTV(\alpha_i\pi_g\pi'') \subseteq FTV(\tau_g\pi'')$$

– $\Longleftarrow$) By Theorem 4 $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_g \vdash t : \tau_g$, and $FTV(\alpha_i\pi_g) \not\subseteq FTV(\tau_g)$. Since t admits type by Observation 4 $\mathcal{A}\pi_g = \mathcal{A}$, so $\mathcal{A} \oplus \{\overline{X_i : \alpha_i\pi_g}\} \Vdash t : \tau_g$.

Lemma 4 (Decrease of opaque variables).
If $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau$ and $\mathcal{A}\pi \oplus \{\overline{X_i : \tau_i'}\} \vdash t : \tau'$ then $opaqueVar_{\mathcal{A}\pi}(t) \subseteq opaqueVar_{\mathcal{A}}(t)$.

Proof. Since $opaqueVar_{\mathcal{A}}(t) = var(t) \setminus transpVar_{\mathcal{A}}(e)$, then $opaqueVar_{\mathcal{A}\pi}(t) \subseteq opaqueVar_{\mathcal{A}}(t)$ is the same as $transpVar_{\mathcal{A}}(t) \subseteq transpVar_{\mathcal{A}\pi}(t)$. Then we have to prove that if a variable X_i of t is transparent wrt. $\mathcal{A}$ then it is also transparent wrt. $\mathcal{A}\pi$.

$\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}$ is the same as $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\}[\overline{\alpha_i/\tau_i}]$, so by Theorem 5 we have that $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_1|\pi_1$. Then the transparent variables of t will be those X_i such that $FTV(\alpha_i\pi_1) \subseteq FTV(\tau_1)$.

$\mathcal{A}\pi \oplus \{\overline{X_i : \tau_i'}\}$ is the same as $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi[\overline{\alpha_i/\tau_i'}]$, because we can assume that the variables $\overline{\alpha_i}$ does not appear in π. Then by Theorem 5 $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi \Vdash t : \tau_2|\pi_2$, and by Lemma 2 there exists a type substitution π'' such that $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi\pi_2 = (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_1\pi''$ and $\tau_2 = \tau_1\pi''$.

Therefore every data variable X_i which is transparent wrt. $\mathcal{A}$ will be also transparent wrt. $\mathcal{A}\pi$, because:

$$\begin{array}{rl} FTV(\alpha_i\pi_1) \subseteq FTV(\tau_1) & X_i \text{ is transparent wrt. } \mathcal{A} \\ FTV(\alpha_i\pi_1\pi'') \subseteq FTV(\tau_1\pi'') & \text{adding } \pi'' \text{ to both sides} \\ FTV(\alpha_i\pi\pi_2) \subseteq FTV(\tau_2) & X_i \text{ is transparent wrt. } \mathcal{A}\pi \end{array}$$

Lemma 5. *If $\mathcal{A} \vdash \mathcal{C}[e] : \tau$ and in that derivation appear a derivation of the form $\mathcal{A} \oplus \mathcal{A}' \vdash e : \tau'$, and $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau'$ then $\mathcal{A} \vdash \mathcal{C}[e'] : \tau$.*

Proof. We proceed by induction over the structure of the contexts:

[ ]) This case is straightforward because $[]e = e$ and $[]e' = e'$.

$\mathbf{e}_1\ \mathcal{C}$) Since $(e_1\ \mathcal{C})[e] = e_1\ \mathcal{C}[e]$, if we have a derivation for $\mathcal{A} \vdash (e_1\ \mathcal{C})[e]$ it must be of the form:

$$[\mathbf{APP}]\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash \mathcal{C}[e] : \tau_1}{\mathcal{A} \vdash e_1\ \mathcal{C}[e] : \tau}$$

A derivation of $\mathcal{A} \oplus \mathcal{A}' \vdash e : \tau'$ must appear in the whole derivation, so it must appear in the derivation $\mathcal{A} \vdash \mathcal{C}[e] : \tau_1$ (according to Observation 7). Since $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau'$ then by the Induction Hypothesis we can state that $\mathcal{A} \vdash \mathcal{C}[e'] : \tau_1$, and we can construct a derivation for $\mathcal{A} \vdash (e_1\ \mathcal{C})[e']$:

$$[\mathbf{APP}]\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash \mathcal{C}[e'] : \tau_1}{\mathcal{A} \vdash e_1\ \mathcal{C}[e'] : \tau}$$

$\mathcal{C}\ \mathbf{e}_1$) Similar to the previous case.

$\mathbf{let}_m\ \mathbf{X} = \mathcal{C}\ \mathbf{in}\ \mathbf{e}_1$) $(let_m\ X = \mathcal{C}\ in\ e_1)[e]$ is equal to $let_m\ X = \mathcal{C}[e]\ in\ e_1$, so a derivation of $\mathcal{A} \vdash (let_m\ X = \mathcal{C}\ in\ e_1)[e] : \tau$ must have the form:

$$[\mathbf{LET}_m]\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash \mathcal{C}[e] : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash e_1 : \tau}{\mathcal{A} \vdash let_m\ X = \mathcal{C}[e]\ in\ e_1 : \tau}$$

Clearly, a derivation for $\mathcal{A} \oplus \mathcal{A}' \vdash e : \tau'$ will appear in the derivation for $\mathcal{A} \vdash \mathcal{C}[e] : \tau_t$ (Observation 7). Since $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau'$ then by the Induction Hypothesis we can state that $\mathcal{A} \vdash \mathcal{C}[e'] : \tau_t$. With this information we can construct a derivation for (let_m X = $\mathcal{C}$ in e_1)[e']:

$$[\mathbf{LET}_m]\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash \mathcal{C}[e'] : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash e_1 : \tau}{\mathcal{A} \vdash let_m\ X = \mathcal{C}[e']\ in\ e_1 : \tau}$$

$\mathbf{let}_m\ \mathbf{X} = e_1\ \mathbf{in}\ \mathcal{C}$) A type derivation of ($\text{let}_m$ X = e_1 in $\mathcal{C}$)[e] will have the form:

$$[\mathbf{LET}_m]\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e] : \tau}{\mathcal{A} \vdash let_m\ X = e_1\ in\ \mathcal{C}[e] : \tau}$$

By Observation 7, the derivation $\mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e] : \tau$ will contain a derivation $(\mathcal{A} \oplus \{X : \tau_t\}) \oplus \mathcal{A}'' \vdash e : \tau'$. It is a premise that $(\mathcal{A} \oplus \{X : \tau_t\}) \oplus \mathcal{A}'' \vdash e' : \tau'$ (in this case $\mathcal{A}' = \{X : \tau_t\} \oplus \mathcal{A}''$), so by the Induction Hypothesis $\mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e'] : \tau$ and we can construct a derivation $\mathcal{A} \vdash let_m\ X = e_1\ in\ \mathcal{C}[e'] : \tau$

$$[\mathbf{LET}_m]\ \frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{X : \tau_t\} \vdash \mathcal{C}[e'] : \tau}{\mathcal{A} \vdash let_m\ X = e_1\ in\ \mathcal{C}[e'] : \tau}$$

rest) The proofs for the cases $let_{pm}\ X = \mathcal{C}\ in\ e_1$, $let_{pm}\ X = e_1\ in\ \mathcal{C}$, $let_p\ X = \mathcal{C}\ in\ e_1$ and $let_p\ X = e_1\ in\ \mathcal{C}$ are similar to the proofs for let_m.

□

Lemma 6.
If $critVar_{\mathcal{A}}(e) = \emptyset$ *and* $critVar_{\mathcal{A}}(e') = \emptyset$ *then* $critVar_{\mathcal{A}}(e[X/e']) = \emptyset$.

Proof. We will proceed by induction over the structure of e.

Base Case

- c) Straightforward because $c[X/e'] = c$, so $critVar_{\mathcal{A}}(c[X/e']) = \emptyset$.
- f) The same as c.
- X) In this case $X[X/e'] = e'$, and $critVar_{\mathcal{A}}(e') = \emptyset$ from the premises.
- Y) Y is a variable distinct from X. Then $Y[X/e'] = Y$, so $critVar_{\mathcal{A}}(Y) = \emptyset$.

Induction Step

- $e_1\ e_2$) By definition $critVar_{\mathcal{A}}(e_1\ e_2) = \emptyset$ implies that $critVar_{\mathcal{A}}(e_1) = \emptyset$ and $critVar_{\mathcal{A}}(e_2) = \emptyset$. Then by the Induction Hypothesis $critVar_{\mathcal{A}}(e_1[X/e']) = \emptyset$ and $critVar_{\mathcal{A}}(e_2[X/e']) = \emptyset$. By definition $(e_1\ e_2)[X/e'] = e_1[X/e']\ e_2[X/e']$, so:

$$\begin{aligned} critVar_{\mathcal{A}}((e_1\ e_2)[X/e']) &= critVar_{\mathcal{A}}(e_1[X/e']\ e_2[X/e']) \\ &= critVar_{\mathcal{A}}(e_1[X/e']) \cup critVar_{\mathcal{A}}(e_2[X/e']) \\ &= \emptyset \cup \emptyset \\ &= \emptyset \end{aligned}$$

- $\lambda t.e$) We assume that $X \notin var(t)$ and $var(t) \cap FV(e') = \emptyset$. We know that $opaqueVar_{\mathcal{A}}(t) \cap FV(e) = \emptyset$ and $critVar_{\mathcal{A}}(e) = \emptyset$ from $critVar_{\mathcal{A}}(\lambda t.e) = \emptyset$. Moreover $opaqueVar_{\mathcal{A}}(t) \subseteq var(t)$, so $opaqueVar_{\mathcal{A}}(t) \cap FV(e') = \emptyset$. Since the interection of set is distributive, we have that $opaqueVar_{\mathcal{A}}(t) \cap (FV(e) \cup FV(e')) = (opaqueVar_{\mathcal{A}}(t) \cap FV(e)) \cup (opaqueVar_{\mathcal{A}}(t) \cap FV(e')) = \emptyset$. Since $FV(e[X/e']) \subseteq FV(e) \cup FV(e')$, then $opaqueVar_{\mathcal{A}}(t) \cap FV(e[X/e']) = \emptyset$. On the other hand by the Induction Hypothesis $critVar_{\mathcal{A}}(e[X/e']) = \emptyset$. Therefore

$$\begin{aligned} critVar_{\mathcal{A}}((\lambda t.e)[X/e']) &= critVar_{\mathcal{A}}(\lambda t.(e[X/e'])) \\ &= (opaqueVar_{\mathcal{A}}(t) \cap FV(e[X/e'])) \cup critVar_{\mathcal{A}}(e[X/e']) \\ &= \emptyset \cup \emptyset \\ &= \emptyset \end{aligned}$$

- $let_m\ t = e_1\ in\ e_2$) We assume that $X \notin var(t)$, $var(t) \cap FV(e') = \emptyset$, and $var(t) \cap FV(e_1) = \emptyset$. Since $critVar_{\mathcal{A}}(let_m\ t = e_1\ in\ e_2) = \emptyset$ then $opaqueVar_{\mathcal{A}}(t) \cap FV(e_2) = \emptyset$, $critVar_{\mathcal{A}}(e_1) = \emptyset$ and $critVar_{\mathcal{A}}(e_2) = \emptyset$. From $var(t) \cap FV(e') = \emptyset$ and $opaqueVar_{\mathcal{A}}(t) \subseteq var(t)$ we know that

$opaqueVar_{\mathcal{A}}(t) \cap FV(e') = \emptyset$. As in the previous case, $opaqueVar_{\mathcal{A}}(t) \cap (FV(e_2) \cup FV(e')) = \emptyset$ and $FV(e_2[X/e']) \subseteq FV(e_2) \cup FV(e')$, therefore $opaqueVar_{\mathcal{A}}(t) \cap FV(e_2[X/e']) = \emptyset$.
On the other hand by the Induction Hypothesis $critVar_{\mathcal{A}}(e_1[X/e']) = \emptyset$ and $critVar_{\mathcal{A}}(e_2[X/e']) = \emptyset$. Therefore

$$\begin{aligned} critVar_{\mathcal{A}}((let_m\ t = e_1\ in\ e_2)[X/e']) &= critVar_{\mathcal{A}}(let_m\ t = e_1[X/e']\ in\ e_2[X/e']) \\ &= (opaqueVar_{\mathcal{A}}(t) \cap FV(e_2[X/e'])) \cup \\ &\quad critVar_{\mathcal{A}}(e_1[X/e'])\ \cup\ critVar_{\mathcal{A}}(e_2[X/e']) \\ &= \emptyset \cup \emptyset \cup \emptyset \\ &= \emptyset \end{aligned}$$

The proofs for the let_{pm} and let_p cases are equal to the let_m case.

Lemma 7.

Let $\mathcal{A}$ be a set of assumptions, τ a type and $\pi \in \mathcal{TS}ubst$ such that for every type variable α which appears in τ and does not appear in $FTV(\mathcal{A})$ then $\alpha \notin Dom(\pi)$ and $\alpha \notin Rng(\pi)$. Then $(Gen(\tau, \mathcal{A}))\pi = Gen(\tau\pi, \mathcal{A}\pi)$.

Proof. We will study what happens with a type variable α of τ in both cases (types that are not variables are not modified by the generalization step).

- $\alpha \in FTV(\tau)$ and $\alpha \in FTV(\mathcal{A})$. In this case it cannot be generalized in $Gen(\tau, \mathcal{A})$, so in $(Gen(\tau, \mathcal{A}))\pi$ it will be transformed into $\alpha\pi$. Because $\alpha \in FTV(\mathcal{A})$, then all the variables in $\alpha\pi$ are in $FTV(\mathcal{A}\pi)$ and they cannot be generalized. Therefore in $Gen(\tau\pi, \mathcal{A}\pi)$ α will also be transformed into $\alpha\pi$.
- $\alpha \in FTV(\tau)$ and $\alpha \notin FTV(\mathcal{A})$. In this case α will be generalized in $Gen(\tau, \mathcal{A})$, and as π does not affect a generalized variable, it will remain in $(Gen(\tau, \mathcal{A}))\pi$. Because α is not in $Dom(\pi)$, then $\alpha\pi = \alpha$. $\alpha \notin Rng(\pi)$ and $\alpha \notin FTV(\mathcal{A})$, so it cannot appear in $\mathcal{A}\pi$. Therefore α will also be generalized in $Gen(\tau\pi, \mathcal{A}\pi)$.

□

Lemma 8 (Generalization and substitutions).

$$Gen(\tau, \mathcal{A})\pi \succ Gen(\tau\pi, \mathcal{A}\pi)$$

Proof. It is clear that if a type variable α in τ is not generalized in $Gen(\tau, \mathcal{A})$ (because it occurs in $FTV(\mathcal{A})$), then in the first type-scheme it will appear as $\alpha\pi$. In the second type scheme it will also appear as $\alpha\pi$ because all the variables in $\alpha\pi$ will be in $\mathcal{A}\pi$ (as $\alpha \in FTV(\mathcal{A}.)$). Therefore in every generic instance of the two type-schemes this part will be the same. On the other hand, if a type variable α is generalized in $Gen(\tau, \mathcal{A})$ then it will also appear generalized in $Gen(\tau, \mathcal{A})\pi$ (π will not affect it). It does not matter what happens with this part $\alpha\pi$ in $Gen(\tau\pi, \mathcal{A}\pi)$ because in every generic instance of $Gen(\tau, \mathcal{A})\pi$ the generalized α will be able to adopt all the types of any generic instance of the part $\alpha\pi$ in $Gen(\tau\pi, \mathcal{A}\pi)$. □

Lemma 9.

If $\mathcal{A} \Vdash^{\bullet} e : \tau|\pi$ *then* $\Pi_{\mathcal{A},e} = \Pi^{\bullet}_{\mathcal{A},e}$.

Proof. From definition of $\Vdash^{\bullet}$ we know that $\mathcal{A} \Vdash^{\bullet} e : \tau|\pi$. We need to prove that $\Pi_{\mathcal{A},e} \subseteq \Pi^{\bullet}_{\mathcal{A},e}$ and $\Pi^{\bullet}_{\mathcal{A},e} \subseteq \Pi_{\mathcal{A},e}$.

- $\Pi_{\mathcal{A},e} \subseteq \Pi^{\bullet}_{\mathcal{A},e}$) We prove that $\pi' \in \Pi_{\mathcal{A},e} \Longrightarrow \pi' \in \Pi^{\bullet}_{\mathcal{A},e}$. If $\pi' \in \Pi_{\mathcal{A},e}$ then $\mathcal{A}\pi' \vdash e : \tau'$, and by Theorem 5 there exists π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$. By Theorem 4 $\mathcal{A}\pi \vdash^{\bullet} e : \tau$, and by Theorem 1-a $\mathcal{A}\pi\pi'' \vdash^{\bullet} e : \tau\pi''$, which is equal to $\mathcal{A}\pi' \vdash^{\bullet} e : \tau\pi''$; so $\pi' \in \Pi^{\bullet}_{\mathcal{A},e}$.
- $\Pi^{\bullet}_{\mathcal{A},e} \subseteq \Pi_{\mathcal{A},e}$) From definition of $\Pi^{\bullet}_{\mathcal{A},e}$

□

Lemma 10.
$\mathcal{A} \vdash^{\bullet} e_1 : \tau_i, \ldots, \mathcal{A} \vdash^{\bullet} e_n : \tau_n \Longleftrightarrow \mathcal{A} \vdash^{\bullet} (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n)$

Proof. Straightforward.

Theorem 1 (Properties of the typing relations).

a) If $\mathcal{A} \vdash^{?} e : \tau$ *then* $\mathcal{A}\pi \vdash^{?} e : \tau\pi$
b) Let s *be a symbol which does not appear in* e. *Then* $\mathcal{A} \vdash^{?} e : \tau \Longleftrightarrow \mathcal{A} \oplus \{s : \sigma_s\} \vdash^{?} e : \tau$.
c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash^{?} e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^{?} e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^{?} e[X/e'] : \tau$.
d) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$, *then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$.

Proof.
a.1) If $\mathcal{A} \vdash e : \tau$ *then* $\mathcal{A}\pi \vdash e : \tau\pi$
We prove it by induction over the size of the type derivation of $\mathcal{A} \vdash e : \tau$.

Base Case

- **[ID]** If we have a derivation of $\mathcal{A} \vdash s : \tau$ using **[ID]** is because τ is a generic instance of the type-scheme $\mathcal{A}(g) = \forall\overline{\alpha_i}.\tau'$. We can change this type-scheme by other equivalent $\forall\overline{\beta_i}.\tau''$ (according to Observation 1) where each variable β_i does not appear in $Dom(\pi)$ nor in $Rng(\pi)$. Then the generic instance τ will be of the form $\tau''[\overline{\beta_i/\tau_i}]$. We need to prove that $(\tau''[\overline{\beta_i/\tau_i}])\pi$ is a generic instance of $(\forall\overline{\beta_i}.\tau'')\pi$. Since π does not involve any variable β_i then $(\tau''[\overline{\beta_i/\tau_i}])\pi = \tau''\pi[\overline{\beta_i/\tau_i\pi}]$. Applying a substitution to a type-scheme is (by definition) applying it only to its free variables, but as no variable β_i appears in π then $(\forall\overline{\beta_i}.\tau'')\pi = \forall\overline{\beta_i}.(\tau''\pi)$. Then it is clear that $\tau''\pi[\overline{\beta_i/\tau_i\pi}]$ is a generic instance of $(\forall\overline{\beta_i}.\tau'')\pi$.

Induction Step
We have six different cases to consider accordingly to the inference rule used in the last step of the derivation.

– [**APP**] In this case we have a derivation

$$[\mathbf{APP}]\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

By the Induction Hypothesis $\mathcal{A}\pi \vdash e_1 : (\tau_1 \to \tau)\pi$ and $\mathcal{A}\pi \vdash e_2 : \tau_1\pi$. $(\tau_1 \to \tau)\pi \equiv \tau_1\pi \to \tau\pi$ so we can construct a derivation

$$[\mathbf{APP}]\ \frac{\mathcal{A}\pi \vdash e_1 : \tau_1\pi \to \tau\pi \qquad \mathcal{A}\pi \vdash e_2 : \tau_1\pi}{\mathcal{A} \vdash e_1\ e_2 : \tau\pi}$$

– [Λ] The derivation has the form

$$[\Lambda]\ \frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau}$$

By the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\})\pi \vdash \lambda t : \tau_t\pi$ and $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\})\pi \vdash e : \tau\pi$. But $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\})\pi \equiv \mathcal{A}\pi \oplus (\{\overline{X_i : \tau_i}\})\pi \equiv \mathcal{A}\pi \oplus \{\overline{X_i : \tau_i\pi}\}$ so we can build the type derivation

$$[\Lambda]\ \frac{\mathcal{A}\pi \oplus \{\overline{X_i : \tau_i\pi}\} \vdash t : \tau_t\pi \qquad \mathcal{A}\pi \oplus \{\overline{X_i : \tau_i\pi}\} \vdash e : \tau\pi}{\mathcal{A}\pi \vdash \lambda t.e : \tau_t \to \tau\pi}$$

– [$\mathbf{LET}_m$] The type derivation is

$$[\mathbf{LET}_m]\ \frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\})\pi \vdash t : \tau_t\pi$, $\mathcal{A}\pi \vdash e_1 : \tau_t\pi$ and $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\})\pi \vdash e_2 : \tau$. As in the previous case $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\})\pi \equiv \mathcal{A}\pi \oplus \{\overline{X_i : \tau_i\pi}\}$, so

$$[\mathbf{LET}_m]\ \frac{\begin{array}{c}\mathcal{A}\pi \oplus \{\overline{X_i : \tau_i\pi}\} \vdash t : \tau_t\pi \\ \mathcal{A}\pi \vdash e_1 : \tau_t\pi \\ \mathcal{A}\pi \oplus \{\overline{X_i : \tau_i\pi}\} \vdash e_2 : \tau\pi\end{array}}{\mathcal{A}\pi \vdash let_m\ t = e_1\ in\ e_2 : \tau\pi}$$

– [$\mathbf{LET}^X_{pm}$] The derivation will be

$$[\mathbf{LET}^X_{pm}]\ \frac{\mathcal{A} \vdash e_1 : \tau_x \qquad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash let^X_{pm}\ X = e_1\ in\ e_2 : \tau}$$

First, we create a substitution π' that maps the variables of τ_x which do not appear in $FTV(\mathcal{A})$ to fresh variables which are not in $FTV(\mathcal{A})$ and do not occur in $Dom(\pi)$ nor in $Rng(\pi)$. Then by the Induction Hypothesis $\mathcal{A}\pi' \vdash e_1 : \tau_x\pi'$. Since π' does not contain in its domain any variable in $FTV(\mathcal{A})$, then $\mathcal{A}\pi' = \mathcal{A}$ and $\mathcal{A} \vdash e_1 : \tau_x\pi'$. π' only substitutes variables which do not appear in $\mathcal{A}$ by variables which are not in $\mathcal{A}$ either, so $Gen(\tau_x, \mathcal{A}) = Gen(\tau_x\pi', \mathcal{A})$. Then $\mathcal{A} \oplus \{X : Gen(\tau_x\pi', \mathcal{A})\} \vdash e_2 : \tau$ is a valid derivation, and by the Induction Hypothesis $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi', \mathcal{A})\})\pi \vdash e_2 : \tau\pi$, which is the same that $\mathcal{A}\pi \oplus \{X : Gen(\tau_x\pi', \mathcal{A})\pi\} \vdash e_2 : \tau\pi$. By construction of π'

we know that for every variable of $\tau_x\pi'$ which does not appear in $\mathcal{A}$ it will not be in $Dom(\pi)$ nor in $Rng(\pi)$. Then we can apply Lemma 7 and we have that $\mathcal{A}\pi \oplus \{X : Gen(\tau_x\pi'\pi, \mathcal{A}\pi)\} \vdash e_2 : \tau\pi$. By the Induction Hypothesis over $\mathcal{A} \vdash e_1 : \tau_x\pi'$ we obtain $\mathcal{A}\pi \vdash e_1 : \tau_x\pi'\pi$. With this information we can construct a derivation

$$[\mathbf{LET}^X_{pm}]\ \frac{\mathcal{A}\pi \vdash e_1 : \tau_x\pi'\pi \qquad \mathcal{A}\pi \oplus \{X : Gen(\tau_x\pi'\pi, \mathcal{A}\pi)\} \vdash e_2 : \tau\pi}{\mathcal{A}\pi \vdash let^X_{pm}\ X = e_1\ in\ e_2 : \tau\pi}$$

- $[\mathbf{LET}^h_{pm}]$ Similar to the $[\mathrm{LET}_m]$ case.
- $[\mathbf{LET}_p]$ Similar to the $[\mathrm{LET}^X_{pm}]$ case, but instead of having to handle one single τ_x we need to handle a set of $\overline{\tau_i}$. The main idea is the same, creating a substitution π' to rename the variables of the $\overline{\tau_i}$ which do not appear in $\mathcal{A}$ and avoids their presence in the substitution π. Then we can apply Lemma 7 to all the generalizations and proceed as in the $[\mathrm{LET}^X_{pm}]$ case.

□

---

a.2) If $\mathcal{A} \vdash^\bullet e : \tau$ then $\mathcal{A}\pi \vdash^\bullet e : \tau\pi$

By definition of $\vdash^\bullet$ we know that $\mathcal{A} \vdash e : \tau$ and $critVar_{\mathcal{A}}(e) = \emptyset$. Then by Theorem 1-a $\mathcal{A}\pi \vdash e : \tau\pi$. To prove that $critVar_{\mathcal{A}\pi}(e) = \emptyset$ we use the decrease of opaque variables, stated in Lemma 4. From $\mathcal{A} \vdash e : \tau$ and $\mathcal{A}\pi \vdash e : \tau\pi$ we know that for every pattern t in e we have a derivation $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t$ and $\mathcal{A}\pi \oplus \{\overline{X_i : \tau'_i}\} \vdash t : \tau'$, being $\overline{X_i}$ the data variables in t. Then we can prove that $critVar_{\mathcal{A}\pi}(e) = \emptyset$ by induction over the structure of e.

Base Case

- s) $critVar_{\mathcal{A}\pi}(s) = \emptyset$ by definition.

Induction Step

- e_1e_2) By the Induction Hypothesis we have that $critVar_{\mathcal{A}\pi}(e_1) = \emptyset$ and $critVar_{\mathcal{A}\pi}(e_2) = \emptyset$, so $critVar_{\mathcal{A}\pi}(e_1e_2) = critVar_{\mathcal{A}\pi}(e_1) \cup critVar_{\mathcal{A}\pi}(e_2) = \emptyset \cup \emptyset = \emptyset$.
- $\lambda t.e$) By the Induction Hypothesis $critVar_{\mathcal{A}\pi}(e) = \emptyset$. $critVar_{\mathcal{A}}(t) = \emptyset$, so $(opaqueVar_{\mathcal{A}}(t) \cap var(t)) = \emptyset$. By Lemma 4 we know that $opaqueVar_{\mathcal{A}\pi}(t) \subseteq opaqueVar_{\mathcal{A}}(t)$, so $(opaqueVar_{\mathcal{A}\pi}(t) \cap var(t)) = \emptyset$. Then $critVar_{\mathcal{A}\pi}(\lambda t.e) = (opaqueVar_{\mathcal{A}\pi}(t) \cap var(t)) \cup critVar_{\mathcal{A}\pi}(e) = \emptyset \cup \emptyset = \emptyset$.
- $let_*\ t = e_1\ in\ e_2$) Similar to the previous case.

□

---

b.1)Let be s a symbol which does not appear in e. Then $\mathcal{A} \vdash e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$.

$\Longrightarrow$) We will proceed by induction over the size of the derivation tree.
Base Case

– **[ID]** In this case the derivation will be:

$$[\textbf{ID}]\frac{}{\mathcal{A} \vdash s : \tau}$$

where $\mathcal{A}(g) \succ \tau$. If we add an assumption over a symbol different from s then $(\mathcal{A} \oplus \{s : \sigma_s\})(g) \succ \tau$, so

$$[\textbf{ID}]\frac{}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash s : \tau}$$

Induction Step

– **[APP]** The derivation will have the form:

$$[\textbf{APP}]\frac{\mathcal{A} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \vdash e_2 : \tau'}{\mathcal{A} \vdash e_1 e_2 : \tau}$$

By the Induction Hypothesis then $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau' \to \tau$ and $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_2 : \tau'$, therefore:

$$[\textbf{APP}]\frac{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \oplus \{s : \sigma_s\} \vdash e_2 : \tau'}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 e_2 : \tau}$$

– $[\Lambda]$ We have a type derivation

$$[\Lambda]\frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau' \qquad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau' \to \tau}$$

By the Induction Hypothesis then $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma_s\} \vdash t : \tau'$ and $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma_s\} \vdash e : \tau$. s does not appear in $\lambda t.e$, so it will different from all the variables X_i and by Observation 3 $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma_s\}$ is the same as $(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_i : \tau_i}\}$. Therefore we can build a type derivation:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash \lambda t.e : \tau' \to \tau}$$

– $[\textbf{LET}_m]$ The type derivation will be:

$$[\textbf{LET}_m]\frac{\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \qquad \mathcal{A} \vdash e_1 : \tau_t \qquad \mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis then $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma_s\} \vdash t : \tau_t$, $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_t$ and $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma_s\} \vdash e : \tau$. As in the previous case $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma_s\} = (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_i : \tau_i}\}$, so we can build a type derivation:

$$[\textbf{LET}_m]\frac{\begin{array}{c}(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t \\ \mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_t \\ (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

– $[\mathbf{LET}^{X}_{pm}]$ The type derivation will be:

$$[\mathbf{LET}^{X}_{pm}]\frac{\mathcal{A} \vdash e_1 : \tau_x \qquad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

Here, $Gen(\tau_x, \mathcal{A})$ may be different from $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})$. This is caused because there are some type variables $\overline{\alpha_i}$ in $FTV(\tau_x)$ such that they appear free in $\mathcal{A}$ but not in $\mathcal{A} \oplus \{s : \sigma_s\}$ (they appear only in a previous assumption for s in $\mathcal{A}$) or because there are some type variables $\overline{\beta_i}$ in $FTV(\tau_x)$ such that they do not occur free in $\mathcal{A}$ but they do appear free in $\mathcal{A} \oplus \{s : \sigma_s\}$ (they are added by σ_s). The first group of variables will be generalized in $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma_s\})$ but not in $Gen(\tau_x, \mathcal{A})$. To handle the second group we can create a type substitution π from $\overline{\beta_i}$ to fresh type variables. This way $Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})$ will be a type-scheme more general than $Gen(\tau_x, \mathcal{A})$, and by Theorem 1-d then $\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\} \vdash e_2 : \tau$. By Theorem 1-a we obtain the derivation $\mathcal{A}\pi \vdash e_1 : \tau_x\pi$, and since $\overline{\beta_i}$ are not in $Dom(\pi)$ then $\mathcal{A} \vdash e_1 : \tau_x\pi$. By the Induction Hypothesis $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_x\pi$ and $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\}) \oplus \{s : \sigma_s\} \vdash e_2 : \tau$. As s is not in $let_m\ X = e_1\ in\ e_2$ then it is different from X, so $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\}) \oplus \{s : \sigma_s\}$ is equal to $(\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\}$.
Therefore we can build the type derivation:

$$[\mathbf{LET}^{X}_{pm}]\frac{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau_x\pi \qquad (\mathcal{A} \oplus \{s : \sigma_s\}) \oplus \{X : Gen(\tau_x\pi, \mathcal{A} \oplus \{s : \sigma_s\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

– $[\mathbf{LET}^{h}_{pm}]$ Similar to the $[\mathrm{LET}_m]$ case.
– $[\mathbf{LET}_p]$ Similar to the $[\mathrm{LET}^{X}_{pm}]$ case, creating a substitution π that solves the problem of the type variables which were generalized wrt. $\mathcal{A}$ but not wrt. $\mathcal{A} \oplus \{s : \sigma_s\}$.

$\Longleftarrow$) We will proceed again by induction over the size of the derivation tree.
Base Case
When the type derivation only applies the [ID] rule the proof is straightforward.

Induction Step

– **[APP]** The derivation will have the form:

$$[\mathbf{APP}]\frac{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \oplus \{s : \sigma_s\} \vdash e_2 : \tau'}{\mathcal{A} \oplus \{s : \sigma_s\} \vdash e_1 e_2 : \tau}$$

By the Induction Hypothesis then $\mathcal{A} \vdash e_1 : \tau' \to \tau$ and $\mathcal{A} \vdash e_2 : \tau'$, therefore:

$$[\mathbf{APP}]\frac{\mathcal{A} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \vdash e_2 : \tau'}{\mathcal{A} \vdash e_1 e_2 : \tau}$$

– [Λ] We have the type derivation:

$$[\Lambda]\frac{(\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{\overline{X_i:\tau_i}\}\vdash t:\tau' \qquad (\mathcal{A}\oplus\{s:\sigma_s\}))\oplus\{\overline{X_i:\tau_i}\}\vdash e:\tau}{\mathcal{A}\oplus\{s:\sigma_s\}\vdash \lambda t.e:\tau'\to\tau}$$

Since s is not in $\lambda t.e$, s will be different from all the variables $\overline{X_i}$ and $(\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{\overline{X_i:\tau_i}\}$ will be the same as $(\mathcal{A}\oplus\{\overline{X_i:\tau_i}\})\oplus\{s:\sigma_s\}$. Having $(\mathcal{A}\oplus\{\overline{X_i:\tau_i}\})\oplus\{s:\sigma_s\}\vdash t:\tau'$ and $(\mathcal{A}\oplus\{\overline{X_i:\tau_i}\})\oplus\{s:\sigma_s\}\vdash e:\tau$ we can apply the Induction Hypothesis and obtain $\mathcal{A}\oplus\{\overline{X_i:\tau_i}\}\vdash t:\tau'$ and $\mathcal{A}\oplus\{\overline{X_i:\tau_i}\}\vdash e:\tau$. With these two derivation we can build:

$$[\Lambda]\frac{\mathcal{A}\oplus\{\overline{X_i:\tau_i}\}\vdash t:\tau' \qquad \mathcal{A}\oplus\{\overline{X_i:\tau_i}\}\vdash e:\tau}{\mathcal{A}\vdash \lambda t.e:\tau'\to\tau}$$

– [$\mathbf{LET}_m$] Similar to the [Λ] case.
– [$\mathbf{LET}^X_{pm}$] This case has to deal with the same problems as in [LET^X_{pm}] of the $\Longrightarrow$) case. We have a type derivation:

$$[\mathbf{LET}^X_{pm}]\frac{\mathcal{A}\oplus\{s:\sigma_s\}\vdash e_1:\tau_x \qquad (\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{X:Gen(\tau_x,\mathcal{A}\oplus\{s:\sigma_s\})\}\vdash e_2:\tau}{\mathcal{A}\oplus\{s:\sigma_s\}\vdash let_{pm}\ X=e_1\ in\ e_2:\tau}$$

Again, the problem is that $Gen(\tau_x,\mathcal{A}\oplus\{s:\sigma_s\})$ may not be the same as $Gen(\tau_x,\mathcal{A})$. As before, there may be variables $\overline{\alpha_i}$ in $FTV(\tau_x)$ which appear free in $\mathcal{A}\oplus\{s:\sigma_s\}$ but not in $\mathcal{A}$, and variables $\overline{\beta_i}$ in $FTV(\tau_x)$ which do not occur free in $\mathcal{A}\oplus\{s:\sigma_s\}$ but they do appear free in $\mathcal{A}$. The first group is not problematic, because they are variables which will be generalized in $Gen(\tau_x,\mathcal{A})$ but not in $Gen(\tau_x,\mathcal{A}\oplus\{s:\sigma_s\})$. To solve the problem with the second group we create a type substitution π from $\overline{\beta}$ to fresh variables. This way $Gen(\tau_x\pi,\mathcal{A})$ will be a more general type-scheme than $Gen(\tau_x,\mathcal{A}\oplus\{s:\sigma_s\})$. Applying Theorem 1-d then $(\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{X:Gen(\tau_x\pi,\mathcal{A})\}\vdash e_2:\tau$. As s is different from X, then $(\mathcal{A}\oplus\{s:\sigma_s\})\oplus\{X:Gen(\tau_x\pi,\mathcal{A})\}$ is the same as $(\mathcal{A}\oplus\{X:Gen(\tau_x\pi,\mathcal{A})\})\oplus\{s:\sigma_s\}$, so the derivation $(\mathcal{A}\oplus\{X:Gen(\tau_x\pi,\mathcal{A})\})\oplus\{s:\sigma_s\}\vdash e_2:\tau$ is correct. Applying the Induction Hypothesis to this derivation we obtain $\mathcal{A}\oplus\{X:Gen(\tau_x\pi,\mathcal{A})\}\vdash e_2:\tau$. By Theorem 1-a $(\mathcal{A}\oplus\{s:\sigma_s\})\pi\vdash e_1:\tau_x\pi$, which is equal to $\mathcal{A}\oplus\{s:\sigma_s\pi\}\vdash e_1:\tau_x\pi$ because $\overline{\beta_i}$ do not occur free in $\mathcal{A}$. Applying the Induction Hypothesis to this derivation, we obtain $\mathcal{A}\vdash e_1:\tau_x\pi$. Therefore we can build the type derivation:

$$[\mathbf{LET}^X_{pm}]\frac{\mathcal{A}\vdash e_1:\tau_x\pi \qquad \mathcal{A}\oplus\{X:Gen(\tau_x\pi,\mathcal{A})\}\vdash e_2:\tau}{\mathcal{A}\vdash let_{pm}\ X=e_1\ in\ e_2:\tau}$$

– [$\mathbf{LET}^h_{pm}$] Similar to the [Λ] case.
– [$\mathbf{LET}_p$] Similar to the [LET^X_{pm}] case.

□

---

b.2) Let be s a symbol which does not appear in e, and σ_s any type. Then $\mathcal{A} \vdash^\bullet e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash^\bullet e : \tau$.

- $\Longrightarrow$) By definition of $\mathcal{A} \vdash^\bullet e : \tau$, $\mathcal{A} \vdash e : \tau$ and $critVar_{\mathcal{A}}(e) = \emptyset$. Since s does not occur in e by Theorem 1-b $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$. It will also be true that $critVar_{\mathcal{A}\oplus\{s:\sigma_s\}}(e) = \emptyset$ because the opaque variables in the patterns will not change by adding the new assumption, and neither the variables appearing in the rest of the expression. Therefore $\mathcal{A} \oplus \{s : \sigma_s\} \vdash^\bullet e : \tau$.
- $\Longleftarrow$) By definition of $\mathcal{A} \oplus \{s : \sigma_s\} \vdash^\bullet e : \tau$, $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$ and $critVar_{\mathcal{A}\oplus\{s:\sigma_s\}}(e) = \emptyset$. s does not appear in e, so by Theorem 1-b $\mathcal{A} \vdash e : \tau$. As in the previous case the critical variables of e will not change by deleting an assumption which is not used, so $\mathcal{A} \vdash^\bullet e : \tau$.

□

---

c.1) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$ and $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ then $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.

We will proceed by induction over the size of the expression e.

Base Case

- **[ID]** If $s \neq X$ then $s[X/e'] \equiv s$. On the contrary, if $s = X$ then the derivation will be:

$$[\textbf{ID}]\frac{}{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x}$$

$X[X/e'] \equiv e'$, and the type derivation $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ comes from the hypothesis.

Induction Step

- **[APP]** Just the application of the Induction Hypothesis.
- [Λ] We can assume that $\lambda t.e$ is such that the variables $\overline{X_i}$ in its pattern do not appear in $\mathcal{A} \oplus \{X : \tau_x\}$ nor in $FV(e')$. The derivation will have the form:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau' \quad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash \lambda t.e : \tau' \to \tau}$$

As X is different from $\overline{X_i}$ then $(\lambda t.e)[X/e'] \equiv \lambda t.(e[X/e'])$, so the first derivation remains the same. We have from the hypothesis that $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$. Since none of the $\overline{X_i}$ appear in e' then by Theorem 1-b we can

add assumptions over that variables and obtain a derivation $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_i : \tau_i}\} \vdash e' : \tau_x$. Because $X \neq X_i$ for all i then by Observation 3 $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_i : \tau_i}\}$ is the same as $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{X : \tau_x\}$. We have $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{X : \tau_x\} \vdash e : \tau$ and $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{X : \tau_x\} \vdash e' : \tau_x$, so applying the Induction Hypothesis we obtain $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$. Therefore we can build a new derivation:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau' \quad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{\overline{X_i : \tau_i}\} \vdash e[X/e'] : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash \lambda t.(e[X/e']) : \tau' \to \tau}$$

- [$\mathbf{LET}_m$] The proof is similar to the [Λ] case, provided that the variables of the pattern t do not occur in $FV(e')$ nor in $\mathcal{A} \oplus \{X : \tau_x\}$.
- [$\mathbf{LET}^X_{pm}$] In this case Y is a fresh variable. The type derivation will be:

$$[\mathbf{LET}^X_{pm}]\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1 : \tau_x \quad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash let_{pm}\ Y = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis $\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1[X/e'] : \tau_x$. $X \neq Y$ and $Y \notin FV(e')$, so by Theorem 1-b we can add an assumption over the variable Y and get a derivation $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e' : \tau_x$. By Observation 3 $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\}$ is equal to $(\mathcal{A} \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\}) \oplus \{X : \tau_x\}$, so by the Induction Hypothesis $(\mathcal{A} \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\}) \oplus \{X : \tau_x\} \vdash e_2[X/e'] : \tau$. Again by Observation 3 $(\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e_2[X/e'] : \tau$. Therefore we can construct a derivation:

$$[\mathbf{LET}^X_{pm}]\frac{\mathcal{A} \oplus \{X : \tau_x\} \vdash e_1[X/e'] : \tau_x \quad (\mathcal{A} \oplus \{X : \tau_x\}) \oplus \{Y : Gen(\tau_x, \mathcal{A} \oplus \{X : \tau_x\})\} \vdash e_2[X/e'] : \tau}{\mathcal{A} \oplus \{X : \tau_x\} \vdash let_{pm}\ Y = e_1[X/e']\ in\ e_2[X/e'] : \tau}$$

- [$\mathbf{LET}^h_{pm}$] Equal to the [LET_m] case.
- [$\mathbf{LET}_p$] The proof follows the same ideas as [LET_m] and [LET^X_{pm}].

□

---

c.2) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e : \tau$ *and* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e' : \tau_x$ *then* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e[X/e'] : \tau$.

From the definition of $\vdash^\bullet$ we know that $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$, $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$, $critVar_{\mathcal{A} \oplus \{X:\tau_x\}}(e) = \emptyset$ and $critVar_{\mathcal{A} \oplus \{X:\tau_x\}}(e') = \emptyset$. Then by Theorem 1-c $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$. By Lemma 6 we also know that $critVar_{\mathcal{A} \oplus \{X:\tau_x\}}(e[X/e']) = \emptyset$, so by definition $\mathcal{A} \oplus \{X : \tau_x\} \vdash^\bullet e[X/e'] : \tau$. □

---

d.1) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$, *then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$.
Base Case

– **[ID]** If $e \neq s$ then is trivial. If $e = s$ then the derivation will be:

$$[\textbf{ID}]\frac{}{\mathcal{A} \oplus \{s : \sigma\} \vdash s : \tau}$$

where $\sigma \succ \tau$. By Definition of generic instance, since $\sigma' \succ \sigma$ then $\sigma' \succ \tau$. So we can build the derivation:

$$[\textbf{ID}]\frac{}{\mathcal{A} \oplus \{s : \sigma'\} \vdash s : \tau}$$

Induction Step

– **[APP]** We have a type derivation:

$$[\textbf{APP}]\frac{\mathcal{A} \oplus \{s : \sigma\} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \oplus \{s : \sigma\} \vdash e_2 : \tau'}{\mathcal{A} \oplus \{s : \sigma\} \vdash e_1 e_2 : \tau}$$

By the Induction Hypothesis we have that $\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau' \to \tau$ and $\mathcal{A} \oplus \{s : \sigma'\} \vdash e_2 : \tau'$. Then we can construct a type derivation with the more general assumptions:

$$[\textbf{APP}]\frac{\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau' \to \tau \qquad \mathcal{A} \oplus \{s : \sigma'\} \vdash e_2 : \tau'}{\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 e_2 : \tau}$$

– $[\Lambda]$ We can assume that s is different from all the variables $\overline{X_i}$. The type derivation will be:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{s : \sigma\}) \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{s : \sigma\}) \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \oplus \{s : \sigma\} \vdash \lambda t.e : \tau' \to \tau}$$

Since s is different from the variables $\overline{X_i}$, then $(\mathcal{A} \oplus \{s : \sigma\}) \oplus \{\overline{X_i : \tau_i}\}$ is the same as $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma\}$. Therefore $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma\} \vdash t : \tau'$ and $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma\} \vdash e : \tau$. By the Induction Hypothesis we have that $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma'\} \vdash t : \tau'$ and $(\mathcal{A} \oplus \{\overline{X_i : \tau_i}\}) \oplus \{s : \sigma'\} \vdash e : \tau$; and changing again the order in the assumptions we can build a derivation:

$$[\Lambda]\frac{(\mathcal{A} \oplus \{s : \sigma'\}) \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau' \qquad (\mathcal{A} \oplus \{s : \sigma'\}) \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau}{\mathcal{A} \oplus \{s : \sigma'\} \vdash \lambda t.e : \tau' \to \tau}$$

– $[\textbf{LET}_m]$ The proof is similar to the $[\Lambda]$ case.
– $[\textbf{LET}^X_{pm}]$ We assume that $s \neq X$. The type derivation is:

$$[\textbf{LET}^X_{pm}]\frac{\begin{array}{c}\mathcal{A} \oplus \{s : \sigma\} \vdash e_1 : \tau_x \\ (\mathcal{A} \oplus \{s : \sigma\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma\})\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \oplus \{s : \sigma\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis we have $\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau_x$. As $\sigma' \succ \sigma$ then by Observation 2 $FTV(\sigma') \subseteq FTV(\sigma)$. Therefore $FTV(\mathcal{A} \oplus \{s : \sigma'\}) = FTV(\mathcal{A}_s) \cup FTV(\sigma') \subseteq FTV(\mathcal{A}_s) \cup FTV(\sigma) = FTV(\mathcal{A} \oplus \{s : \sigma\})$, being $\mathcal{A}_s$ the result of deleting from $\mathcal{A}$ all the assumptions for the symbol s. With this information it is clear that $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\}) \succ Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma\})$

because more variables could be generalized in $Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})$. Then by the Induction Hypothesis $(\mathcal{A} \oplus \{s : \sigma\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\} \vdash e_2 : \tau$. As $s \neq X$ then we can change the order of the assumptions and obtain a derivation $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\}) \oplus \{s : \sigma\} \vdash e_2 : \tau$. Again by the Induction Hypothesis $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\}) \oplus \{s : \sigma'\} \vdash e_2 : \tau$. With these derivations we can build the one we were trying to construct:

$$[\mathbf{LET}^X_{pm}] \frac{\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau_x \quad (\mathcal{A} \oplus \{s : \sigma'\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{s : \sigma'\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

- $[\mathbf{LET}^h_{pm}]$ Similar to the $[\Lambda]$ case.
- $[\mathbf{LET}_p]$ The proof is similar to the $[\mathrm{LET}^X_{pm}]$ case.

□

Theorem 2 (Type preservation of the let transformation).
Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_i}\ t_i \to X_i}\}$ be the rules of the projection functions needed in the transformation of e according to Fig. 3. Let also $\mathcal{A}'$ be the set of assumptions over that functions, defined as $\mathcal{A}' \equiv \{\overline{f_{X_i} : Gen(\tau_{X_i}, \mathcal{A})}\}$, where $\mathcal{A} \Vdash^\bullet \lambda t_i.X_i : \tau_{X_i} | \pi_{X_i}$. Then $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet TRL(e) : \tau$ and $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P})$.

Proof. By structural induction over the expression e.

Base Case

- s) Straightforward.

Induction Step

- $e_1\ e_2$) We have the type derivation:

$$[\mathbf{APP}] \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

Let be $\mathcal{A}^1$ and $\mathcal{A}^2$ the assumptions over the projection functions needed in e_1 and e_2 respectively. The by the Induction Hypothesis $\mathcal{A} \oplus \mathcal{A}^1 \vdash TRL(e_1)$ and $\mathcal{A} \oplus \mathcal{A}^2 \vdash TRL(e_2)$. Clearly the set of assumptions $\mathcal{A}'$ over the projection functions needed in the whole expression is $\mathcal{A}^1 \oplus \mathcal{A}^2$. Then by Theorem 1-b both derivations $\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1)$ and $\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_2)$ are valid, and we can construct the type derivation:

$$[\mathbf{APP}] \frac{\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1) : \tau_1 \to \tau \quad \mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_2) : \tau_1}{\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1)\ TRL(e_2) : \tau}$$

- $let_K\ X = e_1\ in\ e_2$) There are two cases, depending on the K:
  $let_m\ X = e_1\ in\ e_2$:
  The type derivation will be

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{X:\tau_t\}\vdash X:\tau_t\\ \mathcal{A}\vdash e_1:\tau_t\\ \mathcal{A}\oplus\{X:\tau_t\}\vdash e_2:\tau\end{array}}{\mathcal{A}\vdash let_m\ X=e_1\ in\ e_2:\tau}$$

By the Induction Hypothesis $\mathcal{A}\vdash TRL(e_1):\tau_t$ and $\mathcal{A}\oplus\{X:\tau_t\}\vdash TRL(e_2):\tau$. Then we can build the type derivation

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{X:\tau_t\}\vdash X:\tau_t\\ \mathcal{A}\vdash TRL(e_1):\tau_t\\ \mathcal{A}\oplus\{X:\tau_t\}\vdash TRL(e_2):\tau\end{array}}{\mathcal{A}\vdash let_m\ X=TRL(e_1)\ in\ TRL(e_2):\tau}$$

$let_p\ X=e_1\ in\ e_2$:
The type derivation for the original expression is

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{X:\tau_t\}\vdash X:\tau_t\\ \mathcal{A}\vdash e_1:\tau_t\\ \mathcal{A}\oplus\{X:Gen(\tau_t,\mathcal{A})\}\vdash e_2:\tau\end{array}}{\mathcal{A}\vdash let_p\ X=e_1\ in\ e_2:\tau}$$

By the Induction Hypothesis $\mathcal{A}\vdash TRL(e_1):\tau_t$ and $\mathcal{A}\oplus\{X:Gen(\tau_t,\mathcal{A})\}\vdash TRL(e_2):\tau$. Then we can build the type derivation

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{X:\tau_t\}\vdash X:\tau_t\\ \mathcal{A}\vdash TRL(e_1):\tau_t\\ \mathcal{A}\oplus\{X:Gen(\tau_t,\mathcal{A})\}\vdash TRL(e_2):\tau\end{array}}{\mathcal{A}\vdash let_p\ X=TRL(e_1)\ in\ TRL(e_2):\tau}$$

– $let_{pm}\ X=e_1\ in\ e_2$) The type derivation for the original expression is

$$[\mathbf{LET}_{pm}]\frac{\begin{array}{c}\mathcal{A}\vdash e_1:\tau_t\\ \mathcal{A}\oplus\{X:Gen(\tau_t,\mathcal{A})\}\vdash e_2:\tau\end{array}}{\mathcal{A}\vdash let_{pm}\ X=e_1\ in\ e_2:\tau}$$

By the Induction Hypothesis $\mathcal{A}\vdash TRL(e_1):\tau_t$ and $\mathcal{A}\oplus\{X:Gen(\tau_t,\mathcal{A})\}\vdash TRL(e_2):\tau$. The type derivation $\mathcal{A}\oplus\{X:\tau_t\}\vdash X:\tau_t$ is trivial, so we can build the type derivation

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{X:\tau_t\}\vdash X:\tau_t\\ \mathcal{A}\vdash TRL(e_1):\tau_t\\ \mathcal{A}\oplus\{X:Gen(\tau_t,\mathcal{A})\}\vdash TRL(e_2):\tau\end{array}}{\mathcal{A}\vdash let_p\ X=TRL(e_1)\ in\ TRL(e_2):\tau}$$

– $let_m\ t=e_1\ in\ e_2$) In this case the original type derivation is:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{\overline{X_i:\tau_i}\}\vdash t:\tau_t\\ \mathcal{A}\vdash e_1:\tau_t\\ \mathcal{A}\oplus\{\overline{X_i:\tau_i}\}\vdash e_2:\tau\end{array}}{\mathcal{A}\vdash let_m\ t=e_1\ in\ e_2:\tau}$$

It is easy to see that if $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau_t$ then $\mathcal{A} \vdash \lambda t.X_i : \tau_t \to \tau_i$. The assumptions over the projections functions in $\mathcal{A}'$ will be $\{\overline{f_{X_i} : Gen(\tau'_t \to \tau'_i, \mathcal{A})}\}$, where $\mathcal{A} \Vdash \lambda t.X_i : \tau'_t \to \tau'_i | \pi_{X_i}$. Since $\mathcal{A} \vdash \lambda t.X_i : \tau_t \to \tau_i$ we can assume that $\mathcal{A}\pi_{X_i} = \mathcal{A}$ (Observation 4), and by Theorem 5 we know that exists a type substitution π such that $\mathcal{A}\pi_{X_i}\pi = \mathcal{A}\pi = \mathcal{A}$ and $(\tau'_t \to \tau'_i)\pi = \tau_t \to \tau_i$. Therefore we can be sure that $Gen(\tau'_t \to \tau'_i, \mathcal{A}) \succ \tau_t \to \tau_i$, because π substitutes only the type variables in $\tau'_t \to \tau'_i$ which are generalized in $Gen(\tau'_t \to \tau'_i, \mathcal{A})$. If $\mathcal{A}'$ contains all the assumptions over the projection functions needed in the whole expression, it will contains assumptions over projection functions needed in e_1 ($\mathcal{A}^1$), e_2 ($\mathcal{A}^2$) and the pattern t ($\mathcal{A}^t \equiv \{\overline{f_{X_i} : Gen(\tau'_t \to \tau'_i, \mathcal{A})}\}$); so $\mathcal{A}' = \mathcal{A}^1 \oplus \mathcal{A}^2 \oplus \mathcal{A}^t$. Then we can build the type derivation:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\} \vdash Y : \tau_t \\ \mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1) : \tau_t \\ \mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau\end{array}}{\mathcal{A} \oplus \mathcal{A}' \vdash let_m\ Y = TRL(e_1)\ in\ \overline{let_m\ X_i = f_{X_i}\ Y}\ in\ TRL(e_2) : \tau}$$

where the derivation $\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau$ is

$$[\mathbf{LET}_m]\frac{[\mathbf{APP}]\ \dfrac{\mathcal{A}_Y \vdash f_{X_1} : \tau_t \to \tau_1 \quad \mathcal{A}_Y \vdash Y : \tau_t}{\mathcal{A}_Y \vdash f_{X_1}\ Y : \tau_1} \qquad [\mathbf{LET}_m]\ \dfrac{[\mathbf{ID}]\ \dfrac{}{\mathcal{A}_Y \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1} \quad \dfrac{\mathcal{A}_Y \oplus \{\overline{X_i : \tau_i}\} \vdash TRL(e_2) : \tau}{\ldots}}{\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2) : \tau}}{}$$

(being $\mathcal{A}_Y \equiv \mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\}$).
$\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\} \vdash Y : \tau_t$ and $\mathcal{A}_Y \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1$ are just the application of **[ID]** rule. By the Induction Hypothesis $\mathcal{A} \oplus \mathcal{A}^1 \vdash TRL(e_1) : \tau_t$, and by Theorem 1-b we can add the assumptions $\mathcal{A}^2 \oplus \mathcal{A}^t$, obtaining $\mathcal{A} \oplus \mathcal{A}' \vdash TRL(e_1) : \tau_t$. $\mathcal{A}_Y \vdash f_{X_1}\ Y : \tau_1$ is straightforward because $Gen(\tau'_t \to \tau'_i, \mathcal{A}\pi_{X_i}) \succ \tau_t \to \tau_i$ for all the projection functions. It is easy to see that this way the chain of let expressions will "collect" the same assumptions for the variables $\overline{X_i}$ that are introduced by the pattern in the original expression: $\{\overline{X_i : \tau_i}\}$. Then by the Induction Hypothesis $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \oplus \mathcal{A}^2 \vdash TRL(e_2) : \tau$, and by Theorem 1-b we can add the rest of the assumptions and obtain $\mathcal{A} \oplus \{\overline{X_i : \tau_i}\} \oplus \mathcal{A}^2 \oplus \mathcal{A}^1 \oplus \mathcal{A}^t \oplus \{Y : \tau_t\} \vdash TRL(e_2) : \tau$. Reorganizing the set of assumptions (since the symbols are all different), we obtain $\mathcal{A}_Y \oplus \{\overline{X_i : \tau_i}\} \vdash TRL(e_2) : \tau$.

- $let_{pm}\ t = e_1\ in\ e_2$) This case is equal to the previous one because the derivation of the original expression in both cases is the same (as t is a pattern we use $[\mathbf{LET}^h_{pm}]$, and this rule acts equal to $[\mathbf{LET}_m]$) and the transformed expressions are the same.
- $let_p\ t = e_1\ in\ e_2$) The type derivation will be:

$$[\mathbf{LET}_p]\frac{\begin{array}{c}\mathcal{A}\oplus\{\overline{X_i:\tau_i}\}\vdash t:\tau_t\\ \mathcal{A}\vdash e_1:\tau_t\\ \mathcal{A}\oplus\{\overline{X_i:Gen(\tau_i,\mathcal{A})}\}\vdash e_2:\tau\end{array}}{\mathcal{A}\vdash let_p\ t=e_1\ in\ e_2:\tau}$$

As in the previous case, $\mathcal{A}'$ will be $\{\overline{f_{X_i}:Gen(\tau'_t\to\tau'_i,\mathcal{A}\pi_{X_i})}\}$, where $\mathcal{A}\Vdash\lambda t.X_i:\tau'_t\to\tau'_i|\pi_{X_i}$. In addition, $\mathcal{A}\pi_{X_i}=\mathcal{A}$ (by the Observation 4), $Gen(\tau'_t\to\tau'_i,\mathcal{A})\succ\tau_t\to\tau_i$ and $\mathcal{A}'\equiv\mathcal{A}^1\oplus\mathcal{A}^2\oplus\mathcal{A}^t$. Then we can build a type derivation:

$$[\mathbf{LET}_p]\frac{\begin{array}{c}\mathcal{A}\oplus\mathcal{A}'\oplus\{Y:\tau_t\}\vdash Y:\tau_t\\ \mathcal{A}\oplus\mathcal{A}'\vdash TRL(e_1):\tau_t\\ \mathcal{A}'_1\vdash let_m\ X_1=f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2):\tau\end{array}}{\mathcal{A}\oplus\mathcal{A}'\vdash let_p\ Y=TRL(e_1)\ in\ \overline{let_p\ X_i=f_{X_i}\ Y}\ in\ TRL(e_2):\tau}$$

where the derivation $\mathcal{A}_Y\vdash let_m\ X_1=f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2):\tau$ is

$$[\mathbf{LET}_p]\frac{\begin{array}{c}[\mathbf{ID}]\ \dfrac{}{\mathcal{A}'_1\oplus\{X_1:\tau_1\}\vdash X_1:\tau_1}\\ [\mathbf{APP}]\ \dfrac{\begin{array}{c}\mathcal{A}'_1\vdash f_{X_1}:\tau_t\to\tau_1\\ \mathcal{A}'_1\vdash Y:\tau_t\end{array}}{\mathcal{A}'_1\vdash f_{X_1}\ Y:\tau_1}\end{array}\quad [\mathbf{LET}_p]\ \dfrac{\begin{array}{c}\mathcal{A}'_{n+1}\vdash TRL(e_2):\tau\\ \ldots\end{array}}{}}{\mathcal{A}'_1\vdash let_p\ X_1=f_{X_1}\ Y\ in\ \ldots\ in\ TRL(e_2):\tau}$$

being $\mathcal{A}'_1\equiv\mathcal{A}\oplus\mathcal{A}'\oplus\{Y:Gen(\tau_t,\mathcal{A}\oplus\mathcal{A}')\}$ and $\mathcal{A}'_i\equiv\mathcal{A}'_{i-1}\oplus\{X_{i-1}:Gen(\tau_{i-1},\mathcal{A}'_{i-1})\}$.
As in the previous case, all the derivations $\mathcal{A}'_i\vdash f_{X_i}\ Y:\tau_i$ are valid, because $\mathcal{A}'_i\vdash Y:\tau_t$. Notice that $Gen(\tau_t,\mathcal{A})=Gen(\tau_t,\mathcal{A}\oplus\mathcal{A}')$, as Observation 5 states, since $FTV(\mathcal{A})=FTV(\mathcal{A}\oplus\mathcal{A}')$. For the same reason, $Gen(\tau_i,\mathcal{A})=Gen(\tau_i,\mathcal{A}'_i)$, so the chain of let expressions will collect the same set of assumptions over the variables $\overline{X_i}$: $\{\overline{X_i:Gen(\tau_i,\mathcal{A})}\}$. By the Induction Hypothesis, we know that $\mathcal{A}\oplus\{\overline{X_i:Gen(\tau_i,\mathcal{A})}\}\oplus\mathcal{A}^2\vdash TRL(e_2):\tau$; and by Theorem 1-b we can add the assumptions $\mathcal{A}^1\oplus\mathcal{A}^t\oplus\{Y:Gen(\tau_t,\mathcal{A}\oplus\mathcal{A}')$ and obtain $\mathcal{A}\oplus\{\overline{X_i:Gen(\tau_i,\mathcal{A})}\}\oplus\mathcal{A}^2\oplus\mathcal{A}^1\oplus\mathcal{A}^t\oplus\{Y:Gen(\tau_t,\mathcal{A}\oplus\mathcal{A}')\}\vdash TRL(e_2):\tau$. Then reorganizing the assumptions we obtain $\mathcal{A}\oplus\mathcal{A}'\oplus\{Y:Gen(\tau_t,\mathcal{A}\oplus\mathcal{A}')\}\oplus\{\overline{X_i:Gen(\tau_i,\mathcal{A})}\}\vdash TRL(e_2):\tau$. Since $Gen(\tau_i,\mathcal{A})=Gen(\tau_i,\mathcal{A}'_i)$ then the previous derivation is equal to $\mathcal{A}'_{n+1}\vdash TRL(e_2):\tau$.

In all the cases it is true that $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$. Let X_i a data variable which is projected in the transformed expression, and t_i the compound pattern of a let expression where it appears. By Observation 7 we know that in the derivation $\mathcal{A}\vdash^\bullet e:\tau$ will appear a derivation $\mathcal{A}\oplus\mathcal{A}''\oplus\{X_i:\tau'_{X_i}\}\vdash t_i:\tau_i$ for a set of assumptions $\mathcal{A}''$ over some variables and X_i will not be opaque in t_i wrt. $\mathcal{A}\oplus\mathcal{A}''\oplus\{X_i:\tau'_{X_i}\}$. Then it is clear that $\mathcal{A}\vdash\lambda t_i.X_i:\tau_i\to\tau'_{X_i}$, and by Theorem 5 the type inference $\mathcal{A}\Vdash\lambda t_i.X_i:\tau_{X_i}|\pi_{X_i}$ will be correct. By Theorem 4 $\mathcal{A}\pi_{X_i}\vdash\lambda t_i.X_i:\tau_{X_i}$, and since by Observation 3 $\mathcal{A}\pi_{X_i}=\mathcal{A}$, then $\mathcal{A}\vdash\lambda t_i.X_i:$

τ_{X_i} is a valid derivation. Clearly X_i is not opaque in t_i wrt. $\mathcal{A}$, because only the assumptions for non variable symbols are used. Then $critVar_{\mathcal{A}}(\lambda t_i.X_i) = \emptyset$, so $\mathcal{A} \Vdash^{\bullet} \lambda t_i.X_i : \tau_{X_i} | \pi_{X_i}$ and $\mathcal{A} \vdash^{\bullet} \lambda t_i.X_i : \tau_{X_i}$. $\mathcal{A}'$ contains assumptions over projection functions, and they do not appear in $\lambda t_i.X_i$, so by Theorem 1-*b)* we can add these assumptions and obtain $\mathcal{A} \oplus \mathcal{A}' \vdash^{\bullet} \lambda t_i.X_i : \tau_{X_i}$. We know that in $\mathcal{A}'$ there will appear an assumption $\{f_{X_i} : Gen(\tau_{X_i}, \mathcal{A})\}$ for the projection function of the variable X_i, with rule $f_{X_i}\ t_i \to X_i$. We know that $FTV(\mathcal{A}) = FTV(\mathcal{A} \oplus \mathcal{A}')$ because since all the assumptions in $\mathcal{A}$ are of the form $Gen(\tau_{X_i}, \mathcal{A})$ they will not add any type variable, and since no f_{X_i} appears in $\mathcal{A}$ they will not shadow any assumption. Then τ_{X_i} will be a variant of $Gen(\tau_{X_i}, \mathcal{A})$.

Therefore for every data variable X_i which is projected then $\mathcal{A} \vdash \lambda t_i.X_i : \tau_{X_i}$ and τ_{X_i} is a variant of $\mathcal{A} \oplus \mathcal{A}'(f_{X_i}) = Gen(\tau_{X_i}, \mathcal{A})$, so all the program rules $f_{X_i}\ t_i \to X_i \in \mathcal{P}'$ are well-typed wrt. $\mathcal{A} \oplus \mathcal{A}'$ and $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P}')$.

Theorem 3 (Subject Reduction wrt $\vdash$).
If $\mathcal{A} \vdash e : \tau$ and $wt_{\mathcal{A}}(\mathcal{P})$ and $\mathcal{P} \vdash e \to^{l} e'$ then $\mathcal{A} \vdash e' : \tau$.

Proof. We proceed by case distinction over the rule of the let-rewriting relation $\to^{l}$ (Fig. 4) that we use to reduce e to e'.

- **(Fapp)** If we reduce an expression e using the **(Fapp)** rule is because e has the form $f\ t_1\theta \dots t_n\theta$ (being $f\ t_1 \dots t_n \to r$ a rule in $\mathcal{P}$ and $\theta \in \mathcal{PS}ubst$) and e' is $r\theta$. In this case we want to prove that $\mathcal{A} \vdash r\theta : \tau$. Since $wt_{\mathcal{A}}(\mathcal{P})$, then $\mathcal{A} \vdash^{\bullet} \lambda t_1 \dots \lambda t_n.r : \tau'_1 \to \dots \to \tau'_n \to \tau'$, being $\tau'_1 \to \dots \to \tau'_n \to \tau'$ a variant of $\mathcal{A}(f)$. We assume that the variables of the patterns $\overline{t_i}$ do not appear in $\mathcal{A}$ or in $Rng(\theta)$. The tree for this type derivation will be:

$$[\Lambda]\ \frac{\mathcal{A}_1 \vdash t_1 : \tau'_1 \qquad [\Lambda]\ \dfrac{\mathcal{A}_2 \vdash t_2 : \tau'_2 \qquad [\Lambda]\ \dfrac{\dfrac{\mathcal{A}_n \vdash t_n : \tau'_n \qquad \mathcal{A}_n \vdash r : \tau'}{\vdots}}{\mathcal{A}_2 \vdash t_3 \dots t_n : \tau'_3 \to \dots \to \tau'_n \to \tau'}}{\mathcal{A}_1 \vdash \lambda t_2 \dots t_n.r : \tau'_2 \to \dots \to \tau'_n \to \tau'}}{\mathcal{A} \vdash \lambda t_1 \dots t_n.r : \tau'_1 \to \tau'_2 \dots \to \tau'_n \to \tau'}$$

  where $\mathcal{A}_j \equiv (\dots(\mathcal{A} \oplus \{\overline{X_{1i} : \tau''_{1i}}\}) \oplus \dots) \oplus \{\overline{X_{ji} : \tau''_{ji}}\}$ and X_{ji} is the i-th variable of the pattern t_j. We can write $\mathcal{A}_n$ as $\mathcal{A} \oplus \mathcal{A}'$, being $\mathcal{A}'$ the set of assumption over the variables of the patterns. As these variables are all different (the left hand side of the rules is linear), by Theorem 1-b we can add the rest of the assumptions to the $\mathcal{A}_j$ to get $\mathcal{A}_n$ and the derivation will remain valid, so $\forall j \in [1, n].\ \mathcal{A}_n \vdash t_j : \tau'_j$. Besides $critVar_{\mathcal{A}}(\lambda t_1 \dots \lambda t_n.r) = \emptyset$, so *a)* every variable X_{ji} which appears in r is transparent in the pattern t_j where it comes.
  It is a premise that $\mathcal{A} \vdash f\ t_1\theta \dots t_n\theta : \tau$, and the tree of the type derivation will be:

$$[\mathbf{APP}]\ \frac{[\mathbf{APP}]\ \dfrac{\mathcal{A} \vdash f\ t_1\theta \dots t_{n-2}\theta : \tau_{n-1} \to (\tau_n \to \tau) \qquad \mathcal{A} \vdash t_{n-1}\theta : \tau_{n-1}}{\mathcal{A} \vdash f\ t_1\theta \dots t_{n-1}\theta : \tau_n \to \tau} \qquad \mathcal{A} \vdash t_n\theta : \tau_n}{\mathcal{A} \vdash f\ t_1\theta \dots t_n\theta : \tau}$$

where the type derivation $\mathcal{A} \vdash f\ t_1\theta \ldots t_{n-2}\theta : \tau_{n-1} \to (\tau_n \to \tau)$ is

$$\dfrac{\dfrac{\dfrac{}{\mathcal{A} \vdash f : \tau_1 \to \ldots \to \tau_n \to \tau \mathcal{A} \vdash f : \overline{\tau_n} \to \tau}\,[\textbf{ID}] \quad \mathcal{A} \vdash t_1\theta : \tau_1}{\vdots}\,[\textbf{APP}]}{\mathcal{A} \vdash f\ t_1\theta \ldots t_{n-2}\theta : \tau_{n-1} \to (\tau_n \to \tau)}\,[\textbf{APP}]$$

Because of that, we know that *b)* $\forall j \in [1, n]$. $\mathcal{A} \vdash t_j\theta : \tau_j$ and $\mathcal{A} \vdash f : \tau_1 \to \ldots \to \tau_n \to \tau$, being $\tau_1 \to \ldots \to \tau_n \to \tau$ a generic instance of the type $\mathcal{A}(f)$. Then there will exists a type substitution π such that $(\tau'_1 \to \ldots \to \tau'_n \to \tau')\pi = \tau_1 \to \ldots \to \tau_n \to \tau$, so $\forall j \in [1, n]$. $\tau'_j\pi = \tau_j$ and $\tau'\pi = \tau$. What is more, $Dom(\pi)$ does not contain any free type variable in $\mathcal{A}$, since π transforms a variant of the type of $\mathcal{A}(f)$ into a generic instance of the type of $\mathcal{A}(f)$. Then by Theorem 1-a $\mathcal{A}_n\pi \vdash t_j : \tau'_j\pi$, which is equal to *c)* $\mathcal{A} \oplus \mathcal{A}'\pi \vdash t_j : \tau'_j\pi$.
With *a)*, *b)* and *c)* and by Lemma 1 we can state that for every transparent variable X_{ji} in r then $\mathcal{A} \vdash X_{ji}\theta : \tau''_{ji}\pi$. None of the variables in $\mathcal{A}'$ appear in $X_{ji}\theta$, so by Theorem 1-b we can add these assumptions and obtain $\mathcal{A}_n \vdash X_{ji}\theta : \tau''_{ji}\pi$. According to the first derivation, we have $\mathcal{A}_n \vdash r : \tau'$. Here we can apply the Theorem 1-a again and get a derivation $\mathcal{A}_n\pi \vdash r : \tau'\pi$. Because $\mathcal{A}_n\pi \vdash X_{ji}\theta : \tau''_{ji}\pi$, then by Theorem 1-c $\mathcal{A}_n\pi \vdash r\theta : \tau'\pi$. As we have eliminated the variables in the expression, by Theorem 1-b we can delete their assumptions, obtaining a derivation $\mathcal{A}\pi \vdash r\theta : \tau'\pi$ (remember that $\mathcal{A}_n$ is $\mathcal{A} \oplus \mathcal{A}'$). And finally using the information we have about π, this derivation is equal to $\mathcal{A} \vdash r\theta : \tau$, the derivation we wanted to obtain.

– **(LetIn)** In this case $\mathcal{A} \vdash e_1e_2 : \tau$ and $\mathcal{P} \vdash e_1e_2 \to^l let_m\ X = e_2\ in\ e_1$. The type derivation of e_1e_2 will have the form:

$$[\textbf{APP}]\dfrac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1e_2 : \tau}$$

With this information we could build a type judgment for the let_m expression

$$[\textbf{LET}_m]\dfrac{\mathcal{A} \oplus \{X : \tau_1\} \vdash X : \tau_1 \qquad \mathcal{A} \vdash e_2 : \tau_1 \qquad [\textbf{APP}]\ \dfrac{\mathcal{A} \oplus \{X : \tau_1\} \vdash e_1 : \tau_1 \to \tau \qquad \mathcal{A} \oplus \{X : \tau_1\} \vdash X : \tau_1}{\mathcal{A} \oplus \{X : \tau_1\} \vdash e_1X : \tau}}{\mathcal{A} \vdash let_m\ X = e_2\ in\ e_1X : \tau}$$

$\mathcal{A} \oplus \{X : \tau_1\} \vdash X : \tau_1$ is a valid derivation because is an application of the [ID] rule. And since X is a fresh variable, by Theorem 1-b we can add the assumption and obtain $\mathcal{A} \oplus \{X : \tau_1\} \vdash e_1 : \tau_1 \to \tau$.

– **(Bind)** We will distinguish between the let_m and the let_p case. In both cases we assume that the variable X is fresh.

let_m) In the let_m case the type derivation will have the form:

$$[\textbf{LET}_m]\dfrac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \qquad \mathcal{A} \vdash t : \tau_t \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash e : \tau}{\mathcal{A} \vdash let_m\ X = t\ in\ e : \tau}$$

As X is different from all the variables $\overline{X_i}$ of the pattern t, then by Theorem 1-b we can add the assumption over the variable X and obtain the derivation $\mathcal{A} \oplus \{X : \tau_t\} \vdash t : \tau_t$. Applying the Theorem 1-c then $\mathcal{A} \oplus \{X : \tau_t\} \vdash e[X/t] : \tau$. X will not appear in $e[X/t]$, so again by Theorem 1-b we can eliminate the assumption, concluding that $\mathcal{A} \vdash e[X/t] : \tau$.

let_p) Here the type derivations will be:

$$[\mathbf{LET}_p]\frac{\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \quad \mathcal{A} \vdash t : \tau_t \quad \mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash e : \tau}{\mathcal{A} \vdash let_p\ X = t\ in\ e : \tau}$$

and we want to prove that $\mathcal{A} \vdash e[X/t] : \tau$. We have a type derivation for $\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash e : \tau$, and according to Observation 7 there will be derivations $(\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash X : \tau_i$ for every appearance of X in e. In these cases, $\mathcal{A}'_i$ will only contain assumptions over variables $\overline{X_i}$ in let or lambda expressions of e. Suppose that all these variables have been renamed to fresh variables. We can create a type substitution π from the variables $\overline{\alpha_i}$ of τ_t which do not appear in $\mathcal{A}$ to fresh type variables $\overline{\beta_i}$. It is clear that $Gen(\tau_t, \mathcal{A})$ is equivalent to $Gen(\tau_t\pi, \mathcal{A})$, so $\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\} \vdash e : \tau$ is a valid derivation. By Theorem 1-a $\mathcal{A}\pi \vdash t : \tau_t\pi$, and since $\overline{\alpha_i}$ are not in $\mathcal{A}$ then $\mathcal{A} \vdash t : \tau_t\pi$. X and $\overline{X_i}$ are fresh so they do not appear in t and by Theorem 1-b we can add assumptions to the derivation $\mathcal{A} \vdash t : \tau_t\pi$, obtaining $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_t\pi$. The types τ_i will be generic instances of $Gen(\tau_t, \mathcal{A})$, and also of $Gen(\tau_t\pi, \mathcal{A})$. Then for each τ_i there will exist a type substitution π'_i from the generalized variables $\overline{\beta_i}$ in $Gen(\tau_t\pi, \mathcal{A})$ to types that will hold $\tau_t\pi\pi'_i \equiv \tau_i$. By Theorem 1-a we can convert $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_t\pi$ into $((\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i)\pi'_i \vdash t : \tau_t\pi\pi'_i$, and as $\overline{\beta_i}$ are fresh variables then $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_t\pi\pi'_i$ (note that π'_i does not affect $Gen(\tau_t\pi, \mathcal{A})$ because the variables $\overline{\beta_i}$ are generalized). This way in every place of the original derivation where we have $(\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash X : \tau_i$ we could place a derivation $(\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\}) \oplus \mathcal{A}'_i \vdash t : \tau_i$. The resulting expression of this substitution will be $e[X/t]$, so $\mathcal{A} \oplus \{X : Gen(\tau_t\pi, \mathcal{A})\} \vdash e[X/t] : \tau$. It is clear that X does not appear in $e[X/t]$, so by Theorem 1-b we can eliminate the assumption over the X and obtain a derivation $\mathcal{A} \vdash e[X/t] : \tau$, as we wanted to prove.

- **(Elim)** In this case it does not matter what type of let expression it was (let_m or let_p). The rewriting step will be of the form $\mathcal{P} \vdash let_*\ X = e_1\ in\ e_2 \to^l e_2$. The type derivation of $\mathcal{A} \vdash let_*\ X = e_1\ in\ e_2 : \tau$ will have a branch $\mathcal{A} \oplus \{X : \sigma'\} \vdash e_2 : \tau$ for some σ. Since we are using the **(Elim)** rule, X does not appear in e_2 so by Theorem 1-b we can derive the same type eliminating that assumption, obtaining $\mathcal{A} \vdash e_2 : \tau$.
- **($\mathbf{Flat}_m$)** There are two cases, depending on the second let expression. In both cases we assume that $X \neq Y$.
  - $\mathcal{P} \vdash let_m\ X = (let_m\ Y = e_1\ in\ e_2)\ in\ e_3 \to^l let_m\ Y = e_1\ in\ (let_m\ X = e_2\ in\ e_3)$.

The type derivation will be:

$$[\mathbf{LET}_m]\frac{\dfrac{\mathcal{A}\oplus\{Y:\tau_y\}\vdash Y:\tau_y \quad \mathcal{A}\vdash e_1:\tau_y \quad \mathcal{A}\oplus\{Y:\tau_y\}\vdash e_2:\tau_x}{\mathcal{A}\vdash let_m\ Y=e_1\ in\ e_2:\tau_x}[\mathbf{LET}_m] \quad \mathcal{A}\oplus\{X:\tau_x\}\vdash X:\tau_x \quad \mathcal{A}\oplus\{X:\tau_x\}\vdash e_3:\tau}{\mathcal{A}\vdash let_m\ X=(let_m\ Y=e_1\ in\ e_2)\ in\ e_3:\tau}$$

Then we can build a type derivation

$$[\mathbf{LET}_m]\frac{\dfrac{(\mathcal{A}\oplus\{Y:\tau_y\})\oplus\{X:\tau_x\}\vdash X:\tau_x \quad \mathcal{A}\oplus\{Y:\tau_y\}\vdash e_2:\tau_x \quad (\mathcal{A}\oplus\{Y:\tau_y\})\oplus\{X:\tau_x\}\vdash e_3:\tau}{\mathcal{A}\oplus\{Y:\tau_y\}\vdash let_m\ X=e_2\ in\ e_3:\tau}[\mathbf{LET}_m] \quad \mathcal{A}\oplus\{Y:\tau_y\}\vdash Y:\tau_y \quad \mathcal{A}\vdash e_1:\tau_y}{\mathcal{A}\vdash let_m\ Y=e_1\ in\ (let_m\ X=e_2\ in\ e_3):\tau}$$

The only two derivations which do not come from the hypotheses are $(\mathcal{A}\oplus\{Y:\tau_y\})\oplus\{X:\tau_x\}\vdash X:\tau_x$ and $(\mathcal{A}\oplus\{Y:\tau_y\})\oplus\{X:\tau_x\}\vdash e_3:\tau$. The first is the application of the **[ID]** rule. From the hypotheses we have a derivation $\mathcal{A}\oplus\{X:\tau_x\}\vdash e_3:\tau$. Since we are rewriting using the **(Flat)** rule, we are sure that Y is not in e_3 and by Theorem 1-b we can add the assumption over the Y, obtaining the derivation $(\mathcal{A}\oplus\{X:\tau_x\})\oplus\{Y:\tau_y\}\vdash e_3:\tau$. X is different from Y, so according to Observation 3 $(\mathcal{A}\oplus\{X:\tau_x\})\oplus\{Y:\tau_y\}$ is the same as $(\mathcal{A}\oplus\{Y:\tau_y\})\oplus\{X:\tau_x\}$. Therefore $(\mathcal{A}\oplus\{Y:\tau_y\})\oplus\{X:\tau_x\}\vdash e_3:\tau$ is a valid derivation.

- $\mathcal{P}\vdash let_m\ X=(let_p\ Y=e_1\ in\ e_2)\ in\ e_3\rightarrow^l let_p\ Y=e_1\ in\ (let_m\ X=e_2\ in\ e_3)$. Similar to the previous case.

– **(Flat$_p$)** We will treat the two different cases:

- $\mathcal{P}\vdash let_p\ X=(let_p\ Y=e_1\ in\ e_2)\ in\ e_3\rightarrow^l let_p\ Y=e_1\ in\ (let_p\ X=e_2\ in\ e_3)$.

  The type derivation of the original expression is (being $\mathcal{A}_Y\equiv\mathcal{A}\oplus\{Y:Gen(\tau_y,\mathcal{A})\}$)

$$[\mathbf{LET}_p]\frac{\dfrac{\mathcal{A}\oplus\{Y:\tau_y\}\vdash Y:\tau_y \quad \mathcal{A}\vdash e_1:\tau_y \quad \mathcal{A}_Y\vdash e_2:\tau_x}{\mathcal{A}\vdash let_p\ Y=e_1\ in\ e_2:\tau_x}[\mathbf{LET}_p] \quad \mathcal{A}\oplus\{X:\tau_x\}\vdash X:\tau_x \quad \mathcal{A}\oplus\{X:Gen(\tau_x,\mathcal{A})\}\vdash e_3:\tau}{\mathcal{A}\vdash let_p\ X=(let_p\ Y=e_1\ in\ e_2)\ in\ e_3:\tau}$$

With this derivations as hypothesis we can build a type derivation of the new expression

$$[\mathbf{LET}_p]\dfrac{[\mathbf{LET}_p]\dfrac{\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x \quad \mathcal{A}_Y \vdash e_2 : \tau_x \quad \mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau}{\mathcal{A}_Y \vdash let_p\ X = e_2\ in\ e_3 : \tau} \quad \mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \quad \mathcal{A} \vdash e_1 : \tau_y}{\mathcal{A} \vdash let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3) : \tau}$$

$\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y$, $\mathcal{A} \vdash e_1 : \tau_y$ and $\mathcal{A}_Y \vdash e_2 : \tau_x$ are the same derivations that appear in the original type derivation; and $\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x$ holds trivially applying the **[ID]** rule. But the derivation $\mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau$ has to be proven. As before, since $Y \notin FV(e_3)$ by Theorem 1-b we can add an assumption over the Y and the derivation $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\} \vdash e_3 : \tau$ will remain valid. Because $X \neq Y$ then by Observation 3 $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\}$ is the same as $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A})\}$, and the derivation $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_3 : \tau$ will be correct. Clearly $Gen(\tau_x, \mathcal{A}_Y)$ is not equal to $Gen(\tau_x, \mathcal{A})$ because a previous assumption for Y can be shadowed so that some free type variables in $\mathcal{A}$ are not in $\mathcal{A}_Y$. In the generalization step this means that some variables can be generalized in $Gen(\tau_x, \mathcal{A}_Y)$ but not in $Gen(\tau_x, \mathcal{A})$. The other case never happens because adding $\{Y : Gen(\tau_y, \mathcal{A})\}$ to $\mathcal{A}$ never adds free type variables: if some type variable in τ_y is not in $FTV(\mathcal{A})$ then it will be generalized and will not be in $FTV(\mathcal{A}_Y)$ either. Therefore $Gen(\tau_x, \mathcal{A}_Y) \succ Gen(\tau_x, \mathcal{A})$, and by Theorem 1-d the derivation $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau$ is valid.

- $\mathcal{P} \vdash let_p\ X = (let_m\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$.
  The type derivation of the original expression is:

$$[\mathbf{LET}_p]\dfrac{\mathcal{A} \oplus \{X : \tau_x\} \vdash X : \tau_x \quad [\mathbf{LET}_m]\dfrac{\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \quad \mathcal{A} \vdash e_1 : \tau_y \quad \mathcal{A} \oplus \{Y : \tau_y\} \vdash e_2 : \tau_x}{\mathcal{A} \vdash let_m\ Y = e_1\ in\ e_2 : \tau_x} \quad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_3 : \tau}{\mathcal{A} \vdash let_p\ X = (let_m\ Y = e_1\ in\ e_2)\ in\ e_3 : \tau}$$

  and we want to build one of the form (being $\mathcal{A}_Y \equiv \mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}$):

$$[\mathbf{LET}_p]\dfrac{[\mathbf{LET}_p]\dfrac{\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x \quad \mathcal{A}_Y \vdash e_2 : \tau_x \quad \mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau}{\mathcal{A}_Y \vdash let_p\ X = e_2\ in\ e_3 : \tau} \quad \mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y \quad \mathcal{A} \vdash e_1 : \tau_y}{\mathcal{A} \vdash let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3) : \tau}$$

The derivations $\mathcal{A} \oplus \{Y : \tau_y\} \vdash Y : \tau_y$ and $\mathcal{A} \vdash e_1 : \tau_y$ come from the original derivation; and $\mathcal{A}_Y \oplus \{X : \tau_x\} \vdash X : \tau_x$ is the trivial application of the [ID] rule. From the original derivation we have $\mathcal{A} \oplus \{Y : \tau_y\} \vdash e_2 : \tau_x$. It is easy to see that $Gen(\tau_y, \mathcal{A}) \succ \tau_y$, so by Theorem 1-d $\mathcal{A}_Y \vdash e_2 : \tau_x$. We also have from the original derivation that $\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_3 : \tau$. We know that $Y \notin FV(e_3)$, so by Theorem 1-b we can add an assumption over that variable and the derivation $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\} \vdash e_3 : \tau$ will be valid. X is different from Y, so according to Observation 3 the set of assumptions $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\}) \oplus \{Y : Gen(\tau_y, \mathcal{A})\}$ is the same as $(\mathcal{A} \oplus \{Y : Gen(\tau_y, \mathcal{A})\}) \oplus \{X : Gen(\tau_x, \mathcal{A})\}$. By the same reasons given in the previous case $Gen(\tau_x, \mathcal{A}_Y) \succ Gen(\tau_x, \mathcal{A})$, so by Theorem 1-d the derivation $\mathcal{A}_Y \oplus \{X : Gen(\tau_x, \mathcal{A}_Y)\} \vdash e_3 : \tau$ will be valid.

– **(LetAp)** We will distinguish between the different let expressions.

let_m) The rewriting step is $\mathcal{P} \vdash (let_m\ X = e_1\ in\ e_2)e_3 \rightarrow^l let_m\ X = e_1\ in\ e_2e_3$. The type derivation of $(let_m\ X = e_1\ in\ e_2)e_3$ is:

$$[\mathbf{APP}]\dfrac{[\mathbf{LET}_m]\ \dfrac{\begin{array}{c}\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \end{array}}{\mathcal{A} \oplus \{X : \tau_t\} \vdash e_2 : \tau_1 \rightarrow \tau} \qquad \mathcal{A} \vdash e_3 : \tau_1}{\mathcal{A} \vdash (let_m\ X = e_1\ in\ e_2)e_3 : \tau}$$

We want to construct a type derivation of the form:

$$[\mathbf{LET}_m]\dfrac{[\mathbf{APP}]\ \dfrac{\mathcal{A} \oplus \{X : \tau_t\} \vdash e_2 : \tau_1 \rightarrow \tau \qquad \mathcal{A} \oplus \{X : \tau_t\} \vdash e_3 : \tau_1}{\mathcal{A} \oplus \{X : \tau_t\} \vdash e_2e_3 : \tau} \qquad \begin{array}{c}\mathcal{A} \oplus \{X : \tau_t\} \vdash X : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \end{array}}{\mathcal{A} \vdash let_m\ X = e_1\ in\ e_2e_3 : \tau}$$

All the derivations appear in the original derivation, except $\mathcal{A} \oplus \{X : \tau_t\} \vdash e_3 : \tau_1$. Because we are using **(LetAp)**, we are sure that X does not appear in $FV(e_3)$. From the original derivation we have that $\mathcal{A} \vdash e_3 : \tau_1$, and by Theorem 1-b we can add an assumption over the variable X and obtain the derivation $\mathcal{A} \oplus \{X : \tau_t\} \vdash e_3 : \tau_1$.

let_p) Similar to the let_m) case.

– **(Contx)** We have a derivation $\mathcal{A} \vdash \mathcal{C}[e] : \tau$, so according to the Observation 7 in that derivation will appear a derivation a) $\mathcal{A} \oplus \mathcal{A}' \vdash e : \tau'$, being $\mathcal{A}'$ a set of assumptions over variables. If we apply the rule **(Contx)** to reduce an expression $\mathcal{C}[e]$ is because we reduce the expression e using any of the other rules of the let-rewriting relation b) $\mathcal{P} \vdash e \rightarrow^l e'$. We also know by Observation 8 that c) $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P})$. With a), b) and c) the Induction Hypothesis states that $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau'$, and by Lemma 5 then $\mathcal{A} \vdash \mathcal{C}[e'] : \tau$. □

Theorem 4 (Soundness of $\Vdash$ wrt $\vdash$)

1) $\mathcal{A} \Vdash e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash e : \tau$

Proof.
We proceed by induction over the size of the type inference $\mathcal{A} \Vdash e : \tau|\pi$.
Base Case

- [**iID**] We have a type inference of the form:

$$[\mathbf{iID}]\frac{}{\mathcal{A} \Vdash g : \tau|id}$$

where $\mathcal{A}(g) = \sigma$ and τ is a variant of σ. It is clear that if τ is a variant of σ it is also a generic instance of σ, and $\mathcal{A}\ id \equiv \mathcal{A}$ so the following type derivation is valid:

$$[\mathbf{ID}]\frac{}{\mathcal{A} \vdash g : \tau}$$

Induction Step

- [**iAPP**] The type inference is:

$$[\mathbf{iAPP}]\frac{\mathcal{A} \Vdash e_1 : \tau_1|\pi_1 \qquad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2}{\mathcal{A} \Vdash e_1 e_2 : \alpha\pi|\pi_1\pi_2\pi}$$

where $\pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha)$, being α a fresh type variable. By the Induction Hypothesis we have that $\mathcal{A}\pi_1 \vdash e_1 : \tau_1$ and $\mathcal{A}\pi_1\pi_2 \vdash e_2 : \tau_2$. We can apply Theorem 1-a to both derivations and obtain $\mathcal{A}\pi_1\pi_2\pi \vdash e_1 : \tau_1\pi_2\pi$ and $\mathcal{A}\pi_1\pi_2\pi \vdash e_2 : \tau_2\pi$. Since we know that $\tau_1\pi_2\pi = (\tau_2 \to \alpha)\pi = \tau_2\pi \to \alpha\pi$ then we can construct the type derivation:

$$[\mathbf{APP}]\frac{\mathcal{A}\pi_1\pi_2\pi \vdash e_1 : \tau_2\pi \to \alpha\pi \qquad \mathcal{A}\pi_1\pi_2\pi \vdash e_2 : \tau_2\pi}{\mathcal{A}\pi_1\pi_2\pi \vdash e_1 e_2 : \alpha\pi}$$

- [**i**Λ] The type inference will be of the form:

$$[\mathbf{i}\Lambda]\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t|\pi_t \qquad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t \Vdash e : \tau|\pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau|\pi_t\pi}$$

where $\overline{\alpha_i}$ are fresh type variables. By the Induction Hypothesis we have that $\mathcal{A}\pi_t \oplus \{\overline{X_i : \alpha_i\pi_t}\} \vdash t : \tau_t$ and $\mathcal{A}\pi_t\pi \oplus \{\overline{X_i : \alpha_i\pi_t\pi}\} \vdash e : \tau$. We can apply Theorem 1-a to the first derivation and obtain $\mathcal{A}\pi_t\pi \oplus \{\overline{X_i : \alpha_i\pi_t\pi}\} \vdash t : \tau_t\pi$. Therefore the following type derivation is correct:

$$[\Lambda]\frac{\mathcal{A}\pi_t\pi \oplus \{\overline{X_i : \alpha_i\pi_t\pi}\} \vdash t : \tau_t\pi \qquad \mathcal{A}\pi_t\pi \oplus \{\overline{X_i : \alpha_i\pi_t\pi}\} \vdash e : \tau}{\mathcal{A}\pi_t\pi \vdash \lambda t.e : \tau_t\pi \to \tau}$$

- [**iLET**$_m$] In this case the type inference will be:

$$[\mathbf{iLET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{\overline{X_i:\alpha_i}\}\Vdash t:\tau_t|\pi_t\\ \mathcal{A}\pi_t\Vdash e:\tau_1|\pi_1\\ (\mathcal{A}\oplus\{\overline{X_i:\alpha_i}\})\pi_t\pi_1\pi\Vdash e_2:\tau_2|\pi_2\end{array}}{\mathcal{A}\Vdash let_m\ t=e_1\ in\ e_2:\tau_2|\pi_t\pi_1\pi\pi_2}$$

where $\overline{\alpha_i}$ are fresh type variables and $\pi = mgu(\tau_t\pi_1, \tau_1)$. By the Induction Hypothesis we have that $\mathcal{A}\pi_t \oplus \{\overline{X_i : \alpha_i\pi_t}\} \vdash t : \tau_t$, $\mathcal{A}\pi_t\pi_1 \vdash e : \tau_1$ and $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_i : \alpha_i\pi_t\pi_1\pi\pi_2}\} \vdash e_2 : \tau_2$. We can apply Theorem 1-a to the first two derivations and obtain $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_i : \alpha_i\pi_t\pi_1\pi\pi_2}\} \vdash t : \tau_t\pi_1\pi\pi_2$ and $\mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash e : \tau_1\pi\pi_2$. Finally, as $\tau_t\pi_1\pi = \tau_1\pi$ then we can build a type derivation of the form:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\pi_t\pi_1\pi\pi_2\oplus\{\overline{X_i:\alpha_i\pi_t\pi_1\pi\pi_2}\}\vdash t:\tau_1\pi\pi_2\\ \mathcal{A}\pi_t\pi_1\pi\pi_2\vdash e:\tau_1\pi\pi_2\\ \mathcal{A}\pi_t\pi_1\pi\pi_2\oplus\{\overline{X_i:\alpha_i\pi_t\pi_1\pi\pi_2}\}\vdash e_2:\tau_2\end{array}}{\mathcal{A}\pi_t\pi_1\pi\pi_2\vdash let_m\ t=e_1\ in\ e_2:\tau_2}$$

– $[\mathbf{iLET}^X_{pm}]$ The inference will be:

$$[\mathbf{LET}^X_{pm}]\frac{\mathcal{A}\Vdash e_1:\tau_1|\pi_1 \qquad \mathcal{A}\pi_1\oplus\{X:Gen(\tau_1,\mathcal{A}\pi_1)\}\Vdash e_2:\tau_2|\pi_2}{\mathcal{A}\Vdash let_{pm}\ X=e_1\ in\ e_2:\tau_2|\pi_1\pi_2}$$

By the Induction Hypothesis we have the type derivations $\mathcal{A}\pi_1 \vdash e_1 : \tau_1$ and $\mathcal{A}\pi_1\pi_2 \oplus \{X : Gen(\tau_1, \mathcal{A}\pi_1)\pi_2\} \vdash e_2 : \tau_2$. We can construct a type substitution $\pi \in \mathcal{TS}ubst$ such that maps the type variables in $FTV(\tau_1) \setminus FTV(\mathcal{A}\pi_1)$ to fresh variables. Then it is clear that $Gen(\tau_1, \mathcal{A}\pi_1) = Gen(\tau_1\pi, \mathcal{A}\pi_1)$. On the other hand, all the variables in $\tau_1\pi$ which are not in $FTV(\mathcal{A}\pi_1)$ are fresh so they do not appear in π_2, and by Lemma 7 $Gen(\tau_1\pi, \mathcal{A}\pi_1)\pi_2 = Gen(\tau_1\pi\pi_2, \mathcal{A}\pi_1\pi_2)$. Therefore the type derivation

$$\mathcal{A}\pi_1\pi_2 \oplus \{X : Gen(\tau_1\pi\pi_2, \mathcal{A}\pi_1\pi_2)\} \vdash e_2 : \tau_2$$

is correct. By Theorem 1-a we obtain $\mathcal{A}\pi_1\pi\pi_2 \vdash e_1 : \tau_1\pi\pi_2$, and as $Dom(\pi) \cap FTV(\mathcal{A}\pi_1) = \emptyset$ then $\mathcal{A}\pi_1\pi_2 \vdash e_1 : \tau_1\pi\pi_2$.
Finally with these derivations we can build the type derivation we intended:

$$[\mathbf{LET}^X_{pm}]\frac{\mathcal{A}\pi_1\pi_2\vdash e_1:\tau_1\pi\pi_2 \qquad \mathcal{A}\pi_1\pi_2\oplus\{X:Gen(\tau_1\pi\pi_2,\mathcal{A}\pi_1\pi_2)\}\vdash e_2:\tau_2}{\mathcal{A}\pi_1\pi_2\vdash let_{pm}\ X=e_1\ in\ e_2:\tau_2}$$

– $[\mathbf{iLET}^h_{pm}]$ This case is similar to the $[\mathbf{LET}_m]$ case.
– $[\mathbf{iLET}_p]$ In this case we have an inference of the form:

$$[\mathbf{iLET}_p]\frac{\begin{array}{c}\mathcal{A}\oplus\{\overline{X_i:\alpha_i}\}\Vdash t:\tau_t|\pi_t\\ \mathcal{A}\pi_t\Vdash e_1:\tau_1|\pi_1\\ \mathcal{A}\pi_t\pi_1\pi\oplus\{\overline{X_i:Gen(\alpha_i\pi_t\pi_1\pi,\mathcal{A}\pi_t\pi_1\pi)}\}\Vdash e_2:\tau_2|\pi_2\end{array}}{\mathcal{A}\Vdash let_p\ t=e_1\ in\ e_2:\tau_2|\pi_t\pi_1\pi\pi_2}$$

where $\pi = mgu(\tau_t\pi_1, \tau_1)$. By the Induction Hypothesis we have $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_i : Gen(\alpha_i\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)\pi_2}\} \vdash e_2 : \tau_2$ and $\mathcal{A}\pi_t \oplus \{\overline{X_i : \alpha_i\pi_t}\} \vdash t : \tau_t$, $\mathcal{A}\pi_t\pi_1 \vdash e_1 : \tau_1$. Let be $\overline{\beta_i}$ the type variables in all the types $\alpha_i\pi_t\pi_1\pi$ which do

not appear in $\mathcal{A}\pi_t\pi_1\pi$. We can create a type substitution π' from $\overline{\beta_i}$ to fresh variables. It is clear that $Gen(\alpha_i\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi) = Gen(\alpha_i\pi_t\pi_1\pi\pi', \mathcal{A}\pi_t\pi_1\pi)$, as π' only substitutes the variables that will be generalized by fresh ones which will also be generalized, so it is a renaming of the bounded variables (Observation 1). Therefore the derivation

$$\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_i : Gen(\alpha_i\pi_t\pi_1\pi\pi', \mathcal{A}\pi_t\pi_1\pi)\pi_2}\} \vdash e_2 : \tau_2$$

is also valid. Applying the Theorem 1-a to the first two derivations we obtain $\mathcal{A}\pi_t\pi_1\pi\pi'\pi_2 \oplus \{\overline{X_i : \alpha_i\pi_t\pi_1\pi\pi'\pi_2}\} \vdash t : \tau_t\pi_1\pi\pi'\pi_2$ and $\mathcal{A}\pi_t\pi_1\pi\pi'\pi_2 \vdash e_1 : \tau_1\pi\pi'\pi_2$. By construction, no variable in $Dom(\pi')$ or $Rng(\pi')$ is in $FTV(\mathcal{A}\pi_t\pi_1\pi)$, so $\mathcal{A}\pi_t\pi_1\pi\pi'\pi_2 = \mathcal{A}\pi_t\pi_1\pi\pi_2$. By Lemma 7 we know that $Gen(\alpha_i\pi_t\pi_1\pi\pi', \mathcal{A}\pi_t\pi_1\pi)\pi_2 = Gen(\alpha_i\pi_t\pi_1\pi\pi'\pi_2, \mathcal{A}\pi_t\pi_1\pi\pi_2)$, so the derivation $\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{X_i : Gen(\alpha_i\pi_t\pi_1\pi\pi'\pi_2, \mathcal{A}\pi_t\pi_1\pi\pi_2)\} \vdash e_2 : \tau_2$ is correct. With this derivations as premises we can build the expected one:

$$[\mathbf{LET}_p]\frac{\begin{array}{c}\mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_i : \alpha_i\pi_t\pi_1\pi\pi'\pi_2}\} \vdash t : \tau_1\pi\pi'\pi_2 \\ \mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash e_1 : \tau_1\pi\pi'\pi_2 \\ \mathcal{A}\pi_t\pi_1\pi\pi_2 \oplus \{\overline{X_i : Gen(\alpha_i\pi_t\pi_1\pi\pi'\pi_2, \mathcal{A}\pi_t\pi_1\pi\pi_2)}\} \vdash e_2 : \tau_2\end{array}}{\mathcal{A}\pi_t\pi_1\pi\pi_2 \vdash let_p\ t = e_1\ in\ e_2 : \tau_2}$$

(remembering that $\tau_t\pi_1\pi = \tau_1\pi$ because of π is a mgu).

□

---

2) $\mathcal{A} \Vdash^\bullet e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash^\bullet e : \tau$

By definition of $\Vdash^\bullet$ we have that $\mathcal{A} \Vdash e : \tau$ and $critVar_{\mathcal{A}\pi}(e)$. Applying the soundness of $\Vdash$ (Theorem 4) we have that $\mathcal{A}\pi \vdash e : \tau$. Since $\mathcal{A}\pi \vdash e : \tau$ and $critVar_{\mathcal{A}\pi}(e)$, then by definition of $\vdash^\bullet$ we have $\mathcal{A}\pi \vdash^\bullet e : \tau$.

□

Theorem 5 (Completeness of $\Vdash$ wrt $\vdash$).
$\mathcal{A}\pi' \vdash e : \tau' \Longrightarrow \exists\tau, \pi, \pi''.\ \mathcal{A} \Vdash e : \tau|\pi \wedge \mathcal{A}\pi\pi'' = \mathcal{A}\pi' \wedge \tau\pi'' = \tau'$.

Proof.
This proof is based on the proof of completeness of algorithm $\mathcal{W}$ in [12]. We proceed by induction over the size of the type derivation.
Base Case

– **[ID]** In this case we have a type derivation:

$$[\mathbf{ID}]\frac{}{\mathcal{A}\pi' \vdash s : \tau'}$$

if $\mathcal{A}\pi'(s) = \sigma$ and $\sigma \succ \tau'$. Let's suppose that $\mathcal{A}(s) = \forall\overline{\alpha_i}.\tau''$ (with $\overline{\alpha}$ fresh variables), then $\sigma \equiv (\forall\overline{\alpha_i}.\tau'')\pi' = \forall\overline{\alpha_i}.(\tau''\pi')$. Since $\sigma \succ \tau'$ then there exists a type substitution $[\overline{\alpha_i/\tau_i}]$ such that $\tau' = (\tau''\pi')[\overline{\alpha_i/\tau_i}]$.
Let $\overline{\beta_i}$ be fresh variables. As $\tau''[\overline{\alpha_i/\beta_i}]$ is a variant of $\forall\overline{\alpha_i}.\tau''$ then the following type inference is correct:

$$[\textbf{iID}]\frac{}{\mathcal{A} \Vdash s : \tau''[\overline{\alpha_i/\beta_i}]|id}$$

There is also a type substitution $\pi'' \equiv \pi'[\overline{\beta_i/\tau_i}]$ such that $\tau''[\overline{\alpha_i/\beta_i}]\pi'' = \tau''[\overline{\alpha_i/\beta_i}]\pi'[\overline{\beta_i/\tau_i}] = (\tau''\pi')[\overline{\alpha_i/\beta_i}][\overline{\beta_i/\tau_i}] = (\tau''\pi')[\overline{\alpha_i/\tau_i}] = \tau'$. Finally, it is clear that $\mathcal{A}id\pi'' = \mathcal{A}id\pi'[\overline{\beta_i/\tau_i}] = \mathcal{A}\pi'[\overline{\beta_i/\tau_i}] = \mathcal{A}\pi'$ because $\overline{\beta_i}$ are fresh and cannot occur in $FTV(\mathcal{A}\pi')$.

Induction Step

– **[APP]** The type derivation will be:

$$[\textbf{APP}]\frac{\mathcal{A}\pi' \vdash e_1 : \tau_1' \to \tau' \quad \mathcal{A}\pi' \vdash e_2 : \tau_1'}{\mathcal{A}\pi' \vdash e_1 e_2 : \tau'}$$

By the Induction Hypothesis we know that $\mathcal{A} \Vdash e_1 : \tau_1|\pi_1$ and there is a type substitution π_1'' such that $\tau_1\pi_1'' = \tau_1' \to \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1''$. Since $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1''$ then the derivation $(\mathcal{A}\pi_1)\pi_1'' \vdash e_2 : \tau_1'$ is correct, and again by the Induction Hypothesis we know that $\mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2$ and that there exists a type substitution π_2'' such that $\tau_2\pi_2'' = \tau_1'$ and $\mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2''$. We can assume that π_2'' is minimal, so $Dom(\pi_2'') \subseteq FTV(\tau_2) \cup FTV(\mathcal{A}\pi_1\pi_2)$. In order to prove that the existence of a type inference $\mathcal{A} \Vdash e_1 : \alpha\pi|\pi_1\pi_2\pi$ we need to prove that there exists a most general unifier for $\tau_1\pi_2$ and $\tau_2 \to \alpha$ (being α a fresh variable). For that, we will construct a type substitution π_u which will unify these two types. We know that $\mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2''$, so for all the variables which are free in $\mathcal{A}\pi_1$ then $\pi_1'' = \pi_2\pi_2''$. Let α a fresh type variable, $B = Dom(\pi_1'') \setminus FTV(\mathcal{A}\pi_1)$ and $\pi_u \equiv \pi_2'' + \pi_1''|_B + [\alpha/\tau']$. π_u is well defined because the domains of the three substitutions are disjoints. According to Observation 6, the variables in $FTV(\tau_2)$, $Dom(\pi_2)$ or $Rng(\pi_2)$ which are not in $FTV(\mathcal{A}\pi_1)$ are fresh variables and cannot occur in B. Since the variables in B are neither in $FTV(\mathcal{A}\pi_1)$ nor in $Rng(\pi_2)$ then they do not appear in $FTV(\mathcal{A}\pi_1\pi_2)$ either; and as π_2'' is minimal then no variable in B could occur in $Dom(\pi_2'')$. Besides α is fresh, and it can occur neither in π_2'' nor in $\pi_1''|_B$. Applying π_u to $\tau_2 \to \alpha$ we obtain $(\tau_2 \to \alpha)\pi_u = \tau_2\pi_u \to \alpha\pi_u = \tau_2\pi_2'' \to \alpha[\alpha/\tau'] = \tau_1' \to \tau'$. On the other hand, $\tau_1\pi_2\pi_u = \tau_1' \to \tau'$ because if a type variable of τ_1 is in $\mathcal{A}\pi_1$ then $\tau_1\pi_2\pi_u = \tau_1\pi_2\pi_2'' = \tau_1\pi_1'' = \tau_1' \to \tau'$, and if not it will be in B and π_2 will not affect it, so $\tau_1\pi_2\pi_u = \tau_1\pi_u = \tau_1\pi_1''|_B = \tau_1' \to \tau'$. Since π_u is an unifier, then there will exists a most general unifier π of $\tau_1\pi_2$ and $\tau_2 \to \alpha$ [19]. Therefore the following type inference is correct:

$$[\textbf{iAPP}]\frac{\mathcal{A} \Vdash e_1 : \tau_1|\pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2}{\mathcal{A} \vdash e_1 e_2 : \alpha\pi|\pi_1\pi_2\pi}$$

Now we have to prove that there exists a type substitution π'' such that $\alpha\pi\pi'' = \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_2\pi\pi''$. This is easy defining π'' such that $\pi_u = \pi\pi''$ (which is well defined as π_u is an unifier and π is the most general unifier). Then it is clear that $\alpha\pi\pi'' = \alpha\pi_u = \alpha[\alpha/\tau'] = \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2'' = \mathcal{A}\pi_1\pi_2\pi_u = \mathcal{A}\pi_2\pi_2\pi\pi''$.

– [Λ] We assume that the variables $\overline{X_i}$ in the pattern t do not appear in $\mathcal{A}\pi'$ (nor in $\mathcal{A}$). In this case the type derivation is:

$$[\Lambda]\frac{\mathcal{A}\pi' \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau'_t \quad \mathcal{A}\pi' \oplus \{\overline{X_i : \tau_i}\} \vdash e : \tau'}{\mathcal{A}\pi' \vdash \lambda t.e : \tau'_t \to \tau'}$$

Let $\overline{\alpha_i}$ be fresh type variables and $\pi_g \equiv [\overline{\alpha_i/\tau_i}]$. Then the first derivation is equal to $(\mathcal{A} \oplus \{X_i : \alpha_i\})\pi'\pi_g \vdash t : \tau'_t$. By the Induction Hypothesis we know that $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t|\pi_t$ and that exists a type substitution π''_t such that $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi'\pi_g = (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi''_t$ and $\tau_t\pi''_t = \tau'_t$. Because the data variables $\overline{X_i}$ do not appear in $\mathcal{A}$, then it is true that $\mathcal{A}\pi'\pi_g = \mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t$ and for every type variable $\alpha_i\pi'\pi_g = \alpha_i\pi_g = \tau_i = \alpha\pi_t\pi''_t$.
Using these equalities we can write $\mathcal{A}\pi' \oplus \{\overline{X_i : \tau_i}\}$ as $\mathcal{A}\pi_t\pi''_t \oplus \{\overline{X_i : \alpha_i\pi_t\pi''_t}\}$, that is the same as $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi''_t$. Then, the second derivation is equal to $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi''_t \vdash e : \tau'$, and by the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t \Vdash e : \tau_e|\pi_e$ and there exists a type substitution π''_e such that $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi''_t = (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_e\pi''_e$ and $\tau_e\pi''_e = \tau'$. As before, it is also true that $\mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_e\pi''_e$ and for every type variable $\alpha_i\pi_t\pi''_t = \alpha\pi_t\pi_e\pi''_e$. We can assume that π''_e is minimal, so $Dom(\pi''_e) \subseteq FTV(\tau_e) \cup FTV((\mathcal{A} \cup \{\overline{X_i : \alpha_i}\})\pi_t\pi_e)$. Therefore the type inference for the lambda expression exists and have the form:

$$[\mathbf{i}\Lambda]\frac{\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t|\pi_t \quad (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t \Vdash e : \tau_e|\pi_e}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi_e \to \tau_e|\pi_t\pi_e}$$

Now we have to prove that there exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi_e\pi''$ and $(\tau_t\pi_e \to \tau_e)\pi'' = \tau'_t \to \tau'$. Let be $B \equiv Dom(\pi''_t) \setminus FTV((\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t)$ and $\pi'' \equiv \pi''_t|_B + \pi''_e$, which is well defined because the domains are disjoints. According to Observation 6, the variables which are not in $FTV((\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t)$ and appear in $FTV(\tau_e)$, $Dom(\pi_e)$ or in $Rng(\pi_e)$ are fresh, so they cannot be in B. As these variables do not appear in $Rng(\pi_e)$ then they do not appear in $FTV((\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_e)$; so the variables in B are not in $Dom(\pi''_e)$ and the domains of π''_e and $\pi''_t|_B$ are disjoints.
It is clear that $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_e\pi''_e = \mathcal{A}\pi_t\pi_e\pi''$ because π''_e is part of π''. To prove that $(\tau_t\pi_e \to \tau_e)\pi'' = \tau'_t \to \tau'$ we need to prove that $\tau_t\pi_e\pi'' = \tau'_t$ and $\tau_e\pi'' = \tau'$. The second part is straightforward because $\tau' = \tau_e\pi''_e = \tau_e\pi''$. To prove the first one we will distinguish over the type variables in τ_t. For all the type variables of τ_t which are in $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t$ (i.e. they are not in B) we know that $\tau_t\pi_e\pi'' = \tau_t\pi_e\pi''_e = \tau_t\pi''_t = \tau'_t$ because $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi''_t = (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_e\pi''_e$. For the variables in τ_t which are in B the case is simpler because we know they do not appear in $Dom(\pi_e)$, therefore so $\tau_t\pi_e\pi'' = \tau_t\pi'' = \tau_t\pi''_t|_B = \tau'_t$.
– [$\mathbf{LET}_m$] We assume that the variables $\overline{X_i}$ of the pattern t are fresh and do not occur in $\mathcal{A}\pi'$ (nor in $\mathcal{A}$). Then the type derivation will be:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\pi' \oplus \{\overline{X_i : \tau_i}\} \vdash t : \tau'_t \\ \mathcal{A}\pi' \vdash e_1 : \tau'_t \\ \mathcal{A}\pi' \oplus \{\overline{X_i : \tau_i}\} \vdash e_2 : \tau'\end{array}}{\mathcal{A}\pi' \vdash let_m\ t = e_1\ in\ e_2 : \tau'}$$

Let $\overline{\alpha_i}$ be fresh type variables, and $\pi_g \equiv [\overline{\alpha_i/\tau_i}]$. Since $\overline{\alpha_i}$ are fresh it is clear that $\mathcal{A}\pi'\pi_g = \mathcal{A}\pi'$ and $\alpha_i\pi'\pi_g = \alpha_i\pi_g = \tau_i$ for every type variable α_i. Then we can write the first derivation as $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi'\pi_g \vdash t : \tau'_t$ and by the Induction Hypothesis $\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\} \Vdash t : \tau_t|\pi_t$ and there is a type substitution π''_t such that $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi'\pi_g = (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi''_t$ and $\tau_t\pi''_t = \tau'_t$. Since the data variables $\overline{X_i}$ do not appear in $\mathcal{A}\pi'$ then $\mathcal{A}\pi' = \mathcal{A}\pi'\pi_g = \mathcal{A}\pi_t\pi''_t$ and for every type variable $\alpha_i\pi'\pi_g = \alpha_i\pi_g = \tau_i = \alpha_i\pi_t\pi''_t$. Since $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t$ then we can write the second derivation as $\mathcal{A}\pi_t\pi''_t \vdash e_1 : \tau'_t$, and by the Induction Hypothesis $\mathcal{A}\pi_t \Vdash e_1 : \tau_1|\pi_1$ and there exists a type substitution π''_1 such that $\mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_1\pi''_1$ and $\tau_1\pi''_1 = \tau'_t$. We can assume that π''_1 is minimal, so $Dom(\pi''_1) \subseteq FTV(\tau_1) \cup FTV(\mathcal{A}\pi_t\pi_1)$. Now we have to prove that $\tau_t\pi_1$ and τ_1 are unifiable, so there exists a most general unifier [19]. We define $B \equiv FTV(\pi''_1) \setminus FTV(\mathcal{A}\pi_t)$ and $\pi_u \equiv \pi''_1 + \pi''_t|_B$, which is well defined because the domains of the two components are disjoints. According to Observation 6, the variables of $FTV(\tau_1)$, $Dom(\pi_1)$ or $Rng(\pi_1)$ which do not occur in $FTV(\mathcal{A}\pi_t)$ will be fresh variables, so they will not be any of the variables in B. As the variables in B occur neither in $FTV(\mathcal{A}\pi_t)$ nor in $Rng(\pi_1)$, then they do not appear in $\mathcal{A}\pi_1\pi_1$; and as π''_1 is minimal then no variable in B occurs in $Dom(\pi''_1)$.

π_u is an unifier of $\tau_t\pi_1$ and τ_1 because $\tau_t\pi_1\pi_u = \tau_1\pi_u = \tau'_t$. The first case is easy because $\tau_1\pi_u = \tau_1\pi''_1 = \tau'_t$. To prove the second we will distinguish over the type variables of τ_t. For the type variables of τ_t in $\mathcal{A}\pi_t$ (i.e. those which are not in B) we know that $\tau_t\pi_1\pi_u = \tau_t\pi_1\pi''_1 = \tau_t\pi''_t = \tau'_t$, and for the others (those in B) we know they are fresh and do not appear in π_1, so $\tau_t\pi_1\pi_u = \tau_t\pi_u = \tau_t\pi''_t|_B = \tau'_t$. Therefore there will exist a most general unifier π, and $\pi_u = \pi\pi_o$.

We also know that $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_1\pi''_1 = \mathcal{A}\pi_t\pi_1\pi_u = \mathcal{A}\pi_t\pi_1\pi\pi_o$ and for every type variable $\alpha_i\pi_t\pi_1\pi\pi_o = \tau_i$ (for the type variables of $\alpha_i\pi_t$ which are in $\mathcal{A}\pi_t$ then $\alpha_i\pi_t\pi_1\pi\pi_o = \alpha_i\pi_t\pi_1\pi_u = \alpha_i\pi_t\pi_1\pi''_1 = \alpha_i\pi_t\pi''_t = \tau_i$, and for the rest of the variables -those in B- then $\alpha_i\pi_t\pi_1\pi\pi_o = \alpha_i\pi_t\pi_1\pi_u = \alpha_i\pi_t\pi_u = \alpha_i\pi_t\pi''_t|_B = \tau_i$).

Then we can write the third derivation as $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi\pi_o \vdash e_2 : \tau'$, and by the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2|\pi_2$ and there exists a type substitution π''_2 such that $\tau_2\pi''_2 = \tau'$ and $(\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi\pi_o = (\mathcal{A} \oplus \{\overline{X_i : \alpha_i}\})\pi_t\pi_1\pi\pi_2\pi''_2$. Since the variables $\overline{X_i}$ do not appear in $\mathcal{A}$, in particular it is true that $\mathcal{A}\pi_t\pi_1\pi\pi_o = \mathcal{A}\pi_t\pi_1\pi\pi_2\pi''_2$.

With these three type inferences we can build the type inference for the let expression:

$$[\mathbf{iLET}_m]\frac{\begin{array}{c}\mathcal{A}\oplus\{\overline{X_i:\alpha_i}\}\Vdash t:\tau_t|\pi_t\\ \mathcal{A}\pi_t\Vdash e_1:\tau_1|\pi_1\\ (\mathcal{A}\oplus\{\overline{X_i:\alpha_i}\})\pi_t\pi_1\pi\Vdash e_2:\tau_2|\pi_2\end{array}}{\mathcal{A}\Vdash let_m\ t=e_1\ in\ e_2:\tau_2|\pi_t\pi_1\pi\pi_2}$$

being $\pi = mgu(\tau_t\pi_1, \tau_1)$. To finish this case we only have to prove that there exists a type substitution π'' such that $\tau_2\pi'' = \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi_1\pi\pi_2\pi''$. This substitution π'' is π''_2.

– $[\mathbf{LET}^X_{pm}]$ We assume that X does not occur in $\mathcal{A}$. We have a type derivation:

$$[\mathbf{LET}^X_{pm}]\frac{\begin{array}{c}\mathcal{A}\pi'\vdash e_1:\tau'_1\\ \mathcal{A}\pi'\oplus\{X:Gen(\tau'_1,\mathcal{A}\pi')\}\vdash e_2:\tau'_2\end{array}}{\mathcal{A}\pi'\vdash let_{pm}\ X=e_1\ in\ e_2:\tau'_2}$$

By the Induction Hypothesis we have that $\mathcal{A}\Vdash e_1:\tau_1|\pi_1$ and there exists a type substitution π''_1 such that $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi''_1$ and $\tau_1\pi''_1 = \tau'_1$. $Gen(\tau'_1, \mathcal{A}\pi') = Gen(\tau_1\pi''_1, \mathcal{A}\pi_1\pi''_1)$, so by Lemma 8 $Gen(\tau_1, \mathcal{A}\pi_1)\pi''_1 \succ Gen(\tau'_1, \mathcal{A}\pi')$. Then by Theorem 1-d the type derivation $\mathcal{A}\pi'\oplus\{X : Gen(\tau_1, \mathcal{A}\pi_1)\pi''_1\}\vdash e_2 : \tau'_2$ is valid. We can write this derivation as $(\mathcal{A}\pi_1\oplus\{X : Gen(\tau_1, \mathcal{A}\pi_1)\})\pi''_1\vdash e_2 : \tau'_2$ and applying the Induction Hypothesis we obtain that $\mathcal{A}\pi_1\oplus\{X : Gen(\tau_1, \mathcal{A}\pi_1)\}\Vdash e_2 : \tau_2|\pi_2$ and there exists a type substitution π''_2 such that $\tau_2\pi''_2 = \tau'_2$ and $(\mathcal{A}\pi_1\oplus\{X : Gen(\tau_1, \mathcal{A}\pi_1)\})\pi_2\pi''_2 = (\mathcal{A}\pi_1\oplus\{X : Gen(\tau_1, \mathcal{A}\pi_1)\})\pi''_1$. Since X does not appear in $\mathcal{A}$ the last equality means that $\mathcal{A}\pi_1\pi_2\pi''_2 = \mathcal{A}\pi_1\pi''_1$ and $Gen(\tau_1, \mathcal{A}\pi_1)\pi_2\pi''_2 = Gen(\tau_1, \mathcal{A}\pi_1)\pi''_1$. With the previous type inferences we can construct a type inference for the whole expression:

$$[\mathbf{iLET}^X_{pm}]\frac{\begin{array}{c}\mathcal{A}\Vdash e_1:\tau_1|\pi_1\\ \mathcal{A}\pi_1\oplus\{X:Gen(\tau_1,\mathcal{A}\pi_1)\}\Vdash e_2:\tau_2|\pi_2\end{array}}{\mathcal{A}\Vdash let_{pm}\ X=e_1\ in\ e_2:\tau_2|\pi_1\pi_2}$$

In this case it is easy to see that there exists a type substitution (π''_2) such that $\tau_2\pi''_2 = \tau'_2$ and $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi''_1 = \mathcal{A}\pi_1\pi_2\pi''_2$.

– $[\mathbf{LET}^h_{pm}]$ Equal to the $[\mathrm{LET}_m]$ case.

– $[\mathbf{LET}_p]$ The proof of this case follows the same ideas as the cases $[\mathbf{LET}_m]$ and $[\mathbf{LET}^X_{pm}]$.

Theorem 6 (Maximality of $\Vdash^\bullet$).

a) $\Pi^\bullet_{\mathcal{A},e}$ has a maximum element $\iff \exists\tau_g \in SType, \pi_g \in \mathcal{TS}ubst.\mathcal{A}\Vdash^\bullet e : \tau_g|\pi_g$.

b) If $\mathcal{A}\pi'\vdash^\bullet e:\tau'$ and $\mathcal{A}\Vdash^\bullet e:\tau|\pi$ then exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$.

Proof.
a)

– $\Longleftarrow$) If $\mathcal{A}\Vdash^\bullet e:\tau_g|\pi_g$ then by Lemma 9 $\Pi_{\mathcal{A},e} = \Pi^\bullet_{\mathcal{A},e}$. Since $\mathcal{A}\Vdash e:\tau_g|\pi_g$ (by definition of $\Vdash^\bullet$) by Theorem 9 we know that $\Pi_{\mathcal{A},e}$ has a maximum element, and also $\Pi^\bullet_{\mathcal{A},e}$.

– $\Longrightarrow$) We will prove that $\mathcal{A} \not\Vdash^{\bullet} e : \tau_g|\pi_g \Longrightarrow \Pi^{\bullet}_{\mathcal{A},e}$ has not a maximum element.

(A) $\mathcal{A} \not\Vdash^{\bullet} e : \tau_g|\pi_g$ because $\mathcal{A} \not\Vdash e : \tau_g|\pi_g$. We know from Theorem 9 that if $\mathcal{A} \not\Vdash e : \tau_g|\pi_g$ then $\Pi_{\mathcal{A},e}$ has not a maximum element. Then by Theorem 5 it cannot exists any type derivation $\mathcal{A}\pi' \vdash e : \tau'$, so $\Pi_{\mathcal{A},e}$ is empty. Since $\Pi^{\bullet}_{\mathcal{A},e} \subseteq \Pi_{\mathcal{A},e}$ then $\Pi^{\bullet}_{\mathcal{A},e} = \emptyset$ and cannot contain any maximum element.

(B) $\mathcal{A} \not\Vdash^{\bullet} e : \tau_g|\pi_g$ because $\mathcal{A} \Vdash e : \tau_g|\pi_g$ but $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$. We will proceed by case distintion over the cause of the critical variables:

(B.1) $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$ because for every pattern t_j in e and for every variable X_i in t_j that is critical then the cause of the opacity are type variables which appear in $\mathcal{A}\pi_g$. In other words, for those variables X_i then $\mathcal{A} \oplus \{X_i : \alpha_i\} \Vdash t_j : \tau_j|\pi_j$ and $FTV(\alpha_i\pi_j) \nsubseteq FTV(\tau_j)$ and $FTV(\alpha_i\tau_j) \setminus FTV(\tau_j) \subseteq FTV(\mathcal{A}\pi_g)$. It is clear that we can apply a type substitution to $\mathcal{A}\pi_g$ and eliminate the opacity of these variables. In particular we will always be able to find two type substitions π_1 and π_2 such that:

i. $\mathcal{A}\pi_g\pi_1 \vdash e : \tau_1$ and $\mathcal{A}\pi_g\pi_2 \vdash e : \tau_2$.
ii. $critVar_{\mathcal{A}\pi_g\pi_1}(e) = \emptyset$ and $critVar_{\mathcal{A}\pi_g\pi_2}(e) = \emptyset$
iii. No substitution π more general than $\pi_g\pi_1$ and $\pi_g\pi_2$ is in $\Pi^{\bullet}_{\mathcal{A},e}$ because $critVar_{\mathcal{A}\pi}(e) = \emptyset$.

Let be $\overline{\beta_k}$ all the type variables causing opacity, and τ^1 and τ^2 two non unifiable types (*bool* and *char*, for example). Then we can define $\pi_1 \equiv [\overline{\beta_k/\tau^1}]$ and $\pi_2 \equiv [\overline{\beta_k/\tau^2}]$. Since $\mathcal{A} \Vdash e : \tau_g|\pi_g$ by Theorem 4 $\mathcal{A}\pi_g \vdash e : \tau_g$, and by Theorem 1-a $\mathcal{A}\pi_g\pi_1 \vdash e : \tau_g\pi_1$ and $\mathcal{A}\pi_g\pi_2 \vdash e : \tau_g\pi_2$. We have eliminated the cause of opacity, so $critVar_{\mathcal{A}\pi_g\pi_1}(e) = \emptyset$ and $critVar_{\mathcal{A}\pi_g\pi_2}(e) = \emptyset$, i.e., $\pi_g\pi_1, \pi_g\pi_2 \in \Pi^{\bullet}_{\mathcal{A},e}$. Finally since τ^1 and τ^2 are not unifiable, the only substitution more general that $\pi_g\pi_1$ and $\pi_g\pi_2$ that could be in $\Pi^{\bullet}_{\mathcal{A},e}$ is π_g (substitutions more general than π_g cannot be in $\Pi_{\mathcal{A},e}$, and neither in $\Pi^{\bullet}_{\mathcal{A},e}$). But π_g is not in $\Pi^{\bullet}_{\mathcal{A},e}$ because $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$. Therefore $\Pi^{\bullet}_{\mathcal{A},e}$ cannot have a maximum element because we have found two elements in $\Pi^{\bullet}_{\mathcal{A},e}$ that do not have any "greater" element in $\Pi^{\bullet}_{\mathcal{A},e}$.

(B.2) $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$ because there exists some pattern t_j in e in which there is any variable X that is opaque because of type variables that do not occur in $\mathcal{A}\pi_g$. Intuitively in this case these type variables will have appeared because of there exist a symbol in t_j whose type is a type-scheme, and that fresh variables come from the fresh variant used. From Theorem 5 we know that for every π_e in $\Pi_{\mathcal{A},e}$ then $\mathcal{A}\pi_e = \mathcal{A}\pi_g\pi''$ for some type substitution π'. But $critVar_{\mathcal{A}\pi_e}(e) = critVar_{\mathcal{A}\pi_g\pi''}(e) \neq \emptyset$, because we always have fresh type variables causing opacity (since they come from type-schemes, substitutions do not affect them). Therefore for every $\pi_e \in \Pi_{\mathcal{A},e}$ then $critVar_{\mathcal{A}\pi_e}(e) \neq \emptyset$, and as $\Pi^{\bullet}_{\mathcal{A},e} \subseteq \Pi_{\mathcal{A},e}$ then $\Pi^{\bullet}_{\mathcal{A},e} = \emptyset$; so it has not a maximum element.

---

b) By definition of $\vdash^\bullet$ and $\Vdash^\bullet$ we know that $\mathcal{A}\pi' \vdash e : \tau'$ and $\mathcal{A} \Vdash e : \tau|\pi$. Then by Theorem 5 we know that exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$.

Theorem 7 (Soundness of $\mathcal{B}$).
$\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi \Longrightarrow wt_{\mathcal{A}\pi}(\mathcal{P})$.

Proof. From $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ we have $\mathcal{A} \Vdash^\bullet (\varphi(r_1), \ldots, \varphi(r_m)) : (\tau_1, \ldots, \tau_m)|\pi$, and by Theorem 4 then $\mathcal{A}\pi \vdash^\bullet (\varphi(r_1), \ldots, \varphi(r_m)) : (\tau_1, \ldots, \tau_m)$. In order to prove $wt_{\mathcal{A}\pi}(\mathcal{P})$ we need to prove that every rule $r_i \equiv f_i\ t_1 \ldots t_n \to e_i$ in $\mathcal{P}$ is well-typed wrt. $\mathcal{A}\pi$. From Lemma 10 we know that $\mathcal{A}\pi \vdash^\bullet \varphi(r_i) : \tau_i$, so $\mathcal{A}\pi \vdash^\bullet pair\ \lambda t_1 \ldots t_n.e_i\ f_i : \tau_i$. This derivation can only be constructed if $\mathcal{A}\pi \vdash^\bullet \lambda t_1 \ldots t_n.e_i : \tau_i$ and $\mathcal{A}\pi \vdash^\bullet f_i : \tau_i$, and as the last derivation is just an application of rule **[ID]**, $\mathcal{A}\pi(f_i) \succ \tau_i$. We will distinguish between the case that $\mathcal{A}(f_i)$ is a simple type or a closed type-scheme:

a) If $\mathcal{A}(f_i)$ is a simple type, then $\mathcal{A}\pi(f_i)$ too. In this case $\mathcal{A}\pi(f_i) \succ \tau_i$ can only be true if $\mathcal{A}\pi(f_i) = \tau_i$, so trivially τ_i is a variant of $\mathcal{A}\pi(f_i)$. Therefore $\mathcal{A}\pi \vdash^\bullet \lambda t_1 \ldots t_n.e_i : \tau_i$ and τ_i is a variant of $\mathcal{A}\pi(f_i)$, so rule r_i is well-typed wrt. $\mathcal{A}\pi$.
b) $\mathcal{A}(f_i)$ is a closed type scheme, so $\mathcal{A}(f_i) = \mathcal{A}\pi(f_i)$. From step 2.- of $\mathcal{B}$ we know that in this case τ_i is a variant of $\mathcal{A}(f_i)$, and also of $\mathcal{A}\pi(f_i)$. Then since $\mathcal{A}\pi \vdash^\bullet \lambda t_1 \ldots t_n.e_i : \tau_i$ rule r_i is well-typed wrt. $\mathcal{A}\pi$.

Theorem 8 (Maximality of $\mathcal{B}$).
If $wt_{\mathcal{A}\pi'}(\mathcal{P})$ and $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then $\exists\pi''$ such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.

Proof. Since $wt_{\mathcal{A}\pi'}(\mathcal{P})$ we know that for every rule $r_i \equiv f_i\ t_1 \ldots t_n \to e_i$ in $\mathcal{P}$ there exists a type derivation $\mathcal{A}\pi' \vdash^\bullet \lambda t_1 \ldots t_n.e_i : \tau_i'$ and τ_i' is a variant of the type $\mathcal{A}\pi'(f_i)$. Then $\mathcal{A}\pi' \vdash^\bullet f_i : \tau_i'$, and we can construct type derivations $\mathcal{A}\pi' \vdash^\bullet pair\ \lambda t_1 \ldots t_n.e_i\ f_i : \tau_i'$. With these derivations we can build $\mathcal{A}\pi' \vdash^\bullet (\varphi(r_1), \ldots, \varphi(r_m)) : (\tau_1', \ldots, \tau_m')$ by Lemma 10. From $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ we know that $\mathcal{A} \Vdash^\bullet (\varphi(r_1), \ldots, \varphi(r_m)) : (\tau_1, \ldots, \tau_m)|\pi$, so by Theorem 6-*b* there will exist some type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.

Theorem 9 (Maximality of $\Vdash$).
$\Pi_{\mathcal{A},e}$ has a maximum element $\pi \Longleftrightarrow \exists\tau_g, \pi_g \in SType.\mathcal{A} \Vdash e : \tau_g|\pi_g$.

Proof.

$\Longrightarrow$) If $\Pi_{\mathcal{A},e}$ has maximum element π then there will be some type τ such that $\mathcal{A}\pi \vdash e : \tau$. Then by Theorem 5 we know that $\mathcal{A} \Vdash e : \tau_g|\pi_g$.
$\Longleftarrow$) We know from Theorem 5 that for every type substitution $\pi' \in \Pi_{\mathcal{A},e}$ there exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$. Then $\pi|_{FTV(\mathcal{A})} \lesssim \pi'$. From Theorem 4 we know that $\pi|_{FTV(\mathcal{A})}$ is in $\Pi_{\mathcal{A},e}$, so it is the maximum element.

Well-typed Narrowing with Extra Variables in Functional-Logic Programming (Extended Version)*

Technical Report SIC-11-11 (Revised March 30, 2012)
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Francisco López-Fraguas Enrique Martin-Martin Juan Rodríguez-Hortalá
Dpto. de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es emartinm@fdi.ucm.es juanrh@fdi.ucm.es

Abstract

Narrowing is the usual computation mechanism in functional-logic programming (FLP), where bindings for free variables are found at the same time that expressions are reduced. These free variables may be already present in the goal expression, but they can also be introduced during computations by the use of program rules with extra variables. However, it is known that narrowing in FLP generates problems from the point of view of types, problems that can only be avoided using type information at run-time. Nevertheless, most FLP systems use static typing based on Damas-Milner type system and they do not carry any type information in execution, thus ill-typed reductions may be performed in these systems. In this paper we prove, using the let-narrowing relation as the operational mechanism, that types are preserved in narrowing reductions provided the substitutions used preserve types. Based on this result, we prove that types are also preserved in narrowing reductions without type checks at run-time when higher order (HO) variable bindings are not performed and most general unifiers are used in unifications, for programs with transparent patterns. Then we characterize a restricted class of programs for which no binding of HO variables happens in reductions, identifying some problems encountered in the definition of this class. To conclude, we use the previous results to show that a simulation of needed narrowing via program transformation also preserves types.

Categories and Subject Descriptors F.3.3 [*Logics and meanings of programs*]: Studies of Program Constructs—Type Structure; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Theory, Languages, Design

Keywords Functional-logic programming, narrowing, extra variables, type systems

1. Introduction

Functional-logic programming (FLP). Functional logic languages [3, 15, 29] like Toy [23] or Curry [16] can be described as an extension of a lazy purely-functional language similar to Haskell [18], that has been enhanced with logical features, in particular logical variables and non-deterministic functions. Disregarding some syntactic conventions, the following program defining standard list concatenation is valid in all the three mentioned languages:

$$[\,] + + Ys = Ys \qquad [X \mid Xs] + + Ys = [X \mid Xs + + Ys]$$

Logical variables are just free variables that get bound during the computation in a way similar to what it is done in logic programming languages like Prolog [11]. This way FLP shares with logic programming the ability of computing with partially unkown data. For instance, assuming a suitable definition and implementation of equality $==$, the following is a natural FLP definition of a predicate (a *true*-valued function) *sublist* stating that a given list Xs is a sublist of Ys:

$$sublist\ Xs\ Ys = cond\ (Us + + Xs + + Vs == Ys)\ true$$
$$cond\ true\ X = X$$

Notice that the rule for *sublist* is not valid in a functional language due to the presence of the variables Us and Vs, which do not occur in the left hand side of the program rule. They are called *extra variables*. Using *cond* and extra variables makes easy translating pure logic programs into functional logic ones[1]. For instance, the logic program using Peano's natural numbers z (zero) and s (successor)

$$add(z, X, X).$$
$$add(s(X), Y, s(Z)) :- add(X, Y, Z).$$
$$even(X) :- add(Y, Y, X).$$

can be transformed into the following functional logic one:

$$add\ z\ X\ Y = cond\ (X == Y)\ true$$
$$add\ (s\ X)\ Y\ (s\ Z) = add\ X\ Y\ Z$$
$$even\ X = add\ Y\ Y\ X$$

Notice that the rule for *even* is another example of FLP rule with an extra variable Y. The previous examples show that, contrary to

* This paper is the extended version of "**Well-typed Narrowing with Extra Variables in Functional-Logic Programming**", appeared in *Proceeding of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation (PEPM'12)*, pages 83–92, ACM.

[1] As a secondary question here, notice that using *cond* is needed if $==$, as usual, is a two-valued function returning *true* or *false*. Defining directly $sublist\ Xs\ Ys = (Us + + Xs + + Vs == Ys)$ would compute wrong answers: evaluating $sublist\ [1]\ [1, 2]$ produces $true$ but also the wrong value $false$, because there are values of the extra variables Us and Vs such that $Us + +[1] + + Vs == [1, 2]$ evaluates to *false*.

the usual practice in functional programming, free variables may appear freely during the computation, even when starting from an expression without free variables. Nevertheless, despite these connections with logic programming, owing to the functional characteristics of FLP languages, like the nesting of function applications instead of SLD resolution, several variants and formulations of narrowing [19] have been adopted as the computation mechanism in FLP. There are several operational semantics for computing with logical and extra variables [15, 24, 29], and this kind of variables are supported in every modern FLP system.

As FLP languages were already non-deterministic due to the different possible instantiations of logical variables—these are handled by means of a backtracking mechanism similar to that of Prolog—it was natural that these languages eventually evolved to include so-called *non-deterministic functions*, which are functions that may return more than one result for the same input. These functions are expressed by means of program rules whose left hand sides overlap, and that are tried in order by backtracking during the computation, instead of taking a first fit or best fit approach like in pure functional languages. The combination of lazy evaluation and non-deterministic functions gives rise to several semantic options, being *call-time choice* semantics [13] the option adopted by the majority of modern FLP implementations. This point can be easily understood by means of the following program example:

$$coin \to z \quad coin \to s\ z \quad dup\ X \to (X, X)$$

In this example *coin* is a non-deterministic expression, as it can be reduced both to the values z and $s\ z$. But the point is that, according to call-time choice the expression *dup coin* evaluates to *(z, z)* and *(s z, s z)* but not to *(z, s z)* nor *(s z, z)*. Operationally, call-time choice means that all copies of a non-deterministic subexpression, like *coin* in the example, created during the computation reduction share the same value. In Section 2.2 we will see a simple formulation of narrowing for programs with extra variables, that also respects call-time choice, which will be used as the operational procedure for this paper.

Apart from these features, in the Toy system left hand sides of program rules can use not only first order patterns like those available in Haskell programs, but also higher order patterns (*HO-patterns*), which essentially are partial applications of function or constructor symbols to other patterns. This corresponds to an *intensional* view of functions, i.e., different descriptions of the same 'extensional' function can be distinguished by the semantics, and it is formalized and semantically characterized with detail in the HO-CRWL[2] logic for FLP [12]. This is not an exoticism: it is known [24] that extensionality is not a valid principle within the combination of higher order functions, non-determinism and call-time choice. HO-patterns are a great expressive feature [29], however they may have some bad interferences with types, as we will see later in the paper.

Because of all the presented features, FLP languages can be employed to write concise and expressive programs, specially for search problems, as it was explored in [3, 15, 29].

FLP and types. Current FLP languages are strongly typed. Apart from programming purposes, types play a key role in some program analysis or transformations for FLP, as detecting deterministic computations [17], translation of higher order into first order programs [4], or transformation into Haskell [8]. From the point of view of types FLP has not evolved much from Damas-Milner type system [9], so current FLP systems use an almost direct adaptation of that classic type system. However, that approach lacks type preservation during evaluation, even for the restricted case where we drop logical and extra variables. It is known from afar [14] that, even in that simplified scenario, HO-patterns break the type preservation property. In particular that allows us to create polymorphic casting functions [7]—functions with type $\forall\alpha, \beta.\alpha \to \beta$, but that behave like the identity wrt. the reduction of expressions. This has motivated the development of some recent works dealing with *opaque HO-patterns* [22], or liberal type systems for FLP [21]. There are also some preliminary works concerning the incorporation of *type classes* to FLP languages [25, 28], but this feature is still in an experimental phase in current systems.

Regardless of the expressiveness of extra variables these are usually out the scope of the works dealing with types and FLP, in particular in all the aforementioned. But these variables are a distinctive feature of FLP systems, hence in this work our *main goal* is to investigate the properties of a variation of the Damas-Milner type system that is able to handle extra variables, giving an abstract characterization of the problematic issues—most of them were already identified in the seminal work [14]—and then determining sufficient conditions under which type preservation is recovered for programs with extra variables evaluated with narrowing. In particular we are interested in preserving types without having to use type information at run-time, in contrast to what it is done in previous proposals [14].

The rest of the paper is organized as follows. Section 2 contains some technical preliminaries and notations about programs and expressions, and the formulation of the let-narrowing relation $\rightsquigarrow^{l}$, which will be used as the operational mechanism for this paper. In Section 3 we present our type system and study those interactions with let-narrowing that lead to the loss of type preservation. Then we define the well-typed let-narrowing relation $\rightsquigarrow^{lwt}$, a restriction of $\rightsquigarrow^{l}$ that preserves types relying on the abstract notion of well-typed substitution. To conclude that section we present $\rightsquigarrow^{lmgu}$, another restriction of $\rightsquigarrow^{l}$ that is able to preserve types without using type information—in contrast to $\rightsquigarrow^{lwt}$, which uses types at each step to determine that the narrowing substitution is well-typed—at the price of losing some completeness. To cope with this lack of completeness, in Section 4 we look for sufficient conditions under which the narrowing relation $\rightsquigarrow^{lmgu}$ is complete wrt. the computation of well-typed solutions, thus identifying a class of programs for which completeness is recovered, and whose expressiveness is then investigated. In Section 5 we propose a simulation of needed narrowing with $\rightsquigarrow^{lmgu}$ via two well-known program transformations, and show that it also preserves types. The class of programs supported in that section is specially relevant, as it corresponds to a simplified version of the Curry language. Finally Section 6 summarizes some conclusions and future work. Fully detailed proofs, including some auxiliary results, can be found in Appendix A.

2. Preliminaries

2.1 Expressions and programs

We consider a set of *functions* symbols $f, g, \ldots \in FS$ and *constructor* symbols $c, d, \ldots \in CS$, each $h \in FS \cup CS$ with an associated arity $ar(h)$. We also consider a denumerable set of *data variables* $X, Y, \ldots \in \mathcal{V}$. The notation $\overline{o_n}$ stands for a sequence $o_1, \ldots, o_n$ of n syntactic elements o, being o_i the i^{th} element. Figure 1 shows the syntax of patterns $t \in Pat$ and expressions $e \in Exp$. We split the set of patterns into two: *first order patterns* $FOPat \ni fot ::= X \mid c\ \overline{fot_n}$ where $ar(c) = n$, and *higher-order patterns* $HOPat = Pat \setminus FOPat$, i.e., patterns containing some partial application of a symbol of the signature. Expressions $X\ \overline{e_n}$ are called *variable application* when $n > 0$, and expressions with the form $h\ \overline{e_n}$ are called *junk* if $h \in CS$ and $n > ar(h)$ or *active* if $h \in FS$ and $n \geq ar(h)$. The set of *free* and *bound* variables of an expression e—$fv(e)$ and $bv(e)$ resp.—are defined

[2] CRWL [13] stands for *Constructor Based Rewriting Logic*; HO-CRWL is a higher order extension of it.

Data variable		X,Y ...	
Function symbol		f,g ...	
Constructor symbol		c,d ...	
Non-variable symbol	h	::=	$c \mid f$
Symbol	s	::=	$X \mid c \mid f$
Pat	t, p	::=	X
			$\mid c\ \overline{t_n}$ if n $\leq$ ar(c)
			$\mid f\ \overline{t_n}$ if n $<$ ar(f)
FOPat	fot	::=	$X \mid c\ \overline{fot_n}$ if n = ar(c)
Exp	e, r	::=	$X \mid c \mid f \mid e_1\ e_2$
			$\mid let\ X = e_1\ in\ e_2$
PSubst	θ	::=	$[\overline{X_n \mapsto t_n}]$
Cntxt	$\mathcal{C}$	::=	$[\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C}$
			$\mid let\ X = \mathcal{C}\ in\ e$
			$\mid let\ X = e\ in\ \mathcal{C}$
Program rule	R	::=	$f\ \overline{t_n} \to e$ if ar(f) = n
Program	$\mathcal{P}$	::=	$\{\overline{R_n}\}$
Type variable		α,β ...	
Type constructor		C	
Simple type	τ	::=	$\alpha \mid \tau_1 \to \tau_2$
			$\mid C\ \overline{\tau_n}$ if n = ar(C)
Type-scheme	σ	::=	$\forall \overline{\alpha_n}.\tau$
Set of assumptions	$\mathcal{A}$	::=	$\{\overline{s_n : \sigma_n}\}$
TSubst	π	::=	$[\overline{\alpha_n \mapsto \tau_n}]$

Figure 1. Syntax of programs and types

in the usual way. Notice that let-expressions are not recursive, so $fv(let\ X = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus \{X\})$. The set $var(e)$ is the set containing all the variables in e, both free and bound. Notice that for patterns $var(t) = fv(t)$.

Contexts $\mathcal{C} \in Cntxt$ are expressions with one hole, and the application of $\mathcal{C}$ to e—written $\mathcal{C}[e]$—is the standard. The notion of free and bound variables are extended in the natural way to contexts: $fv(\mathcal{C}) = fv(\mathcal{C}[h])$ for any $h \in FS \cup CS$ with $ar(h) = 0$, and $bv(\mathcal{C})$ is defined as $bv([\,]) = \emptyset$, $bv(\mathcal{C}\ e) = bv(\mathcal{C})$, $bv(e\ \mathcal{C}) = bv(\mathcal{C})$, $bv(let\ X = \mathcal{C}\ in\ e) = bv(\mathcal{C})$, $bv(let\ X = e\ in\ \mathcal{C}) = \{X\} \cup bv(\mathcal{C})$.

Data substitution $\theta \in PSubst$ are finite maps from data variables to patterns $[\overline{X_n \mapsto t_n}]$. We write ϵ for the empty substitution, $dom(\theta)$ for the domain of θ and $vran(\theta) = \bigcup_{X \in dom(\theta)} fv(X\theta)$. Given $A \subseteq \mathcal{V}$, the notation $\theta|_A$ represents the restriction of θ to D, and $\theta|_{\setminus A}$ is a shortcut for $\theta|_{\mathcal{V} \setminus A}$. Substitution application over data variables and expressions is defined in the usual way.

Program rules R have the form $f\ \overline{t_n} \to e$, where $ar(f) = n$ and $\overline{t_n}$ is *linear*, i.e., there is no repetition of variables. Notice that we allow extra variables, so it could be the case that e contains variables which do not appear in $\overline{t_n}$. A program $\mathcal{P}$ is a set of program rules.

2.2 Let-narrowing

Let-narrowing [24] is a narrowing relation devised to effectively deal with logical and extra variables, that is also sound and complete wrt. HO-CRWL [12], a standard logic for higher order FLP with call-time choice. Figure 2 contains the rules of the let-narrowing relation $\rightsquigarrow^l$. The first five rules (LetIn)–(LetAp) do not use the program and just change the textual representation of the term graph implied by the let-bindings in order to enable the application of program rules, but keeping the implied term graph untouched. The (Narr) rule performs function application, finding the bindings for the free variables needed to be able to apply the rule, and possibly introducing new variables if the program rule contains some extra variables. Notice that it does not require the use of a most general unifier (mgu) so any unifier can be used. As we will see in Section 3, this later point should be refined in order to ensure type preservation. Rules (VAct) and (VBind) produce HO bindings for variable applications, and are needed for let-narrowing to be complete. These rules are particularly problematic because they have to generate speculative bindings that may involve any function of the program, contrary to (Narr) where the computation of bindings is directed by the program rules for f. Later on we will see how this "wild" nature of the bindings generated by these rules poses especially hard problems to type preservation. Finally, (Contx) allows to apply a narrowing rule in any part of the expression, protecting bound variables from narrowing and avoiding variable capture.

3. Type Preservation

In this section we first present the type system we will use in this work, which is a simple variation of Damas-Milner typing enhanced with support for extra variables. Then we show some examples of $\rightsquigarrow^l$-reductions not preserving types (Section 3.2). Based on the ideas that emerge from these examples, in Section 3.3 we develop a new let-narrowing relation $\rightsquigarrow^{lwt}$ that preserves types. This new relation uses only *well-typed substitutions* in each step, which gives an abstract and general characterization of the requirements a narrowing relation must fulfil in order to preserve types, but it still needs to perform type checks at run-time. To solve this problem, in Section 3.4 we present a restricted let-narrowing $\rightsquigarrow^{lmgu}$ which only uses mgu's as unifiers and drops the problematic rules (VAct) and (VBind). The main advantage of this relation is that if the patterns that can appear in program rules are limited then mgu's are always well-typed, thus obtaining type preservation without using type information at run-time. Sadly this comes at a price, as $\rightsquigarrow^{lmgu}$ loses some completeness wrt. HO-CRWL.

3.1 A type system for extra variables

In Figure 1 we can find the usual syntax for *simple types* τ and *type-schemes* σ. For a simple type τ, the set of *free type variables*—denoted $ftv(\tau)$—is $var(\tau)$, and for type-schemes $ftv(\forall \overline{\alpha_n}.\tau) = var(\tau) \setminus \{\overline{\alpha_n}\}$. A type-scheme is *closed* if $ftv(\sigma) = \emptyset$. We say that a type-scheme is *k-transparent* if it can be written as $\forall \overline{\alpha_n}.\overline{\tau_k} \to \tau$ such that $var(\overline{\tau_k}) \subseteq var(\tau)$.

A set of assumptions $\mathcal{A}$ is a set of the form $\{\overline{s_n : \sigma_n}\}$ such that the assumption for variables are simple types. If $(s_i : \sigma_i) \in \mathcal{A}$ we write $\mathcal{A}(s_i) = \sigma_i$. For sets of assumptions we define $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. The union of set of assumptions is denoted by $\oplus$ with the usual meaning: $\mathcal{A} \oplus \mathcal{A}'$ contains all the assumptions in $\mathcal{A}'$ as well as the assumptions in $\mathcal{A}$ for those symbols not appearing in $\mathcal{A}'$. Based on the previous notion of k-transparency, we say a pattern t is *transparent* wrt. $\mathcal{A}$ if $t \in \mathcal{V}$ or $t \equiv h\ \overline{t_n}$ where $\mathcal{A}(h)$ is n-transparent and $\overline{t_n}$ are transparent patterns. We also say a constructor symbol c is transparent wrt. $\mathcal{A}$ if $\mathcal{A}(c)$ is n-transparent, where $ar(c) = n$.

Type substitutions $\pi \in TSubst$ are mappings from type variables to simple types, where dom and $vran$ are defined similarly to data substitutions. Application of type substitutions to simple types is defined in the natural way, and for type-schemes consists in applying the substitution only to their free variables. This notion is extended to set of assumptions: $\{\overline{s_n : \sigma_n}\}\pi = \{\overline{s_n : \sigma_n \pi}\}$. We say τ is a *generic instance* of $\sigma \equiv \forall \overline{\alpha_n}.\tau'$ if $\tau = \tau'[\overline{\alpha_n \mapsto \tau_n}]$ for some $\overline{\tau_n}$, written $\sigma \succ \tau$. Finally, τ is a *variant* of $\sigma \equiv \forall \overline{\alpha_n}.\tau'$ (denoted by $\sigma \succ_{var} \tau$) if $\tau = \tau'[\overline{\alpha_n \mapsto \beta_n}]$ where $\overline{\beta_n}$ are fresh type variables.

Figure 3 contains the typing rules for expressions considered in this work, which constitute a variation of Damas-Milner typing

(LetIn) $e_1\ e_2 \rightsquigarrow^l_\epsilon let\ X = e_2\ in\ e_1\ X$, if e_2 is an active expression, variable application, junk or let-rooted expression, for X fresh.
(Bind) $let\ X = t\ in\ e \rightsquigarrow^l_\epsilon e[X \mapsto t]$, if $t \in Pat$
(Elim) $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\epsilon e_2$, if $X \notin fv(e_2)$
(Flat) $let\ X = (let\ Y = e_1\ in\ e_2)\ in\ e_3 \rightsquigarrow^l_\epsilon let\ Y = e_1\ in\ (let\ X = e_2\ in\ e_3)$, if $Y \notin fv(e_3)$
(LetAp) $(let\ X = e_1\ in\ e_2)\ e_3 \rightsquigarrow^l_\epsilon let\ X = e_1\ in\ e_2\ e_3$, if $X \notin fv(e_3)$
(Narr) $f\ \overline{t_n} \rightsquigarrow^l_\theta r\theta$, for any fresh variant $(f\ \overline{p_n} \to r) \in \mathcal{P}$ and θ such that $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$.
(VAct) $X\ \overline{t_k} \rightsquigarrow^l_\theta r\theta$, if $k > 0$, for any fresh variant $(f\ \overline{p} \to r) \in \mathcal{P}$ and θ such that $(X\ \overline{t_k})\theta \equiv f\ \overline{p}\theta$
(VBind) $let\ X = e_1\ in\ e_2 \rightsquigarrow^l_\theta e_2\theta[X \mapsto e_1\theta]$, if $e_1 \notin Pat$, for any θ that makes $e_1\theta \in Pat$, provided that $X \notin (dom(\theta) \cap vran(\theta))$
(Contx) $\mathcal{C}[e] \rightsquigarrow^l_\theta \mathcal{C}\theta[e']$, for $\mathcal{C} \neq [\,]$, $e \rightsquigarrow^l_\theta e'$ using any of the previous rules, and:
i) $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$
ii) • if the step is (Narr) or (VAct) using $(f\ \overline{p_n} \to r) \in \mathcal{P}$ then $vran(\theta|_{\setminus var(\overline{p_n})}) \cap bv(\mathcal{C}) = \emptyset$
• if the step is (VBind) then $vran(\theta) \cap bv(\mathcal{C}) = \emptyset$.

Figure 2. Let-narrowing relation $\rightsquigarrow^l$

that now is able to handle extra variables. The main novelty wrt. a regular formulation of Damas-Milner typing with support for pattern matching is that now the (Λ) rule considers extra variables in λ-abstractions: in addition to guessing types for the variables in the pattern t, it also guesses types for the free variables of $\lambda t.e$, which correspond to extra variables. Although λ-abstractions are expressions not included in the syntax of programs showed in Figure 1 and thus they cannot appear in the expressions to reduce[3], we use them as the basis for the notions of well-typed rule and program. Essentially, for each program rule we construct an associated λ-abstraction so the rule is well-typed iff the corresponding λ-abstraction is well-typed. This is reflected in the following definition of *program well-typedness*, an important property assuring that assumptions over functions are related to their rules:

DEFINITION 3.1 (Well-typed program wrt. $\mathcal{A}$). *A program rule $f \to e$ is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ where $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ and $\overline{\tau_n}$ are some simple types. A program rule $(f\ \overline{p_n} \to e)$ (with $n > 0$) is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.e : \tau$ with $\mathcal{A}(f) \succ_{var} \tau$. A program $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ if all its rules are well-typed wrt. $\mathcal{A}$.*

This definition is the same as the one from [22] but it has a different meaning, as it is based on a different definition for the (Λ) rule. Notice that the case $f \to e$ must be handled independently because it does not have any argument. In this case the (Λ) rule is not used to derive the type for e, so the types for the extra variables would not be guessed.

An expression e is well-typed wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash e : \tau$ for some type τ, written as $wt_{\mathcal{A}}(e)$. We will use the metavariable $\mathcal{D}$ to denote particular type derivations $\mathcal{A} \vdash e : \tau$. If $\mathcal{P}$ is well-typed wrt. $\mathcal{A}$ we write $wt_{\mathcal{A}}(\mathcal{P})$.

3.2 Let-narrowing does not preserve types

Now we will see how let-narrowing interacts with types. It is easy to see that let-narrowing steps $\rightsquigarrow^l$ which do not generate bindings for the logical variables—i.e., those using the rules (LetIn), (Bind), (Elim), (Flat) and (LetAp)—preserve types trivially. This is not very surprising because, as we showed in Section 2.2, those steps just change the textual representation of the implied term graph. However, steps generating non trivial bindings can break type preservation easily:

$$\text{(ID)}\ \frac{}{\mathcal{A} \vdash s : \tau}\ \text{if } \mathcal{A}(s) \succ \tau$$

$$\text{(APP)}\ \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

$$\text{(}\Lambda\text{)}\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda t.e : \tau_t \to \tau}\ \text{if } \{\overline{X_n}\} = var(t) \cup fv(\lambda t.e)$$

$$\text{(LET)}\ \frac{\mathcal{A} \vdash e_1 : \tau_x \quad \mathcal{A} \oplus \{X : \tau_x\} \vdash e_2 : \tau}{\mathcal{A} \vdash let\ X = e_1\ in\ e_2 : \tau}$$

Figure 3. Type System

EXAMPLE 3.2. *Consider the function and defined by the rules $\{and\ true\ X \to X, and\ false\ X \to false\}$ with type $(bool \to bool \to bool)$ and the constructor symbols for Peano's natural numbers z and s, with types (nat) and $(nat \to nat)$ respectively. Starting from the expression $and\ true\ Y$—which has type bool when Y has type bool—we can perform the let-narrowing step:*

$$and\ true\ Y \rightsquigarrow^l_{[X_1 \mapsto z, Y \mapsto z]} z$$

This (Narr) step uses the fresh program rule $(and\ true\ X_1 \to X_1)$, but the resulting expression z does not have type bool.

The cause of the loss of type preservation is that the unifier $\theta_1 = [X_1 \mapsto z, Y \mapsto z]$ used in the (Narr) step is ill-typed, because it replaces the boolean variables X_1 and Y by the natural z. The problem with θ_1 is that it instantiates the variables too much, and without using any criterion that ensures that the types of the expressions in its range are adequate.

We have just seen that using the (Narr) rule with an ill-typed unifier may lead to breaking type preservation because of the instantiation of logical variables, like the variable Y above. We may reproduce the same problem easily with extra variables, just consider the function f with type bool defined by the rule $(f \to and\ true\ X)$ for which we can perform the following let-narrowing step:

$$f \rightsquigarrow^l_{[X_2 \mapsto z]} and\ true\ z$$

[3] As there is no general consensus about the semantics of λ-abstractions in the FLP community, due to their interactions with non-determinism and logical variables, we have decided to leave λ-abstractions out of programs and evaluating expressions, thus following the usual applicative programming style of the HO-CRWL logic.

using (Narr) with the fresh rule $(f \to and\ true\ X_2)$*. The resulting expression is obviously ill-typed, and so type preservation is broken again because the substitution used in (Narr) instantiates variables too much and without assuring that the expression in its range have the correct types. The interested reader may easily check that this is also a valid let-rewriting step [24], thus showing that extra variables break type preservation even in the restricted scenario where we drop logical variables. Hence, the type systems in the papers mentioned at the end of Section 1 lose type preservation if we allow extra variables in the programs.*

However, the (Narr) rule is not the only one which can break type preservation. The rules (VAct) and (VBind) also lead to problematic situations:

EXAMPLE 3.3. *Consider the functions and symbols from Example 3.2. Using the rule (VAct) it is possible to perform the step*

$$s\ (F\ z) \leadsto^{l}_{[F \mapsto and\ false, X_3 \mapsto z]} s\ false$$

with the fresh rule $(and\ false\ X_3 \to false)$*. Clearly* $s\ (F\ z)$ *has type* nat *and* F *has type* $(nat \to nat)$*, but the resulting expression is ill-typed. As before, the reason is an ill-typed binding for* F*, which binds* F *with a pattern of type* $(bool \to bool)$*.*

On the other hand, we can perform the step

$$let\ X = F\ z\ in\ s\ X \leadsto^{l}_{[F \mapsto and]} s\ (and\ z)$$

using the rule (VBind). The expression $let\ X = F\ z\ in\ s\ X$ *has type* nat *when* F *has type* $(nat \to nat)$*, but the resulting expression is ill-typed. The cause of the loss of type preservation is again an ill-typed substitution binding, in this case the one for* F *which assigns a pattern of type* $(bool \to bool \to bool)$ *to a variable of type* $(nat \to nat)$*.*

Notice that ill-typed substitutions do not break type preservation necessarily. For example the step $and\ false\ X \leadsto^{l}_{\theta_5} false$ using (Narr) with the fresh rule $(and\ false\ X_5 \to false)$ preserves types, although it can use the ill-typed unifier $\theta_5 \equiv [X \mapsto z, X_5 \mapsto z]$. However, avoiding ill-typed substitutions is a sufficient condition which guarantees type preservation, as we will see soon. Besides, it is important to remark that the bindings for the free variables of the starting expression that are computed in a narrowing derivation are as important as the final value reached at the end of the derivation, because these bindings constitute a solution for the starting expression if we consider it as a goal to be solved, just like the goal expressions used in logic programming. That allows us to use predicate functions like the function *sublists* in Section 1 with some variables as their arguments, i.e., using some arguments in Prolog-like output mode. Therefore, well-typedness of the substitutions computed in narrowing reductions is also important and the restriction to well-typed substitutions is not only reasonable but also desirable, as it ensures that the solutions computed by narrowing respect types.

3.3 Well-typed let-narrowing $\leadsto^{lwt}$

In this section we present a narrowing relation $\leadsto^{lwt}$ which is smaller than $\leadsto^{l}$ in Figure 2 but that preserves types. The idea behind $\leadsto^{lwt}$ is that it only considers steps $e \leadsto^{l}_{\theta} e'$ using well-typed programs where the substitution θ is also well-typed. We say a substitution is well-typed when it replaces data variables by patterns of the same type. Formally:

DEFINITION 3.4 (Well-typed substitution). *A data substitution* θ *is well-typed wrt.* $\mathcal{A}$*, written* $wt_{\mathcal{A}}(\theta)$*, if* $\mathcal{A} \vdash X\theta : \mathcal{A}(X)$ *for every* $X \in dom(\theta)$*.*

Notice that according to the definition of set of assumptions, $\mathcal{A}(X)$ is always a simple type.

As it is usual in narrowing relations, let-narrowing steps can introduce new variables that do not occur in the original expression. Moreover, this new variables do not come only from extra variables but from fresh variants of program rules—using (Narr) and (VAct)—or from invented patterns—using (VBind). Therefore, we need to consider some suitable assumptions over these new variables. However, that set of assumptions over the new variables is not arbitrary but it is closely related to the step used:

EXAMPLE 3.5 ($\mathcal{A}$ associated to a (Narr) step). *Consider the function* f *with type* $\forall\alpha.\alpha \to [\alpha]$ *defined with the rule* $f\ X \to [X, Y]$*. We can perform the narrowing step* $f\ true \leadsto^{l}_{\theta} [true, Y_1]$ *using (Narr) with the fresh variant* $f\ X_1 \to [X_1, Y_1]$ *and* $\theta \equiv [X_1 \mapsto true]$*. Since the original expression is* $f\ true$*, it is clear that* X_1 *must have type* $bool$ *in the new set of assumptions. Moreover,* Y_1 *must have the same type since it appears in a list with* X_1*. Therefore in this concrete step the associated set of assumptions is* $\{X_1 : bool, Y_1 : bool\}$*.*

The following definition establishes when a set of assumptions is associated to a step. Notice that due to the particularities of the rules (VAct) and (VBind), in some cases there is not such set or there are several associated sets.

DEFINITION 3.6 ($\mathcal{A}$ associated to $\leadsto^{l}$ steps). *Given a type derivation* $\mathcal{D}$ *for* $\mathcal{A} \vdash e : \tau$ *and* $wt_{\mathcal{A}}(\mathcal{P})$*, a set of assumptions* $\mathcal{A}'$ *is associated to the step* $e \leadsto^{l}_{\theta} e'$ *iff:*

- $\mathcal{A}' \equiv \emptyset$ *and the step is (LetIn), (Bind), (Elim), (Flat) or (LetAp).*
- *If the step is (Narr) then* $f\ \overline{t_n} \leadsto^{l}_{\theta} r\theta$ *using a fresh variant* $(f\ \overline{p_n} \to r) \in \mathcal{P}$ *and substitution* θ *such that* $(f\ \overline{p_n})\theta \equiv (f\ \overline{t_n})\theta$*. Since* $\mathcal{D}$ *is a type derivation for* $\mathcal{A} \vdash f\ \overline{t_n} : \tau$*, it will contain a derivation* $\mathcal{A} \vdash f : \overline{\tau_n} \to \tau$*. The rule* $f\ \overline{p_n} \to r$ *is well-typed by* $wt_{\mathcal{A}}(\mathcal{P})$*, so we also have (when the rule is* $f \to e$ *it is similar):*

$$(\Lambda)\ \dfrac{\dfrac{\mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash p_n : \tau'_n}{\mathcal{A} \oplus \mathcal{A}_1 \ldots \oplus \mathcal{A}_n \vdash r : \tau'}}{(\Lambda)\ \dfrac{\mathcal{A} \oplus \mathcal{A}_1 \vdash p_1 : \tau'_1 \quad \cdots}{\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'}}$$

  where $\overline{\mathcal{A}_n}$ *are the set of assumptions over variables introduced by* (Λ) *and* $\overline{\tau'_n} \to \tau'$ *is a variant of* $\mathcal{A}(f)$*. Therefore* $(\overline{\tau'_n} \to \tau')\pi \equiv \overline{\tau_n} \to \tau$ *for some type substitution* π *whose domain are fresh type variables from the variant. In this case* $\mathcal{A}'$ *is associated to the (Narr) step if* $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$*.*
- *If the step is (VAct) then we have* $X\ \overline{t_k} \leadsto^{l}_{\theta} r\theta$ *for a fresh variant* $(f\ \overline{p_n} \to r) \in \mathcal{P}$ *and substitution* θ *such that* $(X\ \overline{t_k})\theta \equiv f\ \overline{p_n}\theta$*. Since* $\mathcal{D}$ *is a type derivation for* $\mathcal{A} \vdash X\ \overline{t_k} : \tau$*, it will contain a derivation* $\mathcal{A} \vdash X : \overline{\tau_k} \to \tau$*. The rule* $f\ \overline{p_n} \to r$ *is well-typed by* $wt_{\mathcal{A}}(\mathcal{P})$*, so we have a type derivation* $\mathcal{A} \vdash \lambda p_1 \ldots \lambda p_n.r : \overline{\tau'_n} \to \tau'$ *as in the (Narr) case (similarly when the rule is* $f \to e$*). Let* $\overline{\tau''_k}$ *be* $\tau'_{n-k+1} \to \tau'_{n-k+2} \ldots \to \tau'_n$*, i.e., the last* k *types in* $\overline{\tau'_n}$*. If* $\mathcal{A}' \equiv (\mathcal{A}_1 \oplus \ldots \oplus \mathcal{A}_n)\pi$ *for some substitution* π *such that* $(\overline{\tau''_k} \to \tau')\pi \equiv \overline{\tau_k} \to \tau$ *and* $fv(\mathcal{A}) \cap dom(\pi) = \emptyset$*, then* $\mathcal{A}'$ *is associated to the (VAct) step.*
- *Any* $\mathcal{A}' \equiv \{\overline{X_n : \tau_n}\}$ *is associated to a (VBind) step, if* $\overline{X_n}$ *are those data variables introduced by* $vran(\theta)$*—they do not appear in* $\mathcal{A}$*—and* $\overline{\tau_n}$ *are simple types.*
- $\mathcal{A}'$ *is associated to a (Contx) step if it is associated to its inner step.*

A set of assumptions $\mathcal{A}'$ *is associated to* n $\leadsto^{l}$ *steps* $(e_1 \leadsto^{l} e_2 \ldots \leadsto^{l} e_{n+1})$ *if* $\mathcal{A}' \equiv \mathcal{A}'_1 \oplus \mathcal{A}'_2 \ldots \oplus \mathcal{A}'_n$*, where* $\mathcal{A}'_i$ *is associated to the step* $e_i \leadsto^{l} e_{i+1}$ *and the type derivation* $\mathcal{D}_i$ *for* e_i *using* $\mathcal{A} \oplus \mathcal{A}'_1 \ldots \oplus \mathcal{A}'_{i-1}$ *(*$\mathcal{A}' \equiv \emptyset$ *if* $n = 0$*).*

Based on the previously introduced notions we can define a restriction of let-narrowing that only employs well-typed substitutions, that we will denote by $\leadsto^{lwt}$:

DEFINITION 3.7 ($\leadsto^{lwt}$ let-narrowing). *Consider an expression e, a program $\mathcal{P}$ and set of assumptions $\mathcal{A}$ such that $wt_{\mathcal{A}}(e)$ with a derivation $\mathcal{D}$ and $wt_{\mathcal{A}}(\mathcal{P})$. Then $e \leadsto^{lwt}_{\theta} e'$ iff $e \leadsto^{l}_{\theta} e'$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to $e \leadsto^{l}_{\theta} e'$, $\mathcal{D}$.*

The premises $wt_{\mathcal{A}}(e)$ and $wt_{\mathcal{A}}(\mathcal{P})$ are essential, since the associated set of assumptions wrt. $e \leadsto^{l}_{\theta} e'$ is only well defined in those cases. Note that the step $\leadsto^{lwt}$ cannot be performed if no set of associated assumptions $\mathcal{A}'$ exists. Although $\leadsto^{lwt}$ is strictly smaller than $\leadsto^{l}$—the steps in Examples 3.2 and 3.3 are not valid $\leadsto^{lwt}$-steps—it enjoys the intended type preservation property:

THEOREM 3.8 (Type preservation of $\leadsto^{lwt}$). *If $wt_{\mathcal{A}}(\mathcal{P})$, $e \leadsto^{lwt^*}_{\theta} e'$ and $\mathcal{A} \vdash e : \tau$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

The previous result is the main contribution of this paper. It states clearly that, provided that the substitutions used are well-typed, let-narrowing steps preserve types. Moreover, type preservation is guaranteed for general programs, i.e., programs containing extra variables, non-transparent constructor symbols, opaque HO-patterns ... This result is very relevant because it clearly isolates a sufficient and reasonable property that, once imposed to the unifiers, ensures type preservation. Besides, this condition is based upon the abstract notion of well-typed substitution, which is parameterized by the type system and independent of the concrete narrowing or reduction notion employed. Thus the problem of type preservation in let-narrowing reductions is clarified. New let-narrowing subrelations can be proposed for restricted classes of programs or using particular unifiers and, provided the generated substitutions are well-typed, they will preserve types. We will see an example of that in Section 3.4.

This is an important advance wrt. previous proposals like [14], where the computation of the mgu was interleaved with and inseparable from the rest of the evaluation process in the narrowing derivations. Besides, although the identification of three kinds of problematic situations for the type preservation made in that work was very valuable—especially taking into account it was one of the first studies of the subject in FLP with HO-patterns—having a more general and abstract result is also valuable for the reasons stated above.

3.4 Restricted narrowing using mgu's $\leadsto^{lmgu}$

The $\leadsto^{lwt}$ relation has the good property of preserving types, however it presents a drawback if used as the reduction mechanism of a FLP system: it requires the substitutions generated in each $\leadsto^{lwt}$ step to be well-typed. Since these substitutions are generated just by using the syntactic criteria expressed in the rules of the let-narrowing relation $\leadsto^{l}$, the only way to guarantee this is to perform type checks at run-time, discarding ill-typed substitutions. But, as we mentioned in Section 1, we are interested in preserving types without having to use type information at run-time. Hence, in this section we propose a new let-narrowing relation $\leadsto^{lmgu}$ which preserves types without need of type checks at run-time. The let-narrowing relation $\leadsto^{lmgu}$ is defined as:

DEFINITION 3.9 (Restricted narrowing $\leadsto^{lmgu}$). *$e \leadsto^{lmgu}_{\theta} e'$ iff $e \leadsto^{l}_{\theta} e'$ using any rule from Figure 2 except (VAct) and (VBind), and if the step is $f\ \overline{t_n} \leadsto^{l}_{\theta} r\theta$ using (Narr) with the fresh variant $(f\ \overline{p_n} \to r)$ then $\theta = mgu(f\ \overline{t_n}, f\ \overline{p_n})$.*

As explained in Section 3.2, the rules that break type preservation are (Narr), (VAct) and (VBind). The rules (VAct) and (VBind) present harder problems to preserve types since they replace HO variables by patterns. These patterns are searched in the entire space of possible patterns, producing possible ill-typed substitutions. Since we want to avoid type checks at run-time, and we have not found any syntactic criterion to forbid the generation of ill-typed substitutions by those rules, (VAct) and (VBind) have been omitted from $\leadsto^{lmgu}$. Although this makes $\leadsto^{lmgu}$ a relation strictly smaller than $\leadsto^{lwt}$, it is still meaningful: expressions needing (VAct) or (VBind) to proceed can be considered as *frozen* until other let-narrowing step instantiates the HO variable. This is somehow similar to the operational principle of *residuation* used in some FLP languages such as Curry [15, 16]. Regarding the rule (Narr), Example 3.2 shows the cause of the break of type preservation. In that example, the unifier of $and\ true\ Y$ and $and\ true\ X_1$ is $\theta_1 = [X_1 \mapsto z, Y \mapsto z]$. Although θ_1 is a valid unifier, it instantiates variables unnecessarily in an ill-typed way. In other words, it does not use just the information from the program and the expression, which are well-typed, but it "invents" the pattern z. We can solve this situation easily using the mgu $\theta'_1 = [X_1 \mapsto Y]$, which is well-typed, so by Theorem 3.8 we can conclude that the step preserves types.

Moreover, this solution applies to any (Narr) step (under certain conditions that will be specified later): if we chose mgu's in the (Narr) rule and both the rule and the original expression are well-typed, then the mgu's will also be well-typed. This fact is based in the following result:

LEMMA 3.10 (Mgu well-typedness). *Let $\overline{p_n}$ be fresh linear transparent patterns wrt. $\mathcal{A}$ and let $\overline{t_n}$ be any patterns such that $\mathcal{A} \vdash p_i : \tau_i$ and $\mathcal{A} \vdash t_i : \tau_i$ for some type τ_i. If $\theta \equiv mgu(f\ \overline{p_n}, f\ \overline{t_n})$ then $wt_{\mathcal{A}}(\theta)$.*

The restriction to fresh linear transparent patterns $\overline{p_n}$ is essential, otherwise the mgu may not be well-typed. Consider for example the constructor $cont : \forall\alpha.\alpha \to container$ and a set of assumptions $\mathcal{A}$ containing $(X : nat)$. It is clear that $p \equiv cont\ X$ is linear but non-transparent, because $cont$ is not 1-transparent. Both p and $t \equiv cont\ true$ patterns have type $container$ and $mgu(f\ p, f\ t) = [X \mapsto true] \equiv \theta$ for any function symbol f. However the unifier θ is ill-typed as $\mathcal{A} \not\vdash X\theta : \mathcal{A}(X)$, i.e., $\mathcal{A} \not\vdash true : nat$. Similarly, consider the patterns $p' \equiv (Y, Y)$ and $t' \equiv (cont\ X, cont\ true)$ and a set of assumptions $\mathcal{A}$ containing $(Y : container, X : nat)$. It is easy to see that p' and t' have type $(container, container)$, and p' is transparent but non-linear. The mgu of $f\ p'$ and $f\ t'$ is $[Y \mapsto cont\ true, X \mapsto true]$, which is ill-typed by the same reasons as before.

Due to the previous result, type preservation is only guaranteed for $\leadsto^{lmgu}$-reductions for programs such that left-hand sides of rules contain only transparent patterns. This is not a severe limitation, as it is considered in other works [14], and as we will see in the next section.

THEOREM 3.11 (Type preservation of $\leadsto^{lmgu}$). *Let $\mathcal{P}$ be a program such that left-hand sides of rules contain only transparent patterns. If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $e \leadsto^{lmgu^*}_{\theta} e'$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

So finally, with $\leadsto^{lmgu}$ we have obtained a narrowing relation that is able to ensure type preservation without using any type information at run-time. However, as we mentioned before, this comes at the price of losing completeness wrt. HO-CRWL, not only because we are restricted to using mgu's—which is not a severe restriction, as we will see later—but mainly because we are

not able to use the rules (VAct) and (VBind) any more, which are essential for generating binding for variable applications like those in Example 3.3. We will try to mitigate that problem in Section 4.

4. Reductions without Variable Applications

In this section we want to identify a class of programs in which $\leadsto^{lmgu}$ is sufficiently complete so it can perform well-typed narrowing derivations without losing well-typed solutions. As can be seen in the Lifting Lemma from [24], the restriction of the let-narrowing relation $\leadsto^{l}$ that only uses mgu's in each step is complete wrt. HO-CRWL. Therefore, we strongly believe that the restriction of $\leadsto^{lwt}$ using only mgu's is complete wrt. to the computation of well-typed solutions, although proving it is an interesting matter of future work. For this reason, in this section we are only concerned about determining under which conditions $\leadsto^{lmgu}$ is complete wrt. the restriction of $\leadsto^{lwt}$ to mgu's.

Our experience shows that although we only have to assure that neither (VAct) nor (VBind) are used, the characterization of such a family of programs is harder than expected. In Section 4.1 we show the different approaches tried, explaining their lacks, that led us to a restrictive condition—Section 4.2. This condition limits the expressiveness of the programs, hence we explore the possibilities of that class of programs in Section 4.3.

4.1 Naive approaches

Our first attempt follows the idea that if an expression does not contain any free HO variable (free variable with a functional type of the shape $\tau \to \tau'$) then neither (VAct) nor (VBind) can be used in a narrowing step. This result is stated in the following easy Lemma:

LEMMA 4.1 (Absence of HO variables). *Let e be an expression such that $wt_{\mathcal{A}}(e)$ and for every $X_i \in fv(e)$, $\mathcal{A}(X_i)$ is not a functional type. Then no step $e \leadsto^{l}_{\theta} e'$ can use (VAct) or (VBind).*

Our belief was that if an expression does not contain free HO variables and the program does not have extra HO variables, the resulting expression after a $\leadsto^{lmgu}$ step does not have free HO variables either. This is false, as the following example shows:

EXAMPLE 4.2. *Consider a constructor symbol bfc with type $bfc : (bool \to bool) \to BoolFunctContainer$ and the function f with type $f : BoolFunctContainer \to bool$ defined as $\{f\ (bfc\ F) \to F\ true\}$. We can perform the narrowing reduction*

$$f\ X \leadsto^{lmgu}_{\theta} F_1\ true$$

where $\theta \equiv [X \mapsto bfc\ F_1] = mgu(f\ X, f\ (bfc\ F_1))$. The free variable F_1 introduced has a functional type, however the original expression has not any free HO variable—X has the ground type $BoolFunctContainer$. Moreover, the program does not contain extra variables at all.

The previous example shows that not only free HO variables must be avoided in expressions, but also free variables with *unsafe* types as *BoolFunctContainer*. The reason is that patterns with unsafe types may contain HO variables. Those patterns can appear in left-hand sides of rules, so a narrowing step can unify a free variable with one of these patterns, thereby introducing free HO variables—notice that the unification of X and $bfc\ F_1$ introduces the free HO variable F_1 in the previous example. To formalize these intuitions we define the set of *unsafe* types as those for which problematic patterns can be formed:

DEFINITION 4.3 (Unsafe types). *The set of unsafe types wrt. a set of assumptions $\mathcal{A}$ ($UTypes_{\mathcal{A}}$) is defined as the least set of simple types verifying:*

1. *Functional types ($\tau \to \tau'$) are in $UTypes_{\mathcal{A}}$.*
2. *A simple type τ is in $UTypes_{\mathcal{A}}$ if there exists some pattern $t \in Pat$ with $\{\overline{X_n}\} = var(t)$ such that:*
   a) *$t \equiv \mathcal{C}[X_i]$ with $\mathcal{C} \neq [\,]$*
   b) *$\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$, for some $\overline{\tau_n}$*
   c) *$\tau_i \in UTypes_{\mathcal{A}}$.*

For brevity we say a variable X is *unsafe* wrt. $\mathcal{A}$ if $\mathcal{A}(X)$ is unsafe wrt. $\mathcal{A}$.

Clearly, if an expression does not contain free unsafe variables it does not contain free HO variables either, so by Lemma 4.1 neither (VAct) nor (VBind) could be used in a narrowing step. However, the absence of unsafe variables is not preserved after $\leadsto^{lmgu}$ steps even if the rules do not contain unsafe extra variables:

EXAMPLE 4.4. *Consider the symbols in Example 4.2 and a new function g defined as $\{g \to X\}$ with type $g : \forall\alpha.\alpha$. The extra variable X has the polymorphic type α in the rule for g, so it is safe. The expression $(f\ g)$ does not contain any unsafe variable, however we can make the reduction:*

$$f\ g \leadsto^{lmgu}_{\epsilon} f\ X_1 \leadsto^{lmgu}_{[X_1 \mapsto bfc\ F_1]} F_1\ true$$

The new variable X_1 introduced has type $BoolFunctContainer$, which is unsafe.

Example 4.4 shows that not only unsafe free variables must be avoided, but any expression of unsafe type which can be reduced to a free variable. In this case the problematic expression is g, which has type *BoolFunctContainer* and produces a free variable. Example 4.4 also shows that polymorphic extra variables are a source of problems, since they can take unsafe types depending on each particular use.

4.2 Restricted programs

Based on the problems detected in the previous section, we characterize a restricted class of programs and expressions to evaluate in which $\leadsto^{lwt}$ steps do not apply (VAct) and (VBind). First, we need that the expression to evaluate does not contain unsafe variables. Second, we forbid rules whose extra variables have unsafe types. Finally, we must also avoid polymorphic extra variables, since they can take different types, in particular unsafe ones. The restriction over programs is somehow tight: any program with functions using polymorphic extra variables are out of this family of programs, in particular the function *sublist* in Section 1 and other common functions using extra variables—see Section 4.3 for a detailed discussion.

In order to define formally this family of programs, we propose a restricted notion of well-typed programs. This notion is very similar to that in Definition 3.1, but using the restricted typing rule (Λ^r) for λ-abstractions in Figure 4, which avoids extra variables with polymorphic or unsafe types.

$$(\Lambda^r)\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \{\overline{Y_k : \tau'_k}\} \vdash e : \tau}{\mathcal{A} \vdash \lambda^r t.e : \tau_t \to \tau}$$

where $\{\overline{X_n}\} = var(t)$, $\{\overline{Y_k}\} = fv(\lambda^r t.e)$ such that $\overline{\tau'_k}$ are ground and safe wrt. $\mathcal{A}$.

Figure 4. Typing rule for restricted λ-abstractions

DEFINITION 4.5 (Well-typed restricted program). *A program rule $f \to e$ is well-typed restricted wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ where $\mathcal{A}(f) \succ_{var} \tau$, $\{\overline{X_n}\} = fv(e)$ and $\overline{\tau_n}$ are some ground and*

safe simple types wrt. $\mathcal{A}$. A program rule $(f\ \overline{p_n} \to e)$ (with $n > 0$) is well-typed restricted wrt. $\mathcal{A}$ iff $\mathcal{A} \vdash \lambda^r p_1 \ldots \lambda^r p_n.e : \tau$ with $\mathcal{A}(f) \succ_{var} \tau$. A program $\mathcal{P}$ is well-typed restricted wrt. $\mathcal{A}$ if all its rules are well-typed restricted wrt. $\mathcal{A}$.

If a program $\mathcal{P}$ is well-typed restricted wrt. $\mathcal{A}$ we write $wt^r_{\mathcal{A}}(\mathcal{P})$. Notice that for any $\mathcal{P}$ and $\mathcal{A}$ we have that $wt^r_{\mathcal{A}}(\mathcal{P})$ implies $wt_{\mathcal{A}}(\mathcal{P})$. For the rest of the section we will implicitly use this notion of well-typed restricted programs. Since the notion of well-typed substitution, and as a consequence the notion of $\leadsto^{lwt}$ step, is parameterized by the type system, then further mentions to $\leadsto^{lwt}$ in this section will refer to a relation slightly smaller than the one presented in Section 3.3: a variant of $\leadsto^{lwt}$ based on the type system from Definition 4.5. It is easy to see that this variant also preserves types in derivations. Therefore, although the following results are limited to this variant, they are still relevant.

The key property of well-typed restricted programs is that, starting from an expression without unsafe variables, the resulting expression of a $\leadsto^{lwt}$ reduction do not contain such variables either:

LEMMA 4.6 (Absence of unsafe variables). *Let e be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \leadsto^{lwt^*}_{\theta} e'$ then e' does not contain unsafe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

Notice that the use of mgu's in the $\leadsto^{lwt}$ steps is not necessary in the previous lemma, as the absence of unsafe variables is guaranteed by the well-typed substitution implicit in the definition of the $\leadsto^{lwt}$. Based on Lemma 4.6, it is easy to prove that $\leadsto^{lmgu}$ is complete to the restriction of $\leadsto^{lwt}$ to mgu's:

THEOREM 4.7 (Completeness of $\leadsto^{lmgu}$ wrt. $\leadsto^{lwt}$). *Let e be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \leadsto^{lwt^*}_{\theta} e'$ using mgu's in each step then $e \leadsto^{lmgu^*}_{\theta} e'$.*

Notice that completeness is assured even for programs having non transparent left-hand sides, as well-typedness of substitutions is guaranteed by $\leadsto^{lwt}$.

4.3 Expressiveness of the restricted programs

The previous section states the completeness of $\leadsto^{lmgu}$ wrt. $\leadsto^{lwt}$ for the class of well-typed restricted programs, when only mgu's are used in (Narr) steps. However this class leaves outside a number of interesting functions containing extra variables. For example, the *sublist* function in Section 1 is discarded. The reason is that extra variables of the rule—*Us* and *Vs*—must have type $[\alpha]$, which is not ground. A similar situation happens with other well-known polymorphic functions using extra variables, as the *last* function to compute the last element of a list—$last\ Xs \to cond\ (Ys +\!\!+ [E] == Xs)\ E$ [15]—or the function to compute the inverse of a function at some point—$inv\ F\ X \to cond\ (F\ Y == X)\ Y$. A consequence is that the class of well-typed restricted programs excludes many polymorphic functions using extra variables, since they usually have extra variables with polymorphic types.

However, not all functions using extra variables are excluded from the family of well-typed restricted programs. An example is the *even* function from Section 1 that checks whether a natural number is even or not. The whole rule has type $nat \to nat$ and it contains the extra variable Y of type *nat*, which is ground and safe, making the rule valid. Other functions handling natural numbers and using extra variables as $compound\ X \to cond\ (times\ M\ N == X)\ true$—where *times* computes the product of natural numbers—are also valid, since both M and N have type *nat*. Moreover, versions of the rejected polymorphic functions adapted to concrete ground types are also in the family of well-typed restricted programs. For example, functions as *sublistNat* or *lastBool* with types $[nat] \to [nat] \to bool$ and $[bool] \to bool$ and the same rules as their polymorphic versions are accepted. However, this is not a satisfactory solution: the generation of versions for the different types used implies duplication of code, which is clearly contrary to the degree of code reuse and generality offered by declarative languages—specially by means of polymorphic functions and the different input/output modes of function arguments.

The class of well-typed restricted programs is tighter than desired, and leaves out several interesting functions. Furthermore, for some of those functions—as *subslist* or *last*—we have not discovered any example where unsafe variables were introduced during reduction[4]. Therefore, we plan to further investigate the characterization of such a family in order to widen the number of programs accepted, while leaving out the problematic ones.

5. Type Preservation for Needed Narrowing

In this section we consider the type preservation problem for a simplified version of the Curry language, where features irrelevant to the scope of this paper are ignored, like constraints, encapsulated search, i/o, etc. Therefore we restrict ourselves to *simple Curry programs*, i.e., programs using only first-order patterns and transparent constructor symbols—which implies that all the patterns in left-hand sides are transparent. Besides, programs will be evaluated using the *needed narrowing* strategy [5] and performing resiuation for variable applications—which is simulated by dropping the rules (VAct) and (VBind). We have decided to focus on needed narrowing because it is the most popular on-demand evaluation strategy, and it is at the core of the majority of modern FLP systems.

We use a transformational approach to employ $\leadsto^{lmgu}$ to simulate an adaptation of the needed narrowing strategy for let-narrowing. We rely on two program transformations well-known in the literature. In the first one, we start with an arbitrary simple Curry program and transform it into an *overlapping inductively sequential* (OIS) program [1]. For programs in this class, an *overlapping definitional tree* is available for every function, that encodes the demand structure implied by the left-hand sides of its rules. Then we proceed with the second transformation, which takes an OIS program and transforms it into *uniform format* [31]: programs in which the left-hand sides of the rules for every function f have either the shape $f\ \overline{X}$ or $f\ \overline{X}\ (c\ \overline{Y})\ \overline{Z}$.

There are other well-known transformations from general programs to OIS programs—for example [10]—but we have chosen the transformation in Definition 5.1—which is similar to the transformation in [2], but now extended to generate type assumptions—because of its simplicity. The transformation processes each function independently: it takes the set of rules $\mathcal{P}_f$ for each function f and returns a pair composed by the transformed rules and a set of assumptions for the auxiliary fresh functions introduced by the transformation.

DEFINITION 5.1 (Transformation to OIS). *Let $\mathcal{P}_f \equiv \{f\ \overline{t^1_n} \to e^1, \ldots, f\ \overline{t^m_n} \to e^m\}$ be a set of m program rules for the function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If f is an OIS function, $OIS(\mathcal{P}_f) = (\mathcal{P}_f, \emptyset)$. Otherwise $OIS(\mathcal{P}_f) = (\{f_1\ \overline{t^1_n} \to e^1, \ldots, f_m\ \overline{t^m_n} \to e^m, f\ \overline{X_n} \to f_1\ \overline{X_n}?\ldots?f_m\ \overline{X_n}\}, \{f_m : \mathcal{A}(f)\})$, where $?$ is the non-determistic choice function defined with the rules $\{X?Y \to X, X?Y \to Y\}$.*

[4] The function *inv* can introduce HO variables when combined with a constant function as $zero\ X \to z$ with type $\forall\alpha.\alpha \to nat$: $(inv\ zero\ z)\ true \leadsto^{lwt^*}_{\theta} Y_1\ true$, where Y_1 is clearly unsafe.

The following result states that the transformation OIS preserves types. Notice that any other transformation to OIS format that also preserves types could be used instead.

THEOREM 5.2 ($OIS(\mathcal{P}_f)$ well-typedness). *Let $\mathcal{P}_f$ be a set of program rules for the same function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $OIS(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$.*

After the transformation the assumption for f remains the same and the new assumptions refer to fresh function symbols. Therefore, it is easy to see that the previous result is also valid for programs with several functions.

Now, to transform the program from OIS into uniform format we use the following transformation, which is a slightly variant of the transformation in [31]. Like in the previous transformation, we treat each function independently, returning the translated rules together with the extra assumptions for the auxiliary functions.

DEFINITION 5.3 (Transformation to uniform format). *Let $\mathcal{P}_f \equiv \{f\ \overline{t_n^1} \to e^1, \ldots, f\ \overline{t_n^m} \to e^m\}$ be an OIS program of m program rules for a function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{P}_f$ is already in uniform format, then $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \emptyset)$. Otherwise, we take the uniformly demanded position[5] o and split $\mathcal{P}_f$ into r sets $\overline{\mathcal{P}_r}$ containing the rules in $\mathcal{P}_f$ with the same constructor symbol in position o. Then $\mathcal{U}(\mathcal{P}_f) = (\bigcup_{i=1}^{r} \mathcal{P}'_i \cup \mathcal{P}'', \bigcup_{i=1}^{r} \mathcal{A}'_i \cup \mathcal{A}'')$ where:*

- $\mathcal{U}(\mathcal{P}_i^o) = (\mathcal{P}'_i, \mathcal{A}'_i)$
- *c_i is the constructor symbol in position o in the rules of $\mathcal{P}_i$, with $ar(c_i) = k_i$*
- *$\mathcal{P}_i^o$ is the result of replacing the function symbol f in $\mathcal{P}_i$ by $f_{(c_i,o)}$ and flattening the patterns in position o in the rules, i.e., $f\ \overline{t_j}\ (c_i\ \overline{t'_{k_i}})\ \overline{t''_l} \to e$ is replaced by $f_{(c_i,o)}\ \overline{t_j}\ \overline{t'_{k_i}}\ \overline{t''_l} \to e$*
- *$\mathcal{P}'' \equiv \{f\ \overline{X_j}\ (c_1\ \overline{Y_{k_1}})\ \overline{Z_l} \to f_{(c_1,o)}\ \overline{X_j}\ \overline{Y_{k_1}}\ \overline{Z_l}, \ldots,$ $f\ \overline{X_j}\ (c_r\ \overline{Y_{k_r}})\ \overline{Z_l} \to f_{(c_r,o)}\ \overline{X_j}\ \overline{Y_{k_r}}\ \overline{Z_l}\}$, with $\overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l}$ pairwise distinct fresh variables such that $j + l + 1 = n$*
- *$\mathcal{A}'' \equiv \{f_{(c_1,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau'_{k_1}} \to \overline{\tau_l} \to \tau, \ldots, f_{(c_r,o)} : \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau'_{k_r}} \to \overline{\tau_l} \to \tau\}$ where $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$ and $\mathcal{A} \oplus \{\overline{Y_{k_i} : \tau'_{k_i}}\} \vdash c_i\ \overline{Y_{k_i}} : \tau'$. Notice that since constructor symbols c_i are transparent, these $\overline{\tau'_{k_i}}$ do exist and are univocally fixed.*

This transformation also preserves types. For the same reasons as before, the following result is also valid for programs with several functions.

THEOREM 5.4 ($\mathcal{U}(\mathcal{P}_f)$ well-typedness). *Let $\mathcal{P}_f$ be a set of program rules for the same overlapping inductive sequential function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$.*

We have just seen that we can transform an arbitrary program into uniform format while preserving types. The preservation of the semantics is also stated in [2, 31]. Although these results have been proved in the context of term rewriting, we strongly believe that they remain valid for the call-time choice semantics of the HO-CRWL framework. Similarly, we are strongly confident that the completeness of narrowing with mgu's over a uniform program wrt. needed narrowing over the original program [31] is also valid in the framework of let-narrowing. Combining those results with the type preservation results for $\leadsto^{lmgu}$ and the program transformations—Theorems 3.11, 5.2 and 5.4—we can conclude that a simulation of the evaluation of simple Curry programs using $\leadsto^{lmgu}$ based on the transformations above, is safe wrt. types.

[5] A position in which all the rules in $\mathcal{P}_f$ have a constructor symbol. Notice that this position will always exist because $\mathcal{P}_f$ is an OIS program [1].

6. Conclusions and Future Work

In this paper we have tackled the problem of type preservation for FLP programs with extra variables. As extra variables lead to the introduction of fresh free variables during the computations, we have decided to use the let-narrowing relation $\leadsto^{l}$—which is sound and complete wrt. HO-CRWL, a standard semantics for FLP—as the operational mechanism for this paper. This is also a natural choice because let-narrowing reflects the behaviour of current FLP systems like Toy or Curry, that provide support for extra and logical variables instead of reducing expressions by rewriting only.

The other main technical ingredient of the paper is a novel variation of Damas-Milner type system that has been enhanced with support for extra variables. Based on this type system we have defined the well-typed let-narrowing relation $\leadsto^{lwt}$, which is a restriction of let-narrowing that preserves types. To the best of our knowledge, this is the first paper proposing a polymorphic type system for FLP programs with logical and extra variables such that type preservation is formally proved. As we have seen in Example 3.2 from Section 3 the type systems from [21, 22] lose type preservation when extra variables are introduced. In [4], another remarkable previous work, the proposed type system only supports monomorphic functions and extra variables are not allowed. In [14] only programs with transparent patterns and without extra variables are considered, and functional arguments in data constructors are forbidden. Nevertheless, any of those programs is supported by our $\leadsto^{lwt}$ relation, which has to carry type information at run-time, but just like the extension of the Constructor-based Lazy Narrowing Calculus proposed in [14].

The relevance of Theorem 3.8, which states that $\leadsto^{lwt}$ preserves types, lies in the clarification it makes of the problem of type preservation on narrowing reductions with programs with extra variables. Relying on the abstract notion of well-typed substitution, which is parametrized by the type system and independent of any concrete operational mechanism, we have isolated a sufficient condition that ensures type preservation when imposed to the unifiers used in narrowing derivations. This contrasts with previous works like [14]—the closest to the present paper—in which a most general unifier was implicitly computed. Moreover, $\leadsto^{lwt}$ preserves types for arbitrary programs, something novel in the field of type systems in FLP—to the best of our knowledge. Hence, $\leadsto^{lwt}$ is an intended ideal narrowing relation that always preserves types, but that can only be directly realized by using type checks at run-time. Therefore, $\leadsto^{lwt}$ is most useful when used as a reference to define some imperfect but more practical materializations of it—subrelations of $\leadsto^{lwt}$—that only work for certain program classes but also preserve types while avoiding run-time type checks. An example of this is the relation $\leadsto^{lmgu}$, whose applicability is restricted to programs with transparent patterns, and that also lacks some completeness. This relation is based on two conditions imposed over $\leadsto^{l}$ steps: mgu's are used in every (Narr) step; and the rules (VAct) and (VBind) are avoided. While the former is not a severe restriction—as $\leadsto^{l}$ is complete wrt. HO-CRWL even if only mgu's are allowed as unifiers [24]—the latter is more problematic, because then $\leadsto^{lmgu}$ is not able to generate bindings for variable applications. To mitigate this weakness we have investigated how to prevent the use of (VAct) and (VBind) in $\leadsto^{lwt}$ derivations. After some preliminary attempts that witness the difficulty of the task, and also give valuable insights about the problem, we have finally characterized a class of programs in which these bindings for variable applications are not needed, and studied their expressiveness. Then we have applied the results obtained so far for proving the type preservation for a simplified version of the Curry language. HO-patterns are not supported in Curry, which treats functions as black boxes [4]. Therefore Curry programs do not intend to gen-

erate solutions that include bindings for variable applications, and so the rules (VAct) and (VBind) will not be used to evaluate these programs. Besides, in Curry all the constructors are transparent, and the needed narrowing on-demand strategy is employed in most implementations of Curry. We have used two well-known program transformations to simulate the evaluation of Curry programs with an adaptation of needed narrowing for let-narrowing. Then we have proved that both transformations preserve types which, combined with the type preservation of $\leadsto^{lmgu}$, implies that our proposed simulation of needed narrowing also preserves types.

Regarding future work, we would like to look for new program classes more general than the one presented in Section 4 because, as we pointed out at the end of that section, the proposed class is quite restrictive and it forbids several functions that we think are not dangerous for the types.

Another interesting line of future work would deal with the problems generated by opaque pattens, as we did in [22] for the restricted case where we drop logical and extra variables. We think that an approach in the line of existential types [20] that, contrary to [22], forbids pattern matching over existential arguments, is promising. This has to do with the parametricy property of types systems [30], which is broken in [22] as we allowed matching on existential arguments, and which is completely abandoned from the very beginning in [21]. In fact it was already detected in [14] that the loss of parametricity leads to the loss of type preservation in narrowing derivations—in that paper instead of parametricity the more restrictive property of type generality is considered. All that suggests that our first task regarding this subject should be modifying our type system from [22] to recover parametricity by following an approach to opacity closer to standard existential types.

Acknowledgments

This work has been partially supported by the Spanish projects FAST-STAMP (TIN2008-06622-C03-01), PROMETIDOS-CM (S2009TIC-1465) and GPD-UCM (UCM-BSCH-GR35/10-A-910502).

References

[1] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. 6th Int. Conf. on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.

[2] S. Antoy. Constructor based conditional narrowing. In *Proc. 3rd Int. Conf. on Principles and Practice of Declarative Programming (PPDP'01)*, pages 199–206. ACM, 2001.

[3] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.

[4] S. Antoy and A. Tolmach. Typed higher-order narrowing without higher-order strategies. In *Proc. 4th Int. Symp. on Functional and Logic Programming (FLOPS'99)*, pages 335–352. Springer LNCS 1722, 1999.

[5] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47:776–822, July 2000.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] B. Brassel. Two to three ways to write an unsafe type cast without importing unsafe - Curry mailing list. `http://www.informatik.uni-kiel.de/~curry/listarchive/0705.html`, May 2008.

[8] B. Brassel, S. Fischer, M. Hanus and F. Reck Transforming Functional Logic Programs into Monadic Functional Programs In *Proc. 19th Int. Work. on Functional and (Constraint) Logic Programming (WFLP'10)*, Springer LNCS 6559, pages 30–47, 2011.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'82)*, pages 207–212. ACM, 1982.

[10] R. del Vado Vírseda. Estrategias de estrechamiento perezoso. Master's thesis, Universidad Compluetense de Madrid, 2002.

[11] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard. Reference Manual*. Springer, 1996.

[12] J. González-Moreno, T. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. 14th Int. Conf. on Logic Programming (ICLP'97)*, pages 153–167. MIT Press, 1997.

[13] J. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1): 47–87, 1999.

[14] J. González-Moreno, T. Hortalá-González, and M. Rodríguez-Artalejo. Polymorphic types in functional logic programming. *Journal of Functional and Logic Programming*, 2001(1), July 2001.

[15] M. Hanus. Multi-paradigm declarative languages. In *Proc. 23rd Int. Conf. on Logic Programming (ICLP'07)*, pages 45–75. Springer LNCS 4670, 2007.

[16] M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). `http://www.informatik.uni-kiel.de/~curry/report.html`, March 2006.

[17] M. Hanus and F. Steiner. Type-based nondeterminism checking in functional logic programs. In *Proc. 2nd. Inf. Conf. Principles and Practice of Declarative Programming. (PDP 2000)*, pages 202–213, ACM, 2000.

[18] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. 3rd ACM SIGPLAN Conf. on History of Programming Languages (HOPL III)*, pages 12–1–12–55. ACM, 2007.

[19] J.-M. Hullot. Canonical forms and unification. In *Proc. 5th Conf. on Automated Deduction (CADE-5)*, pages 318–334. Springer LNCS 87, 1980.

[20] K. Läufer and M. Odersky. Polymorphic type inference and abstract data types. *ACM Trans. Program. Lang. Syst.*, 16:1411–1430, 1994.

[21] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. Liberal typing for functional logic programs. In *Proc. 8th Asian Symp. on Programming Languages and Systems (APLAS'10)*, pages 80–96. Springer LNCS 6461, 2010.

[22] F. López-Fraguas, E. Martin-Martin, and J. Rodríguez-Hortalá. New results on type systems for functional logic programming. In *Proc. 18th Int. Workshop on Functional and (Constraint) Logic Programming (WFLP'09), Revised Selected Papers*, pages 128–144. Springer LNCS 5979, 2010.

[23] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. 10th Int. Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[24] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th Int. Symp. on Functional and Logic Programming (FLOPS'08)*, pages 147–162. Springer LNCS 4989, 2008.

[25] W. Lux. Adding Haskell-style overloading to Curry. In *Workshop of Working Group 2.1.4 of the German Computing Science Association GI*, pages 67–76, 2008.

[26] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[27] E. Martin-Martin. Advances in type systems for functional logic programming. Master's thesis, Universidad Complutense de Madrid, July 2009. `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`.

[28] E. Martin-Martin. Type classes in functional logic programming. In *Proc. 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'11)*, pages 121–130. ACM, 2011.

[29] M. Rodríguez-Artalejo. Functional and constraint logic programming. In *Constraints in Computational Logics*, pages 202–270. Springer LNCS 2002, 2001.

[30] P. Wadler. Theorems for free! In *Proc. 4th Int. Conf. on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 347–359. ACM, 1989.

[31] F. Zartmann. Denotational abstract interpretation of functional logic programs. In *Proc. 4th Int. Symp. on Static Analysis (SAS'97)*, pages 141–159. Springer LNCS 1302, 1997.

A. Proofs

The following theorem contains some interesting properties of the typing relation $\vdash$ in Figure 3 that will be used intensively in this appendix. [27] contains detailed proofs for these properties for a very similar type relation whose (Λ) rule does not handle λ-abstractions with extra variables. However, the extension of those proofs to support the new flavour of λ-abstractions is straightforward and has been omitted.

THEOREM A.1 (Properties of the typing relation).

a) If $\mathcal{A} \vdash e : \tau$ then $\mathcal{A}\pi \vdash e : \tau\pi$, for any $\pi \in TSubst$.
b) Let s be a symbol not occurring in e. Then $\mathcal{A} \vdash e : \tau \iff \mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$, for any σ.
c) If $\mathcal{A} \oplus \{X : \tau_x\} \vdash e : \tau$ and $\mathcal{A} \oplus \{X : \tau_x\} \vdash e' : \tau_x$ then $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.

A.1 Proof of Theorem 3.8: Type preservation of $\rightsquigarrow^{lwt}$

In order to prove Type Preservation, we need the following auxiliary result regarding type preservation with contexts and well-typed substitutions:

LEMMA A.2. *Consider $\mathcal{A} \vdash \mathcal{C}[e] : \tau$ containing the subderivation $\mathcal{A} \oplus \overline{[Z_m/\tau_m]} \vdash e : \tau_e$ (being $\overline{[Z_m/\tau_m]}$ the set of assumptions generated for bound variables) and $\mathcal{A} \oplus \overline{[Z_m/\tau_m]} \vdash e' : \tau_e$. Define $\mathcal{A}_0 \equiv \mathcal{A}$ and $\mathcal{A}_i \equiv \mathcal{A}_{i-1} \oplus \{Z_i : \tau_i\}$ for $i \in [1..m]$. In that conditions, if we have a data derivation θ such that $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for every $i \in [0..m]$ and $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$ then $\mathcal{A} \vdash \mathcal{C}\theta[e'] : \tau$.*

Proof By induction on the structure of $\mathcal{C}$.

BASE CASE:

$\boxed{\mathcal{C} \equiv [\,]}$ In this case $\mathcal{A}_m \equiv \mathcal{A} \oplus \overline{[Z_m/\tau_m]}$, so $\mathcal{A}_m \vdash \mathcal{C}[e] : \tau$ with $\mathcal{C}[e] \equiv e$ and $\tau \equiv \tau_e$. By hypothesis we have $\mathcal{A} \oplus \overline{[Z_m/\tau_m]} \vdash e : \tau$, so $\mathcal{A} \oplus \overline{[Z_m/\tau_m]} \vdash \mathcal{C}[e']$.

INDUCTIVE STEP:

$\boxed{\mathcal{C} \equiv \mathcal{C}'\ e_2}$ In this case we have

$$\text{(APP)}\frac{\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau' \to \tau \qquad \mathcal{A}_n \vdash e_2 : \tau'}{\mathcal{A}_n \vdash \mathcal{C}'[e]\ e_2 : \tau}$$

for a $\mathcal{A}_n$ containing assumptions for the bound variables reached up to this point. By the hypothesis we have that $wt_{\mathcal{A}_n}(\theta|_{fv(\mathcal{C})})$, so for any free variable $X \in e_2$ the substitution θ verifies $\mathcal{A}_n \vdash X\theta : \mathcal{A}_n(X)$. Then by Theorem A.1-c) we have $\mathcal{A}_n \vdash e_2\theta : \tau'$. From the hypothesis we know that the derivation $\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau' \to \tau$ contains a subderivation $\mathcal{A} \oplus \overline{[Z_m/\tau_m]} \vdash e : \tau_e$ and $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i \in [n..m]$, so $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C}')})$ for any $i \in [n..m]$ as $fv(\mathcal{C}') \subseteq fv(\mathcal{C})$. From the hypothesis we also have $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$, so $dom(\theta) \cap bv(\mathcal{C}') = \emptyset$ since $bv(\mathcal{C}'\ e_2) = bv(\mathcal{C}')$. Then by the Induction Hypothesis $\mathcal{A}_n \vdash \mathcal{C}'\theta[e'] : \tau' \to \tau$ and since $\mathcal{C}'\theta[e']\ e_2\theta \equiv \mathcal{C}\theta[e']$ we have:

$$\text{(APP)}\frac{\mathcal{A}_n \vdash \mathcal{C}'[e'] : \tau' \to \tau \qquad \mathcal{A}_n \vdash e_2\theta : \tau'}{\mathcal{A}_n \vdash \mathcal{C}\theta[e'] : \tau}$$

$\boxed{\mathcal{C} \equiv \mathcal{C}'\ e_2}$ Similar to the previous case.

$\boxed{\mathcal{C} \equiv let\ Z_n = \mathcal{C}'\ in\ e_2}$ We have a derivation

$$\text{(LET)}\frac{\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau_n \qquad \mathcal{A}_{n+1} \vdash e_2 : \tau}{\mathcal{A}_n \vdash let\ Z_n = \mathcal{C}'[e]\ in\ e_2 : \tau}$$

where $\mathcal{A}_n$ contains assumptions for the bound variables reached up to this point and $\mathcal{A}_{n+1} \equiv \mathcal{A}_n \oplus \{Z_n : \tau_n\}$ by definition. By the hypothesis we have that $wt_{\mathcal{A}_{n+1}}(\theta|_{fv(\mathcal{C})})$, so for any free variable $X \in e_2$ the substitution θ verifies $\mathcal{A}_{n+1} \vdash X\theta : \mathcal{A}_{n+1}(X)$. Then by Theorem A.1-c) we have $\mathcal{A}_{n+1} \vdash e_2\theta : \tau$. From the hypothesis we know that the derivation $\mathcal{A}_n \vdash \mathcal{C}'[e] : \tau_n$ contains a subderivation $\mathcal{A} \oplus \overline{[Z_m/\tau_m]} \vdash e : \tau_e$ and $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i \in [n..m]$, so $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C}')})$ for any $i \in [n..m]$ as $fv(\mathcal{C}') \subseteq fv(\mathcal{C})$. Also from the hypothesis we have have $dom(\theta) \cap bv(let\ Z_n = \mathcal{C}'\ in\ e_2) = \emptyset$, so $dom(\theta) \cap bv(\mathcal{C}') = \emptyset$ since $bv(let\ Z_n = \mathcal{C}'\ in\ e_2) = bv(\mathcal{C}')$. Then by the Induction Hypothesis $\mathcal{A}_n \vdash \mathcal{C}'\theta[e'] : \tau_n$, and considering that $let\ Z_n = \mathcal{C}'\theta[e']\ in\ e_2\theta \equiv \mathcal{C}\theta[e']$ we have:

$$\text{(LET)}\frac{\mathcal{A}_n \vdash \mathcal{C}'\theta[e'] : \tau_n \qquad \mathcal{A}_{n+1} \vdash e_2\theta : \tau}{\mathcal{A}_n \vdash \mathcal{C}\theta[e'] : \tau}$$

$\boxed{\mathcal{C} \equiv let\ Z_n = e_1\ in\ \mathcal{C}'}$ Similar to the previous case, with two main differences. The first one is that $dom(\theta) \cap bv(\mathcal{C}') = \emptyset$ because $bv(\mathcal{C}') \subseteq bv(let\ Z_n = e_1\ in\ \mathcal{C}')$. The second difference is that $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C}')})$ for any $i \in [n+1..m]$ because $wt_{\mathcal{A}_i}(\theta|_{(fv(\mathcal{C}') \setminus \{Z_n\})})$ for any $i \in [n+1..m]$ as $fv(\mathcal{C}') \setminus \{Z_n\} \subseteq fv(\mathcal{C})$, and using the fact that $Z_n \notin dom(\theta)$—since $Z_n \in bv(\mathcal{C})$—then $\theta|_{(fv(\mathcal{C}') \setminus \{Z_n\})} \equiv \theta|_{(fv(\mathcal{C}')}$

Using the previous lemma, we can now prove Type Preservation:

Theorem 3.8 (Type preservation of $\rightsquigarrow^{lwt}$)
If $wt_{\mathcal{A}}(\mathcal{P})$, $e \rightsquigarrow_\theta^{lwt^*} e'$ and $\mathcal{A} \vdash e : \tau$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A} \oplus \mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.

Proof We first prove the result for one step $e \rightsquigarrow_\theta^{lwt} e'$ by case distinction over the used rule. Notice that $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ is true by the hypothesis $e \rightsquigarrow_\theta^{lwt^*} e'$, so we only have to prove $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$. The proofs for the cases (LetIn), (Bind), (Elim), (Flat) and (LetAp) are the same as those in [27]. For the remaining cases:

- (Narr)
  For the sake of simplicity we will prove the case for a function applied to 2 patterns, but the proof for any number of arguments follows the same ideas. We have a narrowing step $f\ t_1\ t_2 \rightsquigarrow_\theta^{lwt} r\theta$ for a fresh variant $(f\ p_1\ p_2 \to r) \in \mathcal{P}$ and a well-typed substitution θ such that $(f\ p_1\ p_2)\theta \equiv (f\ t_1\ t_2)\theta$. From the hypothesis we have:

$$\text{(APP)}\frac{\text{(APP)}\dfrac{\mathcal{A} \vdash f : \tau_1 \to \tau_2 \to \tau \quad \mathcal{A} \vdash t_1 : \tau_1}{\mathcal{A} \vdash f\ t_1 : \tau_2 \to \tau} \qquad \mathcal{A} \vdash t_2 : \tau_2}{\mathcal{A} \vdash f\ t_1\ t_2 : \tau}$$

  Since the rule is well-typed, we also have a type derivation:

$$\text{(}\Lambda\text{)}\frac{\mathcal{A} \oplus \mathcal{A}_1 \vdash p_1 : \tau_1' \qquad \text{(}\Lambda\text{)}\dfrac{\mathcal{A} \oplus \mathcal{A}_1 \oplus \mathcal{A}_2 \vdash p_2 : \tau_2' \quad (A)\ \mathcal{A} \oplus \mathcal{A}_1 \oplus \mathcal{A}_2 \vdash r : \tau'}{\mathcal{A} \oplus \mathcal{A}_1 \vdash \lambda p_2.r : \tau_2' \to \tau'}}{\mathcal{A} \vdash \lambda p_1.\lambda p_2.r : \tau_1' \to \tau_2' \to \tau'}$$

  where $\mathcal{A}_1$ and $\mathcal{A}_2$ are assumptions over $var(p_1) \cup fv(\lambda p_1.\lambda p_2.r)$ and $var(p_2) \cup fv(\lambda p_2.r)$ resp. and $\tau_1' \to \tau_2' \to \tau'$ is a variant of $\mathcal{A}(f)$. Since $\tau_1 \to \tau_2 \to \tau$ is a generic instance of $\mathcal{A}(f)$ then $(\tau_1' \to \tau_2' \to \tau')\pi \equiv \tau_1 \to \tau_2 \to \tau$ for some type substitution π whose domain are fresh type variables from the variant.
  By Theorem A.1-a) we can apply the type substitution π to (A):

$$(A')\ \mathcal{A} \oplus \mathcal{A}_1\pi \oplus \mathcal{A}_2\pi \vdash r : \tau$$

  noticing that $\tau'\pi \equiv \tau$ and $\mathcal{A}\pi \equiv \pi$ since the domain of π are fresh type variables. The set of assumptions associated to this step is $\mathcal{A}' \equiv \mathcal{A}_1\pi \oplus \mathcal{A}_2\pi$, so by the premise $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ and we can use Theorem A.1-c) to apply θ in (A'):

$$(A'')\ \mathcal{A} \oplus \mathcal{A}' \vdash r\theta : \tau$$

- (VAct)
  For the sake of conciseness, we consider the simplified step $X\ t_2 \rightsquigarrow_\theta^{lwt} r\theta$ for a fresh variant $(f\ p_1\ p_2 \to r) \in \mathcal{P}$ such that $(X\ t_2)\theta \equiv f\ p_1\theta\ p_2\theta$. From $wt_{\mathcal{A}}(e)$ we have:

$$\text{(APP)}\frac{\mathcal{A} \vdash X : \tau_2 \to \tau \quad \mathcal{A} \vdash t_2 : \tau_2}{\mathcal{A} \vdash X\ t_2 : \tau}$$

  Since $wt_{\mathcal{A}}(\mathcal{P})$ then the rule is well-typed, and we also have a type derivation:

$$\text{(}\Lambda\text{)}\frac{\mathcal{A} \oplus \mathcal{A}_1 \vdash p_1 : \tau_1' \qquad \text{(}\Lambda\text{)}\dfrac{\mathcal{A} \oplus \mathcal{A}_1 \oplus \mathcal{A}_2 \vdash p_2 : \tau_2' \quad (A)\ \mathcal{A} \oplus \mathcal{A}_1 \oplus \mathcal{A}_2 \vdash r : \tau'}{\mathcal{A} \oplus \mathcal{A}_1 \vdash \lambda p_2.r : \tau_2' \to \tau'}}{\mathcal{A} \vdash \lambda p_1.\lambda p_2.r : \tau_1' \to \tau_2' \to \tau'}$$

  where $\mathcal{A}_1$ and $\mathcal{A}_2$ are set of assumptions for the variables in $var(p_1) \cup fv(\lambda p_1.\lambda p_2.r)$ and $var(p_2) \cup fv(\lambda p_2.r)$ resp. Since the associated set of assumptions is defined by premise, we know that $\mathcal{A}' \equiv \mathcal{A}_1\pi \oplus \mathcal{A}_2\pi$ for some π such that $(\tau_2' \to \tau')\pi \equiv \tau_2 \to \tau$ and $fv(\mathcal{A}) \cap dom(\pi) = \emptyset$. By Theorem A.1-a) we can apply the type substitution π to (A):

$$(A')\ \mathcal{A} \oplus \mathcal{A}_1\pi \oplus \mathcal{A}_2\pi \vdash r : \tau$$

  noticing that $\tau'\pi \equiv \tau$ and $\mathcal{A}\pi \equiv \pi$. By premise $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, so we can use Theorem A.1-c) to apply θ in (A'):

$$\mathcal{A} \oplus \mathcal{A}' \vdash r\theta : \tau$$

- (VBind)

  The step is $let\ X = e_1\ in\ e_2 \rightsquigarrow_\theta^{lwt} e_2\theta[X \mapsto e_1\theta]$, where $e_1 \notin Pat$, $e_1\theta \in Pat$ and $X \notin dom(\theta) \cup vran(\theta)$. From $wt_{\mathcal{A}}(e)$ we have:

$$\text{(LET)}\frac{(A)\ \mathcal{A} \vdash e_1 : \tau_x \quad (B)\ \mathcal{A} \oplus \{X : \tau_x\} \vdash e_2 : \tau}{\mathcal{A} \vdash let\ X = e_1\ in\ e_2 : \tau}$$

  The set of assumptions $\mathcal{A}'$ associate to the step contains assumptions over the new variables introduced by θ, so they cannot appear in e_1 or e_2. Then, by Theorem A.1-b) we can add them to (A) and (B):

$$(A')\ \mathcal{A} \oplus \mathcal{A}' \vdash e_1 : \tau_x$$
$$(B')\ \mathcal{A} \oplus \mathcal{A}' \oplus \{X : \tau_x\} \vdash e_2 : \tau$$

Since $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ then by Theorem A.1-c) and (A') we have

$$(A'')\ \mathcal{A}\oplus\mathcal{A}'\vdash e_1\theta:\tau_x$$

We can assume that $X\notin fv(e_1)$ since our let-expressions are not recursive. By the conditions of the step we know that $X\notin dom(\theta)\cup vran(\theta)$, so $X\notin e_1\theta$ and by Theorem A.1-b) we can add the assumption for X to the derivation (A''):

$$(A''')\ \mathcal{A}\oplus\mathcal{A}'\oplus\{X:\tau_x\}\vdash e_1\theta:\tau_x$$

Since $X\notin dom(\theta)\cup vran(\theta)$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ implies $wt_{\mathcal{A}\oplus\mathcal{A}'\oplus\{X:\tau_x\}}(\theta)$, and by Theorem A.1-c) and (B') we have:

$$(B'')\ \mathcal{A}\oplus\mathcal{A}'\oplus\{X:\tau_x\}\vdash e_2\theta:\tau$$

Finally, by A.1-c) and (A''') we can apply the substitution $[X\mapsto e_1\theta]$ to (B''):

$$(B''')\ \mathcal{A}\oplus\mathcal{A}'\oplus\{X:\tau_x\}\vdash e_2\theta[X\mapsto e_1\theta]:\tau$$

Since $e_2\theta[X\mapsto e_1\theta]$ does not contain X, by Theorem A.1-b) we can remove the assumption over it, obtaining:

$$\mathcal{A}\oplus\mathcal{A}'\vdash e_2\theta[X\mapsto e_1\theta]:\tau$$

- (Contx)

  We have a narrowing step $\mathcal{C}[e]\rightsquigarrow^{lwt}_{\theta}\mathcal{C}\theta[e']$ for $\mathcal{C}\neq[\,]$, $e\rightsquigarrow^{l}_{\theta}e'$ using any of the previous rules. By hypothesis we have $\mathcal{A}\vdash\mathcal{C}[e]:\tau$, so in this derivation there is a subderivation $\mathcal{A}\oplus\mathcal{A}_b\vdash e:\tau_e$ for some $\mathcal{A}_b\equiv\{\overline{Z_m:\tau_m}\}$ containing assumptions for the bound variables in $\mathcal{C}$.
  - If the step $e\rightsquigarrow^{lwt}_{\theta}e'$ uses a rule different from (Narr), (Vact) or (VBind), then $\theta\equiv\epsilon$ and by the proof of those cases $\mathcal{A}\oplus\mathcal{A}_b\vdash e':\tau_e$ (since $\mathcal{A}'\equiv\emptyset$). Then by Lemma 6 in [27] we can replace an expression inside a context by any other of the same type, so $\mathcal{A}\vdash\mathcal{C}\epsilon[e']:\tau$.
  - If the step $e\rightsquigarrow^{l}_{\theta}e'$ uses (Narr) or (VAct) then we have that *i)* $dom(\theta)\cap bv(\mathcal{C})=\emptyset$ and *ii)* the step uses a fresh variant $(f\ \overline{p_n}\to r)\in\mathcal{P}$ such that $vran(\theta|_{\setminus var(\overline{p_n})})\cap bv(\mathcal{C})=\emptyset$. We have $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$ by hypothesis and $\overline{Z_m}$ are bound variables which can be assumed not to appear in $\mathcal{A}$, so $wt_{\mathcal{A}\oplus\mathcal{A}_b\oplus\mathcal{A}'}(\theta)$. Therefore we have $e\rightsquigarrow^{lwt}_{\theta}e'$, and by the proof of one step we have $\mathcal{A}\oplus\mathcal{A}_b\oplus\mathcal{A}'\vdash e':\tau$. The set $\mathcal{A}'$ contains assumptions over new data variables introduced in the step, and $\mathcal{A}_b$ contains assumptions over bound variables so $dom(\mathcal{A}_b)\cap dom(\mathcal{A}')=\emptyset$ and $\mathcal{A}\oplus\mathcal{A}_b\oplus\mathcal{A}'\vdash e':\tau$ implies $\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b\vdash e':\tau$. For the same reasons, $wt_{\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b}(\theta)$. As the variables in $\mathcal{A}'$ can appear neither in e nor in $\mathcal{C}[e]$—and $dom(\mathcal{A}_b)\cap dom(\mathcal{A}')=\emptyset$—then by Theorem A.1-b) we have $\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b\vdash e:\tau_e$ and $\mathcal{A}\oplus\mathcal{A}'\vdash\mathcal{C}[e]:\tau$. Define $\mathcal{A}_0\equiv\mathcal{A}\oplus\mathcal{A}'$ and $\mathcal{A}_i\equiv\mathcal{A}_{i-1}\oplus\{Z_i:\tau_i\}$ for any $i\in[1..m]$. From the fact that $\overline{p_n}$ are fresh variables and *ii)* we can conclude that $var(X\theta)\cap bv(\mathcal{C})=\emptyset$ for every $X\in fv(\mathcal{C})$. We can assume that $bv(\mathcal{C})\cap fv(\mathcal{C})=\emptyset$, so by Theorem A.1-b) and $wt_{\mathcal{A}\oplus\mathcal{A}'\oplus\mathcal{A}_b}(\theta)$ it is clear that $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i\in[0..m]$. Finally, by Lemma A.2 we have that $\mathcal{A}\oplus\mathcal{A}'\vdash\mathcal{C}\theta[e']:\tau$.
  - If the step $e\rightsquigarrow^{lwt}_{\theta}e'$ uses (VBind) then *i)* $dom(\theta)\cap bv(\mathcal{C})=\emptyset$ and *ii)* $vran(\theta)\cap bv(\mathcal{C})=\emptyset$. The proof follows a similar reasoning to the previous case: from *ii)* and assuming $bv(\mathcal{C})\cap fv(\mathcal{C})=\emptyset$ we have $wt_{\mathcal{A}_i}(\theta|_{fv(\mathcal{C})})$ for any $i\in[0..m]$. Therefore by Lemma A.2 we have $\mathcal{A}\oplus\mathcal{A}'\vdash\mathcal{C}\theta[e']:\tau$.

The proof for any number of steps proceeds by induction of the number of steps:
BASE CASE: $e\rightsquigarrow^{lwt^0}_{\epsilon}e'$
In this case $e\equiv e'$ and $\mathcal{A}'\equiv\emptyset$, so trivially $\mathcal{A}\vdash e':\tau$ and $wt_{\mathcal{A}}(\epsilon)$.
INDUCTIVE STEP: $e\rightsquigarrow^{lwt^{n+1}}_{\theta_1\theta'}e'\equiv e\rightsquigarrow^{lwt}_{\theta_1}e_1\rightsquigarrow^{lwt^n}_{\theta'}e'$
As $e\rightsquigarrow^{lwt^{n+1}}_{\theta_1\theta'}e'$, it is possible to check that there is a derivation $e\rightsquigarrow^{lwt^{n+1}}_{\theta_1\theta'}e'$ which uses type derivations $\mathcal{D}_i$ to τ in every inner step, so each set of assumptions $\mathcal{A}'_i$ associated to each step is related also to this derivation $\mathcal{D}_i$. By the proof of one step we have that $\mathcal{A}\oplus\mathcal{A}'_1\vdash e_1:\tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'_1}(\theta_1)$, where $\mathcal{A}'_1$ is the set of assumptions associated to the first step. Since the variables in $\mathcal{A}'_1$ cannot appear in $\mathcal{P}$, the program remains well-typed adding these new assumptions: $wt_{\mathcal{A}\oplus\mathcal{A}'_1}(\mathcal{P})$. Then by the Induction Hypothesis we have that $\mathcal{A}\oplus\mathcal{A}'_1\oplus\mathcal{A}'_n\vdash e':\tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'_1\oplus\mathcal{A}'_n}(\theta')$, where $\mathcal{A}'_n$ is the set of assumptions associated to the reduction $e_1\rightsquigarrow^{lwt^n}_{\theta'}e'$. The set $\mathcal{A}'\equiv\mathcal{A}'_1\oplus\mathcal{A}'_n$ contains assumptions over fresh variables. To prove $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta_1\theta')$ consider an arbitrary variable $X\in dom(\theta_1\theta')$:

- If $X\notin dom(\theta_1)$ then $X\theta_1\theta'\equiv X\theta'$ and $X\in dom(\theta')$. Trivially $\mathcal{A}\oplus\mathcal{A}'\vdash X\theta_1\theta':(\mathcal{A}\oplus\mathcal{A}')(X)$ from $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta')$.
- If $X\in dom(\theta_1)$ then by $wt_{\mathcal{A}\oplus\mathcal{A}'_1}(\theta_1)$ we have $\mathcal{A}\oplus\mathcal{A}'_1\vdash X\theta_1:(\mathcal{A}\oplus\mathcal{A}'_1)(X)$. Since the variables in $\mathcal{A}'_n$ are fresh they do not occur in $X\theta_1$, so by Theorem A.1-b) $\mathcal{A}\oplus\mathcal{A}'\vdash X\theta_1:(\mathcal{A}\oplus\mathcal{A}'_1)(X)$. Similarly, X cannot appear in $\mathcal{A}'_n$, so $(\mathcal{A}\oplus\mathcal{A}'_1)(X)\equiv(\mathcal{A}\oplus\mathcal{A}')(X)$. Finally, since $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta')$ by Theorem A.1-c) we obtain $\mathcal{A}\oplus\mathcal{A}'\vdash X\theta_1\theta':(\mathcal{A}\oplus\mathcal{A}')(X)$.

A.2 Proof of Lemma 3.10: Mgu well-typedness

The proof uses a transformation approach ($\Longrightarrow$) similar to that presented in [6] to obtain mgu's—which follows the same ideas as the one in [26]. The difference is that our transformation does not orient equations prior to apply (Eliminate): it eliminates variables regardless of the side, giving priority to left-hand sides. This is important, since to prove well-typedness of mgu's we need that left-hand sides of equations remain transparent. However, it is easy to see that this transformation behaves the same as the original in [6].

(Delete)	$\{p =^? p\} \uplus S$	$\Longrightarrow$	S
(Decompose)	$\{h\ \overline{p_n} =^? h\ \overline{t_n}\} \uplus S$	$\Longrightarrow$	$\{p_1 =^? t_1, \ldots, p_n =^? t_n\} \cup S$
(EliminateL)	$\{X =^? t\} \uplus S$	$\Longrightarrow$	$\{X =^? t\} \cup S[X \mapsto t]$, if $X \in fv(S) \setminus var(t)$
(EliminateR)	$\{p =^? X\} \uplus S$	$\Longrightarrow$	$\{p =^? X\} \cup S[X \mapsto p]$, if $X \in fv(S) \setminus var(p)$ and $p \notin \mathcal{V}$

The unification procedure $\mathcal{U}(p, t)$ starts with a set of one equation $\{p =^? t\}$ and performs $\Longrightarrow$ steps until it reaches normal form. If the normal form is in *solved form*—$\{\overline{X_n =^? t_n}\} \cup \{\overline{p_m =^? Y_m}\}$ where $p_i \notin \mathcal{V}$, $\{\overline{X_n}, \overline{Y_m}\}$ are pairwise distinct variables and $\{\overline{X_n}, \overline{Y_m}\} \cap (var(t_n) \cup var(p_m))$—the set represents the mgu $[\overline{X_n \mapsto t_n}, \overline{Y_m \mapsto p_m}]$, otherwise it fails.

In order to prove the well-typedness of mgu's obtained by $\mathcal{U}$ we need some extra results about the mentioned transition system. We use $\mathcal{U}$ to compute unifiers of left-hand sides of fresh variants of rules $f\ \overline{p_n}$ and expressions $f\ \overline{t_n}$. This particularity limits the sets of equations that we find along the computation of the mgu to *transparent sets*. To define transparent sets of equations we use the usual notion of *postions in expressions* $o \in \mathcal{O}$ [6], which are strings of positive integers using ϵ as the empty string. Then the *subexpression of e at position o*, denoted as $e|_o$, is defined as $e|_\epsilon = e$, $(h\ e_1\ \ldots\ e_n)|_{io} = e_i|_o$.

DEFINITION A.3 (Transparent set of equations). *We say a set of equations $S \equiv \{\overline{p_n =^? t_n}\}$ is* transparent *if every p_i is transparent and if there exists an equation $(p =^? t) \in S$ and position $o \in \mathcal{O}$ such that $p|_o \equiv X$ and $t|_o \equiv t'$ with t' a non-transparent pattern, then X appears only once in the set of equations—exactly in that position of that equation.*

LEMMA A.4 ($\Longrightarrow$ steps preserve set transparency). *If S is transparent and $S \Longrightarrow^* S'$, then S' is also transparet.*

Proof The proof for one step proceeds by case distinction on the rule used:

- (Delete) Trivially.
- (Decompose) The step is $S \equiv \{h\ \overline{p_n} =^? h\ \overline{t_n}\} \uplus S'' \Longrightarrow \{p_1 =^? t_1, \ldots, p_n =^? t_n\} \cup S'' \equiv S'$. Since $h\ \overline{p_n}$ is a transparent pattern wrt. $\mathcal{A}$, the new patterns $\overline{p_n}$ introduced as left-hand sides are transparent as well. By premise, if there is a equation $(p =^? t) \in S''$ and position $o \in \mathcal{O}$ such that $p|_o \equiv X$ and $t|_o \equiv t'$ with t' a non-transparent pattern, then X appears only once in S, so it appears only once in S' since variables in S and S' are the same. The reasoning is similarly if such a variable X appears in the equation $(h\ \overline{p_n} =^? h\ \overline{t_n})$ since that situation will happen in some equation $(p_i =^? t_i)$.
- (EliminateL) The step is $S \equiv \{X =^? t\} \uplus S'' \Longrightarrow \{X =^? t\} \cup S''[X \mapsto t] \equiv S'$, if $X \in fv(S'') \setminus var(t)$. If t is a non-transparent pattern, then X cannot appear in S'' by the transparency of S, so this rule cannot be applied. On the other hand, if t is transparent then applying the substitution $[X \mapsto t]$ to S'' keeps the left-hand sides transparent. If X appears in the left-hand side of a rule $(p'' =^? t'') \in S''$, we know that if there is a position $o \in \mathcal{O}$ such that $p''|_o \equiv X$ and $t''|_o \equiv t'$ then t' is transparent. Then for all the variables introduced in $p''[X \mapsto t]$ the pattern in the same position in $t''[X \mapsto t]$ will be transparent. If X appears in the right side of some equation, replacing it by t will not generate non-transparent patterns, so there cannot be any equation $(p'' =^? t'') \in S''[X \mapsto t]$ such that $p''|_o \equiv Y$ and $t''|_o \equiv t'$ for some $o \in \mathcal{O}$ and non-transparent pattern t' .
- (EliminateR) The step is $\{p =^? X\} \uplus S'' \Longrightarrow \{p =^? X\} \cup S''[X \mapsto p]$, if $X \in fv(S'') \setminus var(p)$ and $p \notin \mathcal{V}$. The reasoning is the same as the previous case, when t is a transparent pattern.

The proof for any number of steps proceeds trivially by induction on the number of steps.

LEMMA A.5 (Decomposition of patterns). *Let $h\ \overline{t_n}$ be a pattern and $h\ \overline{p_n}$ be a transparent pattern wrt. $\mathcal{A}$ such that $\mathcal{A} \vdash h\ \overline{t_n} : \tau$ and $\mathcal{A} \vdash h\ \overline{p_n} : \tau$. Then every pair of patterns t_i, p_i verify $\mathcal{A} \vdash t_i : \tau_i$ and $\mathcal{A} \vdash p_i : \tau_i$, for some τ_i.*

Proof Since $h\ \overline{p_n}$ is a transparent pattern, $\mathcal{A}(h)$ is n-transparent, so $\mathcal{A}(h) = \forall \overline{\alpha_m}.\overline{\tau'_n} \to \tau'$ such that $var(\overline{\tau'_n}) \subseteq var(\tau')$. Since both patterns have the same type τ, the generic instance $(\mathcal{A}(h) \succ \overline{\tau_n} \to \tau)$ used to derive a type for h in both patterns must be same, forcing the type τ_i of all the patterns to be the same because $var(\overline{\tau'_n}) \subseteq var(\tau')$.

LEMMA A.6 (Type preservation of $\Longrightarrow$ steps). *Let S be a transparent set of equations over patterns and $\mathcal{A}$ be a set of assumptions such that for every equation $(t_1 =^? t'_1) \in S$ it verifies $\mathcal{A} \vdash t_1 : \tau$ and $\mathcal{A} \vdash t'_1 : \tau$ for some τ. If $S \Longrightarrow^* S'$ then for every equation $(t_2 =^? t'_2) \in S$ it verifies $\mathcal{A} \vdash t_2 : \tau$ and $\mathcal{A} \vdash t'_2 : \tau$ for some τ.*

Proof The proof for one step proceeds by case distinction over the rule of the transition $\Longrightarrow$ applied. All the cases are straightforward with the exception of the (Decompose) case. Since S is transparent, we know that the left-hand side of the equation is transparent, so by Lemma A.5 the step preserves types.

The proof for any number of steps is straightforward using Lemma A.4, as set transparency is preserved.

Lemma 3.10 (Mgu well-typedness)
Let $\overline{p_n}$ be fresh linear transparent patterns wrt. $\mathcal{A}$ and let $\overline{t_n}$ be any patterns such that $\mathcal{A} \vdash p_i : \tau_i$ and $\mathcal{A} \vdash t_i : \tau_i$ for some type τ_i. If $\theta \equiv mgu(f\ \overline{p_n}, f\ \overline{t_n})$ then $wt_{\mathcal{A}}(\theta)$.

Proof Easily since $\mathcal{U}(f\ \overline{p_n}, f\ \overline{t_n})$ is the same as the mgu of the set $S \equiv \{p_1 =^? t_1, \ldots, p_n =^? t_n\}$. The set S is transparent—$\overline{p_n}$ are linear and transparent, and no variable in $\overline{p_n}$ appears in appears in $\overline{t_n}$ since they are fresh—so by Lemma A.6 the normal form S' verify that for every equation $(p'_i =^? t'_i)$ both sides have the same type, i.e., $\mathcal{A} \vdash p'_i : \tau_i$ and $\mathcal{A} \vdash t'_i : \tau_i$ for some τ_i. If S' is in solved form then S' has

the form $\{\overline{X_n =^? t''_n}\} \cup \{\overline{p''_m =^? Y_m}\}$ so the associated substitution $\theta \equiv [\overline{X_n \mapsto t''_i}, \overline{Y_m \mapsto p''_m}]$ (the obtained mgu) is well-typed because $\mathcal{A} \vdash t''_i : \mathcal{A}(X_i)$ (for $i \in [1..n]$) and $\mathcal{A} \vdash p''_j : \mathcal{A}(Y_j)$ (for $j \in [1..m]$).

A.3 Proof of Theorem 3.11 :Type preservation of $\rightsquigarrow^{lmgu}$

Theorem 3.11 (Type preservation of $\rightsquigarrow^{lmgu}$) *Let $\mathcal{P}$ be a program such that left-hand sides of rules contain only transparent patterns. If $wt_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash e : \tau$ and $e \rightsquigarrow_\theta^{lmgu^*} e'$ then $\mathcal{A} \oplus \mathcal{A}' \vdash e' : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

Proof Straightforward using Theorem 3.8, since under such conditions every $\rightsquigarrow^{lmgu}$-step is a $\rightsquigarrow^{lwt}$-step—trivially if the used rule is (LetIn)–(LetAp), or by Lemma 3.10 if (Narr) is used.

A.4 Proof of Lemma 4.1: Absence of HO variables

Lemma 4.1: (Absence of HO variables) *Let e be an expression such that $wt_{\mathcal{A}}(e)$ and for every $X_i \in fv(e)$, $\mathcal{A}(X_i)$ is not a functional type. Then no step $e \rightsquigarrow_\theta^l e'$ can use (VAct) or (VBind).*

Proof If (VAct) is applied then e must contain $X\ \overline{t_k}$, which can only be well-typed if X has a functional assumption in $\mathcal{A}$. On the other hand, if (VBind) is applied then e contains an expression $e' \notin Pat$ such that $e'\theta \in Pat$. It is easy to check that this expression e' must have the form $X\ \overline{t_k}$, so the reasoning is the same as in the previous case.

A.5 Proof of Lemma 4.6: Absence of unsafe variables

LEMMA A.7 (Decrease of free variables). *If $e \rightsquigarrow^l e'$ using the rules (LetIn), (Bind), (Elim), (Flat) or (LetAp) then $fv(e') \subseteq fv(e)$.*

Proof Straightforward.

LEMMA A.8 (Free variables of applied contexts). $fv(\mathcal{C}[e]) = fv(\mathcal{C}) \cup (fv(e) \setminus bv(\mathcal{C}))$

Proof Easily by induction on the structure of the context $\mathcal{C}$. The most interesting cases are those involving let-expressions:

- $\mathcal{C} \equiv let\ X = \mathcal{C}'\ in\ e'$.

$$\begin{array}{rcll} fv(\mathcal{C}[e]) \equiv & \underline{fv(let\ X = \mathcal{C}'[e]\ in\ e')} & & \text{Context application} \\ = & \overline{fv(\mathcal{C}'[e])} \cup (fv(e') \setminus \{X\}) & & \text{Definition of } fv \\ = & \overline{(fv(\mathcal{C}')} \cup (fv(e) \setminus bv(\mathcal{C}'))) \underline{\cup (fv(e') \setminus \{X\})} & & \text{Induction Hypothesis} \\ = & \overline{fv(\mathcal{C})} \cup (fv(e) \setminus \overline{bv(\mathcal{C}')}) & & \text{Definition of } fv(\mathcal{C}) \\ = & fv(\mathcal{C}) \cup (fv(e) \setminus \overline{bv(\mathcal{C})}) & & \text{Definition of } bv(\mathcal{C}) \end{array}$$

- $\mathcal{C} \equiv let\ X = e'\ in\ \mathcal{C}'$.

$$\begin{array}{rcll} fv(\mathcal{C}[e]) \equiv & \underline{fv(let\ X = e'\ in\ \mathcal{C}'[e])} & & \text{Context application} \\ = & \overline{fv(e') \cup (fv(\mathcal{C}'[e])} \setminus \{X\}) & & \text{Definition of } fv \\ = & fv(e') \cup (\overline{(fv(\mathcal{C}')} \cup (fv(e) \setminus bv(\mathcal{C}'))) \setminus \{X\}) & & \text{Induction Hypothesis} \\ = & \underline{fv(e') \cup (fv(\mathcal{C}') \setminus \{X\})} \cup (fv(e) \setminus (bv(\mathcal{C}') \cup \{X\}) & & \text{Set manipulation} \\ = & \overline{fv(\mathcal{C}) \cup (fv(e) \setminus (bv(\mathcal{C}') \cup \{X\})} & & \text{Definition of } fv(\mathcal{C}) \\ = & fv(\mathcal{C}) \cup (fv(e) \setminus \overline{bv(\mathcal{C})}) & & \text{Definition of } bv(\mathcal{C}) \end{array}$$

LEMMA A.9. *If $fv(e') \subseteq fv(e)$ then $fv(\mathcal{C}[e']) \subseteq fv(\mathcal{C}[e])$.*

Proof Straightforward using the characterization of free variables of an applied context in Lemma A.8.

LEMMA A.10. *Consider the expressions $f\ \overline{t_n}$ and $f\ \overline{p_n}$ and the set of variables $XS \subseteq fv(f\ \overline{t_n})$ such that every variable in $fv(f\ \overline{t_n}) \setminus XS$ is safe wrt. the same $\mathcal{A}$. Consider also a substitution θ such that $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$, $dom(\theta) \cap XS = \emptyset$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\theta)$, where $\mathcal{A}'$ is a set of assumptions over fresh variables used by θ. Then the following conditions hold:*

a) If $X \in fv(f\ \overline{t_n}) \setminus XS$ then $X\theta$ contain safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$.
b) If $X \in fv(f\ \overline{p_n})$ then every variable $Y \in fv(X\theta)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ or $Y \in XS$.

Proof

a) Let X be a variable in $fv(f\ \overline{t_n}) \setminus XS$ with safe type $\mathcal{A}(X) = \tau$. Since θ is well-typed wrt. $\mathcal{A} \oplus \mathcal{A}'$ and by hypothesis $\mathcal{A}'$ contains only assumptions over fresh variables, then $\mathcal{A} \oplus \mathcal{A}' \vdash X\theta : \tau$, where $\mathcal{A} \oplus \mathcal{A}'(X) = \tau$ remains a safe type wrt. $\mathcal{A} \oplus \mathcal{A}'$. $X\theta$ is a pattern of safe type, so by definition it can only contain safe variables.

b) By *a)* and $dom(\theta) \cap XS = \emptyset$ we know that $f\ \overline{t_n}\theta$ contains variables in XS or safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, and since $f\ \overline{t_n}\theta \equiv f\ \overline{p_n}\theta$ then $f\ \overline{p_n}\theta$ contains variables in XS or safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$ as well.

Lemma 4.6 (Absence of unsafe variables) *Let e be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \rightsquigarrow_\theta^{lwt^*} e'$ then e' does not contain unsafe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, where $\mathcal{A}'$ is a set of assumptions associated to the reduction.*

Proof We first proceed with the case of one step $e \rightsquigarrow_\theta^{lwt} e'$. The original expression e does not contain any free variable with unsafe type, so it cannot contain free HO variables and by Lemma 4.1 the step $e \rightsquigarrow_\theta^{lwt} e'$ do not use (VBind) or (VAct). Then we proceed by case distinction over the $\rightsquigarrow^{lwt}$ rule used:

- If the rule is (LetIn)–(LetAp) then $\theta \equiv \epsilon$ so $wt_{\mathcal{A}\oplus\mathcal{A}'}(\epsilon)$ for any $\mathcal{A}'$. By hypothesis, every $X \in fv(e)$ is safe wrt. $\mathcal{A}$, so by Lemma A.7 $fv(e') \subseteq fv(e)$ and every $X \in fv(e')$ is safe wrt. $\mathcal{A}$. As $\mathcal{A}'$ is the set of assumptions associated to the step and it contains assumptions over fresh variables, every $X \in fv(e')$ is also safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
- If the rule is (Narr) then $e \equiv f\ \overline{t_n} \leadsto_\theta^{lwt} r\theta$ for a fresh variant $(f\ \overline{p_n} \to r)$ and θ such that $f\ \overline{p_n}\theta \equiv f\ \overline{t_n}\theta$. By the hypothesis every variable $X \in fv(f\ \overline{t_n})$ is safe wrt. $\mathcal{A}$, so using Lemma A.10 with $XS = \emptyset$ we obtain that for each $X \in fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$ the pattern $X\theta$ cannot contain any unsafe variable wrt. $\mathcal{A} \oplus \mathcal{A}'$. According to $wt^r_{\mathcal{A}}(\mathcal{P})$ and the definition of the set of assumptions associated to the step, $\mathcal{A}'$ contains ground and safe types wrt. $\mathcal{A}$ for the extra variables in the rule—which are also safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. Any variable $X \in fv(r)$ can be in $fv(f\ \overline{p_n})$ or be an extra varaible. If $X \in fv(f\ \overline{p_n})$ we know that $X\theta$ cannot contain any unsafe variable wrt. $\mathcal{A} \oplus \mathcal{A}'$. On the other hand, if $X \notin fv(f\ \overline{p_n})$ it is an extra varaible, so it is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ and it is not changed by θ—we assume that $dom(\theta) \subseteq fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$. Therefore every variable in $fv(r\theta)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
- If the rule is (Contx) then $e \equiv \mathcal{C}[e] \leadsto_\theta^{lwt} \mathcal{C}\theta[e']$ for $\mathcal{C} \neq [\,]$, $e \leadsto_\theta^{lwt} e'$ using any rule different from (VAct) or (VBind) and verifying that *i)* $dom(\theta) \cap bv(\mathcal{C}) = \emptyset$ and *ii)* if the rule used is (Narr) with $(f\ \overline{p_n} \to r) \in \mathcal{P}$ then $vran(\theta|_{\setminus var(\overline{p_n})}) \cap bv(\mathcal{C}) = \emptyset$. We distinguish cases on the rule used in the step $e \leadsto_\theta^{lwt} e'$:
  - If the rule used is one of (LetIn)–(LetAp) then $\theta \equiv \epsilon$, so the final expression is $\mathcal{C}[e']$. By Lemma A.7 we know that $fv(e') \subseteq fv(e)$ so by Lemma A.9 $fv(\mathcal{C}[e']) \subseteq fv(\mathcal{C}[e])$. Since we have that every $X \in fv(\mathcal{C}[e])$ is safe wrt. $\mathcal{A}$ from the hypothesis, trivially every $Y \in fv(\mathcal{C}[e'])$ is also safe wrt. $\mathcal{A}$. Finally, as $\mathcal{A}'$ contains assumptions over fresh variables, $Y \in fv(\mathcal{C}[e'])$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
  - If the rule used is (Narr) then the step is $\mathcal{C}[f\ \overline{t_n}] \leadsto_\theta^{lwt} \mathcal{C}\theta[r\theta]$ using a fresh variant $(f\ \overline{p_n} \to r) \in \mathcal{P}$ and a unifier θ such that $wt_\theta(\mathcal{A} \oplus \mathcal{A}')$ being $\mathcal{A}'$ the set of assumptions associated to the step—containing ground and safe types for the extra variables of the rule. We assume that $dom(\theta) \subseteq fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$. If we define $XS = bv(\mathcal{C}) \cap fv(f\ \overline{t_n})$ then by Lemma A.10 we know *a)* for every variable $X \in fv(f\ \overline{t_n})$ the pattern $X\theta$ contains only safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$ and *b)* if $X \in fv(f\ \overline{p_n})$ then every $Y \in fv(X\theta)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ or $Y \in XS$. We want to prove that every $Y \in fv(\mathcal{C}\theta[r\theta])$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. By Lemma A.8 we have that $fv(\mathcal{C}\theta[r\theta]) = fv(\mathcal{C}\theta) \cup (fv(r\theta) \setminus bv(\mathcal{C}\theta))$:
    - $Y \in fv(\mathcal{C}\theta)$. We consider two cases: *1)* $Y \in fv(\mathcal{C})$ but $Y \notin dom(\theta)$. Then $Y \in fv(\mathcal{C}[e])$ (by Lemma A.8), so by hypothesis Y is safe wrt. $\mathcal{A}$, and trivially Y is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. *2)* $Y \in fv(Z\theta)$ for some $Z \in fv(\mathcal{C})$. Then $Z \in dom(\theta)$, and as $\overline{p_n}$ has fresh variables then $Z \in fv(f\ \overline{t_n})$. Moreover, since $Z \in fv(\mathcal{C})$ then $Y \notin XS$ because $XS \subseteq bv(\mathcal{C})$. Therefore by *a)* the pattern $X\theta$ contains only safe variables wrt. $\mathcal{A} \oplus \mathcal{A}'$, so Y is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$.
    - $Y \in fv(r\theta) \setminus bv(\mathcal{C}\theta)$. It is easy to see that $bv(\mathcal{C}) = bv(\mathcal{C}\theta)$ as substitutions does not change bound variables. Since $XS \subseteq bv(\mathcal{C})$ then $Y \notin XS$. Then by *b)* we know that if $Y \in fv(Z\theta)$ for some $Z \in fv(f\ \overline{p_n})$ then Y is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$. If $Y \in fv(Z\theta)$ for some $Z \notin fv(f\ \overline{p_n})$ then Z is an extra variable and it is not in $dom(\theta)$ because $dom(\theta) \subseteq fv(f\ \overline{t_n}) \cup fv(f\ \overline{p_n})$, so $Y \equiv Z$. Therefore Y is safe wrt. $\mathcal{A} \oplus \mathcal{A}'$ because $\mathcal{A}'$ contains $\mathcal{A}_f$, where Y has a safe type.

The proof for several steps proceeds by induction on the number n of steps:

BASE CASE: $e_1 \leadsto_\epsilon^{lwt^0} e_1$
Straightforward.

INDUCTIVE STEP: $e_1 \leadsto_\theta^{lwt^{n+1}} e_n \equiv e_1 \leadsto_{\theta_1}^{lwt} e_2 \leadsto_{\theta'}^{lwt^n} e_n$
By the proof of one step we have that $e_1 \leadsto_{\theta_1}^{lwt} e_2$ and every variable in $fv(e_2)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'_1$, where $\mathcal{A}'_1$ is the set of assumptions associated to the first step. Since $\mathcal{A} \oplus \mathcal{A}'_1$ is $\mathcal{A}$ extended with assumptions over variables, we also have that $wt^r_{\mathcal{A}\oplus\mathcal{A}'_1}(\mathcal{P})$. Therefore by the Induction Hypothesis we have that every variable in $fv(e_n)$ is safe wrt. $\mathcal{A} \oplus \mathcal{A}'_1 \oplus \mathcal{A}'$, where $\mathcal{A}'$ is the set of assumptions associated to the reduction $e_2 \leadsto_{\theta'}^{lwt^n} e_n$.

A.6 Proof of Theorem 4.7: Completeness of $\leadsto^{lmgu}$ wrt. $\leadsto^{lwt}$

Theorem 4.7 (Completeness of $\leadsto^{lmgu}$ wrt. $\leadsto^{lwt}$) *Let e be an expression not containing unsafe variables wrt. $\mathcal{A}$ and $\mathcal{P}$ be a program such that $wt^r_{\mathcal{A}}(\mathcal{P})$. If $e \leadsto_\theta^{lwt^*} e'$ using mgu's in each step then $e \leadsto_\theta^{lmgu^*} e'$.*

Proof By Lemma 4.6 we can assure that no expression involved in the reduction $e \leadsto_\theta^{lwt^*} e'$ will contain unsafe variables, so by Lemma 4.1 neither (VAct) nor (VBind) are used in the whole reduction. Since $e \leadsto_\theta^{lwt^*} e'$ uses mgu's, by definition $e \leadsto_\theta^{lmgu^*} e'$.

A.7 Proof of Theorem 5.2: $OIS(\mathcal{P})$ well-typedness

Theorem 5.2 ($OIS(\mathcal{P}_f)$ well-typedness) *Let $\mathcal{P}_f$ be a set of program rules for the same function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $OIS(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$.*

Proof It is easy to check that $wt_{\mathcal{A}\oplus\mathcal{A}'}(f\ \overline{t^i_n} \to e^i)$ for each $f_i \in \overline{f_m}$, since $wt_{\mathcal{A}}(f\ \overline{t^i_n} \to e^i)$ and $\mathcal{A}(f) = (\mathcal{A} \oplus \mathcal{A}')(f_i)$. The rule $f\ \overline{X_n} \to f_1\ \overline{X_n}?\ldots?f_m\ \overline{X_n}$ is also well-typed wrt. $\mathcal{A} \oplus \mathcal{A}'$: consider $\mathcal{A}'' \equiv \{\overline{X_n : \tau_n}\}$, where $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_n} \to \tau$. In this case, $\mathcal{A} \oplus \mathcal{A}' \oplus \mathcal{A}'' \vdash f_i\ \overline{X_n} : \tau$, therefore $\mathcal{A} \oplus \mathcal{A}' \oplus \mathcal{A}'' \vdash f_1\ \overline{X_n}?\ldots?f_m\ \overline{X_n} : \tau$. Since using the ($\Lambda$) rule it is possible to construct $\mathcal{A}''$, we have the type derivation $\mathcal{A} \oplus \mathcal{A}' \vdash \lambda\overline{X_n}.f_1\ \overline{X_n}?\ldots?f_m\ \overline{X_n} : \overline{\tau_n} \to \tau$. Finally, by Theorem A.1-a it is possible to derive a type $(\overline{\tau_n} \to \tau)[\alpha \mapsto \beta]$ with $\overline{\beta}$ fresh, which is a variant of $\forall\overline{\alpha}.\overline{\tau_n} \to \tau$.

A.8 Proof of Theorem 5.4: $\mathcal{U}(\mathcal{P})$ well-typedness

Theorem 5.4 ($\mathcal{U}(\mathcal{P}_f)$ well-typedness) *Let $\mathcal{P}_f$ be a set of program rules for the same overlapping inductive sequential function f such that $wt_{\mathcal{A}}(\mathcal{P}_f)$. If $\mathcal{U}(\mathcal{P}_f) = (\mathcal{P}', \mathcal{A}')$ then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P}')$..*

Proof *(Sketch)* We will see that the new rules $\mathcal{P}''$ added in each step are well-typed wrt. $\mathcal{A} \oplus \mathcal{A}''$, where $\mathcal{A}''$ are the assumptions added in the step. Consider the rule and assumption added for $\mathcal{P}_i$: $f\ \overline{X_j}\ (c_i\ \overline{Y_{k_i}})\ \overline{Z_l} \to f_{(c_i,o)}\ \overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l}$ and $\mathcal{A}''(f_{(c_i,o)}) = \forall\overline{\alpha}.\overline{\tau_j} \to \overline{\tau'_{k_i}} \to \overline{\tau_l} \to \tau$, where $\mathcal{A}(f) = \forall\overline{\alpha}.\overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$ and $\mathcal{A} \oplus \{\overline{Y_{k_i} : \tau'_{k_i}}\} \vdash c_i\ \overline{Y_{k_i}} : \tau'$ by the definition of $\mathcal{U}$ (Definition 5.3). It is clear that

$$\mathcal{A} \oplus \mathcal{A}'' \oplus \{\overline{X_j : \tau_j}, \overline{Y_{k_i} : \tau'_{k_i}}, \overline{Z_l : \tau_l}\} \vdash f_{(c_i,o)}\ \overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l} : \tau$$

and

$$\mathcal{A} \oplus \mathcal{A}'' \oplus \{\overline{X_j : \tau_j}, \overline{Y_{k_i} : \tau'_{k_i}}\} \vdash c_i\ \overline{Y_{k_i}} : \tau'$$

Therefore we can build the type derivation for the λ-abstraction

$$\mathcal{A} \oplus \mathcal{A}'' \vdash \lambda\overline{X_n}.\lambda c_i\ \overline{Y_{k_i}}.\lambda\overline{Z_l}.f_{(c_i,o)}\ \overline{X_j}\ \overline{Y_{k_i}}\ \overline{Z_l} : \overline{\tau_j} \to \tau' \to \overline{\tau_l} \to \tau$$

Finally, by Theorem A.1-a it is possible to derive any variant of $\mathcal{A}(f)$ for this λ-abstraction by using $[\overline{\alpha \mapsto \beta}]$ with $\overline{\beta}$ fresh, so the rule is well-typed. Notice that the recursive call of the transformation can introduce assumptions for new functions, but the previous derivation remains valid by by Theorem A.1-b, since these new functions cannot appear in the expression. Therefore, the rule is well-typed wrt. the final set of assumptions $\mathcal{A}'$ returned by the transformation.