Safe Typing of Functional Logic Programs with Opaque Patterns and Local Bindings[☆]

Francisco J. López-Fraguas[a,*], Enrique Martin-Martin[a], Juan Rodríguez-Hortalá[a]

[a] *Departamento de Sistemas Informáticos y Computación*
Facultad de Informática de la Univ. Complutense de Madrid
C/ Prof. José García Santesmases, s/n. 28040 Madrid, Spain

Abstract

Type systems are widely used in programming languages as a powerful tool providing safety to programs. 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 two variants of a Damas-Milner-like type system where certain uses of HO patterns (even opaque) are permitted while preserving type safety. The considered formal framework is that of programs without *extra* variables and using let-rewriting as reduction mechanism. The other aspect addressed 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.

Keywords: Functional-logic programming, Type systems, Opaque

[☆]This work has been partially supported by the Spanish projects TIN2008-06622-C03-01, S2009TIC-1465 and UCM-BSCH-GR35/10-A-910502.

[*]Corresponding author

Email addresses: fraguas@sip.ucm.es (Francisco J. López-Fraguas), emartinm@fdi.ucm.es (Enrique Martin-Martin), juanrh@fdi.ucm.es (Juan Rodríguez-Hortalá)

patterns, Let bindings

1. Introduction

Type systems for programming languages are an active area of research, no matter which paradigm is considered. In the case of functional programming, most type systems have arisen as extensions of Damas-Milner's [1], for its remarkable simplicity and good properties (decidability, existence of principal types, possibility of type inference, type safety results ...). Functional logic languages [2, 3, 4], in their practical side, have inherited almost directly Damas-Milner's types. In principle, most of the type extensions proposed for functional programming could be also incorporated to functional logic languages (e.g. this has been partially done for type classes [5, 6, 7]). However, if types are meant to be not only a decoration but are devised to provide safety to programs, then we must ensure 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 straightforward adaptations of Damas-Milner typing. One is the presence of so called *higher order (HO) patterns* in programs, an expressive feature allowed in some systems and for which a sensible semantics exists [8]; 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 and that is usually not well documented, not to say formalized.

The rest of the paper is organized as follows. The next two subsections make an introductory discussion to 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. *let-rewriting*, an operational reduction semantics for FL programs presented in [9]. Section 4 contains a type inference relation, which lets us find the most general type of expressions. Section 5 presents a method to infer types for programs. In Section 6 we examine some practical limitations of the type system and propose a variant to overcome them. In Section 7 we further discuss some other aspects of the two presented type systems. Finally, Section 8 contains some conclusions and points to future work.

$t \in Pat$	Set of patterns	Page 7
$e \in Exp$	Set of expressions	Page 7
$fv(e)$	Set of free variables	Page 7
$\mathcal{C}$	One-hole context	Page 7
$ftv(\sigma)$	Set of free type variables	Page 8
$\theta \in PSub$	Data substitution	Page 7
$\pi \in TSub$	type substitution	Page 9
$\mathcal{A}$	Set of assumptions	Page 9
$\oplus$	Union of set of assumptions	Page 9
$vran(\pi)$	Variable range of a substitution	Page 9
$\succ$	Generic instance relation	Page 9
$\succ_{var}$	Variant relation	Page 9
$\vdash$	Basic typing relation	Figure 5, page 10
$wt_{\mathcal{A}}(e)$	Well-typed expression wrt. $\vdash$	Page 11
$\vdash^{\bullet}$	Extended typing relation	Figure 7, page 12
	Opaque variable	Definition 1, page 12
$critVar_{\mathcal{A}}(e)$	Critical variables of e	Definition 2, page 13
$wt_{\mathcal{A}}(\mathcal{P})$	Well-typed program	Definition 3, page 14
$\Psi(e)$	Elimination of compound patterns	Figure 8, page 15
$\rightarrow^{l}$	Let-rewriting relation	Figure 9, page 16
$\Vdash$	Basic type inference relation	Figure 10, page 19
$\Vdash^{\bullet}$	Extended type inference relation	Figure 11, page 20
$\Pi^{\bullet}_{\mathcal{A},e}$	Typing substitution	Definition 4, page 21
$\mathcal{B}(\mathcal{A},\mathcal{P})$	Type inference of a program	Definition 5, page 22
$e^{\circ} \in Exp^{\circ}$	Simple expressions	Page 28
$\vdash^{\circ}$	Relaxed typing relation	Figure 13, page 28
$wt^{\circ}_{\mathcal{A}}(\mathcal{P})$	Well-typed relaxed program	Page 31

Figure 1: Summary of notation

1.1. Higher order patterns

In our setting patterns appear in the left-hand side of rules or let-bindings. Some of them can be HO patterns, if they contain partial applications of function or constructor symbols. The use of HO patterns has practical interest—see e.g. [8, 10, 11] for illustrating examples—and is natural in a setting having an intensional view of functions, where different descriptions of the same 'extensional' function can be observably distinguished[1]. This somehow non-typical behavior does not emerge by the allowance of HO patterns itself; as it is known [9], it stems from the mere combination of HO-functions, lazy evaluation and call-time choice semantics for non-determinism, a cocktail which is present in current FLP systems, whether or not they support HO-patterns. However, HO patterns can be a source of problems from the point of view of the types. In particular, it was shown in [13] that unrestricted use of HO patterns leads to loss of *type preservation*, 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 [14]). 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 direct adaptation of the classical Damas-Milner algorithm. We can extend the program with the functions $unpack\ (snd\ X) \to X$ and $cast\ X \to unpack\ (snd\ X)$, whose inferred types will be $\forall\alpha,\beta.(\alpha \to \alpha) \to \beta$ and $\forall\alpha,\beta.\alpha \to \beta$ respectively. Then 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 because 0 has not type $bool$, as is required by the context.

This loss of type preservation arises when dealing with HO patterns because, in some cases, knowing the type of a pattern of this class does not uniquely determine the type of its subpatterns. This can be easily viewed in the HO pattern $snd\ X$ of the function $unpack$ of the previous example: knowing its type (for example $\alpha \to \alpha$) does not fix the type of X (in fact

[1] By saying that e, e' represent the same 'extensional' function we mean that $e\ x$ and $e'\ x$ behave the same, for each argument x. We remark that some authors [12] have suggested the possibility of other notions of extensionality for which FLP languages would respect extensionality.

X could have any type in that case: *bool*, *[int]*, α, etc.). Notice that situation cannot happen in FO patterns like $[X]$, since knowing it has type $[\alpha]$ univocally forces X to have type α.

This problem of loss of type preservation was first faced in [13], where a rather drastic solution is proposed: simply forbid the appearance of any opaque pattern—a pattern which does not univocally fix the type of its subpatterns—in the left-hand side of any program rule. Nevertheless, as we will see through this work, it is possible to be less restrictive. The key idea 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 say that a term variable of a program rule $f\ \bar{t} \to e$ is *critical* if it is opaque in some pattern in $\bar{t}$ and also appears in the right-hand side e. A precise definition of opaque and critical variables will be given in Section 3. With these notions we can relax the situation in [13], prohibiting only those patterns having critical variables. Trying to cope with more cases that may appear in practice, we develop in Section 6 a variant of our type system, where the focus is moved from opaque data variables to *opacifying type variables*—type variables which make a data variable opaque. In this variant, we check that opacifying type variables—unknown types not fixed by the pattern—are not used in the right-hand side of rules.

We emphasize the fact that, concerning HO-patterns, the two type systems proposed in this paper are conceived to be conservative extensions of the usual Damas-Milner system, in the sense that for programs not making use of HO patterns, types are simply Damas-Milner's usual ones. This makes a big difference with another recent work [15] where we develop a type system that deliberately aims to go beyond Damas-Milner types by radically liberalizing the well-typedness conditions for FLP programs.

1.2. Local definitions

Functional and functional logic languages provide syntax to declare local definitions in the form of let-bindings within expressions. However, different implementations treat them differently regarding the polymorphism given to bound variables. This difference can be observed in the following example.

Example 2 (let-expressions). Consider the expressions $e_1 \equiv \mathit{let}\ F = \mathit{id}\ \mathit{in}\ (F\ \mathit{true}, F\ 0)$ and $e_2 \equiv \mathit{let}\ [F, G] = [\mathit{id}, \mathit{id}]\ \mathit{in}\ (F\ \mathit{true}, F\ 0, G\ 0, G\ \mathit{false})$. 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 2.3, Toy 2.3.2 or PAKCS 1.10.0 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 in a 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, SML of New Jersey v110.73, OCaml 3.12.1 or F# 2.0. In this case, we will say that lets are completely polymorphic, and write let_p.

Finally, we can treat bound variables monomorphically or polymorphically depending on the form of the pattern. If the pattern is a variable, it is treated polymorphically, but if it is compound all its variables are treated monomorphically. This is the case of GHC 7.2.2 and Curry Münster 0.9.11[2]. In these implementations e_1 is well-typed, while e_2 not. We call this kind of let-expression let_{pm}.

Figure 2 summarizes the default behavior of let-expressions wrt. types in various implementations of functional and functional logic languages. As can be seen, the grade of polymorphism is highly variable between systems, even inside the same language. One of the contributions of this paper is to technically clarify this matter by adopting a neutral position, formalizing the different possibilities for the polymorphism of local definitions and proving our results for the three possibilities.

2. Preliminaries

2.1. Expressions and programs

We assume a signature $\Sigma = DC \cup FS$, where DC and FS are two disjoint sets of *data constructor* and *function* symbols respectively, all of them with

[2]In fact, Curry Münster treats polymorphically bound variables only if the bound expressions are non-expansive—see [16] for more information. Similar restrictions apply to other systems, like the value restriction of ML ([17]) or the monomorphism restriction of Haskell ([18]), which prevents generalizing types for let-bounds variables if type class constraints occur in their types.

System	$\mathbf{let}_m$	$\mathbf{let}_{pm}$	$\mathbf{let}_p$
GHC 7.2.2		×	
Hugs Sept. 2006			×
SML of New Jersey v110.73			×
Ocaml 3.12.1			×
F# 2.0			×
Clean 2.3	×		
Toy 2.3.2*	×		
Curry PAKCS 1.10.0 (1)	×		
Curry Münster 0.9.11		×	

(*) we use `where` instead of `let`, not supported by Toy

Figure 2: **Let-expressions in different FP and FLP systems.**

an 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. The notation $\overline{o_n}$ stands for a sequence n objects $o_1 \ldots o_n$, where o_i is the i^{th} element in the sequence. When the number of elements does not play an important role, we write simply $\overline{o}$. Figure 3 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* $FOPat \ni fot ::= X \mid c\ \overline{fot_n}$ where $c \in DC^n$, and *higher order patterns* $HOPat = Pat \setminus FOPat$. We also make a distinction between different types of expressions: $X\ \overline{e_n}$ $(n > 0)$ is a *variable application*, $c\ \overline{e_n}$ ($c \in DC^m$ and $n > m$) is a *junk expression* and $f\ \overline{e_n}$ ($f \in FS^m$ and $n \geq m$) is an *active expression*. As a shortcut we write let_* for any kind of let-expression let_m, let_{pm} or let_p, and $\lambda\overline{t_n}.e$ for $\lambda t_1 \ldots \lambda t_n.e$. The set of *variables*—$var(e)$—and *free variables*—$fv(e)$—of an expression are defined in the usual way. Notice that free variables in let-bindings are defined as $fv(let_*\ t = e_1\ in\ e_2) = fv(e_1) \cup (fv(e_2) \setminus var(t))$, corresponding to the fact that we do not consider recursive let-bindings.

Figure 3 also shows the syntax of *programs*. A *program rule* is defined as $f\ \overline{t_n} \to e$ where $f \in FS^n$ and $\overline{t_n}$ is linear, i.e., every variable appears only once in all the patterns. Program rules must also fulfill that $fv(e) \subseteq \bigcup_{i=1}^{n} var(t_i)$. Therefore, *extra variables*—variables appearing only in the right-hand side of a rule—are not considered in this work. A *one-hole context* is defined as $\mathcal{C} ::= [\,] \mid \mathcal{C}\ e \mid e\ \mathcal{C} \mid \lambda t.\mathcal{C} \mid let_*\ t = \mathcal{C}\ in\ e \mid let_*\ t = e\ in\ \mathcal{C}$. A *data substitution* $\theta \in PSub$ is a finite mapping from data variables to patterns: $[\overline{X_n/t_n}]$.

Data variables		$X, Y \ldots$
Constructor symbol		c
Function symbol		f
Non variable symbol	$h ::=$	$c \mid f$
Symbol	$s ::=$	$X \mid c \mid f$
Pattern	$t ::=$	$X \mid c\ \overline{t_n}$ $(c \in DC^m$ and $n \leq m)$
		$\mid f\ \overline{t_n}$ $(f \in FS^m$ and $n < m)$
Expression	$e ::=$	$X \mid c \mid f \mid e_1\ e_2$
		$\mid \lambda t.e$ $(t$ linear$)$
		$\mid let_m\ t = e_1\ in\ e_2$ $(t$ linear$)$
		$\mid let_{pm}\ t = e_1\ in\ e_2$ $(t$ linear$)$
		$\mid let_p\ t = e_1\ in\ e_2$ $(t$ linear$)$
Program rule	$R ::=$	$f\ \overline{t_n} \to e$ $(f \in FS^n$ and $\bar{t}$ linear$)$
Program	$\mathcal{P} ::=$	$\{R_1, \ldots, R_n\}$

Figure 3: **Syntax of expressions and programs**

Type variables		$\alpha, \beta, \gamma, \ldots$
Simple type	$SType \ni \tau ::=$	$\alpha \mid \tau_1 \to \tau_2$
		$\mid C\ \overline{\tau_n}$ with $C \in \mathcal{TC}^n$
Type-scheme	$TScheme \ni \sigma ::=$	$\forall \overline{\alpha_n}.\tau$ $(n \geq 0)$

Figure 4: **Syntax of types**

Substitution application over data variables and expressions is defined in the usual way.

2.2. Types

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 syntax of *simple types* τ and *type-schemes* σ appears in Figure 4. As a shortcut for $\tau_1 \to \ldots \to \tau_n \to \tau$ we use $\overline{\tau_n} \to \tau$, or simply $\overline{\tau} \to \tau$ if the number of types is not relevant. 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

$\sigma \equiv \forall\overline{\alpha_m}.\overline{\tau_n} \to \tau$ is called *transparent* if $ftv(\overline{\tau_n}) \subseteq ftv(\tau)$, and *closed* if $ftv(\sigma) = \emptyset$.

A *set of assumptions* $\mathcal{A}$ is a set of the form $\{\overline{s_n : \sigma_n}\}$. 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[3]. If $(s_i : \sigma_i) \in \mathcal{A}$ we write $\mathcal{A}(s_i) = \sigma_i$. The set of free type variables for a set of assumptions is defined as $ftv(\{\overline{s_n : \sigma_n}\}) = \bigcup_{i=1}^{n} ftv(\sigma_i)$. The union of set of assumptions is denoted by $\oplus$ and it has the following 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 $\mathcal{A}'$.

A *type substitution* $\pi \in TSub$ is a finite mapping from type variables to simple types $[\overline{\alpha_n/\tau_n}]$. The *domain* and *variable range* of a type substitution π are defined as $dom(\pi) = \{\alpha \in \mathcal{TV} \mid \alpha\pi \neq \alpha\}$ and $vran(\pi) = \bigcup_{\alpha \in dom(\pi)} var(\alpha\pi)$. 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. τ' is a *generic instance* of $\sigma \equiv \forall\overline{\alpha_n}.\tau$ if $\tau' = \tau[\overline{\alpha_n/\tau_n}]$ for some $\overline{\tau_n}$, 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 that is a generic instance of σ' is also a generic instance of σ. Therefore $\forall\overline{\alpha_n}.\tau \succ \forall\overline{\beta_m}.\tau[\overline{\alpha_n/\tau_n}]$ iff $\{\overline{\beta_m}\} \cap ftv(\forall\overline{\alpha_n}.\tau) = \emptyset$— this alternative characterization is proved in [23]. Finally, τ' is a *variant* of $\sigma \equiv \forall\overline{\alpha_n}.\tau$ ($\sigma \succ_{var} \tau'$) if $\tau' = \tau[\overline{\alpha_n/\beta_n}]$ and $\overline{\beta_n}$ are fresh type variables.

3. Type derivation

We propose an extension of the Damas-Milner type system [1] where the task of giving a regular Damas-Milner type and the task of checking critical variables are kept separated. For that we define two different type relations: $\vdash$ and $\vdash^{\bullet}$.

The basic typing relation $\vdash$ in Figure 5 is similar to the classical Damas-Milner system but extended to handle the different kinds of let-expressions and the occurrence of patterns instead of variables in λ-abstractions and let-expressions. The rule [Λ] deals with λ-abstractions $\lambda t.e$ by guessing some types for the variables of the pattern t and using them to derive a type

[3]Non-transparent data constructors are allowed also in other extensions of Damas Milner system, like existential types [19, 20] or generalized algebraic data types [21, 22].

$$\textbf{[ID]}\quad \frac{}{\mathcal{A} \vdash s : \tau} \qquad \text{if } \mathcal{A}(s) \succ \tau$$

$$\textbf{[APP]}\quad \frac{\mathcal{A} \vdash e_1 : \tau_1 \to \tau \quad \mathcal{A} \vdash e_2 : \tau_1}{\mathcal{A} \vdash e_1\ e_2 : \tau}$$

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

$$\textbf{[LET}_m\textbf{]}\quad \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_m\ t = e_1\ \textbf{in}\ e_2 : \tau_2} \qquad \text{if } \{\overline{X_n}\} = var(t)$$

$$\textbf{[LET}^h_{pm}\textbf{]}\quad \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash h\ t_1 \ldots t_m : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_{pm}\ h\ t_1 \ldots t_m = e_1\ \textbf{in}\ e_2 : \tau_2} \qquad \begin{array}{l}\text{if } \{\overline{X_n}\} = var(h\ t_1 \ldots t_m) \\ \text{and } h \in DC \cup FS\end{array}$$

$$\textbf{[LET}^X_{pm}\textbf{]}\quad \frac{\mathcal{A} \vdash e_1 : \tau_1 \quad \mathcal{A} \oplus \{X : Gen(\tau_1, \mathcal{A})\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_{pm}\ X = e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$\textbf{[LET}_p\textbf{]}\quad \frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \vdash e_1 : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \vdash e_2 : \tau_2}{\mathcal{A} \vdash \textbf{let}_p\ t = e_1\ \textbf{in}\ e_2 : \tau_2} \qquad \text{if } \{\overline{X_n}\} = var(t)$$

Figure 5: **Rules of the type system** $\vdash$

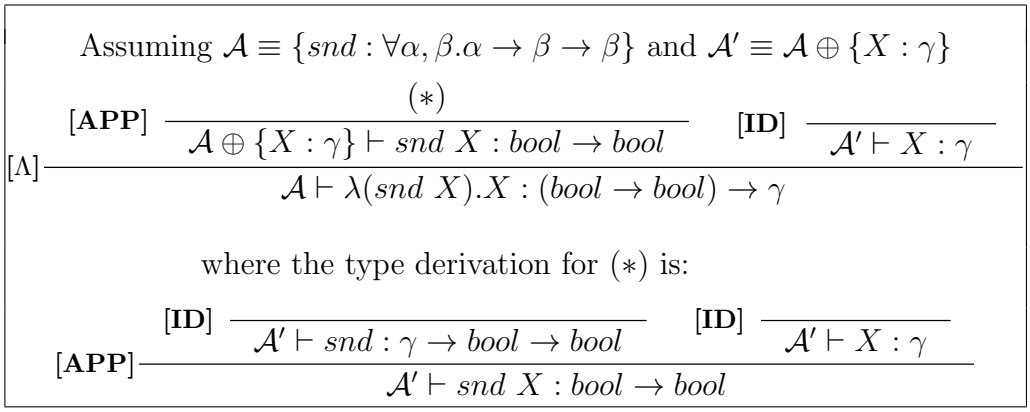

Figure 6: **Example of type derivation using** $\vdash$

for t and e. The rules [LET$_m$] and [LET$^h_{pm}$] are used to derive a type for monomorphic let-expressions. They guess some types for the variables in the pattern, and use them to derive a type for the pattern and the body of the let-expression—notice that both types derived in the binding must be the same. On the contrary, the rules [LET$^X_{pm}$] and [LET$_p$] derive types for polymorphic let-expressions. To obtain this behavior, they use the closure or generalization of a simple type τ wrt. a set of assumptions $\mathcal{A}$ [1, 23]—written $Gen(\tau, \mathcal{A})$—which generalizes all the type variables of τ that do not appear free in $\mathcal{A}$. Formally: $Gen(\tau, \mathcal{A}) = \forall\overline{\alpha_n}.\tau\ where\ \{\overline{\alpha_n}\} = ftv(\tau) \setminus ftv(\mathcal{A})$. The rules [LET$^X_{pm}$] and [LET$_p$] behave as [LET$_m$] and [LET$^h_{pm}$], with the difference that they use the generalization of the guessed types for the variables in the pattern of the binding—instead of the guessed types directly—in order to derive a type for the body of the let-expression. Notice that if two variables are assigned the same type in the binding of a let-expression, this connection can be lost after generalization. This fact can be seen with e_2 in Example 2 (page 5) : although we can assign F and G the type $\alpha \to \alpha$ (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.

As an example of the use of $\vdash$, Figure 6 shows a type derivation for the expression $\lambda(snd\ X).X$. We say that an expression e is well-typed wrt. $\vdash$ and $\mathcal{A}$—written $wt_{\mathcal{A}}(e)$—if $\mathcal{A} \vdash e : \tau$ for some type τ.

The $\vdash^\bullet$ relation in Figure 7 uses $\vdash$ and also enforces the absence of critical variables. To define the set of critical variables of an expression e—

$$[\mathbf{P}] \quad \frac{\mathcal{A} \vdash e : \tau}{\mathcal{A} \vdash^{\bullet} e : \tau} \quad \text{if } \mathit{critVar}_{\mathcal{A}}(e) = \emptyset$$

Figure 7: **Rule of the type system $\vdash^{\bullet}$**

$\mathit{critVar}_{\mathcal{A}}(e)$—we rely on the notion of opaque variable. As we have previously explained in Section 1, a variable is considered opaque in a pattern t if its type cannot be univocally known from the type of t. We use type derivations to formalize this idea: a variable X_i is *opaque* in a pattern 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 whole pattern. Intuitively, if such variables exist, then it is possible to use different instances of the type assumed for X_i (replacing only those variables) and derive the same type for t. Therefore, given a fixed type for t, there are different possible types for the variable X_i, so its type would not be fixed by the type of the pattern. The formal definition is as follows.

Definition 1 (Opaque variable of t wrt. $\mathcal{A}$). Consider a pattern t such that $wt_{\mathcal{A}}(t)$. We say that $X_i \in \{\overline{X_n}\} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\exists \overline{\tau_n}, \tau$ s.t. $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$ and $\mathit{ftv}(\tau_i) \not\subseteq \mathit{ftv}(\tau)$. If X_i is not opaque, we say it is a *transparent* variable of t wrt. $\mathcal{A}$.

Example 3 (Opaque variables of t wrt. $\mathcal{A}$).

- The variable X is opaque in the pattern $\mathit{snd}\ X$ wrt. any set of assumptions $\mathcal{A}_1$ containing the usual type-scheme for $\mathit{snd} : \forall\alpha, \beta.\alpha \to \beta \to \beta$ (see Example 1, page 4) and any type assumption for X. It is clear that $wt_{\mathcal{A}}(\mathit{snd}\ X)$—see Figure 6. However we can build the type derivation $\mathcal{A}_1 \oplus \{X : \gamma\} \vdash \mathit{snd}\ X : \mathit{bool} \to \mathit{bool}$ such that $\mathit{ftv}(\gamma) = \{\gamma\} \not\subseteq \emptyset = \mathit{ftv}(\mathit{bool} \to \mathit{bool})$.

- On the other hand, X is not opaque in the pattern $\mathit{snd}\ [X, \mathit{true}]$. It corresponds to the intuition, since in this case the list in the pattern univocally fixes the type of the variable X to *bool*. Consider a set of assumptions $\mathcal{A}_2$ containing the usual type-schemes for *snd* and the list constructors. Clearly $wt_{\mathcal{A}_2 \oplus \{X:\mathit{bool}\}}(\mathit{snd}\ [X, \mathit{true}])$. Moreover, $X : \mathit{bool}$ is the only assumption for X that can be added to $\mathcal{A}_2$ in order to derive

a type for $snd\ [X, true]$, otherwise the subpattern $[X, true]$ would be ill-typed. 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.

We write $opaqueVar_{\mathcal{A}}(t)$ for the set of opaque variables of t wrt. $\mathcal{A}$. Now, we can define the set of *critical variables* of an expression e wrt. $\mathcal{A}$ as those variables that, being opaque in the pattern of a let-binding or λ-abstraction of e, are indeed used in e. Formally:

Definition 2 (Critical variables).

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

Notice that if we write the function *unpack* of Example 1 (page 4) as $\lambda(snd\ X).X$, it is well-typed wrt. $\vdash$ using the usual type assumption 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) 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 $\pi \in TSub$.

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

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 detailed proof can be found in page 51 in Appendix A. □

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 any type derivation we can replace any 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 to 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. Type preservation

Type preservation 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. Type preservation is only guaranteed for *well-typed* programs, a notion that we formally define now.

Definition 3 (Well-typed program). A program rule $f\ t_1 \dots t_n \to e$ is well-typed wrt. $\mathcal{A}$ if $\mathcal{A} \vdash^\bullet \lambda t_1 \dots \lambda t_n.e : \tau$ 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 to achieve type preservation, as we will explain later. Returning to Example 1 (page 4) we can see that the program is not well-typed because of the rule $unpack\ (snd\ X) \to X$, since $\lambda(snd\ X).X$ is ill-typed wrt. the usual type for snd, as we explained before.

Although the condition that the type of the lambda abstraction associated to a rule must be a variant of the type of the function symbol (and not a generic instance) might seem strange, it is also necessary to guarantee type preservation. An example is given by the program $\mathcal{P} \equiv \{not'\ true \to false, not'\ false \to true\}$ with the assumptions $\mathcal{A} \equiv \{not' :: \forall\alpha.bool \to \alpha\}$. Clearly the type of both rules is $bool \to bool$, which is a generic instance, but not a variant of $\forall\alpha.bool \to \alpha$. If the program $\mathcal{P}$ was accepted as well-typed, type preservation would be violated, since we could reduce the well-typed

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

for $\{\overline{X_n}\} = var(t) \cap fv(e_2)$, $f_{X_i} \in FS^1$ fresh projecting function defined by the rule $f_{X_i}\ t \to X_i$, $Y \in \mathcal{DV}$ fresh, t a non variable pattern.

Figure 8: **Transformation rules of let-expressions with patterns**

expression $1 + (not'\ true)$—with type *int*—obtaining the ill-typed expression $1 + false$.

For type preservation to be meaningful, a precise notion of evaluation is needed. In this paper we consider and slightly adapt to our needs the *let-rewriting* relation of [9], a modification of term rewriting able to deal with HO syntax and let-bindings in a way that perfectly corresponds to the call-time choice semantics of FLP systems as formalized by the HO-CRWL semantic framework [8]. However, the original *let-rewriting* relation does not support let-expressions with compound patterns. Instead of extending the rules of let-rewriting with this feature we propose a transformation Ψ to expressions with only variables as patterns in let-bindings (Figure 8). There are various ways to perform this transformation, which differ in the strictness of the pattern matching. We have chosen the alternative explained in [24] 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 to consider the different kinds of let-expressions in order to preserve the types, as is stated in the following Theorem 2. Notice that the transformation only produces let_m and let_p bindings, which are precisely those accepted by the *let-rewriting* rules in Figure 9. Note also that let_{pm} bindings disappear, as for variable patterns they behave the same as let_p bindings, while for compound patterns they can be replaced by let_m bindings that still respect their monomorphic behavior. The following example shows the result of applying the transformation Ψ to an expression:

(Fapp) $f\ t_1\theta \ldots t_n\theta \ \rightarrow^l \ e\theta, \quad$ if $(f\ t_1 \ldots t_n \rightarrow e) \in \mathcal{P}$ and $\theta \in PSub$

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

Figure 9: **Higher order *let*-rewriting relation $\rightarrow^l$**

Example 4 (Transformation Ψ). Consider the expression

$$e \equiv let_{pm}\ [F, G] = [id, id]\ in\ (F\ true, G\ false)$$

In this case the result of the transformation $\Psi(e)$ is

$$let_m\ Y = [id, id]\ in\ let_m\ F = f_F\ Y\ in\ let_m\ G = f_G\ Y\ in\ (F\ true, G\ false)$$

where the projection functions f_F and f_G are defined as $f_F\ [F, G] \rightarrow F$ and $f_G\ [F, G] \rightarrow G$ respectively.

Theorem 2 (Type preservation of the transformation Ψ)**.** *Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_n}\ t_n \rightarrow X_n}\}$ be the rules of the projection functions needed in the transformation of e according to Figure 8. Let also $\mathcal{A}'$ be the set of assumptions over these functions, defined as $\mathcal{A}' \equiv \{\overline{f_{X_n} : Gen(\tau_{X_n}, \mathcal{A})}\}$, where τ_{X_i} is the most general type such that $\mathcal{A} \vdash^\bullet \lambda t_i.X_i : \tau_{X_i}$. Then $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet \Psi(e) : \tau$ and $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.*

Proof. By induction on the structure of e. A detailed proof can be found in page 54 in Appendix A. □

Theorem 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 $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

Regarding the rules for *let-rewriting* in Figure 9, they are as in the original version in [9], except for the fact that we have added polymorphism annotations to let-expressions and that the original (Flat) rule has been split into two, one for each kind of polymorphism. Although both rules behave the same from the point of view of values, the splitting is needed to guarantee type preservation by ensuring a proper handling of polymorphism annotations.

λ-abstractions are another feature that, being present in the type system, is not supported by the *let-rewriting* relation of [9] nor in the underlying HO-CRWL semantic framework [8]. In this case, since there is no general consensus about the semantic and operational meanings of λ-abstractions in the FLP setting, we have decided to keep things as they are, leaving λ-abstractions out of programs and expressions in the technical results that follow in the rest of the section.

Theorem 3 states the type preservation property for a *let-rewriting* step, but its extension to any number of steps is trivial.

Theorem 3 (Type Preservation)**.** *If* $\mathcal{A} \vdash^{\bullet} e : \tau$*,* $wt_{\mathcal{A}}(\mathcal{P})$ *and* $\mathcal{P} \vdash e \to^{l} e'$ *then* $\mathcal{A} \vdash^{\bullet} e' : \tau$*.*

Proof. By case distinction over the rule of the let-rewriting relation used to reduce e to e'. A detailed proof appears in page 57 in Appendix A. □

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 Example 1—page 4—where the program would be well-typed wrt. $\vdash$ but the type preservation property fails for *and* (*cast* 0) *true*.

The proof of the type preservation 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 replace its variables by other expressions, the only way to obtain the same type τ for the substituted pattern is by replacing the transparent variables

with expressions of the same type. This is not guaranteed with opaque variables, and that is why we forbid their use in expressions.

Lemma 1. *Assume* $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$*, where* $var(t) \subseteq \{\overline{X_n}\}$*. If* $\mathcal{A} \vdash t[\overline{X_n/t_n}] : \tau$ *and* X_j *is a transparent variable of* t *wrt.* $\mathcal{A}$ *then* $\mathcal{A} \vdash t_j : \tau_j$*.*

Proof. A detailed proof can be found in page 61 in Appendix A. □

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 some types. 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 give a relational style to type inference in Figures 10 and 11, to show the similarities with the typing relations $\vdash$ and $\vdash^\bullet$. But in essence, the inference rules represent an algorithm (similar to algorithm $\mathcal{W}$ [1, 23]) 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. Figure 12 contains an example of type inference for the expression $\lambda(snd\ X).X$.

Although $\Vdash$ is an effective inference procedure (it is based on unification) the type inference $\Vdash^\bullet$ uses the notion of critical variables. This notion relies on opaque variables (Definition 1), which have a declarative definition dependent on the existence of a certain type derivation that cannot be used as an effective procedure. However, based on the soundness and completeness of $\Vdash$ wrt. $\vdash$ (the following Theorems 4 and 5) it is possible to provide an effective procedure for checking opacity of variables:

Proposition 1. *Let* t *be a pattern such that* $wt_{\mathcal{A}}(t)$*.* $X_i \in \{\overline{X_n}\} = var(t)$ *is opaque wrt.* $\mathcal{A}$ *iff* $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ *and* $ftv(\alpha_i\pi_g) \not\subseteq ftv(\tau_g)$*.*

[iID] $$\frac{}{\mathcal{A} \Vdash s : \tau | id} \quad \text{if } \mathcal{A}(s) \succ_{var} \tau$$

[iAPP] $$\frac{\mathcal{A} \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_1 \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash e_1\ e_2 : \alpha\pi | \pi_1\pi_2\pi}$$
if α is a fresh type variable and $\pi = mgu(\tau_1\pi_2, \tau_2 \to \alpha)$

[iΛ] $$\frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t | \pi_t \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau | \pi}{\mathcal{A} \Vdash \lambda t.e : \tau_t\pi \to \tau | \pi_t\pi}$$
if $\{\overline{X_n}\} = var(t)$ and $\overline{\alpha_n}$ are fresh type variables

[iLET$_m$] $$\frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_m\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
if $\{\overline{X_n}\} = var(t)$, $\overline{\alpha_n}$ are fresh type variables and $\pi = mgu(\tau_t\pi_1, \tau_1)$

[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$^h_{pm}$] $$\frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash h\ t_1 \ldots t_m : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_{pm}\ h\ t_1 \ldots t_m = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
if $h \in DC \cup FS$, $\{\overline{X_n}\} = var(h\ t_1 \ldots t_m)$, $\overline{\alpha_n}$ are fresh type variables and $\pi = mgu(\tau_t\pi_1, \tau_1)$

[iLET$_p$] $$\frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t | \pi_t \quad \mathcal{A}\pi_t \Vdash e_1 : \tau_1 | \pi_1 \quad \mathcal{A}\pi_t\pi_1\pi \oplus \{\overline{X_n : Gen(\alpha_n\pi_t\pi_1\pi, \mathcal{A}\pi_t\pi_1\pi)}\} \Vdash e_2 : \tau_2 | \pi_2}{\mathcal{A} \Vdash \mathbf{let}_p\ t = e_1\ \mathbf{in}\ e_2 : \tau_2 | \pi_t\pi_1\pi\pi_2}$$
if $\{\overline{X_n}\} = var(t)$, $\overline{\alpha_n}$ are fresh type variables and $\pi = mgu(\tau_t\pi_1, \tau_1)$

Figure 10: **Inference rules for** $\Vdash$

$$[\mathbf{iP}]\frac{\mathcal{A} \Vdash e : \tau|\pi}{\mathcal{A} \Vdash^{\bullet} e : \tau|\pi} \quad \text{if } critVar_{\mathcal{A}\pi}(e) = \emptyset$$

Figure 11: **Inference rule for $\Vdash^{\bullet}$**

Assuming $\mathcal{A} \equiv \{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta\}$ and $\mathcal{A}' \equiv \mathcal{A} \oplus \{X : \gamma\}$

$$[\mathbf{i}\Lambda]\frac{[\mathbf{iAPP}]\ \dfrac{(*)}{\mathcal{A} \oplus \{X : \gamma\} \Vdash snd\ X : \epsilon \to \epsilon|\pi} \qquad [\mathbf{iID}]\ \dfrac{}{\mathcal{A}' \Vdash X : \gamma|id}}{\mathcal{A} \Vdash \lambda(snd\ X).X : (\epsilon \to \epsilon) \to \gamma|\pi}$$

where the type inference for $(*)$ is:

$$[\mathbf{iAPP}]\frac{[\mathbf{iID}]\ \dfrac{}{\mathcal{A}' \Vdash snd : \delta \to \epsilon \to \epsilon|id} \qquad [\mathbf{iID}]\ \dfrac{}{\mathcal{A}' \Vdash X : \gamma|id}}{\mathcal{A}' \Vdash snd\ X : \epsilon \to \epsilon|\pi}$$

where $\pi \equiv [\delta/\gamma, \zeta/\epsilon \to \epsilon]$ is the mgu of $\delta \to \epsilon \to \epsilon$ and $\gamma \to \zeta$
γ, δ, ϵ and ζ are fresh type variables

Figure 12: **Example of type inference using $\Vdash$**

The following results establish the close relationship between type inference and type derivation. Theorem 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 $\Vdash^{?}$)**.** $\mathcal{A} \Vdash^{?} e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash^{?} e : \tau$

Proof. By induction on the size of the type inference $\mathcal{A} \Vdash e : \tau|\pi$. A detailed proof can be found in page 62 in Appendix A.

□

Theorem 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* $\exists\tau, \pi, \pi''$ *such that* $\mathcal{A} \Vdash e : \tau|\pi$, $\mathcal{A}\pi\pi'' = \mathcal{A}\pi'$ *and* $\tau\pi'' = \tau'$.

Proof. By induction over the size of the type derivation. A detailed proof appears in page 63 in Appendix A. □

A result similar to Theorem 5 cannot be obtained for $\Vdash^\bullet$ because of critical variables, as the following example 5 shows.

Example 5 (Inexistence of most general typing substitutions). Let $\mathcal{A} \equiv \{snd' : \alpha \to bool \to bool\}$ and consider the following two valid derivations $\mathcal{A}[\alpha/bool] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to bool$ and $\mathcal{A}[\alpha/int] \vdash^\bullet \lambda(snd'\ X).X : (bool \to bool) \to int$. It is clear that there is no 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 is $[\alpha/\beta]$ (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). $\Pi^\bullet_{\mathcal{A},e} = \{\pi \in TSub \mid \exists \tau \in SType.\ \mathcal{A}\pi \vdash^\bullet e : \tau\}$

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* $\iff \exists \tau_g, \pi_g$ *such that* $\mathcal{A} \Vdash^\bullet e : \tau_g|\pi_g$.
b) *If* $\mathcal{A}\pi' \vdash^\bullet e : \tau'$ *and* $\mathcal{A} \Vdash^\bullet e : \tau|\pi$ *then a type substitution* π'' *exists such that* $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ *and* $\tau' = \tau\pi''$.

Proof. The complete proof can be found in page 65 in Appendix A. □

As a final comment about type inference, we remark that it can be used to effectively compute the types of the projecting functions in Theorem 2. This can be easily checked using Theorem 5 and the fact that the projecting functions only extract transparent variables of the patterns.

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 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* [25, 26]. 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 $\{R_1, \ldots, R_m\}$ is defined as:

$$\mathcal{B}(\mathcal{A}, \{R_1, \ldots, R_m\}) = \pi, \text{if}$$

1. $\mathcal{A} \Vdash^{\bullet} (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau_1, \ldots, \tau_m) | \pi$.
2. Let $f^1 \ldots f^k$ be the function symbols of the rules R_i in $\mathcal{P}$ such that $\mathcal{A}(f^i)$ is a closed type-scheme, and τ^i the type obtained for R_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:

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

using the special constructor *pair* for "tuples" of two elements of the same type, with type $\mathit{pair} : \forall\alpha.\alpha \to \alpha \to \alpha$.

Example 6 (Type Inference of Programs).

- Consider the program $\mathcal{P}$ with rules $\{ugly\ true \rightarrow true, ugly\ 0 \rightarrow true\}$ and the set of assumptions $\mathcal{A} \equiv \{ugly : \forall\alpha.\alpha \rightarrow bool\}$. 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

$$\mathcal{A} \Vdash^{\bullet} (pair\ (\lambda true.true)\ ugly, pair\ (\lambda 0.true)\ ugly) : (bool \rightarrow bool, int \rightarrow bool)|\pi$$

  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 $bool \rightarrow bool$ and $int \rightarrow bool$ inferred for its rules are variants of $\forall\alpha.\alpha \rightarrow bool$. This check will fail, therefore the procedure $\mathcal{B}$ will reject the program.

- Consider the program $\mathcal{P} \equiv \{and\ true\ X \rightarrow X, and\ false\ X \rightarrow false, id\ X \rightarrow X\}$ and the set of assumptions $\mathcal{A} \equiv \{and : \beta, id : \forall\alpha.\alpha \rightarrow \alpha\}$. 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:

$$\mathcal{A} \Vdash^{\bullet} (pair\ (\lambda true.\lambda X.X)\ and, pair\ (\lambda false.\lambda X.false)\ and, pair\ (\lambda X.X)\ id) : (bool \rightarrow bool \rightarrow bool, bool \rightarrow bool \rightarrow bool, \gamma \rightarrow \gamma)|\pi$$

  where $\pi \equiv [\beta/bool \rightarrow bool \rightarrow bool]$[4]. Therefore the type inferred for *and* would be the expected one: $bool \rightarrow bool \rightarrow bool$. Since *id* has a closed type-scheme in $\mathcal{A}$ then the second step will check the type inferred $\gamma \rightarrow \gamma$ is a variant of $\forall\alpha.\alpha \rightarrow \alpha$. The check is correct, therefore $\mathcal{B}$ succeeds with the substitution $[\beta/bool \rightarrow bool \rightarrow bool]$.

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

[4]Note that the bindings for type variables which are not free in $\mathcal{A}$ have been omitted here for the sake of conciseness.

to the assumptions $\mathcal{A}\pi$ (Theorem 7). And second, if the procedure $\mathcal{B}$ succeeds it finds the most general typing substitution (Theorem 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 Example 5.

Theorem 7 (Soundness of $\mathcal{B}$)**.** *If $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then $wt_{\mathcal{A}\pi}(\mathcal{P})$.*

Proof. The complete proof appears in page 68 in Appendix A. □

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. A detailed proof can be found in page 68 in Appendix A. □

Notice that types inferred for the functions are simple types. In order to obtain type-schemes we need an 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 7 (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 $\mathcal{P}$ 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. Tarjan's algorithm [27]. 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. A more relaxed typing relation

The type system of the previous sections accepts more programs as valid than [13], but still avoids the bad consequences that opaque patterns may have, as in the polymorphic casting in Example 1 (page 4). However, it rejects some rules with opaque patterns which do not cause any problem during evaluation, as we will see in the following examples. For simplicity, the opaque patterns in the examples use only *cont*, a constructor with the non-transparent type $cont : \forall\alpha.\alpha \to container$. Notice that *cont* is an example of an *existential data constructor* [19, 20], a notion closely related to opaque patterns but with different motivations and possibilities, as we will discuss in Section 7.1. The first example of harmless ill-typed rules is $f_1\ (cont\ X) \to length\ [X]$, using common syntax and functions for lists. The variable X is opaque in $cont\ X$, so the rule is ill-typed because X occurs in the right-hand side and becomes critical. The type of X is not fixed by the type of the pattern, however this is not a problem here because X is "used" in a context where any type is valid: $length\ [X]$. Therefore the evaluation of $length\ [X]$ will be correct for any type for X, returning an *int*.

A slight modification of the previous example is given by $f_2\ (cont\ (X : Xs)) \to length\ Xs$, which would be also rejected because Xs is critical. Although the type of $cont\ (X : Xs)$ does not fix completely the type of Xs, it fixes it *partially*: Xs must be a list of elements of *some* unknown type. Therefore $length\ Xs$ will not produce any pattern matching failure since *length* is polymorphic and accepts lists of *any* type. In other words, the partial knowledge of the type of Xs is enough to assure the absence of pattern matching errors during evaluation. As before, the resulting type will be *int* regardless of the type of the elements of the list.

A more involved example of this situation similar to f_2 comes from existential types [20]:

Example 8 (getKey function).

$\mathcal{A} \equiv \{mkKey : \forall\alpha.\alpha \to (\alpha \to int) \to key, getKey : key \to int\}$

$\mathcal{P} \equiv \{getKey\ (mkKey\ X\ F) \to F\ X\}$

The *mkKey* constructor contains an element of some *unknown* type and a function from *that* type to integer numbers. The *getKey* function accepts a key and returns the integer number resulting from applying the function to the element of unknown type. This rule would be ill-typed in our type system because both X and F are opaque variables—their types are α and $\alpha \to int$ resp. and the type of the pattern is *key*—and they appear in the right-hand side, so they are critical. However it is clear that this function will not produce any runtime error since the type of the *mkKkey* constructor assures that the function—second argument of *mkKey*—accepts data of the same type than the first element contained in the key.

A slightly different situation can be observed in the rule $f_3\ (cont\ X) \to cont\ X$. The variable X is opaque in $cont\ X$ so the rule is ill-typed since X appears in the right-hand side. However, f_3 does not break type preservation because we place that unknown element inside a new container which can store anything, so its type is shadowed again.

Using the same technique as in f_3, we could write a function for encoding booleans and lists into strings (assuming that ++ concatenates strings):

Example 9 (code function).

$$
\begin{array}{ll}
\mathcal{A} \equiv & \{cont : \forall\alpha.\alpha \to container, code : container \to string\} \\
\mathcal{P} \equiv & \{code\ (cont\ true) \to \text{“}T\text{”} \\
& \ code\ (cont\ false) \to \text{“}F\text{”}, \\
& \ code\ (cont\ [\,]) \to \text{”}[\,]\text{”}, \\
& \ code\ (cont\ (X : Xs)) \to \\
& \qquad code\ (cont\ X) \mathbin{++} \text{“}:\text{”} \mathbin{++} code\ (cont\ Xs)\}
\end{array}
$$

In this example, the *code* function uses the *cont* constructor to enclose the argument, so it can accept booleans and lists. The first three rules are well-typed, since they have not critical variables, but the fourth one is ill-typed because both X and Xs are critical. However, the *code* function does not produce any runtime error when encoding data involving lists and booleans. For example we can encode the list $[true]$, obtaining the reduction $code\ (cont\ (true : [\,])) \to^{l*} \text{“}T : [\,]\text{”}$. If we encode data different from lists or booleans—as $code\ (cont\ 0)$—its evaluation will lead to a pattern matching error since the *code* function is not defined for those data types. This can be explained considering that *code* is in some sense a partial function, similar to the well-typed *head* function which fails when applied to empty lists: $head\ [\,]$.

The previous examples motivate to find a new typing relation covering these functions while dealing safely with opaque patterns. The intuition that

emerges from the examples is that not all the occurrences of opaque variables in the right-hand side are problematic, only those that *use* parts of their type which cause opacity. For instance, in the last rule for *code* both X and Xs has a type variable α causing opacity—we say that α is an *opacifying* variable. However this type variable is not *used* in the right-hand side because it is not forced to be more specific, as X and Xs are applied to the *cont* constructor which accepts an element of any type. Similarly, in the *getKey* function both variables X and F are opaque because their types are α and $\alpha \to int$ resp., and α does not appear in the type of the pattern $mkKey\ X\ F$. However this type variable causing opacity is not *used* in the right-hand side, since the application $F\ X$ does not force it to be more specific. Moreover, the application $F\ X$ is well-typed because it uses only the types *partially* known from the pattern.

The intuitively well-behaved criterion shared by all the previous examples is that opacifying type variables must not be used in right-hand sides of function rules. Violating this criterion is risky for type preservation, as our next examples show. Consider the rule $g_1\ (cont\ X) \to not\ X$ with the assumption $g_1 : container \to bool$. The type of X in the pattern is α, and the type of the whole pattern is *container*, so α is an opacifying type variable. However, X is forced to have type *bool* in the right-hand side. If we allow this rule, we can make the reduction step $g_1\ (cont\ 0) \to^l not\ 0$. It is clear that the original expression is well-typed—with type *bool*—but the resulting expression *not* 0 is ill-typed, so types are not preserved. Apart from being more specific in the right-hand side, another *use* of opacifying type variables that must be avoided is when they occur in the resulting type of the rule, since this situation also compromises type preservation. This corresponds to the intuition of a function returning a result whose type is partially unknown. Consider a rule $g_2\ (cont\ X) \to X$ with the assumption $g_2 : \forall\alpha.container \to \alpha$. It is clear that $g_2\ (cont\ true)$ can have any type, in particular *int*. However if we reduce this expression we obtain *true*, which cannot have type *int* but only *bool*. Therefore, putting $g_2\ (cont\ true)$ in a context expecting an integer, like $1 + g_2\ (cont\ true)$, gives a well-typed expression, but reduction produces the ill-typed expression $1 + true$. Notice that it is always possible to find an example of loss of type preservation regardless of the returning type in the assumption for g_2: it will not preserve types with any type assumption $g_2 : \forall\overline{\alpha_n}.container \to \tau$.

Notice that the explained restrictions about opacifying type variables—not to be more specific in the right-hand sides of rules nor occur in the

$$\textbf{(ID}^\circ\textbf{)} \quad \frac{}{\mathcal{A} \vdash^\circ s : \tau} \qquad \text{if } \mathcal{A}(s) \succ \tau$$

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

$$(\Lambda^\circ)\ \frac{\mathcal{A} \oplus \{\overline{X_n : \alpha_n \pi_g \pi}\} \vdash^\circ t : \tau_g \pi \quad \mathcal{A} \oplus \{\overline{X_n : \alpha_n \pi_g \pi}\} \vdash^\circ e^\circ : \tau_e}{\mathcal{A} \vdash^\circ \lambda t.e^\circ : \tau_g \pi \to \tau_e} \quad \text{if } \begin{cases} \mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g | \pi_g \\ dom(\pi) \subseteq ftv(\tau_g) \\ ftv(\tau_e) \cap \mathit{OpacifyingVars} = \emptyset \\ vran(\pi) \cap \mathit{OpacifyingVars} = \emptyset \end{cases}$$

$$\text{where } \mathit{OpacifyingVars} = \textstyle\bigcup_{i=1}^{n} ftv(\alpha_i \pi_g) \setminus ftv(\tau_g) \text{ and } \{\overline{X_n}\} = var(t)$$

$$\textbf{(LET}^\circ_m\textbf{)}\ \frac{\mathcal{A} \vdash^\circ e_1^\circ : \tau_1 \quad \mathcal{A} \oplus \{X : \tau_1\} \vdash^\circ e_2^\circ : \tau_2}{\mathcal{A} \vdash^\circ let_m\ X = e_1^\circ\ in\ e_2^\circ : \tau_2}$$

Figure 13: **Rules of the relaxed type system**

resulting type of the rule—are very similar to the restrictions of existential types [19, 20]—*Skolem constants* cannot be instantiated nor escape from the scope of a pattern match. However, due to its motivation in *abstract data types*, existential types would reject some of the harmless examples in this section, whereas the type system in the next section accepts all them as valid. Moreover, a possible adaptation of existential types to our FLP setting would have to deal with some problematic subtleties about HO patterns. To explain these points, a detailed comparison of existential types and the type systems in this paper can be found in Section 7.1.

6.1. Formalization of the relaxed type system

In this section we define a new type relation $\vdash^\circ$ that formalizes the intuitions previously presented. Since we are basically interested in the behavior of opacifying type variables during function application, for the sake of conciseness we will focus on a subset of expressions that drops all the variety of let-expressions. Therefore the new type system will only deal with $Exp^\circ \ni e^\circ ::= X \mid c \mid f \mid e_1^\circ\ e_2^\circ \mid let_m\ X = e_1^\circ\ in\ e_2^\circ \mid \lambda t.e^\circ$. It is clear that $Pat \subseteq Exp^\circ$ and $Exp^\circ \subseteq Exp$.

Figure 13 shows the rules of the typing relation $\vdash^{\circ}$. This typing relation is a relaxation of $\vdash^{\bullet}$: it allows the occurrence of opaque data variables in right-hand sides of rules provided they use only the known part of the type of those variables. To keep type safety, it is necessary to guarantee the constraints over opacifying type variables explained in the previous section. Regarding the rules (ID$^{\circ}$), (APP$^{\circ}$) and (LET$^{\circ}_{m}$), they are the same as the corresponding rules for the $\vdash$ typing relation—see Figure 5 in page 10. However, the rule dealing with λ-abstractions is responsible for avoiding the problematic uses of opacifying type variables, so it has several conditions. Unlike the type relation $\vdash^{\bullet}$, where the type derivation for the λ-abstraction ($\vdash$) is kept separated from the check for critical variables, the (Λ°) rule performs the derivation and the check at the same time. The main steps for deriving a type for a λ-abstraction remains the same: choose some type assumptions for the variables in the pattern and use them to derive a type for the pattern and the body. However, in order to avoid incorrect uses of opacifying variables, not all type assumptions are valid now. The first step is obtain the most general type assumptions for the data variables in the pattern[5]—$\{\overline{X_n : \alpha_n \pi_g}\}$—and use them to compute the set of opacifying type variables—$\bigcup_{i=1}^{n} \mathit{ftv}(\alpha_i \pi_g) \setminus \mathit{ftv}(\tau_g)$. Then any instance—$\{\overline{X_n : \alpha_n \pi_g \pi}\}$—of those most general assumptions are valid provided the opacifying variables are not changed. This is achieved by restricting $\mathit{dom}(\pi) \subseteq \mathit{ftv}(\tau_g)$. The reason for not to change opacifying variables is to detect cases where the right-hand side forces them to be more specific. In these situations, the derivation of the right-hand side will fail since they appear as type variables in the assumptions. This checks the first incorrect use of opacifying type variables. The condition $\mathit{ftv}(\tau_e) \cap \mathit{OpacifyingVars} = \emptyset$ assures that no opacifying type variable appears in the resulting type of the λ-abstraction, the second incorrect use of opacifying variables. Finally, the technical condition $\mathit{vran}(\pi) \cap \mathit{OpacifyingVars} = \emptyset$ is set to ensure that substitution π does not introduce opacifying variables, thus avoiding that harmless type variables were replaced by types containing opacifying variables, therefore confusing

[5]Notice that we use $\Vdash$ to compute those most general type assumptions for the data variables, based on the completeness of $\Vdash$ (Theorem 5, page 20). We could formalize this rule with a more declarative flavor by using the most general types $\overline{\tau_n}$, τ_g such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_g$, where $\overline{\tau_n}$ would take the place of $\overline{\alpha_n \pi_g}$. However, we have preferred the use of $\Vdash$ to keep the rule concise, as its meaning concerning most general types is clear.

opacifying variables and the rest.

Using the $\vdash^\circ$ typing relation, the λ-abstractions associated to *getKey* and *code* (all rejected with $\vdash^\bullet$) are now well-typed. The following example shows some of them:

Example 10 (Well-typed expressions wrt. $\vdash^\circ$).

A) Let $\mathcal{A}$ be the set of assumptions defined as $\mathcal{A} \equiv \{mkKey : \forall\alpha.\alpha \to (\alpha \to int) \to key\}$. Then the λ-abstraction associated to the rule $getKey\ (mkKey\ X\ F) \to F\ X$ has type $key \to int$:

$$(\Lambda^\circ)\frac{\mathcal{A} \oplus \{X : \alpha, F : \alpha \to int\} \vdash^\circ mkKey\ (X\ F) : key \quad \mathcal{A} \oplus \{X : \alpha, F : \alpha \to int\} \vdash^\circ F\ X : int}{\mathcal{A} \vdash^\circ \lambda(mkKey\ (X\ F)).F\ X : key \to int}$$

With $\pi \equiv id$ all the conditions of the (Λ°) rule hold:

1. $\mathcal{A} \oplus \{X : \alpha, F : \beta\} \Vdash mkKey\ (X\ F) : key|[\beta/\alpha \to int]$, therefore $OpacifyingVars = \{\alpha\} \setminus ftv(key) = \{\alpha\}$
2. $dom(\pi) = \emptyset \subseteq ftv(key)$
3. $ftv(int) \cap OpacifyingVars = \emptyset$
4. $vran(\pi) \cap OpacifyingVars = \emptyset$

B) If $\mathcal{A}$ contains the usual assumptions for list constructors/functions, the *cont* constructor and the *code* function, then the λ-abstraction associated to the rule $code\ (cont\ (X : Xs)) \to e^\circ$—where $e^\circ \equiv code\ (cont\ X)\ ++\ ":"\ ++\ code\ (cont\ Xs)$—has type $container \to string$:

$$(\Lambda^\circ)\frac{\mathcal{A} \oplus \{X : \alpha, Xs : [\alpha]\} \vdash^\circ cont\ (X : Xs) : container \quad \mathcal{A} \oplus \{X : \alpha, Xs : [\alpha]\} \vdash^\circ e^\circ : string}{\mathcal{A} \vdash^\circ \lambda(cont\ (X : Xs)).e^\circ : container \to string}$$

All the conditions of the (Λ°) rule hold taking $\pi \equiv id$:

1. $\mathcal{A} \oplus \{X : \alpha, Xs : \beta\} \Vdash cont\ (X : Xs) : container|[\beta/[\alpha]]$, therefore $OpacifyingVars = \{\alpha\} \setminus ftv(container) = \{\alpha\}$
2. $dom(\pi) = \emptyset \subseteq ftv(container)$
3. $ftv(string) \cap OpacifyingVars = \emptyset$
4. $vran(\pi) \cap OpacifyingVars = \emptyset$

The interested reader may easily check that the functions f_1–f_3 previously defined are also well-typed wrt. $\vdash^\circ$.

The following example shows some ill-typed expressions wrt. $\vdash^\circ$. In particular it shows the *unpack* function of Example 1 in page 4—causing the polymorphic casting—and also the previously presented functions g_1 and g_2—which break the type preservation property.

Example 11 (Ill-typed expressions wrt. $\vdash^\circ$).

A) Consider $\mathcal{A}_1 \equiv \{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta\}$. Then the λ-abstraction $\lambda(snd\ X).X$—associated to the *unpack* rule—is ill-typed. In this case the type inference for the pattern is $\mathcal{A} \oplus \{X : \alpha\} \Vdash snd\ X : \beta \to \beta | id$, so $Opacifying Vars = \{\alpha\}$ and by the second condition of the (Λ°) rule, $dom(\pi) \subseteq \{\beta\}$. Since π will not substitute α, then the type derivation for the body of the λ-abstraction will be $\mathcal{A} \oplus \{X : \alpha\} \vdash^\circ X : \alpha$. Therefore the third condition of the (Λ°) rule will be violated, since $ftv(\alpha) \cap Opacifying Vars = \{\alpha\} \cap \{\alpha\} \neq \emptyset$.

B) With $\mathcal{A}_2 \equiv \{cont : \forall\alpha.\alpha \to container, not : bool \to bool\}$ the λ-abstraction $\lambda(cont\ X).not\ X$ associated to the rule $g_1\ (cont\ X) \to not\ X$ is ill-typed. The reason is that the type inference in the first condition is $\mathcal{A} \oplus \{X : \alpha\} \Vdash cont\ X : container | id$, so $Opacifying Vars = \{\alpha\}$ and by the second condition $dom(\pi) \subseteq ftv(container) = \emptyset$. Therefore the substitution π can only be id. With the assumptions $\mathcal{A} \oplus \{X : \alpha\}$ is not possible to derive a type for $not\ X$, since it needs X to have type $bool$, so $\lambda(cont\ X).not\ X$ is ill-typed.

C) With $\mathcal{A}_2$, the λ-abstraction $\lambda(cont\ X).X$ associated to $g_2\ (cont\ X) \to X$ is also ill-typed. The reason is very similar to the case of *unpack* in example A).

6.2. Properties of the relaxed type system

The new type relation $\vdash^\circ$ has been devised as to accept some interesting programs that $\vdash^\bullet$ rejects, but keeping the type preservation property: in well-typed programs, reduction steps preserve types. We now prove that, obtaining for $\vdash^\circ$ a result similar to Theorem 3 in page 17. For the notion of well-typed program—written $wt^\circ_{\mathcal{A}}(\mathcal{P})$—we simply replace $\vdash^\bullet$ by $\vdash^\circ$ in Definition 3 in page 14. Similarly, reductions are done with the let-rewriting

relation of Figure 9 (page 16), but simplified taking into account that only monomorphic let-expressions are considered here. Finally, as was already done for Theorem 3, we assume that neither the program $\mathcal{P}$ nor the expression to be evaluated contain λ-abstractions.

Theorem 9 (Type Preservation)**.** *If* $wt^{\circ}_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^{\circ} e^{\circ}_1 : \tau$ *and* $e^{\circ}_1 \rightarrow^{l} e^{\circ}_2$, *then* $\mathcal{A} \vdash^{\circ} e^{\circ}_2 : \tau$.

Proof. By case distinction over the applied let-rewriting rule. The complete proof can be found in page 68 in Appendix A. □

The typing relation $\vdash^{\circ}$ suffers from a problem also present in $\vdash^{\bullet}$: there are expressions that do not have a most general typing substitution. The Example 5 for $\vdash^{\bullet}$ (page 21) is also valid for $\vdash^{\circ}$:

Example 12 (Inexistence of a most general typing substitution).
Consider $\mathcal{A} \equiv \{snd' : \alpha \rightarrow bool \rightarrow bool\}$ and the expression $e^{\circ} \equiv \lambda(snd'\ X).X$. Applying the substitution $\pi_1 \equiv [\alpha/bool]$ to $\mathcal{A}$ it is possible to derive the type $(bool \rightarrow bool) \rightarrow bool$ for e°:

$$(\Lambda^{\circ})\frac{\begin{array}{c}\mathcal{A}\pi_1 \oplus \{X : bool\} \vdash^{\circ} snd'\ X : bool \rightarrow bool \\ \mathcal{A}\pi_1 \oplus \{X : bool\} \vdash^{\circ} X : bool\end{array}}{\mathcal{A}\pi_1 \vdash^{\circ} \lambda(snd'\ X).X : (bool \rightarrow bool) \rightarrow bool}$$

The conditions of the (Λ°) rule hold with $\pi \equiv id$:

1. $\mathcal{A}\pi_1 \oplus \{X : \beta\} \Vdash snd'\ X : bool \rightarrow bool | [\beta/bool]$, so $OpacifyingVars = \emptyset$
2. $dom(\pi) = \emptyset \subseteq ftv(bool \rightarrow bool)$
3. $ftv(bool) \cap OpacifyingVars = \emptyset$
4. $vran(\pi) \cap OpacifyingVars = \emptyset$

For similar reasons, it is possible to derive a type for e° applying $\pi_2 \equiv [\alpha/int]$ to $\mathcal{A}$: $\mathcal{A}\pi_2 \vdash^{\circ} \lambda(snd'\ X).X : (bool \rightarrow bool) \rightarrow int$. However there is not a substitution more general than π_1 and π_2 which makes $\lambda(snd'\ X).X$ well-typed. The only substitution more general than π_1 and π_2 is $\pi_g \equiv [\alpha/\gamma]$ (for some γ), but applying this substitution to $\mathcal{A}$ the inference of the first condition is: $\mathcal{A}\pi_g \oplus \{X : \beta\} \Vdash snd'\ X : bool \rightarrow bool | [\beta/\gamma]$. Then $OpacifyingVars = \{\gamma\}$ and by the second condition—$dom(\pi) \subseteq ftv(bool \rightarrow bool) = \emptyset$—$\pi$ must be id. Therefore the derivation of the body of the λ-abstraction will be $\mathcal{A}\pi_g \oplus \{X : \gamma\} \vdash^{\circ} X : \gamma$, which violates the third condition: $ftv(\gamma) \cap OpacifyingVars = \{\gamma\} \neq \emptyset$.

This paper does not address a type inference procedure $\Vdash^{\circ}$ for the relaxed type system. We discern two complications regarding this type inference procedure. First, as Example 12 shows, any type inference procedure should fail for those expressions not having a most general typing substitution, or compute a substitution which is not most general. Although the lack of principal types also occurs in $\vdash^{\bullet}$, it introduces more complications in $\vdash^{\circ}$ because the check for opacifying variables is interleaved into the inference process—in contrast to $\Vdash^{\bullet}$, where the inference of types ($\Vdash$) is keep separated from the check of critical variables. The other complication is the lack of closure under arbitrary substitutions of $\vdash^{\circ}$ in λ-abstractions:

Example 13 ($\vdash^{\circ}$ is not closed under type substitutions). Consider a set of assumptions $\mathcal{A} \equiv \{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta, f : \beta_1 \to \beta_1, id : \forall\alpha.\alpha \to \alpha\}$ and the expression $e^{\circ} \equiv \lambda(snd\ (f\ X)).id$. The type $(\beta' \to \beta') \to \beta'' \to \beta''$ is valid for e°:

$$(\Lambda^{\circ})\frac{\begin{array}{c}\mathcal{A} \oplus \{X : \beta_1\} \vdash^{\circ} snd\ (f\ X) : \beta' \to \beta' \\ \mathcal{A} \oplus \{X : \beta_1\} \vdash^{\circ} id : \beta'' \to \beta''\end{array}}{\mathcal{A} \vdash^{\circ} \lambda(snd\ (f\ X)).id : (\beta' \to \beta') \to \beta'' \to \beta''}$$

where $\mathcal{A} \oplus \{X : \gamma_1\} \Vdash snd\ (f\ X) : \beta' \to \beta' | [\gamma_1 \mapsto \beta_1]$ and we took $\pi \equiv id$. It is clear that the conditions about the opacifying variables ($\{\gamma_1\}$) hold, since $vran(id) = \emptyset$ and $\{\gamma_1\} \cap ftv(\beta'' \to \beta'') = \emptyset$. However, taking the type substitution $\pi_1 \equiv [\beta_1 \mapsto \delta, \beta'' \mapsto \delta]$ we have that $\mathcal{A}\pi_1 \not\vdash^{\circ} \lambda(snd\ (f\ X)).id : (\beta' \to \beta') \to \delta \to \delta$. Since $\mathcal{A}\pi_1 \oplus \{X : \gamma_2\} \Vdash snd\ (f\ X) : \beta'_2 \to \beta'_2 | [\gamma_2 \mapsto \delta] \equiv \pi_2$, any possible derivation would have the shape:

$$(\Lambda^{\circ})\frac{\begin{array}{c}\mathcal{A}\pi_1 \oplus \{X : \gamma_2\pi_2\pi\} \vdash^{\circ} snd\ (f\ X) : \beta' \to \beta' \\ \mathcal{A}\pi_1 \oplus \{X : \gamma_2\pi_2\pi\} \vdash^{\circ} id : \delta \to \delta\end{array}}{\mathcal{A}\pi_1 \vdash^{\circ} \lambda(snd\ (f\ X)).id : (\beta' \to \beta') \to \delta \to \delta}$$

The set opacifying variables is $\{\delta\} \setminus \{\beta'_2\} = \{\delta\}$, which violates the third condition in the (Λ°) rule: $ftv(\delta \to \delta) \cap \{\delta\} \neq \emptyset$.

As the closure property is essential in the proof of soundness of classical type inference procedures—e.g. algorithm $\mathcal{W}$ [1], algorithm $\mathcal{T}$ [23], $\Vdash$ and $\Vdash^{\bullet}$ in this paper, or even $\mathcal{W}_{\exists}$ for existential types [20]—we strongly believe that a type inference approach for $\vdash^{\circ}$ using unification following the ideas of these algorithms will not be sound. In contrast to type inference,

we are convinced that type checking of programs is possible provided the types of all symbols are given. We base our confidence in that problematic situations—see Examples 12 and 13—only appear when the assumptions for function symbols are not closed, as it is usual during type inference. However, if these types are closed and given by the programmer, the task of type checking a program becomes easier. The design of a sound (and possibly maximal) type inference procedure—following approaches different from $\mathcal{W}$—and a type checking mechanism for programs are considered as very interesting topics of future work, since they are key aspects regarding the integration of $\vdash^{\circ}$ in FLP systems.

7. Discussion

7.1. Comparing existential types to $\vdash^{\bullet}$ and $\vdash^{\circ}$

Existentially quantified types are useful when packing values and operations over those values [28, 19], as they allow to pack a value of some *unknown* type together with functions which operate on *that* unknown type. Thus, the value of unknown type can be seen as the representation of the data type, whereas the functions can be seen as the interface of the data type. An elegant way of allowing this programming pattern are *existential data constructors* [29, 20], which are widely accepted in current Haskell systems as well as other FLP systems. They are basically data constructors that mention type variables in its arguments that do not appear in the result type. A typical example of such constructors is the *mkKey* constructor of arity 2 from Example 8 (page 25), which has type $\forall\alpha.\alpha \to (\alpha \to int) \to key$. Noticeably both its arguments contain the type variable α, but its result type *key* Regarding the logical meaning of functional types as implications, the type of *mkKey* is equivalent to $(\exists\alpha.\alpha \to (\alpha \to int)) \to key$, that is from where the term *existential* applied to these constructors comes. In the case of the *mkKey* constructor, the first argument is the concrete and unknown representation, and the second one is a function that accepts a value of that unknown representation and returns an integer. For example, (*mkKey [true,false] length*) and (*mkKey 3 inc)*) are correct values of type *key*, while (*mkKey 1 not*) and (*mkKey [true] inc*) are not. Values of type *key* can be seen as different implementations of keys, and as they are just values, they are first-class citizens that can be passed as functions parameters and returned as functions results.

Using existential data constructors—as formalized in [20]—the program rule *getKey (mkKey X F)* $\rightarrow$ *F X* from Example 8 (page 25) has type *key* $\rightarrow$ *int*. The reason to consider this rule well-typed is the same as in Section 6: *F X* cannot produce any runtime type error since the *mkKey* constructor guarantees that the function F accepts data of the same type of X. However, there are two main restrictions that the use of existential data constructors imposes. The first one is that an existentially quantified type variable cannot go beyond the scope of the matching. For example, a rule like f_1 (*mkKey X F*) $\rightarrow X$ must be rejected because we do not know the exact type of X—it is existentially quantified, so we only know that is has *some* type—and we cannot assign a type to the rule. The second restrictions is that existentially quantified variables cannot be forced to be a more specific type. Thus, the rule f_2 (*mkKey X F*) $\rightarrow$ *not X* will be rejected, as the existential type of X is forced to be *bool*. The same happens with f_3 (*mkKey true F*) $\rightarrow$ *false*, as the first argument of *mkKey*—whose type is existentially quantified—is forced to be *bool*. Considering these restrictions, the formalization in [20] achieves a semantically sound type system and a type inference algorithm—following the ideas of algorithm $\mathcal{W}$ in [1]—which is sound and complete.

Comparing existential data constructors to the type systems presented here, we conclude that $\vdash^{\bullet}$ and existential data constructors are not comparable (in terms of well-typed programs), whereas $\vdash^{\circ}$ seems to be strictly more general than existential data constructors. Regarding the first comparison, notice that the previous rule f_3 (*mkKey true F*) $\rightarrow$ *false* with type f_3 : *key* $\rightarrow$ *bool* is well-typed wrt. $\vdash^{\bullet}$ The reason is that the only opaque variable is F, but as it does not appear in the right-hand side it is not critical. However, as we have explained before, this rule is rejected in systems with existential data constructors. A similar situation are the first two rules of *code* in Example 9 (page 26). They are well-typed according $\vdash^{\bullet}$ (as they do not contain any critical variable) but they are rejected using existential data constructors. On the other hand, the *getKey* function from Example 8—accepted by existential data constructors—is ill-typed wrt. $\vdash^{\bullet}$ as both F and X are critical variables. Regarding the second comparison, we have checked that typical examples of existential data constructors—as *getKey* or all the examples in Section 4 of [20]—are accepted by $\vdash^{\circ}$. However, the previous rule f_3 or the rules of *code* in Example 9 are rejected by existential data constructors, although they are well-typed according $\vdash^{\circ}$. Based on these examples and the tighter treatment of *Skolem constants* compared to opaci-

fying variables, we conjecture that $\vdash^{\circ}$ is more general than existential data constructors, although a formal proof is left as future work.

The observed differences between existential data constructors and the type systems $\vdash^{\bullet}$ and $\vdash^{\circ}$ can be explained by their different motivations. Existential data constructors pursue *abstract data types*, so they want to hide implementations and force the use of interfaces. This is easily observed in $f_3\ (mkKey\ true\ F) \to false$, which is rejected because it tries to *inspect* if the internal representation of the key is the pattern *true*. This is different from the goal pursued by $\vdash^{\bullet}$ and $\vdash^{\circ}$, which want to solve an inherent type problem of HO patterns while giving the maximum liberality possible. That is the reason why the rule of f_3 is accepted, because it does not break type preservation. The expression $f_3\ (mkKey\ true\ F)$ is well-typed and is rewritten to *false* in a type preserving way, while $f_3\ (mkKey\ [1, 2]\ F)$ is well-typed but cannot be rewritten using the rule for f_3. The latter example can be viewed as a point where the type system is sidestepped, as *[1,2]* is matched with *true*. However, this matching does not break type preservation, and moreover, is this ability of matching patterns of different types in opaque positions—as the first argument of *mkKey*—that allow to write functions with a generic behavior as *code* in Example 9 (page 26).

To conclude this comparison, we will remark some difficulties that arise when applying existential data constructors directly to our FLP setting. Intuitively[6], pattern matching introduces new distinct types—called *Skolem constants*—for each existential type variable. For example, in the previously introduced rule $f_2\ (mkKey\ X\ F) \to not\ X$, *mkKey* would be assigned the type $\kappa \to (\kappa \to int) \to key$ (where κ is a fresh Skolem constant), so X and F must have types κ_1 and $(\kappa \to int)$ respectively. This explains why $not\ X$ is ill-typed (as X has type κ) and $F\ X$ has type *int*. It also explains that the rule $f_3\ (mkKey\ true\ F) \to false$ was rejected, since the first argument of *mkKey* cannot have type *bool* but κ. This behavior is based on the fact that patterns are composed by *completely* applied data constructors and variables. Thus, given *mkKey* of type $\forall\alpha.\alpha \to (\alpha \to int) \to key$, the quantified variable α must be replaced by a fresh Skolem constant in every occurrence in a pattern. This rule is broken when considering partially applied constructor symbols, as is the case in our FLP setting. For example, in the pattern $mkKey\ X$ the type of X is completely fixed given the type of the

[6]According to the formalization in [20].

whole pattern. Therefore, although *mkKey* is an existential data constructor, the introduction of Skolem constants is only necessary when applied to two arguments. This situation also happens with function symbols, which can be used to build HO patterns in our setting: Skolem constants can be needed or not depending on the number of arguments of the application.

Another difficulty that a direct adaptation of existential data constructors must address concerns to a more practical aspect: type inference/checking of programs. The introduction of Skolem constants relies on that types of the symbols are known beforehand. This is easily achieved when patterns are formed only by data constructors, as their type is given by the data declaration. However, this cannot be guaranteed for function symbols, as programs may not have explicit type signatures for them. A possible (and sensible) solution for this problem could be to demand type signatures for functions occurring in patterns. Notice that the inference relation $\Vdash^{\bullet}$ does not suffer from this problem, as it first perform a standard Damas-Milner type inference and then it checks for the absence of critical variables using the inferred types for the functions.

7.2. Is $\vdash^{\circ}$ strictly more liberal than $\vdash^{\bullet}$?

The design of the type system $\vdash^{\circ}$ comes from a set of examples that, being interesting in practice, were all rejected by $\vdash^{\bullet}$. This might easily lead to the conclusion that $\vdash^{\circ}$ is more liberal, i.e., that every well-typed expression wrt. $\vdash^{\bullet}$ is also well-typed wrt. $\vdash^{\circ}$ (that was indeed our impression at a first moment). However, this is not true in a technical sense, as the following example shows.

Example 14 (Well-typed expression wrt. $\vdash^{\bullet}$ but ill-typed wrt. $\vdash^{\circ}$). Consider $\mathcal{A} \equiv \{snd : \forall\alpha, \beta.\alpha \to \beta \to \beta, f : \gamma \to \gamma \to \gamma\}$ and the expression $e^{\circ} \equiv \lambda(snd\ (f\ X)).f$. Clearly $\mathcal{A} \vdash^{\bullet} e^{\circ} : (bool \to bool) \to \gamma \to \gamma \to \gamma$ because $\mathcal{A} \vdash e^{\circ} : (bool \to bool) \to \gamma \to \gamma \to \gamma$ and $critVar_{\mathcal{A}}(e^{\circ}) = \emptyset$—since the body of the λ-abstraction does not contain variables. However, e° is ill-typed wrt. $\vdash^{\circ}$. If we apply the (Λ°) rule, the first condition forces a type inference $\mathcal{A} \oplus \{X : \alpha\} \Vdash snd\ (f\ X) : \delta \to \delta | [\alpha/\gamma]$, so $OpacifyingVars = \{\gamma\} \setminus \{\delta\} = \{\gamma\}$ and $\tau_g \equiv \delta \to \delta$. Since we want to derive a type $\tau_g\pi \equiv bool \to bool$ for the pattern $snd\ (f\ X)$, the substitution π must be $[\delta/bool]$, which verifies the second condition $dom(\pi) = \{\delta\} \subseteq ftv(\delta \to \delta) = \{\delta\}$. Then, we have the type derivation $\mathcal{A} \oplus \{X : \gamma\} \vdash^{\circ} snd\ (f\ X) : bool \to bool$. As $vran(\pi) = \emptyset$, it fulfills trivially the fourth condition. However, the third condition cannot

be fulfilled. The only possible derivation for the body of the λ-abstraction is $\mathcal{A} \oplus \{X : \gamma\} \vdash^\circ f : \gamma \to \gamma \to \gamma$—using the (ID$^\circ$) rule—but in this case $ftv(\gamma \to \gamma \to \gamma) \cap OpacifyingVars = \{\gamma\} \neq \emptyset$.

The problem here is that the typing relation $\vdash^\circ$ only checks opacifying type variables, losing their link with data variables. In this example the type of the body of the λ-abstraction is $\gamma \to \gamma \to \gamma$, which contains the type variable γ assumed for X that does not appear in the type of the pattern, so it is opacifying. However, this type variable γ does not come from the use of the data variable X but of the function symbol f, so it does not cause any actual problem in a reduction step.

For $\vdash^\circ$ to be a relaxation of $\vdash^\bullet$ in a strict sense we would need to slightly refine the (Λ°) rule, in particular the third condition $ftv(\tau_e) \cap OpacifyingVars = \emptyset$, where $ftv(\tau_e)$ should be replaced by a finer notion of free type variables in τ_e originated by data variables from t appearing in e°. Alternatively, a simpler approach to solve this problem could be allowing $ftv(\tau_e) \cap OpacifyingVars \neq \emptyset$ provided $critVar_{\mathcal{A}}(\lambda t.e^\circ) = \emptyset$. Developing these ideas is left as future work.

Notice that Example 14 uses a function whose assumption contains free type variables: $f : \gamma \to \gamma \to \gamma$. Otherwise, using $f : \forall\gamma.\gamma \to \gamma \to \gamma$ the expression $\lambda(snd\ (f\ X)).f$ would be well-typed wrt. $\vdash^\circ$. We strongly conjecture that for closed set of assumptions $\vdash^\circ$ is indeed a relaxation of $\vdash^\bullet$, i.e., if $\mathcal{A}$ is closed and $\mathcal{A} \vdash^\bullet e^\circ : \tau$ then $\mathcal{A} \vdash^\circ e^\circ : \tau$. A consequence of this conjecture is that $wt_{\mathcal{A}}(\mathcal{P})$ implies $wt^\circ_{\mathcal{A}}(\mathcal{P})$ when $\mathcal{A}$ is closed, a condition usually assumed in practice when checking program well-typedness.

7.3. The systems $\vdash^\bullet$ and $\vdash^\circ$ as conservative extensions of Damas-Milner

The type systems $\vdash^\bullet$ and $\vdash^\circ$ make a technical treatment of critical variables and opacifying variables resp. in order to support opaque patterns safely from the point of view of type preservation. However, when opaque patterns are not present in programs, $\vdash^\bullet$ and $\vdash^\circ$ behave like the classical Damas-Milner type system (i.e., like $\vdash$). This happens when programs contain only FO patterns and all constructor symbols have transparent type assumptions, the usual situation in Damas-Milner. This justifies to consider them as conservative extensions of Damas-Milner.

It is easy to check that with the limitations of FO patterns and transparent constructors, there cannot be opaque variables in patterns. Therefore there are not critical variables in expressions either, and $\vdash^\bullet$ is the same as $\vdash$. On the other hand, the rules of $\vdash^\circ$ are a subset of the rules of $\vdash$, that

only differs in the treatment of λ-abstractions. Under the mentioned limitations it is clear that patterns in λ-abstractions will not have opacifying type variables: all type variables inferred for data variables will appear in the type of the pattern. If there are not opacifying variables the third and fourth point of rule [Λ°] in Figure 13 hold trivially. However, the first and second points are more technically involved. To show this case consider a well-typed λ-abstraction wrt. $\vdash$:

$$[\Lambda]\frac{\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \quad \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau}{\mathcal{A} \vdash^\circ \lambda t.e : \tau_t \to \tau}$$

From the completeness of $\Vdash$ (Theorem 5, page 20) we know that there exists a type inference $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ (first point) and a type substitution π such that $\tau_g\pi = \tau_t$ and $\alpha_i\pi_g\pi = \tau_i$ for any $i \in [1..n]$. Since there are not opacifying variables then $\bigcup_{i=1}^{n} ftv(\alpha_i\pi_g) \subseteq ftv(\tau_g)$, so $dom(\pi) \subseteq ftv(\tau_g)$ (second point). Therefore, in this restricted scenario $\vdash^\circ$ is also the same as $\vdash$.

7.4. Parametricity

The notion of parametricity of a type system was introduced by Reynolds in [30]. The basic idea of parametricity is that the polymorphic type variables in the types of function symbols cannot be instantiated in the left-hand side of program rules. For example, for a function $f : \forall\alpha.[\alpha] \to [\alpha]$, the program rule $f\ [true] \to true$ breaks parametricity because in its left-hand side the type *bool* has replaced the polymorphic variable α in the type for f. This way parametricity puts limits to the inspection of function arguments that is performed during the pattern matching process: any function with the same type as function f above would be able to inspect the structure of its argument list, corresponding to the type constructor [ ], but would not be allowed to inspect the elements of that list, corresponding to the polymorphic variable α. Hence functions with that type will only return some rearrangement of the input list, maybe dropping or duplicating some of its elements but never introducing new elements.

In [31] Wadler used parametricity to derive his famous "Theorems for free", which are equational rules for expressions based on the polymorphic types of the functions they include. That work was originally developed for the framework of polymorphic λ-calculus, and therefore its applicability to actual FP languages with lazy evaluation, unbounded recursion and other

practical features is limited, so free theorems must be weakened with additional conditions in order to recover soundness [31, 32, 18, 33]. Things get even worse in the FLP setting, as non-determinism invalidates not only free theorems but also equational rules for concrete functions that hold in a deterministic setting [34].

Although the applicability of free theorems is limited in practical FP and FLP languages, in any case in [32, 18, 33, 34], where Damas-Milner typing is used, the basic principle of parametricity still holds. in a typical use of parametricity we can state that only two extensionally different functions with type $\forall\alpha.\alpha \to \mathit{bool}$ can be defined in Haskell, those corresponding to *const true* and *const false*—using *const* function from Haskell's prelude—because parametricity forbids the inspection of the polymorphic argument. On the other hand, in our setting not only free theorems but the more fundamental notion of parametricity is lost. The point is that, through the use of *cont* (as defined in Section 6) and other opaque patterns, in our type systems we can define an infinite number of extensionally different functions with that type, like for example the function g in the program $\{g\ X \to h\ (\mathit{cont}\ X), h\ (\mathit{cont}\ [\,]) \to \mathit{true}, h\ (\mathit{cont}\ (X : Xs)) \to \mathit{false}\}$, with $g : \forall\alpha.\alpha \to \mathit{bool}$, $h : \mathit{container} \to \mathit{bool}$ and $\mathit{cont} : \forall\alpha.\alpha \to \mathit{container}$, which is accepted both by $\vdash^{\bullet}$ and $\vdash^{\circ}$. In this case the break of parametricity is less apparent because the polymorphic variable α which is replaced by $[\beta]$ in the rules is hidden in the type for h by the opacity of *cont*. As a consequence free theorems are invalidated by the lack of parametricity but, as free theorems were already heavily compromised by non-determinism, this has a limited impact in the FLP setting.

We remark that all these new problems about parametricity do not appear in programs where opaque patterns are absent, since in that case our type system coincides with Damas-Milner. This contrasts with [15], where we proposed another type system that breaks parametricity even further. In that work the instantiation of polymorphic type variables is allowed to be performed not only in opaque patterns but in any pattern of the program. This leads to a very liberal type system that restricts the programs only as needed to ensure type preservation, almost reaching the limit of permissiveness, but where, conversely, type annotations are needed for every function of the program as no type inference procedure for programs is available.

8. 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 [13] treats with technical detail a type system for functional logic programming. Our paper makes clear contributions when compared to [13]:

- By introducing the notion of critical variables, we are more liberal in the treatment of opaque data variables, but still preserving the essential property of type preservation; 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* [19, 20] or *generalized algebraic data types* [22, 35], a connection that we plan to further investigate in the future.

- We propose also a variant of the type system that moves the emphasis from opaque data variables to type variables causing opacity. This turns out to lead to a wider range of practical examples which make a type-safe use of opaque variables.

- Our type system considers local pattern bindings and λ-abstractions (also with patterns), that were missing in [13]. In addition to that, we have made a rather exhaustive analysis and formalization of different possibilities for polymorphism in local bindings.

- Type preservation was proved in [13] wrt. a narrowing calculus. Here we do it wrt. a small-step operational semantics closer to real computations [9], although, as in [13], λ-abstractions are not considered by the operational semantics and the type preservation results.

- In [13] programs came with explicit type declarations. Here we provide algorithms for inferring types for programs without such declarations that can become part of the typing stage of a FL compiler.

- We have implemented and integrated into a branch of the system Toy[7] [2] the stratified inference procedure presented in Section 5. Since the system is a rapid prototype directly based on actual Toy, adopting in particular its syntax, it presents some minor differences with

[7] Available at `http://gpd.sip.ucm.es/Toy2SafeOpaquePatterns`

the type system presented here: it only supports monomorphic *where* declarations instead of the variety of let-bindings of Section 1.2, and it does not provide syntactic support for defining constructors with non-transparent types.

We have in mind several lines for future work. We plan to complete the mentioned branch of the Toy system adding support for the different kinds of let-expressions and non-transparent constructors. We also plan to refine the relaxed typing relation $\vdash^{\circ}$ to be a strict generalization of $\vdash^{\bullet}$, and develop for $\vdash^{\circ}$ effective type checking/inference procedures in order to be integrated into the system. In this respect, it could be interesting to characterize our type systems as instances of the HM(X) scheme presented in [36], a nice framework into which different extensions of Hindley-Milner system can be fit. In particular, expressing $\vdash^{\circ}$ as an instance of HM(X)—and proving that the underlying constraint system fulfills some properties—will provide a sound and complete type inference algorithm for free.

The type systems and the results in the paper apply to programs without extra variables (variables occurring only in right hand sides of rules) and with reductions performed by let-rewriting. The range of applicability is wide, since it is known that in some semantic sense extra variables can be dispensed from programs [37, 38]. Still, programming with extra variables and using narrowing instead of rewriting are important practical and theoretical features of functional logic programming [4, 39]. Therefore we want to investigate how our type systems and results extend to cope with those features, where narrowing is realized by means of *let-narrowing* reductions of [9], and taking into account known problems [13, 40] in the interaction of HO narrowing and types. We have achieved some first results in [41].

Acknowledgments *We thank the reviewers for their stimulating criticisms and comments, which have been very useful for improving the paper. We also thank the reviewers of the previous workshop version of the paper [42].*

[1] L. Damas, R. Milner, Principal type-schemes for functional programs, in: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82), ACM, 1982, pp. 207–212.

[2] F. López-Fraguas, J. Sánchez-Hernández, Toy: A multiparadigm declarative system, in: Proceedings of the 10th International Conference on

Rewriting Techniques and Applications (RTA '99), volume 1631 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 244–247.

[3] M. Hanus, Curry: An integrated functional logic language (version 0.8.2), Available at `http://www.informatik.uni-kiel.de/~curry/report.html`, 2006.

[4] M. Hanus, Multi-paradigm declarative languages, in: Proceedings of the 23rd International Conference on Logic Programming (ICLP '07), volume 4670 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 45–75.

[5] J. J. Moreno-Navarro, J. Mariño, A. del Pozo-Pietro, Á. Herranz-Nieva, J. García-Martín, Adding type classes to functional-logic languages, in: 1996 Joint Conference on Declarative Programming (APPIA-GULP-PRODE '96), pp. 427–438.

[6] W. Lux, Adding Haskell-style overloading to Curry, in: Workshop of Working Group 2.1.4 of the German Computing Science Association GI, pp. 67–76.

[7] E. Martin-Martin, Type classes in functional logic programming, in: Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '11), ACM, 2011, pp. 121–130.

[8] J. González-Moreno, T. Hortalá-González, M. Rodríguez-Artalejo, A higher order rewriting logic for functional logic programming, in: Proceedings of the 14th International Conference on Logic Programming (ICLP '97), MIT Press, 1997, pp. 153–167.

[9] F. López-Fraguas, J. Rodríguez-Hortalá, J. Sánchez-Hernández, Rewriting and call-time choice: the HO case, in: Proceedings of the 9th International Symposium on Functional and Logic Programming (FLOPS '08), volume 4989 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 147–162.

[10] R. Caballero, F. López-Fraguas, A functional-logic perspective on parsing, in: Proceedings of the 4th International Symposium on Functional and Logic Programming (FLOPS '99), Springer, 1999, pp. 85–99.

[11] R. Caballero, Y. García-Ruiz, F. Sáenz-Pérez, Integrating xpath with the functional-logic language toy, in: R. Rocha, J. Launchbury (Eds.), Proceedings of the 13th International Symposium on Practical Aspects of Declarative Languages (PADL '11), volume 6539 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 145–159.

[12] S. Fischer, Call-time choice and extensionality - post to the Curry mailing list, `http://www.informatik.uni-kiel.de/~curry/listarchive/0995.html`, 2011.

[13] J. González-Moreno, T. Hortalá-González, M. Rodríguez-Artalejo, Polymorphic types in functional logic programming, Journal of Functional and Logic Programming 2001 (2001).

[14] 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`, 2008.

[15] F. López-Fraguas, E. Martin-Martin, J. Rodríguez-Hortalá, Liberal typing for functional logic programs, in: Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10), volume 6461 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 80–96.

[16] W. Lux, Münster Curry user's guide, release 0.9.11, `http://danae.uni-muenster.de/~lux/curry/user.html`, 2007.

[17] A. K. Wright, Simple imperative polymorphism, Lisp and Symbolic Computation 8 (1995) 343–355.

[18] P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, A history of Haskell: Being lazy with class, in: Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL III), ACM, 2007, pp. 12–1–12–55.

[19] J. C. Mitchell, G. D. Plotkin, Abstract types have existential type, ACM Transactions on Programming Languages and Systems 10 (1988) 470–502.

[20] K. Läufer, M. Odersky, Polymorphic type inference and abstract data types, ACM Transactions on Programming Languages and Systems 16 (1994) 1411–1430.

[21] H. Xi, C. Chen, G. Chen, Guarded recursive datatype constructors, SIGPLAN Notices 38 (2003) 224–235.

[22] J. Cheney, R. Hinze, First-Class Phantom Types, Technical Report TR2003-1901, Cornell University, 2003.

[23] L. Damas, Type Assignment in Programming Languages, Ph.D. thesis, University of Edinburgh, 1985. Also appeared as Technical report CST-33-85.

[24] S. Peyton Jones, The Implementation of Functional Programming Languages, Prentice-Hall International Series in Computer Science, 1987.

[25] A. Mycroft, Polymorphic type schemes and recursive definitions, in: Proceedings of the 6th Colloquium on International Symposium on Programming, Springer, 1984, pp. 217–228.

[26] A. J. Kfoury, J. Tiuryn, P. Urzyczyn, Type reconstruction in the presence of polymorphic recursion, ACM Transactions on Programming Languages and Systems 15 (1993) 290–311.

[27] R. Tarjan, Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (1972) 146–160.

[28] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys 17 (1985) 471–522.

[29] N. Perry, The Implementation of Practical Functional Programming Languages, Ph.D. thesis, Imperial College, 1991.

[30] J. C. Reynolds, Types, abstraction and parametric polymorphism, Information Processing (1983) 513–523.

[31] P. Wadler, Theorems for free!, in: Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89), ACM, 1989, pp. 347–359.

[32] D. Seidel, J. Voigtländer, Automatically generating counterexamples to naive free theorems, in: Proceedings of 10th International Symposium on Functional and Logic Programming (FLOPS '10), volume 6009 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 175–190.

[33] P. Johann, J. Voigtländer, Free theorems in the presence of *seq*, in: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04), ACM, 2004, pp. 99–110.

[34] J. Christiansen, D. Seidel, J. Voigtländer, Free theorems for functional logic programs, in: Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV '10), ACM, 2010, pp. 39–48.

[35] T. Schrijvers, S. Peyton Jones, M. Sulzmann, D. Vytiniotis, Complete and decidable type inference for GADTs, in: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09), ACM, 2009, pp. 341–352.

[36] M. Odersky, M. Sulzmann, M. Wehr, Type inference with constrained types, Theory and Practice of Object Systems 5 (1999) 35–55.

[37] S. Antoy, M. Hanus, Overlapping rules and logic variables in functional logic programs, in: Proceedings of the 22nd International Conference on Logic Programming (ICLP '06), volume 4079 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 87–101.

[38] J. de Dios Castro, F. J. López-Fraguas, Extra variables can be eliminated from functional logic programs, Electronic Notes in Theoretical Computer Science 188 (2007) 3–19.

[39] S. Antoy, M. Hanus, Functional logic programming, Communications of the ACM 53 (2010) 74–85.

[40] S. Antoy, A. Tolmach, Typed higher-order narrowing without higher-order strategies, in: Proceedings of the 4th International Symposium on Functional and Logic Programming (FLOPS '99), volume 1722 of *Lecture Notes in Computer Science*, Springer, 1999, pp. 335–352.

[41] F. López-Fraguas, E. Martin-Martin, J. Rodríguez-Hortalá, 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), ACM, 2012, pp. 83–92.

[42] F. López-Fraguas, E. Martin-Martin, J. Rodríguez-Hortalá, New results on type systems for functional logic programming, in: 18th International Workshop on Functional and (Constraint) Logic Programming (WFLP '09), Revised Selected Papers, volume 5979 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 128–144.

[43] E. Martin-Martin, Advances in Type Systems for Functional Logic Programming, Master's thesis, Universidad Complutense de Madrid, 2009. Available at `http://gpd.sip.ucm.es/enrique/publications/master/masterThesis.pdf`.

Appendix A. Detailed proofs for the results

This section contains detailed proofs for all the results in the paper. In some proofs, we have shortened or omitted some cases for the sake of conciseness. However, in these cases the complete proof can be found in [43].

In this section we use some standard operations over type substitutions. Given a set of type variables $T \subseteq \mathcal{TV}$, the notation $\pi|_T$ represents the substitution π restricted to T. The *simultaneous composition* of π_1 and π_2—denoted as $\pi_1 + \pi_2$—is well defined only when $dom(\pi_1) \cap dom(\pi_2) = \emptyset$:

$$\alpha(\pi_1 + \pi_2) = \begin{cases} \alpha\pi_1 & \text{if } \alpha \in dom(\pi_1) \\ \alpha\pi_2 & \text{otherwise} \end{cases}$$

The following two results are easy remarks about type inference/derivation that will be used in several proofs.

Remark 1.
If $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau$ then we can assume that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash e : \tau'|\pi$ such that $\mathcal{A}\pi = \mathcal{A}$.

Explanation. Intuitively, the inference finds the most general type substitution that must be applied to the set of assumptions in order to derive a type for the expression. In this case it is possible to derive a type using $\mathcal{A}$ so the type substitution π from the inference would not need to affect $\mathcal{A}$, just only $\overline{\alpha_n}$ and the fresh variables generated during inference. □

Remark 2.
Any type derivation $\mathcal{A} \vdash e : \tau$ contains a type derivation for every subexpression e' of e. That is, the derivation will have a part of the tree rooted by $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e' : \tau'$, being τ' a suitable type for e', and being $\{\overline{X_n : \tau_n}\}$ a set of assumptions over variables in 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 $\{\overline{X_n : \tau_n}\}$ will be empty because the only rules used to type a pattern are $[ID]$ and $[APP]$.

Appendix A.1. Proof for Proposition 1

Proposition 1
Let t be a pattern such that $wt_{\mathcal{A}}(t)$. $X_i \in \{\overline{X_n}\} = var(t)$ is opaque wrt. $\mathcal{A}$ iff $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ and $ftv(\alpha_i\pi_g) \nsubseteq ftv(\tau_g)$.

Proof.

$\Longrightarrow$ Since X_i is opaque then there are τ_n, τ such that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$ and $ftv(\tau_i) \nsubseteq ftv(\tau)$. This type derivation can be written as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})[\overline{\alpha_n/\tau_n}] \vdash t : \tau$, so by Theorem 5 $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ and there exists some $\pi'' \in TSub$ s.t. $\tau_g\pi'' = \tau$, $\mathcal{A}\pi_g\pi'' = \mathcal{A}$ and $\alpha_i\pi_g\pi'' = \tau_i$. To prove

$$ftv(\tau_i) \nsubseteq ftv(\tau) \Longrightarrow ftv(\alpha_i\pi_g) \nsubseteq ftv(\tau_g)$$

we prove the equivalent implication

$$ftv(\alpha_i\pi_g) \subseteq ftv(\tau_g) \Longrightarrow ftv(\tau_i) \subseteq ftv(\tau)$$

which is trivial since $\alpha_i\pi_g\pi'' = \tau_i$ and $\tau_g\pi'' = \tau$.

$\Longleftarrow$ By Theorem 4 $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_g \vdash t : \tau_g$, and $ftv(\alpha_i\pi_g) \nsubseteq ftv(\tau_g)$. Since $wt_{\mathcal{A}}(t)$ then by Remark 1 $\mathcal{A}\pi_g = \mathcal{A}$, so $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_g}\} \vdash t : \tau_g$.

□

Appendix A.2. Proof for Theorem 1

In order to prove Theorem 1 we first need to prove some remarks and auxiliary lemmas. The following remark relates free type variables and generic instances.

Remark 3.
If $\sigma \equiv \forall\overline{\alpha_n}.\tau \succ \sigma'$ then $ftv(\sigma) \subseteq ftv(\sigma')$.

Proof. It is clear from the alternative characterization of generic instance (Section 2). Let α be an arbitrary type variable in $ftv(\sigma)$, i.e., $\alpha \in var(\tau) \setminus \{\overline{\alpha_n}\}$. Then $\alpha \notin dom([\overline{\alpha_n/\tau_n}])$ and $\alpha \notin \{\overline{\beta_m}\}$. Therefore $\alpha \in var(\tau[\overline{\alpha_n/\tau_n}]) \setminus \{\overline{\beta_m}\} = ftv(\sigma')$. □

The following lemma states that if type inference of an expression e succeeds with $\mathcal{A}\pi$, then type inference for e also succeeds with $\mathcal{A}$. Moreover, there is a precise relationship between the types and substitutions found in both cases.

Lemma 2.
If $\mathcal{A}\pi \Vdash e : \tau_1|\pi_1$ *then* $\exists\tau_2 \in SType, \pi_2\ \pi'' \in TSub$ *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 $\pi'' \in TSub$ such that $\tau_2\pi'' = \tau_1$ and $\mathcal{A}\pi_2\pi'' = \mathcal{A}\pi\pi_1$. □

The following is an important result used to prove that type derivations $\vdash^\bullet$ are closed under type substitutions—part a) of Theorem 1. It states that if a pattern t has type with $\mathcal{A}$ and $\mathcal{A}\pi$ (for some assumptions for its variables) then the opaque variables wrt. $\mathcal{A}\pi$ are a subset of the opaque variables wrt. $\mathcal{A}$, i.e., they decrease under application of substitutions.

Lemma 3 (Decrease of opaque variables)**.**
If $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau$ *and* $\mathcal{A}\pi \oplus \{\overline{X_n : \tau'_n}\} \vdash t : \tau'$ *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. $\mathcal{A}\pi$.

$\mathcal{A} \oplus \{\overline{X_n : \tau_n}\}$ is the same as $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}[\overline{\alpha_n/\tau_n}]$, so by Theorem 5 we have that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_1 | \pi_1$. Then the transparent variables of t will be those X_i such that $ftv(\alpha_i \pi_1) \subseteq ftv(\tau_1)$.

$\mathcal{A}\pi \oplus \{\overline{X_n : \tau'_n}\}$ is the same as $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi[\overline{\alpha_n/\tau'_n}]$, because we can assume that the variables $\overline{\alpha_n}$ do not appear in π. Then by Theorem 5 $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi \Vdash t : \tau_2 | \pi_2$, and by Lemma 2 there exists a type substitution π'' such that $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi\pi_2 = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_1\pi''$ 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:

$$\begin{array}{rl} ftv(\alpha_i\pi_1) \subseteq ftv(\tau_1) & X_i \text{ is transparent wrt. } \mathcal{A} \\ ftv(\alpha_i\pi_1\pi'') \subseteq ftv(\tau_1\pi'') & \\ ftv(\alpha_i\pi\pi_2) \subseteq ftv(\tau_2) & X_i \text{ is transparent wrt. } \mathcal{A}\pi \end{array}$$

□

The following lemma states that given an expression e without critical variables, if you replace in e a variable by any other expression without critical variables, then the resulting expression has not critical variables either.

Lemma 4.
If $critVar_{\mathcal{A}}(e) = \emptyset$ and $critVar_{\mathcal{A}}(e') = \emptyset$ then $critVar_{\mathcal{A}}(e[X/e']) = \emptyset$.

Proof. Straightforward by induction over the structure of e. □

The next lemma states that if all fresh variables (wrt. a set of assumptions $\mathcal{A}$) of a simple type τ do not appear in a type substitution π, then generalizing τ wrt. $\mathcal{A}$ and applying π is the same as generalizing the type with the substitution applied ($\tau\pi$) wrt. the set of assumptions with the substitution applied ($\mathcal{A}\pi$).

Lemma 5.
Let $\mathcal{A}$ be a set of assumptions, τ a type and $\pi \in TSub$ 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 $\alpha \notin vran(\pi)$. 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 $\alpha\pi = \alpha$. Since $\alpha \notin vran(\pi)$ and $\alpha \notin ftv(\mathcal{A})$ then it cannot appear in $\mathcal{A}\pi$, so α will also be generalized in $Gen(\tau\pi, \mathcal{A}\pi)$.

□

With the previous results we can now prove Theorem 1.

Theorem 1 (Properties of the typing relations)

a) *If* $\mathcal{A} \vdash^? e : \tau$ *then* $\mathcal{A}\pi \vdash^? e : \tau\pi$, *for any* $\pi \in TSub$.
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* $\mathcal{A} \oplus \{X : \tau_x\} \vdash^? e[X/e'] : \tau$.
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 $\mathcal{A} \vdash e : \tau$. The only interesting cases are $[\text{LET}^X_{pm}]$ and $[\text{LET}_p]$

- $[\mathbf{LET}^X_{pm}]$ The derivation will be of the form

$$[\mathbf{LET}^X_{pm}]\ \frac{\mathcal{A} \vdash e_1 : \tau_x \qquad \mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A})\} \vdash e_2 : \tau}{\mathcal{A} \vdash let^X_{pm}\ X = e_1\ in\ e_2 : \tau}$$

  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 replaces

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 we have $(\mathcal{A} \oplus \{X : Gen(\tau_x\pi', \mathcal{A})\})\pi \vdash e_2 : \tau\pi$, 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 5 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

$$[\mathbf{LET}^X_{pm}] \frac{\mathcal{A}\pi \vdash e_1 : \tau_x\pi'\pi \qquad \mathcal{A}\pi \oplus \{X : Gen(\tau_x\pi'\pi, \mathcal{A}\pi)\} \vdash e_2 : \tau\pi}{\mathcal{A}\pi \vdash let^X_{pm}\ X = e_1\ in\ e_2 : \tau\pi}$$

- $[\mathbf{LET}_p]$ 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 $\overline{\tau_n}$.

---

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 $\mathcal{A}\pi \vdash e : \tau\pi$. The fact that $critVar_{\mathcal{A}\pi}(e) = \emptyset$ follows easily from the decrease of opaque variables stated in Lemma 3

---

b.1)Let be s a symbol which does not appear in e. Then $\mathcal{A} \vdash e : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$.

Both directions are proved easily by induction over the size of the derivation tree.

---

b.2) Let be s a symbol which does not appear in e, and σ_s any type-scheme. 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, Theorem 1-b ensures that $\mathcal{A} \oplus \{s : \sigma_s\} \vdash e : \tau$.

It is clear 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. 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 we have $\mathcal{A} \vdash e : \tau$. 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* $\mathcal{A} \oplus \{X : \tau_x\} \vdash e[X/e'] : \tau$.

Easily by induction over the size of the expression e.

---

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 4 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) If $\mathcal{A} \oplus \{s : \sigma\} \vdash e : \tau$ *and* $\sigma' \succ \sigma$, *then* $\mathcal{A} \oplus \{s : \sigma'\} \vdash e : \tau$.

By induction over the size of the derivation tree. All the cases are straightforward with the exception of [LET^X_{pm}] and [LET_p]. Since they are pretty similar, we only explain the [LET^X_{pm}] case:

[$\mathbf{LET}^X_{pm}$] We assume that $s \neq X$. The type derivation takes the form:

$$[\mathbf{LET}^X_{pm}]\frac{\begin{array}{c}\mathcal{A} \oplus \{s : \sigma\} \vdash e_1 : \tau_x \\ (\mathcal{A} \oplus \{s : \sigma\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma\})\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \oplus \{s : \sigma\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

By the Induction Hypothesis we have $\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau_x$. As $\sigma' \succ \sigma$ then by Remark 3 $ftv(\sigma') \subseteq ftv(\sigma)$. 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 $\mathcal{A}$ any assumption 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 $(\mathcal{A} \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\}) \oplus \{s : \sigma'\} \vdash e_2 : \tau$. Therefore:

$$[\mathbf{LET}^X_{pm}] \frac{\mathcal{A} \oplus \{s : \sigma'\} \vdash e_1 : \tau_x \quad (\mathcal{A} \oplus \{s : \sigma'\}) \oplus \{X : Gen(\tau_x, \mathcal{A} \oplus \{s : \sigma'\})\} \vdash e_2 : \tau}{\mathcal{A} \oplus \{s : \sigma'\} \vdash let_{pm}\ X = e_1\ in\ e_2 : \tau}$$

□

Appendix A.3. Proof for Theorem 2

In order to prove Theorem 2 we use the following easy remark:

Remark 4.
If $ftv(\mathcal{A}) = ftv(\mathcal{A}')$ then $Gen(\tau, \mathcal{A}) = Gen(\tau, \mathcal{A}')$

Theorem 2 (Type preservation of the let transformation)

Assume $\mathcal{A} \vdash^\bullet e : \tau$ and let $\mathcal{P} \equiv \{\overline{f_{X_n}\ t_n \to X_n}\}$ be the rules of the projection functions needed in the transformation of e according to Figure 8. Let also $\mathcal{A}'$ be the set of assumptions over these functions, defined as $\mathcal{A}' \equiv \{\overline{f_{X_n} : Gen(\tau_{X_n}, \mathcal{A})}\}$, where τ_{X_i} is the most general type such that $\mathcal{A} \vdash^\bullet \lambda t_i.X_i : \tau_{X_i}$. Then $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet \Psi(e) : \tau$ and $wt_{\mathcal{A} \oplus \mathcal{A}'}(\mathcal{P})$.

Proof. By structural induction over the expression e. The interesting cases are those for let-expressions containing compound patterns:

- $let_m\ t = e_1\ in\ e_2$) In this case the original type derivation is:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \vdash let_m\ t = e_1\ in\ e_2 : \tau}$$

It is easy to see that if $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t$ then $\mathcal{A} \vdash \lambda t.X_i : \tau_t \to \tau_i$. The assumptions over the projections functions in $\mathcal{A}'$ will be $\{\overline{f_{X_n} : Gen(\tau'_t \to \tau'_n, \mathcal{A})}\}$, where $\mathcal{A} \vdash^{\bullet} \lambda t.X_i : \tau'_t \to \tau'_i$ and $\tau'_t \to \tau'_i$ is the most general type for the λ-abstraction, so $(\tau'_t \to \tau'_i)\pi = \tau_t \to \tau_i$ for some π. Therefore we can be sure that $Gen(\tau'_t \to \tau'_i, \mathcal{A}) \succ \tau_t \to \tau_i$, because π replaces 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 contain assumptions over projection functions needed in e_1 ($\mathcal{A}^1$), e_2 ($\mathcal{A}^2$) and the pattern t ($\mathcal{A}^t \equiv \{\overline{f_{X_n} : Gen(\tau'_t \to \tau'_n, \mathcal{A})}\}$); so $\mathcal{A}' = \mathcal{A}^1 \oplus \mathcal{A}^2 \oplus \mathcal{A}^t$. Then we can build the type derivation:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\} \vdash Y : \tau_t \\ \mathcal{A} \oplus \mathcal{A}' \vdash \Psi(e_1) : \tau_t \\ \mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ \Psi(e_2) : \tau\end{array}}{\mathcal{A} \oplus \mathcal{A}' \vdash let_m\ Y = \Psi(e_1)\ in\ \overline{let_m\ X_n = f_{X_n}\ Y}\ in\ \Psi(e_2) : \tau}$$

where the derivation $\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ \Psi(e_2) : \tau$ is

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}_Y \vdash f_{X_1}\ Y : \tau_1 \qquad [\mathbf{LET}_m]\dfrac{\begin{array}{c}\mathcal{A}_Y \oplus \{\overline{X_n : \tau_n}\} \vdash \Psi(e_2) : \tau \\ \vdots \\ \mathcal{A}_Y \oplus \{X_1 : \tau_1\} \vdash \ldots\ in\ \Psi(e_2)\end{array}} \qquad \mathcal{A}_Y \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1\end{array}}{\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ \Psi(e_2) : \tau}$$

(being $\mathcal{A}_Y \equiv \mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\}$).

$\mathcal{A}_Y \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1$ is just the application of **[ID]** rule. By the Induction Hypothesis $\mathcal{A} \oplus \mathcal{A}^1 \vdash \Psi(e_1) : \tau_t$, and by Theorem 1-b we can add the assumptions $\mathcal{A}^2 \oplus \mathcal{A}^t$, obtaining $\mathcal{A} \oplus \mathcal{A}' \vdash \Psi(e_1) : \tau_t$. The type derivation $\mathcal{A}_Y \vdash f_{X_1}\ Y : \tau_1$ is straightforward because $Gen(\tau'_t \to \tau'_i, \mathcal{A}) \succ \tau_t \to \tau_i$ 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 $\overline{X_n}$ that are introduced by the pattern in

the original expression: $\{\overline{X_n : \tau_n}\}$. Then by the Induction Hypothesis $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \mathcal{A}^2 \vdash \Psi(e_2) : \tau$, and by Theorem 1-b we can add the rest of the assumptions and obtain $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \oplus \mathcal{A}^2 \oplus \mathcal{A}^1 \oplus \mathcal{A}^t \oplus \{Y : \tau_t\} \vdash \Psi(e_2) : \tau$. Reorganizing the set of assumptions (since the symbols are all different), we obtain $\mathcal{A}_Y \oplus \{\overline{X_n : \tau_n}\} \vdash \Psi(e_2) : \tau$.

- $let_{pm}\ t = e_1\ in\ e_2$) This case is equal to the previous one.
- $let_p\ t = e_1\ in\ e_2$) The type derivation will be:

$$[\mathbf{LET}_p]\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau_t \\ \mathcal{A} \vdash e_1 : \tau_t \\ \mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \vdash e_2 : \tau\end{array}}{\mathcal{A} \vdash let_p\ t = e_1\ in\ e_2 : \tau}$$

As in the previous case, $\mathcal{A}'$ will be $\{\overline{f_{X_n} : Gen(\tau'_t \to \tau'_n, \mathcal{A})}\}$, where $\mathcal{A} \vdash^{\bullet} \lambda t.X_i : \tau'_t \to \tau'_i$ and $\tau'_t \to \tau'_i$ is the most general type for the λ-abstraction. In addition, $Gen(\tau'_t \to \tau'_i, \mathcal{A}) \succ \tau_t \to \tau_i$ and $\mathcal{A}' \equiv \mathcal{A}^1 \oplus \mathcal{A}^2 \oplus \mathcal{A}^t$. Then we can build a type derivation:

$$[\mathbf{LET}_p]\frac{\begin{array}{c}\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : \tau_t\} \vdash Y : \tau_t \\ \mathcal{A} \oplus \mathcal{A}' \vdash \Psi(e_1) : \tau_t \\ \mathcal{A}'_1 \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ \Psi(e_2) : \tau\end{array}}{\mathcal{A} \oplus \mathcal{A}' \vdash let_p\ Y = \Psi(e_1)\ in\ \overline{let_p\ X_n = f_{X_n}\ Y}\ in\ \Psi(e_2) : \tau}$$

where the derivation $\mathcal{A}_Y \vdash let_m\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ \Psi(e_2) : \tau$ is

$$[\mathbf{LET}_p]\frac{\begin{array}{c}\mathcal{A}'_1 \vdash f_{X_1}\ Y : \tau_1 \qquad \mathcal{A}'_1 \oplus \{X_1 : \tau_1\} \vdash X_1 : \tau_1 \\ [\mathbf{LET}_p]\ \dfrac{\begin{array}{c}\vdots \\ \mathcal{A}'_2 \vdash let_p\ X_2 = f_{X_2}\ Y\ in \ldots in\ \Psi(e_2)\end{array}}{\begin{array}{c}\mathcal{A}'_{n+1} \vdash \Psi(e_2) : \tau \\ \vdots\end{array}}\end{array}}{\mathcal{A}'_1 \vdash let_p\ X_1 = f_{X_1}\ Y\ in\ \ldots\ in\ \Psi(e_2) : \tau}$$

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, $\mathcal{A} \oplus \mathcal{A}' \vdash \Psi(e_1) : \tau_t$ and every derivation $\mathcal{A}'_i \vdash f_{X_i}\ Y : \tau_i$ is valid since $\mathcal{A}'_i \vdash Y : \tau_t$. Notice that by Remark 4 $Gen(\tau_t, \mathcal{A}) = Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')$ 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: $\{\overline{X_n : Gen(\tau_n, \mathcal{A})}\}$. By the Induction Hypothesis, we know that $\mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \oplus \mathcal{A}^2 \vdash \Psi(e_2) : \tau$; 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 $\mathcal{A} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \oplus \mathcal{A}^2 \oplus \mathcal{A}^1 \oplus \mathcal{A}^t \oplus \{Y : Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')\} \vdash \Psi(e_2) : \tau$. Then reorganizing the assumptions we obtain $\mathcal{A} \oplus \mathcal{A}' \oplus \{Y : Gen(\tau_t, \mathcal{A} \oplus \mathcal{A}')\} \oplus \{\overline{X_n : Gen(\tau_n, \mathcal{A})}\} \vdash \Psi(e_2) : \tau$. Since $Gen(\tau_i, \mathcal{A}) = Gen(\tau_i, \mathcal{A}'_i)$ then the previous derivation is equal to $\mathcal{A}'_{n+1} \vdash \Psi(e_2) : \tau$.

In all these cases $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$ holds. We have that $\mathcal{A} \vdash^\bullet \lambda t_i.X_i : \tau_{X_i}$ and τ_{X_i} is the most general type for this λ-abstraction. Then by Theorem 1-*b)*, we can add the assumptions $\mathcal{A}'$ over the projecting functions, obtaining $\mathcal{A} \oplus \mathcal{A}' \vdash^\bullet \lambda t_i.X_i : \tau_{X_i}$. Clearly, τ_{X_i} is a variant of $Gen(\tau_{X_i}, \mathcal{A}) = (\mathcal{A} \oplus \mathcal{A}')(f_{X_i})$. Therefore for every projecting rule $wt_{\mathcal{A}\oplus\mathcal{A}'}(f_{X_i}\ t_i \to X_i)$—$\mathcal{A} \oplus \mathcal{A}' \vdash \lambda t_i.X_i : \tau_{X_i}$ and τ_{X_i} is a variant of $(\mathcal{A} \oplus \mathcal{A}')(f_{X_i})$—so $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

Moreover, the most general type τ_{X_i} for $\lambda t_i.X_i$ is always defined. From $\mathcal{A} \vdash^\bullet e : \tau$ we have that $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash t_i : \tau_{t_i}$—see the previous cases—and the projected variables X_i are not opaque in t_i wrt. $\mathcal{A}$. Therefore we can build the type derivation for the λ-abstraction $\mathcal{A} \vdash \lambda t_i.X_i : \tau_{t_i} \to \tau_i$. By completeness and soundness of the type inference (Theorems 5 and 4) we know that $\mathcal{A} \vdash \lambda t_i.X_i : \tau_{X_i}$ and τ_{X_i} is the most general type for the λ-abstraction. Finally, as X_i is not opaque in t_i wrt. $\mathcal{A}$ then $\mathcal{A} \vdash^\bullet \lambda t_i.X_i : \tau_{X_i}$. □

Appendix A.4. Proof for Theorem 3

In order to prove Theorem 3 we first state an easy remark regarding the well-typedness of programs when the set of assumptions is extended with assumptions for variables.

Remark 5.
If $wt_{\mathcal{A}}(\mathcal{P})$ and $\mathcal{A}'$ is a set of assumptions for variables, then $wt_{\mathcal{A}\oplus\mathcal{A}'}(\mathcal{P})$.

Explanation. 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. □

We also need an auxiliary lemma stating that when reducing an expression without critical variables using any let-rewriting rule, the resulting expression does not contain any critical variable.

Lemma 6. *If* $wt_{\mathcal{A}}(\mathcal{P})$, $e \to^{l} e'$ *and* $critVar_{\mathcal{A}}(e) = \emptyset$ *then* $critVar_{\mathcal{A}}(e') = \emptyset$.

Proof. By case distinction on the let-rewriting rule applied. The proofs for the cases (Fapp)–(LetAp) follow easily considering that if $e \to^{l} e'$ then $fv(e') \subseteq fv(e)$—notice that extra variables in right-hand sides of rules are not allowed. For the (Contx) rule, it follows from the fact that $fv(e') \subseteq fv(e)$ implies $fv(\mathcal{C}[e']) \subseteq fv(\mathcal{C}[e])$. □

The next lemma is important when proving the (Contx) case of Theorem 3. It states that in a type derivation you can replace one subexpression e by other expression e' if they have the same type in that place.

Lemma 7.
If a derivation for $\mathcal{A} \vdash \mathcal{C}[e] : \tau$ *contains 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. Easily by induction on the structure of the contexts, using Remark 2. □

Theorem 3 (Type Preservation)
If $\mathcal{A} \vdash^{\bullet} e : \tau$ *and* $wt_{\mathcal{A}}(\mathcal{P})$ *and* $\mathcal{P} \vdash e \to^{l} e'$ *then* $\mathcal{A} \vdash^{\bullet} e' : \tau$.

Proof. We proceed by case distinction over the rule of the let-rewriting relation $\to^{l}$ (Figure 9, page 16) that we use to reduce e to e'. By Lemma 6 we know that $critVar_{\mathcal{A}}(e') = \emptyset$, so we only have to prove that $\mathcal{A} \vdash e' : \tau$. For the sake of conciseness we only explain the most interesting cases. However the complete proof can be found in [43].

(Fapp) We consider a function which accepts two arguments, but the proof for any other number of arguments follows the same ideas. If we reduce an expression e using the (Fapp) rule then the step is $e \equiv f\ t_1\theta\ t_2\theta \to^{l} r\theta$ (being $f\ t_1\ t_2 \to r$ a rule in $\mathcal{P}$ and $\theta \in PSub$). In this case we want to prove that $\mathcal{A} \vdash r\theta : \tau$. Since $wt_{\mathcal{A}}(\mathcal{P})$, then $\mathcal{A} \vdash^{\bullet} \lambda t_1.\lambda t_2.r : \tau_1' \to \tau_2' \to \tau_r'$, being $\tau_1' \to \tau_2' \to \tau_r'$ a variant of $\mathcal{A}(f)$. We assume that the

variables of the patterns t_1 and t_2 do not appear in $\mathcal{A}$ or in $vran(\theta)$. Then the tree for this type derivation will be:

$$[\Lambda]\ \frac{\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \vdash t_1 : \tau'_1 \qquad [\Lambda]\ \dfrac{\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\} \vdash t_2 : \tau'_2}{\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\} \vdash r : \tau'}}{\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \vdash \lambda t_2.r : \tau'_2 \to \tau'}}{\mathcal{A} \vdash \lambda t_1.\lambda t_2.r : \tau'_1 \to \tau'_2 \to \tau'}$$

As variables $\overline{X_n}$ and $\overline{Y_m}$ are all different (the left hand side of the rules is linear), by Theorem 1-b we can add the assumptions over $\overline{Y_m}$ to the derivation of t_1, obtaining $\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\} \vdash t_1 : \tau'_1$. Besides $critVar_{\mathcal{A}}(\lambda t_1.\lambda t_2.r) = \emptyset$, so (A) every variable X_i which appears in r is transparent in the pattern t_1 (resp. Y_j in t_2).

It is a premise that $\mathcal{A} \vdash f\ t_1\theta\ t_2\theta : \tau$, so the type derivation will be:

$$[\mathbf{APP}]\ \frac{[\mathbf{APP}]\ \dfrac{\mathcal{A} \vdash f : \tau_1 \to \tau_2 \to \tau \qquad \mathcal{A} \vdash t_1\theta : \tau_1}{\mathcal{A} \vdash f\ t_1 : \tau_2 \to \tau} \qquad \mathcal{A} \vdash t_2\theta : \tau_2}{\mathcal{A} \vdash f\ t_1\theta\ t_2\theta : \tau}$$

Therefore we know that (B) $\mathcal{A} \vdash t_1\theta : \tau_1$, $\mathcal{A} \vdash t_2\theta : \tau_2$ and $\mathcal{A} \vdash f : \tau_1 \to \tau_2 \to \tau$, being $\tau_1 \to \tau_2 \to \tau$ a generic instance of the type $\mathcal{A}(f)$. Then there will exist a type substitution π such that $\tau'_1\pi = \tau_1$, $\tau'_2\pi = \tau_2$ and $\tau'\pi = \tau$. Moreover, $dom(\pi)$ does not contain any free type variable in $\mathcal{A}$, since π transforms a variant of the type $\mathcal{A}(f)$ into a generic instance of $\mathcal{A}(f)$. Then by Theorem 1-a we have

$$(\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\} \oplus \{\overline{Y_m : \tau''_{2_m}}\})\pi \vdash t_2 : \tau'_2\pi$$
$$(\mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}}\}) \oplus \{\overline{Y_m : \tau''_{2_m}}\})\pi \vdash t_1 : \tau'_1\pi$$

which is equal to

$$(C)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash t_2 : \tau_2$$
$$(C)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\}) \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash t_1 : \tau_1$$

With (A), (B), (C) and by Lemma 1 we can state that for every transparent variable X_i in r then $\mathcal{A} \vdash X_i\theta : \tau''_{1_i}\pi$ (resp. $\mathcal{A} \vdash Y_j\theta : \tau''_{2_i}\pi$).

None of the variables $\overline{X_n}$, $\overline{Y_m}$ appear in $X_i\theta$ or $Y_j\theta$, so by Theorem 1-b we can add these assumptions and obtain

$$(D)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash X_i\theta : \tau''_{1_i}\pi$$
$$(D)\ \mathcal{A} \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash Y_j\theta : \tau''_{2_i}\pi$$

Applying Theorem 1-a to the derivation of r we obtain

$$\mathcal{A}\pi \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash r : \tau'\pi$$

Using (D) and Theorem 1-c we can replace data variables by expressions of the same type:

$$\mathcal{A}\pi \oplus \{\overline{X_n : \tau''_{1_n}\pi}\} \oplus \{\overline{Y_m : \tau''_{2_m}\pi}\} \vdash r\theta : \tau'\pi$$

Since no variable $\overline{X_n}$, $\overline{Y_m}$ appear in $r\theta$ by Theorem 1-b we can remove their assumptions, obtaining a derivation $\mathcal{A}\pi \vdash r\theta : \tau'\pi$. Finally, using the fact that $\mathcal{A}\pi = \mathcal{A}$ and $\tau'\pi = \tau$, this last derivation is equal to $\mathcal{A} \vdash r\theta : \tau$.

(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) Easily by Theorem 1-c.

let_p) The proof of this case is based on the fact that if τ_t is a valid type for a pattern t ($\mathcal{A} \vdash t : \tau_t$) then any generic instance of its generalization $Gen(\tau_t, \mathcal{A}) \succ \tau'$ will be also a valid type for t. Therefore we can replace every occurrence of X in e by t inside the derivation $\mathcal{A} \oplus \{X : Gen(\tau_t, \mathcal{A})\} \vdash e : \tau$ and the resulting derivation of $e[X/t]$ is still correct.

(Flat_p) We will treat the two different cases:

- $\mathcal{P} \vdash let_p\ X = (let_p\ Y = e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3)$.
  It is straightforward to build the derivation of $\mathcal{A} \vdash let_p\ Y = e_1\ in\ (let_p\ X = e_2\ in\ e_3) : \tau$ using the subderivations in:

$$[\mathbf{LET}_p]\ \frac{\begin{array}{c}\mathcal{A}\oplus\{Y:\tau_y\}\vdash Y:\tau_y\\ \mathcal{A}\vdash e_1:\tau_y\\ \mathcal{A}_Y\vdash e_2:\tau_x\end{array}}{\mathcal{A}\vdash let_p\ Y=e_1\ in\ e_2:\tau_x}$$

$$[\mathbf{LET}_p]\frac{\mathcal{A}\oplus\{X:\tau_x\}\vdash X:\tau_x \qquad \mathcal{A}\oplus\{X:Gen(\tau_x,\mathcal{A})\}\vdash e_3:\tau}{\mathcal{A}\vdash let_p\ X=(let_p\ Y=e_1\ in\ e_2)\ in\ e_3:\tau}$$

(being $\mathcal{A}_Y \equiv \mathcal{A}\oplus\{Y:Gen(\tau_y,\mathcal{A})\}$). The only interesting case is the derivation of e_3, but if follows from Theorem 1-d and the fact that $Gen(\tau_x,\mathcal{A}\oplus\{Y:Gen(\tau_y,\mathcal{A})\}) \succ Gen(\tau_x,\mathcal{A})$.

- $\mathcal{P}\vdash let_p\ X=(let_m\ Y=e_1\ in\ e_2)\ in\ e_3 \rightarrow^l let_p\ Y=e_1\ in\ (let_p\ X=e_2\ in\ e_3)$.

  Similar to the previous case, but considering also that $Gen(\tau_y,\mathcal{A})\succ\tau_y$.

(Contx) We have a derivation $\mathcal{A}\vdash\mathcal{C}[e]:\tau$, so according to Remark 2, it will contain 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 Remark 5 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 7 then $\mathcal{A}\vdash\mathcal{C}[e']:\tau$.

□

Appendix A.5. Proof for Lemma 1

Lemma 1
Assume $\mathcal{A}\oplus\{\overline{X_n:\tau_n}\}\vdash t:\tau$, *where* $var(t)\subseteq\{\overline{X_n}\}$. *If* $\mathcal{A}\vdash t[\overline{X_n/t_n}]:\tau$ *and* X_j *is a transparent variable of* t *wrt.* $\mathcal{A}$ *then* $\mathcal{A}\vdash t_j:\tau_j$.

Proof. According to Remark 2, the derivation of $\mathcal{A}\vdash t[\overline{X_n/t_n}]:\tau$ contains derivations for every subpattern t_i, and they have the form $\mathcal{A}\vdash t_i:\tau_i'$ 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 $\overline{\tau_n'}$ as assumptions for the original variables $\overline{X_n}$ we can construct a derivation of $\mathcal{A}\oplus\{\overline{X_n:\tau_n'}\}\vdash t:\tau$, simply replacing the derivations for the subpatterns $\mathcal{A}\vdash t_i:\tau_i'$ with derivations for the variables $\mathcal{A}\oplus\{\overline{X_n:\tau_n'}\}\vdash X_i:\tau_i'$ in the original derivation for $\mathcal{A}\vdash t[\overline{X_n/t_n}]:\tau$. Since X_j is a transparent variable of t wrt. $\mathcal{A}$, by

definition $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ and $ftv(\alpha_j\pi_g) \subseteq ftv(\tau_g)$. By Theorem 5, if any type for t can be derived from $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\}\pi_s$ 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: $\pi^1 \equiv \{\overline{\alpha_n \mapsto \tau_n}\}$ and $\pi^2 \equiv \{\overline{\alpha_n \mapsto \tau'_n}\}$, 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 $\mathcal{A} \oplus \{X_n : \alpha_n\} \Vdash t : \tau_g|\pi_g$ implies $(\mathcal{A} \oplus \{X_n : \alpha_n\})\pi_g \vdash t : \tau_g$, and by Theorem 1-a this implies that $(\mathcal{A} \oplus \{X_n : \alpha_n\})\pi_g\pi \vdash t : \tau_g\pi$; so $\tau_g\pi = \tau$ (the same thing happens with π': $\tau_g\pi' = \tau$). Since X_j is transparent in t, then $ftv(\alpha_j\pi_g) \subseteq ftv(\tau_g)$. As $\tau_g\pi = \tau$ and $\tau_g\pi' = \tau$, we have that for every type variable $\beta \in ftv(\tau_g)$ $\beta\pi = \beta\pi'$. As every type variable β in $ftv(\alpha_j\pi_g)$ is also in $ftv(\tau_g)$ then $\tau_j = (\alpha_j\pi_g)\pi = (\alpha_j\pi_g)\pi' = \tau'_j$. □

Appendix A.6. Proof for Theorem 4

Theorem 4 (Soundness of $\Vdash^?$)
$\mathcal{A} \Vdash^? e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash^? e : \tau$

Proof.
1) $\mathcal{A} \Vdash e : \tau|\pi \Longrightarrow \mathcal{A}\pi \vdash e : \tau$
The proof is similar to the proof of the soundness of algorithm $\mathcal{W}$ in [23]. It follows easily by induction over the size of the type inference $\mathcal{A} \Vdash e : \tau|\pi$ and using Theorem 1-a.
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 $\Vdash$ (point 1 of this Theorem) we have that $\mathcal{A}\pi \vdash e : \tau$. Since $\mathcal{A}\pi \vdash e : \tau$ and $critVar_{\mathcal{A}\pi}(e) = \emptyset$; then, by definition, $\mathcal{A}\pi \vdash^\bullet e : \tau$. □

Appendix A.7. Proof for Theorem 5

In order to prove Theorem 5 we need some auxiliary results. The following remark states that type inference is unique (upon renaming of fresh type variables).

Remark 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 $vran(\pi)$ which do not occur in $ftv(\mathcal{A})$ are fresh variables generated by the inference process, so the result will remain valid if we change those variables by any other fresh types variables.

The following lemma states that the type resulting of generalizing a simple type τ wrt. a set of assumptions $\mathcal{A}$ and then applying a substitution π is always more general than the type that results of generalizing $\tau\pi$ wrt. $\mathcal{A}\pi$.

Lemma 8.
$Gen(\tau, \mathcal{A})\pi \succ Gen(\tau\pi, \mathcal{A}\pi)$

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 $Gen(\tau, \mathcal{A})\pi$ since 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 $Gen(\tau\pi, \mathcal{A}\pi)$. □

Theorem 5 (Completeness of $\Vdash$ wrt. $\vdash$)
If $\mathcal{A}\pi' \vdash e : \tau'$ *then* $\exists \tau, \pi, \pi''$ *such that* $\mathcal{A} \Vdash e : \tau|\pi$, $\mathcal{A}\pi\pi'' = \mathcal{A}\pi'$ *and* $\tau\pi'' = \tau'$.

Proof. This proof proceeds by induction over the size of the type derivation, and it has similarities with the proof of completeness of algorithm $\mathcal{W}$ in [23]. For the sake of conciseness we have shortened or omitted some cases, however the complete proof can be found in [43].

[APP] The type derivation will be:

$$[\mathbf{APP}]\frac{\mathcal{A}\pi' \vdash e_1 : \tau_1' \to \tau' \quad \mathcal{A}\pi' \vdash e_2 : \tau_1'}{\mathcal{A}\pi' \vdash e_1 e_2 : \tau'}$$

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 $\mathcal{A}\pi' = \mathcal{A}\pi_1\pi_1''$. We can write the derivation of e_2 as $(\mathcal{A}\pi_1)\pi_1'' \vdash e_2 : \tau_1'$, so again by the Induction Hypothesis $\mathcal{A}\pi_1 \Vdash e_2 : \tau_2|\pi_2$ and there exists a type substitution π_2'' such that $\tau_2\pi_2'' = \tau_1'$ and $\mathcal{A}\pi_1\pi_1'' = \mathcal{A}\pi_1\pi_2\pi_2''$. We can assume that π_2'' is minimal, so $dom(\pi_2'') \subseteq ftv(\tau_2) \cup ftv(\mathcal{A}\pi_1\pi_2)$. Using

π''_1 and π''_2 and defining $B \equiv dom(\pi''_1) \setminus ftv(\mathcal{A}\pi_1)$ we can define the substitution $\pi_u \equiv \pi''_2 + \pi''_1|_B + [\alpha/\tau']$ for α fresh. It is well defined since the domains of the three substitutions are disjoint—according to Remark 6, the variables in $ftv(\tau_2)$, $dom(\pi_2)$ or $vran(\pi_2)$ which are not in $ftv(\mathcal{A}\pi_1)$ are fresh variables and cannot occur in B. Moreover, π_u is a unifier of $\tau_1\pi_2$ and $\tau_2 \to \alpha$, so there will exist a most general unifier π of $\tau_1\pi_2$ and $\tau_2 \to \alpha$ and the inference $\mathcal{A} \vdash e_1 e_2 : \alpha\pi|\pi_1\pi_2\pi$ is valid. Finally, taking the substitution π'' such that $\pi_u = \pi\pi''$ 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 variables $\overline{X_n}$ in the pattern t do not appear in $\mathcal{A}\pi'$ (nor in $\mathcal{A}$). The derivation will be:

$$[\Lambda]\frac{\begin{array}{c}\mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau'_t \\ \mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash e : \tau'\end{array}}{\mathcal{A}\pi' \vdash \lambda t.e : \tau'_t \to \tau'}$$

Defining $\pi_g \equiv [\overline{\alpha_n/\tau_n}]$—for $\overline{\alpha_n}$ fresh—and using the Induction Hypothesis we have $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t$ where $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t$ and $\tau_t\pi''_t = \tau'_t$ for some π''_t. Also by the Induction Hypothesis we have $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t \Vdash e : \tau_e|\pi_e$ where $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_e\pi''_e$ and $\tau_e\pi''_e = \tau'$ for some π''_e. Therefore we can construct the inference $\mathcal{A} \Vdash \lambda t.e : \tau_t\pi_e \to \tau_e|\pi_t\pi_e$. Finally, with $B \equiv dom(\pi''_t) \setminus ftv((\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t)$ we can define $\pi'' \equiv \pi''_t|_B + \pi''_e$—which is correct as the domains are disjoint—such that $\mathcal{A}\pi_t\pi_e\pi'' = \mathcal{A}\pi'$ and $(\tau_t\pi_e \to \tau_e)\pi'' = \tau'_t \to \tau'$.

[$\mathbf{LET}_m$] We assume that variables $\overline{X_n}$ of the pattern t are fresh and do not occur in $\mathcal{A}\pi'$ (nor in $\mathcal{A}$). The derivation is:

$$[\mathbf{LET}_m]\frac{\begin{array}{c}\mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash t : \tau'_t \\ \mathcal{A}\pi' \vdash e_1 : \tau'_t \\ \mathcal{A}\pi' \oplus \{\overline{X_n : \tau_n}\} \vdash e_2 : \tau'\end{array}}{\mathcal{A}\pi' \vdash let_m\ t = e_1\ in\ e_2 : \tau'}$$

Defining $\pi_g \equiv [\overline{\alpha_n/\tau_n}]$—for $\overline{\alpha_n}$ fresh—by the Induction Hypothesis we have $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t$ where $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi'\pi_g =$

$(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi''_t$ and $\tau_t\pi''_t = \tau'_t$ for some π''_t. Also by the Induction Hypothesis—noticing that $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi''_t$—we have $\mathcal{A}\pi_t \Vdash e_1 : \tau_1|\pi_1$ where $\mathcal{A}\pi_t\pi''_t = \mathcal{A}\pi_t\pi_1\pi''_1$ and $\tau_1\pi''_1 = \tau'_t$ for some π''_1. Taking $B \equiv ftv(\pi''_t) \setminus ftv(\mathcal{A}\pi_t)$, the substitution $\pi_u \equiv \pi''_1 + \pi''_t|_B$–is well defined because the domains are disjoint—unifies $\tau_t\pi_1$ and τ_1, so there will be a most general unifier π such that $\pi_u = \pi\pi_o$. We 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 $\alpha_i\pi_t\pi_1\pi\pi_o = \tau_i$ (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 $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi\pi_o \vdash e_2 : \tau'$, and by the Induction Hypothesis $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2|\pi_2$ and there exists a type substitution π''_2 such that $\tau_2\pi''_2 = \tau'$ and $(\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi\pi_o = (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi\pi_2\pi''_2$. Since the variables $\overline{X_n}$ do not appear in $\mathcal{A}$, in particular it is true that $\mathcal{A}\pi_t\pi_1\pi\pi_o = \mathcal{A}\pi_t\pi_1\pi\pi_2\pi''_2$. Then we can build the type inference

$$[\mathbf{iLET}_m]\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_t|\pi_t \\ \mathcal{A}\pi_t \Vdash e_1 : \tau_1|\pi_1 \\ (\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\})\pi_t\pi_1\pi \Vdash e_2 : \tau_2|\pi_2\end{array}}{\mathcal{A} \Vdash let_m\ t = e_1\ in\ e_2 : \tau_2|\pi_t\pi_1\pi\pi_2}$$

Finally, there is a type substitution $\pi'' \equiv \pi''_2$ such that $\tau_2\pi'' = \tau'$ and $\mathcal{A}\pi' = \mathcal{A}\pi_t\pi_1\pi\pi_2\pi''$.

$[\mathbf{LET}^X_{pm}]$ Easily using the Induction Hypothesis with Lemma 8 and Theorem 1-d.

$[\mathbf{LET}^h_{pm}]$ 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 $[\mathbf{LET}^X_{pm}]$.

□

Appendix A.8. Proof for Theorem 6

In order to prove Theorem 6 we will use a result stating that if there exists a most general substitution which gives type to e wrt. $\vdash$, then the inference $\Vdash$ will succeed, and vice versa. We will use the notion of $\Pi_{\mathcal{A},e}$, which denotes the set collecting all type substitutions π such that $\mathcal{A}\pi$ gives some type to e using $\vdash$.

Lemma 9 (Maximality of $\Vdash$)**.**
$\Pi_{\mathcal{A},e}$ has a maximum element $\pi \iff \exists \tau_g, \pi_g$ such that $\mathcal{A} \Vdash e : \tau_g | \pi_g$.

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 $\mathcal{A}\pi' = \mathcal{A}\pi_g\pi''$. Then $\pi_g|_{ftv(\mathcal{A})}$ is more general than π'. From Theorem 4 we know that $\pi_g|_{ftv(\mathcal{A})}$ is in $\Pi_{\mathcal{A},e}$, so it is the maximum element.

□

The next lemma shows an interesting property of the inference relation $\Vdash$. It states that if $\Vdash^\bullet$ succeeds for an expression e and a set of assumptions $\mathcal{A}$ then the sets of typing substitutions $\Pi_{\mathcal{A},e}$ and $\Pi^\bullet_{\mathcal{A},e}$ are the same. In other words, any type substitution that gives a type to e wrt. $\vdash$ does not produce critical variables and therefore it also gives a type to e wrt. $\vdash^\bullet$.

Lemma 10.
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 $\mathcal{A} \Vdash e : \tau | \pi$. 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 and the premise we have $\mathcal{A}\pi \vdash^\bullet e : \tau$, 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 the definition of $\Pi^\bullet_{\mathcal{A},e}$

□

Theorem 6 (Maximality of $\Vdash^\bullet$)

a) $\Pi^\bullet_{\mathcal{A},e}$ has a maximum element $\iff \exists \tau_g, \pi_g$ such that $\mathcal{A} \Vdash^\bullet e : \tau_g | \pi_g$.
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 10 $\Pi_{\mathcal{A},e} = \Pi^{\bullet}_{\mathcal{A},e}$. Since $\mathcal{A} \Vdash e : \tau_g|\pi_g$ (by definition of $\Vdash^{\bullet}$) by Lemma 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 $\mathcal{A} \not\Vdash e : \tau_g|\pi_g$. Then by Theorem 5 there cannot exist 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 does not contain a 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 distinction over the cause of the existence of 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 the cause of the opacity are type variables appearing in $\mathcal{A}\pi_g$—i.e., $\mathcal{A} \oplus \{X_m : \alpha_m\} \Vdash t_j : \tau_j|\pi_j$ and $ftv(\alpha_i\pi_j) \not\subseteq ftv(\tau_j)$ and $ftv(\alpha_i\pi_j) \setminus ftv(\tau_j) \subseteq ftv(\mathcal{A}\pi_g)$. Let $\overline{\beta_k}$ be all the type variables causing opacity, and τ^1 and τ^2 two non unifiable ground types (*bool* and *char*, for example). Defining $\pi_1 \equiv [\overline{\beta_k/\tau^1}]$ and $\pi_2 \equiv [\overline{\beta_k/\tau^2}]$ we have—by Theorem 4 and 1-a—$\mathcal{A}\pi_g\pi_1 \vdash e : \tau_g\pi_1$ and $\mathcal{A}\pi_g\pi_2 \vdash e : \tau_g\pi_2$. Clearly $critVar_{\mathcal{A}\pi_g\pi_1}(e) = \emptyset$ and $critVar_{\mathcal{A}\pi_g\pi_2}(e) = \emptyset$—because all the type variables have been replaced by ground types—so $\pi_g\pi_1, \pi_g\pi_2 \in \Pi^{\bullet}_{\mathcal{A},e}$. Since τ^1 and τ^2 are not unifiable, the only substitution more general than $\pi_g\pi_1$ 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 $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$.

(B.2) $critVar_{\mathcal{A}\pi_g}(e) \neq \emptyset$ because there exists some pattern t_j in e in which there is some 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 must have appeared because of the presence of a symbol in t_j whose type is a type-scheme, and those 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 there exists a type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$ and $\tau' = \tau\pi''$. □

Appendix A.9. Proof for Theorem 7

Theorem 7 (Soundness of $\mathcal{B}$)
If $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then $wt_{\mathcal{A}\pi}(\mathcal{P})$.

Proof. Easy, because as $\mathcal{B}(\mathcal{A}, \mathcal{P}) = \pi$ then by Theorem 4 we have $\mathcal{A}\pi \vdash^{\bullet} (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau_1, \ldots, \tau_m)$. Because of the type of *pair* and point 2 of $\mathcal{B}$, it is clear that each τ_i is a variant of $\mathcal{A}\pi(f_i)$.

□

Appendix A.10. Proof for Theorem 8

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 $\mathcal{A}\pi'(f_i)$, so $\mathcal{A}\pi' \vdash^{\bullet} f_i : \tau'_i$. With these derivations we can build $\mathcal{A}\pi' \vdash^{\bullet} (\varphi(R_1), \ldots, \varphi(R_m)) : (\tau'_1, \ldots, \tau'_m)$. 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 must exist some type substitution π'' such that $\mathcal{A}\pi' = \mathcal{A}\pi\pi''$.

□

Appendix A.11. Proof for Theorem 9

In order to prove Theorem 9 we need to state some auxiliary remarks and lemmas:

Remark 7. If $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau_g|\pi_g$ such that $\mathcal{A}\pi_g = \mathcal{A}$ and $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_g\pi}\} \vdash^\circ t : \tau_g\pi$ for some π such that $dom(\pi) \subseteq ftv(\tau_g)$, then $\mathcal{A}\pi = \mathcal{A}$.

Explanation. If some variable $\beta \in ftv(\mathcal{A})$ appears in $ftv(\tau_g)$, then some application inside t contains a symbol whose assumption in $\mathcal{A}$ contains β free. If we use π to replace β in the assumptions for data variables but we leave it unchanged in $\mathcal{A}$, then that application inside t will be ill-typed. □

Remark 8. If we have $\mathcal{A}\oplus\{\overline{X_n : \alpha_n}\} \Vdash t_1 : \tau_{g_1}|\pi_{g_1}$ and $\mathcal{A}\oplus\{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\}\oplus \{\overline{Y_m : \beta_m}\} \Vdash t_2 : \tau_{g_2}|\pi_{g_2}$, then we can assume that no type variable γ in $\bigcup_{i=1}^n ftv(\alpha_i\pi_{g_1})$ such that $\mathcal{A} \notin ftv(\mathcal{A})$ can occur in $\bigcup_{i=1}^m ftv(\beta_i\pi_{g_2})$, and vice versa.

Explanation. Intuitively, all type variables involved in a type inference which do not occur in $ftv(\mathcal{A})$ are fresh. □

The following five results are easy consequences of Remark 1, Theorem 5 a), b), c) and Theorem 1 resp. since they are restricted to expressions $e^\circ \in Exp^\circ$ not containing λ-abstractions. For these expressions, the typing rules for $\vdash^\circ$ and $\vdash$ are the same, so $\mathcal{A} \vdash^\circ e^\circ : \tau$ implies $\mathcal{A} \vdash e^\circ : \tau$.

Remark 9.
If $\mathcal{A} \oplus \{\overline{X_n : \tau_n}\} \vdash^\circ t : \tau$ then we can assume that $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t : \tau'|\pi$ such that $\mathcal{A}\pi = \mathcal{A}$.

Lemma 11 (Closure under type substitutions in $\vdash^\circ$ derivations)**.** *Let $e^\circ \in Exp^\circ$ be an expression not containing λ-abstractions and $\mathcal{A} \vdash^\circ e^\circ : \tau$. Then $\mathcal{A}\pi \vdash^\circ e^\circ : \tau\pi$, for any $\pi \in TSub$.*

Lemma 12 (Adding and removing assumptions in $\vdash^\circ$ derivations)**.** *Let $e^\circ \in Exp^\circ$ be an expression not containing λ-abstractions and s be a symbol not occurring in e. Then $\mathcal{A} \vdash^\circ e^\circ : \tau \iff \mathcal{A} \oplus \{s : \sigma_s\} \vdash^\circ e^\circ : \tau$.*

Lemma 13 (Replacing variables by patterns in $\vdash^\circ$ derivations)**.** *Let $e^\circ \in Exp^\circ$ be an expression not containing λ-abstractions, $\mathcal{A} \oplus \{X : \tau_t\} \vdash^\circ e^\circ : \tau$ and $\mathcal{A} \oplus \{X : \tau_t\} \vdash^\circ t : \tau_t$. Then $\mathcal{A} \oplus \{X : \tau_t\} \vdash^\circ e^\circ[X/t] : \tau$.*

Lemma 14 (Completeness of $\Vdash$ wrt. $\vdash^\circ$ for patterns)**.** *If $\mathcal{A}\pi' \vdash t : \tau'$ then $\exists\tau, \pi, \pi''$ such that $\mathcal{A} \Vdash t : \tau|\pi$, $\mathcal{A}\pi\pi'' = \mathcal{A}\pi'$ and $\tau\pi'' = \tau'$.*

Using the previous results, we can now prove Theorem 9:

Theorem 9
If $wt^\circ_{\mathcal{A}}(\mathcal{P})$, $\mathcal{A} \vdash^\circ e^\circ_1 : \tau$ *and* $e^\circ_1 \to^l e^\circ_2$, *then* $\mathcal{A} \vdash^\circ e^\circ_2 : \tau$.

Proof. By case distinction over the rule of the let-rewriting rule (Figure 9, page 16) applied. The proof for the cases for (LetIn), (Bind), (Elim), (Flat$_m$), (LetAp) and (Contx) is the same as the proof for Theorem 3. Therefore we only have to prove the (Fapp) case.

We consider a function which accepts two arguments, but the proof for any other number of arguments follows the same ideas. If we reduce an expression e°_1 using the rule (Fapp) then the step is $e^\circ_1 \equiv f\ t_1\theta\ t_2\theta \to^l r\theta$, where $(f\ t_1\ t_2 \to r) \in \mathcal{P}$ and $\theta \in PSub$. Since $wt^\circ_{\mathcal{A}}(\mathcal{P})$ then $\mathcal{A} \vdash^\circ \lambda t_1.\lambda t_2.r : \tau_{g_1}\pi_1 \to \tau_{g_2}\pi_2 \to \tau_r$ and $\tau_{g_1}\pi_1 \to \tau_{g_2}\pi_2 \to \tau_r$ is a variant of $\mathcal{A}(f)$. We assume that data variables in the patterns t_1 and t_2 do not appear in $\mathcal{A}$ or in $vran(\theta)$—otherwise we could rename the variables appearing in the rule. Then the type derivation of this λ-abstraction is:

$$(\Lambda^\circ)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \vdash^\circ t_1 : \tau_{g_1}\pi_1 \\ \mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \vdash^\circ \lambda t_2.r : \tau_{g_2}\pi_2 \to \tau_r\end{array}}{\mathcal{A} \vdash^\circ \lambda t_1.\lambda t_2.r : \tau_{g_1}\pi_1 \to \tau_{g_2}\pi_2 \to \tau_r}$$

where the derivation $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \vdash^\circ \lambda t_2.r : \tau_{g_2}\pi_2 \to \tau_r$ is:

$$(\Lambda^\circ)\frac{\begin{array}{c}\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \oplus \{\overline{Y_m : \beta_m\pi_{g_2}\pi_2}\} \vdash^\circ t_2 : \tau_{g_2}\pi_2 \\ \mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \oplus \{\overline{Y_m : \beta_m\pi_{g_2}\pi_2}\} \vdash^\circ r : \tau_r\end{array}}{\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \vdash^\circ \lambda t_2.r : \tau_{g_2}\pi_2 \to \tau_r}$$

For this derivation the following conditions hold:

(a_1) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n}\} \Vdash t_1 : \tau_{g_1}|\pi_{g_1}$

(b_1) $dom(\pi_1) \subseteq ftv(\tau_{g_1})$

(c_1) $vran(\pi_1) \cap (\bigcup_{i=1}^n ftv(\alpha_i\pi_{g_1}) \setminus ftv(\tau_{g_1})) = \emptyset$

(d_1) $ftv(\tau_{g_2}\pi_2 \to \tau_r) \cap (\bigcup_{i=1}^n ftv(\alpha_i\pi_{g_1}) \setminus ftv(\tau_{g_1})) = \emptyset$

(a_2) $\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \oplus \{\overline{Y_m : \beta_m}\} \Vdash t_2 : \tau_{g_2}|\pi_{g_2}$

(b_2) $dom(\pi_2) \subseteq ftv(\tau_{g_2})$

(c_2) $vran(\pi_2) \cap (\bigcup_{i=1}^{m} ftv(\beta_i \pi_{g_2}) \setminus ftv(\tau_{g_2})) = \emptyset$

(d_2) $ftv(\tau_r) \cap (\bigcup_{i=1}^{m} ftv(\beta_i \pi_{g_2}) \setminus ftv(\tau_{g_2})) = \emptyset$

Since $\tau_{g_1}\pi_1 \to \tau_{g_2}\pi_2 \to \tau_r$ is a variant of $\mathcal{A}(f)$ then any generic instance of $\mathcal{A}(f)$ will be of the form $\tau_{g_1}\pi_1\pi' \to \tau_{g_2}\pi_2\pi' \to \tau_r\pi'$, where $dom(\pi') \cap ftv(\mathcal{A}) = \emptyset$. Therefore the type derivation of $f\ t_1\theta\ t_2\theta$ will be:

$$(\text{APP}^\circ)\dfrac{\dfrac{\mathcal{A} \vdash^\circ f : \tau_{g_1}\pi_1\pi' \to \tau_{g_2}\pi_2\pi' \to \tau_r\pi' \quad \mathcal{A} \vdash^\circ t_1\theta : \tau_{g_1}\pi_1\pi'}{\mathcal{A} \vdash^\circ f\ t_1\theta : \tau_{g_2}\pi_2\pi' \to \tau_r\pi'} \quad \mathcal{A} \vdash^\circ t_2\theta : \tau_{g_2}\pi_2\pi'}{\mathcal{A} \vdash^\circ f\ t_1\theta\ t_2\theta : \tau_r\pi'}$$

In the type derivations of $t_1\theta$ we can replace the expressions introduced by θ by the original data variables provided we add suitable assumptions to the set of assumptions. If the sub-derivation $\mathcal{A} \vdash^\circ X_i\theta : \tau_{1_i}$ appears in the type derivation of $t_1\theta$, then adding the assumptions $\{\overline{X_n : \tau_{1_n}}\}$ to $\mathcal{A}$ we have the derivation

$$(A)\ \mathcal{A} \oplus \{\overline{X_n : \tau_{1_n}}\} \vdash^\circ t_1 : \tau_{g_1}\pi_1\pi'$$

The same can be done with the type derivation of $t_2\theta$, provided that the sub-derivations $\mathcal{A} \vdash^\circ Y_i\theta : \tau_{2_i}$ appears in the type derivation of $t_2\theta$:

$$(B)\ \mathcal{A} \oplus \{\overline{Y_m : \tau_{2_m}}\} \vdash^\circ t_2 : \tau_{g_2}\pi_2\pi'$$

Notice that since program rules are linear then every data variable will appear only once in each pattern, and they will be different among different patterns. Therefore only one sub-derivation of $X_i\theta$ or $Y_i\theta$ will appear in the complete derivation for $t_1\theta$ and $t_2\theta$, so the types τ_{1_i} and τ_{2_i} are well determined.

From (A) and (a_1) and by the completeness of $\Vdash$ stated in Theorem 14 we have that there exists some $\pi_1'' \in TSub$ such that:

(C_1) $\tau_{g_1}\pi_1'' = \tau_{g_1}\pi_1\pi'$

(C_2) $\alpha_i\pi_{g_1}\pi_1'' = \tau_{1_i}$ for every $i \in [1..n]$

(C_3) $\mathcal{A}\pi_{g_1}\pi_1'' = \mathcal{A}$

Since is it possible to derive a type for t_1 without applying any substitution to $\mathcal{A}$—see the type derivation of $\lambda t_1.\lambda t_2.r$—then by Remark 9 we can assume that $\mathcal{A}\pi_{g_1} = \mathcal{A}$, so by ($C_3$) it follows $\mathcal{A}\pi_1'' = \mathcal{A}$ (C'$_3$).

The data variables $\overline{X_n}$ do not occur in t_2, so by Theorem 12 we can add any assumption τ_i' for them in the derivation (B), obtaining $\mathcal{A} \oplus \{\overline{X_n : \tau_n'}\} \oplus \{\overline{Y_m : \tau_{2_m}}\} \vdash^\circ t_2 : \tau_{g_2}\pi_2\pi'$. As before we know from (a_2), the previous derivation and the completeness of $\Vdash$ stated in Theorem 14 that there exists $\pi_2'' \in \mathit{TSub}$ such that:

(D_1) $\tau_{g_2}\pi_2'' = \tau_{g_2}\pi_2\pi'$

(D_2) $\beta_i\pi_{g_2}\pi_2'' = \tau_{2_i}$ for every $i \in [1..m]$

(D_3) $\mathcal{A}\pi_{g_2}\pi_2'' = \mathcal{A}$

As before, by Remark 9 we can assume that $\mathcal{A}\pi_{g_2} = \mathcal{A}$, so by ($D_3$) it follows $\mathcal{A}\pi_2'' = \mathcal{A}$ (D'$_3$).

Using π', π_1'' and π_2'' we build the substitution π^0 as follows:

$$\pi^0(\alpha) = \begin{cases} \pi'(\alpha) & \text{if } \alpha \notin \mathit{ftv}(\mathcal{A}) \wedge (\alpha \in \mathit{ftv}(\tau_r) \vee \alpha \in \mathit{vran}(\pi_1) \vee \\ & \quad \alpha \in \mathit{vran}(\pi_2) \vee \alpha \in \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_{g_1}) \cap \mathit{ftv}(\tau_{g_1}) \vee \\ & \quad \alpha \in \bigcup_{i=1}^m \mathit{ftv}(\beta_i\pi_{g_2}) \cap \mathit{ftv}(\tau_{g_2})) \\ \pi_1''(\alpha) & \text{if } \alpha \notin \mathit{ftv}(\mathcal{A}) \wedge \alpha \in \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_{g_1}) \setminus \mathit{ftv}(\tau_{g_1}) \\ \pi_2''(\alpha) & \text{if } \alpha \notin \mathit{ftv}(\mathcal{A}) \wedge \alpha \in \bigcup_{i=1}^m \mathit{ftv}(\beta_i\pi_{g_2}) \setminus \mathit{ftv}(\tau_{g_2}) \end{cases}$$

This substitution is well defined because its conditions are exclusive:

- The first and second conditions are exclusive. Let γ be a variable such that $\gamma \notin \mathit{ftv}(\mathcal{A})$:
  - If $\gamma \in \mathit{ftv}(\tau_r)$ then $\gamma \notin \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_g) \setminus \mathit{ftv}(\tau_{g_1})$ by (d_1).
  - If $\gamma \in \mathit{vran}(\pi_1)$ then $\gamma \notin \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_g) \setminus \mathit{ftv}(\tau_{g_1})$ by (c_1).
  - If $\gamma \in \mathit{vran}(\pi_2)$ then $\gamma \notin \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_g) \setminus \mathit{ftv}(\tau_{g_1})$ by (d_1), because as (b_2) then $\mathit{vran}(\pi_2) \subseteq \mathit{ftv}(\tau_{g_2}\pi_2 \to \tau_r)$.
  - If $\gamma \in \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_{g_1}) \cap \mathit{ftv}(\tau_{g_1})$ then $\gamma \notin \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_{g_1}) \setminus \mathit{ftv}(\tau_{g_1})$.
  - If $\gamma \in \bigcup_{i=1}^m \mathit{ftv}(\beta_i\pi_{g_2}) \cap \mathit{ftv}(\tau_{g_2})$ then $\gamma \notin \bigcup_{i=1}^n \mathit{ftv}(\alpha_i\pi_g) \setminus \mathit{ftv}(\tau_{g_1})$ by Remark 8.
- The first and third conditions are exclusive. Let γ be a variable such that $\gamma \notin \mathit{ftv}(\mathcal{A})$:

- If $\gamma \in ftv(\tau_r)$ then $\gamma \notin \bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2}) \setminus ftv(\tau_{g_2})$ by (d_2).
- If $\gamma \in vran(\pi_1)$ then $\gamma \notin \bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2}) \setminus ftv(\tau_{g_2})$. Since no data variable $\overline{X_n}$ is used in the type inference (a_2), we can assume that all type variables in $\bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2})$ which do not occur in $ftv(\mathcal{A})$ will be fresh type variables generated during type inference, so they will not appear in $vran(\pi_1)$.
- If $\gamma \in vran(\pi_2)$ then $\gamma \notin \bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2}) \setminus ftv(\tau_{g_2})$ by (c_2).
- If $\gamma \in \bigcup_{i=1}^{n} ftv(\alpha_i\pi_{g_1}) \cap ftv(\tau_{g_1})$ then $\gamma \notin \bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2}) \setminus ftv(\tau_{g_2})$ by Remark 8.
- If $\gamma \in \bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2}) \cap ftv(\tau_{g_2})$ then $\gamma \notin \bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2}) \setminus ftv(\tau_{g_2})$.

- The second and third conditions are exclusive. Let γ be a variable such that $\gamma \notin ftv(\mathcal{A})$. Then by Remark 8 we have that if $\gamma \in \bigcup_{i=1}^{n} ftv(\alpha_i\pi_{g_1}) \setminus ftv(\tau_{g_1})$ then $\gamma \notin \bigcup_{i=1}^{m} ftv(\beta_i\pi_{g_2}) \setminus ftv(\tau_{g_2})$.

Using this substitution π^0 we have that:

(E_1) $\mathcal{A}\pi^0 = \mathcal{A}$ (by definition of π^0).

(E_2) $\tau_r\pi^0 = \tau_r\pi'$ (by definition of π^0).

(E_3) $\alpha_i\pi_{g_1}\pi_1\pi^0 = \alpha_i\pi_{g_1}\pi_1''$, for every $i \in [1..n]$

(E_4) $\beta_i\pi_{g_2}\pi_2\pi^0 = \beta_i\pi_{g_2}\pi_2''$, for every $i \in [1..m]$

To prove (E_3), let us consider $\gamma \in ftv(\alpha_i\pi_{g_1})$ and distinguish two cases:

- If $\gamma \in ftv(\mathcal{A})$: by Remark 9 we can assume that $\mathcal{A}\pi_{g_1} = \mathcal{A}$, so by Remark 7 we have $\mathcal{A}\pi_1 = \mathcal{A}$. Then $\gamma\pi_1\pi^0 = \gamma\pi^0$. Since $\gamma \in ftv(\mathcal{A})$, by definition of π^0 we have $\gamma\pi^0 = \gamma$. On the other hand, $\gamma\pi_1'' = \gamma$ by (C'_3). Therefore $\gamma\pi_1\pi^0 = \gamma\pi^0 = \gamma = \gamma\pi_1''$.

- If $\gamma \notin ftv(\mathcal{A})$:
  - If $\gamma \in ftv(\tau_{g_1})$:
    - If $\gamma \in dom(\pi_1)$ then for all $\delta \in ftv(\gamma\pi_1)$ we have $\delta\pi^0 = \delta\pi'$— because $\delta \in vran(\pi_1)$. Therefore $\gamma\pi_1\pi^0 = \gamma\pi_1\pi'$, and since $\gamma \in \tau_{g_1}$ then by (C_1) $\gamma\pi_1\pi' = \gamma\pi_1''$.

* If $\gamma \notin dom(\pi_1)$ then $\gamma\pi_1\pi^0 = \gamma\pi^0 = \gamma\pi'$—because $\gamma \in \bigcup_{i=1}^{n} ftv(\alpha_i\pi_{g_1}) \cap ftv(\tau_{g_1})$. Since $\gamma \notin dom(\pi_1)$ then $\gamma\pi' = \gamma\pi_1\pi' = \gamma\pi_1''$ by (C_1) as before.

– If $\gamma \notin ftv(\tau_{g_1})$ then by (b_1) $\gamma \notin dom(\pi_1)$. Therefore $\gamma\pi_1\pi^0 = \gamma\pi^0 = \gamma\pi_1''$—because $\gamma \in \bigcup_{i=1}^{n} ftv(\alpha_i\pi_{g_1}) \setminus ftv(\tau_{g_1})$.

The proof of (E_4) follows the same steps as the proof of (E_3). Then by (C_2) and (E_3) we know that (F_1) $\alpha_i\pi_{g_1}\pi_1\pi^0 = \tau_{1_i}$ (for every $i \in [1..n]$). Similarly, by (D_2) and (E_4) we have (F_2) $\beta_i\pi_{g_2}\pi_2\pi^0 = \tau_{2_i}$ (for every $i \in [1..m]$). Applying the closure of type derivations stated in Theorem 11—notice that r will not contain any λ-abstraction—to the type derivation

$$\mathcal{A} \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1}\} \oplus \{\overline{Y_m : \beta_m\pi_{g_2}\pi_2}\} \vdash^\circ r : \tau_r$$

which comes from the derivation of $\lambda t_1.\lambda t_2.r$ then we obtain:

$$\mathcal{A}\pi^0 \oplus \{\overline{X_n : \alpha_n\pi_{g_1}\pi_1\pi^0}\} \oplus \{\overline{Y_m : \beta_m\pi_{g_2}\pi_2\pi^0}\} \vdash^\circ r : \tau_r\pi^0$$

Using (E_1), (E_2), (F_1) and (F_2), this is the same as:

$$(G)\ \mathcal{A} \oplus \{\overline{X_n : \tau_{1_n}}\} \oplus \{\overline{Y_m : \tau_{2_m}}\} \vdash^\circ r : \tau_r\pi'$$

We had that $\mathcal{A} \vdash^\circ X_i\theta : \tau_{1_i}$ (for every $i \in [1..n]$) and $\mathcal{A} \vdash^\circ Y_i\theta : \tau_{2_i}$ (for every $i \in [1..m]$). Since the patterns are linear and the data variables $\overline{X_n}$, $\overline{Y_m}$ do not appear in $vran(\theta)$ then by Theorem 12 we can add assumptions to the data variables, obtaining

$$(H_1)\ \mathcal{A} \oplus \{\overline{X_n : \tau_{1_n}}\} \oplus \{\overline{Y_m : \tau_{2_m}}\} \vdash^\circ X_i\theta : \tau_{1_i} \quad \text{(for every } i \in [1..n])$$

$$(H_2)\ \mathcal{A} \oplus \{\overline{X_n : \tau_{1_n}}\} \oplus \{\overline{Y_m : \tau_{2_m}}\} \vdash^\circ Y_i\theta : \tau_{2_i} \quad \text{(for every } i \in [1..m])$$

Applying repeatedly Theorem 13 with (G), (H_1) and (H_2) we have

$$(I)\ \mathcal{A} \oplus \{\overline{X_n : \tau_{1_n}}\} \oplus \{\overline{Y_m : \tau_{2_m}}\} \vdash^\circ r\theta : \tau_r\pi'$$

Finally, the expression $r\theta$ will not contain the variables $\overline{X_n}$, $\overline{Y_m}$ because they do not occur in $vran(\theta)$, so applying Theorem 12 to (I) we obtain:

$$\mathcal{A} \vdash^\circ r\theta : \tau_r\pi'$$

□