UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

ANÁLISIS DE PROPIEDADES DE SEGURIDAD Y CONSUMO ACOTADO DE MEMORIA EN UN LENGUAJE FUNCIONAL SIN RECOLECCIÓN DE BASURA

MEMORIA PARA OPTAR AL GRADO DE DOCTOR
PRESENTADA POR

Manuel Montenegro Montes

Bajo la dirección de los doctores

Ricardo Peña Marí
Clara María Segura Díaz

Madrid, 2012

Análisis de propiedades de seguridad y consumo acotado de memoria en un lenguaje funcional sin recolección de basura

Tesis doctoral

Autor: **Manuel Montenegro Montes**

Directores: **Ricardo Peña Marí y Clara María Segura Díaz**

Facultad de Informática

Universidad Complutense de Madrid

Septiembre de 2011

Resumen

La mayoría de lenguajes funcionales abstraen al programador del manejo de la memoria. Suelen disponer de un *recolector de basura* encargado de determinar, en tiempo de ejecución, qué partes de memoria ya no se necesitan y pueden ser liberadas. La principal ventaja de este enfoque es que los programadores no tienen que molestarse con asuntos relativos a la memoria. Sin embargo, este enfoque puramente implícito dificulta la tarea de predecir en tiempo de compilación los tiempos de vida en ejecución de las estructuras de datos. Por el contrario, otros lenguajes delegan completamente el manejo de la memoria en el programador. De este modo, se conoce en tiempo de compilación cuándo se liberan las estructuras de datos. No obstante, este enfoque es proclive a errores, ya que el programador puede, por error, tratar de acceder a zonas de memoria que han sido liberadas (*punteros descolgados*), debido a comparticiones inesperadas entre las estructuras de datos.

Safe es un lenguaje funcional de primer orden que proporciona un enfoque semiexplícito al manejo de la memoria mediante dos mecanismos: regiones (inferidas por el compilador) y ajuste de patrones destructivo (controlado por el programador). De este modo se pretende prescindir de un recolector de basura para liberar las estructuras de datos que ya no se utilizan y, con ello, aumentar la predictibilidad en el uso de memoria por parte de los programas. Esto permitirá diseñar un análisis estático para inferir cotas superiores a la cantidad de memoria que un programa necesita para su ejecución.

La incorporación de mecanismos explícitos de destrucción de memoria en un programa implica el riesgo de acceder a memoria ya liberada. El primer objetivo de la tesis es el desarrollo de un análisis para garantizar la seguridad de un programa con respecto a los accesos a memoria (en particular, la ausencia de punteros descolgados). Una vez que el compilador comprueba esta propiedad de seguridad para un programa concreto, el segundo objetivo es la formalización de dicha propiedad en forma de certificado, de modo que pueda ser comprobada automáticamente por el consumidor de dicho programa. Finalmente, se desarrollará un análisis basado en técnicas de interpretación abstracta para inferir cotas superiores al consumo de memoria de programas. Puesto que la meta final del proyecto *Safe* es la certificación de las propiedades de seguridad y consumo acotado de memoria, en esta tesis se ha puesto especial énfasis en la corrección formal de cada uno de los análisis expuestos.

Agradecimientos

En primer lugar, y como no podría ser de otra forma, me gustaría agradecer a mis directores, Ricardo Peña, y Clara Segura, por su enorme esfuerzo e implicación en este trabajo, y por la gran cantidad de cosas que he aprendido con ellos a lo largo de estos últimos seis años.

Thanks to the members of the Institute for Computing and Information Sciences of Radboud University Nijmegen, especially Marko van Eekelen, Olha Shkaravska, Alejandro Tamalet, and Rody Kersten, for their warm hospitality during my visit to Nijmegen, for the fruitful work we started there, and for the fruitful work that is (hopefully) still to come. Double ration of acknowledgements to Olha Shkaravska, because of her detailed comments on the first draft of this work.

También quiero dar gracias a mi hermana Estela, porque ya se está convirtiendo en la reprógrafa oficial de mis trabajos (y el soporte oficial de mis neuras por entregarlos yo siempre a última hora), y a David, por su ayuda para poder imprimir este último.

Gracias también a mis compañeros del despacho 220, que luego pasó a ser 219, por su grata compañía durante todo este tiempo. También a los del despacho 220, que luego siguió siendo 220, por la misma razón. Y a Gabi, y a Lidia, de nuevo, por la misma razón.

Por último, y no menos importante, gracias a mi familia, por estar siempre donde están, y a Ger, Javi, Miguel, Roberto, Nacho, Juanan, e Iván por sus ánimos para inducirme a terminar este mamotreto que ud. tiene en sus manos.

Las publicaciones asociadas a este trabajo han sido llevadas a cabo como parte de los proyectos TIN2008-06622-C03-01 (FAST-STAMP, financiado por el Ministerio de Ciencia e Innovación), S2009/TIC-1465 (PROMETIDOS, financiado por la Comunidad de Madrid), S-0505/TIC-0407 (PROMESAS, financiado por la Comunidad de Madrid), TIN2004-07943-C04 (SELF, financiado por el Ministerio de Ciencia y Tecnología), y la beca FPU AP2006-02154, del Ministerio de Educación.

Introducción

En esta tesis se describe el análisis estático de propiedades de seguridad de punteros y de existencia de cotas superiores de consumo de memoria para un lenguaje funcional, llamado *Safe*. Este lenguaje se ha desarrollado en los últimos años como plataforma de investigación para el análisis y certificación formal de propiedades de programas relacionadas con el uso de la memoria. Se introdujo para investigar la viabilidad del paradigma funcional para la programación de microcontroladores y sistemas empotrados con requisitos de memoria estrictos. En esta introducción motivaremos la existencia del lenguaje *Safe* y se enumerarán los objetivos de esta tesis.

Motivación

Los sistemas informáticos no son completamente fiables. Tan sólo hacen lo que un programador les dice que hagan, pero, a veces, los programadores cometen errores. A medida que el software desempeña un papel cada vez más relevante en los sistemas informáticos, su fiabilidad adquiere mayor importancia, especialmente en aquellos casos en los que un mal funcionamiento puede desembocar en graves consecuencias económicas, o en pérdida de vidas humanas. El amplio área de investigación de los *métodos formales* se encarga del desarrollo de técnicas matemáticamente rigurosas para la especificación de propiedades deseables en un programa, así como de su verificación. Entre este conjunto de propiedades deseables destacan aquellas relacionadas con su corrección, que aseguran que un programa realmente hace lo que se espera que haga. Esta clase de propiedades es conocida comúnmente como *propiedades funcionales*.

Independientemente de la corrección de un programa, existen otras propiedades deseables que resultan de importancia en la seguridad de los sistemas software. Son las *propiedades no funcionales.* Un ejemplo de este tipo de propiedades es el hecho de que un programa termine su trabajo antes de una determinada cantidad de tiempo, o que sus necesidades de memoria no superen un límite establecido. Estos dos ejemplos se enmarcan en un área de investigación más amplio, llamado *análisis de recursos*. En este contexto, un programa es considerado como un consumidor de recursos (por el término *recurso* podemos entender tiempo, memoria, energía, etc.) y el objetivo consiste en calcular una cota superior de los recursos consumidos durante cualquier ejecución posible del programa. Esta tesis se centra en el análisis de *cotas de memoria*. El consumo de memoria adquiere bastante importancia en varios contextos. Por ejemplo, en el ámbito de la programación de un dispositivo empotrado es necesario asegurar que los programas ejecutados en dicho dispositivo no interrumpen su funcionamiento debido a que intentan usar más memoria de la disponible en el sistema. También resulta útil conocer con antelación cuánta memoria necesitará el programa en el caso peor, con el objetivo de reducir costes de hardware y energía.

Creemos que la elección de un paradigma de programación adecuado adquiere gran relevancia en

la fiabilidad de los sistemas software: los lenguajes más amigables (desde el punto de vista del programador) son aquellos que incluyen conceptos del problema que se ha de resolver, en lugar de conceptos de la máquina donde se intenta resolver el problema. El *paradigma declarativo* se encuentra dentro de esta categoría, ya que está orientado a la especificación de los cómputos, en lugar de cómo se llevan a cabo dichos cómputos. Los programas funcionales están basados en el concepto matemático de función, definida como un conjunto de ecuaciones. El paradigma funcional abstrae al programador del modo en el que estas funciones son evaluadas. La principal ventaja de la programación funcional pura es la ausencia de efectos laterales, que redundan en una deseable propiedad conocida como *transparencia referencial*: una función se define exclusivamente mediante una correspondencia entre su entrada y su salida (dicho de otro modo, el resultado de una función sólo depende de sus parámetros de entrada). Esto implica que, en un lenguaje funcional puro, una expresión siempre se evalúa al mismo valor, independientemente del contexto en el que dicha evaluación se produce.

La ausencia de una noción de estado facilita la tarea de razonar sobre las propiedades funcionales de un programa. Por otra parte, los lenguajes funcionales son, en general, más adecuados para su análisis estático, debido a esta ausencia de estado. No obstante, la inferencia de cotas de memoria requiere cierta cautela. En la mayoría de lenguajes funcionales, el manejo de memoria queda delegado al sistema de ejecución, que reserva memoria a medida que va siendo necesitada por el programa, siempre que haya suficiente espacio de memoria disponible. Existe un *recolector de basura* encargado de determinar, en tiempo de ejecución, qué partes de memoria ya no se necesitan y pueden ser desechadas. La principal ventaja de este enfoque es que los programadores no tienen que molestarse con asuntos relativos a la memoria (considerados, a menudo, de bajo nivel). Sin embargo, también hay desventajas. El retraso introducido por la recolección de basura puede impedir que un programa devuelva una respuesta en un determinado tiempo, lo cual resulta inaceptable en sistemas de tiempo real. Por otra parte, la recolección de basura dificulta la tarea de predecir en tiempo de compilación los tiempos de vida de las estructuras de datos, especialmente en los casos donde el sistema de ejecución no especifica las condiciones bajo las que se activa un recolector de basura. Resulta deseable disponer de un enfoque de manejo de memoria alternativo a la recolección de basura con el objetivo de calcular cotas de consumo de memoria en un lenguaje funcional. En las siguientes secciones se explorarán varias posibilidades.

Gestión de memoria basada en regiones

Bajo este tipo de gestión de memoria cada objeto se crea en un región. Las regiones se crean y se destruyen como si estuviesen dispuestas en una pila: la última región creada es la primera en ser destruida. La creación y destrucción de regiones se lleva a cabo en tiempo constante. El compilador ha de determinar: (1) cuándo crear y destruir regiones y (2) en qué región colocar los objetos cuando se crean. La principal ventaja del manejo de memoria basado en regiones con respecto a un recolector de basura es que el primero adelanta a tiempo de compilación la decisión sobre cuándo crear y destruir una estructura de datos.

Los primeros trabajos en esta técnica de manejo de memoria estaban destinados a lenguajes imperativos. Entre dichos trabajos se encuentra el de Ruggieri y Murtagh [102], en el que cada región esta asociada con el registro de activación de un procedimiento. Cada región se crea al inicio de una llamada a función, y se destruye cuando dicha llamada finaliza. Este mecanismo es refinado en el influyente trabajo de Tofte y Talpin [115, 116], que proporciona un enfoque basado en tipos para inferir regiones en un lenguaje funcional (ML), que incluye funciones de orden superior, y recursión polimórfica sobre regiones. Este trabajo ha sido implementado en el compilador MLKit [113]. Las regiones también han

sido aplicadas al paradigma orientado a objetos [28, 27]. Además, la especificación de *Java* para sistemas de tiempo real (*Real Time Specification for Java*) [22] proporciona un modelo que incluye áreas de memoria que son manejadas mediante regiones, y no están sujetas a recolección de basura.

En ausencia de recolección de basura, el manejo de memoria basado en regiones se adecua bien a la inferencia de cotas de memoria, pues la creación y destrucción de regiones queda determinada en tiempo de compilación, así como las regiones en las que se sitúan las distintas estructuras de datos. Este enfoque también resulta eficiente en cuanto a tiempo de ejecución [17], ya que tanto la creación como la liberación de regiones se lleva a cabo en tiempo constante. Sin embargo, una desventaja importante de las regiones es su naturaleza estrictamente anidada. Las regiones se crean y se destruyen en un orden LIFO[1], que no siempre se corresponde con los tiempos de vida de las estructuras de datos. Tal y como lo describen Berger, Zorn y McKinley [17]:

> *Regions provide high-performance but force the programmer to retain all memory associated with a region until the last object in the region dies [...]. We show that the performance gains of regions (up to 44%) can come at the expense of excessive memory retention (up to 230%). More importantly, the inability to free individual objects within regions greatly complicates the programming of server applications like Apache which rely on regions to avoid resource leaks.*

> Las regiones proporcionan una alta eficiencia, pero obligan al programador a retener toda la memoria asociada a una región hasta que el último objeto de la misma muere [...]. Nosotros mostraremos que el aumento de la eficiencia con regiones (hasta un 44 %) puede obtenerse a cambio de una excesiva retención de memoria (hasta un 230 %). Más importante: la incapacidad para liberar objetos individuales dentro de las regiones complica en gran medida la programación de aplicaciones del lado del servidor, como *Apache*, que utiliza regiones para evitar fugas de memoria.

Existen varios enfoques para romper esta estricta disciplina de regiones anidadas: reinicio de regiones [19], desacoplar la creación/liberación de regiones de su definición [2], anotar el programa con primitivas para crear/destruir regiones [54, 21], y regiones lineales [45], entre otros. Estos enfoques requieren, en menor o mayor medida, un profundo conocimiento sobre cómo el compilador infiere regiones. Algunos de ellos requieren de cierta ayuda del programador en forma de anotaciones en el programa. Como alternativa, se pueden combinar regiones y recolección de basura para reducir el impacto de fugas de memoria [114].

Destrucción explícita de celdas de memoria

Como alternativa a un recolector de basura, y en contrapartida al manejo puramente automático de memoria con regiones, se puede delegar la tarea de destrucción del memoria al programador. Podemos encontrar este enfoque en algunos lenguajes imperativos (C y C++ son conocidos ejemplos de esto), pero es muy inusual en lenguajes declarativos. En presencia de destrucción explícita, el compilador siempre sabe cuándo se crearán y destruirán las estructuras de datos en tiempo de ejecución, ya que dicha información viene proporcionada en el código fuente. Esto facilita la tarea de analizar cotas de memoria, con respecto a los lenguajes con recolección de basura.

[1] *Last In, First Out*. El primero en entrar es el último en salir.

No obstante, la destrucción explícita es una alternativa propensa a errores. La ejecución del programa puede verse interrumpida si se intenta acceder a aquellas partes de memoria que han sido liberadas previamente. El término *puntero descolgado* se utiliza comúnmente para hacer referencia a dichas partes de memoria. Este tipo de accesos a memoria liberada pueden surgir a causa de una compartición inesperada entre estructuras de datos. Por otra parte, los enfoques basados en destrucción explícita tampoco están exentos de fugas de memoria (esto es, memoria sin usar que no es liberada por el programa), ya que el programador puede olvidar liberar las estructuras de datos involucradas en su programa.

Existen varias técnicas basadas en tipos para evitar la presencia de punteros descolgados en lenguajes funcionales. En [57], Hofmann presenta LFPL, un lenguaje funcional con un sistema de tipos lineal [118] que permite la modificación *in situ* de estructuras de datos alojadas en el *heap.* En dicho lenguaje se hacen explícitas en el programa las posiciones de memoria en el *heap.* En [10, 11] se refina esta técnica mediante la distinción entre tres aspectos de uso distintos. Con ello se consiguen aceptar más programas seguros (con respecto a punteros descolgados) que su predecesor.

Dentro del paradigma imperativo, la *lógica de separación* [100] extiende la lógica de Hoare para poder especificar propiedades de programas que manejan estructuras de datos mutables. Esta lógica ha sido puesta en práctica en el analizador *Space Invader* [24], que utiliza técnicas de interpretación abstracta para la demostración de proposiciones en lógica de separación. En [69, 67], Konečný desarrolla una combinación de lógica de separación con el trabajo de LFPL, en la que se hace distinción entre los distintos niveles que componen una estructura de datos, e infiere asertos de separación entre dichos niveles.

Todos estos sistemas pueden demostrar la ausencia de punteros descolgados, pero no pueden demostrar su presencia. Esto implica la existencia de programas que, pese a ser seguros con respecto a punteros, son rechazados por el compilador, pues el problema de decisión consistente en determinar si un puntero está descolgado o no, es indecidible. Los análisis actuales sólo pueden aproximar esta propiedad. En el caso en que un programa seguro con respecto a punteros se considere (erróneamente) inseguro por un análisis estático, la implementación puede actuar de dos maneras distintas: avisar al programador de un posible puntero descolgado, aún permitiendo la ejecución del programa, o bien, interrumpir la generación de código objeto, de modo que los programas considerados inseguros no pueden ser ejecutados. La primera posibilidad ya no puede garantizar la propiedad de seguridad de punteros. La segunda de ellas adquiere más relevancia en el contexto de sistemas críticos en seguridad, y fuerza al programador a modificar su programa para eliminar aquellas destrucciones que hacen que el programa sea rechazado por el análisis. Esto puede repercutir en la aparición de fugas de memoria. No obstante, y al igual que en los enfoques de memoria basados en regiones, la destrucción explícita puede combinarse con técnicas de recolección de basura.

El enfoque de *Safe* a la gestión de memoria

En aquellas aplicaciones en las que la seguridad de punteros resulta imprescindible, los mecanismos expuestos anteriormente pueden inferir una aproximación de esta propiedad. Dicha aproximación se realiza de manera que la seguridad de punteros queda garantizada, aún cuando ciertos programas seguros puedan ser rechazados. La sobreaproximación es inherente a la inferencia de propiedades indecidibles. En el caso particular de la seguridad de punteros, la sobreaproximación puede conllevar fugas de memoria.

El manejo de memoria en *Safe* consiste en una combinación de regiones y destrucción explícita mediante ajuste de patrones destructivo (al estilo del lenguaje LF de Hofmann y Jost [58]). Se garantiza

la seguridad de punteros mediante un sistema de tipos. El obstáculo de sobreaproximar esta propiedad sigue siendo insalvable, pero la combinación de estos mecanismos puede aliviar sus consecuencias. La destrucción explícita permite la destrucción selectiva de porciones de estructuras de datos dentro de una región, sin necesidad de esperar a que la región completa sea liberada. En aquellos casos en los que el compilador prohíbe la destrucción explícita de una estructura de datos determinada, el manejo de memoria basado en regiones garantiza que la memoria ocupada por la misma será liberada en algún momento futuro, en particular, cuando se lleve a cabo la destrucción de la correspondiente región. Por tanto, cada técnica proporciona una vía de escape a las limitaciones de la otra técnica.

Debido a lo explicado anteriormente, decimos que *Safe* proporciona un enfoque *semiexplícito* del manejo de memoria, pues proporciona destrucción implícita mediante regiones (inferidas automáticamente) y destrucción explícita (indicada por el programador). Como desventaja a este enfoque, el programador necesita cierta compresión sobre el mecanismo mediante el cual las regiones se crean y se destruyen, con el objetivo de evaluar la necesidad de destrucción explícita en su programa. Las consecuencias de esto se suavizan en *Safe* mediante la definición de un modelo de memoria basado en regiones más simple que el original de Tofte y Talpin [116]. Existe una sola región en cada llamada a función, de modo que el programador sólo ha de considerar si una estructura de datos determinada es local a una determinada función.

Una ventaja importante de combinar regiones y destrucción explícita es el hecho de que ya no es necesario un recolector de basura. De este modo, los tiempos de vida de las estructuras de datos quedan completamente determinadas en tiempo de compilación y, como consecuencia, resulta viable analizar el consumo de memoria de programas *Safe*.

Certificación de propiedades de seguridad

La certificación de propiedades de un programa consiste en demostrar matemáticamente las mismas. En el ámbito del Código con Demostración Asociada (*Proof Carrying Code*, PCC), estas demostraciones se comprueban automáticamente con una herramienta adecuada. La línea de investigación de PCC comenzó con el influyente trabajo de Necula [88], y tiene por objetivo crear una infraestructura en la que se adjunta a un programa una demostración formal que garantice la presencia de ciertas propiedades deseables del mismo. Estas demostraciones formales componen un *certificado*. El productor de código genera, además del programa compilado, el correspondiente certificado, y el consumidor de código debe comprobar que el certificado recibido es consistente con el programa proporcionado, y que las demostraciones contenidas en el certificado son correctas. La comprobación del certificado se realiza de manera automática, y, normalmente, con ayuda de un asistente de demostraciones, como Isabelle/HOL [93].

Entre las primeras publicaciones en relación a compiladores-certificadores tenemos las de [89, 91, 30], en las que los autores desarrollan un compilador-certificador para un subconjunto de *bytecode* de Java. El proyecto *MRG* (*Mobile Resource Guarantees*) aborda la generación de código móvil junto con un certificado que demuestra consumo de memoria acotado. Los certificados se generan a nivel de un lenguaje *bytecode*, similar al de Java, llamado *Grail* [9]. Más recientemente, en el proyecto *Mobius* (*Mobility, Ubiquity, and Security*) [14] se intenta extender el paradigma PCC a entornos más generales, como sistemas concurrentes y distribuidos, análisis de recursos, etc. Está orientado a los dispositivos móviles que satisfacen la especificación *MIDP* (*Mobile Information Device Profile*) de Java ME.

Análisis de consumo de memoria

El área de investigación de análisis de recursos, aunque relativamente reciente, ha obtenido una atención considerable durante los últimos años, principalmente debido al uso de técnicas matemáticas (tales como programación lineal, o resolución de recurrencias) en los lenguajes de programación. En particular, la inferencia de cotas de memoria es una tarea muy compleja que requiere varios análisis previos, siendo cada uno de ellos un desafío en sí mismo. Los primeros resultados de análisis de consumo de memoria estaban orientados al paradigma funcional. Las técnicas adquiridas se adaptaron posteriormente a lenguajes de uso más general, como Java o C++.

Hughes y Pareto presentan en [61] un lenguaje funcional de primer orden con un sistema de tipos y efectos que garantiza la terminación y ejecución en espacio de memoria acotado de los programas. Este sistema combina las regiones de Tofte y Talpin con un sistema de tipos con tamaños [62, 96].

El primer método para obtener cotas de memoria de manera completamente automática es debido a Hofmann y Jost [58]. Su análisis, basado en un sistema de tipos con anotaciones de recursos, puede inferir cotas de memoria lineales en un lenguaje funcional de primer orden con destrucción explícita. Estas técnicas se han aplicado a subconjuntos de lenguajes imperativos, tales como Java [59] (sin inferencia de tipos), y C [48] (para el cálculo de cotas superiores al tiempo de ejecución). Un sistema de tipos anotado también sirve como base para el análisis de consumo de memoria de pila desarrollado por Campbell [25, 26]. Las técnicas de Hofmann y Jost se extienden en [64, 63] a programas de orden superior. Este último trabajo proporciona un marco general capaz de acomodar distintas nociones de coste. Más recientemente, Hoffmann y Hofmann han extendido el trabajo inicial de [58] a la inferencia de cotas de memoria polinómicas [56, 55].

El método clásico de análisis de recursos, debido a Wegbreit [119], consiste en la generación de una relación de recurrencia a partir del programa analizado y, en una segunda fase, el cálculo de una expresión en forma cerrada (esto es, sin recursión), equivalente a la relación de recurrencia. Vasconcelos y Hammond siguen este enfoque en [117], que funciona de manera completamente automática en la generación de relaciones de recurrencia, pero requiere el uso de un resolutor externo para obtener una expresión en forma cerrada. El sistema COSTA [5] sigue un procedimiento similar, pero proporciona un resolutor de recurrencias propio, PUBS [3, 4], que puede manejar relaciones de recurrencia no deterministas de varias variables. COSTA esta basado en técnicas de interpretación abstracta, trabaja a nivel de *bytecode* Java, y soporta varias nociones distintas de coste, tales como número de instrucciones ejecutadas, consumo de memoria, y número de llamadas a un determinado método. Dado que el manejo de memoria en Java se basa en un recolector de basura, su enfoque a consumo de memoria es paramétrico en el comportamiento de dicho recolector [7]. Las cotas calculadas mediante este sistema van más allá de expresiones lineales: puede calcular cotas polinómicas, logarítmicas y exponenciales.

El proyecto SPEED [49], aunque no está orientado directamente a cotas de consumo de memoria, calcula cotas simbólicas no lineales mediante la combinación de técnicas, tales como transformaciones de bucles, instrumentación con contadores, y generación de invariantes.

El problema de inferencia de cotas superiores de memoria está directamente relacionado con la inferencia de relaciones de tamaño entre las estructuras de datos. El trabajo principal sobre tipos con tamaños es el de Hughes, Pareto y Sabry [62], mencionado anteriormente, y que se limita a comprobar los tipos proporcionados. El problema de la inferencia para este sistema de tipos es abordado por Chin y Khoo [29]. Otra técnica consiste en el uso de técnicas de interpretación abstracta en el dominio de los poliedros convexos, desarrollada por Benoy y King [16]. También se aplica interpretación abstracta en

[108] para aproximar la altura[2] de las estructuras de datos. Todas estas técnicas se limitan a la obtención de relaciones de tamaño lineales. El trabajo de Shkaravska, van Eekelen y van Kesteren [105] permite obtener relaciones de tamaño polinómicas. Proporciona un sistema de tipos, en el que la comprobación de tipos es decidible bajo ciertas condiciones determinables a través de la estructura sintáctica del programa. Mediante una combinación de técnicas basadas en pruebas e interpolación polinómica se aborda la inferencia de tipos en este sistema. Más recientemente, este enfoque se ha adaptado al cálculo de cotas polinómicas al número de iteraciones en un bucle para programas Java [106].

Objetivos de la tesis

El proyecto *Safe*, considerado en su totalidad, tiene como meta el estudio de la viabilidad de los lenguajes funcionales para la programación de dispositivos con requisitos estrictos de memoria. Los objetivos de esta tesis pretenden acercarse a esta meta general. Son los siguientes:

1. **Desarrollar un análisis estático eficiente y completamente automático para garantizar la seguridad de punteros.**

   Partiendo del modelo de memoria de *Safe*, explicado informalmente en las secciones previas, el análisis debe acomodar la gestión de memoria basada en regiones, así como la destrucción explícita. Este análisis estará basado en un sistema de tipos, que será demostrado correcto con respecto a la semántica operacional del lenguaje.

   En relación a la inferencia de tipos, consideraremos las regiones y destrucción explícita por separado:

   (a) Un algoritmo de inferencia de regiones anotará el código fuente del programa con variables de región, de modo que el programa resultante esté correctamente tipado con respecto al sistema de tipos mencionado anteriormente. Este proceso de anotación se realizará de manera óptima, de manera que cada estructura de datos será situada en la región con el tiempo de vida más corto que no comprometa la integridad de la estructura de datos. El algoritmo soportará funciones con recursión polimórfica sobre regiones, y su corrección será demostrada con respecto al sistema de tipos.

   (b) Un algoritmo de inferencia de marcas asegurará que la destrucciones explícitas indicadas por el programador no generan punteros descolgados en tiempo de ejecución. En este contexto, las relaciones de compartición que se producen entre las estructuras de datos adquieren un carácter determinante. La aproximación de estas relaciones, que vendrá dada por un análisis ya existente, servirá como base para el algoritmo de inferencia de seguridad de destrucción. Dicho algoritmo también será demostrado correcto con respecto al sistema de tipos.

2. **Certificar seguridad de punteros de manera automática.**

   Suponiendo que un programa *Safe* está correctamente tipado con respecto al sistema de tipos mencionado anteriormente, describiremos cómo el compilador genera un certificado, cuya validez podrá ser comprobada por el asistente de demostraciones Isabelle. Este certificado demostrará la propiedad de seguridad de punteros del programa a compilar. De modo general, el proceso de generación de certificados puede dividirse en dos tareas:

[2] La altura de una estructura de datos es la cadena más larga de punteros que puede obtenerse a partir de uno dado.

(a) La primera tarea consiste en la demostración de que los programas bien tipados no realizan accesos a punteros descolgados. Esta tarea incluye la formalización de la corrección del sistema de tipos en un asistente de demostraciones. Los teoremas correspondientes a esta tarea son genéricos, de modo que se prueban una sola vez, y se reutilizan en cada certificado.

(b) La segunda tarea incluye la demostración de que el programa a compilar está bien tipado. Combinando esta parte con los resultados de la primera tarea, el certificado demostrará que el programa a certificar no realiza accesos a punteros descolgados.

Esta tesis sólo abordará la segunda tarea: la generación de la parte del certificado que es específica de cada programa. En [37, 35] puede encontrarse más información sobre la parte genérica.

3. **Desarrollar un modelo de costes de memoria para programas *Safe*.**

   Un modelo de costes de memoria permite describir formalmente las necesidades de memoria de los programas escritos en un lenguaje. Nuestro modelo de coste formará parte de la semántica operacional de paso grueso de *Safe*, y tendrá en cuenta tanto el consumo de pila como el de *heap*. Este modelo será derivado formalmente a partir de la traducción del código fuente *Safe* a código de la máquina virtual imperativa SVM (*Safe* Virtual Machine), y se demostrará su corrección con respecto a dicha traducción.

4. **Desarrollar un análisis basado en interpretación abstracta para la inferencia de cotas de memoria de pila y *heap*.**

   Partiendo del modelo de costes del punto anterior, nuestro objetivo será la utilización de técnicas de interpretación abstracta [33] para la inferencia de expresiones no lineales, monótonas y en forma cerrada que acoten los consumos de pila y *heap* de un determinado programa. Las cotas resultantes estarán expresadas en función de los tamaños de los parámetros de entrada.

   Incluso si nos limitamos a un lenguaje funcional de primer orden, como *Safe*, la inferencia de cotas de memoria seguras es una tarea demasiado compleja, que incluye el desarrollo de herramientas auxiliares, tales como un análisis de tamaños de las estructuras de datos apuntadas por las variables del programa, y un análisis del tamaño del árbol de llamadas. Cada uno de estos análisis podría ser, por sí mismo, objeto de una tesis doctoral. Por tanto, nos centraremos en la inferencia de cotas de memoria de pila y *heap*, suponiendo que la información de tamaños y árbol de llamadas viene dada externamente.

   Se demostrará formalmente la corrección de los resultados del análisis con respecto al modelo de costes de memoria desarrollado en el punto (3). También se especificarán ciertas condiciones en la información externa sobre el árbol de llamadas, que permitirán demostrar la *reductividad* de las cotas obtenidas en presencia de las mismas. La ventaja de tener cotas reductivas es que nos permitirán obtener una secuencia no creciente de cotas como resultado de iterar el análisis varias veces para un determinado programa.

En este trabajo se ha puesto especial énfasis en la corrección formal de cada uno de los análisis. Pensamos que, en general, las demostraciones formales son necesarias cuando se desarrollan lenguajes destinados a sistemas críticos en seguridad, pero, en particular, son decisivas en el proyecto *Safe*, ya que los objetivos de dicho lenguaje comprenden la certificación formal de las propiedades que están siendo demostradas. En este trabajo sólo abordaremos la certificación de propiedades de seguridad de punteros, pero también se ha trabajado en la certificación de cotas de memoria para *Safe* [40, 35].

Estructura del trabajo

Dado que el doctorando (autor de esta tesis) aspira a la mención de Doctor Europeo, la tesis está redactada en inglés. Su estructura es la siguiente:

- El Capítulo 1 motiva la aparición del lenguaje *Safe* como alternativa a otros modelos de memoria existentes. Se definen los objetivos de la tesis y la estructura de la misma. La traducción a español de este capítulo se encuentra en las páginas precedentes.

- En el Capítulo 2 presentaremos, a través de varios ejemplos, los conceptos básicos del lenguaje *Safe*. Después se formalizará la semántica operacional del lenguaje, que servirá como base para derivar una máquina virtual para la ejecución de programas *Safe* (SVM). La traducción de *Safe* a instrucciones de la SVM nos permitirá definir un modelo de costes, cuya corrección se demostrará correcta con respecto a la traducción. Con este modelo de costes habremos conseguido el tercer objetivo de la lista anterior. Este capítulo es una versión extendida del trabajo descrito en [82, 86].

- El Capítulo 3 presenta un sistema de tipos que garantiza que la asignación de regiones y la destrucción explícita se realizan de forma segura. Allí suponemos que el código fuente del programa siendo analizado viene anotado con variables de región, y mostraremos que los programas bien tipados no contienen accesos a punteros descolgados. El problema de la inferencia de derivaciones de tipo correctas será detallado en los dos capítulos siguientes. Este capítulo es una versión mejorada del trabajo [84].

- El Capítulo 4 aborda el problema de la inferencia de regiones. Dado un programa *Safe*, el algoritmo decora su árbol abstracto de sintaxis con variables de región, de manera que el programa resultante está bien tipado. La corrección y optimalidad del algoritmo serán demostradas formalmente. Este algoritmo ha sido publicado en [85].

- El Capítulo 5 se encarga de la destrucción explícita. En él se desarrollará un algoritmo que garantice la seguridad de las destrucciones explícitas indicadas por el programador. Será demostrado correcto y completo con respecto al sistema de tipos del Capítulo 3. Este capítulo es una versión extendida de la publicación [81].

- Lo expuesto en los capítulos 3, 4, y 5 está destinado a obtener derivaciones de tipos para un programa *Safe* (primer objetivo). Una vez se han obtenido estas derivaciones, el Capítulo 6 detalla cómo se representan formalmente como un guión de demostraciones en Isabelle/HOL, con el fin de certificar la seguridad de punteros (segundo objetivo). Este trabajo corresponde a la publicación [36].

- En el Capítulo 7 se desarrollará un análisis basado en técnicas de interpretación abstracta para la inferencia de cotas de memoria de pila y de *heap* para programas *Safe* (cuarto objetivo). En primer

lugar se definirá un dominio abstracto, y una función de abstracción que asociará expresiones *Safe* con elementos del dominio abstracto. Esta función será demostrada correcta con respecto al modelo de costes del Capítulo 2. Este capítulo extiende nuestro trabajo previo sobre análisis de consumo de memoria [83].

- Finalmente, en el Capítulo 8 se discutirán los resultados obtenidos, y se esbozarán algunas líneas de investigación futuras.

La traducción a español de este último capítulo se muestra en las páginas siguientes.

Conclusiones y trabajo futuro

Conclusiones

En esta tesis se han diseñado e implementado varios análisis para la demostración de propiedades de seguridad de punteros y consumo acotado de memoria en programas escritos en un lenguaje funcional de primer orden que combina regiones y destrucción explícita. Recapitularemos los objetivos de este trabajo, y evaluaremos hasta qué punto se han conseguido.

1. **Desarrollar un análisis estático eficiente y completamente automático para garantizar la seguridad de punteros** [98, 84, 85, 81]**.**

   Hemos definido un sistema de tipos que superpone variables de tipo de región y marcas a los tipos algebraicos de un sistema estándar de tipos Hindley-Milner. Las variables de tipo región reflejan las regiones en las que una estructura de datos puede estar situada, y permite que el compilador rastree las estructuras de datos que estarán, en tiempo de ejecución, localizadas en la región temporal de trabajo de la función en ejecución. También asegura que tanto los parámetros de entrada como el resultado de la función quedan fuera de dicha región. Las marcas indican si una estructura de datos va a ser destruida mediante ajuste de patrones explícito. Se ha hecho uso de un análisis auxiliar para aproximar las relaciones de compartición entre las distintas variables. Estas relaciones determinan qué variables quedarán corruptas cuando una estructura de datos determinada sea liberada.

   Hemos proporcionado un algoritmo de inferencia para este sistema de tipos con destrucción. Está compuesto de dos partes separadas:

   (a) Un algoritmo de inferencia de regiones, que anota el programa con variables de región, e infiere las variables de tipo región de la derivación de tipos. El hecho de que nuestro lenguaje sea de primer orden, y de que nuestro modelo de memoria sea más simple que el de [116], nos permite obtener un algoritmo de inferencia más simple y eficiente que el de Tofte y Birkedal [112].

   (b) Un algoritmo de inferencia de marcas, que anota los tipos con marcas y garantiza que el ajuste de patrones destructivo se realiza de forma segura.

   Hemos comprobado el funcionamiento de los algoritmos de inferencia con varios casos de prueba. Una cantidad considerable de algoritmos que manipulan estructuras de datos en espacio de *heap* constante han sido correctamente tipados por el algoritmo. En aquellos casos de prueba que no han sido tipados, hemos identificado el análisis de compartición auxiliar como la principal fuente de pérdida de precisión.

El sistema de tipos se ha demostrado correcto con respecto a la semántica de *Safe*, mientras que la corrección de los algoritmos de inferencia ha sido demostrada correcta con respecto al sistema de tipos. Además, el algoritmo de inferencia de marcas ha resultado ser completo con respecto a dicho sistema.

2. **Certificar seguridad de punteros de manera automática** [36]**.**

   Hemos mostrado cómo el compilador genera un guión en Isabelle/HOL que demuestra que el programa certificado está correctamente tipado. Junto con un conjunto de teoremas (previamente demostrado) que garantiza la seguridad de punteros de programas correctamente tipados, se certifica que el programa objeto es seguro con respecto a punteros.

3. **Desarrollar un modelo de costes de memoria para programas *Safe*** [82]**.**

   A partir de la semántica operacional de paso grueso del lenguaje, y en una serie de refinamientos sucesivos, hemos derivado una máquina abstracta imperativa, llamada SVM, en la se ejecutan los programas *Safe*. Se ha formalizado la traducción entre *Safe* y las instrucciones de la SVM. Esta traducción nos ha permitido establecer una conexión entre cada expresión sintáctica de *Safe* y su consumo de memoria, tanto de pila como de *heap*. Se ha formalizado esta conexión mediante anotaciones de recursos, que se han adjuntado a la semántica operacional de paso grueso. Estas anotaciones dan lugar a un modelo de costes para *Safe*, y su corrección ha sido demostrada con respecto al consumo real de los programas *Safe* cuando se ejecutan en la SVM.

4. **Desarrollar un análisis basado en interpretación abstracta para la inferencia de cotas de memoria de pila y *heap*** [83]**.**

   Hemos desarrollado un análisis que, dado un programa *Safe*, devuelve una expresión que acota sus consumos de memoria de pila y *heap* en función de los tamaños de los parámetros de entrada. Nuestro dominio abstracto es el conjunto de funciones monótonas con el orden $\sqsubseteq$ habitual.

   Como primer paso, hemos definido una función de interpretación abstracta que modela las necesidades de *heap* y pila de cada expresión. Se ha demostrado la corrección de esta función con respecto al modelo de costes desarrollado anteriormente. Tras esto, hemos presentado unos algoritmos que abordan el problema de inferencia de costes de funciones recursivas. Hemos demostrado que el resultado de estas funciones es reductivo, suponiendo ciertas condiciones. Esto implica que podemos iterar la función de interpretación abstracta con el fin de mejorar los resultados de estos algoritmos en cada iteración.

   En una primera fase hemos considerado la inferencia de funciones recursivas en ausencia de recursión polimórfica sobre regiones y destrucción explícita, pero se ha mostrado la viabilidad de la integración de estas características en nuestros algoritmos de inferencia, siempre y cuando el resultado se encuentre en el dominio abstracto considerado.

   Las principales ventajas de nuestro enfoque son su flexibilidad para incluir funciones de un amplio rango de clases de complejidad, a pesar de que, en este momento sólo se admiten funciones monótonas. Resulta notable el hecho de que las cotas obtenidas pueden ser mejoradas mediante simples iteraciones del algoritmo de inferencia.

Trabajo futuro

En esta sección describiremos algunas extensiones y mejoras posibles que se realizarán en un futuro. Con respecto a nuestro sistema de tipos con destrucción, tenemos los siguientes problemas a abordar:

- **Mejoras en el análisis de compartición.**

  En algunos de los casos de estudio del Capítulo 5 se ha mostrado la necesidad de un análisis de compartición más preciso que el actualmente implementado [98], para que el programador no tenga que ajustar manualmente sus resultados. Nuestro análisis de compartición actual distingue entre las áreas recursivas y no recursivas de una estructura de datos, y tiene en cuenta las variables que pueden apuntar a cada una de estas partes. No obstante, cuando se aproximan las relaciones de compartición en el contexto de las llamadas a función, esta distinción se ha de perder para no comprometer la corrección del dicho análisis. Una mejora que puede aumentar la precisión de nuestro análisis es la identificación de los descendientes particulares de una estructura de datos que están siendo apuntados por una determinada variable.

- **Tipos principales.**

  En el Capítulo 4 se ha mostrado la ausencia de tipos principales en las reglas $\vdash_{Reg}$ en algunos casos patológicos como, por ejemplo, la función que hace una copia de su parámetro de entrada y devuelve la copia como resultado. El motivo tras esto es el hecho de que el operador de copia debe aplicarse a un tipo *algebraico*, pero puede ser *cualquier* tipo algebraico. La sintaxis de nuestros tipos nos permite especificar tipos algebraicos concretos, tales como $[\alpha]@\rho_1$, o *Tree Int* @ ρ_2, y variables de tipo polimórficas, que pueden instanciarse a cualquier tipo, ya sea básico o algebraico. No obstante, no existe ningún mecanismo en nuestra sintaxis de tipos para expresar un tipo algebraico arbitrario, en el que la variable de tipo de región más externa esté fijado. Una solución a este problema pasa por añadir una nueva clase de tipos que englobe las estructuras algebraicas sin especificar, cuya región más externa es conocida. Planeamos, en un futuro próximo, integrar esta extensión en el sistema de tipos.

- **Tipos de datos mutuamente recursivos**

  La implementación actual de *Safe* no permite la especificación de tipos de datos mutuamente recursivos. La extensión del sistema de tipos con esta clase de tipos más amplia tendría repercusiones considerables en el análisis de compartición, pero también en la semántica del propio lenguaje. En ausencia de recursión mutua, es fácil hacer referencia a la parte recursiva de una estructura de datos, tan sólo basta considerar las posiciones recursivas de las constructoras que componen dicha parte recursiva. Como consecuencia, los hijos recursivos de una celda sólo están a un puntero de distancia de la misma. Esto deja de cumplirse si dos tipos de datos determinados son mutuamente recursivos, pues podría haber varias celdas a mitad del camino entre una determinada y sus hijos recursivos. Bajo estas circunstancias, el concepto de *posición recursiva* adquiere mayor dificultad en su definición.

- **Tipos de datos con recursión anidada.**

  Safe permite tipos de datos recursivos, pero la recursión ha de aparecer en el nivel más externo. El añadir soporte para recursión a niveles más profundos (recursión anidada) conlleva los mismos problemas que el soporte para tipos de datos mutuamente recursivos. De hecho, los tipos de datos con recursión anidada pueden traducirse a conjuntos de tipos de datos mutuamente recursivos.

- **Destrucción anidada.**

  Aunque los tipos en peligro (*in-danger*) pueden detectar aquellas estructuras de datos cuya parte no recursiva está corrupta, no dan ninguna información sobre qué parte concreta de la estructura está corrupta. Por ejemplo, supongamos una variable x que tiene tipo $[[\alpha]@\rho_1]\#@\rho_2$. El hecho de que x tenga un tipo en peligro implica que:

  - La lista más externa apuntada por x puede ser destruida, y/o
  - Algunas de las listas internas pueden ser destruidas, y/o
  - Algunos de los elementos de las listas internas pueden ser destruidos.

  Si extendemos nuestro sistema con tipos condenados anidados podremos aumentar la precisión del sistema de tipos para determinar hasta qué nivel de anidamiento puede destruirse la estructura de datos apuntada por x. Por ejemplo, $[[\alpha]@\rho_1]!@\rho_2$ sólo permite la destrucción de la lista más externa, mientras que $[[\alpha]!@\rho_1]!@\rho_2$ permitiría al programador destruir las listas internas, al tiempo que se asegura que ninguno de los elementos de tipo α se destruirá. Para llevar a cabo esta mejora se necesita aumentar la precisión del análisis de compartición auxiliar, de modo que pueda diferenciar entre los distintos niveles (capas) de una estructura de datos. Esto resulta más preciso que hacer distinción entre la parte recursiva de una estructura de datos y la parte no recursiva.

En relación a nuestro análisis de consumo de memoria basado en interpretación abstracta, señalamos los siguientes aspectos a mejorar:

- **Soporte para tipos de recursos diferentes.**

  Esta tesis tenía como objetivo obtener cotas de consumo de memoria, pero el área de investigación de análisis de recursos es más amplia; aborda más tipos de recursos. La mayoría de ideas expuestas en el Capítulo 7 pueden aplicarse a otros tipos de recursos. Una propiedad deseable en cualquier análisis de recursos es el disponer de algún tipo de parametricidad en el recurso que se quiere acotar. Por ejemplo, el sistema COSTA [5] proporciona soporte para diferentes tipos de recursos: números de instrucciones ejecutadas, consumo de memoria en presencia de diferentes modelos de recolección de basura, número de llamadas a un determinado método.

  El trabajo de Aspinall et al. [9] presenta el concepto genérico de *álgebra de recursos*, permitiendo un marco general en el que se pueden integrar diferentes tipos de recursos. La generalización de nuestra función de interpretación abstracta para álgebras de recursos arbitrarias puede llevarse a cabo sin dificultad. No obstante, el cálculo de las cotas iniciales requiere cierta cautela. El algoritmo que calcula la cota inicial Δ_0 (ver Capítulo 7) sólo puede aplicarse en aquellos recursos que son *composicionalmente aditivos*, esto es, aquellos recursos en los que el coste de una expresión compuesta es la suma de los costes de sus subexpresiones básicas. El algoritmo para el cálculo de la cota inicial σ_0 puede aplicarse en aquellos recursos cuyo nivel de ocupación/consumo puede crecer o decrecer a lo largo de la ejecución del programa, y en los que se requiere calcular el mayor nivel de consumo máximo. El algoritmo de cálculo de la μ_0 también calcula un nivel máximo de consumo, pero también conlleva cierta componente aditiva. Un marco de recursos general debería establecer ciertas condiciones en las álgebras de recursos aditivas para determinar qué algoritmo puede ser utilizado para calcular las cotas iniciales al consumo de cada tipo de recurso.

- **Soporte para modelos de tamaño distintos.**

  De la misma manera en la que es útil considerar distintas clases de recursos, también resulta conveniente considerar diferentes modelos de tamaños. El término *tamaño* se refiere en función de

la cual se expresan las funciones de coste, y proporciona cierta idea sobre la dificultad que ofrece la entrada para ser procesada. En este sentido, el tamaño de una estructura de datos no tiene por qué corresponderse con la cantidad de memoria de *heap* ocupada por la estructura de datos. *Safe* tiene una definición fija para el tamaño de una estructura de datos: el número de celdas de su parte recursiva. La definición de tamaño también está fija en el sistema COSTA: el tamaño de una estructura de datos es la longitud de la cadena de punteros más larga que parte de la celda raíz de la estructura. El modelo de costes puede influir de modo decisivo en el orden de complejidad resultante. Por ejemplo, supongamos una función que hace una copia de un árbol binario dado. En *Safe* se obtiene una complejidad en espacio de $\mathcal{O}(t)$, mientras que, según COSTA, la complejidad es de $\mathcal{O}(2^t)$. La diferencia está en el significado de la t: en *Safe*, t representa el número de nodos del árbol, mientras que en COSTA representa su altura.

Un buen punto de partida es el trabajo de Tamalet et al. [109], que presenta un sistema de tipos con tamaños con soporte para tipos de datos arbitrarios. Los tamaños se definen asignando un peso a cada constructor de datos. El enfoque basado en análisis amortizado de Jost et al. [64, 63] puede también asignar potenciales distintos a cada constructor.

- **Cotas superiores no monótonas.**

  Tal como explicamos en el Capítulo 7, el dominio abstracto de nuestro sistema es el conjunto $\mathbb{F}$ de funciones monótonas. La condición de monotonía es esencial para asegurar la corrección de la función de interpretación abstracta, pero esta condición es demasiado restrictiva cuando se introduce la destrucción explícita, ya que ésta puede producir cargos negativos en el *heap*. Existen algunas funciones interesantes que no pueden ser procesadas por nuestro algoritmo debido a esta restricción como, por ejemplo, la función que destruye una estructura de datos completamente sin construir nada a cambio. En estos casos podemos observar una especie de dualidad con respecto a nuestro enfoque actual. Supongamos una función que sólo destruye una celda en cada llamada recursiva, y no realiza nada más. Si nos fijamos en el consumo de *heap*, el peor caso se corresponde con el número *mínimo* de llamadas recursivas, ya que, en ese caso, se destruyen menos celdas, lo que implica menos memoria liberada. Por tanto, a la hora de analizar una función cuyos costes decrecen monótonamente en cada llamada, debemos considerar *cotas inferiores* en el tamaño del árbol de llamadas. Aquellas funciones de coste que no son ni monótonamente crecientes, ni monótonamente decrecientes deberán considerarse de manera fragmentada.

- **Convergencia de una secuencia de funciones de coste.**

  Nuestro enfoque permite obtener una secuencia no creciente de costas superiores mediante la iteración de la función de interpretación abstracta. En algunos casos se ha obtenido una cadena estrictamente decreciente de funciones, sin alcanzar un punto fijo. En muchos de estos casos hemos sido capaces de representar gráficamente el comportamiento de esta secuencia, y hemos conseguido determinar (manualmente) la función de coste a la que converge, que resultó ser el punto fijo de la función de interpretación abstracta. Resulta interesante estudiar las condiciones bajo las cuales se puede asegurar la convergencia de estas secuencias, y la función resultante que se puede obtener cuando el número de iteraciones tiende a infinito.

Finalmente, y en relación al lenguaje *Safe* en sí, hay dos extensiones que merece la pena considerar:

- **Funciones de orden superior.**

  El mayor desafío que es objeto de trabajo futuro es la extensión del lenguaje con funciones de orden superior. Esto requerirá cambios en lenguaje en casi todos sus aspectos: sintaxis, semántica,

máquina virtual, inferencia de regiones y de marcas, análisis de compartición, análisis de consumo de memoria, y certificación.

La adaptación de algunos de los análisis actualmente existentes a un lenguaje de orden superior puede, relativamente, llevarse a cabo sin dificultad. Esto incluye el algoritmo de inferencia de regiones, y el análisis de consumo de memoria. Con respecto al primero, el compilador MLKit [116, 113] ya soporta inferencia de regiones para funciones de orden superior. Dado que planeamos mantener la decisión de mantener una sola región por llamada a función, esperamos que nuestro algoritmo siga siendo más simple que el de [112]. Con respecto al análisis de consumo de memoria adaptado a funciones de orden superior, las expresiones de coste de los parámetros funcionales deberán aparecer simbólicamente en la expresión resultante. Sólo cuando se pasen funciones concretas como parámetros (y, por tanto, sus costes de memoria hayan sido inferidos), estos símbolos podrán ser reemplazados por funciones de coste concretas. Un enfoque diferente, basado en el análisis amortizado, es el de [64], que extiende el trabajo de [58] para incluir soporte de funciones de orden superior.

- **Lenguajes objeto alternativos.**

  La implementación actual de *Safe* genera *bytecode* de Java como resultado de compilación. Una gran ventaja de la máquina virtual de Java es su disponibilidad en muchas plataformas de software y hardware. No obstante, es algo restrictiva en lo que a manejo de memoria se refiere. Como consecuencia de ello, la destrucción explícita de una celda se trata mediante la unión de la celda destruida a una lista de celdas libres. Esta celda será reutilizada por el sistema de ejecución en una reserva de memoria posterior. Además, la naturaleza estáticamente tipada de la máquina virtual de Java hace muy laborioso el proceso de traducción. Resulta conveniente dirigir la generación de *bytecode* a otras máquinas virtuales y entornos de compilación que permitan más flexibilidad en el manejo de memoria. Un ejemplo de este tipo de entornos es la máquina virtual de bajo nivel LLVM (*Low Level Virtual Machine*), que proporciona un conjunto de instrucciones independientes del lenguaje. Los programas escritos en este lenguaje pueden ser traducidos posteriormente a código máquina.

Safety properties and memory bound analysis in a functional language without a garbage collector

PhD Thesis

Author: **Manuel Montenegro Montes**

Advisors: **Ricardo Peña Marí and Clara María Segura Díaz**

Facultad de Informática

Universidad Complutense de Madrid

September 2011

Abstract

In most functional languages, memory management is delegated to the runtime system. There usually exists a *garbage collector* in charge of determining, at runtime, which parts of the memory are no longer needed and can be disposed of. The main advantage of this approach is that programmers are not bothered about memory management issues. However, this purely implicit approach makes it difficult to predict, at compile time, the lifetimes of the data structures at runtime. Other languages delegate the task of memory management to the programmer. In this way, the lifetimes of data structures are known at compile time. However, this is an error-prone approach, since the programmer might try to access to memory areas which have been disposed of previously (*dangling pointers*), mainly due to unexpected sharing between data structures.

Safe is a first-order functional language that provides a semiexplicit approach to memory management by means of two mechanisms: regions (inferred by the compiler), and destructive pattern matching (controlled by the programmer). In this way, we can do without a garbage collector to destroy the data structures that are no longer used. As a consequence, the behaviour of programs (regarding memory usage) becomes more predictable, and eases the task of developing a static analysis to infer upper bounds on the memory needs of a program.

The incorporation of explicit mechanisms for memory destruction implies the risk of accessing dangling pointers. The first goal of this thesis is to develop a static analysis for guaranteeing the safety of a program with regard to its memory accesses (specifically, the absence of dangling pointers). Once the compiler has checked the presence of this property for a given program, our next goal is to formalize this property as a certificate, so the latter can be automatically checked by the code receiver. Finally, we will develop an abstract interpretation-based analysis for inferring upper bounds to the memory needs of a program. Since the final aim of the *Safe* project is the certification of pointer-safety properties and bounded memory execution, we have put emphasis on the formal correctness of each analysis.

Contents

Chapter 1

Introduction

This thesis describes the automatic analysis of pointer-safety properties and memory bounds for a first-order functional language called *Safe*. This language has been developed in the last few years as a research platform for analysing and formally certifying properties of programs, with regard to memory usage. It was introduced for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. In this chapter we will motivate the existence of *Safe*, and specify the goals of this thesis.

1.1 Motivation

Computer systems are not fully reliable. They just do what a programmer tells them to do, and programmers make mistakes. As software plays an increasingly relevant role on computer systems, its reliability becomes a major concern, especially in those cases when economic damages or human losses might arise as a consequence of a malfunction. The broad research area of *formal methods* deals with the development of mathematically rigorous techniques for the specification of desirable properties of programs, and their verification. Among the set of desirable properties of a program, the most decisive ones are those related with its correctness, which ensures that a program does what the programmer expects it to do. This class of properties is commonly known as *functional properties*.

Without regard to the correctness of the results of a program, there are some other desirable properties that are relevant to the safety of software systems. These are called *non-functional properties*. An example is the fact that a program performs its task in a given amount of time, or that its memory needs do not exceed a given limit. These two examples become part of a broader research field, whose name is *resource analysis.* In this framework, a program is conceived as a resource consumer (resource may be understood as time, memory, energy, etc.) and the aim is to compute an upper bound to the resources being consumed by every possible execution of the program. In this work, we are particularly interested in the analysis of *memory bounds*. Memory consumption is specially relevant to several scenarios: for instance, when programming embedded devices, it is necessary to make sure that the programs running in these devices do not stop working because they try to use more memory than it is available. It is also useful to know in advance how much memory would be needed by the program, in order to reduce hardware and energy costs.

We believe that the choice of the programming paradigm plays a special role in the reliability of software systems: the friendliest languages (from the programmer's point of view) are those that include concepts from the problem to be solved, rather than those of the machine in which that problem

is solved. *Declarative paradigms* fall into this category, since they emphasize the specification of the computations, instead of how these computations are carried out. *Functional programming* belongs to this class of paradigms. Functional programs are based on the mathematical concept of a function, defined as a set of equations. This paradigm abstracts the way in which these functions are evaluated. The main advantage of pure functional programming is the absence of side effects, from which the property of *referential transparency* follows: a function is defined exclusively by a correspondence between its input and its output (in other words, the result of a function solely depends on its input parameters). This implies that, in a pure functional setting, an expression is always evaluated to the same value, without regard to the execution context where the evaluation takes place.

The absence of the notion of a state makes the task of reasoning about the functional properties of a program easier. As a consequence, functional languages are, in general, better suited to several analyses, because of this lack of state. However, the inference of memory bounds requires special attention. In most functional programming languages, memory management is delegated to the runtime system, which allocates memory as it is needed by the program, provided there is enough space available. A *garbage collector* is in charge of determining, at runtime, which parts of the memory are no longer needed, and can be safely disposed of. The main advantage of this approach is that programmers do not bother about low-level details on memory management, but there are also some drawbacks. On the one hand, the time delay introduced by garbage collection may prevent a program from providing an answer in a required reaction time, which may be unacceptable in the context of real-time systems. On the other hand, garbage collection makes it difficult to predict at compile time the lifetimes of data structures, specially in those cases where the runtime system does not specify under which conditions a garbage collection takes place. This does not necessarily prevent the development of a static analysis for inferring memory bounds: a liveness analysis can be used to predict the part of the heap that may still be accessed by a given program, whereas the total amount of allocated heap is not relevant, as dead memory will be eventually reclaimed by a garbage collector. However, it is difficult to know the actual allocated memory *at a given execution time*. This is specially inconvenient when several programs are executed in the same device: if one of them runs out of memory, the remaining ones might not activate their garbage collectors unless they also need more memory. In order to compute memory bounds for functional programs, it is desirable to set out a memory management approach different from garbage collection. In the next sections we shall explore some possibilities.

1.1.1 Region-based memory management

In region-based languages, each allocated object is created in a region. Regions are created and disposed of in a stack-like fashion: the last region being created is the first one being destroyed. Allocation and deallocation of regions take place in constant time. The compiler has to determine: (1) when regions are created and destroyed, and (2) in which region objects must be created at runtime. The main improvement of region-based memory management on garbage collection approaches is that they bring forward to compile-time the decision on when a given data structure will be created or destroyed.

Earlier works in this memory management technique were targeted towards imperative languages. These works include that of Ruggieri and Murtagh [102], in which each region is associated with the activation record of a procedure. Each region is created at the beginning of a function call and it is removed when this function call finishes. This mechanism is improved in the seminal work of Tofte and Talpin [115, 116], which provides a type-based approach to region inference in a functional language (ML), including higher-order functions, and region-polymorphic recursion. This work have been im-

plemented in the MLKit Compiler [113]. Regions have also been applied to object-oriented languages [28, 27]. Moreover, the Real Time Specification for Java [22] includes a model for providing memory areas that are managed using regions, and are not subject to garbage collection.

In absence of garbage collection, region-based memory management is well-suited to the automatic inference of memory bounds, since the allocation/deallocation of regions is determined at compile-time, as well as the regions in which the different data structures are located. Regions are also time efficient [17], since both allocation and deallocation take place in constant time. However, a major disadvantage of regions is their strict nested discipline. Regions are created and destroyed in a LIFO basis, which does not always correspond to the lifetimes of data structures. As described by Berger, Zorn, and McKinley [17]:

> Regions provide high-performance but force the programmer to retain all memory associated with a region until the last object in the region dies [...]. We show that the performance gains of regions (up to 44%) can come at the expense of excessive memory retention (up to 230%). More importantly, the inability to free individual objects within regions greatly complicates the programming of server applications like Apache which rely on regions to avoid resource leaks.

There exist many approaches for breaking this strict nested discipline of regions: region resetting [19], dissociating region allocation/deallocation from its definition [2], annotating the program with primitives for allocating/deallocating regions [54, 21], and linear regions [45], among others. These approaches require, to a greater or lesser extent, a deep knowledge about how regions are inferred by the compiler, and some of them require help from the programmer in the form of annotations. Alternatively, regions can be combined with garbage collection in order to reduce the impact of memory leaks [114].

1.1.2 Explicit cell deallocation

Another alternative to garbage collectors, in contrast to the fully automatic approach of region-based memory management, is to delegate the task of deallocating memory to the programmer. This approach can be mainly found in some imperative languages (being C and C++ well-known examples of this), but it is very rare in declarative languages. In presence of explicit deallocation, the compiler always knows when data structures are allocated or deallocated at runtime, since this information is contained within the source code. This makes the task of analysing memory bounds easier than in presence of garbage collection.

However, explicit deallocation is an error-prone approach. The execution of a program may result in abortion if it tries to access to those parts of memory which have been previously disposed of. These parts of memory are commonly known as *dangling pointers*. These kind of accesses to dead memory may arise as a consequence of unexpected sharing between the different data structures. A program is said to be *pointer-safe* if dead memory is never accessed during its execution. Moreover, explicit deallocation-based approaches are not exempt from memory leaks (i.e. unused memory that is not released by the program), since the programmer may forget to release the data structures involved in the program.

There exist several type-based techniques for avoiding dangling pointers in a functional setting. In [57] Hofmann describes LFPL, a linearly-typed [118] functional language that allows the in-place modification of heap-allocated data structures. In this language, the positions of the heap are made explicit in the program. This approach is refined in [10, 11], by making a distinction between three

different usage aspects. This scheme, while still being safe, allows more pointer-safe programs than its predecessor.

In the context of imperative languages, *separation logic* [100] provides an extension of Hoare logic, which allows specifying properties about programs which manage shared mutable data structures. These ideas have been applied to Space Invader [24], which uses abstract interpretation-based techniques for proving assertions on separation logic. A combination of separation logic with the work on LFPL is developed by Konečný in [69, 67], which distinguishes different layers in the data structures, and infers separation assertions between layers.

All these systems can prove the absence of dangling pointers, but they cannot prove their presence. This means that there may be pointer-safe programs being rejected by the compiler, since the decision problem of determining whether a pointer is dangling or not is undecidable. Current analyses can only compute an approximation of this property. In case a pointer-safe program is (wrongly) considered unsafe by a static analysis, the corresponding implementation may proceed in two ways: either it warns the programmer about a potential dangling pointer, but allows the execution of the program, or it aborts the generation of the object code, so ill-typed programs cannot be executed. In the first case, the pointer-safety property of the program is no longer guaranteed. The second possibility, which makes more sense in the context of safety-critical systems, the programmer is forced to modify his/her program in order to remove those deallocations that make the program be rejected by the analysis. This may result in memory leaks. Nevertheless, and as in the case of region-based memory management, explicit destruction can be combined with garbage collection-related approaches.

1.1.3 *Safe*'s approach to memory management

In those scenarios where pointer-safety is a must, both of the mechanisms shown in previous sections are able to infer an approximation of this property. This approximation is done in such a way that pointer safety is guaranteed, although some pointer-safe programs are rejected. Over-approximation is inherent to the inference of undecidable properties. In the context of pointer-safety, over-approximation in data structure lifetimes may lead to memory leaks.

Safe's memory management is a combination of region-based memory management and explicit deallocation via destructive pattern matching (in the style of Hofmann and Jost's `LF` language [58]) . Pointer-safety is guaranteed by means of a type system. The problem of over-approximating this property is still insurmountable, but the combination of these mechanisms can alleviate its consequences. Explicit deallocation provides a way to selectively destroy data structures inside a region, without having to wait the whole region to be deallocated. In those cases where the compiler forbids the explicit deallocation of some data structure, region-based memory management guarantees that the memory occupied by this data structure will be eventually reclaimed when the deallocation of the corresponding region takes place. As a consequence, each approach provides an escape mechanism to the limitations of the other.

Given the above, we say that *Safe* provides a *semi-explicit approach* to memory management, since it features implicit (i.e. automatically inferred) regions, and explicit deallocation. A drawback of this approach is that the programmer needs some understanding on how regions are created and destroyed, in order to be able to judge the need for explicit deallocation in his/her program. In *Safe* this is alleviated by defining a model of region-based memory management which is simpler than the original system of Tofte and Talpin [116]. There exists a single region per function call, so the programmer only has to think whether a given data structure is local to that function call.

A major advantage of combining regions with explicit deallocation is that a garbage collector is not needed. As a consequence, the lifetime of data structures is completely determined at compile-time[1], so *Safe* programs are amenable to analyses on memory consumption.

1.2 Certification of safety properties

Certifying program properties consists in providing mathematical evidence about them. In a *Proof Carrying Code* (PCC) environment, these proofs should be mechanically checked by an appropriate tool. The research line of PCC started with the seminal work of Necula [88]. The PCC paradigm aims to create an infrastructure in which programs are associated with formal proofs guaranteeing the presence of some desirable properties. These formal proofs make up a *certificate*. The code producer generates both the compiled program and its certificate, and the code consumer has to check that the certificate is consistent with the program being supplied, and that the proofs contained within the certificate are correct. The certificate checking is done in an automatic way, and usually with the help of a proof assistant, such as Isabelle/HOL [93].

From the code producer's point-of-view, there are several approaches to the certificate generation. In the standard PCC approach, the properties are determined and certified at the object-code level. In those cases where the properties being certified are obtained as a result of a static analysis, the latter would have to be carried out at the object-code level. Performing static analysis on object code usually involves a loss of accuracy in the property being determined, since part of the information of the source code is lost when the latter is translated to lower-level code. Another possibility involves determining the certified properties at the source code level, and then generating the certificate at that level. Then, both the code and the certificate are translated into their low-level counterparts, and sent to the code consumer. The main drawback of having low-level certificates is that it requires a huge effort for formalizing the invariants and properties of the elements that take part in the runtime system (stack frames, program counters, a.o.), in contrast to the high-level certificates, whose properties are proved with respect to the higher-level language semantics, much simpler. Moreover, low-level certificates are of considerable size, since they include the proofs of these additional properties and invariants, and this results in longer checking times.

In *Safe*, certificates are generated at the source-code level, and they are sent, together with the source code, to the code consumer. The latter checks the validity of the certificate, and then performs the translation of the source code into its lower-level counterpart. This translation is certified [39, 38], so it guarantees the preservation of the properties being certified, when translating from higher-level code to low-level code. A notable advantage of this approach is that the translation has to be certified only once by the code consumer. Once done, the certified compiler is valid for every program. This has the additional advantage that certificates are of shorter size, since they only include the program-specific proofs. This comes at the cost of supplying the source code to the consumer, which might not be an option in some PCC scenarios.

Among the first publications devoted to certifying compilers we have that of [89, 91, 30], in which the authors develop a certifying compiler for a subset of Java bytecode. The *MRG* (*Mobile Resource Guarantees*) project [103] aims at the generation of mobile code with a certificate proving bounded heap memory consumption. Certificates are generated at the level of a Java bytecode-alike language called *Grail* [9]. More recently, the *Mobius* (*Mobility, Ubiquity, and Security*) project [14] aims to extend the PCC

[1]The lifetime of a data structure is determined by both the programmer (when using explicit deallocation), and the compiler (when assigning regions to data structures).

paradigm to more general scenarios, such as distributed and concurrent systems, resource analysis, security attacks, etc. It is targeted toward mobile devices satisfying the MIDP (*Mobile Information Device Profile*) specification of Java ME.

1.3 Memory consumption analysis

Although relatively new, the field of resource analysis has gained considerable attention in the last years, mainly due to the application of mathematical techniques (such as linear programming, or recurrence solving) to programming languages. In particular, the inference of memory bounds is a very complex task that involves several auxiliary analyses, each one a challenge by itself. The first results on memory consumption analysis were targeted towards the functional programming paradigm. The techniques developed were subsequently adapted to mainstream languages, such as Java or C++.

Hughes and Pareto introduce in [61] a first-order functional language with a type and effect system guaranteeing termination and execution in bounded space. This system is a combination of Tofte and Talpin's approach to regions with sized types [62, 96].

The first fully automatic way to infer closed-form memory bounds is due to Hofmann and Jost [58]. Their analysis, based on a type system with resource annotations, can infer linear heap memory bounds on first-order functional programs with explicit deallocation. These techniques have been applied to subsets of imperative languages, such as Java [59] (without type inference), and C [48] (for computing worst-case execution time). The annotated type system also serves as a basis for a stack consumption analysis due to Campbell [25, 26]. Hofmann and Jost's approach is extended in [64, 63] to higher-order programs. The latter work provides a general framework that can accommodate different notions of cost. More recently, Hoffmann and Hofmann have extended the initial work of [58] to polynomial memory bounds [56, 55].

The classical approach to resource analysis, due to Wegbreit [119] involves the generation of a recurrence relation from the program being analysed, and, in a second phase, the computation of a closed-form expression (without recursion) equivalent to that recurrence relation. Vasconcelos and Hammond pursue this approach in [117], which is fully automatic in the generation of recurrence equations, but requires the use of an external solver for obtaining a closed form. The COSTA System [5] follows a similar approach, but it provides its own recurrence relation solver, PUBS [4], which can handle multivariate, non-deterministic recurrence relations. COSTA is an abstract interpretation-based analyser which works at the level of Java bytecode, and supports several notions of cost, such as the number of bytecode instructions executed, heap consumption, and number of calls to a particular method. Since memory management in Java is based on garbage collection, their approach to memory consumption is parametric on the behaviour of the garbage collector [7]. The bounds computed by this system go beyond linear expressions; it can compute polynomial, logarithmic, and exponential bounds.

Although not directly aimed at memory bounds, the SPEED project [49] computes non-linear symbolic bounds by combining techniques such as loop transformations, counter instrumentation, and invariant generation.

The problem of inferring memory bounds is closely related to the inference of *size relations* between data structures. The seminal work on type-based sizes is the above mentioned of Hughes, Pareto, and Sabry [62], which is restricted to type checking. The problem of inference is addressed by Chin and Khoo [29]. Another approach involves using abstract interpretation-based techniques on the domain of convex polyhedra, as done by Benoy and King [16]. Abstract interpretation is also applied in [108] to

the approximation of the height[2] of data structures. All the techniques described so far are restricted to infer linear size relations. The work of Shkaravska, van Eekelen and van Kesteren [105] is able to infer polynomial size relations. It provides a type system, in which checking is decidable under certain syntactic conditions. Type inference is performed by a combination of testing and polynomial interpolation-based techniques. More recently, these techniques have been applied to the computation of polynomial loop-bound functions for Java programs [106].

1.4 Goals and structure of the work

When considered as a whole, the *Safe* project aims to study the appropriateness of functional languages to devices with strict memory requirements. The goals of this thesis are intended to get closer to this general objective. These goals are the following:

1. **Develop an efficient, fully automatic, type-based, static analysis for ensuring pointer safety.**

   Given the memory model of *Safe*, informally explained in Section 1.1.3, the analysis must accommodate region-based memory management and explicit destruction. This analysis will be based on a type system, which will be proven correct with respect to the operational semantics of the language.

   With regard to type inference, we will consider regions and explicit destruction separately:

   (a) A region inference algorithm will annotate the source program with region variables, in such a way that the resulting program is well-typed w.r.t. the above mentioned type system. This annotation process will be done in an optimal way, in the sense that every data structure will be placed in the region with the shortest lifetime that does not compromise the pointer safety of this data structure. The algorithm will support polymorphic-recursive functions, and will be proved correct with respect to the type system.

   (b) A mark inference algorithm will ensure that the explicit destruction specified by the programmer does not generate dangling pointers at runtime. In this case, the sharing relations between the different data structures play a fundamental role. The approximation of these relations given by an already existing sharing analysis will serve as a basis for the inference of safe destruction. The inference algorithm will also be proven correct with respect to the type system.

2. **Certify pointer-safety in an automatic way.**

   Assuming that a *Safe* program is well-typed with regard to the type system described above, we will describe how the compiler generates a certificate that can be mechanically checked by the Isabelle proof assistant. This certificate proves the pointer-safety property of the program being compiled. The process of certificate generation can be split into two tasks:

   (a) The first task involves proving that well-typed programs do not access dangling pointers. This tasks involves a formalization of the type system's correctness in a proof assistant. The corresponding theorems are generic, in the sense that they are proved once for all, and reused in every certificate.

[2] The height of a data structure is the longest chain of pointers that can be followed from a given one.

(b) The second task involves proving that the specific program being certified is well-typed. By combining this part with the results of the first task, the certificate proves that the program being certified does not access dangling pointers.

In this thesis we shall deal only with the latter point: the generation of the program-specific part of the certificate. The details of the generic part can be found in [37, 35].

3. **Develop a memory cost model for *Safe* programs.**

   A memory cost model provides a way to formally describe the memory needs of each syntactical construction of the language. Our cost model will be included in the *Safe* big-step operational semantics, and it will take both heap and stack memory needs into account. This model will be formally derived from the translation of *Safe* source code to the code of the imperative SVM (*Safe* virtual machine), and it will be proven correct with respect to this translation.

4. **Develop an abstract interpretation-based analysis for inferring heap and stack memory bounds.**

   Given the cost model of the previous point, we aim to use abstract interpretation-based techniques [33] for inferring non-linear, monotonic, closed-form expressions bounding the heap and stack memory costs of a program. The resulting bounds will be functions on the sizes of the program's input.

   Even if we restrict ourselves to a first-order functional language, like *Safe*, the inference of safe memory bounds is a very complex task, which involves considering several preliminary results, such as size analysis, and call-tree size analysis. Each one of these analysis could be, by itself, a subject of a PhD thesis. Therefore, we will focus on the inference of heap and stack memory bounds by assuming that the size and call-tree information is given externally.

   We will formally prove the correctness of the results of the analysis with respect to the memory cost model. We will also specify certain conditions on the externally-given call tree information, which allows to prove the *reductivity* of the resulting bounds. The advantage of having reductive bounds is that they will allow us to obtain a decreasing sequence of bounds, as a result of iterating the analysis several times for the input program.

In this work we have put emphasis on the formal correctness of each of the analyses. We believe that, in general, formal proofs play an important role when developing languages devoted to safety-critical systems, but, in particular, they are very relevant to the *Safe* project, since its goals include the formal certification of the properties being proved. In this work we only deal with the certification of pointer-safety properties, but much work has also been done in *Safe* for certifying memory bounds [40, 35].

The structure of the thesis is as follows:

- In Chapter 2 we introduce, through several examples, the basic concepts of the *Safe* language. Then we formalize the operational semantics of the language, which serve as a basis for deriving and abstract virtual machine for *Safe* programs (SVM). The translation of *Safe* into instructions of SVM will allow us to derive a memory cost model, whose correctness will be proved with respect to the translation. With this cost model we will have achieved the third goal shown above. This chapter is an extended version of the work described in [82, 86].

- Chapter 3 presents a type system guaranteeing that regions and explicit destruction are done in a safe way. There we assume that the source program is already annotated with regions, and show that well-typed programs do not access dangling pointers. The problem of inferring correct

typing derivations is deferred to the next two chapters. This chapter is an improved version of the work in [84].

- Chapter 4 addresses the problem of region inference. Given a *Safe* program, the algorithm decorates its abstract syntax tree with region variables, in such a way that the resulting program is well-typed. The correctness and optimality of the algorithm are formally proved. Our algorithm has been published in [85].

- Chapter 5 deals with explicit destruction. In this chapter we develop an algorithm guaranteeing the safety of the explicit deallocations specified by the programmer. It is proved correct and complete with respect the type system of Chapter 3. This chapter is an extended version of the publication [81].

- The work of Chapters 3, 4, and 5 is aimed at obtaining typing derivations for a *Safe* program (first goal). Once these derivations have been obtained, Chapter 6 explains how to represent them formally as Isabelle/HOL proof scripts, in order to certify pointer-safety (second goal). It corresponds to the publication [36].

- In Chapter 7 we develop an abstract interpretation-based analysis for inferring heap and stack memory bounds for *Safe* programs (fourth goal). Firstly we define the abstract domain, and the abstraction function which maps program expressions to elements of the abstract domain. This function is proved correct with respect to the cost model of Chapter 2. This chapter extends our previous work on space consumption analysis [83].

- Finally, in Chapter 8 we discuss the obtained results, and draft some future research directions.

1.4.1 Notation and basic assumptions

We denote by $\mathbb{N}$ the set of natural numbers (including zero), whereas $\mathbb{R}^+$ denotes the set of non-negative real numbers.

The notation $\overline{x_i}^n$ abbreviates the sequence $x_1\ x_2 \ldots x_n$. More generally, if $e(i)$ is an expression which depends on the i variable, $\overline{e(i)}^n$ denotes the sequence $e(1)\ e(2) \ldots e(n)$. With this sequence notation, we normally use i and j to denote the bound variable that ranges from 1 to n. When this notation occurs nested, i stands from the bound variable of the outermost sequence, whereas j is the bound variable of the innermost one. For instance, the abbreviation **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n$ denotes the following expression:

$$\begin{array}{lrcl} \textbf{case}\ x\ \textbf{of} & C_1\ x_{11}\ x_{12} \ldots x_{1,n_1} & \to & e_1 \\ & C_2\ x_{21}\ x_{22} \ldots x_{2,n_2} & \to & e_2 \\ & \vdots & & \vdots \\ & C_n\ x_{n1}\ x_{n2} \ldots x_{n,n_n} & \to & e_n \end{array}$$

The superscript denoting the number of elements will be left out when it is not relevant (for instance, as in $\overline{x_i}$). The sequence notation can also be applied when defining a set with a finite number of elements. For instance, $\{\overline{\rho_i}^m\}$ denotes the set $\{\rho_1, \ldots, \rho_m\}$.

If A and B are sets, we use $A \to B$ to denote the set of *partial* functions (or *partial mappings*) which map elements from A to elements of B. By abuse of terminology, these partial functions will be denoted simply functions, or mappings. If f belongs to $A \to B$, the domain of f is denoted by dom f. By convention, the application of a function $f(x)$ implicitly assumes that $x \in$ dom f. Our partial functions

may be considered as a set of bindings of the form $x \mapsto y$, where $x \in A$ and $y \in B$, provided that no element of A appears bound to two different values in the same function. This allows us to use the usual notation of sets with mappings. For instance:

- $\Gamma_1 \subseteq \Gamma_2$ denotes that $\text{dom}\ \Gamma_1 \subseteq \text{dom}\ \Gamma_2$, and that $\forall x \in \text{dom}\ \Gamma_1 . \Gamma_1(x) = \Gamma_2(x)$.
- $[\overline{a_i \mapsto b_i}^n]$ abbreviates the function $[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$.
- $[x \mapsto 0 \mid x \in D]$ is the function which maps every element of D to 0.
- If $f, g \in A \to B$ and $\text{dom}\ f \cap \text{dom}\ g = \emptyset$, the function $f \uplus g$ has $(\text{dom}\ f \cup \text{dom}\ g)$ as its domain, and is defined as follows:
$$(f \uplus g)(x) = \begin{cases} f(x) & \text{if } x \in \text{dom}\ f \\ g(x) & \text{if } x \in \text{dom}\ g \end{cases}$$
- The notation $\Gamma \backslash x$ denotes the restriction of Γ to the set $(\text{dom}\ \Gamma) \backslash \{x\}$.

In the context of type environments, we use the more familiar notation $x_i : y_i$ for denoting their bindings, and we use $+$ instead of $\uplus$ to denote the disjoint union of two environments.

Remark. Deviating from other authors, the notation $f[x \mapsto y]$ specifies that the binding $x \mapsto y$ *already belongs to* f. On the contrary, $f \uplus [x \mapsto y]$ denotes the function which maps x to y, and behaves likes f for the rest of elements in its domain.

Chapter 2

Syntax and resource-aware semantics of *Safe*

2.1 Introduction

As it was pointed up in the last chapter, a novel feature in *Safe* is the combination of two mechanisms for replacing garbage collection: region-based memory management and destructive pattern matching. In this chapter we will make this idea more precise. We start with an informal description of these mechanisms from a programmer's point of view. Then we formally define the syntax and semantics of the language, and in a series of successive formal steps we derive an abstract virtual machine for executing *Safe* programs. This allows us to obtain a *memory model*, which will be useful when inferring upper-bounds to memory consumption (Chapter 7). This chapter provides the theoretical foundations which will serve as a basis for the correctness results of next chapters.

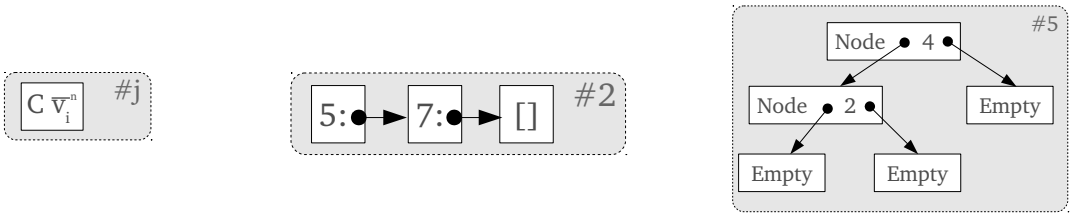
This chapter is an extended version of the work described in [82]. It also includes some implementation aspects of the *Safe* runtime system and its compiler. These were taken from [80]. Firstly we present in Section 2.2 a high-level view of the language *Safe*, and how its memory facilities are implemented. After this, a desugared variant of *Safe* is formally defined via its syntax (Section 2.3) and semantics (Section 2.4). In Section 2.5 we present the abstract machine in which *Safe* programs are run and the translation process between *Safe* programs and the code being executed by the abstract machine (Section 2.6). The abstract machine will serve as a reference implementation which will allow us to determine the memory consumption of a *Safe* program. This will be made apparent by enriching our semantics with a resource vector specifying how much memory is needed by a program in order to be executed (Section 2.7). Finally, Section 2.9 gives a broad overview of the *Safe* compiler's implementation and Section 2.10 concludes.

2.2 Language concepts: *Safe* by example

Safe is a first-order polymorphic functional language, whose syntax is similar to that of (first-order) Haskell or ML, but with some facilities to manage memory. Polymorphic data types are defined in the same way as in Haskell. As an example, we have the following **data** declarations of binary search trees and lists:

$$\begin{array}{lcl} \textbf{data}\ \mathit{BSTree}\ \alpha & = & \mathit{Empty} \mid \mathit{Node}\ (\mathit{BSTree}\ \alpha)\ \alpha\ (\mathit{BSTree}\ \alpha) \\ \textbf{data}\ [\alpha] & = & [\,] \mid (\alpha : [\alpha]) \end{array}$$

Functions are defined as a set of equations with the same syntax as Haskell functions. For example,



(a) Cell (b) List (c) Binary tree

Figure 2.1: Graphical representation of cells and regions.

the following definitions compute the length of a list and the number of nodes in a tree:

$$
\begin{array}{lll}
\mathit{length} & :: [\alpha] \to \mathit{Int} & \\
\mathit{length} & [\,] & = 0 \\
\mathit{length} & (x : xs) & = 1 + \mathit{length}\ xs
\end{array}
$$

$$
\begin{array}{lll}
\mathit{size} & :: \mathit{BSTree}\ \alpha \to \mathit{Int} & \\
\mathit{size} & \mathit{Empty} & = 0 \\
\mathit{size} & (\mathit{Node}\ i\ x\ d) & = 1 + \mathit{size}\ i + \mathit{size}\ d
\end{array}
$$

Safe's memory model is based on *heap regions*. Regions are disjoint parts of the heap where data structures are built. A region can be created and disposed of in constant time.

A *cell* is a piece of memory big enough to hold a data constructor with its parameters. In implementation terms, a cell contains the identifier of a data constructor, and a representation of the values to which this constructor is applied. These values can be either basic (integers or booleans), or pointers to other cells. With the term "big enough" we mean that a cell being disposed of the heap may be immediately reused by the runtime system. A naive implementation would define this size as the space taken by the biggest constructor (i.e. with the highest number of parameters). In a more efficient approach we would have a fixed number of cell sizes, all of them multiples of the smallest one. In any case, the aim is to be able to reuse a cell in constant time.

We represent regions and cells as in Figure 2.1. A cell is depicted as a white square which contains the constructor and its arguments to which it is applied. Pointers are represented via arrows between cells, whereas basic values are shown in the cell itself. Shaded rectangles correspond to regions, which are labelled with a number identifying them (see Figure 2.1a). As an example, Figure 2.1b shows a list of integers, in which the constructor $(:)$ is shown in infix form.

Cells are combined in order to build *data structures.* A data structure (DS in the following) is the set of cells that results from taking a particular cell (the *root*) and following the transitive closure of the relation $C_1 \to C_2$, which denotes that C_1 and C_2 are cells *of the same type,* and there is a pointer in C_1 to C_2. This relation will be described more precisely in the next chapter, as well as the concept of type. An important thing to note, however, is that we only consider as part of a DS the set of cells with the same type as the root cell. For instance, if we have a list of lists (type $[[\alpha]]$), the cells that make up the recursive spine of the outer list constitute a DS, to which the inner lists do not belong, even when there are pointers from the outer list to them. Each one of the inner lists constitute a separate DS on its own.

During the design of the language several decisions were taken. The first one involves the correspondence between DSs and regions.

Axiom 2.1. *A DS completely resides in a single region.*

This decision poses a constraint to the data constructors: the recursive children of a cell (i.e. those with the same type) must belong to the region of the father.

Axiom 2.2. *A DS can be part of another DS, and two DS may share a third DS.*

As an example, consider the binary tree of Figure 2.1c. The left and the right subtrees of the root are separate DS, which belong to the whole binary tree, which is another DS.

Axiom 2.3. *Basic values (integers and booleans) occurring in the heap do not belong to any region by themselves. They are contained within cells.*

2.2.1 Region-based memory management in *Safe*

A distinctive aspect of *Safe* is the way in which regions are created and destroyed:

Axiom 2.4. *Allocation of regions takes place at function calls. Deallocation of regions takes place when a function call finishes.*

This implies that new regions are created as functions are called, so there exists a correspondence between the function call stack and regions, which are also created and disposed of in a stack-like fashion. Since function calls have nested lifetimes (for instance, if f calls to a function g, the execution of the latter begins after the execution of f has started, and it finishes before the execution f has finished), regions also have nested lifetimes. That is why they are stored in a stack-like fashion: the last region being created is the first being destroyed.

The region associated to a given function call f is its *working region*. The function may create DSs in this region, provided these DS are not accessed outside the function's context, since they will be destroyed when the function finishes. A function may also access the working regions of the function calls situated below it in the call stack. These regions must be passed as parameters by the functions calling f. Each region existing at a given execution point is uniquely identified by a natural number ranging from 0 (which identifies the bottommost region in the stack) to the number k of active regions minus one (which identifies the topmost one).

An important point is the fact that regions are not handled directly by the *Safe* programmer. The compiler determines which DSs will be created in the working region and which regions should be passed as parameters between functions. However, in order to get an idea on how regions are inferred, we will consider a syntactically-extended version of *Safe*, which we call *Safe with regions*. In this version regions become apparent. The main syntactical additions of *Safe* with regions include the following:

- A function definition may have additional region parameters $r_1 \ldots r_m$ separated by a @ from the rest of formal parameters. As an example, we may have the following function definition:

  $$f\ x_1\ x_2\ x_3\ @\ r_1\ r_2 = \ldots$$

  These extra parameters will contain, at runtime, the identifiers of the regions in which f will build its output. These regions are, in general, different from the regions in which the DSs given as input live. However, in those cases when the result is build upon a DS passed as parameter, at least one output region will be the same as the one in which that DS lives. The compiler determines, in each call to f, whether output regions are equal or different from the input regions.

- The working region is referred to by the identifier *self*.
- When calling a function with these extra parameters, the region arguments are also separated from the rest of arguments by the @ symbol. For example:

$$f\ 4\ x\ z\ @\ self\ r_1$$

  where r_1 is a region variable in scope.

- Each constructor expression is attached a region variable which contains, at runtime, the identifier of the region where the resulting cell will be built. For example:

$$[\,] @ r_2 \qquad (4\ :\ [\,] @ self\ ) @ self$$

  In the latter example, the outermost *self* annotates the application of the list constructor $(:)$.

Example 2.5. Consider a function *append* for concatenating two lists. The following is *Safe* code, as written by the programmer:

$$\begin{array}{llll} append & :: [\alpha] \to [\alpha] \to [\alpha] & & \\ append & [\,] \quad\ ys & = & ys \\ append & (x : xs) \quad ys & = & x : append\ xs\ ys \end{array}$$

This function is annotated by the compiler as follows:

$$\begin{array}{llll} append & [\,] \quad\ ys\ @\ r & = & ys \\ append & (x : xs) \quad ys\ @\ r & = & (x\ :\ append\ xs\ ys\ @\ r)\ @\ r \end{array}$$

There is a new region parameter r, which is used to build the resulting list, and is passed to the subsequent recursive calls. Assume this function is called in the context of a function f:

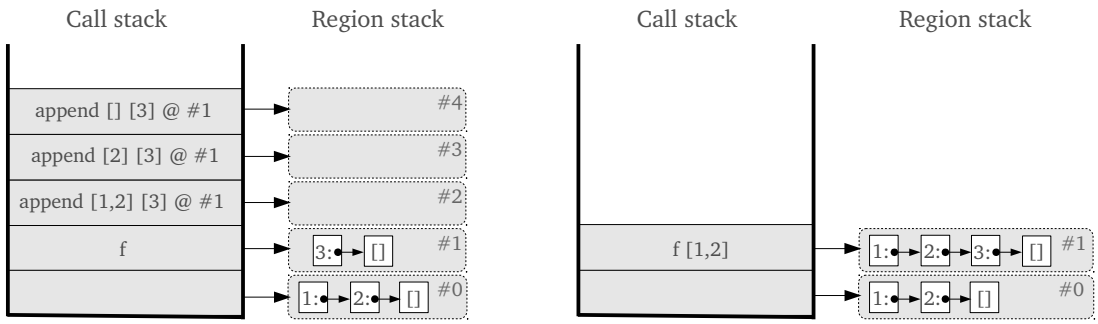
$$f\ xs = \ldots\ append\ xs\ [3]\ \ldots$$

and that its definition is annotated as follows:

$$f\ xs = \ldots\ append\ xs\ ((3 : [\,] @ self) @ self)\ @\ self\ \ldots$$

Figure 2.2a shows the correspondence between call and region stacks when evaluating the call $f\ [1,2]$. Assuming that the identifier of the working region corresponding to this call is number 1, Figure 2.2a shows the regions created when the execution reaches the base case of *append*. This function receives the region identifier bound to the *self* region of f, that is, 1. Each recursive call to *append* has its own working region (regions 2, 3 and 4), but nothing is built there, since the result is built in region 1, which is the value of the region parameter being propagated through the recursive calls to *append*. When the initial call to *append* finishes, we get the situation of Figure 2.2b. Notice that the result is forced to be built in the same region as the list passed as second parameter. This is because this parameter is reused in the base case of *append*, and a DS must be contained within a single region. The compiler takes this into account when annotating the call to *append*. □

The working region *self* of a function is used to build temporary DSs which are not part of the result. An example of a function with this kind of behaviour is *treesort*.



(a) Base case of *append*

(b) After *append* finishes

Figure 2.2: Call and region stacks of the *append* function.

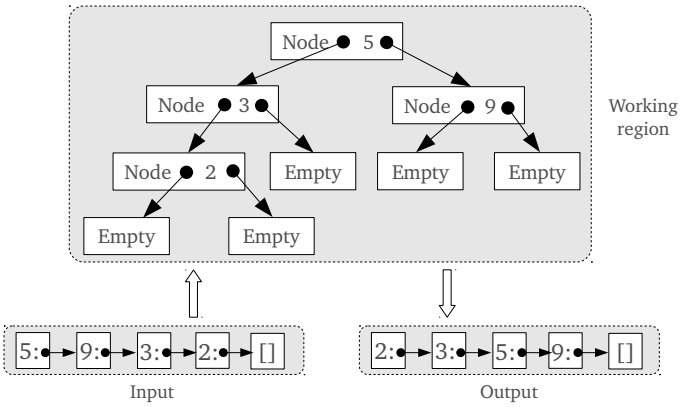


Figure 2.3: DSs involved in the *treesort* function. The working region contains the intermediate representation of the list as a binary tree.

Example 2.6. A tree sort algorithm builds a binary search tree from the input list to be sorted. Then it does an inorder traversal of the tree, so that the elements come out in sorted order. Figure 4.12 on page 154 shows the full *Safe* code with regions. For the purposes of this example, the *treesort* function is enough:

$$\begin{aligned} &\mathit{treesort} \;::\; [\alpha] \to [\alpha] \\ &\mathit{treesort}\ \mathit{xs} = \mathit{inorder}\ (\mathit{mkTree}\ \mathit{xs}) \end{aligned}$$

where *mkTree* builds a binary search tree from the list given as parameter, and *inorder* performs an inorder traversal of a binary search tree by adding the visited elements to a list that is returned as result. Now we show the *Safe* code with regions:

$$\mathit{treesort}\ \mathit{xs}\ @\ r = \mathit{inorder}\ (\mathit{mkTree}\ \mathit{xs}\ @\ \mathit{self})\ @\ r$$

Both functions *inorder* and *mkTree* receive a region parameter specifying where to build the resulting list (resp. tree). The *mkTree* function is given the *self* identifier, so the tree will be built in the working region. The *inorder* function receives the parameter given to *treesort*, which is the output region in which the sorted list will be built (Figure 2.3). When *treesort* finishes its working region will disappear from the heap, together with the temporary tree. □

Safe provides a built-in facility for copying data structures: the @ notation. The expression *ys*@ returns a copy of the DS pointed to by *ys*. The copy of the data structure will be located in a (possibly) different region, if this does not contradict Axiom 2.1. The copy facility is useful when the programmer does not want to build DS upon already existing ones.

Example 2.7. The *append* function of Example 2.5 forces the resulting list to be located in the same region as the list passed as second parameter. This is because the result is linked to this parameter, so that the latter becomes part of the former, and, by Axiom 2.1, they must live in the same region. Let us consider the following variant in which the result is built upon a *copy* of the list passed as second parameter:

$$\begin{array}{lllcl} \mathit{appendC} & [\,] & \mathit{ys} & = & \mathit{ys}@ \\ \mathit{appendC} & (x : \mathit{xs}) & \mathit{ys} & = & x : \mathit{appendC}\ \mathit{xs}\ \mathit{ys} \end{array}$$

The compiler annotates every copy expression with the region variable in which the copy will be returned. In the case of *appendC* function, it produces the following code with regions:

$$\begin{array}{lllcl} \mathit{appendC} & [\,] & \mathit{ys}\ @\ r & = & \mathit{ys}\ @\ r \\ \mathit{appendC} & (x : \mathit{xs}) & \mathit{ys}\ @\ r & = & (x\ :\ \mathit{appendC}\ \mathit{xs}\ \mathit{ys}\ @\ r)\ @\ r \end{array}$$

The copy of *ys* is created in the output region r, which may now be different from the region of the second parameter *ys*. (Figure 2.4). □

Example 2.8. The copy of a DS might not be able to live in a region different from that of the DS being copied. In the following example:

$$\mathit{duplicate}\ t = \mathit{Node}\ t\ 0\ (t@)$$

The original binary tree t and its copy t@ are forced to live in the same region, as they belong to the same DS. □

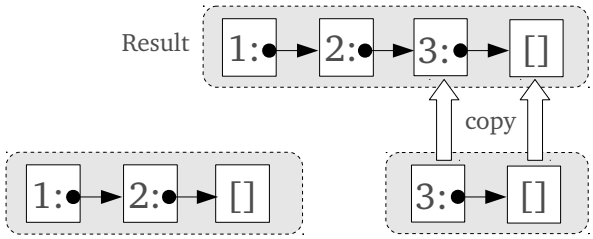


Figure 2.4: Runtime behaviour of *appendC* [1, 2] [3]: the list passed as second parameter is copied, so it does not share cells with the result, with may be built in an independent region.

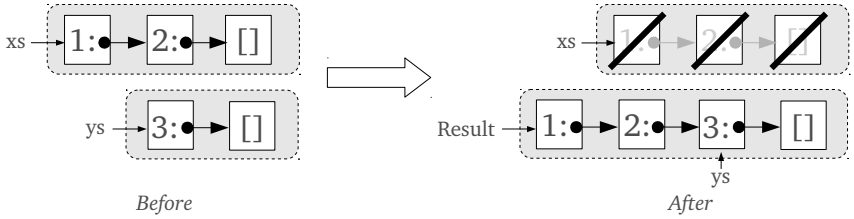


Figure 2.5: Regions of the input parameters before and after a call to *appendD* [1, 2] [3]. Every cell of the input list is removed in each recursive call to *appendD*.

2.2.2 Destructive pattern matching

Destructive pattern matching allows the selective disposal of a DS inside a region, without the need of waiting the whole region to be disposed of. This allows the programmer to break the strict nested discipline of nested regions, an issue which was discussed in Section 1.1.1.

Destructive pattern matching, denoted by (!) or a **case**! expression, deallocates the cell corresponding to the outermost constructor of the DS being matched against. In this case we say that the DS involved in the destructive pattern matching is *condemned*.

Axiom 2.9. *Every function has the following capabilities over their input data structures:*

- *A function may only read a DS which is not a condemned parameter.*
- *A function may read (before destroying) and destroy a DS which is a condemned parameter.*

As an example, we will consider a destructive variant of *append*.

Example 2.10. Assume the following definition:

$$\begin{array}{llcl} appendD & [\,]! & ys & = & ys \\ appendD & (x : xs)! & ys & = & x : appendD\ xs\ ys \end{array}$$

The (!) mark in the first parameter specify that the cell to which the pattern matching is done will be destroyed at runtime. The function destroys the first cons cell of the list passed as first parameter. The remaining cells will be destroyed in the subsequent recursive calls to *appendD*, until we reach the base case in which the empty list matches the first equation, and it is also destroyed (Figure 2.5).

This version of *append* needs no additional heap space: a cell is destroyed and another cell is built in each recursive call. The destruction of the first parameter is reflected in the type of the function:

$$appendD :: [\alpha]! \to [\alpha] \to [\alpha]$$

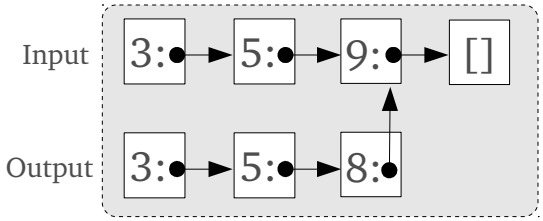


Figure 2.6: Inserting the number 8 in the list $[3,5,9]$. The part of the list before the new cell must be reconstructed.

□

Destructive pattern matching allows the programmer to define functions requiring constant additional heap space. (see Chapter 3 for more examples). This facility is also very useful for breaking the restriction (imposed by regions) of having DSs with nested lifetimes.

Example 2.11. Given a list of elements, the *insertion sort* algorithm starts with an empty list and does successive ordered insertions in it with the elements of the input list. The function for inserting an element in an ordered list is defined as follows:

$$
\begin{aligned}
&\mathit{insert}\ x\ [\,] = [x] \\
&\mathit{insert}\ x\ (y : ys) \\
&\qquad\qquad \mid x \leq y = x : y : ys \\
&\qquad\qquad \mid x > y = y : \mathit{insert}\ x\ ys
\end{aligned}
$$

The compiler produces the following region-annotated version:

$$
\begin{aligned}
&\mathit{insert}\ x\ [\,] @ r = (x : [\,] @ r) @ r \\
&\mathit{insert}\ x\ (y : ys) @ r \\
&\qquad\qquad \mid x \leq y = (x : (y : ys) @ r) @ r \\
&\qquad\qquad \mid x > y = (y : \mathit{insert}\ x\ ys @ r) @ r
\end{aligned}
$$

In the equation guarded by $x \leq y$ the result is built upon the input ys, so the region of the output list must be the same as that of the input list (Figure 2.6). As a result, the following *inssort* function,

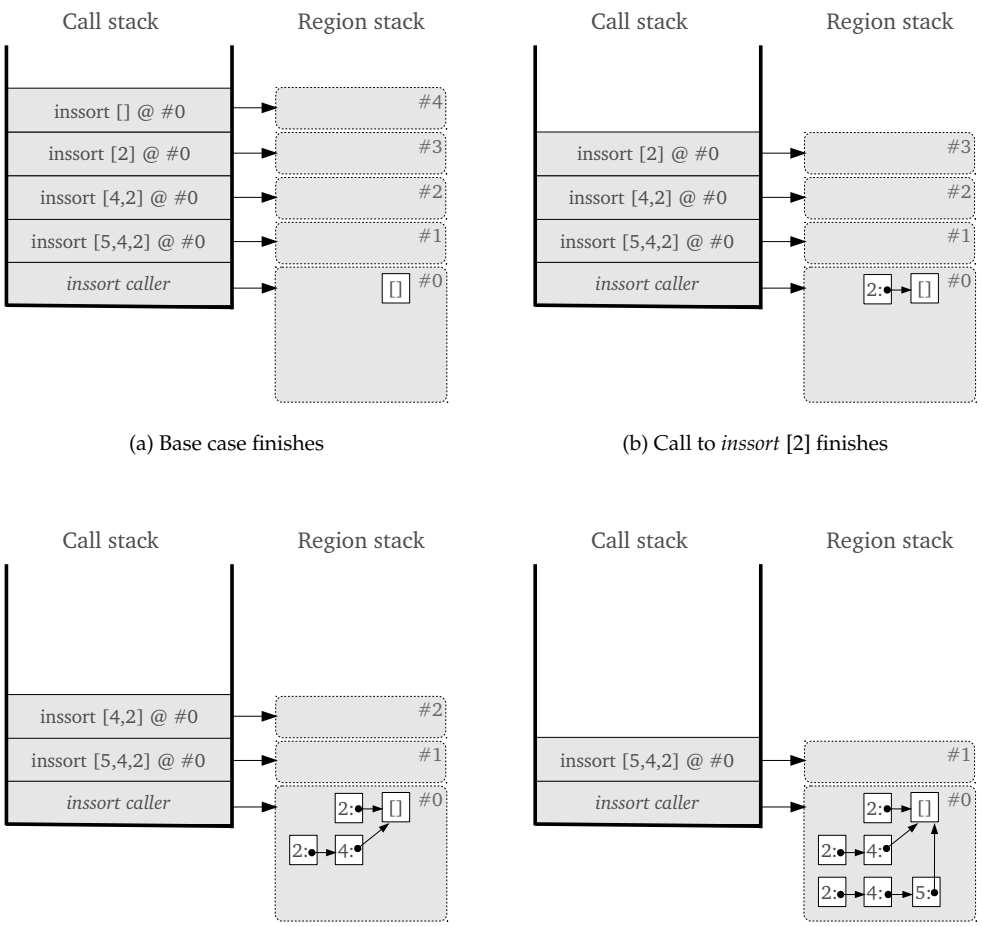
$$
\begin{aligned}
&\mathit{inssort}\quad [\,] &&= [\,] \\
&\mathit{inssort}\quad (x : xs) &&= \mathit{insert}\ x\ (\mathit{inssort}\ xs)
\end{aligned}
$$

whose regions are inferred by the compiler in the following way,

$$
\begin{aligned}
&\mathit{inssort}\quad [\,] \quad @\ r &&= [\,] @ r \\
&\mathit{inssort}\quad (x : xs) \quad @\ r &&= \mathit{insert}\ x\ (\mathit{inssort}\ xs @ r) @ r
\end{aligned}
$$

builds every intermediate result in the region of the empty list being created in the base case, that is, the output region r, whereas the working regions of the calls to *inssort* remain unused. This implies that the *inssort* function has a $\mathcal{O}(n^2)$ worst-case space complexity, where n is the number of elements of the input list (Figure 2.7).

We could return a copy of ys in the *insert* function. In this way the result is built upon that copy,



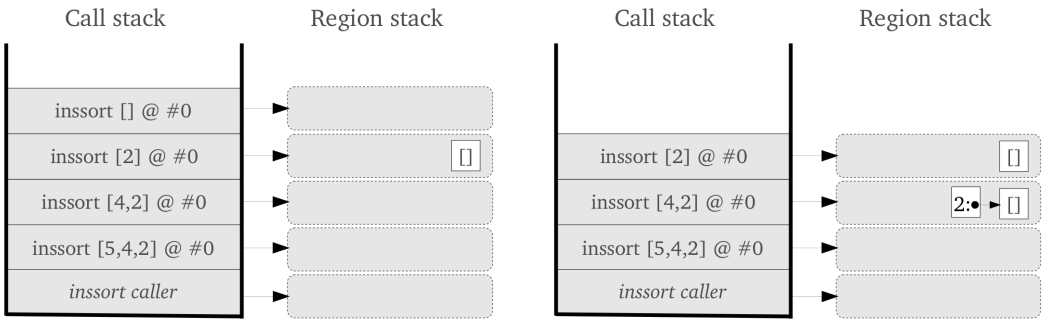
(a) Base case finishes

(b) Call to *inssort* [2] finishes

(c) Call to *inssort* [4,2] finishes

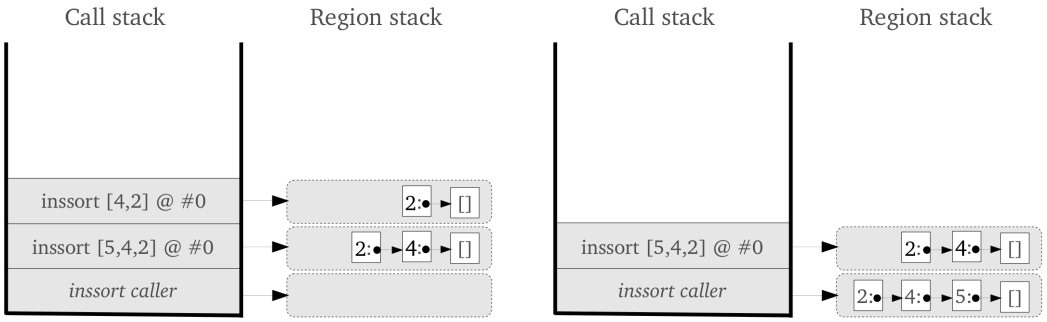
(d) Call to *inssort* [5,4,2] finishes

Figure 2.7: Memory consumption of the *inssort* function.



(a) Base case finishes

(b) Call to *inssort* [2] finishes



(c) Call to *inssort* [4,2] finishes

(d) Call to *inssort* [5,4,2] finishes

Figure 2.8: When doing a copy of the input list before inserting, *inssort* becomes $\mathcal{O}(n)$ space.

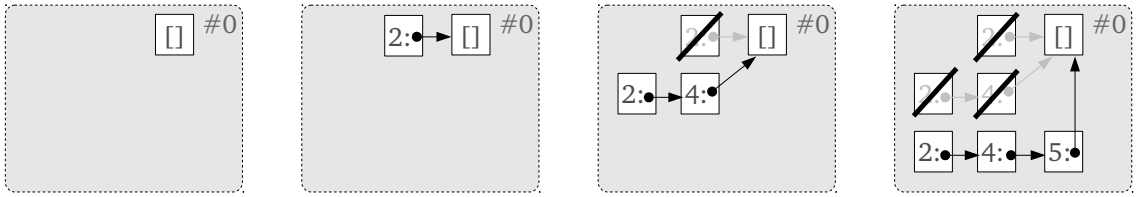
which does not necessarily live in the same region as the input list. This will allow the function *inssort* to build the intermediate lists in the working region *self*:

$$\begin{array}{llll} \mathit{inssort} & [\,] & @\, r & = & [\,] @ r \\ \mathit{inssort} & (x : xs) & @\, r & = & \mathit{insert}\ x\ (\mathit{inssort}\ xs\ @\ \mathit{self})\ @\ r \end{array}$$

This results in an algorithm of $\mathcal{O}(n)$ space complexity (see Figure 2.8). However, it could not be trivial for the programmer to discover that doing a copy may result in avoiding such memory leaks. A more direct approach is to consider a destructive version of *insert*:

$$\begin{array}{l} \mathit{insertD}\ x\ [\,]! = [x] \\ \mathit{insertD}\ x\ (y : ys)! \\ \qquad \mid x \leq y = x : (y : ys) \\ \qquad \mid x > y = y : \mathit{insertD}\ x\ ys \end{array}$$

This version only needs an additional cell in memory to build the new list node. Assuming that we replace the call to *insert* by *insertD* in the *inssort* function, Figure 2.9 shows the state of the output region when returning from every recursive call. We can even develop a destructive version of *inssort* that also



(a) Base case finishes (b) Call to *inssort* [2] finishes (c) Call to *inssort* [4,2] finishes (d) Call to *inssort* [5,4,2] finishes

Figure 2.9: Insertion sort destroying the intermediate results in each call to *insert*

```
void  PushRegion ()      -- creates a top empty region
void  PopRegion ()       -- removes the topmost region
cell  ReserveCell ()     -- returns a fresh cell
void  InsertCell (p, j)  -- inserts cell pinto region j
void  ReleaseCell (p)    -- releases cell p
```

Figure 2.10: The interface of the *Safe* Memory Management System.

disposes of the input list.

$$\begin{array}{llll} \mathit{inssortD} & [\,]! \;\; @\, r & = & [\,] \;@\, r \\ \mathit{inssortD} & (x : xs)! \;\; @\, r & = & \mathit{insertD}\; x\; (\mathit{inssortD}\; xs \;@\; r) \;@\; r \end{array}$$

The cell of the input list being destroyed in the pattern matching can be reused by the *insertD* function. Thus *inssortD* needs no additional heap space for building the result.

□

2.2.3 Runtime system implementation

As we said above, the heap is implemented as a stack of regions. Each region is pushed initially empty, this action being associated to a *Safe* function invocation. During function execution new cells can be added to (or removed from) any active region as a consequence of constructor applications and destructive pattern matching. Upon function termination the whole topmost region is deallocated. In Figure 2.10 we show, in a *Java*-like syntax, the main interface between a running *Safe* program and the *Memory Management System* (MMS). The MMS maintains a pool of fresh cells, so that 'allocating' and 'deallocating' a cell respectively mean removing it from, or adding it to the pool.

Notice that access to an arbitrary region is needed in *InsertCell*, whereas *ReleaseCell* is only provided with the cell pointer as an argument. We have implemented all the methods running in constant time by representing the regions and the pool as circular doubly-linked lists. Figure 2.11 shows a picture of the heap. Removing a region amounts to joining two circular lists, which can obviously be done in constant time. The region stack is represented by a static array of dynamic lists, so that constant time access to each region is provided.

2.2.4 *Full-Safe* vs *Core-Safe*

The functions presented in previous sections were written in *Full-Safe*, which is the language in which the programmer writes his programs. However, *Full-Safe* results cumbersome when designing program analyses, since the number of language syntactic constructs to consider becomes overwhelming.

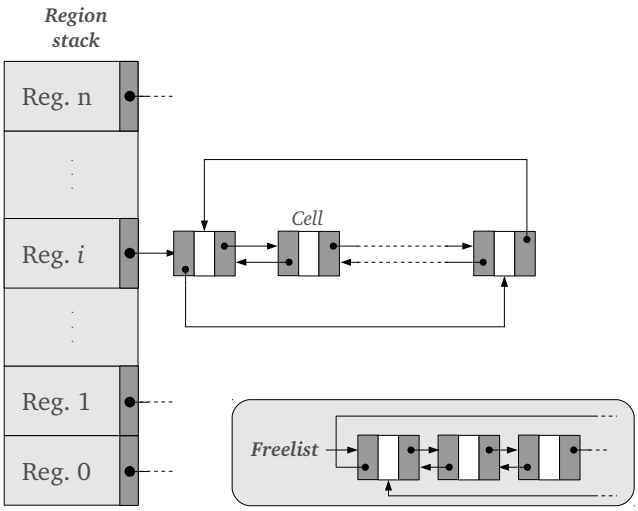


Figure 2.11: A picture of the *Safe* Virtual Machine heap and fresh cells pool

For this reason, we have a simplified variant of *Full-Safe* (which is called *Core-Safe*), with a fewer number of syntactic expressions. This is similar to the translation of Haskell programs into a Core language, as done in the GHC compiler [60]. The details of the translation phase from *Full-Safe* to *Core-Safe* are beyond the scope of this thesis (see [31] for details), but this process follows these general guidelines:

- Each function is represented by a single equation.
- Pattern matching is translated into **case** expressions, whose syntax are similar to that of Haskell.
- Destructive pattern matching is translated into **case**! expressions, which will be explained in Section .
- Region variables are made explicit in *Core-Safe*.
- Only atomic expressions (constants and variables) are allowed in function and constructor applications. Non-atomic expressions occurring inside function and constructor arguments must be introduced via **let** bindings, in the style of A-normal form [44]. For instance, $f\ (2+4)$ is transformed into **let** $z = 2+4$ **in** $f\ z$.

Example 2.12. The translation phase applied to the *append* and *appendD* functions defined previously yields the following result:

$$
\begin{aligned}
\mathit{append}\ \mathit{xs}\ \mathit{ys}\ @\ r \quad &= \quad \textbf{case}\ \mathit{xs}\ \textbf{of} \\
&\qquad [\,] \rightarrow \mathit{ys} \\
&\qquad (x : \mathit{xx}) \rightarrow \textbf{let}\ x_1 = \mathit{append}\ \mathit{xx}\ \mathit{ys}\ @\ r\ \textbf{in}\ (x : x_1)@r
\end{aligned}
$$

$$
\begin{aligned}
\mathit{appendD}\ \mathit{xs}\ \mathit{ys}\ @\ r \quad &= \quad \textbf{case!}\ \mathit{xs}\ \textbf{of} \\
&\qquad [\,] \rightarrow \mathit{ys} \\
&\qquad (x : \mathit{xx}) \rightarrow \textbf{let}\ x_1 = \mathit{appendD}\ \mathit{xx}\ \mathit{ys}\ @\ r\ \textbf{in}\ (x : x_1)@r
\end{aligned}
$$

□

$$
\begin{array}{rrcll}
\textbf{Prog} \ni & prog & \to & \overline{data_i};\, \overline{def_i}\,;e & \\
\textbf{DecData} \ni & data & \to & \textbf{data}\ T\ \overline{\alpha_i}\ @\ \overline{\rho_j} = \overline{C_i\ \overline{s_{ij}}^{n_i}\ @\ \rho} & \\
\textbf{DecFun} \ni & def & \to & f\ \overline{x_i}\ @\ \overline{r_j} = e & \\
\textbf{Exp} \ni & e & \to & a & \{\text{atom: literal } c \text{ or variable } x\} \\
 & & & \mid x\ @\ r & \{\text{copy}\} \\
 & & & \mid a_1 \oplus a_2 & \{\text{basic operator application}\} \\
 & & & \mid C\ \overline{a_i}\ @\ r & \{\text{constructor application}\} \\
 & & & \mid f\ \overline{a_i}\ @\ \overline{r_j} & \{\text{function application}\} \\
 & & & \mid \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 & \{\textbf{let} \text{ declaration: nonrecursive, monomorphic}\} \\
 & & & \mid \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i} & \{\text{read-only pattern matching}\} \\
 & & & \mid \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i} & \{\text{destructive pattern matching}\}
\end{array}
$$

Figure 2.12: *Core-Safe* language definition.

Since every analysis and inference algorithm in this thesis works at the *Core-Safe* level, this language will be described in detail in next section. However, and for the sake of clarity, we will use *Full-Safe* for most medium- and large-sized examples. We shall even use region-annotated *Full-Safe* code, when regions are relevant.

2.3 Syntax of *Core-Safe*

In Figure 2.12 we show the syntax of *Core-Safe* programs and expressions. A program *prog* is a sequence of **data** declarations, followed by a sequence of function definitions $\overline{def}_i$ and a main expression e, whose result is the result of the program. The **data** declarations section follows a syntax similar to that of Haskell, with the addition of region type variables ρ. The role of region type variables, and the syntax of the sequence of types $\overline{s_{ij}}$ are not relevant at this point, and we shall defer their explanation until Chapter 3.

A function definition is a function name f, followed by a list of formal parameters $\overline{x_i}$ (which are variables), a list of formal *region* parameters $\overline{r_j}$ (which are called *region variables*) and the *body* of the function e. The sets of function symbols, variables and region variables are respectively denoted by **Fun**, **Var** and **RegVar**.

We denote by **Exp** the set of *Core-Safe* expressions. Basic expressions include: atomic expressions (literals or variables), copy expressions (Section 2.2.1), function and constructor applications, and a special kind of function applications that we consider to be built-in: basic operator applications. The set of basic operators $\oplus$ is left unspecified. We only demand that applications of these operators require no additional heap space and only two stack words for the arguments. In this work we will only make distinction between basic operator applications and the rest of function applications when this distinction is relevant: when translating *Core-Safe* to machine language, and when considering resource consumption. In the rest of cases basic operator applications are treated as plain function applications.

We assume the existence of a set **Cons** of constructor names, and that, for every constructor C, the set of its recursive positions (denoted by $RecPos(C)$) is known at runtime. For instance,

$$RecPos([\,]) = \emptyset \qquad RecPos(:) = \{2\} \qquad RecPos(Empty) = \emptyset \qquad RecPos(Node) = \{1,3\}$$

We also assume that there is no mutual recursion between functions. This is done for the sake of sim-

plicity. Every analysis and the algorithm described in the next chapters can be adapted with relative ease in order to support mutual recursion.

The **let** construct allows having non-recursive, monomorphic intermediate declarations. Throughout this thesis we use the terms *auxiliary* and *main* expression to refer to the e_1 and e_2 components, respectively.

Core-Safe supports two kinds of pattern matching: read-only (**case**) and destructive (**case!**). In the latter, the cell against which the patterns are matched is also disposed of, so its space can be reused by the runtime system.

By the notation $fv(e)$ we denote the set of variables occurring free in e. These do *not* include region variables and function names. When e is in the context of a function definition, $scope(e)$ denotes the set of variables in scope in the given expression (again, excluding region variables and function names).

2.4 Semantics of *Safe*

In the following subsections we will define a big-step and a small-step operational semantics for *Safe*. All the correctness results in this thesis are proven with respect to the big-step operational semantics. However, the small-step operational semantics is useful as an intermediate step for deriving the *Safe* virtual machines of Section 2.5.

Both semantics describe how a *Core-Safe* expression e is reduced to a *value* (normal form). We use $v, v_i, \ldots$ metavariables to denote *values*, which are defined by the following grammar:

$$\begin{array}{lcll} \mathbf{Val} \ni v & ::= & p \in \mathbf{Loc} & \{ \text{ heap pointer } \} \\ & | & c \in \mathbf{Int} \cup \mathbf{Bool} & \{ \text{ literal: integer or boolean } \} \end{array}$$

Since one of the aims of the language is the inference of safety properties regarding memory pointers, our semantics needs a model of the heap. A *heap* h is defined as a finite mapping from heap pointers to construction cells. *Heap pointers* specify memory locations. We assume the existence of a denumerable set of pointers **Loc** and use $p, p_i, q, \ldots$ to denote elements from this set. A construction cell w is an element of the form $(j, C\ \overline{v_i}^n)$, where j is a natural number, $C \in \mathbf{Cons}$ a constructor symbol of arity n, and $\overline{v_i}^n$ is the list of values to which C is applied. The number j stands for the region of the heap in which the cell is located. With this heap model the region number may be considered as a property of a cell. This implies, on the one hand, that every cell belongs to a region and, on the other hand, that every cell belongs to a *single* region (in other words, regions are *disjoint*). For example, the following mapping h_0 models the heap shown in Figure 2.1b.

$$h_0 = \begin{bmatrix} p_1 & \mapsto & (2, 5 : p_2) \\ p_2 & \mapsto & (2, 7 : p_3) \\ p_3 & \mapsto & (2, [\,]) \end{bmatrix}$$

The notation $region(w)$ represents the region where w lives (that is, the first component of the pair $(j, C\ \overline{v_i}^n)$), whereas $fresh_h(p)$ denotes that the pointer p is fresh in h, that is, it does not occur neither in its domain nor in their cells.

2.4.1 Big-step semantics

In Figure 2.14 we show the big-step operational semantics of *Core-Safe* expressions. A judgement of the form $E \vdash h, k, e \Downarrow h', k, v$ means that expression e is successfully reduced to a normal form v under a runtime environment E and a heap h with $k+1$ regions (ranging from 0 to k) and that a final heap h' with the same number k of regions[1] is produced as a side effect. A runtime environment E (also denoted *value environment*) is a pair of partial functions in $(\mathbf{Var} \to \mathbf{Val}) \times (\mathbf{RegVar} \to \mathbb{N})$ which map, respectively, program variables x to values and region variables r to actual regions (i.e. natural numbers) in the heap. By abuse of notation, and since we use different metavariables to distinguish program variables from region variables (respectively x and r), we will consider both mappings as a single one. Whenever we use $E(x)$ or $E(r)$ we will know which mapping we are referring to. We adopt the convention that, for every value environment E, if c is a literal, $E(c) = c$.

We assume that, during the evaluation of an expression, a program signature Σ is propagated through the $\Downarrow$ judgements. This signature maps function names to program definitions. Since the function name itself occurs in its definition, we shall use the notation $(f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f) \in \Sigma$ for denoting the result of $\Sigma(f)$. Sometimes a signature will be made explicit in the semantic judgements when it is attached to the arrow symbol ($\Downarrow_\Sigma$).

The semantics of a program $prog \equiv \overline{data_i}; \overline{def_i}; e$ is the result of evaluating its main expression e in an environment Σ containing all the function declarations $\overline{def_i}$, under an empty heap with a single region and a value environment which maps the *self* identifier to that region:

$$[self \mapsto 0] \vdash [\,], 0, e \Downarrow h', k', v \tag{2.1}$$

Now we explain in detail the semantic rules. Rules [*Lit*] and [*Var*] just say that literals and heap pointers are normal forms. Rule [*Copy*] executes a copy expression by copying the data structure pointed to by p and living in a region j' into a (possibly different) region j. The runtime system function *copy* follows the pointers in recursive positions of the structure starting at p and creates in region j a copy of all recursive cells. Some restricted type information is assumed to be available in our runtime system (namely, the recursive positions of each constructor) so that this function can be implemented.

Definition 2.13. The *copy* function is defined as follows:

$$\begin{array}{rcl} copy(h_0[p \mapsto (k, C\ \overline{v_i}^n)], p, j) & = & (h_n \uplus [p' \mapsto (j, C\ \overline{v_i'}^n)], p') \\ \quad \textbf{where} \quad fresh_{h_n}(p') & & \\ \quad \forall i \in \{1..n\}.(h_i, v_i') = \left\{ \begin{array}{ll} (h_{i-1}, v_i) & \text{if } i \notin RecPos(C) \\ copy(h_{i-1}, v_i, j) & \text{otherwise} \end{array} \right. & \end{array}$$

Should *copy* find a dangling pointer during the traversal, then the whole rule would fail. If there is no failure, the normal form becomes a fresh pointer p' pointing to the copy. The pointers in non recursive positions of all the copied cells are kept identical in the new cells. This implies that both data structures (the original and the copy), may share some subparts. For instance, if the original DS is a list of lists, the structure created by *copy* is a copy of the outermost list, while the innermost lists become shared between the old and the new list (see Figure 2.13).

Rule [*App*] shows when a new region is allocated. Notice that the body of the function is executed in a heap with $k+2$ regions (from 0 to $k+1$). The formal identifier *self* is bound to the newly created

[1] Actually, the latter k is redundant, as the final heap always has the same number of regions as the initial one. However, in this thesis we shall make the k of the final configuration explicit.

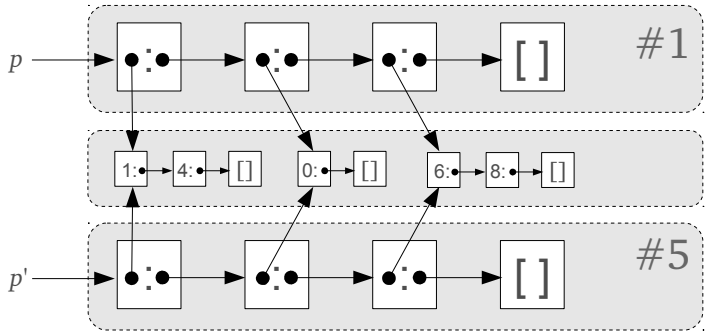


Figure 2.13: Result (pointed to by p') of copying the list of lists pointed to by p into the region number 5. Only the outermost list spine is copied, whereas the innermost lists are shared between original and copy.

region $k+1$ so that the function body may create DSs in this region or pass this region as a parameter to other function calls. Before returning from the function, all cells created in region $k'+1$ are deleted. This action is a source of possible dangling pointers. By the notation $h\mid_k$ we denote the heap obtained by deleting from h those bindings living in regions greater than k:

$$h\mid_k \stackrel{\text{def}}{=} h\mid_{P(k,h)} \qquad \text{where } P(k,h) = \{p \in \operatorname{dom} h \mid region(h(p)) \leq k\}$$

Rule [*Cons*] generates a fresh location p pointing to the newly constructed cell. The parameters of the corresponding constructor are looked up into the value environment E.

Rule [*Let*] shows the eagerness of the language: first, the auxiliary expression e_1 is reduced to normal form and then the main expression e_2 is evaluated. In the latter evaluation the environment is extended by binding the program variable x_1 to the normal form to which e_1 is reduced.

The [*Case*] rule is the usual one for an eager language, whereas the [*Case!*] rule expresses what happens in a destructive pattern matching: the binding of the discriminant variable disappears from the heap. This action is another source of possible dangling pointers.

Now we use these semantics to prove some general properties of the language. In principle we are interested only in those value environments that map the *self* identifier to the highest possible region number, and the remaining region variables to numbers strictly lower than the one bound to *self*. When a value environment meets these requirements, it is said to be *admissible*.

Definition 2.14. A value environment E is admissible with respect to k iff $E(self) = k$ and for every other region variable $r \in \operatorname{dom} E$ it holds that $E(r) < k$.

The next Proposition shows that admissibility is preserved by the evaluation of an expression.

Proposition 2.15. *Let us consider a judgement $E \vdash h, k, e \Downarrow h', k, v$ in which E is admissible w.r.t. k. Then any value environment occurring in the derivation of this judgement is also admissible w.r.t. its corresponding k.*

Proof. The property is true at the initial judgement and is preserved in every inductive rule. The only relevant case is the [*App*] rule. □

It is easy to show that the initial value environment in the execution of a *Core-Safe* program (2.1) is admissible and, hence, that all the value environments taking place in the execution of the program are admissible. This allows us to leave out the conditions $j \leq k$ in [*Copy*] and $E(r) \leq k$ in [*Cons*], since they are guaranteed to hold when their corresponding judgements are admissible.

$$\frac{}{E \vdash h, k, c \Downarrow h, k, c} \; [Lit]$$

$$\frac{}{E[x \mapsto v] \vdash h, k, x \Downarrow h, k, v} \; [Var]$$

$$\frac{}{E \vdash h, k, a_1 \oplus a_2 \Downarrow h, k, E(a_1) \oplus E(a_2)} \; [PrimOp]$$

$$\frac{j \leq k \qquad (h', p') = copy(h, p, j)}{E[x \mapsto p, r \mapsto j] \vdash h, k, x \mathbin{@} r \Downarrow h', k, p'} \; [Copy]$$

$$\frac{E(r) \leq k \qquad fresh_h(p)}{E \vdash h, k, C\ \overline{a_i}^n \mathbin{@} r \Downarrow h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)], k, p} \; [Cons]$$

$$\frac{(g\ \overline{y_i}^n \mathbin{@} \overline{r'_j}^m = e_g) \in \Sigma \qquad [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^m, self \mapsto k+1] \vdash h, k+1, e_g \Downarrow h', k+1, v}{E \vdash h, k, g\ \overline{a_i}^n \mathbin{@} \overline{r_j}^m \Downarrow h'|_k, k, v} \; [App]$$

$$\frac{E \vdash h, k, e_1 \Downarrow h_1, k, v_1 \qquad E \uplus [x_1 \mapsto v_1] \vdash h_1, k, e_2 \Downarrow h', k, v}{E \vdash h, k, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow h', k, v} \; [Let]$$

$$\frac{E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow h', k, v}{E[x \mapsto p] \vdash h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})], k, \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow h', k, v} \; [Case]$$

$$\frac{E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow h', k, v}{E[x \mapsto p] \vdash h \uplus [p \mapsto (j, C_r\ \overline{v_i}^{n_r})], k, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow h', k, v} \; [Case!]$$

Figure 2.14: Big-step operational semantics of *Core-Safe* expressions.

We are interested in the following question: given a value environment E, an initial heap h with k regions, and a *Core-Safe* expression (assuming E admissible w.r.t. k), is the judgement $E \vdash h, k, e \Downarrow h', k, v$ is derivable for some h' and v? The answer would be negative if the derivation of $E \vdash h, k, e \Downarrow h', k, v$ requires proving a judgement in which none of the rules of Figure 2.14 is applicable. This happens in the following situations:

1. Access to a variable or region variable which does not belong to the domain of the value environment E (rules [*Var*], [*Copy*], [*App*], [*Cons*], [*Case*] and [*Case!*]).
2. Call to a function not contained in the signature Σ (rule [*App*]).
3. Unsatisfiability of the disjointness conditions imposed by the $\uplus$ operator in value environments. For example, when we obtain $E \uplus [x_1 \mapsto v_1]$ and the x_1 variable is already in dom E. (rules [*Let*], [*Case*] and [*Case!*]).
4. Impossibility of obtaining a finite derivation for a given judgement.
5. Pattern matching-related errors: the data constructor of the cell pointed to by the discriminant of a **case**$(!)$ does not match any of the provided alternatives (rules [*Case*] and [*Case!*]).
6. A literal is found in E, when a pointer was expected (rules [*Copy*], [*Case*], [*Case!*]).
7. Access to a *dangling pointer*, that is, a location p not belonging to the domain of h (rules [*Copy*], [*Case*] and [*Case!*]).

The first two points are easy to address. Most compilers provide a contextual constraints checker for ensuring that every variable only occurs in the scope of its definition. *Safe*'s compiler is not an exception, so we will always assume that, during the evaluation of e under a value environment E, it holds that $fv(e) \subseteq$ dom E. Moreover, we assume that every function being called (direct or indirectly) from e is defined in the signature Σ under which e is evaluated. With respect to the third point, an intermediate renaming phase in the *Safe* compiler prevents us from having two variables with the same name and non-disjoint scopes. Therefore we can safely assume that the disjoint union $E \uplus [x_1 \mapsto v_1]$ in the [*Let*] rule, and those occurring in the value environments of the [*Case*] and [*Case!*] rules are well-defined.

The fourth point (non-finite derivations) arises when dealing with non-terminating expressions. In general, semantic definitions written in a big-step style can only specify finite computations, since a finite proof of the judgement $E \vdash h, k, e \Downarrow h', k, v$ implicitly assumes that the evaluation of e terminates under the environment E and initial heap h. Whenever we prove a semantic property of a program via the rules of Figure 2.14, the finiteness of the corresponding derivation is implicitly assumed. To approximate whether a program terminates or not under a given input is a currently active research field, and there are numerous techniques for proving termination of programs. This topic is closely related to resource analysis, since a terminating program always consume a finite amount of resources. In Chapter 3.2 this connection will become more apparent.

The fifth and sixth points are addressed with a Hindley-Milner type system [34] which guarantees the absence of failed pattern matchings, since every **case**$(!)$ expression must have a branch for every constructor of the type of its discriminant.

The last point is one of the aims of this thesis. In Chapter 3 we will introduce a type system guaranteeing that dangling pointers are never accessed by a program. First we need to focus on how dangling pointers are created: they arise as a result of removing a binding from the heap. If we inspect the rules in Figure 2.14, it turns out there are two possible sources of dangling pointers:

- The deallocation of the topmost region $h\ |_k$ when the evaluation of a function application finishes ([*App*] rule).
- The removal of a pointer mapped by a **case**! discriminant ([*Case!*] rule).

Example 2.16. Consider the following region-annotated function definitions:

$$copyToSelf\ xs = xs\ @\ self$$

$$f\ xs\ ys\ @\ r = \textbf{let}\ zs = appendD\ xs\ ys\ @\ r\ \textbf{in}\ length\ xs + length\ zs$$

The first one builds a copy of the input list in the working region and returns a reference to it. However, and since the working region disappears when the function finishes, *copyToSelf* actually returns a dangling pointer. In the second function, the *length* function access the *xs* list, which has been previously destroyed by *appendD*. The execution of *length* fails because *xs* refers to a dangling pointer. Both functions are rejected by the type system of Chapter 3. □

2.4.2 Small-step semantics

In Figure 2.15 we show the small-step semantic rules. There are two kinds of judgements:

- The first kind, $E, h, k_0, k, e \longrightarrow h', k_0, v$, is applied when an expression e is evaluated to a value v in a single step. These correspond to literals, variables, copy expressions and constructors.
- The other kind, $E, h, k_0, k, e \longrightarrow E', h', k_0, k', e'$, covers the remaining cases: function application, **let**, **case** and **case**! expressions.

In the configurations, k denotes the highest region available in h, as in the big step semantics. The meaning of k_0 will be explained below. Notice that, in the rules, **let** expressions are marked with a natural number δ and an environment E. In rule [*App*-s], the number of available regions is incremented by one, as a new local region is allocated and assigned number $k+1$. Additionally, this rule discards the environment E, as in the function body only the arguments and the *self* region are in scope. However, and due to **let** expressions, a continuation is possible after function application. Thus we need to recover the discarded environment and the original value of k. The environment and the number δ are attached to the **let**-binding in order to remember the newly created regions during the evaluation of the auxiliary expression, so that the original k can be later recovered. The initial values of δ and E are respectively 0 and $\perp$, which are assumed to be annotated in the text of the program being executed. Rule [*Let2*-s] saves the environment for the first time, whereas the [*Let4*-s] rule updates the information as necessary during the evaluation of the auxiliary expression. In case this evaluation is successful, rules [*Let1*-s] or [*Let3*-s] are applied to proceed with the evaluation of the main expression e_2.

New regions being created during the evaluation of the bound expression e_1 of a **let** cannot contain the result of the evaluation because these regions are removed from the heap when e_1 is reduced to a normal form. Region k_0 denotes the highest region that must be available when the machine stops reducing the expression. It can be considered a lower watermark indicating which was the topmost region when the expression under evaluation began to be evaluated. When a normal form is reached, we remove all the regions $\{k, k-1, \ldots, k_0+1\}$ from the heap. Initially $k = k_0 = 0$. Rule [*App*-s] increments k while rules [*Lit*-s], [*Var*-s], [*Copy*-s] and [*Cons*-s] discard all the local regions greater than k_0.

$$\frac{k \geq k_0}{E, h, k_0, k, c \longrightarrow h \mid_{k_0}, k_0, c} \text{ [Lit-s]}$$

$$\frac{k \geq k_0}{E[x \mapsto v], h, k_0, k, x \longrightarrow h \mid_{k_0}, k_0, v} \text{ [Var-s]}$$

$$\frac{k \geq k_0 \qquad k \geq j \qquad (h', p') = copy(h, p, j)}{E[x \mapsto p, r \mapsto j], h, k_0, k, x \mathbin{@} r \longrightarrow h' \mid_{k_0}, k_0, p'} \text{ [Copy-s]}$$

$$\frac{E(r) \leq k \qquad fresh_h(p)}{E, h, k_0, k, C\ \overline{a_i}^n \mathbin{@} r \longrightarrow (h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)]) \mid_{k_0}, k_0, p} \text{ [Cons-s]}$$

$$\frac{(g\ \overline{y_i}^n \mathbin{@} \overline{r'_j}^m = e_g) \in \Sigma}{E, h, k_0, k, g\ \overline{a_i}^n \mathbin{@} \overline{r_j}^m \longrightarrow [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^m, self \mapsto k+1], h, k_0, k+1, e_g} \text{ [App-s]}$$

$$\frac{E, h, k, k, e_1 \longrightarrow h', k, v_1}{E, h, k_0, k, \mathbf{let}\ x_1 =^{\perp}_0 e_1\ \mathbf{in}\ e_2 \longrightarrow E \uplus [x_1 \mapsto v_1], h', k_0, k, e_2} \text{ [Let1-s]}$$

$$\frac{E, h, k, k, e_1 \longrightarrow E', h', k, k+\eta, e'_1}{E, h, k_0, k, \mathbf{let}\ x_1 =^{\perp}_0 e_1\ \mathbf{in}\ e_2 \longrightarrow E', h', k_0, k+\eta, \mathbf{let}\ x_1 =^{E}_{\eta} e'_1\ \mathbf{in}\ e_2} \text{ [Let2-s]}$$

$$\frac{E' \neq \perp \qquad E, h, k, k+\delta, e_1 \longrightarrow h', k, v_1}{E, h, k_0, k+\delta, \mathbf{let}\ x_1 =^{E'}_{\delta} e_1\ \mathbf{in}\ e_2 \longrightarrow E' \uplus [x_1 \mapsto v_1], h', k_0, k, e_2} \text{ [Let3-s]}$$

$$\frac{E' \neq \perp \qquad E, h, k, k+\delta, e_1 \longrightarrow E'', h', k, k+\eta, e'_1}{E, h, k_0, k+\delta, \mathbf{let}\ x_1 =^{E'}_{\delta} e_1\ \mathbf{in}\ e_2 \longrightarrow E'', h', k_0, k+\eta, \mathbf{let}\ x_1 =^{E'}_{\eta} e'_1\ \mathbf{in}\ e_2} \text{ [Let4-s]}$$

$$\frac{E(x) = p}{E, h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})], k_0, k, \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^m \longrightarrow E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}], h, k_0, k, e_r} \text{ [Case-s]}$$

$$\frac{E(x) = p}{E, h \uplus [p \mapsto (j, C_r\ \overline{b_i}^{n_r})], k_0, k, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^m \longrightarrow E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}], h, k_0, k, e_r} \text{ [Case!-s]}$$

Figure 2.15: Small-step operational semantics of *Core-Safe* expressions.

This small-step semantics is equivalent to the big-step semantics described in Section 2.4.1: for any k and $k_0 \leq k$, $E \vdash h, k, e \Downarrow h', k, v$ if and only if $E, h, k_0, k, e \longrightarrow^* h' \mid_{k_0}, k_0, v$. However, we leave out the proof of this fact, since these semantics are only of interest for deriving the SAFE-M2 and SVM machines described below. We shall directly prove the equivalence between big-step semantics and the translation to SVM code in Section 2.8.

2.5 Virtual machines

In this section we formally describe two abstract machines for executing *Safe* expressions: SAFE-M2 and SVM (*Safe Virtual Machine*). The former is functional and it serves as an intermediate step between the SVM and the small-step semantics defined previously. The SVM is an imperative abstract machine based on the execution of a sequence of instructions.

2.5.1 SAFE-M2 abstract machine

Our next refinement is to introduce an abstract machine, called SAFE-M2 because there was a previous one called SAFE-M1 now abandoned. Both are named after Sestoft's *Mark-1* and *Mark-2* abstract machines [104]. A configuration of the machine is a 7-tuple $(h, k_0, k, e, E, S, \Sigma)$, where h is the heap, k_0, k are region numbers as in the small-step semantics, e is the expression being executed, E is the runtime environment, S is a stack, and Σ is a function giving the code of every defined *Core-Safe* function. In Figure 2.16 we show the transitions of the abstract machine SAFE-M2. The only new element w.r.t. the small-step semantics is the stack S, which is intended to replace the annotations in **let** expressions. A stack is a sequence of *continuation frames* of the form (k_0, x_1, e_2, E). More precisely,

$$
\begin{array}{rcll}
S & ::= & (k_0, x_1, e_2, E) : S & \{ \text{ where } k_0 \in \mathbb{N}, x_1 \in \mathbf{Var}, e_2 \in \mathbf{Exp}, E \in (\mathbf{Var} \to \mathbf{Val}) \times (\mathbf{RegVar} \to \mathbb{N}) \} \\
 & | & [\,] & \{ \text{ empty stack } \}
\end{array}
$$

The e_2 corresponds to the pending main expression of a **let** whose auxiliary expression e_1 is under evaluation. Region k_0 is the topmost region where the normal form of e_2 will be returned; higher regions will be disposed of after that normal form is reached. Variable x_1 is the **let**-bound variable free in e_2, and E is the environment which has been kept before the evaluation of e_1, and that will be restored before the evaluation of e_2. Corresponding to the inductive small-step semantic rules ([*Let1*-s] to [*Let4*-s]), the abstract machine rule [*Let*-M2] pushes a continuation to the stack and proceeds with the evaluation of the auxiliary expression e_1. When the normal form of e_1 is reached in rules [*Lit2*-M2] and [*Var2*-M2], the continuation is popped and the machine proceeds with the evaluation of the main expression. If there are no continuations in the stack, the machine stops. This situation would happen if (and only if) the normal form of the main expression is reached.

Notice that the current environment is discarded in rules [*Lit2*-M2] and [*Var2*-M2] when a continuation is popped from the stack. Also, it is discarded in rule [*App*-M2] when a function body is entered and the formal arguments become the only variables in scope. In Section 2.6 this will have the important consequence that tail recursion is translated so that only a constant stack space is needed. Notice also in rule [*Let*-M2] that the current environment is saved in the stack but it is not discarded from the control. One important aspect of the translation given in Section 2.6 is that it manages to avoid this implicit duplication of environments.

The current environment is extended with new bindings in the following situations:

Initial/final configuration	Condition	Label
$(h,\ k_0,\ k,\ c,\ E,\ S,\ \Sigma)$ $\rightarrow\ (h|_{k_0},\ k_0,\ k_0,\ c,\ E,\ S,\ \Sigma)$	$k > k_0$	[*Lit1*-M2]
$(h,\ k,\ k,\ c,\ E',\ (k_0, x_1, e, E) : S,\ \Sigma)$ $\rightarrow\ (h,\ k_0,\ k,\ e,\ E \uplus [x_1 \mapsto c],\ S,\ \Sigma)$		[*Lit2*-M2]
$(h[p \mapsto (j, C\ \overline{v_i}^n)],\ k_0,\ k,\ x,\ E[x \mapsto p],\ S,\ \Sigma)$ $\rightarrow\ (h|_{k_0},\ k_0,\ k_0,\ x,\ E,\ S,\ \Sigma)$	$k > k_0$	[*Var1*-M2]
$(h[p \mapsto (j, C\ \overline{v_i}^n)],\ k,\ k,\ x,\ E'[x \mapsto p],\ (k_0, x_1, e, E) : S,\ \Sigma)$ $\rightarrow\ (h,\ k_0,\ k,\ e,\ E \uplus [x_1 \mapsto p],\ S,\ \Sigma)$		[*Var2*-M2]
$(h,\ k_0,\ k,\ a_1 \oplus a_2,\ E,\ S,\ \Sigma)$ $\rightarrow\ (h',\ k_0,\ k,\ y,\ E \uplus [y \mapsto E(a_1) \oplus E(a_2)],\ S,\ \Sigma)$	$\mathit{fresh}(y)$	[*PrimOp*-M2]
$(h[p \mapsto (l, C\ \overline{v_i}^n)],\ k_0,\ k,\ x\ @\ r,\ E[x \mapsto p, r \mapsto j],\ S,\ \Sigma)$ $\rightarrow\ (h',\ k_0,\ k,\ y,\ E \uplus [y \mapsto p'],\ S,\ \Sigma)$	$(h', p') = \mathit{copy}(h, p, j)$ $j \leq k, \mathit{fresh}(y)$	[*Copy*-M2]
$(h,\ k_0,\ k,\ C\ \overline{a_i}^n\ @\ r,\ E,\ S,\ \Sigma)$ $\rightarrow\ (h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)],\ k_0,\ k,\ y,\ E \uplus [y \mapsto p],\ S,\ \Sigma)$	$E(r) \leq k$ $\mathit{fresh}_h(p), \mathit{fresh}(y)$	[*Cons*-M2]
$(h,\ k_0,\ k,\ g\ \overline{a_i}^n\ @\ \overline{r_j}^m,\ E,\ S,\ \Sigma)$ $\rightarrow\ (h,\ k_0,\ k+1,\ e_g,\ [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^m, \mathit{self} \mapsto k+1],\ S,\ \Sigma)$	$(g\ \overline{y_i}^n\ @\ \overline{r'_j}^m = e_g) \in \Sigma$	[*App*-M2]
$(h,\ k_0,\ k,\ \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2,\ E,\ S,\ \Sigma)$ $\rightarrow\ (h,\ k,\ k,\ e_1,\ E,\ (k_0, x_1, e_2, E) : S,\ \Sigma)$		[*Let*-M2]
$(h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})],\ k_0,\ k,\ \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i},\ E[x \mapsto p],\ S,\ \Sigma)$ $\rightarrow\ (h,\ k_0,\ k,\ e_r,\ E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}],\ S,\ \Sigma)$		[*Case*-M2]
$(h \uplus [p \mapsto (j, C_r\ \overline{v_i}^{n_r})],\ k_0,\ k,\ \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i},\ E[x \mapsto p],\ S,\ \Sigma)$ $\rightarrow\ (h,\ k_0,\ k,\ e_r,\ E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}],\ S,\ \Sigma)$		[*Case!*-M2]

Figure 2.16: Transition rules of the SAFE-M2 abstract machine.

Initial/final configuration		Condition
	$(\texttt{DECREGION} : is,\ h,\ k_0,\ k,\ S,\ cs)$	$k \geq k_0$
$\rightarrow$	$(is,\ h|_{k_0},\ k_0,\ k_0,\ S,\ cs)$	
	$([\texttt{POPCONT}],\ h,\ k,\ k,\ b : (k_0, \mathbf{p}) : S,\ cs[\mathbf{p} \mapsto is])$	
$\rightarrow$	$(is,\ h,\ k_0,\ k,\ b : S,\ cs)$	
	$(\texttt{PUSHCONT}\ \mathbf{p} : is,\ h,\ k_0,\ k,\ S,\ cs[\mathbf{p} \mapsto is'])$	
$\rightarrow$	$(is,\ h,\ k,\ k,\ (k_0, \mathbf{p}) : S,\ cs)$	
	$(\texttt{COPY} : is,\ h,\ k_0,\ k,\ p : j : S,\ cs)$	$(h', p') = copy(h, p, j)$
$\rightarrow$	$(is,\ h',\ k_0,\ k,\ p' : S,\ cs)$	$j \leq k$
	$([\texttt{CALL}\ \mathbf{p}],\ h,\ k_0,\ k,\ S,\ cs[\mathbf{p} \mapsto is])$	
$\rightarrow$	$(is,\ h,\ k_0,\ k+1,\ S,\ cs)$	
	$(\texttt{PRIMOP} \oplus : is,\ h,\ k_0,\ k,\ c_1 : c_2 : S,\ cs)$	$c = c_1 \oplus c_2$
$\rightarrow$	$(is,\ h,\ k_0,\ k,\ c : S,\ cs)$	
	$([\texttt{MATCH}\ l\ \overline{\mathbf{p}_j}^m],\ h[S!l \mapsto (j, C_r^m\ \overline{v_i}^n)],\ k_0,\ k,\ S,\ cs[\overline{\mathbf{p}_j \mapsto is_j}^m])$	
$\rightarrow$	$(is_r,\ h,\ k_0,\ k,\ \overline{v_i}^n : S,\ cs)$	
	$([\texttt{MATCH!}\ l\ \overline{\mathbf{p}_j}^m],\ h \uplus [S!l \mapsto (j, C_r^m\ \overline{v_i}^n)],\ k_0,\ k,\ S,\ cs[\overline{\mathbf{p}_j \mapsto is_j}^m])$	
$\rightarrow$	$(is_r,\ h,\ k_0,\ k,\ \overline{v_i}^n : S,\ cs)$	
	$(\texttt{BUILDENV}\ \overline{K_i}^n : is,\ h,\ k_0,\ k,\ S,\ cs)$	
$\rightarrow$	$(is,\ h,\ k_0,\ k,\ \overline{Item_k(K_i)}^n : S,\ cs)$	(1)
	$(\texttt{BUILDCLS}\ C_r^m\ \overline{K_i}^n\ K : is,\ h,\ k_0,\ k,\ S,\ cs)$	$Item_k(K) \leq k$, $fresh(b)$
$\rightarrow$	$(is,\ h \uplus [b \mapsto (Item_k(K), C_r^m\ \overline{Item_k(K_i)}^n)],\ k_0,\ k,\ b : S,\ cs)$	(1)
	$(\texttt{SLIDE}\ m\ n : is,\ h,\ k_0,\ k,\ \overline{b_i}^m : \overline{b'_i}^n : S,\ cs)$	
$\rightarrow$	$(is,\ h,\ k_0,\ k,\ \overline{b_i}^m : S,\ cs)$	

$$(1)\ \ Item_k(K) \stackrel{\text{def}}{=} \begin{cases} S!j & \text{if } K = \texttt{Pos}\ j \in \mathbb{N} \\ c & \text{if } K = \texttt{Lit}\ c \\ k & \text{if } K = self \end{cases}$$

Figure 2.17: Transition rules of the SVM abstract machine.

- In rules [*Case*-M2] and [*Case!*-M2], as soon as let-bound or case-bound variables become free variables in scope in the continuation expression.
- In rules [*Copy*-M2] and [*Cons*-M2], the environment is extended with a fresh program variable y. This is merely an artifact due to the fact that a fresh data structure must be referenced in the control expression.
- In rules [*Lit2*-M2] and [*Var2*-M2], the environment E saved in the continuation must be extended with the new binding introduced by **let**.

2.5.2 SVM imperative abstract machine

The main differences of the SVM with respect to SAFE-M2 are two:

- Expressions are replaced by a sequence of imperative instructions.

- There are no runtime environments: values and continuations live in the stack.

Instructions and instruction sequences of the SVM are given by the following grammar:

$$
\begin{array}{rcllll}
\iota & ::= & \texttt{DECREGION} & \mid \texttt{COPY} & \mid \texttt{MATCH}\ l\ \overline{\mathbf{p}_i} & \mid \texttt{BUILDCLS}\ C\ \overline{K_i}\ K \\
 & \mid & \texttt{POPCONT} & \mid \texttt{CALL}\ \mathbf{p} & \mid \texttt{MATCH!}\ l\ \overline{\mathbf{p}_i} & \mid \texttt{SLIDE}\ m\ n \\
 & \mid & \texttt{PUSHCONT}\ \mathbf{p} & \mid \texttt{PRIMOP}\ \oplus & \mid \texttt{BUILDENV}\ \overline{K_i} & \\
\mathit{is} & ::= & [\,] & & & \\
 & \mid & \iota : \mathit{is} & & &
\end{array}
$$

In this section we introduce only the semantics of these instructions in the context of the SVM machine. Their specific role will become clearer when dealing with the translation from *Core-Safe* to SVM code. During the execution a *code store* (resulting from the compilation of program fragments) is kept. A code store maps *code pointers* to instruction sequences. We use the metavariables $\mathbf{p}, \mathbf{p}_i, \ldots$ for denoting code pointers. We assume the existence of a denumerable set **PCode** containing them. The l variable occurring in the `MATCH` and `MATCH!` instructions stands for a natural number representing a stack position (counting from the topmost element). Finally, the elements K occurring in `BUILDENV` and `BUILDCLS`, which will be called *keys*, are generated by the following grammar:

$$
\begin{array}{rcll}
K & ::= & \texttt{Pos}\ j & \{\, j \in \mathbb{N}, \text{stack position} \,\} \\
 & \mid & \texttt{Lit}\ c & \{\, \text{literal} \,\} \\
 & \mid & \mathit{self} & \{\, \text{working region identifier} \,\}
\end{array}
$$

A SVM configuration c consists of six components

$$c \equiv (\mathit{is},\ h,\ k_0,\ k,\ S,\ \mathit{cs})$$

where *is* is the instruction sequence currently being executed, h is a heap with k regions, and k_0 plays a similar role as in the small-step semantics and the SAFE-M2 abstract machine: it denotes the identifier of the highest region available after a normal form is computed. The *cs* variable denotes a code store containing all the program fragments, whereas S denotes a stack which may contain values, region numbers and continuations of the form $(k_0, \mathbf{p})$. We use the metavariable b for denoting stack values:

$$
\begin{array}{rcll}
b & ::= & j & \{\, \text{where } j \in \mathbb{N} \,\} \\
 & \mid & v & \{\, \text{where } v \in \mathbf{Val} \,\} \\
 & \mid & (k_0, \mathbf{p}) & \{\, \text{continuation: } k_0 \in \mathbb{N},\ \mathbf{p} \in \mathbf{PCode} \,\}
\end{array}
$$

In Figure 2.17 we show the semantics of SVM instructions in terms of transitions between configurations. By C_r^m we denote the data constructor which is the r-th in its **data** definition out of a total of m data constructors. By $S!j$ we denote the j-th element of the stack S counting from the top and starting at 0 (i.e. $S!0$ is the topmost element). Notice that continuations take up two words in the stack, so, if $n \geq 2$ we get $((k_0, \mathbf{p}) : S)!n = S!(n-2)$. The notation $\overline{b_i}^n : S$ abbreviates the stack $b_1 : \cdots : b_n : S$.

Instruction `DECREGION` deletes from the heap all the regions, if any, between the current region k and region k_0, excluding the latter. It will be used when a normal form is reached.

Instruction `POPCONT` pops a continuation from the stack or stops the execution if there is none. It is always assumed to be the last element of the current instruction sequence (the notation $[\texttt{POPCONT}]$ stands for $\texttt{POPCONT} : [\,]$). Notice that b (which is usually a value) is left in the stack so that it can be accessed by the continuation. Instruction `PUSHCONT` pushes a continuation in the stack. It is used in the

translation of a **let**.

The COPY instruction just mimics its counterpart [*Copy*-M2] in the SAFE-M2 abstract machine.

Instruction CALL jumps to a new instruction sequence and creates a new region. It will be used in the compilation of a function application.

Instruction PRIMOP operates two basic values located in the stack and replaces them by the result of the operation.

Instruction MATCH performs a vectored jump depending on the constructor of the matched closure. The vector of sequences pointed to by the $\mathbf{p}_j$ corresponds to the compilation of a set of **case** alternatives. The MATCH! instruction additionally destroys the matched cell.

The BUILDENV instruction receives a list of keys K_i and creates a portion of environment on top of the stack: If a key K is a natural number j, the item $S!j$ is copied and pushed on the stack; if it is a basic constant c, it is directly pushed on the stack; if it is the identifier *self*, then the current region number k is pushed on the stack.

Instruction BUILDCLS allocates fresh memory and constructs a new cell in the style of the [*Cons*-M2] rule in SAFE-M2. Similarly to BUILDENV, it receives a list of keys and uses the same conventions. It also receives the constructor C_r^m of the cell being created.

Finally, instruction SLIDE removes some parts of the stack. It will be used to remove environments when they are no longer needed.

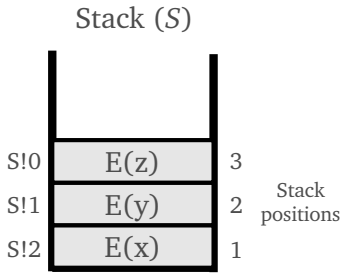
2.6 Translating *Core-Safe* into SVM code

A major difference of SVM with respect to M2 is the lack of a value environment E in the SVM. The values of E are assumed to live in the stack S. In order to perform the translation from *Core-Safe* into SVM instructions, we need to set up a correspondence between program variables (which are no longer present during the execution of a SVM program) and the positions of the stack which contain the values of these variables.

The main idea of the translation is to keep a compile-time environment ρ mapping program variables to stack positions. As the stack grows dynamically, a first idea is to assign numbers to the variables from the bottom of the environment to the top. In this way, if the current environment occupies the top m positions of the stack and $\rho(x) = 1$, then $S!(m-1)$ will contain the runtime value corresponding to x (see Figure 2.18).

A second idea is to reuse the current environment when pushing a continuation into the stack. Let us recall the [*Let*-M2] rule of Figure 2.16. The environment E being pushed into the stack is the same as the environment in which the auxiliary expression e_1 is evaluated. In a naive implementation, the current environment E would be duplicated, and its copy pushed into the stack. Our aim is to *share* the environment instead of duplicating it, and to push only the following two elements of the continuation: k_0 and e (more precisely, the code pointer containing the SVM code of e). The variable x_1 is not needed, since the compilation process will ensure that a pointer to its value will be on top of the stack when the continuation is popped.

In order to carry out this sharing between runtime environments, we split the whole compile-time environment ρ into a list of smaller environments $[\rho_1, \ldots, \rho_n]$, each one topped with a continuation, except the topmost one (ρ_1). Each individual block ρ_i consists of a triple (Δ_i, l_i, n_i) with the actual environment Δ_i mapping variables to numbers in the range $(1 \ldots m_i)$, its length $l_i = m_i + n_i$, and an indicator n_i whose value is 2 for all the blocks except for the first one, whose value is $n_1 = 0$. (see Figure 2.19). This value stands for the length of the continuation $(k_0, \mathbf{p})$: we assume that a continuation needs



(a) Stack representation

$$\begin{bmatrix} x & \mapsto & 1 \\ y & \mapsto & 2 \\ z & \mapsto & 3 \end{bmatrix}$$

(b) Compile-time environment

Figure 2.18: A first approach to mapping program variables to stack offsets.

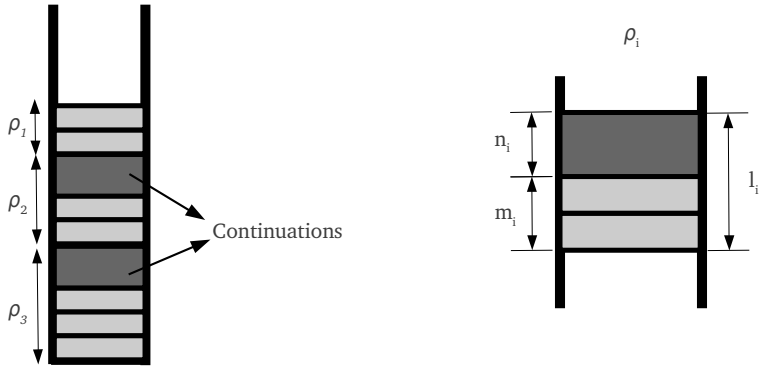


Figure 2.19: Division of ρ into a list of smaller environments.

two words in the stack and that the remaining items need one word. The positions given by the smaller environments Δ_i are relative to the bottommost position of their block in the stack.

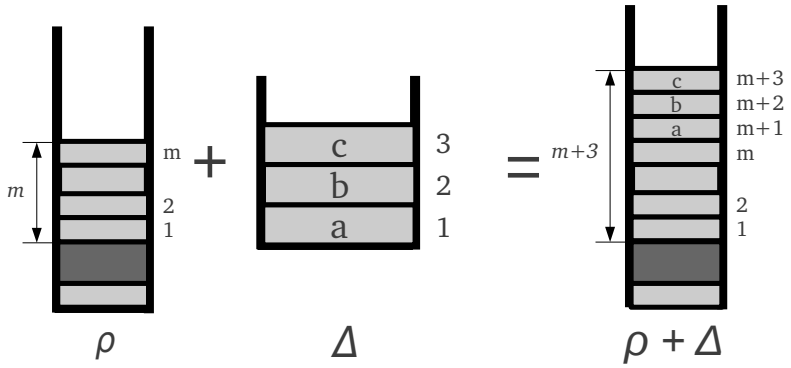
Given this definition of a compile-time environment, we can compute the offset w.r.t. the top of the stack of a given variable x, defined in the k-th block. We use the notation $\rho(x)$ to denote this offset.

$$\rho(x) \stackrel{\text{def}}{=} \sum_{i=1}^{k} l_i - \Delta_k(x)$$

We assume by convention that $\rho(self) = self$. As soon we introduce new variables in scope, we need to update the environment ρ accordingly. Only the topmost environment can be extended with new bindings. We define the following operations and functions on compile-time environments:

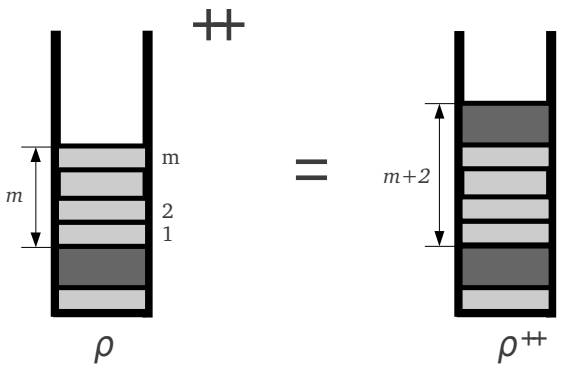
1. **Operator** $+$. It adds new bindings to the compile-time environment:

$$((\Delta, m, 0) : \rho) + [\overline{x_i \mapsto j_i}^n] \stackrel{\text{def}}{=} (\Delta \uplus [\overline{x_i \mapsto m + j_i}^n], m + n, 0) : \rho$$



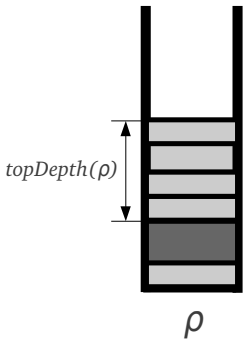
2. **Operator** $^{++}$. It finishes the topmost block, and starts a new one above it:

$$((\Delta, m, 0) : \rho)^{++} \stackrel{\text{def}}{=} ([\,], 0, 0) : (\Delta, m + 2, 2) : \rho$$



3. **Function** *topDepth*. It returns the length of the topmost block. Its result is undefined when applied to empty environments.

$$topDepth((\Delta, m, 0) : \rho) \stackrel{\text{def}}{=} m$$



Given these conventions, in Figure 2.20 we show the translation function *trE* which receives a *Core-Safe* expression with a compile-time environment ρ, and returns a list of SVM instructions and

$$
\begin{array}{lll}
trE\ c\ \rho & = & \texttt{BUILDENV}\ [c]; \\
 & & \mathit{NormalForm}\ \rho \\
 & & \\
trE\ x\ \rho & = & \texttt{BUILDENV}\ [\rho(x)]; \\
 & & \mathit{NormalForm}\ \rho \\
 & & \\
trE\ (x\ @\ r)\ \rho & = & \texttt{BUILDENV}\ [\rho(x), \rho(r)]; \\
 & & \texttt{COPY}; \\
 & & \mathit{NormalForm}\ \rho \\
 & & \\
trE\ (a_1 \oplus a_2)\ \rho & = & \texttt{BUILDENV}\ [\rho(a_1), \rho(a_2)]; \\
 & & \texttt{PRIMOP}\ \oplus; \\
 & & \mathit{NormalForm}\ \rho \\
 & & \\
trE\ (g\ \overline{a_i}^n\ @\ \overline{r_j}^m)\ \rho & = & \texttt{BUILDENV}\ [\overline{\rho(a_i)}^n, \overline{\rho(r_j)}^m]; \\
 & & \texttt{SLIDE}\ (n+m)\ (\mathit{topDepth}(\rho)); \\
 & & \texttt{CALL}\ \mathbf{p} \\
\multicolumn{3}{l}{\quad \textbf{where}\ (g\ \overline{y_i}^n\ @\ \overline{r'_j}^m = e_g) \in \Sigma} \\
\multicolumn{3}{l}{\qquad [\mathbf{p} \mapsto\ trE\ e_g\ [([\overline{r'_j \mapsto m-j+1}^m, \overline{y_i \mapsto n-i+m+1}^n], n+m, 0)]] \in cs} \\
trE\ (C_l^m\ \overline{a_i}^n\ @\ r)\ \rho & = & \texttt{BUILDCLS}\ C_l^m\ [\overline{\rho(a_i)}^n]\ (\rho(r)); \\
 & & \mathit{NormalForm}\ \rho \\
 & & \\
trE\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2)\ \rho & = & \texttt{PUSHCONT}\ \mathbf{p}; \qquad \&\ cs \uplus [\mathbf{p} \mapsto trE\ e_2\ (\rho + [x_1 \mapsto 1])] \\
 & & trE\ e_1\ \rho^{++} \\
 & & \\
trE\ (\textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^n)\ \rho & = & \texttt{MATCH}\ (\rho(x))\ \overline{\mathbf{p}_i}^n \quad \&\ cs \uplus [\overline{\mathbf{p}_i \mapsto trA\ alt_i\ \rho}^n] \\
 & & \\
trE\ (\textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^n)\ \rho & = & \texttt{MATCH!}\ (\rho(x))\ \overline{\mathbf{p}_i}^n \quad \&\ cs \uplus [\overline{\mathbf{p}_i \mapsto trA\ alt_i\ \rho}^n] \\
 & & \\
trA\ (C\ \overline{x_i}^n \to e_i)\ \rho & = & trE\ e_i\ (\rho + [\overline{x_i \mapsto n-i+1}^n])
\end{array}
$$

Figure 2.20: Translation schemes from *Core-Safe* expressions to SVM instructions

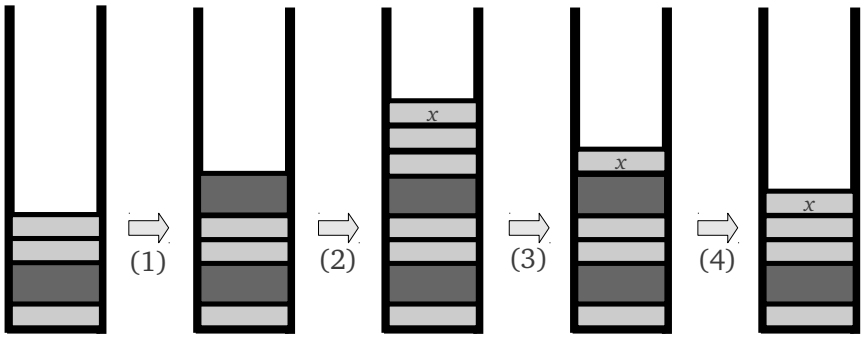


Figure 2.21: Evaluation of a **let** expression in the SVM. The white areas separate the different blocks, First, a continuation is inserted (1). Then, the evaluation of the auxiliary expression takes place until a normal form is reached and it is placed at the top of the stack (2). The previous environment is discarded (3) and the topmost continuation is removed (4), so the evaluation of the main expression begins, with the value of the bound variable at the top of the stack.

a code store. The latter is handled as an accumulator parameter of *trE* that is implicitly passed to every subexpressions being compiled. The notation & $cs \uplus [...]$ should be understood as the addition of the specified binding to the code store accumulated so far. The expression *NormalForm* ρ is a compilation macro defined as follows:

$$\begin{aligned} \mathit{NormalForm}\ \rho \quad \stackrel{\text{def}}{=} \quad & \texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) \\ & \texttt{DECREGION} \\ & \texttt{POPCONT} \end{aligned}$$

This macro is used when the expression being translated gives place to a normal form (i.e. no additional evaluations are needed). This is the case of literal, variables, copy and constructor expressions, and also the case of primitive operators.

When evaluating a **let** expression (see Figure 2.21), a continuation is inserted before evaluating the auxiliary expression e_1. During translation, every binding added to the compile-time environment ρ will be added to a separate block, which is now the topmost one, until the translation of e_1 finishes. This corresponds to the removal of topmost block and the continuation at runtime, so that the evaluation of the main expression e_2 begins.

A interesting case is that of function application (Figure 2.22). Firstly the actual parameters are inserted into the stack. Since the execution of the function being called occur in a different context, the previous environment can be discarded before the evaluation of the body of the function. The removal of the previous environment allows us to obtain constant stack space for tail-recursive functions, as the following example shows.

Example 2.17. Let us consider the following function definitions:

$$\begin{aligned} \mathit{sum}\ \mathit{xs} \quad = \quad & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & \quad [\,] \to 0 \\ & \quad (x : \mathit{xx}) \to \textbf{let}\ x_1 = \mathit{sum}\ \mathit{xx}\ \textbf{in}\ x + x_1 \end{aligned}$$

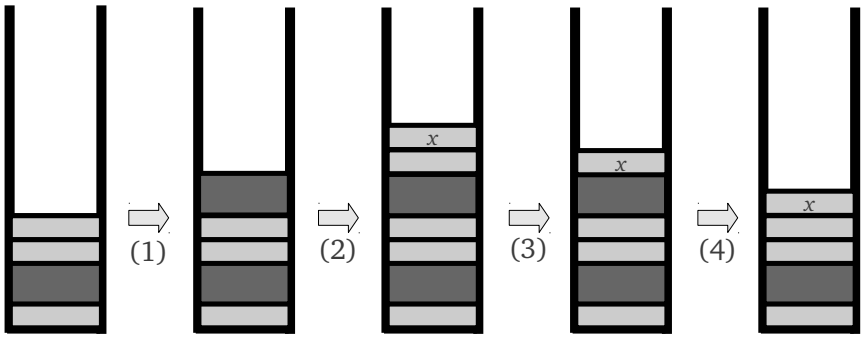


Figure 2.22: Evaluation of a function application in the SVM. The arguments of the function (dark-gray elements) are inserted at the top of the stack (1). Then, the previous environment, until the topmost continuation, is discarded in (2), so that, the only stack elements available during the evaluation of the body of the function are the arguments (corresponding to the formal parameters of the function, which are the only variables in scope before its evaluation).

$$\begin{aligned} sumAc\ xs\ ac \quad = \quad & \textbf{case}\ xs\ \textbf{of} \\ & [\,] \to ac \\ & (x : xx) \to \textbf{let}\ x_1 = x + ac\ \textbf{in}\ sumAc\ xx\ x_1 \end{aligned}$$

Function *sum* adds the elements of a list. Function *sumAc* is its tail-recursive version with an accumulator parameter *ac*. The translation function *trE* applied to *sum* returns the following SVM code:

```
1   sum :  MATCH 0 [p1, p2]          8        SLIDE 1 0
2    p1 :  BUILDENV [Lit 0]          9        CALL sum
3          SLIDE 1 1                 10   p3 : BUILDENV [Pos 0, Pos 1]
4          DECREGION                 11        PRIMOP +
5          POPCONT                   12        SLIDE 1 4
6    p2 :  PUSHCONT p3                13        DECREGION
7          BUILDENV [Pos 3]          14        POPCONT
```

Let us assume we execute *sum* with the list $[5, 7]$ given as input. In other words, we execute the SVM code starting from the *sum* label, with an initial heap $h = [p_1 \mapsto (1, (:)\ 5\ p_2), p_2 \mapsto (1, (:)\ 7\ p_3), p_3 \mapsto (1, [\,])]$, and assuming p_1 at the top of the evaluation stack. Figure 2.23 shows how the stack evolves during the execution of *sum*. The arrows between stacks contain the number of the SVM instruction being executed in each step (as labelled in the code above).

With respect to *sumAc*, the translation generates the following SVM code:

```
1   sumAc :  MATCH 0 [p1, p2]        8        PRIMOP +
2    p1 :    BUILDENV [Pos 1]        9        SLIDE 1 0
3            SLIDE 1 2               10       DECREGION
4            DECREGION               11       POPCONT
5            POPCONT                 12   p3 : BUILDENV [Pos 2, Pos 0]
6    p2 :    PUSHCONT p3             13       SLIDE 2 5
7            BUILDENV [Pos 2, Pos 5] 14       CALL sumAc
```

Under the same conditions as above, Figure 2.24 shows how the stack changes as the execution of *sumAc*

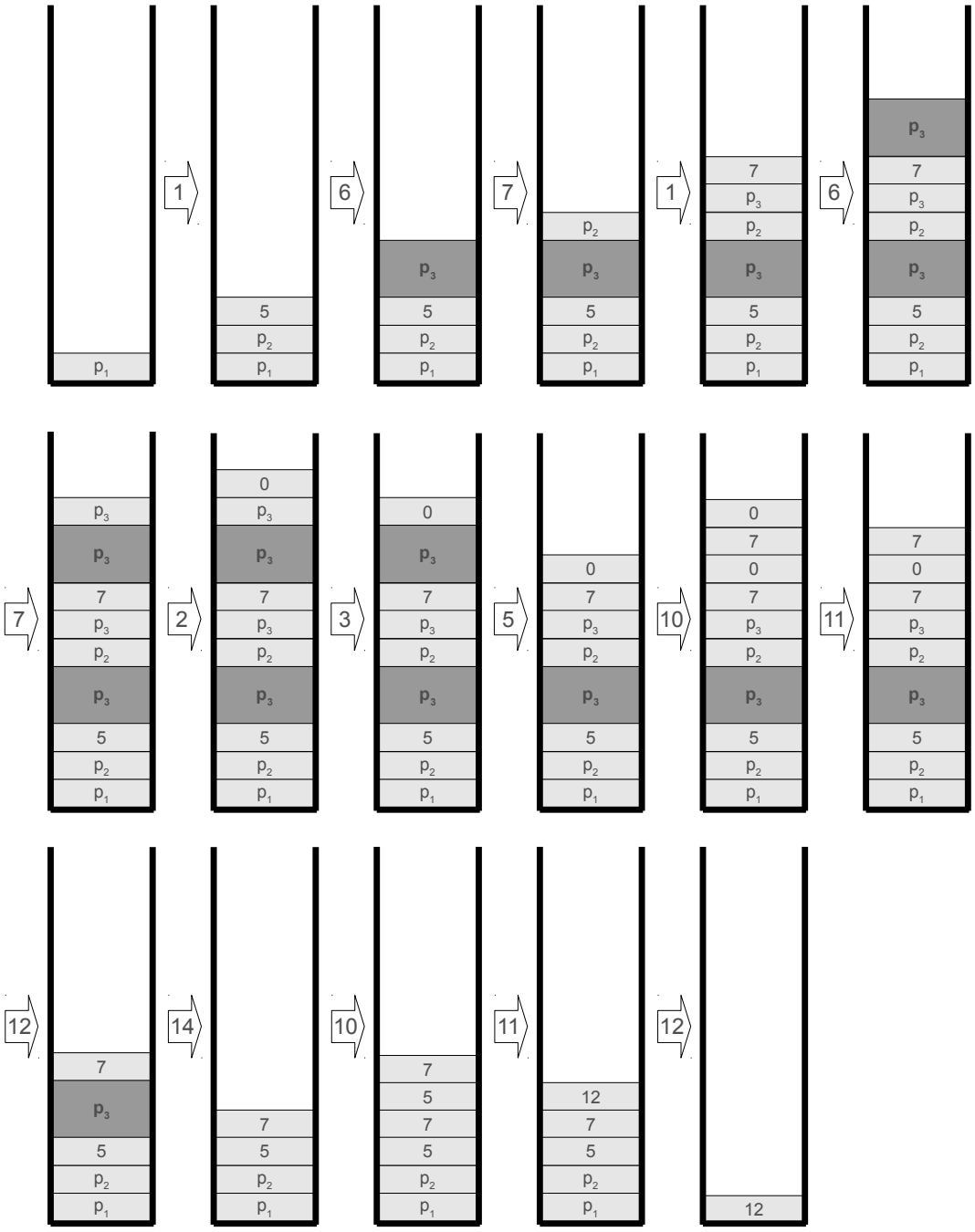


Figure 2.23: Contents of the stack during the execution of *sum*.

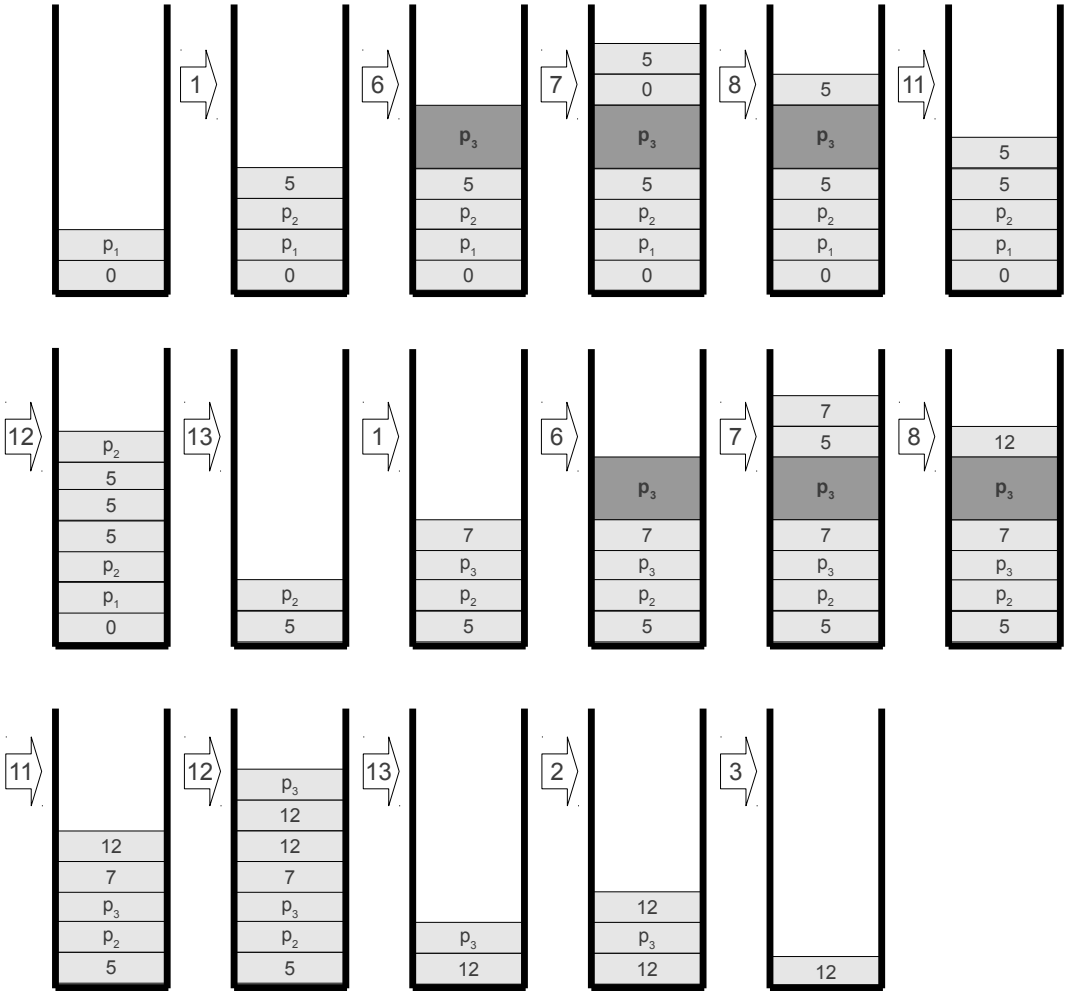


Figure 2.24: Contents of the stack during the execution of *sumAc*.

progresses. In tail-recursive functions there are no continuations left in the stack when doing a recursive call. This means we can discard the whole environment in the current function call before proceeding to the next recursive call. As a result, *sumAc* runs in constant stack space (6 words), while *sum* needs linear stack space ($5n + 1$ words, where n is the number of elements in the input list). □

Although the bytecode of the SVM is not very close to that of the Java Virtual Machine (JVM), the compiler provides a further translation phase from SVM to JVM bytecode (Figure 2.25). This is described in [38]. We have two translation phases: from *Core-Safe* to the SVM (shown in this section), and from SVM to the JVM. It is worth mentioning that both phases have been formally certified [38, 39]. The proof of the latter phase consists in establishing a bijection between the heap of the SVM and the (more complex) heap of the JVM, which is preserved along the execution of both machines. This does not only certify the correctness of the translation itself, but also certifies that the memory consumption properties and the absence of dangling pointers are preserved in the JVM.

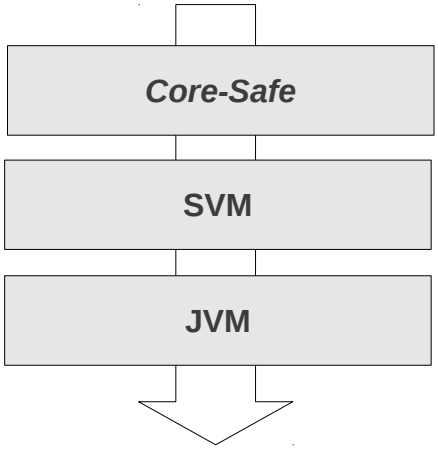


Figure 2.25: The SVM serves as an intermediate step between the *Core-Safe* code and the Java Virtual Machine (JVM). Both translations have been certified, and preserve memory costs. The translation from *Core-Safe* to SVM is deal with in this chapter. The translation from SVM to JVM is described in [38].

$$\frac{}{E \vdash h, k, td, c \Downarrow h, k, c, ([\,]_k, 0, 1)}\ [Lit]$$

$$\frac{}{E[x \mapsto v] \vdash h, k, td, x \Downarrow h, k, v, ([\,]_k, 0, 1)}\ [Var]$$

$$\frac{}{E \vdash h, k, td, a_1 \oplus a_2 \Downarrow h, k, E(a_1) \oplus E(a_2), ([\,]_k, 0, 2)}\ [PrimOp]$$

$$\frac{j \leq k \qquad (h', p') = copy(h, p, j) \qquad m = size(h, p)}{E[x \mapsto p, r \mapsto j] \vdash h, k, td, x\ @\ r \Downarrow h', k, p', ([j \mapsto m]_k, m, 2)}\ [Copy]$$

$$\frac{(g\ \overline{y_i}^n\ @\ \overline{r'_j}^l = e_g) \in \Sigma \qquad [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^l, self \mapsto k+1] \vdash h, k+1, n+l, e \Downarrow h', k+1, v, (\delta, m, s)}{E \vdash h, k, td, g\ \overline{a_i}^n\ @\ \overline{r_j}^l \Downarrow h' \mid_k, k, v, (\delta \mid_k, m, \max\{n+l, s+n+l-td\})}\ [App]$$

$$\frac{E(r) = j \qquad j \leq k \qquad fresh_h(p)}{E \vdash h, k, td, C\ \overline{a_i}^n\ @\ r \Downarrow h \uplus [p \mapsto (j, C\ \overline{E(a_i)}^n)], k, v, ([j \mapsto 1]_k, 1, 1)}\ [Cons]$$

$$\frac{E \vdash h, k, 0, e_1 \Downarrow h', k, v_1, (\delta_1, m_1, s_1) \qquad E \cup [x_1 \mapsto v_1] \vdash h', k, td+1, e_2 \Downarrow h'', k, v, (\delta_2, m_2, s_2)}{E \vdash h, k, td, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow h'', k, v, (\delta_1 + \delta_2, \max\{m_1, |\delta_1| + m_2\}, \max\{2 + s_1, 1 + s_2\})}\ [Let]$$

$$\frac{C = C_r \qquad E(x) = p \qquad E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, td + n_r, e_r \Downarrow h', k, v, (\delta, m, s)}{E \vdash h[p \mapsto (j, C\ \overline{v_i}^{n_r})], k, td, \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n} \Downarrow h', k, v, (\delta, m, s + n_r)}\ [Case]$$

$$\frac{C = C_r \qquad E(x) = p \qquad E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, td + n_r, e_r \Downarrow h', k, v, (\delta, m, s)}{E \vdash h \uplus [p \mapsto (j, C\ \overline{v_i}^{n_r})], k, td, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n} \Downarrow h', k, v, (\delta + [j \mapsto -1]_k, \max\{0, m-1\}, s + n_r)}\ [Case!]$$

Figure 2.26: Resource-aware operational semantics of *Core-Safe* expressions.

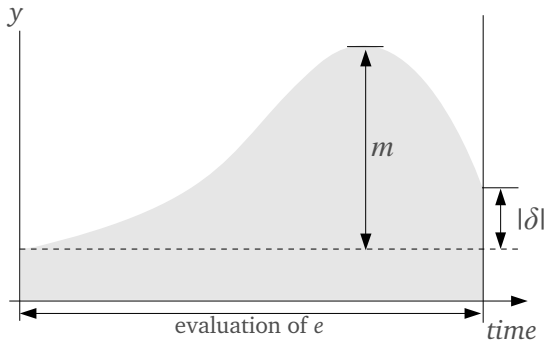


Figure 2.27: Intuitive meaning of δ and m components in the resource vector. The y coordinate represents the number of cells in the heap.

2.7 Resource-aware semantics

Once the resource consumption of each instruction of the SVM is known, we enrich the big-step semantics given in Section 2.4.1 with a resource vector (δ, m, s), which can be conceived as a side effect of evaluating an expression e.

- The first component is a partial function $\delta : \mathbb{N} \to \mathbb{Z}$ giving, for each region k, the signed difference between the number of cells after and before evaluating the expression. A positive difference means that more cells have been created than destroyed. A negative one describes the opposite situation.
- The component m is a natural number describing the *minimum* number of fresh cells in the heap needed to successfully evaluate e. This number corresponds to the maximum amount of cells existing simultaneously in memory during the evaluation of this expression.
- The component s is a natural number whose meaning is analogous to that of the m component. The s component describes the minimum size of stack (in words) needed for the evaluation of the expression.

Figure 2.27 gives an intuition on the meaning of the first two components. Assume the evaluation of an expression e. The figure represents the global amount of cells in memory as the evaluation of e proceeds. In this case, the evaluation of e reclaims memory until some time point, from which memory is disposed of. The m value represents the maximum amount of memory taken during the evaluation of e, whereas δ represents the difference of memory amount between the initial and final heaps. Notice, however, that the δ contains this difference *for every region* in the heap. What is represented in Figure 2.27 is the global balance $|\delta|$ of heap cells between the final and initial heaps, which will be formally defined below. Also notice that both values m and δ are relative to the memory consumption level at the beginning of the evaluation of e (dashed line in Figure 2.27).

If we represented the stack consumption in the style of Figure 2.27, the s component of the resource vector would take the role of the m component in the heap consumption. There is no component for denoting the difference in stack words between the final and initial heaps, because the final stack contains the same elements of the initial stack with an additional element at the top, which is the result of evaluating the expression, so this difference is always 1.

The domain of δ is the set $\{0..k\}$, where k is the number of regions in the heap to which the δ refers. The notation $[\ ]_k$ stands for the function $[i \mapsto 0 \mid i \in \{0..k\}]$, whereas $[i \mapsto n]_k$ abbreviates the function

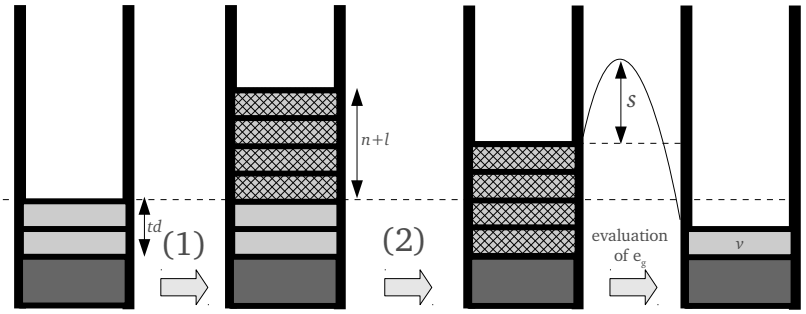


Figure 2.28: Stack consumption while evaluating a function application: we have to take the maximum between the number of arguments pushed onto the stack (1) and the maximum stack level reached during evaluation of the function's body, assuming that the arguments are already in the stack and the previous environment has been discarded (2).

$[i \mapsto n] \uplus [j \mapsto 0 \mid j \in \{0..k\} \setminus \{i\}]$. The *total balance* of cells (denoted by $|\delta|$) is the sum of the balances obtained in each region:

$$|\delta| \stackrel{\text{def}}{=} \sum_{i \in \text{dom}\, \delta} \delta(i)$$

The notation $\delta_1 + \delta_2$ represents the component-wise addition of δ_1 and δ_2, provided these have the same domain:

$$\delta_1 + \delta_2 \stackrel{\text{def}}{=} [\delta_1(i) + \delta_2(i) \mid i \in \text{dom}\, \delta_1 \cap \text{dom}\, \delta_2]$$

The enriched semantic rules are shown in Figure 2.26. In addition to the resource vector (δ, m, s), we need a new component td in order to simulate the *topDepth* function of compile time environments. This component represents the number of stack words inserted after the topmost continuation, and it influences the s, since an amount td of words are removed from the stack before function calls.

The evaluation of atom (rules [*Lit*] and [*Var*]) does not require memory consumption, and it requires a stack word space to push the result into. The evaluation of a primitive operator application requires two stack words, since the operands have to be pushed into the stack before computing the result. The evaluation of a copy expression [*Copy*] requires as many heap cells as the size of the recursive spine of the structure being copied. The *size* function defines this notion of size:

$$size(h[p \mapsto (j, C\ \overline{v_i}^n)], p) \stackrel{\text{def}}{=} 1 + \sum_{i \in RecPos(C)} size(h, v_i)$$

In rule [*App*], by $\delta|_k$ we mean a function like δ but restricted to the domain $\{0..k\}$. The computation $\max\{n + l, s + n + l - td\}$ of fresh stack words takes into account that the first $n + l$ words are needed to store the actual arguments, then the current environment of length td is discarded, and then the function body is evaluated (Figure 2.28).

In rule [*Let*], a continuation (2 words) is stacked before evaluating e_1, and this a leaves a value in the stack before evaluating e_2. That is why we obtain $\max\{2 + s_1, 1 + s_2\}$ as stack consumption (see Figure 2.29). With regard to the heap consumption, we take the maximum between the memory needs of e_1 and those of e_2, but taking the balance $|\delta_1|$ into account (Figure 2.30).

Example 2.18. In Figure 2.31 we show the resource vector corresponding to the execution of most of the examples in this chapter. The concrete input DSs are shown above the table. We assume that these DSs have been created before evaluating each function call, so their building costs are not reflected in

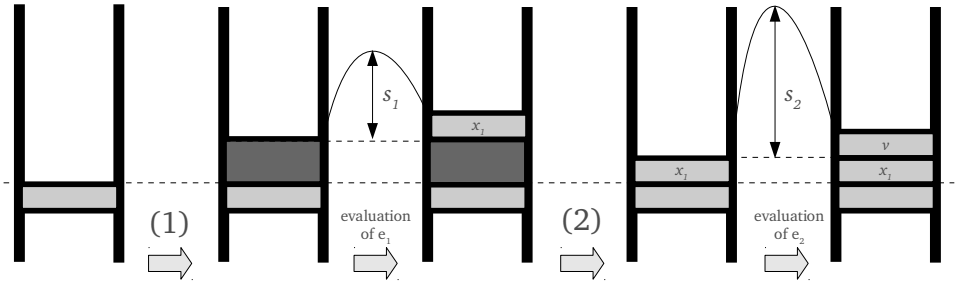


Figure 2.29: Stack consumption while evaluating a **let** expression: we have to take the maximum between the execution of e_1 assuming that a continuation has been pushed before into the stack (1), and the execution of e_2 assuming that the value of x_1 has been pushed after discarding the continuation (2).

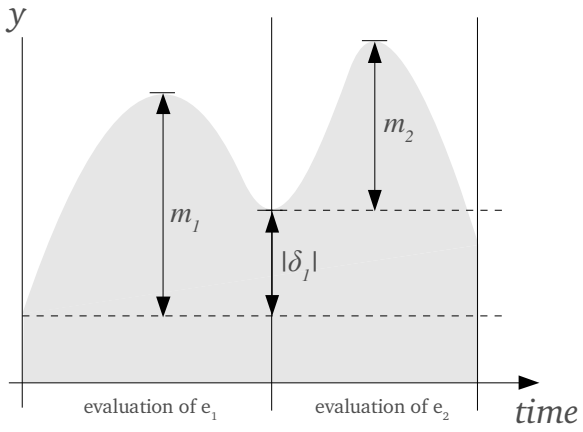


Figure 2.30: Heap consumption of two expressions executed in sequence (**let**). The execution of the first expression leaves $|\delta_1|$ cells in the heap. We take this point as the reference level on which we measure the memory needs of the second expression m_2. Hence, the memory needs of the whole sequence is given by $\sqcup\{m_1, |\delta_1| + m_2\}$.

$xs = [1,2,3]$ (4 cells) $\quad ys = [4,5]$ (3 cells) $\quad zs = [5,4,3,2,1]$ (6 cells)

$t = Node\ (Node\ Empty\ 2\ Empty)\ 4\ (Node\ Empty\ 7\ Empty)$ (7 cells)

Expression	δ	m	s
append xs ys @ r	$[E(r) \mapsto 3]_k$	3	23
appendC xs ys @ r	$[E(r) \mapsto 6]_k$	6	24
appendD xs ys @ r	$[E(r) \mapsto -1]_k$	0	23
insert 10 *xs @ r*	$[E(r) \mapsto 5]_k$	5	32
insertD 10 *xs @ r*	$[E(r) \mapsto 1]_k$	1	32
inssort zs @ r	$[E(r) \mapsto 21]_k$	21	41
inssortD zs @ r	$[E(r) \mapsto 0]_k$	0	41
mkTree zs @ r	$[E(r) \mapsto 11]_k$	11	50
inorder t @ r	$[E(r) \mapsto 4]_k$	5	29
treesort zs @ r	$[E(r) \mapsto 6]_k$	18	62
sumAc xs 0	$[\,]_k$	0	6
sumAc zs 0	$[\,]_k$	0	6

Figure 2.31: Memory consumption results. The table shows, for each expression, the resource vector (δ, m, s) resulting from its evaluation.

the table.

Function *append* creates as many cells in the output region as the number of cons cells in the list passed as first parameter. In *appendC* we need three additional cells for copying the list passed as second parameter. Function *appendD* destroys four cells of the input list and builds three in the output region. Its execution needs no additional heap space. The heap cost of *insert* and *inssort* is proportional to the length of the input list, whereas in *insertD* and *inssortD* this cost is constant. In the case of *insertD* we need an additional cell for storing the new element. Function *treesort* leaves in the output region as many cells as the input list. However, more cells are needed in order to build the intermediate tree. Finally, the calls to *sumAc* produce no heap costs and a constant stack cost. □

2.8 Correctness of the translation into SVM

Now we show that the pair translation-abstract machine is sound and complete with respect the semantics defined in last section. The correctness of the translation involves the proof of the following facts:

1. Both big-step semantics and SVM lead to the same result and final heap, whenever their initial configurations are *equivalent*.

2. The (δ, m, s) vector of the semantics models the actual memory needs of an expression when being evaluated in the SVM.

The first fact proves that the translation from *Core-Safe* to SVM preserves the *semantics* of a given program, whereas the second one shows the preservation of its *memory needs*. In order to prove these results, we must, on the one hand, make precise the idea of a semantic configuration being equivalent to a SVM configuration. On the other hand, we have to determine the memory costs of a program when being run in the SVM. First, we note that both the semantics and the SVM machine rules are syntax driven, and that their computations are deterministic (up to fresh names generation).

Lemma 2.19. *Given a value environment E, an initial heap h with k regions, a natural number td, and a Core-Safe expression e, if $E \vdash h, k, td, e \Downarrow h, k, v, (\delta, m, s)$ and $E \vdash h, k, td, e \Downarrow h', k, v', (\delta', m', s)$, then $h = h'$ (up to pointer renaming), $v = v'$, and $(\delta, m, s) = (\delta', m', s')$*

Proof. By induction on the $\Downarrow$- derivation. All cases are straightforward. □

Lemma 2.20. *Given an initial configuration c_{init} such that $c_{init} \to c_1$ and $c_{init} \to c_1'$. Then $c_1 = c_1'$ (modulo pointer renaming).*

Proof. By case distinction on the rule being applied. All cases are straightforward. □

The main difference between the big-step operational semantics of Section 2.4.1 and the SVM machine is the way in which value environments are represented. In the big-step semantics we have a mapping E from variables to values, whereas in the SVM we have a stack. The correspondence between variables and positions of the stack is given by the ρ environment used in the translation (see Section 2.6).

Obviously, we cannot ensure the equivalence of big-step semantics and SVM if their starting point (i.e. mapping E in big-step semantics, stack S in the SVM) denote different value environments. This is the motivation for the following:

Definition 2.21. We say that the environment E and the pair (ρ, S) are equivalent, denoted $E \equiv (\rho, S)$, if $\mathrm{dom}\, E = \mathrm{dom}\, \rho$, and $\forall x \in \mathrm{dom}\, \rho \backslash \{self\}\, .\, E(x) = S!(\rho(x))$.

A stack S' is said to be a *suffix* of S if there exists a number $n \geq 0$ of stack elements b_i such that $S = \overline{b_i}^n : S'$. Given a SVM configuration $c_0 = (is, h, k_0, k, S, cs)$ and S' a suffix of S, we denote by $c_0 \to_{S'}^* c_n$ a derivation in which all the stacks in intermediate configurations have S' as a suffix. Should the topmost instruction of a configuration create a stack smaller than S', then the machine would stop at that configuration.

Now we follow with some definitions regarding the maximum memory consumption produced by a SVM program.

Definition 2.22. The function *sizeST*, which returns the size (in words) of a stack, is defined as follows:

$$
\begin{array}{ll}
sizeST([\,]) & = 0 \\
sizeST(v : S) & = 1 + sizeST(S) \text{ if } v \text{ is not a continuation.} \\
sizeST((k_0, \mathbf{p}) : S) & = 2 + sizeST(S)
\end{array}
$$

Analogously, the size (in cells) of a heap is given by the function *sizeH*:

$$sizeH(h) = |\mathrm{dom}\, h|$$

The reason for having $2 + sizeST(S)$ in the case where a continuation is at the top of the stack is that any sensible implementation of the SVM would need two words for storing the pair $(k_0, \mathbf{p})$.

We shall extend the notation of these functions to configurations, so $sizeST(c)$ (resp. $sizeH(c)$) will be used to denote the size of the stack (resp. heap) of the given configuration c.

Given $c_0 = (is, h, k_0, k, S, cs)$ and $c_0 \to_{S'} \cdots \to_{S'} c_n$ a SVM derivation, the *maximum number of fresh cells* of the derivation (denoted $maxFreshCells(c_0 \to_{S'}^* c_n)$) is the highest difference in cells between the

heaps of the configurations $c_0, \ldots c_n$ and the initial heap h. Likewise, we define the *maximum number of fresh words* created in the stack S, denoted $maxFreshWords(c_0 \rightarrow^*_{S'} c_n)$. Finally, by $diff(k, h, h')$ we denote a function giving for each region in $\{0, \ldots, k\}$ the signed difference in cells between h' and h.

Definition 2.23. Given a derivation $c_0 \rightarrow_{S'} \ldots c_i \ldots \rightarrow_{S'} c_n$, we define:

$$\begin{array}{rcl} maxFreshCells(c_0 \rightarrow^*_{S'} c_n) & = & \max\{sizeH(c_i) - sizeH(c_0) \mid 0 \leq i \leq n\} \\ maxFreshWords(c_0 \rightarrow^*_{S'} c_n) & = & \max\{sizeST(c_i) - sizeST(c_0) \mid 0 \leq i \leq n\} \end{array}$$

From these definitions the following properties can be easily obtained:

$$\begin{array}{rl} maxFreshCells(c_0 \rightarrow^*_{S'} c_i \rightarrow^*_{S'} c_n) = \max\{ & maxFreshCells(c_0 \rightarrow^*_{S'} c_i), \\ & maxFreshCells(c_i \rightarrow^*_{S'} c_n) \\ & \quad + sizeH(c_i) - sizeH(c_0) \quad \} \end{array}$$

$$\begin{array}{rl} maxFreshWords(c_0 \rightarrow^*_{S'} c_i \rightarrow^*_{S'} c_n) = \max\{ & maxFreshWords(c_0 \rightarrow^*_{S'} c_i), \\ & maxFreshWords(c_i \rightarrow^*_{S'} c_n) \\ & \quad + sizeST(c_i) - sizeST(c_0) \quad \} \end{array}$$

Definition 2.24. Given two heaps h and h' and a number k of regions, we denote by $diff(k, h, h')$ a function δ such that $\mathrm{dom}\ \delta = \{0..k\}$ and:

$$\forall i \in \{0..k\} \,.\, \delta(i) = \left|\{p \in \mathrm{dom}\ h' \mid region(h'(p)) = i\}\right| - \left|\{p \in \mathrm{dom}\ h \mid region(h(p)) = i\}\right|$$

The definitions of *diff* and *sizeH* are related by the following property, which is easy to establish from the corresponding definitions. Let h and h' be two heaps with k regions. Then:

$$sizeH(h') - sizeH(h) = \sum_{i=0}^{k} diff(i, h, h')$$

The next Lemma shows some properties of the *copy* function. In particular, that the size of the copy and the DS being copied are the same.

Lemma 2.25. *If $(h', p') = copy(h, p, j)$ then:*

1. $h \subseteq h'$
2. $size(h', p') = size(h, p)$
3. $sizeH(h') - sizeH(h) = size(h', p')$
4. $\forall k \geq j, diff(k, h, h') = [j \mapsto size(h', p')]_k$

Proof. Straightforward induction on the size of the DS pointed to by p. □

Now we prove that, if we translate an expression e into a sequence is of SVM instructions and the SVM executes this sequence, then the number of regions in the heap of the final configuration is equal to the k_0 component of the initial configuration.

Lemma 2.26. *For all $S, S', h, h', td, k_0, k, e, v, \rho, cs, cs'$ of their respective types, such that*

$$\begin{array}{ll} td = topDepth(\rho) & (is, cs) = trE\ e\ \rho \\ S' = drop\ td\ S & cs' \supseteq cs \\ k_0 \leq k & \end{array}$$

If $\underbrace{(is, h, k_0, k, S, cs')}_{c_{init}} \rightarrow^*_{S'} \underbrace{([\texttt{POPCONT}], h', k', k', v : S', cs')}_{c_{final}}$ *then* $k' = k_0$.

Proof. This can be trivially proved by induction on the length of $\rightarrow^*_{S'}$ and by cases over the expression e whose translation is executed. □

The following theorem establishes the correctness of the translation, showing that the computed value and the resource consumption of a given expression e in the big-step semantics are the same as those obtained by executing in the SVM the translation of e (see Appendix C).

We denote by *drop n S* the stack resulting from removing the n topmost elements of S. That is, *drop* n $(\overline{b_i}^n : S) = S$ and undefined in case the number of elements in the input stack is less than n.

Theorem 2.27. *For all* $S, S', E, h, h', td, k_0, k, e, v, \delta, m, s, \rho, cs, cs'$ *of their respective types, if*

$$\begin{array}{llll} E \equiv (\rho, S) & (is, cs) = trE\ e\ \rho & td = topDepth(\rho) & E \text{ admissible w.r.t. } k \\ S' = drop\ td\ S & cs' \supseteq cs & k_0 \leq k & \end{array}$$

then $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ *if and only if*

1. $\underbrace{(is, h, k_0, k, S, cs')}_{c_{init}} \rightarrow^*_{S'} \underbrace{([\texttt{POPCONT}], h' \mid_{k_0}, k_0, k_0, v : S', cs')}_{c_{final}}$
2. $\delta = diff(k, h, h')$
3. $m = maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final})$
4. $s = maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final})$

Proof. The ($\Rightarrow$) direction is shown by induction on the $\Downarrow$-derivation. The ($\Leftarrow$) direction of the theorem can be proved by induction on the length of the $\rightarrow^*_{S'}$ derivation. In booth proofs we distinguish cases depending on the expression being translated.

- **Case** [*Lit*]: $e \equiv c$

  Let $(is, cs) = trE\ e\ \rho$ be the SVM code generated, where $cs = [\,]$ and:

$$is = \texttt{BUILDENV}\ [c] : \overbrace{\texttt{SLIDE}\ 1\ (topDepth(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_3}}_{is_2}}^{is_1}$$

  In addition, by the rule [*Lit*] we have $E \vdash h, k, td, c \Downarrow h, k, c, ([\,]_k, 0, 1)$. For all cs' we get the following derivation:

$$\begin{array}{ll} & \overbrace{(\texttt{BUILDENV}\ [c] : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\ \rightarrow_{S'} & \\ & (\texttt{SLIDE}\ 1\ (topDepth(\rho)) : is_2,\ h,\ k_0,\ k,\ c : S,\ cs') \\ \rightarrow_{S'} & \quad \{\text{since } td = topDepth(\rho)\} \\ & (\texttt{DECREGION} : is_3,\ h,\ k_0,\ k,\ c : drop\ td\ S,\ cs') \\ \rightarrow_{S'} & \end{array}$$

$$\underbrace{(\texttt{[POPCONT]},\ h|_{k_0},\ k_0,\ k_0,\ c : \mathit{drop}\ \mathit{td}\ S,\ cs')}_{c_{\mathit{final}}}$$

Because of the assumption $S' = \mathit{drop}\ \mathit{td}\ S$ the property (1) holds. Moreover, we get $\mathit{diff}(k,h,h) = [i \mapsto 0 \mid 0 \leq i \leq k] = \delta$, so property (2) holds. Finally, both properties (3) and (4) hold because $\mathit{maxFreshCells}(c_{\mathit{init}} \to^*_{S'} c_{\mathit{final}}) = 0$ and $\mathit{maxFreshWords}(c_{\mathit{init}} \to^*_{S'} c_{\mathit{final}}) = 1$.

- **Case** [*Var*]: $e \equiv x$

  The proof is similar to the case $e \equiv c$. Now $E[x \mapsto v] \vdash h,k,\mathit{td},x \Downarrow h,k,v,([\,]_k,0,1)$ holds and the code generated is as follows:

$$\mathit{is} = \texttt{BUILDENV}\ [\rho(x)] : \overbrace{\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{\mathit{is}_3}}_{\mathit{is}_2}}^{\mathit{is}_1}$$

  From which the following derivation is obtained:

$$\begin{array}{ll} & \overbrace{(\texttt{BUILDENV}\ [\rho(x)] : \mathit{is}_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{\mathit{init}}} \\ \to_{S'} & \quad \{\text{since } S!(\rho(x)) = E(x) = v\} \\ & (\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : \mathit{is}_2,\ h,\ k_0,\ k,\ v : S,\ cs') \\ \to_{S'} & \quad \{\text{since } \mathit{td} = \mathit{topDepth}(\rho)\} \\ & (\texttt{DECREGION} : \mathit{is}_3,\ h,\ k_0,\ k,\ v : \mathit{drop}\ \mathit{td}\ S,\ cs') \\ \to_{S'} & \\ & \underbrace{(\texttt{[POPCONT]},\ h|_{k_0},\ k_0,\ k_0,\ v : \mathit{drop}\ \mathit{td}\ S,\ cs')}_{c_{\mathit{final}}} \end{array}$$

  Hence property (1) holds. The proof of (2), (3) and (4) is analogous to the case $e \equiv c$.

- **Case** [*Copy*]: $e \equiv x\ @\ r$

  On the one hand, by using the [*Copy*] rule we know that:

$$E \vdash h,k,\mathit{td},x\ @\ r \Downarrow h',k,p',([j \mapsto \mathit{size}(h,p)]_k, \mathit{size}(h,p), 2)$$

  where $E(x) = p$, $E(r) = j \leq k$ and $(h',p') = \mathit{copy}(h,p,j)$. On the other hand, we obtain $(\mathit{is},[\,]) = \mathit{trE}\ e\ \rho$ where:

$$\mathit{is} = \texttt{BUILDENV}\ [\rho(x),\rho(r)] : \overbrace{\texttt{COPY} : \overbrace{\texttt{SLIDE}\ 1\ (\mathit{topDepth}(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{\mathit{is}_4}}_{\mathit{is}_3}}^{\mathit{is}_2}}^{\mathit{is}_1}$$

  For any cs', the first step of the $\to^*_{S'}$ derivation is as follows:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [\rho(x),\rho(r)] : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\text{ since } E(x) = p = S!(\rho(x))\ \} \\
 & (\texttt{COPY} : is_2,\ h,\ k_0,\ k,\ p : Item_k(\rho(r)) : S,\ cs')
\end{array}
$$

Let us proceed by case distinction: on one hand, if $r \neq self$ then $Item_k(\rho(r)) = S!(\rho(r)) = E(r) = j$. On the other hand, if $r = self$ then we have $Item_k(\rho(r)) = Item_k(self) = k$, but since $E(self) = k$ (by Proposition 2.15) and $E(self) = j$ (by rule [*Copy*]), we have that $Item_k(\rho(r)) = j$. Hence the current machine configuration can be rewritten as $(\texttt{COPY} : is_2, h, k_0, k, p : j : S, cs')$.

We shall now resume the $\rightarrow^*_{S'}$ derivation:

$$
\begin{array}{ll}
 & (\texttt{COPY} : is_2,\ h,\ k_0,\ k,\ p : j : S,\ cs') \\
\rightarrow_{S'} & \quad \{\text{ since } (h', p') = copy(h, p, j) \text{ and } j \leq k\} \\
 & (\texttt{SLIDE}\ 1\ (topDepth(\rho)) : is_3,\ h',\ k_0,\ k,\ p' : S,\ cs') \\
\rightarrow_{S'} & \quad \{\text{since } td = topDepth(\rho)\} \\
 & (\texttt{DECREGION} : is_4,\ h',\ k_0,\ k,\ p' : drop\ td\ S,\ cs') \\
\rightarrow_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\ h'|_{k_0},\ k_0,\ k_0,\ p' : drop\ td\ S,\ cs')}_{c_{final}}
\end{array}
$$

Therefore (1) holds, because of the assumption $S' = drop\ td\ S$. Properties (2) and (3) follow from Lemma 2.25. With respect to (4), from the resulting $\rightarrow^*_{S'}$ derivation it can be easily shown that $maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final}) = 2$.

- **Case** [*PrimOp*]: $e \equiv a_1 \oplus a_2$

  The translation yields the following instruction sequence:

$$
is = \overbrace{\texttt{BUILDENV}\ [\rho(a_1),\rho(a_2)] : \texttt{PRIMOP} \oplus : \overbrace{\texttt{SLIDE}\ 1\ (topDepth(\rho)) : \underbrace{\texttt{DECREGION} : \overbrace{[\texttt{POPCONT}]}^{is_4}}_{is_3}}^{is_2}}^{is_1}
$$

  By executing it, we obtain the following derivation:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [\rho(a_1),\rho(a_2)] : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\text{since } S!(\rho(a_i)) = E(a_i) \text{ for every } i \in \{1..2\}, \text{ see below }\} \\
 & (\texttt{PRIMOP} \oplus : is_2,\ h,\ k_0,\ k,\ E(a_1) : E(a_2) : S,\ cs') \\
\rightarrow_{S'} & \\
 & (\texttt{SLIDE}\ 1\ (topDepth(\rho)) : is_3,\ h,\ k_0,\ k,\ (E(a_1) \oplus E(a_2)) : S,\ cs') \\
\rightarrow_{S'} & \\
 & (\texttt{DECREGION} : is_4,\ h,\ k_0,\ k,\ (E(a_1) \oplus E(a_2)) : drop\ td\ S,\ cs') \\
\rightarrow_{S'} &
\end{array}
$$

$$([\texttt{POPCONT}],\ h|_{k_0},\ k_0,\ k_0,\ (E(a_1) \oplus E(a_2)) : drop\ td\ S,\ cs') \equiv c_{final}$$

Hence, property 1 holds. From the initial and final configurations we obtain $diff(k, h, h) = [i \mapsto 0 \mid 0 \leq i \leq k] = \delta$, so property (2) holds. Properties (3) and (4) hold because $maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final}) = 0$ and $maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final}) = 2$.

- **Case** [*Cons*]: $e \equiv C\ \overline{a_i}^n\ @\ r$

  In the big-step semantics we get $E \vdash h, k, e \Downarrow h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)], k, p, ([E(r) \mapsto 1]_k, 1, 1)$ where p is a fresh pointer. The translation yields the following instruction sequence:

$$is = \texttt{BUILDCLS}\ C\ [\overline{\rho(a_i)}^n]\ \rho(r) : \overbrace{\texttt{SLIDE}\ 1\ (topDepth(\rho)) : \underbrace{\texttt{DECREGION} : \overbrace{[\texttt{POPCONT}]}^{is_3}}_{is_2}}^{is_1}$$

  By executing this, we obtain the following derivation:

$$\begin{array}{ll} & \overbrace{(\texttt{BUILDCLS}\ C\ [\overline{\rho(a_i)}^n]\ \rho(r) : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\ \rightarrow_{S'} & \quad \{\text{since } S!(\rho(a_i)) = E(a_i) \text{ for every } i \in \{1..n\}, \text{ see below }\} \\ & (\texttt{SLIDE}\ 1\ (topDepth(\rho)) : is_2,\ h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)],\ k_0,\ k,\ p : S,\ cs') \\ \rightarrow_{S'} & \quad \{\text{since } td = topDepth(\rho)\} \\ & (\texttt{DECREGION} : is_3,\ h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)],\ k_0,\ k,\ p : drop\ td\ S,\ cs') \\ \rightarrow_{S'} & \\ & \underbrace{([\texttt{POPCONT}],\ (h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)])|_{k_0},\ k_0,\ k_0,\ p : drop\ td\ S,\ cs')}_{c_{final}} \end{array}$$

  In the first step we have assumed that $Item_k(\rho(r)) = E(r)$. The proof of this is similar to that seen in the [*Copy*] rule. Thus (1) holds. From the initial and final configurations we obtain:

$$diff(k, h, h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)]) = [E(r) \mapsto 1]_k$$

$$maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final}) = 1$$

$$maxStackWords(c_{init} \rightarrow^*_{S'} c_{final}) = 1$$

  from which (2), (3) and (4) follow trivially.

- **Case** [*App*]: $e \equiv g\ \overline{a_i}^n\ @\ \overline{r_j}^l$

  We assume, by rule [*App*] that $E_g \vdash h, k+1, n+l, e_g \Downarrow h', k+1, v, (\delta, m, s)$, where E_g is defined as follows:

$$E_g = [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^l, self \mapsto k+1]$$

  and $(g\ \overline{y_i}^n\ @\ \overline{r'_j}^l = e_g) \in \Sigma$. Furthermore, let (is_g, cs_g) be the result of $trE\ e_g\ \rho_g$, where e_g is the body of the function to be called and ρ_g is defined as follows:

$$\rho_g = [(\overline{[r'_j \mapsto (l-j)+1}^l, \overline{y_i \mapsto (n-i)+(l+1)}^n], n+l, 0)]$$

Assume that $cs(\mathbf{p}) = is_g$. We obtain $(is, cs) = trE\ e\ \rho$, where:

$$is = \texttt{BUILDENV}\ [\overline{\rho(a_i)}^{n}, \overline{\rho(r_j)}^{l}] : \overbrace{\texttt{SLIDE}\ (n+l)\ (topDepth(\rho)) : \underbrace{[\texttt{CALL}\ \mathbf{p}]}_{is_2}}^{is_1}$$

For any cs' such that $cs' \supseteq cs$, the code generated leads to the following SVM derivation:

$$\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [\overline{\rho(a_i)}^{n}, \overline{\rho(r_j)}^{l}] : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\
\rightarrow_{S'} & \\
 & (\texttt{SLIDE}\ (n+l)\ (topDepth(\rho)) : is_2,\ h,\ k_0,\ k,\ \overline{S!(\rho(a_i))}^{n} : \overline{S!(\rho(r_j))}^{l} : S,\ cs') \\
\rightarrow_{S'} & \quad \{\ \text{since } td = topDepth(\rho)\ \} \\
 & ([\texttt{CALL}\ \mathbf{p}],\ h,\ k_0,\ k,\ \overline{S!(\rho\ a_i)}^{n} : \overline{S!(\rho\ r'_j)}^{l} : drop\ td\ S,\ cs') \\
\rightarrow_{S'} & \quad \{\ \text{since } cs'(\mathbf{p}) = cs(\mathbf{p}) = is_g\ \} \\
 & (is_g,\ h,\ k_0,\ k+1,\ \overline{S!(\rho\ a_i)}^{n} : \overline{S!(\rho\ r'_j)}^{l} : drop\ td\ S,\ cs') \equiv c_f \\
\rightarrow^{*}_{S'} & \quad \{\ \text{by i.h. (see below) and } k_0 \leq k\ \} \\
 & \underbrace{([\texttt{POPCONT}],\ h'|_{k_0},\ k_0,\ k_0,\ v : drop\ td\ S,\ cs')}_{c_{final}}
\end{array}$$

where $S' = drop\ td\ S$. In order to apply the induction hypothesis in the last step, we prove that every assumption of the theorem holds for the $\Downarrow$ derivation corresponding to e_f. The only nontrivial assumption to prove is that $E_g \equiv (\rho_g, S_g)$, denoting by S_g the result from pushing the actual parameters $\overline{S!(\rho\ a_i)}^{n}$ and $\overline{S!(\rho\ r'_j)}^{l}$ into S. Firstly, let $x \in \operatorname{dom} \rho_g$. We prove that $S_g!(\rho_g(x)) = E_g(x)$. On the one hand, if $x = y_i$ for some $i \in \{1..n\}$, we get $\rho_g(x) = n + l - [(n-i) + (l+1)] = i - 1$. Hence:

$$S_g!(\rho_g(x)) = S_g!(i-1) = S!(\rho(a_i)) = E(a_i) = E_g(y_i)$$

On the other hand, if $x = r'_j$ for some $j \in \{1..l\}$ then we get $\rho_g(x) = n + l - [(l-j)+1] = n + j - 1$ and:

$$S_g!(\rho_g(x)) = S_g!(n+j-1) = S!(\rho(r_j)) = E(r_j) = E_g(r'_j)$$

So $E_g \equiv (\rho_g, S_g)$, and the induction hypothesis can be applied on the derivation of e_g, which proves property (1), as we have shown above. The proof for (2) can be easily established from the fact that $diff(k, h, h') = diff(k+1, h, h')|_k$ and that $\delta = diff(k+1, h, h')$ by induction hypothesis. Property (3) also follows trivially from the induction hypothesis. With respect to (4), let us denote:

$$\begin{array}{lcl}
S_1 & = & \overline{S!(\rho(a_i))}^{n} : \overline{S!(\rho(r_j))}^{l} : S \\
S_2 & = & \overline{S!(\rho(a_i))}^{n} : \overline{S!(\rho(r_j))}^{l} : drop\ td\ S \\
S_3 & = & v : drop\ td\ S
\end{array}$$

$$\begin{array}{rl}
maxFreshWords(c_{init} \rightarrow^{*}_{S'} c_f) = & \max\{sizeST(S_1) - sizeST(S), \\
 & \quad sizeST(S_2) - sizeST(S)\} \\
= & \{n+l, n+l-td\}
\end{array}$$

$$= n + l$$

On the other hand, $maxFreshWords(c_f \rightarrow^*_{S'} c_{final}) = s$ by induction hypothesis. Therefore:

$$\begin{aligned} maxFreshWords(c_{init} \rightarrow^*_{S'} c_f \rightarrow^*_{S'} c_{final}) &= \max\{maxFreshWords(c_{init} \rightarrow^*_{S'} c_f), \\ &\qquad maxFreshWords(c_f \rightarrow^*_{S'} c_{final}) \\ &\qquad + sizeST(S_2) - sizeST(S)\} \\ &= \max\{n + l, s + n + l - td\} \end{aligned}$$

- **Case** [*Let*]: $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2$

  By rule [*Let*] we get:

$$\begin{aligned} &E \vdash h, k, 0, e_1 \Downarrow h', k, v_1, (\delta_1, m_1, s_1) \\ &E_1 \vdash h', k, td + 1, e_2 \Downarrow h'', k, v, (\delta_2, m_2, s_2) \end{aligned}$$

  Where we define $E_1 = E \uplus [x_1 \mapsto v_1]$. In addition, we assume $(is_1, cs_1) = trE\ e_1\ \rho^{++}$ and $(is_2, cs_2) = trE\ e_2\ \rho_1$ where $\rho_1 = \rho + [x_1 \mapsto 1]$. On the other hand, we get $cs \supseteq cs_1$, $cs \supseteq cs_2$ and $is = \texttt{PUSHCONT}\ \mathbf{p} : is_1$, in which $cs_1(\mathbf{p}) = is_2$. Given $cs' \supseteq cs$, the trace corresponding to the execution of e starts as follows:

$$\begin{aligned} &\overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\ \rightarrow_{S'}\ &(is_1,\ h,\ k,\ k,\ (k_0, \mathbf{p}) : S,\ cs') \equiv c_1 \end{aligned}$$

  In order to apply induction hypothesis on e_1 we have to check that $E \equiv (\rho^{++}, (k_0, \mathbf{p}) : S)$, which trivially holds from the hypothesis $E \equiv (\rho, S)$ and because dom ρ = dom ρ^{++}. The rest of the assumptions hold trivially.

  In this case $td = 0$, so by applying i.h. we obtain:

$$\begin{aligned} &c_1 \\ \rightarrow^*_{(k_0,\mathbf{p}):S}\ &([\texttt{POPCONT}],\ h'|_k,\ k,\ k,\ v_1 : drop\ 0\ ((k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2 \end{aligned}$$

  which trivially implies $c_1 \rightarrow^*_{S'} c_2$.

  Consequently:

$$\begin{aligned} &\overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\ \rightarrow_{S'}\ &(is_1,\ h,\ k,\ k,\ (k_0, \mathbf{p}) : S,\ cs') \\ \rightarrow^*_{S'}\ &([\texttt{POPCONT}],\ h'|_k,\ k,\ k,\ v_1 : drop\ 0\ ((k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2 \\ \rightarrow_{S'}\ &(is_2,\ h'|_k,\ k_0,\ k,\ v_1 : S,\ cs') \equiv c_3 \end{aligned}$$

$\rightarrow^*_{S'}$ { by i.h. (see below) }

$([\texttt{POPCONT}],\ h''|_{k_0},\ k_0,\ k_0,\ v : drop\ (td+1)\ (v_1 : S),\ cs')$

$\equiv$

$\underbrace{([\texttt{POPCONT}],\ h''|_{k_0},\ k_0,\ k_0,\ v : S',\ cs')}_{c_{final}}$

In order to apply the induction hypothesis on e_2 in the last step we have to check that $E_1 \equiv (\rho_1, v_1 : S)$. Let $x \in \mathrm{dom}(E_1)$: if $x = x_1$ then $(v_1 : S)!(\rho_1(x)) = (v_1 : S)!0 = v_1 = E_1(x)$. If $x \neq x_1$ then $(v_1 : S)!(\rho_1(x)) = (v_1 : S)!(\rho(x)+1) = S!(\rho(x)) = E(x) = E_1(x)$. Additionally $drop\ (td+1)\ (v_1 : S) = drop\ td\ S = S'$. The rest of the assumptions hold trivially.

Hence (1) holds. With respect to (2), let $i \in \{1..k\}$

$$\begin{aligned}
(diff(k,h,h''))(i) &= |\{p \in h'' \mid region(h''(p)) = i\}| - |\{p \in h \mid region(h(p)) = i\}| \\
&= |\{p \in h'' \mid region(h''(p)) = i\}| - |\{p \in h' \mid region(h'(p)) = i\}| \\
&\quad + |\{p \in h' \mid region(h'(p)) = i\}| - |\{p \in h \mid region(h(p)) = i\}| \\
&= (diff(k,h,h'))(i) + (diff(k,h',h''))(i) \\
&= \delta_1(i) + \delta_2(i)
\end{aligned}$$

Therefore $diff(k,h,h'') = \delta_1 + \delta_2$. Properties (3) and (4) are proven as follows:

$$\begin{aligned}
maxFreshCells(c_{init} \rightarrow^*_{S'} c_2) &= m_1 \\
maxFreshCells(c_2 \rightarrow^*_{S'} c_{final}) &= m_2 \\
maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final}) &= \max\ \{m_1, m_2 + sizeH(c_2) - sizeH(c_{init})\} \\
&= \max\ \{m_1, m_2 + \textstyle\sum_{i=0}^{k}(diff(k,h,h'))(i)\} \\
&= \max\ \{m_1, m_2 + |\delta_1|\}
\end{aligned}$$

$$\begin{aligned}
maxFreshWords(c_{init} \rightarrow^*_{S'} c_1) &= 2 \\
maxFreshWords(c_{init} \rightarrow^*_{S'} c_2) &= \max\ \{2, 2+s_1\} = 2+s_1 \\
maxFreshWords(c_{init} \rightarrow^*_{S'} c_2 \rightarrow^*_{S'} c_3) &= \max\ \{2+s_1, 1\} = 2+s_1 \\
maxFreshWords(c_{init} \rightarrow^*_{S'} c_3 \rightarrow^*_{S'} c_{final}) &= \max\ \{2+s_1, 1+s_2\}
\end{aligned}$$

- **Case** [*Case*]: $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

  We assume that the r-th **case** alternative is executed under an environment $E_r \stackrel{\text{def}}{=} E \uplus [\overline{x_{ri} \mapsto v_i}^{n_r}]$, where the values v_i are the parameters of the data construction pointed to by $E(x)$.

$$E_r \vdash h, k, td + n_r, e_r \Downarrow h', k, v, (\delta, m, s)$$

  In addition, let us denote $\rho_r \stackrel{\text{def}}{=} \rho + [\overline{x_{ri} \mapsto n_r - i - 1}^{n_r}]$. If $(is_r, cs_r) = trE\ e_r\ \rho_r$ then it holds that $cs \supseteq cs_r$. Moreover, we get $is = \texttt{MATCH}\ (\rho(x))\ \overline{\mathbf{p}_j}^{m}$ with $cs(\mathbf{p}_r) = is_r$ for each $r \in \{1..n\}$. The SVM derivation corresponding to is results as follows:

$$
\begin{array}{ll}
 & \overbrace{(\texttt{MATCH}\ (\rho(x))\ \overline{\mathbf{p}_j}^m,\ h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})],\ k_0,\ k,\ S,\ cs'[\mathbf{p}_r \mapsto is_r])}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\text{ since } S!(\rho(x)) = E(x) = p \} \\
 & (is_r,\ h,\ k_0,\ k,\ \overline{v_i}^{n_r} : S,\ cs') \equiv c_1 \\
\rightarrow^*_{S'} & \quad \{\text{ by i.h. (see below) }\} \\
 & ([\texttt{POPCONT}],\ h|_{k_0},\ k_0,\ k_0,\ v : drop\ (td + n_r)\ (\overline{v_i}^{n_r} : S),\ cs') \\
\equiv & \\
 & \underbrace{([\texttt{POPCONT}],\ h|_{k_0},\ k_0,\ k_0,\ v : drop\ td\ S,\ cs')}_{c_{final}}
\end{array}
$$

In the same way as in the previous cases, we have to ensure that $E_r \equiv (\rho_r, \overline{v_i}^{n_r} : S)$. Let $z \in \mathrm{dom}\,\rho_r$. We make the following case distinction: on one hand, if $z = x_{ri}$ for some $i \in \{1..n_r\}$ then $\rho_r(z) = i - 1$ and hence:

$$(\overline{v_{ir}}^n : S)!(\rho_r(z)) = (\overline{v_{ir}}^n : S)!(i-1) = v_i = E_r(x_{ri}) = E_r(z)$$

On the other hand, if $z \neq x_{ri}$ for all $i \in \{1..n_r\}$ then $\rho_r(z) = \rho(x) + n_r$. Therefore:

$$(\overline{v_{ir}}^n : S)!(\rho_r(z)) = (\overline{v_{ir}}^n : S)!(\rho(x) + n_r) = S!(\rho(x)) = E(z) = E_r(z)$$

Hence we get $E_r \equiv (\rho_r, \overline{v_i}^{n_r} : S)$ and (1) holds by applying i.h. since $drop\ (td + n_r)\ S_g = drop\ td\ S$. Properties (2) and (3) follow trivially from the induction hypothesis. With respect to (4) we get:

$$
\begin{array}{ll}
maxFreshWords(c_{init} \rightarrow^*_{S'} c_1) & = n_r \\
maxFreshWords(c_1 \rightarrow^*_{S'} c_{final}) & = s \\
maxFreshWords(c_{init} \rightarrow^*_{S'} c_{final}) & = \max\ \{n_r, s + n_r\} = s + n_r
\end{array}
$$

- **Case** [*Case!*]: $e \equiv \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^n$

  The proofs of (1) and (4) are similar to those seen for the nondestructive **case**. Property (2) follows trivially from the induction hypothesis and the definition of *diff*. With respect to (3), let c_1 denote the SVM state prior to the execution of the branch e_r. Then:

$$
\begin{array}{ll}
maxFreshCells(c_{init} \rightarrow^*_{S'} c_1) & = 0 \\
maxFreshCells(c_1 \rightarrow^*_{S'} c_{final}) & = m \\
maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final}) & = \max\ \{0, m-1\}
\end{array}
$$

  which proves the desired result.

The ($\Leftarrow$) direction of the theorem can be proved by induction on the length of the $\rightarrow^*_{S'}$ derivation. Since this proof is mostly standard, we only describe it briefly: since it is known that the SVM is deterministic, we prove that, given a *Safe* program, if the SVM eventually halts then a corresponding $\Downarrow$-derivation can be built. We distinguish cases depending on the expression e being evaluated. The base cases $e \equiv c$, $e \equiv x$, $e \equiv x\ @\ r$, $e \equiv a_1 \oplus a_2$, and $e \equiv C\ \overline{a_i}^n\ @\ r$ are straightforward: if the SVM

machine halts with the corresponding result on the top of stack, this result can be used with rules [*Lit*], [*Var*], [*Copy*] and [*Cons*].

As an example, let us consider $e \equiv x$. Let $(is, cs) = trE\ e\ \rho$ be the SVM code generated, where $cs = [\,]$ and:

$$is = \texttt{BUILDENV}\ [\rho(x)] : \overbrace{\texttt{SLIDE}\ 1\ (topDepth(\rho)) : \underbrace{\texttt{DECREGION} : \underbrace{[\texttt{POPCONT}]}_{is_3}}_{is_2}}^{is_1}$$

Given any cs', the only possible derivation is the following one:

$$\begin{array}{ll}
 & \overbrace{(\texttt{BUILDENV}\ [\rho(x)] : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\
\rightarrow_{S'} & \quad \{\text{since } S!(\rho(x)) = E(x) = v\} \\
 & (\texttt{SLIDE}\ 1\ (topDepth(\rho)) : is_2,\ h,\ k_0,\ k,\ v : S,\ cs') \\
\rightarrow_{S'} & \quad \{\text{since } td = topDepth(\rho)\} \\
 & (\texttt{DECREGION} : is_3,\ h,\ k_0,\ k,\ v : drop\ td\ S,\ cs') \\
\rightarrow_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\ h|_{k_0},\ k_0,\ k_0,\ v : drop\ td\ S,\ cs')}_{c_{final}}
\end{array}$$

Trivially, by rule [*Var*], $E[x \mapsto v] \vdash h, k, td, x \Downarrow h, k, v, ([\,]_k, 0, 1)$. Property (2) holds because $diff(k, h, h) = [i \mapsto 0 \mid 0 \leq i \leq k] = \delta$. Property (3) and (4) hold because $maxFreshCells(c_{init} \rightarrow^*_{S'} c_{final}) = 0$ and $maxFreshWords\ (c_{init} \rightarrow^*_{S'} c_{final}) = 1$.

With respect to the remaining cases, the $\rightarrow^*_{S'}$ execution sequence is made up of a set of preliminary actions (building the variable environment in the case of function application, pushing a continuation in the case of **let** expression or executing a `MATCH`/`MATCH!` at the beginning of a **case**/**case!**) followed by the evaluation of the corresponding subexpressions (i.e, either the function being called or the main/auxiliary expressions of a **let** or the **case**/**case!** branch). The induction hypothesis can be applied to these ($\rightarrow^*$)-subderivations in order to get the required assumptions of the corresponding $\Downarrow$ rule.

As an example, consider $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2$. Assume $(is_1, cs_1) = trE\ e_1\ \rho^{++}$ and $(is_2, cs_2) = trE\ e_2\ \rho_1$ where $\rho_1 = \rho + [x_1 \mapsto 1]$. Then $is = \texttt{PUSHCONT}\ \mathbf{p} : is_1$, in which $cs_1(\mathbf{p}) = is_2$. Given $cs' \supseteq cs$, the trace corresponding to the execution of e must be as follows:

$$\begin{array}{ll}
 & \overbrace{(\texttt{PUSHCONT}\ \mathbf{p} : is_1,\ h,\ k_0,\ k,\ S,\ cs')}^{c_{init}} \\
\rightarrow_{S'} & \\
 & (is_1,\ h,\ k,\ k,\ (k_0, \mathbf{p}) : S,\ cs') \equiv c_1 \\
\rightarrow^*_{S'} & \{\text{by Lemma 2.26}\} \\
 & ([\texttt{POPCONT}],\ h',\ k,\ k,\ v_1 : (k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2]) \equiv c_2 \\
\rightarrow_{S'} & \\
 & (is_2,\ h',\ k_0,\ k,\ v_1 : S,\ cs') \equiv c_3 \\
\rightarrow^*_{S'} & \\
 & \underbrace{([\texttt{POPCONT}],\ h'',\ k_0,\ k_0,\ v : S',\ cs')}_{c_{final}}
\end{array}$$

Notice that c_1 contains a continuation $(k_0, \mathbf{p})$ on top of the stack. The only machine instruction which removes such continuation from the stack is `POPCONT`. So, in order to reach c_{final} an intermediate configuration $c_2 \equiv ([\texttt{POPCONT}],\ h',\ k',\ k',\ v_1 : (k_0, \mathbf{p}) : S),\ cs'[\mathbf{p} \mapsto is_2])$ must be reached so that the execution can proceed. By Lemma 2.26, $k' = k$.

In fact, it holds that $c_1 \rightarrow^*_{(k_0,\mathbf{p}):S} c_2$, so we can apply i.h. to obtain $E \vdash h, k, 0, e_1 \Downarrow h', k, v_1, (\delta_1, m_1, s_1)$, as $(k_0, \mathbf{p}) : S = drop\ 0\ ((k_0, \mathbf{p}) : S)$.

Notice also that $E \uplus [x_1 \mapsto v_1] \equiv \rho + [x_1 \mapsto 1]$ and that $drop\ (td + 1)\ (v_1 : S) = drop\ td\ S = S'$, so we can also apply i.h. to $c_3 \rightarrow^*_{S'} c_{final}$ to obtain $E \uplus [x_1 \mapsto v_1] \vdash h', k, td + 1, e_2 \Downarrow h'', k, v, (\delta_2, m_2, s_2)$. The reasoning about resources consumption is the same as the opposite implication. □

2.9 A global overview of the *Safe* certifying compiler

The *Safe* compiler is mostly implemented in Haskell, and it runs in several phases. These are shown in Figure 2.32. First, the input is scanned and parsed in order to obtain an abstract syntax tree (AST), which is represented as a Haskell term. For the implementation of this phase we have used standard tools [42, 77]. Then, the compiler's front-end performs the following phases:

- **Renamer/Contextual constraints checker [31]:** The analyses implemented in the subsequent phases assume that all bound variables have different names. Thus all variables are renamed in this phase. Additionally, the contextual constraints of the language are checked, e.g. that every occurrence of a variable is in scope, functions and data constructors are called with the right number of arguments, etc.

- **Hindley-Milner type and region inference [85]:** This phase decorates every expression, data constructor and function definition with its corresponding type, after inferring it. The AST is also decorated with information about the regions in which each data structure lives. In particular, it determines whether a given data structure is local or not to the current function call. Chapter 4 deals extensively with this phase.

- ***Core-Safe* transformation [31]:** The original *Safe* program is translated into a desugared and semantically equivalent *Core-Safe* version. The previously inferred type decorations are preserved.

- **Sharing analysis [98]:** Given two variables belonging to the same function definition, it computes whether the respective data structures pointed to by them may share memory locations at runtime. The program is decorated with the sharing information that is needed by the following phase.

- **Destruction analysis [84, 81]:** It infers a typing for the source program w.r.t. the type system given in Chapter 3, guaranteeing that dangling pointers are not generated as a consequence of destructive pattern matching. The design of this phase is dealt with in Chapter 5.

- **Memory Consumption Analysis [83]:** It computes an upper-bound to the memory requirements of a *Safe* program. Chapter 7 explains the way in which these bounds are obtained.

The compiler can produce several kinds of outputs. So far we have implemented the following backends:

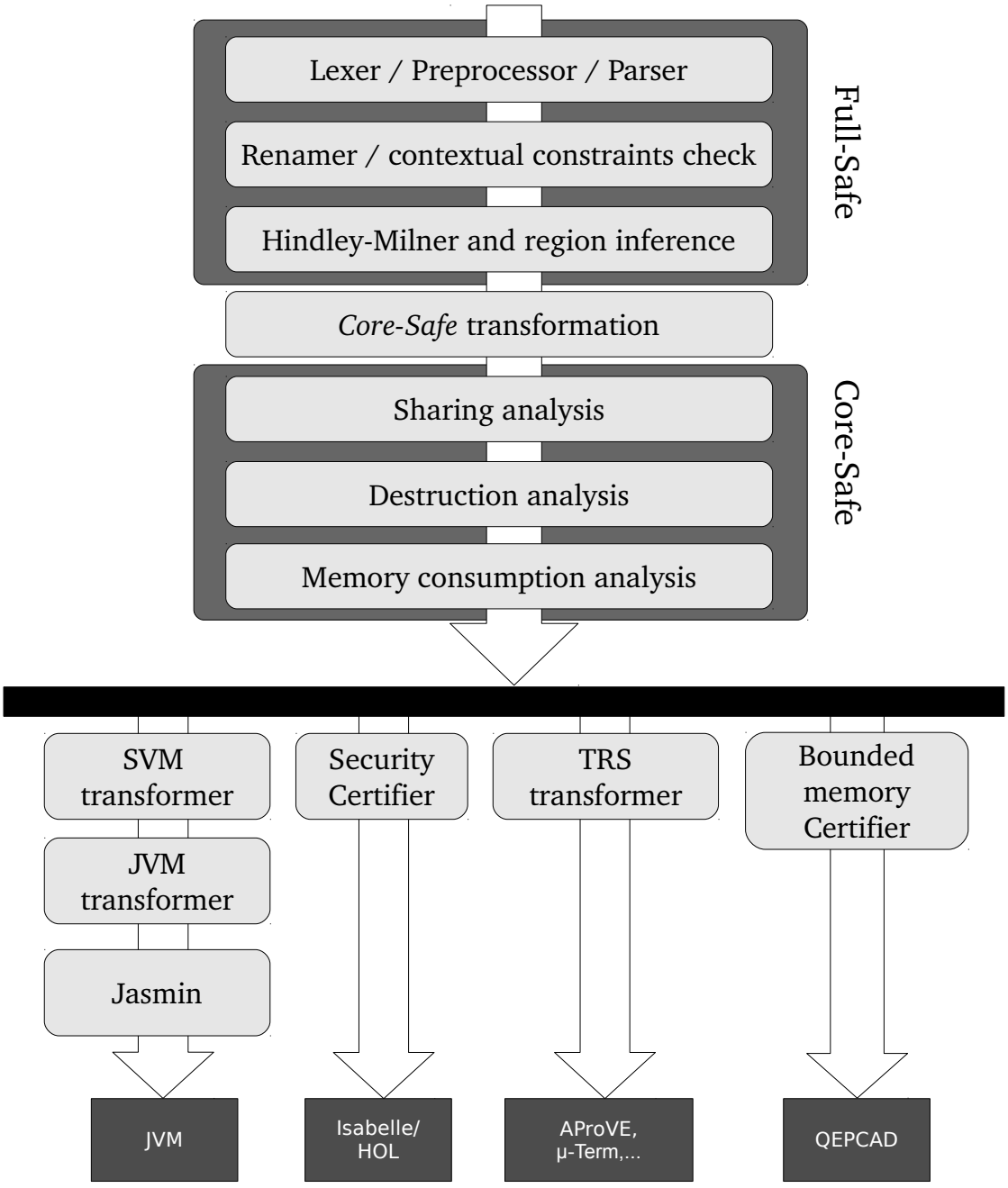


Figure 2.32: Front-end and back-ends of *Safe*'s compiler.

- **Java bytecode generation [39, 38]:** The SVM translation process explained in Section 2.6 is used as an intermediate step for obtaining Java bytecode, which can be run under the Java Virtual Machine (JVM).

- **Certification of security properties [36]:** By using the information given by the destruction analysis phase, this back-end generates an Isabelle/HOL script proving the absence of dangling pointers in the compiled program. In a PCC scenario, the code consumer would receive this formal proof attached to the Java bytecode, in order to verify these safety properties. Chapter 6 describes the process of generating the certificates.

- **Certification of memory bounds [40]:** Given the results of the memory consumption analysis, this back-end certifies that the computed memory bounds are correct for the program being analysed. This phase generates first-order formulas in Tarski's theory of closed fields. These formulas are proven valid with the QEPCAD system [23].

- **TRS transformer [73]:** It translates the *Core-Safe* definitions into Term Rewriting Systems (TRS), so that their termination can be checked by existing tools, like AProVE [47] or MU-TERM [72].

- **Other back-ends:** For instance, translation of *Safe* programs into C.

This work concerns mainly the design and correctness properties of some of these phases, but hardly any implementation detail. These can be found in [80], or in [79] (the latter in Spanish). The *Safe* compiler is available for download at `http://dalila.sip.ucm.es/safe/`. There also exists a web-based interface at `http://dalila.sip.ucm.es/~safe/`.

2.10 Conclusions and related work

In this chapter we have applied a semi-systematic method for refining operational semantics and abstract machines in order to find the way from an abstract view of the language to an efficient implementation. Other contributions include a semantics enriched with memory costs, and the proof of correctness of these costs when translating *Safe* to imperative code. This resource-aware semantics is the basis for proving correct the memory consumption static analysis presented in Chapter 7, and for certifying these bounds, done in [40].

The use of regions in functional languages to avoid garbage collection is not new, as it was explained in Section 1.1.1. Tofte and Talpin [116] introduced in MLKit (a variant of ML) the use of nested regions by means of a **letregion** construct. A lot of work has been done on this system [2, 19, 113]. Their main contribution is a *region inference* algorithm adding region annotations at the intermediate language level. A small difference with these approaches is that, in *Safe*, region allocation and deallocation are synchronized with function calls instead of being introduced by a special language construct. This simplifies the process of inferring regions, as we shall explain in Chapter 4. However, this comes at the cost of granularity for determining the region scopes: MLKit allows several lexically-scoped regions in the same function. A more relevant difference is that *Safe* has an additional mechanism allowing the programmer to selectively destroy data structures inside a region.

A difficulty with the original Tofte and Talpin's system is the fact that regions have nested lifetimes. There exist a few programs (such as the *inssort* function shown in Example 2.11) that may result in memory leaks due to this restriction. In [45] this problem is alleviated by defining a variant of λ-calculus with type-safe primitives for creating, accessing and destroying regions, which are not restricted to

have nested lifetimes. Programs are written in a C-like language called *Cyclone* having explicit memory management primitives, then they are translated into this variant of λ-calculus, and then type checked. So, the price of this flexibility is explicit region control. In our language *Safe*, regions also suffer from the nested lifetimes constraint, since both region allocation and deallocation are bound to function calls, which are necessarily nested. However, the destructive pattern matching facility compensates for this, since it is possible to dispose of a data structure without deallocating the whole region in which it resides. Allocation and destruction of distinct data structures are not necessarily nested, and the type system presented here protects the programmer against missuses of this feature. Again, the price of this flexibility is explicit deallocation of cells. Allocation is implicit in constructions and the target region of the allocation is inferred by the compiler. It is arguable whether it is better to explicitly manage regions or cells. In [98] a more detailed comparison with all these works can be found.

The destructive pattern matching features of the language have been inspired by Hofmann and Jost's `match` construct [58], whose operational behaviour is similar to that of *Safe*'s **case**!. The main difference is that they lack a compile time analysis guaranteeing the safe use of this dangerous feature, since it is beyond the scope of their work.

There have been other successful derivations of abstract machines starting from high level descriptions of the semantics. For instance, in [52] and [1] a number of such derivations are done. Well known abstract machines for the λ-calculus such as SECD, Krivine's, CLS and CAM are derived and proved correct. These papers propose general schemes for achieving this kind of derivations. The differences with the present work are the following:

- They concentrate on the pure λ-calculus and they consider neither sharing nor heaps. Algebraic types, **case** and **let** expressions are not considered either.
- In the second paper, the starting point is a denotational meaning of the source language, while here we start from an operational semantics.
- In order to refine their machines they use predefined correct transformations such as closure conversion, transformation into continuation passing style, defunctionalization and inlining.
- They ignore the compilation issues from the source language to machine instructions, and also resource consumption.

In [65] a broad survey of both abstract and virtual machines for the λ-calculus and for practical functional languages is done. The author presents in detail some well-known and other less known abstract machines. When the machines execute compiled code, also the translation schemes are provided. The aim of the book is to serve as a text for a graduate course and no attempt is done to provide proofs of correctness either of the machines or of the compilation schemes.

For the first abstract machine M2 we have found inspiration in Sestoft's derivation of abstract machines for a lazy λ-calculus [104]. For the rest of the derivation, Peña (supervisor of this thesis) has reported some previous experience in [41], but, in that occasion, the destination machine was known in advance. The present work represents a 'real' derivation in the sense that the destination machine has been invented from scratch.

Compared to other eager machines such as Landin's SECD machine [70], it is an added value of our abstract machine that the standard translation yields constant stack space for tail recursion, as we have shown in Example 2.17. For instance, in the G-machine the compiler needs to explicitly identify tail recursion and to do a special translation in this case, i.e. it is considered as an optimization of the code

generation phase. The same happens in other compiled virtual machines such as π-RED$^+$ [46]. Additionally, our SVM machine does not need a garbage collector and all memory allocation/deallocation actions have been implemented in constant time.

For the semantics enriched with a resource vector, we have found inspiration in [9]. Some other resource-enriched semantics have been proposed. See for example [58, 56] for a big-step semantics of a simple functional language with some information about the number of cells needed by an expression, while [64] extends the same idea to a higher-order language. In [51] Hammond and Michaelson define a model for computing heap and stack costs for a subset of the Hume language [50].

The translation shown in this chapter has been formally verified by de Dios and Peña with the help of the Isabelle/HOL proof assistant. They address the translation from *Core-Safe* to SVM in [39] , and the translation from SVM instructions to Java bytecode in [38].

Chapter 3

Type system

3.1 Introduction

The destruction features provided by *Safe* allow a memory efficient implementation of several data structures and algorithms. However, there exists a risk at runtime: the execution may access some parts of the memory that have been disposed of previously. A program is said to be *pointer-safe* if this eventuality never arises at runtime for any input given to the program.

In this chapter we devise a typed-based approach to prove that a given program is pointer-safe. This problem is, in general, undecidable: the type system developed in this chapter can prove a program pointer-safe, but it cannot prove the absence of this property. As a consequence, the type system may reject some pointer-safe programs. It is desirable to get as few rejected pointer-safe programs as possible. In Section 3.5 we show some examples of successfully typed programs, which should give an idea about the power of the type system.

The main result of this chapter is the proof of the following fact: *the existence of a typing derivation for a program guarantees its pointer-safety*. However, in this chapter we do not explain *how* to get this typing derivation. This question is postponed to the next two chapters. The contents of this chapter are based on the work described in [84]. However, the type system of this chapter contains significant improvements with respect to that of [84]. In particular, it accepts more pointer-safe programs, and admits an inference algorithm (Chapter 5) which is more efficient than that of [84].

3.2 Type system concepts

Since the *Safe* language is largely based on Haskell, it inherits many of the ideas in Haskell's type system, which is based, in turn, on Hindley-Milner type system. In particular, it supports polymorphic algebraic data types and inherits the list data type (denoted by $[\alpha]$, being α the type of the elements of the list) with its standard constructors $[\,]$ and $(:)$, as well as the tuple data type, denoted by $(\alpha_1, \ldots, \alpha_n)$.

When extending the Hindley-Milner type system to *Core-Safe*, one must assign a type to region variables. For this purpose we define a new category of types: *region type variables*. The type of a region variable is a region type variable (abbreviated as RTV in the following). These RTVs, which will be denoted by $\rho, \rho_1, \ldots$, act as ordinary polymorphic variables in Haskell. However, only region variables are allowed to have a RTV as its type.

Algebraic data types are annotated with RTVs, which always coincide with the types of the region

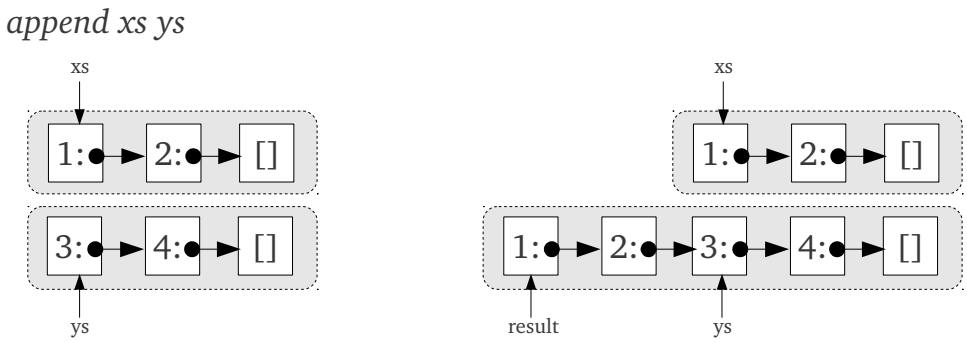


Figure 3.1: Representation of the input lists in the heap before (left-hand side) and after (right-hand side) a call to the *append* function. The input lists can be located in different regions, but the result have to be constructed in the same region as the second list, since the latter is reused in the base case.

variables used in the creation of the corresponding DS. For example, if r has type ρ, the expression $[\,] \,@\, r$ has type $[\alpha]@\rho$. We have these kind of annotations with the aim of stating a connection between data structures and region variables, and connections among different data structures. For example, if two variables have $[\alpha]@\rho$ and $[\beta]@\rho$ as their respective types, their corresponding lists must live in the same region at runtime. Moreover, if there exists another region variable r' with type ρ, every DS being constructed with this variable at runtime will also live in the same region as these two lists. It is important to have a way to determine these connections at compile time, because it allows us to know, in particular, whether a given data structure resides in the temporary region *self*, which is the only region variable of type ρ_{self}.

Example 3.1. Recall from Example 2.5 the following function for appending two lists:

$$\begin{array}{lll} \mathit{append}\ xs\ ys\ @\ r = & \textbf{case}\ xs\ \textbf{of} & \\ & [\,] \quad\ \ \to & ys \\ & (x : xx) \to & \textbf{let}\ x_1 = \mathit{append}\ xx\ ys\ @\ r \\ & & \textbf{in}\ (x : x_1)@r \end{array}$$

Let us assume that the r variable has type ρ. Without considering region annotations in data types, we would obtain this type for *append*.

$$\mathit{append} :: \forall \alpha\ \rho\ .\ [\alpha] \to [\alpha] \to \rho \to [\alpha]$$

Now let us determine the RTVs that would be attached to each $[\alpha]$. The branch guarded by $(x : xx)$ returns the result of the construction $(x : x_1)@r$, which has type $[\alpha]@\rho$. Notice, however, that the branch guarded by $[\,]$ returns the list passed as second parameter. As we will see below, the typing rules demand that the result of all branches in a **case** expression must have the same type. Therefore, the type of the second parameter must be $[\alpha]@\rho$, which results in the following type for *append*:

$$\mathit{append} :: \forall \alpha\ \rho\ \rho_1.\ [\alpha]@\rho_1 \to [\alpha]@\rho \to \rho \to [\alpha]@\rho$$

The ρ_1 annotation in the first parameter means that the region of the corresponding list may be different from that of the second parameter and from the result (Figure 3.1)

Assume the variation of the *append* function given in Example 2.7 that, in the base case, returns a

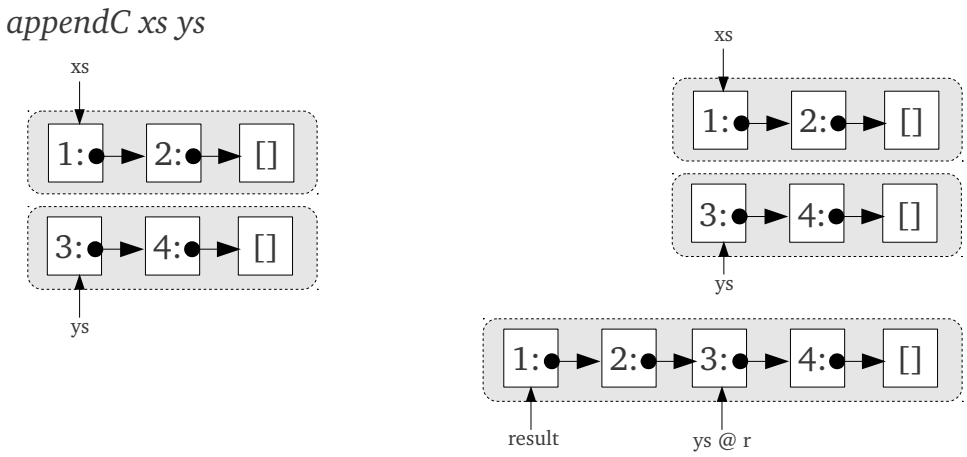


Figure 3.2: Representation of the input lists in the heap before (left-hand side) and after (right-hand side) a call to the *appendC* function. Now the result can be constructed from scratch in a different region.

copy of the second parameter instead of the original:

$$\begin{array}{lll} \mathit{appendC}\ xs\ ys\ @\ r = & \textbf{case}\ xs\ \textbf{of} & \\ & [\,] & \rightarrow\ ys\,@\,r \\ & (x : xx) & \rightarrow\ \textbf{let}\ x_1 = \mathit{appendC}\ xx\ ys\ @\ r \\ & & \quad\ \textbf{in}\ (x : x_1)@r \end{array}$$

In this case, the result of the expression $ys\ @\ r$ has type $[\alpha]@\rho$, whose RTV need not be the same as that of *ys*. Thus we obtain the following type for *appendC*:

$$\mathit{appendC} :: \forall \alpha\ \rho\ \rho_1\ \rho_2.\ [\alpha]@\rho_1 \rightarrow [\alpha]@\rho_2 \rightarrow \rho \rightarrow [\alpha]@\rho$$

The function *appendC* does not require that the list passed as second parameter lives in the same region as the result (Figure 3.2) □

Algebraic data types can be associated with more than one RTV. For example, the following definition:

$$\textbf{data}\ \mathit{TBL}\ \alpha\ \beta\ @\ \rho_1\ \rho_2\ \rho_3 = \mathit{TBL}\ [(\alpha, \beta)@\rho_1]@\rho_2\ @\ \rho_3$$

defines a concrete implementation of the table abstract data type, represented as a list of $(key, value)$ pairs. Data structures of this type may spread up to three different regions: one for the *TBL* constructor, another for the spine of the list containing the pairs, and another one for the pairs themselves (Figure 3.3).

In general, our algebraic types take the form $T\ \overline{s_i}\ @\ \overline{\rho_j}$, where T is a *type constructor* (*TBL* in our example), and $\overline{s_i}$, $\overline{\rho_j}$ are the type arguments and RTVs that are applied to this constructor. The last RTV of the list $\overline{\rho_j}$ is the type of the region where the data structures of this type are built. This RTV is called the *outermost region*. In our example, the outermost region of the *TBL* data type is ρ_3. When the compiler decorates recursive **data** declarations with RTVs (see Section 4.3), the recursive occurrences of the type being defined must have the same RTVs as in the type definition. As a consequence, data types are not polymorphic recursive, unlike function definitions, in which region-polymorphic recursion is allowed.

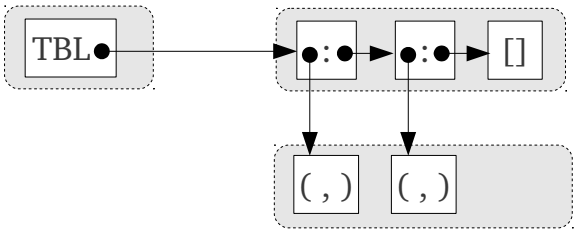


Figure 3.3: The *TBL* datatype can be distributed over three different regions. The outermost region contains the constructor *TBL*.

The following example shows the RTV-annotated definition of a binary search tree data type:

$$\textbf{data}\ \mathit{BSTree}\ \alpha\ @\ \rho = \mathit{Empty}\ @\ \rho \mid \mathit{Node}\ (\mathit{BSTree}\ \alpha\ @\ \rho)\ \alpha\ (\mathit{BSTree}\ \alpha\ @\ \rho)\ @\ \rho$$

In the following we will write $T\ @\ \overline{\rho_j}$ instead of $T\ \overline{s_i}\ @\ \overline{\rho_j}$ when the $\overline{s_i}$ are not relevant. We shall even use the notation $T\ @\ \rho$ to highlight only the outermost region.

The type of a variable also reflects whether it is destroyed by means of a **case**! expression. This is represented by a (!) symbol in the corresponding types. For example, in the following expression, the variable *xs* gets $[\alpha]!@\rho$ as its type:

$$\begin{array}{ll}\textbf{case!}\ xs\ \textbf{of} & [\,] \to \ldots \\ & (x : xx) \to \ldots\end{array}$$

Types with a (!) mark are called *condemned*. Variables with condemned types can occur in the discriminant of a destructive **case**!, as well as in the destructive arguments of a function call. In our example, the variable *xx* also gets a condemned type $[\alpha]!@\rho$. On the contrary, if a variable is not affected (even indirectly[1]) by a **case**!, we say that it has a *safe* type. We use the term *underlying type* to refer to the Hindley-Milner component of a type with its RTVs, without regard to whether this type is condemned or safe.

In a standard Hindley-Milner type system, a variable always has the same type along its scope. This does not always hold in our type system: the underlying types are preserved along the scope, but some of them may have a condemned mark (!) only in a few of its occurrences. Let us consider the following example:

$$\textbf{let}\ x_1 = (\textbf{case}\ xs\ \textbf{of}\ [\,] \to \ldots)\ \textbf{in}\ (\textbf{case!}\ xs\ \textbf{of}\ [\,] \to \ldots)$$

The variable *xs* has type $[\alpha]@\rho$ in the auxiliary expression of the **let**, and type $[\alpha]!@\rho$ in the main expression. Each subexpression will be typed under a different typing environment: let us denote by Γ_1 the typing environment for the auxiliary expression, and by Γ_2 the typing environment for the main expression. The variable *xs* is bound to a safe type in Γ_1 and to a condemned type in Γ_2. For typing the whole **let** expression, both environments are merged into a single environment Γ which assigns a condemned type to *xs*, meaning that at some point of the execution of the **let**, the DS pointed to by this variable will be destroyed. However, not every combination is possible. If we swapped the subexpressions of our **let** as follows:

[1]Even if a DS is not directly involved in a **case**!, it may point to a cell being destroyed. We shall explain this in more detail in Section 3.3.

$$
\begin{array}{rrcll}
\textbf{Type} \ni & \tau & \to & t & \{\text{external}\} \\
 & & & \mid r & \{\text{in-danger}\} \\
 & & & \mid \sigma & \{\text{polymorphic function}\} \\
 & & & \mid \rho & \{\text{region type, being } \rho \in \textbf{RegType}\} \\
 & t & \to & s & \{\text{safe}\} \\
 & & & \mid d & \{\text{condemned}\} \\
\textbf{SafeType} \ni & s & \to & T\ \overline{s}\ @\ \overline{\rho}^m & \{\text{algebraic, with } m \geq 1\} \\
 & & & \mid \alpha & \{\text{variable, being } \alpha \in \textbf{TypeVar}\} \\
 & & & \mid B & \{\text{basic, being } B \in \{Int, Bool\}\} \\
\textbf{CdmType} \ni & d & \to & T\ \overline{s}\ !@\ \overline{\rho}^m & \{\text{condemned algebraic, with } m \geq 1\} \\
\textbf{DgrType} \ni & r & \to & T\ \overline{s}\ \#@\ \overline{\rho}^m & \{\text{in-danger algebraic, with } m \geq 1\} \\
\textbf{FunType} \ni & tf & \to & \overline{t_i} \to \overline{\rho} \to s & \{\text{function}\} \\
\textbf{SafeFunType} \ni & sf & \to & \overline{s_i} \to \overline{\rho} \to s & \{\text{safe function}\} \\
 & \sigma & \to & \forall \overline{\alpha}\ \overline{\rho}.tf & \{\text{type scheme}\}
\end{array}
$$

Figure 3.4: Syntax of type expressions

$$\textbf{let}\ x_1 = (\textbf{case}!\ xs\ \textbf{of}\ [\,] \to \ldots)\ \textbf{in}\ \ (\textbf{case}\ xs\ \textbf{of}\ [\,] \to \ldots)$$

the outermost **let** would be ill-typed. In Section 3.4 we will describe precisely which combinations of safe and condemned marks are allowed.

Example 3.2. Revisiting our *append* example, let us consider its destructive variant (Example 2.10):

$$
\begin{array}{lll}
appendD\ xs\ ys\ @\ r = & \textbf{case}!\ xs\ \textbf{of} & \\
 & [\,] \to & ys \\
 & (x : xx) \to & \textbf{let}\ x_1 = appendD\ xx\ ys\ @\ r \\
 & & \textbf{in}\ (x : x_1)@r
\end{array}
$$

The function *appendD* gets the following type:

$$appendD :: \forall \alpha\ \rho\ \rho_1.\ [\alpha]!@\rho_1 \to [\alpha]@\rho \to \rho \to [\alpha]@\rho$$

□

3.3 Type expressions and environments

The syntax of type expressions is shown in Figure 3.4. Since our language is first-order, we distinguish between functional types σ and non-functional types, which can be RTVs ρ or algebraic data types t, r.

Algebraic types may be *safe* types s, *in-danger* types r or *condemned* types d. In-danger types arise as an intermediate step during typing and are useful to control the side-effects of the destructions. But notice that the types of functions only include either safe or condemned types; that is why there is

a separate syntactical category t of external types which rules out the possibility of having in-danger types in function signatures.

The intended semantics of each category of algebraic types is the following:

- **Safe types (s):** The DSs of this type cannot be directly or indirectly destroyed, so they are kept intact during the evaluation of the expression being typed. Basic types *Int* and *Bool* and type variables are also included in this category.
  - *Capabilities:* DSs of safe types can be read, copied and used to build other DSs.
  - *Prohibitions:* DSs of safe types can neither be used in the discriminant of a **case**! expression, nor point to a cell that is being destroyed by a **case**!. Variables with safe types cannot appear in destructive positions of function applications.
  - *Guarantees:* All the cells pointed to by a DS of a safe type are guaranteed to remain intact during the evaluation of the expression being typed.
- **Condemned types (d):** The DSs of this type can receive the action of a destructive **case**!, but this destructive action is limited to the recursive spine of the DS.
  - *Capabilities:* A DS of this type can be read, copied and used to build other DSs before being destroyed. It can be part in the discriminant of a **case**!. In this case, its recursive spine can also be destroyed through its recursive pattern variables.
  - *Prohibitions:* The non-recursive part of a DS of a condemned type cannot point to a cell being destroyed.
  - *Guarantees:* The non-recursive descendants of a condemned DS are ensured to remain intact during the evaluation of the expression being typed, but the recursive spine of a condemned DS may get totally or partially corrupted during this evaluation.
- **In-danger types (r):** The DSs of this type can point to a recursive descendant of a condemned DS, so they contain potentially dangling pointers.
  - *Capabilities:* A DS of this type can be read, copied and used to build other DSs before being corrupted. It can also occur in the discriminant of a **case**!. Its recursive spine can be destroyed, as well as its non-recursive descendants.
  - *Prohibitions:* None.
  - *Guarantees:* None. Both recursive and non-recursive descendants of an in-danger DS can be destroyed.

From these informal descriptions, we can easily see that safe types are the "strongest" ones, in the sense that they are the most restrictive (e.g. they cannot point to anything being destroyed), but they offer more guarantees with respect to the integrity of their corresponding DSs. On the other hand, in-danger types are the most permissive, but they cannot ensure nothing about which part of the corresponding DS is kept intact. This motivates the following order on these categories: $s \leq d \leq r$. Before presenting this order relation in a formal way, let us introduce some notation: we use the names **SafeType**, **CdmType** and **DgrType** to denote the set of all safe, condemned and in-danger types, respectively. We also define the sets **ExpType** and **UnsafeType** as follows:

$$\begin{aligned} \mathbf{ExpType} &\stackrel{\text{def}}{=} \mathbf{SafeType} \cup \mathbf{CdmType} \cup \mathbf{DgrType} \\ \mathbf{UnsafeType} &\stackrel{\text{def}}{=} \mathbf{CdmType} \cup \mathbf{DgrType} \end{aligned}$$

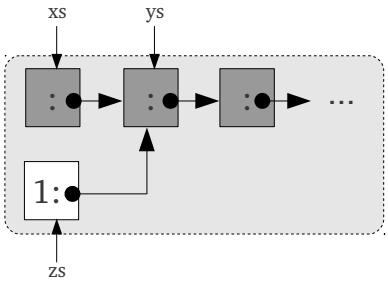
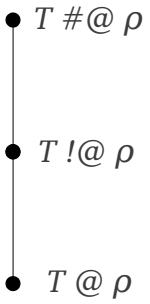


Figure 3.5: Situation described in Example 3.3: the recursive spine of *xs* is condemned (gray cells). Since *zs* points to it, it gets an in-danger type.

The predicate $utype?(\tau, \tau')$, where $\tau, \tau' \in \mathbf{ExpType}$, tells whether two types have the same underlying type (i.e. without regard to ! or # marks). By abuse of notation, we extend this predicate to function schemes as follows: $utype?(\sigma, \sigma') \Leftrightarrow \sigma = \sigma'$.

Given these notational conventions, the $\leq$ partial order on types is defined as follows:

$$\tau_1 \leq \tau_2 \overset{def}{\Longleftrightarrow} utype?(\tau_1, \tau_2) \wedge (\tau_1 = \tau_2 \vee \tau_1 \in \mathbf{SafeType} \vee \tau_2 \in \mathbf{DgrType}) \tag{3.1}$$



Example 3.3. In order to illustrate the meanings of these types, let us consider the following *Core-Safe* definition:

$$\begin{aligned} f\ xs\ @\ r \quad = \quad & \mathbf{let}\ ys = (\mathbf{case}\ xs\ \mathbf{of}\ (x : xx) \rightarrow xx)\ \mathbf{in} \\ & \mathbf{let}\ zs = (1 : ys)@r\ \mathbf{in}\ \mathbf{case!}\ xs\ \mathbf{of}\ \ldots \end{aligned}$$

The variable *ys* refers to the tail of the list pointed to by *xs*, which is used in the definition of *zs* to build another list (Figure 3.5). After that, the first cons cell of the list pointed to by *xs* is disposed by means of a **case**!. In order to type the **case**! *xs* ... expression under an environment Γ, the variable *xs* must appear in this environment with a condemned type $[Int]!@\rho$. Moreover, since both *ys* and *zs* are sharing a recursive descendant of *xs*, they must occur in Γ with an unsafe type. Notice that the variable *xs* may appear in the typing environment of the non-destructive **case** with a safe type $[Int]@\rho$, since it is not being destroyed in that expression. □

A functional type *tf* has the form $\overline{t_i}^n \rightarrow \overline{\rho_j}^m \rightarrow s$, where $\overline{t_i}^n$ and $\overline{\rho_j}^m$ are the types of the data and region parameters, and *s* is the type of the result. Notice that function parameters are restricted by syntax to be safe types, condemned types or RTVs. The motivation for not allowing in-danger types in function signatures is the following: when a parameter is condemned we know clearly that

only the recursive substructure of the corresponding DS may be condemned. When the function is applied to an argument we know that the recursive substructure of such argument may be partially or totally destroyed. However, when a parameter is in-danger, the only thing we know is that some part (recursive or not) of the whole DS may be dangling, but we do not know which part. This is a very imprecise information to put in the type of a function. In spite of this restriction, in-danger types are still useful for controlling and propagating the side effects of explicit destruction through a function's body.

We distinguish a special category of functional types, **SafeFunType**, which contains the types of those functions whose parameters have safe types. We use the notation sf for denoting an element of this set. Constructor types are a particular instance of these types. They have a single region argument ρ which coincides with the outermost region variable of the resulting algebraic type $T\ \overline{s_i}\ @\ \overline{\rho_j}^m$. Constructors are given types indicating that the recursive substructure and the structure itself must live in the same region, as we have pointed up before. For example, in the case of lists and trees:

$$
\begin{array}{lcl}
[\,] & :: & \forall \alpha\ \rho\ .\ \rho \to [\alpha]@\rho \\
(:) & :: & \forall \alpha\ \rho\ .\ \alpha \to [\alpha]@\rho \to \rho \to [\alpha]@\rho \\
\mathit{Empty} & :: & \forall \alpha\ \rho\ .\ \rho \to \mathit{BSTree}\ \alpha\ @\ \rho \\
\mathit{Node} & :: & \forall \alpha\ \rho\ .\ \mathit{BSTree}\ \alpha\ @\ \rho \to \alpha \to \mathit{BSTree}\ \alpha\ @\ \rho \to \rho \to \mathit{BSTree}\ \alpha\ @\ \rho
\end{array}
$$

By abuse of notation, we shall also consider type schemes σ to belong to **FunType** or **SafeFunType**, depending on the functional type after the $\forall$ quantifiers.

Expressions are typed in the context of *type environments*, which contain the type of variables, and function/constructor names in scope. A type environment Γ contains region type bindings $[r : \rho]$, variable type bindings $[x : t]$, $[x : r]$ and polymorphic scheme assignments to functions $[f : \sigma]$ and constructor symbols $[C : \sigma]$. Formally, a type environment is a triple of mappings in this set:

$$\Gamma : (\mathbf{Var} \to \mathbf{ExpType}) \times (\mathbf{RegVar} \to \mathbf{RegType}) \times (\mathbf{Fun} \cup \mathbf{Cons} \to \mathbf{FunType})$$

However, by abuse of notation, we consider Γ as a single function. It will be clear which specific component is meant, when applying this function to a variable x, region variable r, function symbol f, or constructor symbol C. We also define several operators (summarized in Table 3.1) for composing two type environments:

- The usual $+$ operator demands disjoint domains. Its result is the union of the bindings of the environments to which it is applied.

$$\Gamma_1 + \Gamma_2 \text{ defined} \Leftrightarrow \operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma_2 = \emptyset$$

$$\operatorname{dom}(\Gamma_1 + \Gamma_2) = \operatorname{dom}\Gamma_1 \cup \operatorname{dom}\Gamma_2$$

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \in \operatorname{dom}\Gamma_1 \\ \Gamma_2(x) & \text{otherwise} \end{cases}$$

- The $\oplus$ operator allows a variable to occur in several environments, but only if it has the same *safe* type in all of them.

$$\Gamma_1 \oplus \Gamma_2 \text{ defined} \Leftrightarrow \forall x \in \operatorname{dom}\Gamma_1 \cap \operatorname{dom}\Gamma_2.\ \Gamma_1(x) = \Gamma_2(x) \in \mathbf{SafeType}$$

$\bullet$	$\Gamma_1 \bullet \Gamma_2$ defined if:	Result of $\Gamma_1 \bullet \Gamma_2$:
$+$	Γ_1 and Γ_2 have disjoint domains	Union of Γ_1 and Γ_2 bindings
$\oplus$	Common variables in Γ_1 and Γ_2 have the same safe type	
$\sqcup$	Common variables in Γ_1 and Γ_2 have the same underlying type	Union of Γ_1 and Γ_2 bindings (highest types take precedence)

Table 3.1: Informal descriptions of the operators on type environments.

$$\text{dom}\,(\Gamma_1 \oplus \Gamma_2) = \text{dom}\,\Gamma_1 \cup \text{dom}\,\Gamma_2$$

$$(\Gamma_1 \oplus \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \in \text{dom}\,\Gamma_1 \\ \Gamma_2(x) & \text{otherwise} \end{cases}$$

- The $\sqcup$ operator, when applied to the environments Γ_1 and Γ_2, allows a variable to occur with a different type in each environment, provided that the underlying type in each of the occurrences is the same. The result of $\Gamma_1 \sqcup \Gamma_2$ contains all the bindings from Γ_1 and Γ_2. If the same variable occurs in both environments with a different type, the higher (weaker) version takes priority over the lower one.

$$\Gamma_1 \sqcup \Gamma_2 \text{ defined} \;\Leftrightarrow\; \forall x \in \text{dom}\,\Gamma_1 \cap \text{dom}\,\Gamma_2 . \mathit{utype}?(\Gamma_1(x), \Gamma_2(x))$$

  It is easy to see that, if τ_1 and τ_2 have the same underlying type, then either $\tau_1 \leq \tau_2$ or $\tau_2 \leq \tau_1$ w.r.t. the order relation given in 3.1. In these cases we use the notation $\tau_1 \sqcup \tau_2$ to denote the maximal type between both.

$$\text{dom}\,(\Gamma_1 \sqcup \Gamma_2) = \text{dom}\,\Gamma_1 \cup \text{dom}\,\Gamma_2$$

$$(\Gamma_1 \sqcup \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \notin \text{dom}\,\Gamma_2 \\ \Gamma_2(x) & \text{if } x \notin \text{dom}\,\Gamma_1 \\ \Gamma_1(x) \sqcup \Gamma_2(x) & \text{if } x \in \text{dom}\,\Gamma_1 \cap \text{dom}\,\Gamma_2 \end{cases}$$

It is easy to see that these three operators are commutative and associative.

Example 3.4. Given the following type environments:

$$\begin{aligned} \Gamma_1 &= [x : [\alpha]@\rho_2] \\ \Gamma_2 &= [x : [\alpha]!@\rho_2, y : \mathit{Int}] \\ \Gamma_3 &= [x : [\alpha]!@\rho_2, z : \mathit{Bool}] \\ \Gamma_4 &= [x : [\alpha]\#@\rho_2, y : \mathit{Bool}] \end{aligned}$$

- None of the $\Gamma_1 + \Gamma_2$, $\Gamma_2 + \Gamma_3$, $\Gamma_1 + \Gamma_3$, $\Gamma_1 + \Gamma_4$, $\Gamma_2 + \Gamma_4$,$\Gamma_3 + \Gamma_4$ is defined.
- $\Gamma_2 \oplus \Gamma_3$ is undefined, since x has an unsafe type in both environments.
- $\Gamma_2 \sqcup \Gamma_4$ is undefined, since the underlying types of $\Gamma_2(y)$ and $\Gamma_4(y)$ are not equal. However, $\Gamma_2 \sqcup \Gamma_3$, $\Gamma_3 \sqcup \Gamma_4$ as well as $\Gamma_1 \sqcup \Gamma_3 \sqcup \Gamma_4$ are defined as follows:

$$\begin{aligned} \Gamma_2 \sqcup \Gamma_3 &= [x : [\alpha]!@\rho_2, y : \mathit{Int}, z : \mathit{Bool}] \\ \Gamma_3 \sqcup \Gamma_4 &= [x : [\alpha]\#@\rho_2, y : \mathit{Bool}, z : \mathit{Bool}] \\ \Gamma_1 \sqcup \Gamma_3 \sqcup \Gamma_4 &= [x : [\alpha]\#@\rho_2, y : \mathit{Bool}, z : \mathit{Bool}] \end{aligned}$$

$$\frac{\Gamma' \vdash e : s \qquad \Gamma \supseteq \Gamma'}{\Gamma \vdash e : s}\ [\text{EXT}] \qquad \frac{\Gamma + [x : \tau_1] \vdash e : s \qquad \tau_1 \leq \tau_2}{\Gamma + [x : \tau_2] \vdash e : s}\ [\text{WEAK}]$$

$$\frac{}{\emptyset \vdash c : B}\ [\text{LIT}] \qquad \frac{}{[x : s] \vdash x : s}\ [\text{VAR}]$$

$$\frac{}{[x : T\,@\,\rho', r : \rho] \vdash x\,@\,r : T\,@\,\rho}\ [\text{COPY}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash e_1 : s_1 \qquad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s \qquad \mathit{utype?}(\tau_1, s_1) \\ \forall x \in \operatorname{dom}\Gamma_1.\Gamma_1(x) \in \mathbf{UnsafeType} \Rightarrow x \notin \mathit{fv}(e_2)\end{array}}{\Gamma_1 \sqcup \Gamma_2 \vdash \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 : s}\ [\text{LET}]$$

$$\frac{\begin{array}{ll}\overline{t_i}^n \to \overline{\rho_j}^m \to s \trianglelefteq \sigma \qquad \Gamma = \Gamma_R + [f : \sigma] + \bigoplus_{i=1}^n [a_i : t_i] + [\overline{r_j : \rho_j}^m] \\ \Gamma_R = \bigsqcup_{a_i \in \mathbf{CdmType}} \mathit{SHR}(a_i, f\ \overline{a_i}^n\ @\ \overline{r_j}^m) & \bigwedge_{t_i \in \mathbf{CdmType}} \mathit{isTree}(a_i)\end{array}}{\Gamma \vdash f\ \overline{a_i}^n\ @\ \overline{r_j}^m : s}\ [\text{APP}]$$

$$\frac{\overline{s_i}^n \to \rho \to s \trianglelefteq \sigma \qquad \Gamma = [C : \sigma] + \bigoplus_{i=1}^n [a_i : s_i] + [r : \rho]}{\Gamma \vdash C\ \overline{a_i}^n\ @\ r : s}\ [\text{CONS}]$$

$$\frac{\begin{array}{ll}\forall i \in \{1..n\}.\Gamma(C_i) = \sigma_i \qquad \mathit{utype?}(\Gamma(x), T\,@\,\rho) & \forall i \in \{1..n\}.\overline{s_{ij}}^{n_i} \to \rho \to T\ @\ \rho \trianglelefteq \sigma_i \\ \forall i \in \{1..n\}.\Gamma + [\overline{x_{ij} : \tau_{ij}}^{n_i}] \vdash e_i : s & \forall i \in \{1..n\}.\forall j \in \{1..n_i\}.\mathit{utype?}(s_{ij}, \tau_{ij})\end{array}}{\Gamma \vdash \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ [\text{CASE}]$$

$$\frac{\begin{array}{ll}\forall i \in \{1..n\}.\ \Gamma(C_i) = \sigma_i & \forall i \in \{1..n\}.\ \overline{s_{ij}}^{n_i} \to \rho \to T\,@\,\rho \trianglelefteq \sigma_i \\ \Gamma_R = \mathit{SHR}(x, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) & \forall i \in \{1..n\}.\ \forall j \in \{1..n_i\}.\mathit{inh}(t_{ij}, s_{ij}, T\ !@\,\rho) \\ \forall i \in \{1..n\}.\forall z \in \operatorname{dom}\Gamma_R \cup \{x\}.z \notin \mathit{fv}(e_i) & \forall i \in \{1..n\}.\ \Gamma + [\overline{x_{ij} : t_{ij}}^{n_i}] \vdash e_i : s\end{array}}{\Gamma_R \sqcup (\Gamma \backslash x) + [x : T\ !@\,\rho] \vdash \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ [\text{CASE!}]$$

Figure 3.6: Typing rules for expressions.

□

The $\leq$ order relation defined on types can be extended to typing environments as follows:

$$\Gamma_1 \leq \Gamma_2 \overset{def}{\iff} \operatorname{dom}\Gamma_1 \subseteq \operatorname{dom}\Gamma_2 \wedge \forall x \in (\operatorname{dom}\Gamma_1) \cap \mathbf{Var}.\Gamma_1(x) \leq \Gamma_2(x)$$

This relation is motivated by the fact that, if an expression is typeable under an environment Γ_0, it should be typeable under every environment Γ such that $\Gamma \geq \Gamma_0$ (i.e. Γ is less restrictive than Γ_0). In the following section we describe in detail the typing relation.

3.4 Typing rules

Figure 3.6 shows the rules for typing expressions. These rules define the derivation of judgements of the form $\Gamma \vdash e : s$, meaning that the expression e has a safe type s under an environment Γ. Some of these rules use the operators on type environments explained previously, and they implicitly assume that the application of each of these operators results in a well-defined environment. If it does not, the corresponding rule cannot be applied.

There are rules for typing literals [LIT] and variables [VAR]. Notice that these expressions are typed

only under the minimal typing environment w.r.t. the $\leq$ relation on environments. If we want to type these expressions under a weaker environment, we can do it by means of the [EXT] and [WEAK] rules. The first one is useful for extending the environment with fresh bindings, whereas the second one allows us to downgrade the type category of a given variable to a weaker one. That is, safe variables can be demoted to unsafe, whereas condemned variables can be demoted to in-danger. Both [EXT] and [WEAK] rules are useful in the context of **case** and **case!** expressions, since their corresponding typing rules (which we shall explain later) demands all the **case**(!) branches to be typeable under the same environment (excluding pattern variables, which may differ between branches).

Example 3.5. Given the following expression:

$$\begin{array}{l}\textbf{case } b \textbf{ of} \\ \quad \begin{array}{rcl} \mathit{True} & \rightarrow & e_1 \\ \mathit{False} & \rightarrow & e_2 \end{array}\end{array}$$

Assume we have derived the following typing judgements for each branch

$$[x : [\alpha]!@\rho] \vdash e_1 \qquad [x : [\alpha]@\rho, z : \mathit{Int}] \vdash e_2$$

We can apply the [EXT] rule to the first judgement, and the [WEAK] rule to the second one, so both expressions are typable under the same environment:

$$\frac{[x : [\alpha]!@\rho] \vdash e_1}{[x : [\alpha]!@\rho, z : \mathit{Int}] \vdash e_1}\ [\text{EXT}] \qquad \frac{[x : [\alpha]@\rho, z : \mathit{Int}] \vdash e_2}{[x : [\alpha]!@\rho, z : \mathit{Int}] \vdash e_2}\ [\text{WEAK}]$$

□

The [COPY] rule allows any variable of an algebraic type to be copied. The outermost region variable ρ' in the type of the DS being copied is replaced by the type ρ of the destination region.

Rule [LET] allows the bound variable to get a safe, condemned or in-danger type in the main expression e_2, provided it has the same underlying type as the expression e_1. The side condition in this rule demands that any variable with an unsafe type in e_1 cannot be mentioned (i.e. occur free) in e_2, since it may contain dangling references. Lastly, if the DS pointed by a variable is not guaranteed to remain intact during the evaluation of one of the sub-expressions of the **let**, it cannot be guaranteed to have a safe type when typing the whole **let** expression. That is why we take the least upper bound $\Gamma_1 \sqcup \Gamma_2$ of the environments typing each sub-expression.

Rule [APP] deals with function application. The operator $\trianglelefteq$ denotes the instantiation of a type scheme, in which the bound variables of that scheme can be replaced by safe types. The use of the $\oplus$ operator avoids a variable to be used in two or more different positions unless they are all safe parameters. Otherwise undesired side-effects could happen, such as trying to destroy the same cell twice in the function being called. The notation $\Gamma_R = \mathit{SHR}(x, e)$ is defined as follows:

$$\begin{array}{rcl} \Gamma_R = \mathit{SHR}(x, e) & \overset{def}{\iff} & \operatorname{dom} \Gamma_R = \mathit{sharerec}(x, e) \backslash \{x\} \\ & & \wedge \forall y \in \operatorname{dom} \Gamma_R.\ \Gamma_R(y) \in \textbf{DgrType} \end{array}$$

The $\mathit{sharerec}(x, e)$ notation denotes the set of variables in the scope of e that may point to a recursive descendant of the DS pointed to by x at runtime. This set is computed by an abstract interpretation-based sharing analysis, which is described in [98]. This analysis approximates the sharing relations

between the different variables occurring in the body of a function, but it shall not be detailed here. In Section 3.6.3 we state its correctness properties, since they are essential for proving the correctness of the type system.

The *SHR* notation specifies that Γ_R contain every variable in the scope of e that may point to a recursive descendant of x, and these variables are mapped by Γ_R to an in-danger type. This notation can be extended to join the sharing information of several variables as follows:

$$\Gamma_R = \bigsqcup_{i \in I} SHR(x_i, e) \overset{def}{\iff} \forall i \in I.\ \exists \Gamma_i.\ (\Gamma_i = SHR(x_i, e) \wedge \Gamma_i \leq \Gamma_R)$$

In our particular [APP] rule, the Γ_R environment contains all those variables that may point to an argument being passed as a condemned parameter of f. The disjoint union of Γ_R with the rest of the arguments ensure that there are no arguments with an in-danger type in the context of the function application, in the same way as there are no in-danger parameters in the function definition. It also propagates the information about possibly damaged pointers downwards in the derivation tree.

Example 3.6. Given a function g with the following type:

$$g :: [Int]!@\rho_1 \to [Int]!@\rho_1 \to [Int]@\rho_1 \to [Int]@\rho_1 \to Int$$

The function call $g\ xs\ ys\ xs\ zs$ is ill-typed, because the following environment is undefined:

$$[xs : [Int]!@\rho_1] \oplus [ys : [Int]!@\rho_1] \oplus [xs : [Int]@\rho_1] \oplus [zs : [Int]@\rho_1]$$

If we swap the second and third arguments, the result $g\ xs\ xs\ ys\ zs$ is also ill-typed for the same reasons:

$$[xs : [Int]!@\rho_1] \oplus [xs : [Int]!@\rho_1] \oplus [ys : [Int]@\rho_1] \oplus [zs : [Int]@\rho_1]$$

However, we can provide the same argument twice in two safe positions. For example, in $g\ xs\ ys\ zs\ zs$ we would obtain the following environment:

$$[xs : [Int]!@\rho_1] \oplus [ys : [Int]!@\rho_1] \oplus [zs : [Int]@\rho_1] \oplus [zs : [Int]@\rho_1]$$

which is well-defined and equivalent to $[xs : [Int]!@\rho_1, ys : [Int]!@\rho_1, zs : [Int]@\rho_1]$. Notice that we cannot provide arguments with an in-danger type. For example, in the following code fragment:

$$\begin{array}{l} \textbf{case}\ xs\ \textbf{of} \\ \quad (x : xx) \to f\ xs\ ys\ xx\ zs \end{array}$$

the function call cannot be typed, since xx is a recursive descendant of a variable in a condemned position (xs). Therefore, it occurs in the Γ_R environment with an in-danger type, and the following is undefined:

$$\underbrace{[xx : r]}_{\Gamma_R} + ([xs : [Int]!@\rho_1] \oplus [ys : [Int]!@\rho_1] \oplus [xx : [Int]@\rho_1] \oplus [zs : [Int]@\rho_1])$$

□

The *isTree* predicate is given by an auxiliary analysis, which is run jointly with the sharing analysis of [98]. An intuition on this predicate can be given by considering the recursive spine of a DS as a directed

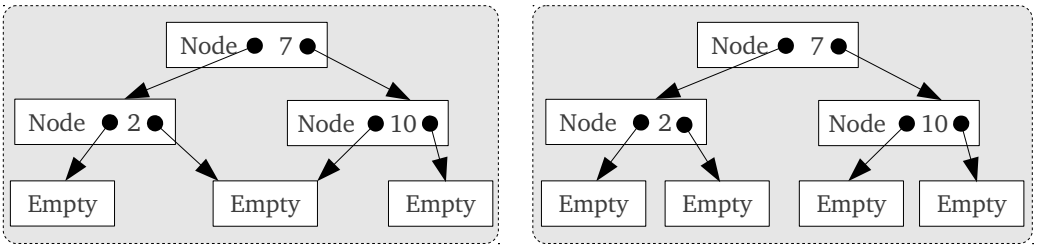


Figure 3.7: Internal sharing in binary search trees (BSTs). The graph of the BST at the left-hand side is not a tree, but a directed acyclic graph. If such a BST is pointed to by a variable, the predicate *isTree* will never hold at compile-time for that variable. On the contrary, the graph of the BST at the right-hand side is a tree.

graph, in which the vertices are the cells of its spine and the edges represent the connections between cells: there is an edge from w_1 to w_2 iff w_1 contains a pointer to w_2. The predicate $isTree(x)$ holds if the graph associated to the DS pointed to by x is guaranteed to be a tree at runtime or, in other words, if we cannot reach a recursive descendant of that DS via two different paths (see Figure 3.7). Otherwise, the graph associated to that DS is not a tree, but a directed acyclic graph (DAG). The $\bigwedge_{a_i \in \mathbf{CdmType}} isTree(a_i)$ condition in the [APP] rule demands that there should not be internal sharing between the cells pointed to by the variables occurring in condemned positions of the function. Otherwise the function being called might attempt to destroy a child of a condemned DS twice.

Example 3.7. Consider the following function for destroying a binary search tree:

$$
\begin{aligned}
&\mathit{destroyBST} :: \mathit{BSTree}\ \alpha! \to \mathit{Int} \\
&\mathit{destroyBST}\ \mathit{Empty} = 0 \\
&\mathit{destroyBST}\ (\mathit{Node}\ l\ x\ r) = \quad \mathbf{let}\ x_1 = \mathit{destroyBST}\ l\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{let}\ x_2 = \mathit{destroyBST}\ r\ \mathbf{in}\ 0
\end{aligned}
$$

If we execute the following expression:

$$\mathbf{let}\ x_1 = \mathit{Empty}\ \mathbf{in}\ \mathit{destroyBST}\ (\mathit{Node}\ x_1\ 0\ x_2)$$

there would be an attempt to destroy the DS pointed to by x_1 twice. □

For the purposes of the type system, a constructor application $C\ \overline{a_i}^n\ @\ r$ can be considered as a function application in which all the parameters have a safe type. Hence the [CONS] rule is a particular instance of [APP].

Rule [CASE] allows its discriminant variable to be safe, in-danger, or condemned, since it only reads that variable. We demand the Hindley-Milner types of the discriminant and the pattern variables to be instances of the types of the corresponding constructors, but pattern variables may also occur with a (!) mark or a (#) mark in the environments. This provides more flexibility in comparison with the type system of [84], in which the marks of the pattern variables can be restricting, depending on the mark of the discriminant.

In rule [CASE!] the discriminant is destroyed and consequently the text should not try to reference it in the alternatives. The same happens to those variables sharing a recursive substructure of x, as they may be corrupted. These substructures can only be accessed through the pattern variables occurring in recursive positions of the constructor. The corrupted variables are added to the environment Γ_R with

$$\begin{array}{rcl} inh(s,s,d) & \Leftrightarrow & \neg utype?(s,d) \\ inh(d,s,d) & \Leftrightarrow & utype?(s,d) \end{array}$$

Figure 3.8: Definitions of inheritance compatibility.

$$\frac{\begin{array}{c}\rho_{self} \notin rtv(s) \cup (\bigcup_{i=1}^{n} rtv(t_i)) \cup \{\overline{\rho_j}^{\,l}\} \\ \Gamma + [\overline{x_i : t_i}^{\,n}] + [\overline{r_j : \rho_j}^{\,l}] + [self : \rho_{self}] + [f : \forall \overline{\rho}.\overline{t_i}^{\,n} \to \overline{\rho_j}^{\,l} \to s] \vdash e : s\end{array}}{\{\Gamma\}\ \ f\ \overline{x_i}^{\,n}\ @\ \overline{r_j}^{\,l} = e\ \ \{\Gamma + [f : \forall \overline{\alpha}\,\overline{\rho}\,.\,\overline{t_i}^{\,n} \to \overline{\rho_j}^{\,l} \to s]\}}\ \text{[FUN]}$$

Figure 3.9: Typing rule for function definitions.

in-danger types. Relation *inh*, defined in Figure 3.8, determines the types inherited by pattern variables: recursive ones are condemned while non-recursive ones must be safe. This reflects the intended meaning of safe and condemned types: since the discriminant is condemned, only the recursive part of the DSs pointed to the pattern variables in recursive positions may be destroyed, whereas those variables in non-recursive positions must be left intact.

In Figure 3.9 we show the [FUN] rule for typing function definitions. The informal meaning of a judgement $\{\Gamma\}\ def\ \{\Gamma'\}$ is that, assuming that the type of every function called in the body of *def* is contained in Γ, the definition *def* is well-typed and Γ' is the extension of Γ with the type of the function being defined. We use $\forall\overline{\alpha}\,\overline{\rho}$ to denote the standard generalization of a functional type with respect all its type variables, and $\forall\overline{\rho}$ to denote the generalization of the region variables of a functional type. The function's body is typed under an environment which contains the types of the data and region parameters. The *self* region is assigned a special type ρ_{self} which must be distinct from the region types of the r_j, the region types occurring in the result, and the region types of the input parameters. With this constraint we ensure that the deallocation of the *self* region when the function finishes does not cause dangling pointers, since the result of the function does not reside there.

Notice that the $\forall\overline{\rho}$ notation allows us to have region-polymorphic recursion: inside the body e, different applications may use different regions. This kind of polymorphism cannot be applied to ordinary type variables, since the type reconstruction problem would be undecidable, as shown by Henglein [53]. However, if we restrict ourselves to recursion over RTVs, this problem becomes decidable. In Chapter 4 we explain the advantages of allowing this kind of recursion.

Example 3.8. The following function is ill-typed:

$$f\ xs = \textbf{case}\ xs\ \textbf{of}\ (x : xx) \to xx\,@\,self$$

The type of its body is $[\alpha]@\rho_{self}$, so the condition $\rho_{self} \notin rtv(s)$ does not hold. Operationally, this function would build a copy of the tail of the input list into the working region and return a reference to the result. However, the result is lost after the function finishes because of the *self* deallocation. Hence, this function always returns a dangling pointer. □

The following property of the type system will be useful in its correctness proof.

Lemma 3.9 (Substitution lemma)**.** *If $\Gamma \vdash e : s$ and θ is a substitution that maps type variables α to safe types and region types ρ to region types, then $\theta(\Gamma) \vdash e : \theta(s)$.*

Proof. Since θ maps type variables only to safe types, and RTVs to RTVs, the application of θ to a type τ does not change its category. Formally:

$$\tau \in X \Rightarrow \theta(\tau) \in X \qquad \text{where } X \in \{\textbf{SafeType}, \textbf{DgrType}, \textbf{CdmType}, \textbf{RegType}, \textbf{FunType}\} \tag{3.2}$$

This implies, in particular:

1. $utype?(\tau, \tau') \Leftrightarrow utype?(\theta(\tau), \theta(\tau'))$ for all $\tau, \tau' \in \mathbf{ExpType}$, which can be proven by induction on the structure of τ.

2. $inh(\tau_1, \tau_2, \tau_3) \Leftrightarrow inh(\theta(\tau_1), \theta(\tau_2), \theta(\tau_3))$ for any $\tau_1, \tau_2, \tau_3 \in \mathbf{ExpType}$, which follows from (1) and (3.2).

3. $\Gamma_1 \diamond \Gamma_2$ is defined iff $\theta(\Gamma_1) \diamond \theta(\Gamma_2)$ is defined, for $\diamond \in \{+, \oplus, \sqcup\}$ and any Γ_1,Γ_2. This follows from (1) and (3.2).

4. $\theta(\Gamma_1 \diamond \Gamma_2) = \theta(\Gamma_1) \diamond \theta(\Gamma_2)$ for $\diamond \in \{+, \oplus, \sqcup\}$ and any Γ_1, Γ_2. This follows from (3) and (3.2).

Therefore, the premises in the derivation $\theta(\Gamma) \vdash e : \theta(s)$ follow from their counterparts in $\Gamma \vdash e : s$, and the lemma can be easily proved by induction on the latter derivation. □

3.5 Case studies

In this section we show some examples of successfully typed programs. The first two examples show the typing derivations of two simple functions.

Example 3.10. Given the *appendD* function of Example 2.12, it gets the following type: $[\alpha]!@\rho_1 \to [\alpha]@\rho \to \rho \to [\alpha]@\rho$. We denote by $\sigma_{appendD}$ its region-polymorphic variant $\forall \rho_1\ \rho.[\alpha]!@\rho_1 \to [\alpha]@\rho \to \rho \to [\alpha]@\rho$. Let us start typing the recursive application *appendD xx ys @ r*. The sharing analysis gives the following information:

$$sharerec(xx, appendD\ xx\ ys\ @\ r) = \{xs\} \qquad isTree(xx)$$

Hence the following relation holds,

$$[xs : [\alpha]\#@\rho_1] = SHR(xx, appendD\ xx\ ys\ @\ r)$$

and the following typing environment is well-defined:

$$[xs : [\alpha]\#@\rho_1] + [f : \sigma_{appendD}] + ([xx : [\alpha]!@\rho_1] \oplus [ys : [\alpha]@\rho]) + [r : \rho]$$

Hence, we can use the [APP] rule for deriving the following judgement:

$$\underbrace{[xs : [\alpha]\#@\rho_1,\ f : \sigma_{appendD},\ xx : [\alpha]!@\rho_1,\ ys : [\alpha]@\rho,\ r : \rho]}_{\Gamma_1} \vdash appendD\ xx\ ys\ @\ r : [\alpha]@\rho \tag{3.3}$$

We can apply the [CONS] rule in a similar way to obtain:

$$\underbrace{[x : \alpha, r : \rho]}_{\Gamma_2} + [x_1 : [\alpha]@\rho] \vdash (x : x_1)@r : [\alpha]@\rho \tag{3.4}$$

The result of $\Gamma_1 \sqcup \Gamma_2$ is well-defined and equal to $[xs : [\alpha]\#@\rho_1,\ f : \sigma_{appendD},\ xx : [\alpha]!@\rho_1,\ ys : [\alpha]@\rho,\ r : \rho,\ x : \alpha]$. Besides this, neither xs nor xx occur free in $(x : x_1)@r$, so we can apply the [LET] rule with

(3.3) and (3.4) in order to type the recursive case of the function:

$$[f : \sigma_{appendD},\ xs : [\alpha]\#@\rho_1,\ ys : [\alpha]@\rho,\ r : \rho] + [x : \alpha,\ xx : [\alpha]!@\rho_1] \vdash \mathbf{let}\ x_1 = \ldots \mathbf{in} \ldots : [\alpha]@\rho \tag{3.5}$$

Now we move on to the base case. By applying the [VAR] rule we get,

$$[ys : [\alpha]@\rho] \vdash ys : [\alpha]@\rho$$

and, by the [EXT] rule,

$$[f : \sigma_{appendD},\ xs : [\alpha]\#@\rho_1,\ ys : [\alpha]@\rho,\ r : \rho] \vdash ys : [\alpha]@\rho \tag{3.6}$$

Now we use (3.5) and (3.6) in order to apply the [CASE!] rule. It is worth noting that in this case the environment Γ_R is empty, and that the rest of side conditions in that rule hold. Hence we get:

$$[f : \sigma_{appendD},\ xs : [\alpha]!@\rho_1,\ ys : [\alpha]@\rho,\ r : \rho] \vdash \mathbf{case}\ xs\ \mathbf{of} \ldots : [\alpha]@\rho$$

Lastly, we apply the [EXT] rule in order to add the binding of the working region variable *self*:

$$[f : \sigma_{appendD},\ xs : [\alpha]!@\rho_1,\ ys : [\alpha]@\rho,\ r : \rho, self : \rho_{self}] \vdash \mathbf{case}\ xs\ \mathbf{of} \ldots : [\alpha]@\rho$$

Therefore, the body of the expression is well-typed. This judgement also serves as a base for applying the [FUN] rule, so the function definition is also well-typed. □

Example 3.11. We consider the following variant of the *treesort* function (Example 2.6 on page 24):

$$\begin{aligned} &treesortD\ xs\ @\ r = \\ &\qquad \mathbf{let}\ x_1 = mkTreeD\ xs\ @\ self \\ &\qquad \mathbf{in}\ inorder\ x_1\ @\ r \end{aligned}$$

where the function *mkTreeD* builds a binary search tree from a list, while destroying the latter.

We assume that *mkTreeD* and *inorder* have already been typed, obtaining their respective type schemes σ_{mkTree} and $\sigma'_{inorder}$:

$$\begin{aligned} \sigma_{mkTreeD} &= \forall \rho_1\ \rho_2.[Int]!@\rho_1 \to \rho_2 \to BSTree\ Int\ @\ \rho_2 \\ \sigma'_{inorder} &= \forall \alpha\ .\sigma_{inorder} \\ \sigma_{inorder} &= \forall \rho_1\ \rho_2.BSTree\ \alpha\ @\ \rho_1 \to \rho_2 \to [\alpha]@\rho_2 \end{aligned}$$

Firstly we apply the *mkTreeD* application with the [APP] rule:

$$\underbrace{[mkTreeD : \sigma_{mkTreeD},\ xs : [Int]!@\rho_1,\ self : \rho_{self}]}_{\Gamma_1} \vdash mkTreeD\ xs\ @\ self : BSTree\ Int\ @\ \rho_{self}$$

We proceed similarly with the *inorder* application:

$$\underbrace{[inorder : \sigma_{inorder}, r : \rho]}_{\Gamma_2} + [x_1 : BSTree\ Int\ @\ \rho_{self}] \vdash inorder\ x_1\ @\ r : [Int]@\rho$$

With these two judgements, and because the only variable with an unsafe type in Γ_1 does not occur free

in the main expression of the **let**, we can apply the [LET] rule so as to get:

$$\underbrace{[\mathit{mkTreeD} : \sigma_{\mathit{mkTreeD}},\ \mathit{inorder} : \sigma_{\mathit{inorder}}]}_{\Gamma} + [xs : [\mathit{Int}]!@\rho_1,\ r : \rho,\ \mathit{self} : \rho_{\mathit{self}}] \vdash \mathbf{let}\ x_1 = \ldots\ \mathbf{in}\ \ldots : [\mathit{Int}]@\rho$$

and, by the [EXT] rule:

$$\Gamma + [xs : [\mathit{Int}]!@\rho_1,\ r : \rho,\ \mathit{self} : \rho_{\mathit{self}}] + [\mathit{treesortD} : \sigma] \vdash \mathbf{let}\ x_1 = \ldots\ \mathbf{in}\ \ldots : [\mathit{Int}]@\rho$$

assuming $\sigma = \forall\rho_1\ \rho.[\mathit{Int}]!@\rho_1 \to \rho \to [\mathit{Int}]@\rho$. We can use this judgement for applying [FUN] and obtain:

$$\{\Gamma\}\mathit{treesortD}\ xs\ @\ r = \ldots \{\Gamma + [\mathit{treesort} : \sigma]\}$$

Therefore, the function is well-typed.

We can build a non-destructive version of *treesortD* by creating a copy of the original list and passing this copy to the *treesortD* function:

$$\mathit{treesort}\ xs\ @\ r = \mathbf{let}\ xs' = xs\ @\ \mathit{self}\ \mathbf{in}\ \mathit{treesortD}\ xs'\ @\ r$$

We get the following type scheme: $\forall\rho_1\ \rho.[\mathit{Int}]@\rho_1 \to \rho \to [\mathit{Int}]@\rho$. To type the function's body, we apply the [LET] rule in a similar way as in the destructive version. □

The following examples are more involved. By the sake of clarity we just show the *Full-Safe* code (annotated with regions in the following example) and give an intuition of the typing derivation, without representing it in detail. All these examples have been accepted by the *Safe* compiler.

Example 3.12. In this example we provide an implementation of the insertion sort algorithm, which does not require additional heap memory. We start with the *insertD* function, which destructively inserts an element into an sorted list, such that the result is also a sorted list.

$$\begin{aligned}
&\mathit{insertD}\ x\ [\,]!\ @\ r = (x : [\,]\ @\ r)\ @\ r\\
&\mathit{insertD}\ x\ (y : ys)!\ @\ r\\
&\qquad\qquad \mid x \leq y = (x : (y : ys)@r)@r\\
&\qquad\qquad \mid x > y = (y : \mathit{insertD}\ x\ ys\ @\ r)@r
\end{aligned}$$

This function destroys the list passed as second parameter as it is traversed. In the recursive case, the cell being destroyed $(y : ys)$ can be reused by the result of the function. However, in the base cases, the function needs two available cells to build the result, whereas only one is destroyed in the pattern matching. Thus, in global terms, this function needs a free cell available in the heap in order to insert the element into the list.

The type scheme for this function is $\forall\rho\,.\,\mathit{Int} \to [\mathit{Int}]!@\rho \to \rho \to [\mathit{Int}]@\rho$. The variable *ys* gets a condemned type $[\mathit{Int}]!@\rho$ after the pattern matching, since it is the recursive child of the list being destroyed. However, this variable is used to build the $(y : ys)$ list in the expression guarded by $x \leq y$. Since the [CONS] rule demands a safe type for every variable passed to the $(:)$ constructor we have to apply the [WEAK] rule so the $(y : ys)@r$ expression can be typed under an environment in which *ys* is condemned. With respect to the recursive call, its second parameter must be of a condemned type, and so is *ys*, so the [APP] rule can be applied.

We continue with the *inssortD* function, which repeatedly inserts each element of the input list into the result,which is initially an empty list.

```
data Leftist α = LTEmpty | LTNode Int (Leftist α) α (Leftist α)

cons (LTEmpty)! a r = LTNode 1 r a (LTEmpty)
cons (LTNode n ll la lr)! a r =
        | n ≥ m = LTNode (m + 1) (LTNode n ll la lr) a r
        | n < m = LTNode (n + 1) r a (LTNode n ll la lr)
          where (LTNode m _ _ _) = r

join LTEmpty! r! = r
join (LTNode n ll a lr)! LTEmpty! = LTNode n ll a lr
join (LTNode n ll a lr)! (LTNode m rl b rr)!
        | a < b = cons ll a (join lr (LTNode m rl b rr)
        | a ≥ b = cons rl b (join rr (LTNode n ll b lr)

emptyPQueue = LTEmpty

addPQueue q a = join q (LTNode 1 LTEmpty a LTEmpty)

minPQueue (LTNode _ _ a _) = a

delMinPQueue (LTNode _ l _ r) = join l r
```

Figure 3.10: Priority queue data type implemented as a leftist tree.

```
inssortD [ ]! @ r = [ ] @ r
inssortD (x : xs)! @ r = insertD x (inssortD xs @ r) @ r
```

In each recursive call, the first cell of the input list is destroyed. This cell can be reused for building the empty list in the base case, or for calling the *insertD* function, which needs a free cell available as it has been explained above.

The type of this function is $\forall\rho\,.\,[Int]!@\rho \to \rho \to [Int]@\rho$. The *xs* variable is condemned, since it is the recursive child of the cell being destroyed. Hence it can be used in the recursive call to *inssortD*. The result of this call is used destructively in the *insertD* function. □

Example 3.13. The destructive features of *Safe* can be used to implement data structures whose updating needs no additional heap space. For instance, let us consider an implementation of priority queues based on leftist trees. The source code, taken from [94], is shown in Figure 3.10. The main difference with respect to the version given in [94] is the use of destructive pattern matching in some functions for the sake of constant heap space consumption.

First we define the data type of leftist trees (*Leftist* α). A leftist tree is a binary tree in which the shortest path from any node to any of its leafs is always the rightmost one. The first argument of the *LTNode* constructor contains the length of this path. Leftist trees are maintained in such a way that this length is kept as low as possible. This results in a well-balanced tree. The function *cons* acts as a wrapper of the *LTNode* constructor in order to preserve this invariant. This function is destructive on its first parameter, which is reconstructed in the function's body. This is done in order to optimize the memory usage, since otherwise we would need two additional memory cells for executing the base case. By applying destructive pattern matching we can reuse the cell corresponding to the input parameter, so we would only need one memory cell to build the tree. We could have also done without destructive

pattern matching with the following definition:

$$
\begin{array}{lcl}
\mathit{cons}\ l\ a\ r & = & \textbf{case}\ l\ \textbf{of} \\
 & & \quad \mathit{LTEmpty} \to \mathit{LTNode}\ 1\ r\ a\ l \\
 & & \quad \mathit{LTNode}\ n\ \_\ \_\ \_ \to \textbf{case}\ n \geq m\ \textbf{of} \\
 & & \qquad\qquad\qquad\qquad \mathit{True} \to \mathit{LTNode}\ (m+1)\ l\ a\ r \\
 & & \qquad\qquad\qquad\qquad \mathit{False} \to \mathit{LTNode}\ (n+1)\ r\ a\ l \\
 & & \textbf{where}\ (\mathit{LTNode}\ m\ \_\ \_\ \_) = r
\end{array}
$$

With this definition we would still need a single cell, while preserving the tree given as first parameter. This comes at the cost of readability: now the pattern matching is made explicit via **case**. We could avoid this by incorporating the as-patterns facility of Haskell [76]:

$$
\begin{array}{l}
\mathit{cons}\ l@(\mathit{LTEmpty})\ a\ r = \mathit{LTNode}\ 1\ r\ a\ (\mathit{LTEmpty}) \\
\mathit{cons}\ l@(\mathit{LTNode}\ n\ \_\ \_\ \_)\ a\ r = \\
\qquad \mid n \geq m = \mathit{LTNode}\ (m+1)\ l\ a\ r \\
\qquad \mid n < m = \mathit{LTNode}\ (n+1)\ r\ a\ l \\
\qquad\quad \textbf{where}\ (\mathit{LTNode}\ m\ \_\ \_\ \_) = r
\end{array}
$$

Since the current version of *Safe* does not support this kind of as-patterns, we keep the destructive version shown in Figure 3.10.

The *join* function returns the union of two priority queues. Both parameters of this function are condemned, so this operation can be implemented in constant heap space. The fact that both parameters are condemned implies that, in the case in which the tree passed as first parameter is empty, we have to reuse the tree passed as second parameter, since the destruction of the latter does not begin until the second equation of *join*. However, we cannot do the same for the case in which the second parameter is empty, since by the time the execution reaches this equation the destructive pattern matching on the first parameter has already taken place. We are no longer able to reuse a cell that has been destroyed. That is why we have to rebuild the first parameter in the second equation of *join*. The *Core-Safe* translation (without regions) of this function may shed a light on the order in which these destructions are done:

$$
\begin{array}{lcl}
\mathit{join}\ l\ r & = & \textbf{case!}\ l\ \textbf{of} \\
 & & \quad \mathit{LTEmpty} \to r \\
 & & \quad \mathit{LTNode} \to \textbf{case!}\ r\ \textbf{of} \\
 & & \qquad\qquad \mathit{LTEmpty} \to l \quad \{\text{error: } l \text{ has been destroyed}\} \\
 & & \qquad\qquad \ldots
\end{array}
$$

The remaining four functions make up the interface of priority queues: creation of an empty queue (*emptyPQueue*), insertion (*addPQueue*), access to the minimal element (*minPQueue*), and removal of the minimal element (*delMinPQueue*). All these functions need constant additional heap space. They are well-typed with respect to the type system explained in this chapter, and they get the following types:

$$
\begin{aligned}
\mathit{cons} &:: \mathit{Leftist}\ \alpha\ !@\ \rho \to \alpha \to \mathit{Leftist}\ \alpha\ !@\ \rho \to \rho \to \mathit{Leftist}\ \alpha\ @\ \rho \\
\mathit{join} &:: \mathit{Leftist\ Int}\ !@\ \rho \to \mathit{Leftist\ Int}\ !@\ \rho \to \rho \to \mathit{Leftist\ Int}\ @\ \rho \\
\mathit{emptyPQueue} &:: \rho \to \mathit{Leftist}\ \alpha\ @\ \rho \\
\mathit{addPQueue} &:: \mathit{Leftist\ Int}\ !@\ \rho \to \mathit{Int} \to \rho \to \mathit{Leftist\ Int}\ @\ \rho \\
\mathit{minPQueue} &:: \mathit{Leftist}\ \alpha\ @\ \rho \to \alpha \\
\mathit{delMinPQueue} &:: \mathit{Leftist\ Int}\ !@\ \rho \to \rho \to \mathit{Leftist\ Int}\ @\ \rho
\end{aligned}
$$

All type variables and RTVs in each type are assumed to be universally quantified. The *join* function performs a comparison of the elements of the priority queue with the $\leq$ operator. *Safe* does not support type classes in this moment (in particular the *Ord* class), so this operator has type $\mathit{Int} \to \mathit{Int} \to \mathit{Int}$. As a consequence, the *join* function must be applied to priority queues of integers. The same happens with *addPQueue* and *delMinPQueue*, because of their respective calls to *join*. The *cons* and *join* functions forces the input data structures to live in the same region, since they are coalesced in a single data structure. The corresponding RTV ρ has to be inserted into the signatures of those functions that directly or indirectly build a priority queue. The *minPQueue* function does not build anything, and it does not require this parameter. □

Example 3.14. As another example of a data structure with efficient memory usage, let us consider the following implementation of AVL Trees. Again, the definitions shown here are destructive variants of those appearing in [94]. The AVL tree data type is defined as follows:

$$
\mathbf{data}\ \mathit{AVLTree}\ \alpha = \mathit{AVLEmpty} \mid \mathit{AVLNode\ Int}\ (\mathit{AVLTree}\ \alpha)\ \alpha\ (\mathit{AVLTree}\ \alpha)
$$

The *AVLNode* constructor takes an integer standing for the depth of the tree. This is necessary in order to keep the invariant that the balance factor of each node (that is, the difference between the depths of the left and right subtrees) must be -1, 0 or $+1$. Any tree whose nodes satisfy this property is well-balanced. The function *sJoinAVL* is a wrapper of the *AVLTree* constructor and maintains the depth information. It is defined as follows:

$$
\mathit{sJoinAVL}\ l\ a\ r = \mathit{AVLNode}\ (1 + \max\ (\mathit{depth}\ l)\ (\mathit{depth}\ r))\ l\ a\ r
$$

The insertion function in an AVL tree is similar to the insertion in a binary search tree. However, we have to ensure that the invariant of the data structure still holds. If, after insertion, the right subtree of a node outweighs its left subtree (that is, we have a balance factor greater or equal than $+2$) a left rotation or a double left rotation may be needed. This case is handled by the *rJoinAVL* function, which is destructive on its third parameter:

$$
\begin{aligned}
&\mathit{rJoinAVL}\ l\ a\ (\mathit{AVLNode}\ \_\ \mathit{rl}\ \mathit{ra}\ \mathit{rr})! \\
&\quad \mid \mathit{rrd} \geq \mathit{rld} = \mathit{sJoinAVL}\ (\mathit{sJoinAVL}\ l\ a\ \mathit{rl})\ \mathit{ra}\ \mathit{rr} \\
&\quad \mid \mathit{rrd} < \mathit{rld} = \mathbf{case!}\ \mathit{rl}\ \mathbf{of} \\
&\qquad\qquad\qquad (\mathit{AVLNode}\ \mathit{rll}\ \mathit{rla}\ \mathit{rlr}) \to \mathit{sJoinAVL}\ (\mathit{sJoinAVL}\ l\ a\ \mathit{rll})\ \mathit{rla}\ (\mathit{sJoinAVL}\ \mathit{rlr}\ \mathit{ra}\ \mathit{rr}) \\
&\qquad \mathbf{where}\ \mathit{rld} = \mathit{depth}\ \mathit{rl} \\
&\qquad\qquad\ \ \mathit{rrd} = \mathit{depth}\ \mathit{rr}
\end{aligned}
$$

Variables *rl*, *rr*, *rll* and *rlr* get a condemned type, since they are recursive children of data structures being destroyed. As a consequence, *rl* can be used in the destructive **case**!.

Similarly, if the left subtree of a node outweights its right subtree after an insertion, a right rotation or a double right rotation would be necessary. The function *lJoinAVL* handles these cases. Its code is similar to that of *rJoinAVL* and it will not be shown here. So far, we get the following type signatures:

$$\begin{aligned} sJoinAVL &:: AVLTree\ \alpha\ @\ \rho \to \alpha \to AVLTree\ \alpha\ @\ \rho \to \rho \to AVLTree\ \alpha\ @\ \rho \\ rJoinAVL &:: AVLTree\ \alpha\ @\ \rho \to \alpha \to AVLTree\ \alpha\ !@\ \rho \to \rho \to AVLTree\ \alpha\ @\ \rho \\ lJoinAVL &:: AVLTree\ \alpha\ !@\ \rho \to \alpha \to AVLTree\ \alpha\ @\ \rho \to \rho \to AVLTree\ \alpha\ @\ \rho \end{aligned}$$

All these functions need constant heap space (one cell) to run. The *joinAVL* function is another wrapper for *AVLNode*, but, unlike *sJoinAVL*, this one ensures that the resulting tree is well-balanced. It is defined as follows:

$$\begin{aligned} &joinAVL\ l!\ a\ r! \\ &\quad |\ abs\ (ld - rd) \leq 1 \quad = \quad sJoinAVL\ l\ a\ r \\ &\quad |\ ld == rd + 2 \quad = \quad lJoinAVL\ l\ a\ r \\ &\quad |\ ld + 2 == rd \quad = \quad rJoinAVL\ l\ a\ r \\ &\quad |\ ld > rd + 2 \quad = \quad \textbf{case}!\ l\ \textbf{of}\ (AVLNode\ ll\ la\ lr) \to joinAVL\ ll\ la\ (joinAVL\ lr\ a\ r) \\ &\quad |\ ld + 2 < rd \quad = \quad \textbf{case}!\ r\ \textbf{of}\ (AVLNode\ rl\ ra\ rr) \to joinAVL\ (joinAVL\ l\ a\ rl)\ ra\ rr \\ &\quad\ \ \textbf{where}\ ld = depth\ l \\ &\qquad\qquad rd = depth\ r \end{aligned}$$

The *Core-Safe* code corresponding to this function is a sequence of **case** expressions whose branches are the right hand sides of each guard in *joinAVL*. The DS pointed to by *r* is destroyed in the call to *rJoinAVL*. Similarly, the DS corresponding to *l* is destroyed in the call to *lJoinAVL*. We get the following type for *joinAVL*:

$$joinAVL :: AVLTree\ \alpha\ !@\ \rho \to \alpha \to AVLTree\ \alpha\ !@\ \rho \to \rho \to AVLTree\ \alpha\ @\ \rho$$

Given the auxiliary functions above, the implementation of the usual insertion and deletion operations is straightforward. Their corresponding functions are accepted by the type system:

$$\begin{aligned} insertAVL &:: Int \to AVLTree\ Int\ !@\ \rho \to \rho \to AVLTree\ Int\ @\ \rho \\ deleteAVL &:: Int \to AVLTree\ Int\ !@\ \rho \to \rho \to AVLTree\ Int\ @\ \rho \end{aligned}$$

□

3.6 Correctness of the type system

The correctness proof of the type system is rather involved. In a first step (Section 3.6.1) we will set up a connection between region type variables and runtime region numbers, and we will show that this connection remains constant during the evaluation of an expression. In Section 3.6.2 we will introduce some notation and definitions in order to formalize the notion of sharing between data structures.

Some of the rules of the type system contain appearances of a *sharerec* function and an *isTree* predicate, which are determined by a set of external static analyses. Although these analyses are not described in this thesis, we have to impose some requirements on them in order to prove the correctness of the type system. Hence, Section 3.6.3 deals with the notion of a *correct* sharing analysis. Finally, in Section 3.6.4 we put all these results together in order to show that the evaluation of a well-typed program does not fail as a consequence of accessing dangling pointers.

3.6.1 Preservation of region consistency

A key point in the correctness proof of the type system involves finding a correspondence between static region types and runtime region identifiers. For instance, assume a *Safe* program in which a variable x has type $[Int]@\rho$ at a given point. When executing that program, x must point to a list of integers. Assume that list is located in a region whose identifier is 7. At this point of execution, there is a correspondence between the outermost RTV in the type of x and the actual region in which the DS pointed to by x lives. This correspondence is given by the binding $[\rho \mapsto 7]$. More generally, we can define this correspondence by means of a *region instantiation*, defined as follows:

Definition 3.15. A region instantiation $\eta : \mathbf{RegType} \to \mathbb{N}$ is a function from RTVs to natural numbers (interpreted as region identifiers).

An important concept regarding RTVs and region identifiers is the notion of *consistency*. Roughly speaking, the consistency property demands that the region instantiations determined by the types and actual regions of the different variables in scope do not contradict each other. In the example above, if there exists another variable y of type $BSTree\ \alpha\ @\ \rho$, the DS pointed to by this variable must be located, at runtime, in the same region as x. Moreover, every DS being constructed with a region variable of type ρ will also live in that region. Otherwise we would reach an inconsistent configuration.

Definition 3.16. Two region instantiations η and η' are said to be consistent if they bind common RTVs to the same number, that is: $\forall \rho \in \mathrm{dom}\ \eta \cap \mathrm{dom}\ \eta' . \eta(\rho) = \eta'(\rho)$.

The union of two region instantiations η and η' (denoted by $\eta \uplus \eta'$) is defined only if η and η' are consistent and its result is the union of the corresponding bindings. The following definition specifies how to build a region instantiation.

Definition 3.17. Given a heap h, a pointer p, and a safe type s, the function *build* is defined as follows:

$$
\begin{array}{lcll}
build(h, c, s) & = & \emptyset & \text{if } s \in \{\alpha, B\} \\
build(h, p, T\ @\ \rho) & = & \emptyset & \text{if } p \notin \mathrm{dom}\ h \\
build(h, p, T\ @\ \rho) & = & [\rho \mapsto j] \uplus \biguplus_{i=1}^{n} build(h, v_i, s_i) & \text{if } p \in \mathrm{dom}\ h \\
\multicolumn{4}{r}{\textbf{where} \quad h(p) = (j, C\ \overline{v_i}^n)} \\
\multicolumn{4}{r}{\overline{s_i}^n \to \rho \to T\ @\ \rho \trianglelefteq \Gamma(C)}
\end{array}
$$

provided the resulting region instantiation is well-defined. This means that the $\uplus$ operator is only applied to consistent region instantiations.

Example 3.18. Given the heap in the left-hand side of Figure 3.11, in which the x variable points to a list of lists, assume x has type $[[Int]@\rho_2]@\rho_1$. Every variable pointing directly to one of the cells of the outermost list must have ρ_1 as the outermost RTV in its type. On the contrary, if a variable points to one

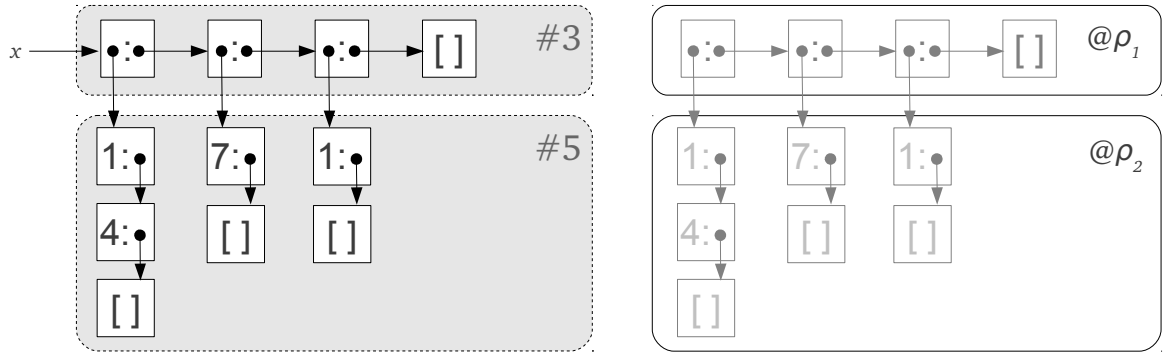


Figure 3.11: Correspondence between the runtime regions in which a DS lives (left) and the RTVs in its type (right).

of the innermost lists, that variable must be of type $[Int]@\rho_2$. In this case, the *build* function returns the following region instantiation, which is well-defined:

$$\eta = [\rho_1 \mapsto 3, \rho_2 \mapsto 5]$$

□

Example 3.19. Assume the following heap:

$$h = \begin{bmatrix} p_1 & \mapsto & (2, (:)\ 0\ p_2) \\ p_2 & \mapsto & (3, [\,]) \end{bmatrix}$$

The result of $build(h, p_1, s)$ is not well-defined for any algebraic type s. Assume $s = T\ @\ \rho$. By applying the definition of *build* we get $build(h, p_1, T\ @\ \rho) = [\rho \mapsto 2] \uplus [\rho \mapsto 3]$, which is not well-defined. The reason behind this is that the list pointed to by p_1 is distributed between two different regions (2 and 3), and a DS must live in a single region. □

Besides the triple (h, p, s), the result of *build* also depends on a type environment Γ containing the type of the constructors that can be reached by following the pointers that stem from p. By abuse of notation, we consider this typing environment implicit, as the *build* function only needs the types of the constructors, which are defined globally in the program.

We can extend the *build* function to deal with condemned and in-danger types:

$$\begin{aligned} build(h, p, T\ !@\ \rho) &= build(h, p, T\ @\ \rho) \\ build(h, p, T\ \#@\ \rho) &= build(h, p, T\ @\ \rho) \end{aligned}$$

So far we have determined the region instantiation determined by a single variable. We can extend the *build* function in order to group the region instantiations of all the variables in a runtime environment E, and whose types can be found in a type environment Γ.

Definition 3.20. Given a heap h, a value environment E, and a type environment Γ, we define the function $build^*$ as follows:

$$build^*(h, E, \Gamma) = \biguplus_{x \in X} build(h, E(x), \Gamma(x)) \uplus \biguplus_{r \in X} [\Gamma(r) \mapsto E(r)] \quad \text{where } X = \operatorname{dom} E \cap \operatorname{dom} \Gamma$$

provided the resulting region instantiation is well-defined.

Notice that the result of $build^*$ is well-defined only if the region instantiations of the different variables in X do not contradict each other.

Example 3.21. Assume the following heap h:

Assume a runtime environment E mapping x and y to the cells of the figure above. If we define $\Gamma = [x : [Int]@\rho, y : BSTree\ Int\ @\ \rho']$, then $build^*(h, E, \Gamma)$ results in the following well-defined region instantiation:

$$build^*(h, E, \Gamma) = [\rho \mapsto 1, \rho' \mapsto 2]$$

On the contrary, if we consider the type environment $\Gamma' = [x : [Int]@\rho, y : BSTree\ Int\ @\ \rho]$, then $build^*(h, E, \Gamma)$ is not well-defined, since the tree pointed to by y does not live in the same region as the list pointed to by x. □

Assume a heap h such that $build(h, p, \tau)$ is well-defined for some pointer p and type τ. If h' is a heap that results from removing some pointers from h, it is easy to see that $build(h', p, \tau)$ is also well-defined, although it could contain less bindings than the original $build$. However, if we extend h with new bindings we cannot ensure that the corresponding $build$ is well-defined. For instance, in Example 3.19, $build(h|_{\{p_1\}}, p_1, [\alpha]@\rho)$ is well-defined, but $build(h, p_1, [\alpha]@\rho)$ is not. When extending a heap, we can ensure the preservation of consistency only if the newly added cells are not reachable from the already existing ones. A heap h' is said to strictly extend a heap h if $h \subseteq h'$ and no pointer in $\mathrm{dom}\, h' \backslash \mathrm{dom}\, h$ is reachable from any pointer in h. That is, if the pointers we add in h' are fresh. We denote this by $h \subseteq_s h'$. The following lemma formalises all these results.

Lemma 3.22. *Let h and h' be two heaps. The following two properties hold for each pointer $p \in dom(h)$ and region instantiation η:*

1. If $h \subseteq h'$, then $build(h', p, t) \supseteq build(h, p, t)$, provided $build(h', p, t)$ is well-defined
2. If $h \subseteq_s h'$, then $build(h', p, t) = build(h, p, t)$

Proof. By induction on the size of the structure pointed to by p. □

During the evaluation of an expression some cells can be removed from the heap, or new cells can be created, but the contents of a cell are never updated. The following lemma follows from this fact.

Lemma 3.23. *Let $E \vdash h, k, e \Downarrow h', k, v$ be an execution of an expression. For all Γ, if $build^*(h, E, \Gamma)$ is well-defined, then so is $build^*(h', E, \Gamma)$. Moreover, it holds that $build^*(h', E, \Gamma) \subseteq build^*(h, E, \Gamma)$.*

Proof. This holds because the closure of the variables in E can only decrease as e is evaluated (because of destruction), but the contents of the cells in the heap are never updated (see Lemma 3.29 below). □

The following auxiliary lemma shows that consistency is preserved when copying a data structure to a different region (see Definition 2.13).

Lemma 3.24. *Let us assume that* $(h', p') = copy(h, p, j)$. *If* η *is consistent with* $build(h, p, T\,@\,\rho)$ *and with* $[\rho' \mapsto j]$, *then* $build(h', p', T\,@\,\rho')$, *is well-defined and consistent with* η.

Proof. By induction on the size of the structure pointed to by p. Let us assume that $h(p) = (k, C\ \overline{v_i}^n)$ and that $\overline{s_i}^n \to \rho \to T\,@\,\rho \trianglelefteq \Gamma(C)$ and $\overline{s'_i}^n \to \rho' \to T\,@\,\rho' \trianglelefteq \Gamma(C)$. From the way in which the types of the constructors are inferred, it follows that $s_i = s'_i$ if $i \notin RecPos(C)$. We get:

$$build(h, p, T\,@\,\rho) \quad = \quad [\rho \mapsto k] \uplus build(h, v_1, s_1) \uplus \cdots \uplus build(h, v_n, s_n)$$

By the definition of *copy*, there is a chain of heaps $h = h_0 \subseteq_s h_1 \subseteq_s \ldots \subseteq_s h_n \subseteq_s h'$ such that for every $i \in \{1 \ldots n\}$, there exists a value v'_i such that either $(h_i, v'_i) = copy(h_{i-1}, v_i, j)$ or $(h_i, v'_i) = (h_{i-1}, v_i)$. Since η is consistent with each $build(h, v_i, s_i)$, it is also consistent with each $build(h_{i-1}, v_i, s_i)$ by Lemma 3.22 (2). We have to prove that η is also consistent with $build(h_i, v'_i, s'_i)$. We distinguish cases for each $i \in \{1 \ldots n\}$:

- $i \notin RecPos(C)$

  Then $h_i = h_{i-1}$, $s'_i = s_i$ and $v'_i = v_i$. Then the consistency of η with $build(h_i, v'_i, s'_i)$ follows trivially from the consistency of η with $build(h_{i-1}, v_i, s_i)$.

- $i \in RecPos(C)$

  Then $(h_i, v'_i) = copy(h_{i-1}, v_i, j)$, $s_i = T\,@\,\rho$ and $s'_i = T\ @\ \rho'$. We apply the induction hypothesis in order to obtain that $build(h_i, v'_i, s'_i)$ is well-defined and consistent with η.

Since h' is a strict extension of every h_i, we can apply Lemma 3.22 (2) again in order to establish the consistency of η with respect to each $build(h', v'_i, s'_i)$. By the definition of *build* we get:

$$build(h, p', T\,@\,\rho') = [\rho' \mapsto j] \uplus build(h', v'_1, s'_1) \uplus \cdots \uplus build(h', v'_n, s'_n)$$

which proves the lemma, since η is consistent with each component in the right-hand side.

□

We can prove now that consistency is preserved during the evaluation of an expression e. As usual, this is proved by induction on the corresponding $\Downarrow$-derivation. Nevertheless, we have to be careful when handling the case of function applications. In principle, we can safely assume that the type schemes of the functions being referenced in e are present in the type environment, but we have no clue on how these type schemes have been obtained. We say that these types schemes are *correct* if they have been obtained via the [FUN] rule of Figure 3.9.

Definition 3.25. A typing environment Γ is said to be *correct* with respect to a function signature Σ if for every function symbol $f \in \text{dom}\ \Gamma$ it holds that $f\ \overline{x_i}\,@\,\overline{r_j} = e_f \in \Sigma$ for some $\overline{x_i}$, $\overline{r_j}$, e_f, and the following judgement is derivable by using the [FUN] rule:

$$\{\Gamma' \backslash f\}\ f\ \overline{x_i}\,@\,\overline{r_j} = e_f\ \{\Gamma'\}$$

for some $\Gamma' \subseteq \Gamma$ such that $f \in \text{dom}\ \Gamma'$.

The function signature Σ is usually clear from the context, so let us say in the following that a given Γ is correct without making any mention to Σ. Now we are able to establish that consistency is preserved during the evaluation of an expression.

Theorem 3.26. *Let us assume the execution of an expression:*

$$E \vdash h, k, e \Downarrow h', k, v \tag{3.7}$$

Then, for every Γ, s, η such that $\Gamma \vdash e : s$ holds and Γ is a correct environment, if $build^*(h, E, \Gamma)$ is well-defined and consistent with η, then $build(h', v, s)$ is well-defined and consistent with η, and for every sub-expression e_i such that the judgement $E_i \vdash h_i, k, e_i \Downarrow h'_i, k, v_i$ belongs to the derivation of (3.7) and $\Gamma_i \vdash e_i : s_i$ for some E_i, h_i, e_i, h'_i, v_i, Γ_i, the result of $build^*(h_i, E_i, \Gamma_i)$ is well-defined and consistent with η.

Proof. Let us focus on the last rule applied in the derivation of $\Gamma \vdash e : s$. If this rule is [EXT] or [WEAK] we obtain $\Gamma_1 \vdash e : s$ for some $\Gamma_1 \subset \Gamma$. By repeating this process we can prove the existence of Γ' and Γ'' such that $\Gamma' \leq \Gamma'' \subseteq \Gamma$, $\Gamma' \vdash e : s$, and the last rule applied in this latter derivation is neither [EXT] nor [WEAK]. It is easy to see that $build^*(E, h, \Gamma')$ is well-defined and consistent with η, since the marks do not play any rule in the definition of *build*.

Given these considerations, we proceed by induction on the depth of the $\Downarrow$-derivation, We distinguish cases on the expression being evaluated:

- **Case** $e \equiv c$

  We get $\Gamma' \vdash c : B$, and $build(h, c, B) = \emptyset$, which is trivially well-defined and consistent with η.

- **Case** $e \equiv x$

  Since $x \in \text{dom } E$, $build(h, E(x), \Gamma'(x))$ is well-defined and consistent with η, and hence $build(h, v, s)$ is also well-defined and consistent with η.

- **Case** $e \equiv x \,@\, r$

  We get $\Gamma' = [x : T \,@\, \rho', r : \rho] \vdash x \,@\, r : T \,@\, \rho$. Let us assume that $[r \mapsto j, x \mapsto p] \subseteq E$. Then η is consistent with $[\rho \mapsto j]$. Therefore, by using Lemma 3.24, we prove that η is consistent with $build(h', p', T \,@\, \rho)$, where $(h', p') = copy(h', p, j)$.

- **Case** $e \equiv C\ \overline{a_i}^n \,@\, r$

  We assume that $\Gamma' \vdash C\ \overline{a_i}^n \,@\, r : T \,@\, \rho$ with $\Gamma'(r) = \rho$ and $E(r) = j$. We get:

$$\begin{aligned} build^*(h, E, \Gamma) &\supseteq [\rho \mapsto j] \uplus build(h, E(a_1), \Gamma'(a_1)) \uplus \ldots \uplus build(h, E(a_n), \Gamma'(a_n)) \\ &= [\rho \mapsto j] \uplus build(h', E(a_1), \Gamma'(a_1)) \uplus \ldots \uplus build(h', E(a_n), \Gamma'(a_n)) \end{aligned}$$

  The last equality is due to h' being a strict extension of h, and Lemma 3.22 (2). All the components of the right-hand side of this equality are pairwise consistent, and also consistent with η. Therefore, the lemma holds, as the result of $build(h, v, s)$ is just this right-hand side.

- **Case** $e \equiv g\ \overline{a_i}^n \,@\, \overline{r_j}^m$

  We assume that $\Gamma'(g) = \forall \overline{\alpha}\, \overline{\rho}.\overline{t_{g,i}}^n \to \overline{\rho_{g,j}}^m \to s_g$ and that $g\ \overline{x_i}^n \,@\, \overline{r'_j}^m = e_g \in \Sigma$. Since Γ is correct with respect to Σ (and hence Γ' also is) we can apply the [FUN] rule, so that $\{\Gamma_0 \backslash g\} g\ \overline{x_i}^n \,@\, \overline{r'_j}^m = e_g\{\Gamma_0\}$ for some environment Γ_0 containing the type of g. Thus we get:

$$\Gamma_0 \vdash e_g : s_g \qquad \text{where } \Gamma_0 \supseteq [\overline{x_i : t_{g,i}}^n, \overline{r'_j : \rho_{g,j}}^m, self : \rho^g_{self}] \tag{3.8}$$

  The region type ρ^g_{self} is a fresh variable, so we can ensure that is different from the $\overline{\rho_{g,j}}^m$, from the $\overline{\Gamma'(r_j)}^m$, and from any variable appearing in the domain of η. From one of the premises of the

[APP] rule we can assume the existence of a type substitution θ such that $\theta(\overline{t_{g,i}}^n \to \overline{\rho_{g,j}}^m \to s_g) = \overline{\Gamma'(a_i)}^n \to \overline{\Gamma'(r_j)}^m \to s$ and that $\rho^g_{self} \notin \mathrm{dom}\,\theta$. We apply Lemma 3.9 to (3.8) so as to obtain:

$$\Gamma_g \vdash e_g : s \qquad \text{where } \Gamma_g = \theta(\Gamma_0) \supseteq [\overline{x_i : \Gamma'(a_i)}^n, \overline{r'_j : \Gamma'(r_j)}^m, self : \rho^g_{self}]$$

Assume the following execution of the function's body:

$$E_g \vdash h, k+1, e_g \Downarrow h^+, k+1, v$$

where $h' = h^+|_k$ and $E_g \stackrel{\text{def}}{=} [\overline{x_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^m, self \mapsto k+1]$. In order to apply the induction hypothesis, we have to prove that $build^*(h, E_g, \Gamma_g)$ is well-defined and consistent with η. This is a consequence of $build^*(h, E, \Gamma)$ having these properties and ρ^g_{self} being a fresh variable:

$$\begin{aligned} build^*(h, E_g, \Gamma_g) &= \biguplus_{i=1}^{n} build(h, E_g(x_i), \Gamma_g(x_i)) \uplus \biguplus_{j=1}^{m} build(h, E_g(r'_j), \Gamma_g(r'_j)) \uplus [\rho^g_{self} \mapsto k+1] \\ &= \biguplus_{i=1}^{n} build(h, E(a_i), \Gamma'(r_j)) \uplus \biguplus_{j=1}^{m} build(h, E(r_j), \Gamma'(r_j)) \uplus [\rho^g_{self} \mapsto k+1] \end{aligned}$$

Therefore we obtain, by induction hypothesis, a region instantiation $build(h^+, v, s)$ which is well-defined and consistent with η. The result of $build(h', v, s)$ also has these properties, by Lemma 3.22 (1).

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

  We assume that $\Gamma' = \Gamma_1 \sqcup \Gamma_2, \Gamma_1 \vdash e_1 : s_1, \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s_2$ and the following executions:

  $$E \vdash h, k, e_1 \Downarrow h_1, k, v_1$$

  $$E \uplus [x_1 \mapsto v_1] \vdash h_1, k, e_2 \Downarrow h', k, v$$

  From the assumptions on $build^*(h, E, \Gamma)$ we can prove that both $build^*(h, E, \Gamma_1)$ and $build^*(h, E, \Gamma_2)$ are well-defined and consistent with η, and so is $build^*(h_1, E, \Gamma_2)$, because of Lemma 3.23. By the induction hypothesis, $build(h_1, v_1, s_1)$ is well-defined and consistent with η and hence the same applies to $build^*(h_1, E \uplus [x_1 \mapsto v_1], \Gamma_2 + [x_1 : \tau_1])$. Again, the induction hypothesis on the derivation of e_2 gives the required result.

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n$

  Let us assume that $E(x) = p$ and that $h(p) = (j, C_r\ \overline{v_j}^{n_r})$. That is, the r-th branch is executed:

  $$E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow h', k, v$$

  By the typing rule for **case** expressions, we obtain $\Gamma' + [\overline{x_{rj} : \tau_{rj}}^{n_r}] \vdash e_r : s$ where $\overline{s_j}^{n_r} \to \rho \to T\,@\,\rho \trianglelefteq \Gamma(C_r)$ and $utype?(\tau_{ri}, s_j)$ for every $j \in \{1 \ldots n_r\}$. If $\Gamma(x) = T\,@\,\rho$, the region instantiation $build(h, p, T\,@\,\rho)$ is contained within $build^*(h, E, \Gamma)$ and hence is consistent with η. Let us unfold

its definition:

$$\begin{aligned} build(h, p, T\,@\,\rho) &= [\rho \mapsto j] \uplus build(h, v_1, s_1) \uplus \ldots \uplus build(h, v_{n_r}, s_{n_r}) \\ &= [\rho \mapsto j] \uplus build(h, v_1, \tau_{r1}) \uplus \ldots \uplus build(h, v_{n_r}, \tau_{r,n_r}) \end{aligned}$$

Each one of the region instantiations of the right-hand side is well-defined and consistent with η. Therefore the result of $build^*(h, E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}], \Gamma' + [\overline{x_{rj} : \tau_{rj}}^{n_r}])$ is also well-defined and consistent with η, and we can apply the induction hypothesis on the derivation of the evaluation of e_r, which leads to the desired result.

- **Case** $e \equiv$ **case**! x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

  The proof is very similar to that of the non-destructive **case**. The only difference is the fact that p is now a dangling pointer, but we still can prove that $build^*(h \backslash p, E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}], \Gamma' + [\overline{x_{rj} : t_{rj}}^{n_r}])$, because of Lemma 3.22 (1).

□

To sum up, we have set up a correspondence between the actual regions where a DSs resides and the RTVs assigned by the type system: if two variables have the same outer region ρ at runtime, the cells bound to them at runtime will live in the same actual region. Since the type system (see rule [FUN] in Figure 3.9) demands the ρ_{self} variable not to occur in the type of the function's result, every DS returned by the function call does not have cells in its working region (assuming *self* is the only variable pointing to the topmost region, and the only variable of type ρ_{self}). This implies that the deallocation of the $k+1$-th region at the end of a function call does not generate dangling pointers. This will be detailed in the correctness proof of Section 3.6.4.

3.6.2 Reachability and harmless semantics

The concepts introduced so far are related to region-based memory management. The next sections on the type system's correctness are devoted to the safety of explicit deallocation via **case**! expressions. A crucial aspect in this part of the proof is the notion of *sharing* between DSs. Two DSs are said to share if there exists a cell in the heap that is reachable from both. Therefore, in order to formally define sharing we have to specify a notion of *reachability* in the heap. It was already pointed up in Section 3.4 that the connections between the cells in a heap can be represented by means of a directed acyclic graph. Thus, our notion of cell reachability is directly related to the notion of reachability in DAGs. However, for the purposes of the type system it is necessary to distinguish the cells that are reachable from a given one by following those pointers occurring in the recursive positions of the constructors being traversed (which we shall call their *recursive descendants*). This motivates the following definition:

Definition 3.27. Given a heap h, we define the child ($\to_h$) and recursive child ($\twoheadrightarrow_h$) relations on heap pointers as follows:

$$\begin{aligned} p \to_h q &\stackrel{\text{def}}{=} h(p) = (j, C\ \overline{v_i}^n) \wedge q \in \overline{v_i}^n \\ p \twoheadrightarrow_h q &\stackrel{\text{def}}{=} h(p) = (j, C\ \overline{v_i}^n) \wedge q = v_i \\ &\qquad \text{for some } i \in RecPos(C) \end{aligned}$$

The reflexive transitive closure of these relations is respectively denoted by $\to_h^*$ and $\twoheadrightarrow_h^*$. If $p \to_h^* q$ we say that q is *reachable* from p in h. If $p \twoheadrightarrow_h^* q$ we say that q is a *recursive descendant* of p in h. These definitions serve as a basis for specifying the following notions of reachability:

Definition 3.28.

$closure(h, p)$	$\stackrel{\text{def}}{=}$	$\{q \mid p \rightarrow_h^* q\}$	Set of locations reachable in h by location p.
$live(h, E, L)$	$\stackrel{\text{def}}{=}$	$\bigcup_{x \in L} closure(h, E(x))$	Live part of h, i.e. reachable from any free variable.
$recReach(h, p)$	$\stackrel{\text{def}}{=}$	$\{q \mid p \twoheadrightarrow_h^* q\}$	Set of recursive descendants of $E(x)$ including itself.
$closed(h, p)$	$\stackrel{\text{def}}{=}$	$closure(h, p) \subseteq \text{dom}\, h$	If there are no dangling pointers in $closure(h, p)$.
$closed(h, E, L)$	$\stackrel{\text{def}}{=}$	$\forall x \in L.\, closed(h, E(x))$	If there are no dangling pointers in $live(h, E, L)$.

As a convention, we define $closure(h, c) = \emptyset$ for any heap h and literal c. There are several general properties of the language regarding closures. The first one specifies that the closure of a given pointer can decrease during the execution of a program as a consequence of destruction, but it never increases.

Lemma 3.29. *Let $E \vdash h, k, e \Downarrow h', k, v$ be an execution of an expression. Then, for every pointer in dom h, $closure(h, p) \supseteq closure(h', p)$.*

Proof. By induction on the $\Downarrow$-derivation. If p does not belong to the domain of h', then $closure(h', p) = \emptyset$ and the lemma holds trivially, so let us assume $p \in \text{dom}\, h'$. All cases are straightforward: In rules [*Lit*], [*Var*] and [*PrimOp*] we get $h = h'$. In [*Copy*] and [*Cons*] only fresh cells are created, and these cells cannot be reachable from those already existing in h. In [*App*], [*Let*], [*Case*], [*Case!*] the result follows directly from the induction hypothesis. The only case in which the closure strictly decreases is when removing the location pointed to by the discriminant of a **case**! in [*Case!*] rule. □

The second property shows that, if the evaluation of an expression returns a pointer belonging to the initial heap, this pointer must have been accessed through (at least) one of the free variables in that expression. The same applies to the pointers being removed from the initial heap.

Lemma 3.30. *Let us assume that $E \vdash h, k, e \Downarrow h', k, v$.*

1. *If v is a pointer in dom h, then $v \in live(h, E, fv(e))$.*
2. *For every $p \in dom\ h \backslash dom\ h'$, $p \in live(h, E, fv(e))$.*

Proof. By induction on the size of the $\Downarrow$-derivation. The only base case applicable to (1) is the [*Var*] rule, in which the result follows trivially. In the recursive cases (1) follows trivially from the induction hypothesis. With regard to (2) the only relevant case is that of [*Case!*] rule, since none of the pointers removed in the [*App*] rule as a consequence of deallocating the topmost region, belong to the initial heap. However, the case of the [*Case!*] rule is also trivial, as the discriminant is a free variable referring to the pointer being removed. □

Finally, the next property shows that only the live part of the heap is relevant when evaluating an expression. The rest of the heap can be safely discarded from the $\Downarrow$-judgements.

Lemma 3.31. *Let us assume an execution $E \vdash h, k, e \Downarrow h', k, v$ and p a pointer in dom h. If $p \notin live(h, E, fv(e))$, then we can discard the binding of p in h so as to obtain $E \vdash h \backslash p, k, e \Downarrow h' \backslash p, k, v$, and this derivation has the same size as the original one.*

Proof. By induction on the size of the $\Downarrow$-derivation. All cases are straightforward, except in the case in which the last rule applied is [*Let*]. In this case the first property of Lemma 3.30 is necessary after applying the induction hypothesis to the bound expression. After this, the result holds trivially. □

In Figure 2.14 we showed the big-step operational semantic rules of *Core-Safe* and we identified the cases in which none of the rules could be applicable for some initial heap h, expression e and runtime environment E. One of these situations is produced when accessing a location p that does not belong to the domain of the heap (that is, a dangling pointer). We want to show that this situation never takes place when the expression is well-typed w.r.t. the type system shown in this chapter, but we are not concerned about the rest of the pathological cases explained in Section 2.4.1 (pattern-matching errors, non-termination, etc.). For this reason we assume the execution of an expression under a *harmless* big-step operational semantics that does not remove pointers from the heap, thus avoiding the possibility of having dangling pointers. We use the $\Downarrow^*$ notation to refer to the judgements of this semantics. In this way, given E, h, k, e of their respective types, if there does not exist a final heap h' and a value v such that the judgement $E \vdash h, k, e \Downarrow^* h', k, v$ is not derivable, we can ensure that this is not due to an access to a dangling pointer, but to another of the reasons explained in Section 2.4.1. On the contrary, in the following sections we will show that $\Downarrow^*$-judgements imply $\Downarrow$-judgements in the context of well-typed programs (see Appendix C).

The rules of the harmless semantics are shown in Figure 3.12. Notice the similarity with those of Figure 2.14. The only difference is that the topmost region is not removed after executing the function's body in the [*App*] rule, and that the binding pointed to by the discriminant of a destructive **case**! is not removed from the heap. Thus, a destructive **case**! behaves just like its non-destructive counterpart.

It is easy to see that whenever we have a judgement $E \vdash h, k, e \Downarrow^* h', k, v$, we get $h' \supseteq h$, since nothing is destroyed during $\Downarrow^*$-evaluations. Moreover, Lemmas 3.29, 3.30 and 3.31 also hold for $\Downarrow^*$-judgements. In particular, we get equality in Lemma 3.29: $\mathit{closure}(h, p) = \mathit{closure}(h', p)$, and Lemma 3.30(2) holds vacuously.

3.6.3 Correctness of the sharing analysis

Recall that our type system depends on some functions and predicates (namely, *sharerec* and *isTree*) which are defined by some auxiliary analyses whose definition are beyond the scope of this thesis. In particular:

$\mathit{sharerec}(x, e)$ Approximates the sharing relations between DSs. It contains all the variables in the scope of e that, at runtime, may point to a recursive descendant of x.

$\mathit{isTree}(x)$ Approximates the internal sharing property of a DS. This predicate holds for all the variables that, at runtime, point to a DS whose associated graph is a tree.

Although we do not explain the analyses themselves (see [98] for more details) we need to formalize their safety properties in order to prove the correctness of the type system. In other words, we have to determine which properties must satisfy a sharing analysis and an internal sharing analysis so that, when connecting their results to our system, the correctness of the latter is not compromised.

Let us start with the *isTree* predicate. Firstly we have to precise the notion of a DS having internal sharing at runtime. The following definition allows us to describe the path of pointers in the heap that must be taken if we start in a pointer p and we end in a recursive descendant q of p.

Definition 3.32. Given two pointers p and q in a heap h, a sequence of natural numbers $[i_1, \ldots, i_m]$ is said to describe the relation $p \twoheadrightarrow_h^* q$ iff there exists a sequence of pointers $[p_1, \ldots, p_m, p_{m+1}] \subseteq \mathrm{dom}\ h$ and a $C \in \mathbf{Cons}$ such that:

1. $p = p_1$ and $q = p_{m+1}$.

$$\frac{}{E \vdash h, k, c \Downarrow^* h, k, c}\ [Lit^*]$$

$$\frac{}{E[x \mapsto v] \vdash h, k, x \Downarrow^* h, k, v}\ [Var^*]$$

$$\frac{}{E \vdash h, k, a_1 \oplus a_2 \Downarrow^* h, k, E(a_1) \oplus E(a_2)}\ [PrimOp^*]$$

$$\frac{j \leq k \qquad (h', p') = copy(h, p, j)}{E[x \mapsto p, r \mapsto j] \vdash h, k, x\,@\,r \Downarrow^* h', k, p'}\ [Copy^*]$$

$$\frac{E(r) \leq k \qquad fresh_h(p)}{E \vdash h, k, C\ \overline{a_i}^n\ @\ r \Downarrow^* h \uplus [p \mapsto (E(r), C\ \overline{E(a_i)}^n)], k, p}\ [Cons^*]$$

$$\frac{(g\ \overline{y_i}^n\ @\ \overline{r'_j}^m = e_g) \in \Sigma \qquad [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^m, self \mapsto k+1] \vdash h, k+1, e_g \Downarrow^* h', k+1, v}{E \vdash h, k, g\ \overline{a_i}^n\ @\ \overline{r_j}^m \Downarrow^* h', k, v}\ [App^*]$$

$$\frac{E \vdash h, k, e_1 \Downarrow^* h_1, k, v_1 \qquad E \uplus [x_1 \mapsto v_1] \vdash h_1, k, e_2 \Downarrow^* h', k, v}{E \vdash h, k, \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \Downarrow^* h', k, v}\ [Let^*]$$

$$\frac{E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow^* h', k, v}{E[x \mapsto p] \vdash h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})], k, \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow^* h', k, v}\ [Case^*]$$

$$\frac{E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow^* h', k, v}{E[x \mapsto p] \vdash h[p \mapsto (j, C_r\ \overline{v_i}^{n_r})], k, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow^* h', k, v}\ [Case!^*]$$

Figure 3.12: Harmless operational semantics of *Core-Safe* expressions. The only difference with respect to the standard big-step semantics of Figure 2.14 is the absence of destruction in the cases of function application and destructive **case**!.

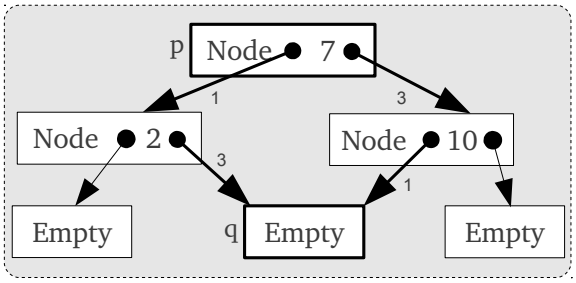


Figure 3.13: DS with internal sharing: there are two paths from p to q.

2. For every $j \in \{1..m\}$, $i_j \in \mathit{RecPos}(C)$.

3. For every $j \in \{1..m\}$, $h(p_j) = (l, C\ \overline{v_k}^n)$ for some $l \in \mathbb{N}$, $\overline{v_k}^n \subseteq \mathbf{Val}$, and $v_{i_j} = p_{j+1}$.

Example 3.33. Given heap h of the Figure 3.13, the relation $p \twoheadrightarrow_h^* q$ is described by the sequence $[1,3]$, but also by the sequence $[3,1]$. □

If a recursive descendant of p can be reached from that pointer by following two different paths, there is internal sharing (confluence) in the DS pointed to by p.

Definition 3.34. Given two pointers p and p' in a heap h, we say that there is a *confluence* from p to p' (denoted $p \rightrightarrows_h^* p'$) iff $p \twoheadrightarrow_h^* p'$ and this relation can be described by two distinct sequences of natural numbers.

Definition 3.35. A pointer in a heap p is said to be a *tree* in h if, for every $p' \in \mathit{recReach}(h, p)$: $p \not\rightrightarrows_h^* p'$.

Example 3.36. The DS pointed to by p in Figure 3.13 is not a tree, as there is a confluence from p to q. □

Once we have made the notion of internal sharing precise, we can define the correctness of an internal sharing analysis. A sensible correctness statement for such an analysis would be the following:

> *Given an execution of an expression $E \vdash h, k, e \Downarrow h', k, v$, for every variable $x \in \mathit{dom}\ E$ such that $E(x)$ is not a tree in h, the internal sharing analysis determines that the predicate $\mathit{isTree}(x)$ does not hold.*

This definition, although reasonable, is too strong for what the type system demands. One can develop an analysis carried out at a function definition level that *assumes that the DSs pointed to by its formal parameters are trees*, and its results of would be suitable for the type system. The rationale behind this is the fact that the type system only needs the *isTree* predicate for checking that destructive pattern matching is done to DSs without internal sharing (otherwise the program might destroy a recursive descendant of a DS twice). So, assume a given variable occurs in the discriminant of a **case**!:

1. If that variable is not a formal parameter of the function, the internal sharing analysis will determine whether that variable is a tree or not, and update the sharing information of the pattern variables of the corresponding branches accordingly.

2. If that variable is a formal parameter of the function, the DS to which it points might not be a tree, but the internal sharing analysis would not warn the type system about this, since the analysis assumes the formal parameters to be trees. However, a parameter appearing in a **case**! would

have a condemned type, and the rule for typing function applications [APP] ensures that the arguments of condemned positions are trees, so the assumption of the internal sharing analysis is correct.

Given the above, the correctness of an internal sharing analysis that only approximates the internal sharing property when the type system needs it, would be defined as follows.

Assumption 3.37. *Given an expression e being executed in the context of a function f.*

$$E \vdash h, k, e \Downarrow^* h', k, v$$

Let P be the set of formal parameters of this function. For every $x \in dom\ E$ and pointer p such that $E(x) \rightrightarrows_h^* p$, at least one of the following properties hold:

$$\begin{array}{ll} (TREE) & \neg isTree(x) \\ (CONF) & \exists y_0 \in P. y_0 \rightrightarrows_h^* p \end{array}$$

In other words, if there is internal sharing at runtime, either the analysis warns the type system about it, or this internal sharing occurs in a DS pointed to by a function parameter.

We follow a similar pattern with the *sharerec* function defined by the sharing analysis. One would expect the following correctness property:

> *Given an execution of an expression $E \vdash h, k, e \Downarrow h', k, v$, for every pair of variables $x, y \in dom\ E$ such that $recReach(h, E(x)) \cap closure(h, E(y)) \neq \emptyset$ the sharing analysis determines that $y \in sharerec(x, e)$.*

But, again, the type system does not need such a strong property. A sharing analysis that approximates this information while assuming that the parameters of the function to which e belongs do not share with each other would be suitable. The sharing analysis is only needed to determine the variables pointing to a recursive descendant of something being destroyed. So, whenever we have two variables x and y such that $recReach(h, E(x)) \cap closure(h, E(y)) \neq \emptyset$ and x is being destroyed, there are three possibilities:

1. This sharing relation is captured by the sharing analysis, so we get $y \in sharerec(x, e)$.

2. The sharing analysis does not warn the type system about this sharing relation, because the latter is due to the sharing between different parameters which the analysis assumed to be disjoint, but actually they are not (Figure 3.14). This case does not compromise the safety of the type system, since the function application rule [APP] checks that the DSs pointed to by the arguments occurring in condemned positions do not share between them, or with the remaining parameters. Since x is being destroyed, at least one of these parameters to which the sharing is due is condemned.

3. The sharing analysis does not warn the type system about this sharing relation, since it is due to a parameter from which x and y are recursive descendants (Figure 3.15). As a consequence, this parameter would have internal sharing, which the internal sharing analysis does not report, as it was explained above. Again, this does not compromise the safety of the type system, since that parameter would be condemned.

These three possibilities are formalized as follows.

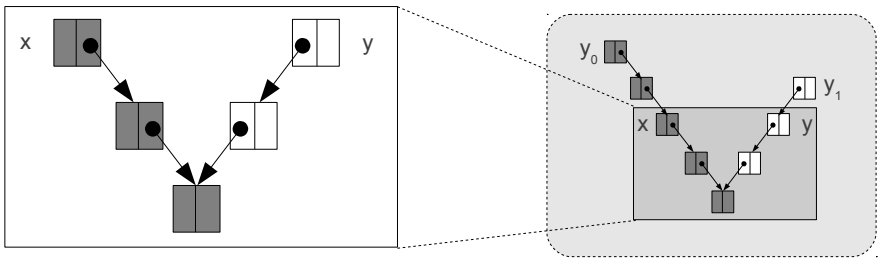


Figure 3.14: If y points to a recursive descendant of x, but this relation is not reported by the sharing analysis, it may be due to sharing between some parameters y_0, y_1 in the context function.

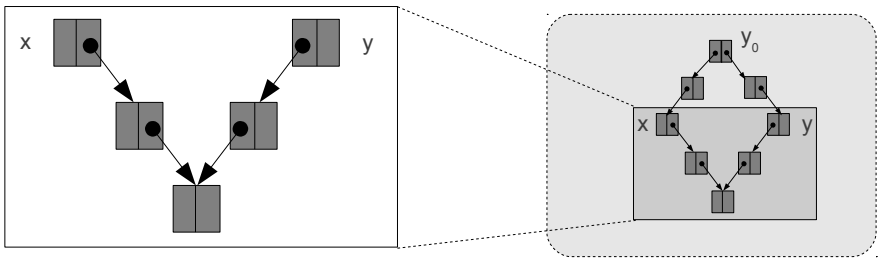


Figure 3.15: If y points to a recursive descendant of x, but this relation is not reported by the sharing analysis, it may be due to a parameter y_0 with internal sharing.

Assumption 3.38. *Given an expression e being executed in the context of a function f.*

$$E \vdash h, k, e \Downarrow^* h', k, v$$

Let P be the set of formal parameters of this function. For every $x, z \in dom\, E$ and pointer p such that $p \in closure(h, E(x)) \cap recReach(h, E(z))$, at least one of the following properties hold:

$$\begin{array}{ll} (SHR) & x \in sharerec(z, e) \\ (SHP) & \exists y_0, y_1 \in P. y_0 \neq y_1 \wedge E(y_0) \twoheadrightarrow_h^* p \wedge E(y_1) \rightarrow_h^* p \\ (CONF) & \exists y_0 \in P. y_0 \rightrightarrows_h^* p \end{array}$$

The sharing relations that take place between parameters deserve special attention. Assume that a parameter y_0 points to a recursive descendant of a local variable x in the context of an expression e. As it was explained before, the sharing analysis might not reflect this sharing relation, because the DS pointed to by this parameter y_0 is the same as the DS pointed to by another hypothetical parameter y_1, both of which are assumed to be disjoint by the sharing analysis. However, and since x is a local variable which has been defined within the context function definition, the sharing analysis should have enough information to determine that at least one of these parameters belongs to $sharerec(x, e)$. That is why the following assumption.

Assumption 3.39. *Assume the execution of an expression in the context of f:*

$$E \vdash h, k, e \Downarrow^* h', k, v$$

Let P be the set of formal parameters of f. For every $x \in dom\, E$ and pointer $p \in recReach(h, E(x)) \cap closure(h, E(y_0))$ for some $y_0 \in P$, then there exists a parameter $y_1 \in P$ such that $p \in recReach(h, E(x)) \cap$

closure$(h, E(y_1))$ *and* $y_1 \in$ *sharerec*(x, e).

Notice that, in the case in which x is not a local variable, but a formal parameter of f, this assumption holds trivially (we would get $y_1 = x$). The sharing analysis currently implemented in the compiler (and described in [98]) satisfies all these assumptions.

3.6.4 Preservation of closedness

The last step is to put together the definitions and results presented in last sections in order to prove the correctness of the type system. The essential fact we aim to prove is the following:

> *If we can derive* $E \vdash h, k, e \Downarrow^* h', k, v$ *and* e *is well-typed, then we can derive* $E \vdash h, k, e \Downarrow h'', k, v$ *for some* $h'' \subseteq h'$.

In operational terms, this means that the execution of the program cannot fail because of cell destruction. However, we have to impose some constraints on our initial environment E and heap h under which the $\Downarrow^*$-evaluation is done. Otherwise the claim above might not hold, for instance, when some of the variables occurring free in e contains a dangling pointer in our initial state. In particular, we demand that the the live part of the heap at the beginning of the evaluation is closed, that is, *closed*$(h, E, fv(e))$.

Besides this, some additional constraints are needed on the parameters of the function to which the expression e belongs. As it was explained in last section, the sharing analysis is not required to capture all the sharing that takes place at runtime, but only the sharing that takes place under certain circumstances (namely, when a variable points to a recursive descendant of a condemned one). The same happens with the internal sharing analysis: it is not required to report the internal sharing occurring in function parameters. The disjointness property shown below specifies these conditions.

Definition 3.40 (Disjointness property)**.** A value environment E, a heap h, and a typing environment Γ_0 satisfy the disjointness property (written $P_{disj}(E, h, \Gamma_0)$) iff

1. Γ_0 does not contain variables mapped to in-danger types.
2. For every pair of distinct variables $y_0, y_1 \in \text{dom } \Gamma_0$ such that $\Gamma_0(y_0) \in$ **CdmType**, it holds that *recReach*$(h, E(y_0)) \cap$ *closure*$(h, E(y_1)) = \emptyset$.
3. For every variable $y_0 \in \text{dom } \Gamma_0$ such that $\Gamma_0(y_0) \in$ **CdmType**, $E(y_0)$ is a tree.

Assume Γ_0 contains the types of the parameters of a function, whose body is typeable under this environment. The first condition follows trivially from the [FUN] rule of the type system, which does not allow parameters having in-danger types. The second condition specifies that the recursive spine of condemned parameters cannot share with any other parameter. The last one prevents the DSs pointed to by condemned parameters from having internal sharing. In the correctness proof below we have to prove that the disjointness property is an invariant which is propagated along the execution.

Example 3.41. Assume a function definition with three parameters y_0, y_1 and y_2. The first two parameters have a safe type, whereas y_2 has a condemned type. Consider the three configurations shown in Figure 3.16. The first configuration does not satisfy the disjointness property, since y_1 points to the recursive closure of a condemned parameter (y_2). The second configuration does not satisfy the disjointness property, as y_2 is a condemned parameter with internal sharing. Finally, the third configuration does satisfy the disjointness property. □

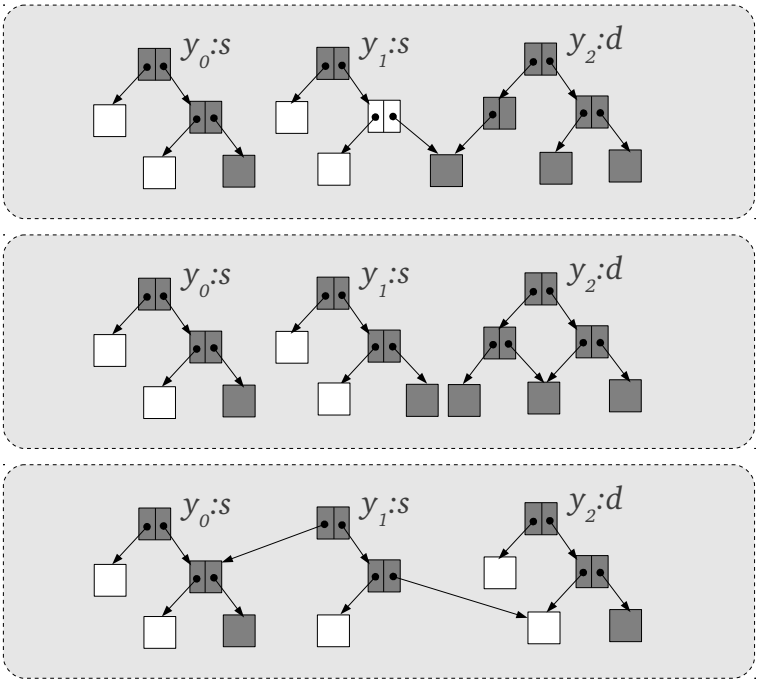


Figure 3.16: Sharing between the parameters of a function. The first two parameters y_0 and y_1 are safe, whereas y_2 is condemned. Gray cells represent the recursive spine of each parameter.

Our next step is to define the invariants that are propagated at the level of expressions, i.e. from a $\Downarrow^*$-judgement to the $\Downarrow^*$-judgements in its derivation.

Definition 3.42. A initial configuration $(\Gamma, E, h, k, L, \Gamma_0)$ is said to be good whenever:

1. $closed(h, E, L)$.
2. The heap h is acyclic. That is, there is no pointer p in h such that $p \rightarrow_h^+ p$.
3. $P_{disj}(E, h, \Gamma_0)$ holds.
4. $build^*(h, E, \Gamma)$ is well-defined.
5. $E(self) = k$, and for every other region variable $r \in \text{dom } E$, $E(r) < k$.
6. If $self \in \text{dom } \Gamma$, then $\Gamma(self) = \rho_{self}$. For every other RTV $r \in \text{dom } \Gamma$, $\Gamma(r) \neq \rho_{self}$.

In the context of the proof, the Γ denotes an environment typing the expression being evaluated, and Γ_0 an environment typing the body of the function to which the expression being evaluated belongs. The first condition has been introduced above. It is easy to see that the second condition is an invariant, since new cells are constructed only in fresh locations of the heap, and the language does not allow in-place update (Lemma 3.29). The disjointness property (third condition) is propagated between function calls, but it is also preserved during the execution of a given function call, since it refers to the parameters of the context function, which remain constant as this call is evaluated. The fourth condition refers to the consistency of the initial configuration. The fact that consistency is an invariant preserved during evaluation has already been proved in Theorem 3.26. The last two conditions state that *self* is the only region variable of type ρ_{self}, and the only region variable pointing to the topmost region of the heap. These conditions are aimed at proving that the topmost region deallocation does not generate dangling pointers. The propagation of the latter property is a consequence of Proposition 2.15.

The following definition states some properties that hold if the evaluation starts from a good configuration.

Definition 3.43. A final configuration (s, v, h) is said to be good whenever $build(h, v, s)$ is well-defined and $closed(h, v)$ holds.

The correctness theorem of the type system shows that the evaluation of an expression transforms good initial configurations into good final configurations. It also proves that $\Downarrow^*$-evaluation of well-typed expressions implies $\Downarrow$-evaluation. Before proving that theorem, we need the following technical result:

Lemma 3.44. *Assume a judgement $\Gamma_0 + [x : \tau] \vdash e_0 : s_0$. For every judgement $\Gamma \vdash e : s$ occurring in its derivation, if $x \in dom\,\Gamma$ then $\Gamma(x) \leq \tau$.*

Proof. By induction on the depth of the $\Gamma \vdash e : s$ w.r.t. the judgement $\Gamma_0 + [x : \tau] \vdash e_0 : s_0$. All cases are straightforward. □

Basically, this result states that when we move from the leafs of the typing derivation to its root, the variables in the typing environment can become weaker, but not stronger. Now we can prove the correctness theorem.

Theorem 3.45 (Correctness of type system)**.** *For every E, h, h', k, e, v, Γ_0, Γ, s_0, s, L of their respective types, and every function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$, if $\Gamma_0 \vdash e_f : s_0$ and e is a sub-expression of e_f such that:*

1. $E \vdash h, k, e \Downarrow^* h', k, v$.
2. $\Gamma \vdash e : s$ *belongs to the derivation of* $\Gamma_0 \vdash e_f : s_0$.
3. $(\Gamma, E, h, k, L, \Gamma_0)$ *is good, being* $L = fv(e)$.

it holds that:

1. *There exists a heap $h'' \subseteq h'$ such that $E \vdash h, k, e \Downarrow h'', k, v$.*
2. *For every pointer $p \in dom\,h \backslash dom\,h''$, and every variable $z \in dom\,E$ such that $E(z) \rightarrow_h^* p$ we obtain $\Gamma(z) \in$* **UnsafeType**.
3. *For every pointer $p \in dom\,h \backslash dom\,h''$, and every variable $z \in dom\,E$ such that $E(z) \rightarrow_h^* p$, but $E(z) \not\twoheadrightarrow_h^* p$ we get $\Gamma(z) \in$* **DgrType**.
4. *(s, v, h'') is good.*

Remark. Before getting into the details of the proof, let us briefly explain the meaning of the second and third conclusions of the theorem. Both are related with the intended semantics of safe, condemned, and in-danger types that was explained in Section 3.3. The second condition states that, if part of the closure of a variable disappears from the heap during the evaluation of e, that variable must appear in Γ with an unsafe type. The third condition is more specific: if part of the closure of a variable *beyond its recursive spine* is destroyed during the evaluation of e, that variable must appear in the environment Γ with an in-danger type.

Proof. By induction on the size of the $\Downarrow^*$ derivation. We distinguish cases on the structure of expression e. In each subcase, we can assume the existence of a Γ' with $\mathrm{dom}\,\Gamma' \subseteq \mathrm{dom}\,\Gamma$, such that $\Gamma' \vdash e : s$ and the last rule applied in this derivation is neither [EXT] nor [WEAK]. We can easily prove that, for every $x \in \mathrm{dom}\,\Gamma'$, if $\Gamma'(x)$ is unsafe, so is $\Gamma(x)$. Moreover, if $\Gamma'(x)$ is in-danger, so is $\Gamma(x)$.

One of the key points of the proof is the fact that the goodness property is propagated through the evaluation tree. When proving this we shall leave out the proof of h being acyclic during the evaluation of e. This is a general property of the language. The judgement $P_{disj}(E, h, \Gamma_0)$ also propagates through the evaluation in the current context function, since the closure of the parameters remains unchanged (see Lemma 3.29). Hence we omit the proof of the P_{disj} propagation, except in the case of function application, where a change of context takes place. The propagation of consistency is also a consequence of Theorem 3.26, and we shall not reproduce its proof again. The fact that *self* is the only region pointing to topmost region in the heap is propagated by Proposition 2.15. The last condition on the definition of good configuration is propagated because the type of region variables remain constant through the typing derivation.

To sum up, for proving the propagation of the goodness property we shall concentrate on the closedness property (and the disjointness property, when calling a function) since the remaining ones either are trivial, or have been proved before. Let us distinguish cases on the expression being evaluated:

- **Case** $e \equiv c$

  Trivial, since a literal is closed by convention and $\mathrm{dom}\, h \backslash \mathrm{dom}\, h' = \emptyset$.

- **Case** $e \equiv x$

  We get $L = \{x\}$ and, by hypothesis, $closed(h, E(x))$, which is equivalent to $closed(h', v)$. In this case we also obtain $\mathrm{dom}\, h \backslash \mathrm{dom}\, h' = \emptyset$.

- **Case** $e \equiv x\,@\,r$

  From the definition of *copy* function it follows that whenever $(h', p) = copy(h, E(x), j)$, p is closed in h' if and only if $E(x)$ is closed in h. But $closed(h, E(x))$ holds by hypothesis, so $closed(h', v)$ also does. We obtain, in this case, $\mathrm{dom}\, h \backslash \mathrm{dom}\, h' = \emptyset$.

- **Case** $e \equiv C\,\overline{a_i}^n\,@\,r$

  Each a_i is closed in h by hypothesis. Since the result v is built from closed components, it is also closed. Hence $closed(h', v)$ holds. Again, $\mathrm{dom}\, h \backslash \mathrm{dom}\, h'$ is empty.

- **Case** $e \equiv g\,\overline{a_i}^n\,@\,\overline{r_j}^m$

  Let us assume that $g\,\overline{y_i}^n\,@\,\overline{r'_j}^m = e_g \in \Sigma$. By the $[App^*]$ rule we get:

$$\underbrace{[\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^m, self \mapsto k+1]}_{E_g} \vdash h, k+1, e_g \Downarrow_f^* h', k+1, v$$

  Moreover, if $\Gamma(g) = \forall \overline{\alpha\rho}.\overline{t_{i,g}}^n \to \overline{\rho_{j,g}}^m \to s_g$, the following derivation follows from the fact that Γ is correct w.r.t. Σ:

$$\underbrace{\Gamma_\omega + [\overline{y_i : t_{i,g}}^n] + [\overline{r'_j : \rho_{j,g}}^m] + [self : \rho_{self}]}_{\Gamma_g} \vdash e_g : s_g \qquad \text{for some } \Gamma_\omega \subseteq \Gamma \tag{3.9}$$

  By the [APP] rule we know that the type of the application is a concrete instance of $\Gamma(g)$ and there exists a type substitution θ such that:

$$\overline{\theta(t_{i,g})}^n \to \overline{\theta(\rho_{j,g})}^m \to \theta(s_g) = \overline{\Gamma(a_i)}^n \to \overline{\Gamma(\rho_j)}^m \to s$$

Since ρ_{self} does not occur in $\Gamma(g)$, we can safely assume that $\theta(\rho_{self}) = \rho_{self}$. By applying the substitution lemma (Lemma 3.9) to (3.9) we obtain:

$$\underbrace{\theta(\Gamma_\omega) + [\overline{y_i : \Gamma(a_i)}^{\,n}] + [\overline{r'_j : \Gamma(\rho_j)}^{\,m}] + [self : \rho_{self}]}_{\theta(\Gamma_g)} \vdash e_g : s \tag{3.10}$$

From the induction hypothesis we obtain the following implication:

$$\left.\begin{array}{l} E_g \vdash h, k+1, e_g \Downarrow^* h', k+1, v \\ \theta(\Gamma_g) \vdash e_g : s \\ (\theta(\Gamma_g), E_g, h, k+1, fv(e_g), \theta(\Gamma_g)) \text{ good} \end{array}\right\} \Rightarrow \left\{\begin{array}{l} E_g \vdash h, k+1, e_g \Downarrow h'', k+1, v \quad \text{for some } h'' \subseteq h' \\ \forall p \in \operatorname{dom} h \backslash \operatorname{dom} h''. \forall z \in \operatorname{dom} E_g. E_g(z) \rightarrow_h^* p \\ \qquad \Rightarrow \theta(\Gamma_g)(z) \in \textbf{UnsafeType} \\ \forall p \in \operatorname{dom} h \backslash \operatorname{dom} h''. \forall z \in \operatorname{dom} E_g. E_g(z) \rightarrow_h^* p \\ \qquad \wedge E_g(z) \not\twoheadrightarrow_h^* p \\ \qquad \Rightarrow \theta(\Gamma_g)(z) \in \textbf{DgrType} \\ (v, h'') \text{ good} \end{array}\right. \tag{3.11}$$

Let us prove the antecedent of this implication. The first fact follows from the assumption. The second one follows from (3.10). This derivation takes part in a different context definition $g\ \overline{y_i}\ @\ \overline{r'_j} = e_g$. Now we prove that the configuration $(\theta(\Gamma_g), E_g, h, k+1, fv(e_g), \theta(\Gamma_g))$ is good. The fact $closed(h, E_g, fv(e_g))$ follows from $closed(h, E, \{\overline{a_i}^n\})$, which holds by assumption. The fact $P_{disj}(E_g, h, \theta(\Gamma_g))$ can be proven by contradiction. It is trivial that the first property of $P_{disj}(E_g, h, \theta(\Gamma_g))$ holds, so let us firstly assume that the second one does not. That is, there exist distinct parameters $y_i, y_j \in \operatorname{dom} \theta(\Gamma_g)$ such that $\theta(\Gamma_g)(y_i) \in \textbf{CdmType}$ and a pointer p such that $E_g(y_i) \twoheadrightarrow_h^* p$ and $E_g(y_j) \rightarrow_h^* p$. We unfold the definitions of $\theta(\Gamma_g)$ and E_g in order to obtain:

$$\Gamma(a_i) \in \textbf{CdmType} \qquad E(a_i) \twoheadrightarrow_h^* p \qquad E(a_j) \rightarrow_h^* p$$

If $a_i = a_j$ then a_j has also a condemned type in Γ, and the result of the operator $\oplus$ is not defined, contradicting the fact that the expression is well-typed. If $a_i \neq a_j$ then, by Assumption 3.38, we have three possibilities: (SHR), (SHP), or $(CONF)$. If we are able to prove neither (SHP) nor $(CONF)$ are possible, we will get $a_j \in sharerec(a_i, e)$, which implies $a_j \in \operatorname{dom} \Gamma_R$ and the disjoint sum of Γ_R with $\bigoplus_{i=1}^n [a_i : t_i]$ in the [APP] rule would be undefined. So let us prove that assuming (SHP) or $(CONF)$ leads to a contradiction: in any of these cases, there exists a parameter y_0 of the caller function f such that $E(y_0) \twoheadrightarrow_h^* p$. By Assumption 3.39, there exists another parameter y_2 such that $E(y_2) \rightarrow_h^* p$ and $y_2 \in sharerec(a_i, e)$.

- If $y_2 = a_i$ then $\Gamma(y_2) = \Gamma(a_i) \in \textbf{CdmType}$, which implies, by Lemma 3.44, that $\Gamma_0(y_2)$ can be condemned or in-danger. If the latter is in-danger we obtain a contradiction, since Γ_0 does not have in-danger parameters. If it is condemned, we distinguish cases depending on whether (SHP) or $(CONF)$ holds:

  (SHP) In this case there exists a parameter y_1, different from y_0 (but not necessarily different from y_2) such that $E(y_1) \rightarrow_h^* p$. This contradicts property (2) of $P_{disj}(E, h, \Gamma_0)$, as $E(y_2) = E(a_i) \twoheadrightarrow_h^* p$, and either y_1 or y_0 is a parameter different from y_2 pointing to its recursive closure.

  $(CONF)$ If $y_2 = y_0$ then $\Gamma(y_0) = \Gamma(y_2) \in \textbf{CdmType}$, which contradicts property (3) of $P_{disj}(E, h, \Gamma_0)$. If $y_2 \neq y_0$, then we get two different parameters such that $E(y_0) \twoheadrightarrow_h^* p$,

$E(y_2) \twoheadrightarrow_h^* p$, and $\Gamma(y_2)$ is condemned. This contradicts $P_{disj}(E, h, \Gamma_0)$ (property (2)).

– If $y_2 \neq a_i$ then $\Gamma(y_2) \in$ **DgrType**, as y_2 belongs to the Γ_R environment of the [APP] rule. This implies that $\Gamma_0(y_2) \in$ **DgrType** as well, which, again, leads to a contradiction.

Now we assume that property (3) of $P_{disj}(E_g, h, \theta(\Gamma_g))$ does not hold. That is, there exists some parameter $y_i \in \text{dom } \theta(\Gamma_g)$ with a condemned type and a pointer p in its recursive closure such that $E_g(y_i) \rightrightarrows_h^* p$. Equivalently, there exists some argument $a_i \in \text{dom } \Gamma$ with a condemned type and a pointer p in its recursive closure such that $E(a_i) \rightrightarrows_h^* p$. By Assumption 3.37, we get:

$$\begin{array}{ll} (TREE) & \neg isTree(a_i), \text{ or} \\ (CONF) & \exists y_0. E(y_0) \rightrightarrows_h^* p, \text{ where } y_0 \text{ is a parameter of } f \end{array}$$

$(TREE)$ clearly contradicts the [APP] rule, so let us assume $(CONF)$. Again, by Assumption 3.39, there exists a parameter y_2 of f such that $E(y_2) \rightarrow_h^* p$ and $y_2 \in sharerec(a_i, e)$. We distinguish cases as above:

– If $y_2 = a_i$, then $\Gamma(y_2) = \Gamma(a_i) \in$ **CdmType**, which implies that $\Gamma_0(y_2)$ is condemned or in-danger. The latter case leads to a contradiction with property 1 of $P_{disj}(E, h, \Gamma_0)$. If $\Gamma_0(y_2)$ is condemned and $y_2 \neq y_0$ we get a contradiction with property 2 of $P_{disj}(E, h, \Gamma_0)$ as $E(y_2) = E(a_i) \twoheadrightarrow_h^* p$ and $E(y_0) \twoheadrightarrow_h^* p$, where y_2 has a condemned type. If $\Gamma_0(y_2)$ is condemned and $y_2 = y_0$, then $E(y_2) = E(y_0) \rightrightarrows_h^* p$, which contradicts (3).

– If $y_2 \neq a_i$, then $\Gamma(y_2)$ is in-danger (since y_2 occurs within Γ_R of the [APP] rule), and so is $\Gamma_0(y_2)$. This contradicts (1) of $P_{disj}(E, h, \Gamma_0)$.

Since we have proven the left-hand side of the implication in (3.11) we get the facts of the right-hand side. The first conclusion of the theorem follow trivially the first fact:

$$E \vdash h, k, g\, \overline{a_i}^n \mathbin{@} \overline{r_j}^m \Downarrow h''|_k, k, v$$

For proving the second and third conclusions, let us assume a variable $z \in \text{dom } E$ such that $E(z) \rightarrow_h^* p$ and $p \in \text{dom } h \backslash \text{dom } h''|_k$. Since none of the cells in h belongs to the $k+1$-th region, $p \in \text{dom } h \backslash \text{dom } h''$. By Lemma 3.30 p belongs to $live(h, E, \{\overline{a_i}\})$, so there exists an argument a_i such that $E(a_i) = E_g(y_i) \rightarrow_h^* p$. Since y_i has an unsafe type in $\theta(\Gamma_g)$ (by (3.11)), a_i has an unsafe type in Γ. But, since a_i is an argument of the function, that type cannot be in-danger, so it is condemned. Let us distinguish cases:

– If p is a recursive descendant of a_i we have the same situation as before: z points to a recursive child of a condemned argument. This implies (SHR), (SHP) or $(CONF)$. Again, the only possibility that does not lead to a contradiction is the first one, so we can safely assume it to hold. Since $z \in sharerec(a_i, e)$ holds, z belongs to the Γ_R environment in the [APP] rule, so $\Gamma(z) \in$ **DgrType**, and we are done.

– The case in which p is not a recursive descendant of a_i leads to a contradiction: it would imply that $E_g(y_i) \rightarrow_h^* p$, but $E_g(y_i) \not\twoheadrightarrow_h^* p$, from which it follows that $\theta(\Gamma_g)(y_i) \in$ **DgrType** by (3.11), but $\theta(\Gamma_g)$ cannot have parameters with an in-danger type.

For proving the fourth conclusion we have to prove that $closed(h'', v)$ implies $closed(h''|_k, v)$. That is, $closure(h'', v)$ does not contain cells in the $k+1$-th region of the heap. By Theorem 3.26,

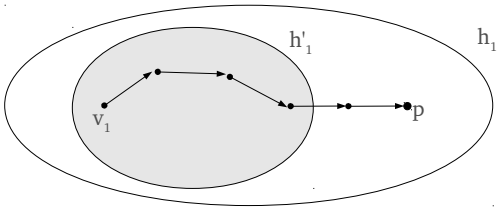


Figure 3.17: If p is reachable from v_1 and $p \notin \operatorname{dom} h_1'$, then v_1 is not closed in h_1'.

$build(h'', v, \theta(s))$ is well-defined and consistent with $build^*(h, E_g, \theta(\Gamma_g))$. If there existed a cell in $closure(h'', v)$ in the topmost region of the heap, we would obtain the binding $[\rho_{self} \mapsto k+1] \in build(h'', v, \theta(s))$, since *self* is the only region variable such that $E_g(self) = k+1$ and $\theta(\Gamma_g)(self) = \rho_{self}$. But this contradicts the fact that $\theta(s)$ does not contain the ρ_{self} RTV, a constraint imposed by the [FUN] rule when applied to the definition of g. Hence, $closed(h''|_k, v)$ holds.

- **Case** $e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2$

  Assume the following execution for e_1

  $$E \vdash h, k, e_1 \Downarrow_f^* h_1, k, v_1$$

  and that $\Gamma' = \Gamma_1 \sqcup \Gamma_2$ where Γ_1 and $\Gamma_2 + [x_1 : \tau_1]$ are typing environments of the corresponding expressions e_1 and e_2. The facts $closed(h, E, fv(e_1))$ and $P_{disj}(E, h, \Gamma_0)$ follow from the assumptions of the theorem. So, the initial configuration of e_1 is good, and we can apply the induction hypothesis in order to get:

  $$E \vdash h, k, e_1 \Downarrow h_1', k, v_1 \qquad \text{for some } h_1' \subseteq h_1 \tag{3.12}$$

  $$closed(h_1', v_1)$$

  $$\forall p \in \operatorname{dom} h \backslash \operatorname{dom} h_1'. \forall z \in \operatorname{dom} E. E(z) \rightarrow_h^* p \Rightarrow \Gamma_1(z) \in \textbf{UnsafeType} \tag{3.13}$$

  $$\forall p \in \operatorname{dom} h \backslash \operatorname{dom} h_1'. \forall z \in \operatorname{dom} E. E(z) \rightarrow_h^* p \wedge E(z) \not\rightarrow_h^* p \Rightarrow \Gamma_1(z) \in \textbf{DgrType} \tag{3.14}$$

  With regard to the main expression e_2, we assume the following $\Downarrow^*$-execution:

  $$\underbrace{E \uplus [x_1 \mapsto v_1]}_{E_1} \vdash h_1, k, e_2 \Downarrow_f^* h', k, v \tag{3.15}$$

  Firstly we should prove that we can safely substitute h_1' for h_1 in this judgement. However, this task becomes easier if we prove $closed(h_1', E_1, fv(e_2))$ first. Let $z \in fv(e_2)$. We already know that $closed(h_1', E_1(x_1))$, so let us assume that $z \neq x_1$. If $E_1(z)$ is not closed in h_1', there must exist a pointer p such that $E_1(z) \rightarrow_{h_1'}^* p$, but $p \notin \operatorname{dom} h_1'$. Notice that we get $E_1(z) \rightarrow_{h_1}^* p$ as well, since $h_1' \subseteq h_1$ and, by Lemma 3.29, $E_1(z) \rightarrow_h^* p$. Let us distinguish cases:

  - $p \notin \operatorname{dom} h$

    Since $z \in fv(e_2)$ and is distinct from x_1, we obtain $z \in fv(e)$. This case leads to a contradiction because of the assumption $closed(h, E, fv(e))$.

  - $p \in \operatorname{dom} h$

    By (3.13) the z variable gets an unsafe type in Γ_1, so it cannot occur free in e_2 by the side condition of [LET]. This case also leads to a contradiction.

Thus $closed(h_1', E_1, fv(e_2))$ holds. Now we prove that $p \notin live(h_1, E_1, fv(e_2))$ for every pointer $p \in \mathrm{dom}\, h_1 \backslash \mathrm{dom}\, h_1'$. Again, we prove this by contradiction: if $p \notin \mathrm{dom}\, h_1'$ but $p \in live(h_1, E_1, fv(e_2))$, then $closed(h_1', E_1, fv(e_2))$ would not hold, but we just have proved that it does hold, so $p \notin live(h_1, E_1, fv(e_2))$. Thus we can substitute h_1' for h_1 in (3.15) (by Lemma 3.31), and we get:

$$E_1 \vdash h_1', k, e_2 \Downarrow_f^* h_2', k, v \qquad \text{for some } h_2' \subseteq h' \tag{3.16}$$

It is easy to prove that the configuration $(\Gamma_2, E_1, h_1', k, fv(e_2), \Gamma_0)$ is good, as we already know that $closed(h_1', E_1, fv(e_2))$ holds. Thus we can apply the induction hypothesis to (3.16) in order to obtain,

$$E_1 \vdash h_1', k, e_2 \Downarrow h'', k, v \qquad \text{for some } h'' \subseteq h_2' \subseteq h' \tag{3.17}$$

$$closed(h'', v)$$

It is straightforward to see that $(\Gamma_2, E_1, h_1, k, fv(e_2), \Gamma_0)$ is good, so the following can be obtained by instantiating the induction hypothesis and by taking h_1 as the initial heap, instead of h_1':

$$\forall p \in \mathrm{dom}\, h_1 \backslash \mathrm{dom}\, h''. \forall z \in \mathrm{dom}\, E_1. E_1(z) \rightarrow_{h_1}^* p \Rightarrow \Gamma_2(z) \in \mathbf{UnsafeType} \tag{3.18}$$

$$\forall p \in \mathrm{dom}\, h_1 \backslash \mathrm{dom}\, h''. \forall z \in \mathrm{dom}\, E_1. E_1(z) \rightarrow_{h_1}^* p \wedge E_1(z) \not\twoheadrightarrow_{h_1}^* p \Rightarrow \Gamma_2(z) \in \mathbf{DgrType} \tag{3.19}$$

Thus we can derive, from (3.12) and (3.17), $E \vdash h, k, e \Downarrow h'', k, v$. The final step is to prove that for every pointer $p \in \mathrm{dom}\, h \backslash \mathrm{dom}\, h''$ and every variable $z \in \mathrm{dom}\, E$ such that $E(z) \rightarrow_h^* p$ has an unsafe type in $\Gamma_1 \sqcup \Gamma_2$. Moreover, if $E(z) \not\twoheadrightarrow_h^* p$, then z has an in-danger type in that environment.

- $p \notin \mathrm{dom}\, h_1'$

  Then p has been removed from the heap during the evaluation of e_1. By (3.13) we get $\Gamma_1(z) \in$ **UnsafeType** and hence $(\Gamma_1 \sqcup \Gamma_2)(z) \in$ **UnsafeType**. In the case that $E(z) \not\twoheadrightarrow_h^* p$ we get, more specifically, $\Gamma_1(z) \in$ **DgrType** by (3.14) and, hence, $(\Gamma_1 \sqcup \Gamma_2)(z) \in$ **DgrType**.

- $p \in \mathrm{dom}\, h_1'$

  In this case p has been removed from the heap during the evaluation of e_2. Since $h_1' \subseteq h_1$, we know that $p \in \mathrm{dom}\, h_1$. Besides this, the $\Downarrow^*$-derivation does not remove pointers from the heap, so $h \subseteq h_1$. This means that $E(z) \rightarrow_h^* p$ implies $E(z) \rightarrow_{h_1}^* p$. We can apply (3.18) and obtain $\Gamma_2(z) \in$ **UnsafeType**, which implies $(\Gamma_1 \sqcup \Gamma_2)(z) \in$ **UnsafeType**. Similarly, $E(z) \not\twoheadrightarrow_h^* p$ implies $E(z) \not\twoheadrightarrow_{h_1}^* p$, so by applying (3.19) we get $\Gamma_2 \in$ **DgrType**, from which it follows that $(\Gamma_1 \sqcup \Gamma_2)(z) \in$ **DgrType**.

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\, \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

  We assume the execution of the r-th branch:

$$\underbrace{E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}]}_{E_r} \vdash h, k, e_r \Downarrow_f^* h', k, v$$

From the assumption $closed(h, E, fv(e))$ it follows that $closed(h, E_r, fv(e_r))$. If we denote by Γ_r the typing environment of e_r, the configuration $(\Gamma_r, E_r, h, k, fv(e_r), \Gamma_0)$ is good. By induction hypothesis we get:

$$E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow h'', k, v \qquad \text{for some } h'' \subseteq h'$$

$$closed(h'', v)$$

$$\forall p \in \mathrm{dom}\, h \backslash \mathrm{dom}\, h'', \forall z \in \mathrm{dom}\ E_r . E_r(z) \rightarrow_h^* p \Rightarrow \Gamma_r(z) \in \textbf{UnsafeType} \tag{3.20}$$

$$\forall p \in \mathrm{dom}\, h \backslash \mathrm{dom}\, h'', \forall z \in \mathrm{dom}\ E_r . E_r(z) \rightarrow_h^* p \wedge E_r(z) \not\twoheadrightarrow_h^* \Rightarrow \Gamma_r(z) \in \textbf{DgrType} \tag{3.21}$$

From the first fact we can derive $E \vdash h, k, e \Downarrow h'', k, v$. Now we prove that for every variable $z \in \mathrm{dom}\, E$ pointing to a location $p \in \mathrm{dom}\, h \backslash \mathrm{dom}\, h''$ has an unsafe type. Since $E(z) = E_r(z) \rightarrow_h^* p$, by (3.20) we get $\Gamma_r(z) \in \textbf{UnsafeType}$. Since z cannot be any of the pattern variables, we get $z \in \mathrm{dom}\, \Gamma$ and the required result follows trivially. In a similar way we can apply (3.21) in order to obtain $\Gamma(z) \in \textbf{DgrType}$ in the case where $E(z) \not\twoheadrightarrow_h^* p$.

- **Case** $e \equiv$ **case!** x **of** $\overline{C_i\, \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n}$

  Assume the r-th branch is executed:

$$\underbrace{E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}]}_{E_r} \vdash \underbrace{h \uplus [p \mapsto w]}_{h^+}, k, e_r \Downarrow_f^* h', k, v$$

  By the [CASE!] rule, x has an condemned type in Γ', and hence an unsafe type in Γ.

  Let us prove that $E(x) = p \notin live(h^+, E_r, fv(e_r))$ by contradiction: if $p \in live(h^+, E_r, fv(e_r))$, there exists a variable $z \in fv(e_r)$ such that $E_r(z) \rightarrow_{h^+}^* p$. The z variable cannot be one of the pattern variables $\overline{x_{rj}}^{n_r}$, since otherwise we would have $E(x) = p \rightarrow_{h^+} E_r(z) \rightarrow_{h^+}^* p$, contradicting the fact that the initial heap is acyclic. Therefore, z is not a pattern variable, so $E_r(z) = E(z)$. Obviously, $E(x) \twoheadrightarrow_{h^+}^* p$. By Assumption 3.38 we have three possibilities: (SHR), (SHP), or $(CONF)$. We can prove that neither (SHP) nor $(CONF)$ are possible by following a similar reasoning as in the end of the case $e \equiv f\, \overline{a_i}^n\, @\, \overline{r_j}^m$ (we only have to substitute x for a_i). Thus, the only possibility is (SHR), that is, $z \in sharerec(x, e)$. Then z belongs to the Γ_R environment of the [CASE!] rule, and $z \notin fv(e_r)$, leading to a contradiction.

  Since $p \notin live(h^+, E_r, fv(e_r))$, by Lemma 3.31 we can safely discard the binding in the evaluation of e_r in order to get a $\Downarrow^*$-derivation of the same size as the previous one:

$$E_r \vdash h, k, e_r \Downarrow_f^* h' \backslash p, k, v$$

  Moreover, since the initial configuration is good and $p \notin live(h^+, E_r, fv(e_r))$, $closed(h, E_r, fv(e_r))$ holds and hence the configuration $(\Gamma_r, E_r, h, k, fv(e_r), \Gamma_0)$ is good, being Γ_r the typing environment of the e_r. We can apply the induction hypothesis in order to get:

$$E_r \vdash h, k, e_r \Downarrow h'', k, v \qquad \text{for some } h'' \subseteq h' \backslash p \subseteq h'$$

$$closed(h'', v)$$

$$\forall q \in \mathrm{dom}\, h \backslash \mathrm{dom}\, h'' . \forall z \in \mathrm{dom}\ E_r .\ E_r(z) \rightarrow_h^* q \Rightarrow \Gamma_r(z) \in \textbf{UnsafeType} \tag{3.22}$$

$$\forall q \in \mathrm{dom}\, h \backslash \mathrm{dom}\, h'' . \forall z \in \mathrm{dom}\ E_r .\ E_r(z) \rightarrow_h^* 1 \wedge E_r(z) \not\twoheadrightarrow_h^* q \Rightarrow \Gamma_r(z) \in \textbf{DgrType} \tag{3.23}$$

  Thus (v, h'') is good. Now we prove that for every pointer $q \in \mathrm{dom}\, h^+ \backslash \mathrm{dom}\, h''$ and every variable $z \in \mathrm{dom}\, E$ such that $E(z) \rightarrow_{h^+}^* q$, then z occurs in Γ with an unsafe type. If $q \neq p$ (i.e., q is not the pointer being destroyed by this **case**!) we further distinguish cases:

– If $E(z) \to^*_{h^+} p$, we have already proved that $z \in \mathit{sharerec}(x, e)$, so z must occur in Γ_R with an in-danger type.

– If $E(z) \not\to^*_{h^+} p$, then p cannot be in the middle of the path between $E(z)$ and q in h^+, so $E(z) \to^*_h q$, and we proceed as in the non-destructive case.

If $q = p$, let z such that $E(z) \to^*_{h^+} p$. If z is the discriminant of the **case**!, then it has a condemned type. If $z \neq x$ then we have already shown that $z \in \mathit{sharerec}(x, e)$. Hence z gets an in-danger type, as it belongs to the Γ_R environment. Finally we prove, under the same conditions, that $\Gamma(z) \in \mathbf{DgrType}$ whenever $E(z) \to^*_{h^+} q$, but $E(z) \not\twoheadrightarrow^*_{h^+} q$. We distinguish cases:

$q = p$ The z variable cannot be the **case**! discriminant, since $E(x) \twoheadrightarrow_{h^+} p$ and we are assuming that $E(z) \not\twoheadrightarrow_{h^+} p$. So, we safely assume that $z \neq x$. We have already shown that $z \in \mathit{sharerec}(x, e)$, which forces z to appear with an in-danger type in the Γ_R environment of [CASE!]. Hence $\Gamma(z) \in \mathbf{DgrType}$.

$q \neq p$

– If z is not the **case**! discriminant x, then $\Gamma_r(z) \leq \Gamma(z)$, since z cannot be a pattern variable. Again, we have to distinguish cases depending on whether $E(z)$ points to p in h^+. If it does, then $z \in \mathit{sharerec}(x, e)$, as we have proved before, and belongs to Γ_R, so it has an in-danger type. If it does not, we get $E(z) \to^*_h q$, and we just apply (3.23) to get $\Gamma_r(z) \in \mathbf{DgrType}$, which implies $\Gamma(z) \in \mathbf{DgrType}$.

– If z is the **case**! discriminant x, we cannot ensure that $\Gamma_r(x) \leq \Gamma(x)$, since the [CASE!] rule replaces the binding of the x variable in Γ_r with the binding $[x : T\ !@\,\rho]$. However, we shall see that this case leads to a contradiction. Since $q \neq p$, we know that $E(x)$ reaches q through some of its pattern variables, that is, $E(x) \to_{h^+} E_r(x_{ri}) \to^*_{h^+} q$ for some $i \in \{1..n_r\}$. Since $q \neq p$, this is equivalent to $E(x) \to_h E_r(x_{ri}) \to^*_h q$. We distinguish cases:

  * If $i \in \mathit{RecPos}(C)$ we get $E(x) \twoheadrightarrow_{h^+} E_r(x_{ri}) \to^*_h q$. Since we are assuming that $E(z) \not\twoheadrightarrow^*_{h^+} q$ we know that $E_r(x_{ri}) \not\twoheadrightarrow^*_{h^+} q$. By (3.23), $\Gamma_r(x_{ri}) \in \mathbf{DgrType}$, which contradicts the *inh* predicate in [CASE!].
  * If $i \notin \mathit{RecPos}(C)$ we get $E_r(x_{ri}) \to^*_h q$, which implies, by (3.22), $\Gamma_r(x_{ri}) \in \mathbf{UnsafeType}$ which, again, contradicts the *inh* assumptions in [CASE!].

□

This theorem can be instantiated to show that well-typed programs do not access dangling pointers.

Corollary 3.46. *Assume a Core-Safe program prog. Let Σ be an environment containing its function definitions and e the main expression of the program. If the following judgement is derivable:*

$$[\mathit{self} \mapsto 0] \vdash [\,], 0, e \Downarrow^* h', k, v$$

and $\Gamma + [\mathit{self} : \rho_{\mathit{self}}] \vdash e : s$ holds for some type s and some type environment Γ which only contains functional type schemes and is correct w.r.t. Σ, then we obtain:

$$[\mathit{self} \mapsto 0] \vdash [\,], 0, e \Downarrow h'', k, v$$

for some $h'' \subseteq h$.

Proof. If we denote by Γ_0 the environment $\Gamma + [self : \rho_{self}]$, we have to show that the initial configuration

$$(\Gamma_0, [self \mapsto 0], [\,], 0, fv(e), \Gamma_0)$$

is good. Obviously, the empty heap is closed and acyclic. The disjointness property holds vacuously, since there are not function parameters in the main expression. Moreover, $build^*([\,], [self \mapsto 0], \Gamma_0) = [\rho_{self} \mapsto 0]$, which is well-defined. Finally, the only region variable in the runtime environment pointing to the topmost region (0) is *self*, and this is the only region variable of type ρ_{self} in Γ_0, since Γ only contain functional type schemes. Therefore, the initial configuration is good, and we can apply Theorem 3.45 to get the desired result. □

3.7 Towards the type inference of function definitions

The type system shown in this chapter provides us a way to establish the correctness of a *Core-Safe* program, subject to the existence of a typing derivation for it. However, it does not give any clue on how to build these typing derivations. Moreover, it assumes that the program to be typed is annotated with region variables, even when these variables are not specified by the programmer.

The aim of the next two chapters is to devise a mechanical way to annotate a *Core-Safe* program with region variables, and to construct its corresponding typing derivation. This is achieved by two algorithms:

- A region inference algorithm which, given a region-free *Core-Safe* program, annotates its abstract syntax tree with region variables. This algorithm is only concerned with Hindley-Milner types and region type variables, regardless of whether these types are safe, in-danger or condemned.
- A destruction safety inference algorithm, relying on an ancillary sharing analysis and concerning exclusively the destructive pattern matching feature of the language, without regard to regions. It determines whether the type of a given variable should be safe, in-danger or condemned.

None of these algorithms can, on its own, guarantee the existence of a typing derivation. We can only build a correct typing derivation from the results of *both* algorithms. In order to prove the correctness of these algorithms we would need to consider them jointly, as a single algorithm. But, in this case, the correctness proof and the algorithm itself would become unwieldy.

Instead of putting the two algorithms together, we split the type system in two simpler type systems in such a way that, if a program is typable according to these two simpler systems, then so it is according to the initial one. As a consequence, we obtain:

- A type system dealing exclusively with regions, and leaving out the in-danger or condemned nature of types. It is defined as a set of judgements $\Gamma_R \vdash_{Reg} e : s$.
- A type system dealing exclusively with safe, in-danger or condemned marks, without regard to regions. It is defined as a set of judgements $\Gamma_D \vdash_{Dst} e : \mathtt{s}$.

The environments occurring in the $\vdash_{Reg}$ derivations are called *region environments*. They map variables and function and constructor symbols to safe types and safe type signatures respectively. On the other hand, the environments in the $\vdash_{Dst}$ derivations are called *mark environments*. A mark environment maps type variables to marks in the set $\mathbf{Mark} = \{\mathtt{s}, \mathtt{r}, \mathtt{d}\}$ and function and constructor names to mark signatures. A mark signature is an expression of the form $\overline{m_i}^n \to \mathtt{s}$, in which each m_i stand for the

$$\frac{}{\Gamma \vdash_{Reg} c : B}\ [LIT_{Reg}]$$

$$\frac{}{\Gamma + [x : s] \vdash_{Reg} x : s}\ [VAR_{Reg}]$$

$$\frac{}{\Gamma + [x : T@\rho', r : \rho] \vdash_{Reg} x@r : T@\rho}\ [COPY_{Reg}]$$

$$\frac{\Gamma \vdash_{Reg} e_1 : s_1 \qquad \Gamma + [x_1 : s_1] \vdash_{Reg} e : s}{\Gamma \vdash_{Reg} \mathbf{let}\, x_1 = e_1\, \mathbf{in}\, e_2 : s}\ [LET_{Reg}]$$

$$\frac{\overline{s_i}^n \to \overline{\rho_j}^m \to s \trianglelefteq \sigma \qquad \Gamma \vdash_{Reg} a_i : s_i}{\Gamma + [f : \sigma] + [\overline{r_j : \rho_j}^m] \vdash_{Reg} f\, \overline{a_i}^n\, @\, \overline{r_j}^m : s}\ [APP_{Reg}]$$

$$\frac{\Gamma(C) = \sigma \qquad \overline{s_i}^n \to \rho \to s \trianglelefteq \sigma \qquad \forall i \in \{1..n\}.\, \Gamma \vdash_{Reg} a_i : s_i}{\Gamma + [r : \rho] \vdash_{Reg} C\, \overline{a_i}^n\, @\, r : s}\ [CONS_{Reg}]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.\, \overline{s_{ij}}^{n_i} \to \rho \to s_x \trianglelefteq \Gamma(C_i) \\ \Gamma(x) = s_x \qquad \forall i \in \{1 \ldots n\}.\Gamma + [\overline{x_{ij} : s_{ij}}^{n_i}] \vdash_{Reg} e_i : s\end{array}}{\Gamma \vdash_{Reg} \mathbf{case}(!)\, x\, \mathbf{of}\, \overline{C_i\, \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ [CASE(!)_{Reg}]$$

Figure 3.18: Typing rules for deriving $\vdash_{Reg}$ judgements.

mark of the corresponding function parameter, whereas the s mark at the end stands for the result of the function (which is always safe).

Formally, we have two kinds of judgements:

$\Gamma_R \vdash_{Reg} e : s$ where $\Gamma_R : (\mathbf{Var} \to \mathbf{SafeType}) \times (\mathbf{RegVar} \to \mathbf{RegType}) \times (\mathbf{Fun} \cup \mathbf{Cons} \to \mathbf{SafeFunType})$
$\Gamma_D \vdash_{Dst} e : \mathtt{s}$ where $\Gamma_D : (\mathbf{Var} \to \mathbf{Mark}) \times (\mathbf{Fun} \to \mathbf{MarkSig})$

Figures 3.18 and 3.19 contain the typing rules for the $\vdash_{Reg}$ and $\vdash_{Dst}$ type systems respectively. All these rules are simplified versions of those occurring in Figure 3.6 and hence they will not be explained here again. The $inh_{i,C}$ predicate appearing in $[CASE!_{Dst}]$ rule is defined in Figure 3.20. Again, this predicate is a simplified version of its counterpart in Figure 3.8. The operators defined in Section 3.3 also apply to region and type environments: $+$ forbids disjoint domains, whereas $\oplus$ allows them if they bound common variables to a safe mark. The result of $\Gamma_{D,1} \sqcup \Gamma_{D,2}$ maps common variables in $\Gamma_{D,1}$ and $\Gamma_{D,2}$ to the maximal mark, assuming the following order between marks: $\mathtt{s} \leq \mathtt{d} \leq \mathtt{r}$. This operator demands the functions belonging to the domain of $\Gamma_{D,1}$ and $\Gamma_{D,2}$ to be bound to the same mark signature in both of them.

For proving that the union of the $\vdash_{Reg}$ and $\vdash_{Dst}$ type systems are equivalent to the original $\vdash$ type system, we have to specify how to combine the region environments with the mark environments. This is done by means of an operator $\circ$ defined in Figure 3.21. By abuse of notation, we extend the domain of this operator to safe type schemes and mark signatures as follows:

$$(\overline{s_i}^n \to \overline{\rho_j}^m \to s) \circ (\overline{m_i}^n \to \mathtt{s}) \stackrel{\text{def}}{=} (\overline{s_i \circ m_i}^n \to \overline{\rho_j}^m \to s)$$

If the number of s_i is different from the number of m_i, the result is undefined.

In the following, Γ_R always denotes a region environment and Γ_D denotes a mark environment.

$$\frac{\Gamma' \vdash_{Dst} e : \mathsf{s} \quad \Gamma \supseteq \Gamma'}{\Gamma \vdash_{Dst} e : \mathsf{s}}\ [EXT_{Dst}] \qquad \frac{\Gamma + [x : m_1] \vdash_{Dst} e : \mathsf{s} \quad m_1 \leq m_2}{\Gamma + [x : m_2] \vdash_{Dst} e : \mathsf{s}}\ [WEAK_{Dst}]$$

$$\frac{}{\emptyset \vdash_{Dst} c : \mathsf{s}}\ [LIT_{Dst}] \qquad \frac{}{[x : \mathsf{s}] \vdash_{Dst} x : \mathsf{s}}\ [VAR_{Dst}] \qquad \frac{}{[x : \mathsf{s}] \vdash x\,@\,r : \mathsf{s}}\ [COPY_{Dst}]$$

$$\frac{\begin{array}{c}\Gamma_1 \vdash_{Dst} e_1 : \mathsf{s} \qquad \Gamma_2 + [x_1 : m] \vdash_{Dst} e_2 : \mathsf{s} \\ \forall x \in \mathrm{dom}\,\Gamma_1 . \Gamma_1(x) \in \{\mathsf{r}, \mathsf{d}\} \Rightarrow x \notin fv(e_2)\end{array}}{\Gamma_1 \sqcup \Gamma_2 \vdash_{Dst} \mathbf{let}\, x_1 = e_1\, \mathbf{in}\, e_2 : \mathsf{s}}\ [LET_{Dst}]$$

$$\frac{\begin{array}{c}R = \bigcup_{i=1}^{n} \{sharerec(a_i, f\,\overline{a_i}^n\,@\,\overline{r_j}^m) \backslash \{a_i\} \mid m_i = \mathsf{d}\} \qquad \bigwedge_{m_i = \mathsf{d}} isTree(a_i) \\ \Gamma = \Gamma_R + [f : \overline{m_i}^n \to \mathsf{s}] + \bigoplus_{i=1}^{n} [a_i : m_i] \qquad \Gamma_R = [y : \mathsf{r} \mid y \in R]\end{array}}{\Gamma \vdash_{Dst} f\,\overline{a_i}^n\,@\,\overline{r_j}^m : \mathsf{s}}\ [APP_{Dst}]$$

$$\frac{\Gamma = \bigoplus_{i=1}^{n} [a_i : \mathsf{s}]}{\Gamma \vdash_{Dst} C\,\overline{a_i}^n\,@\,r : \mathsf{s}}\ [CONS_{Dst}]$$

$$\frac{\forall i \in \{1..n\} . \Gamma + [\overline{x_{ij} : m_{ij}}^{n_i}] \vdash_{Dst} e_i : \mathsf{s}}{\Gamma \sqcup [x : \mathsf{s}] \vdash_{Dst} \mathbf{case}\, x\, \mathbf{of}\, \overline{C_i\,\overline{x_{ij}}^{n_i} \to e_i}^n : \mathsf{s}}\ [CASE_{Dst}]$$

$$\frac{\begin{array}{c}R = sharerec(x, \mathbf{case}\, x\, \mathbf{of}\, \overline{C_i\,\overline{x_{ij}}^{n_i} \to e_i}^n) \backslash \{x\} \quad \forall i \in \{1..n\} . \forall j \in \{1..n_i\} . inh_{i,C_i}(m_{ij}) \\ \forall z \in R \cup \{x\}, \forall i \in \{1..n\} . z \notin fv(e_i) \qquad \forall i \in \{1..n\} . \Gamma + [\overline{x_{ij} : m_{ij}}^{n_i}] \vdash_{Dst} e_i : \mathsf{s} \\ \Gamma_R = [y : \mathsf{r} \mid y \in R]\end{array}}{\Gamma_R \sqcup (\Gamma \backslash x) + [x : \mathsf{d}] \vdash_{Dst} \mathbf{case!}\, x\, \mathbf{of}\, \overline{C_i\,\overline{x_{ij}}^{n_i} \to e_i}^n : \mathsf{s}}\ [CASE!_{Dst}]$$

Figure 3.19: Typing rules for deriving $\vdash_{Dst}$ judgements.

$$\begin{array}{ll} inh_{i,C}(\mathsf{s}) & \text{if } i \notin RecPos(C) \\ inh_{i,C}(\mathsf{d}) & \text{if } i \in RecPos(C) \end{array}$$

Figure 3.20: Inheritance compatibility definitions for the $\vdash_{Dst}$ rules.

$$\circ : \mathbf{SafeType} \times \mathbf{Mark} \to \mathbf{ExpType}$$

$$\begin{array}{rcl} B \circ m & = & B \\ \alpha \circ m & = & \alpha \\ T\,\overline{s_i}^n\,@\,\overline{\rho_j}^m \circ \mathsf{s} & = & T\,\overline{s_i}^n\,@\,\overline{\rho_j}^m \\ T\,\overline{s_i}^n\,@\,\overline{\rho_j}^m \circ \mathsf{r} & = & T\,\overline{s_i}^n\,\#@\,\overline{\rho_j}^m \\ T\,\overline{s_i}^n\,@\,\overline{\rho_j}^m \circ \mathsf{d} & = & T\,\overline{s_i}^n\,!@\,\overline{\rho_j}^m \end{array}$$

Figure 3.21: Compositionality of types.

Definition 3.47. Two environments Γ_R and Γ_D are composable if the following conditions hold:

1. $\text{dom}\ \Gamma_D = \text{dom}\ \Gamma_R \backslash (\mathbf{RegType} \cup \mathbf{Cons})$
2. For every variable $x \in \text{dom}\ \Gamma_D$, $\Gamma_R(x) \circ \Gamma_D(x)$ is well-defined.
3. For every function symbol $f \in \text{dom}\ \Gamma_D$, $\Gamma_R(f) \circ \Gamma_D(f)$ is well-defined.

In this case, we define the composition of two typing environments $\Gamma_R \circ \Gamma_D$ as follows:

$$\begin{array}{rcll} \Gamma_R \circ \Gamma_D & \stackrel{\text{def}}{=} & [x \mapsto \Gamma_R(x) \circ \Gamma_D(x) & | \quad x \in \text{dom}\ \Gamma_D] \\ & + & [f \mapsto \Gamma_R(f) \circ \Gamma_D(f) & | \quad f \in \text{dom}\ \Gamma_D] \\ & + & [r \mapsto \Gamma_R(r) & | \quad r \in \text{dom}\ \Gamma_R] \\ & + & [C \mapsto \Gamma_R(C) & | \quad C \in \text{dom}\ \Gamma_R] \end{array}$$

Now we can state formally the equivalence between the two smaller type systems and the original one. Before this, we need two auxiliary lemmas, whose proof is straightforward and will be omitted here.

Lemma 3.48 (Environment reduction)**.** *If x (resp. r, f) does not occur in e, and it does belong to the domain of Γ, then for any $s' \in$ **SafeType**:*

$$\Gamma + [x : s'] \vdash_{Reg} e : s \Leftrightarrow \Gamma \vdash_{Reg} e : s$$

*and respectively with $[r : \rho]$ and $[f : sf]$ for any $\rho \in$ **RegType** and $sf \in$ **SafeFunType**.*

Proof. By induction on the size of the corresponding $\vdash_{Reg}$ derivation. □

Lemma 3.49. *Given a region environment Γ (resp. mark environment) such that $\Gamma \vdash_{Reg} e : s$ (resp. $\Gamma \vdash_{Dst} e : \mathsf{s}$). If $x \in fv(e)$, then $x \in dom\,\Gamma$. Similarly for the function and constructor symbols being applied in e.*

Proof. By induction on the size of the $\vdash_{Reg}$ (resp. $\vdash_{Dst}$) derivation. □

Theorem 3.50. *For any two composable environments Γ_R and Γ_D, any expression e and any safe type s:*

$$\Gamma_R \circ \Gamma_D \vdash e : s \quad \Longleftrightarrow \quad \Gamma_R \vdash_{Reg} e : s \ \wedge\ \Gamma_D \vdash_{Dst} e : \mathsf{s}$$

Proof. Let us denote by Γ the result of $\Gamma_R \circ \Gamma_D$, which exists because Γ_R and Γ_D are composable. First we prove the $(\Rightarrow)$ implication by induction on the $\vdash$ derivation. We distinguish cases on the last rule applied:

- **Case** [EXT]

  From one of the premises of this rule we get $\Gamma' \vdash e : s$, where $\Gamma' \subseteq \Gamma$. We can break down Γ' as follows $\Gamma' = \Gamma'_R \circ \Gamma'_D$. By induction hypothesis we obtain $\Gamma'_R \vdash_{Reg} e : s$ and $\Gamma'_D \vdash_{Dst} e : \mathsf{s}$. From the latter judgement we can derive $\Gamma_D \vdash_{Dst} e : \mathsf{s}$ by applying $[EXT_{Dst}]$, if necessary. The variables occurring in the domain of Γ, but not in the domain of Γ' do not appear free in e. Hence, by applying Lemmas 3.49 and 3.48 we obtain $\Gamma_R \vdash_{Reg} e : s$.

- **Case** [WEAK]

  In this case we get $\Gamma' \vdash e : s$ for some $\Gamma' \leq \Gamma$. If we split Γ' into $\Gamma'_R \circ \Gamma'_D$ it holds that $\Gamma'_R = \Gamma_R$, since Γ and Γ' only differ in their marks. Hence, by induction hypothesis we get $\Gamma'_R \vdash_{Reg} e : s$ and $\Gamma'_D \vdash_{Dst} e : \mathsf{s}$. From the first fact we get $\Gamma_R \vdash_{Reg} e : s$, whereas in the second we can apply the $[WEAK_{Dst}]$ rule.

- **Cases** [LIT], [VAR], **and** [COPY]

  All these cases are trivial.

- **Case** [LET]

  We get, for some Γ_1 and Γ_2 such that $\Gamma = \Gamma_1 \sqcup \Gamma_2$, the following judgements:

  $$\Gamma_1 \vdash e_1 : s_1 \qquad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s$$

  Each typing environment can be separated as follows:

  $$\Gamma_1 = \Gamma_{1,R} \circ \Gamma_{1,D} \qquad \Gamma_2 = \Gamma_{2,R} \circ \Gamma_{2,D}$$

  By applying the induction hypothesis to each we obtain:

  $$\begin{array}{ll} \Gamma_{1,R} \vdash_{Reg} e_1 : s_1 & \Gamma_{1,D} \vdash_{Dst} e_1 : \mathtt{s} \\ \Gamma_{2,R} + [x_1 : s_1] \vdash_{Reg} e_2 : s & \Gamma_{2,D} + [x_1 : m] \vdash_{Dst} e_2 : \mathtt{s} \end{array}$$

  where $s_1 \circ m = \tau_1$. It is easy to show that $\Gamma_{1,D} \sqcup \Gamma_{2,D}$ is equivalent to Γ_D. Moreover, the side condition of [LET] implies its counterpart in $[LET_{Dst}]$, as, for every x, $\Gamma_{1,D}(x) \in \{\mathtt{d}, \mathtt{r}\}$ implies $\Gamma_1(x) \in \mathbf{UnsafeType}$. Hence, by applying the $[LET_{Dst}]$ rule to the judgements on the right-hand side, we get $\Gamma_D \vdash_{Dst} e : \mathtt{s}$. Moreover, we know that the variables occurring in both $\Gamma_{1,R}$ and $\Gamma_{2,R}$ get the same type in both environments, because of the *utype*? constraint imposed by the $\sqcup$ operator. Hence we can extend both environments by using Lemma 3.48 in order to get $\Gamma_R \vdash_{Reg} e_1 : s_1$ and $\Gamma_R + [x_1 : s_1] \vdash_{Reg} e_2 : s$. Now we can apply the $[LET_{Reg}]$ rule and obtain $\Gamma_R \vdash_{Reg} e : s$.

- **Case** [APP]

  We get $\Gamma = \Gamma_0 + [f : \forall \overline{\alpha}\, \overline{\rho}.tf] + [\overline{r_j : \rho_j}^m] + \bigoplus_{i=1}^n [\overline{a_i : t_i}^n]$, where the Γ_R of the [APP] rule has been substituted by Γ_0 to avoid confusion with the region environment Γ_R. The Γ environment is split into:

  $$\Gamma_R = \Gamma_{0,R} + [f : \forall \overline{\alpha}\, \overline{\rho}.sf] + [\overline{r_j : \rho_j}^m] + \bigoplus_{i=1}^{n} [\overline{a_i : s_i}^n]$$

  $$\Gamma_R = \Gamma_{0,D} + [f : \overline{m_i} \to m] + \bigoplus_{i=1}^{n} [\overline{a_i : m_i}^n]$$

  where $t_i = s_i \circ m_i$ for all $i \in \{1..n\}$. From the premises of the [APP] rule it follows $\Gamma_D \vdash_{Dst} e : \mathtt{s}$. For applying the $[APP_{Reg}]$ rule we have to show that $\Gamma_{0,R} + \bigoplus_{i=1}^n [\overline{a_i : s_i}^n] \vdash_{Reg} a_i : s_i$ for each $i \in \{1..n\}$, but we can prove this easily by case distinction on whether each a_i is a literal or a variable.

- **Case** [CONS]

  Similar to the [APP] case.

- **Cases** [CASE] and [CASE!]

  Again let us rename the Γ_R environment of [CASE!] by a less confusing name Γ_0. The premises of both judgements $\Gamma_R \vdash_{Reg} e : s$ and $\Gamma_D \vdash_{Dst} e : \mathtt{s}$ follow trivially from their counterparts in the $\vdash$ derivation. Before applying the $[CASE!_{Reg}]$ rule, we may need to apply Lemma 3.48 to the

$\vdash_{Reg}$ typing of each **case** branch, in order to append the Γ_0 environment and be able to derive $\Gamma_R \vdash_{Reg} e : s$.

Now we prove the ($\Leftarrow$) implication by induction on the $\vdash_{Dst}$ derivation. In some cases we will prove that $\Gamma'_R \circ \Gamma_D \vdash e : s$ holds for some subset Γ'_R of Γ_R such that $\mathrm{dom}\,\Gamma_R \backslash \mathrm{dom}\,\Gamma'_R \subseteq \mathbf{RegVar} \cup \mathbf{Cons}$. From this judgement we can apply the [EXT] rule of the type system in order to obtain the desired result: $\Gamma_R \circ \Gamma_D \vdash e : s$. Again, we distinguish cases on the last rule applied in the $\vdash_{Dst}$ derivation:

- **Case** $[EXT_{Dst}]$

  From this rule we get $\Gamma'_D \vdash_{Dst} e : \mathsf{s}$ for some $\Gamma'_D \subseteq \Gamma_D$. Since every variable x occurring in $\mathrm{dom}\,\Gamma_D \backslash \mathrm{dom}\;\Gamma'_D$ does not occur free in e, we can remove the bindings of those variables in $\mathrm{dom}\,\Gamma_D \backslash \mathrm{dom}\,\Gamma'_D$ from Γ_R by applying Lemma 3.49 in order to obtain an environment Γ'_R composable with Γ'_D such that $\Gamma'_R \vdash_{Reg} e : s$. By induction hypothesis, $\Gamma'_R \circ \Gamma'_D \vdash e : s$. Moreover, $\Gamma'_R \circ \Gamma'_D \subseteq \Gamma$. Hence we can apply [EXT] so as to get $\Gamma \vdash e : s$.

- **Case** $[WEAK_{Dst}]$

  Assume $\Gamma_D + [x : m_2] \vdash e : \mathsf{s}$ for some x and m_2. By the $[WEAK_{Dst}]$ rule there exists another mark $m_1 \leq m_2$ such that $\Gamma_D + [x : m_1] \vdash e : \mathsf{s}$. If $\Gamma_D + [x : m_2]$ is composable with Γ_R, so is $\Gamma_D + [x : m_1]$, since they only differ in their marks. By induction hypothesis we get $(\Gamma_D + [x : m_1]) \circ \Gamma_R \vdash e : s$, in which $\Gamma_R(x) \circ m_1 \leq \Gamma_R(x) \circ m_2$. We can apply the [WEAK] rule of the type system so as to get the desired result.

- **Cases** $[LIT_{Dst}]$**,** $[VAR_{Dst}]$**, and** $[COPY_{Dst}]$

  The theorem holds trivially in all these cases. It is possible that the environment Γ_R contains more bindings than Γ_D, but these extra bindings must be of the form $[r : \rho]$ or $[C : \sigma]$, since Γ_R and Γ_D are composable. We can discard these bindings (Lemma 3.48) from Γ_R before applying the corresponding $\vdash$ rule ([LIT], [VAR], [COPY]) so as to get $\Gamma'_R \circ \Gamma_D \vdash e : s$ for some $\Gamma'_R \subseteq \Gamma_R$. After this, we apply the [EXT] rule to recover these bindings and get $\Gamma_R \circ \Gamma_D \vdash e : s$.

- **Case** $[LET_{Dst}]$

  In this case $\Gamma_D = \Gamma_{1,D} \sqcup \Gamma_{2,D}$ for some $\Gamma_{1,D}$,$\Gamma_{2,D}$. With the first environment we can type the auxiliary expression: $\Gamma_{1,D} \vdash_{Dst} e_1 : \mathsf{s}$, whereas we get $\Gamma_R \vdash_{Reg} e_1 : s_1$. However, since $\mathrm{dom}\,\Gamma_{1,D} \subseteq \mathrm{dom}\,\Gamma_D \subseteq \mathrm{dom}\,\Gamma_R$, we can remove bindings from Γ_R (by using Lemma 3.48) in order to get a typing environment $\Gamma_{1,R}$ composable with $\Gamma_{1,D}$, since every variable not occurring in $\Gamma_{1,D}$ is not free in e_1 (as Lemma 3.49 states).

  On the other hand, we have $\Gamma_{2,D} + [x_1 : m] \vdash_{Dst} e_2 : \mathsf{s}$ for some mark m and we can also remove bindings from Γ_R in order to obtain a $\Gamma_{2,R}$ composable with $\Gamma_{2,D}$ in which $\Gamma_{2,R} + [x_1 : s] \vdash_{Reg} e_2 : s$.

  By applying the induction hypothesis on the two $\vdash_{Dst}$ judgements, we get $\Gamma_{1,R} \circ \Gamma_{1,D} \vdash e_1 : s_1$ and $\Gamma_{2,R} \circ \Gamma_{2,D} + [x_1 : \tau_1] \vdash e_2 : s$, where $\tau_1 = s_1 \circ m$. Moreover, if x has an unsafe type in $\Gamma_{1,R} \circ \Gamma_{1,D}$, then $\Gamma_{1,D}(x) \in \{\mathsf{d}, \mathsf{r}\}$, so the side condition of [LET] follows from its counterpart in $[LET_{Dst}]$. It is obvious that $utype?(s_1, \tau_1)$, so let us apply the [LET] rule in order to obtain:

  $$(\Gamma_{1,R} \circ \Gamma_{1,D}) \sqcup (\Gamma_{2,R} \circ \Gamma_{2,D}) \vdash e : s$$

  Now we have to prove that $(\Gamma_{1,R} \circ \Gamma_{1,D}) \sqcup (\Gamma_{2,R} \circ \Gamma_{2,D})$ is well-defined and equal to $\Gamma_R \circ \Gamma_D$. The well-definedness follows from $\Gamma_{1,R}$ and $\Gamma_{2,R}$ mapping common variables to the same safe type

(because both environments are subsets of a same Γ_R). For the inclusion $(\Gamma_{1,R} \circ \Gamma_{1,D}) \sqcup (\Gamma_{2,R} \circ \Gamma_{2,D}) \subseteq \Gamma_R \circ \Gamma_D$, we distinguish cases on whether a given variable belongs to the domain of $(\Gamma_{1,R} \circ \Gamma_{1,D})$, or the domain of $(\Gamma_{2,R} \circ \Gamma_{2,D})$, or both. In any case the inclusion holds. The $\supseteq$ inclusion is a direct consequence of the fact that $\operatorname{dom} \Gamma_D = \operatorname{dom} \Gamma_{1,D} \cup \operatorname{dom} \Gamma_{2,D}$, so each variable in the domain of $\Gamma_R \circ \Gamma_D$ must also appear in the domain of $(\Gamma_{1,R} \circ \Gamma_{1,D}) \sqcup (\Gamma_{2,R} \circ \Gamma_{2,D})$, and also associated to the same type, because of the $\subseteq$ inclusion.

- **Case** $[APP_{Dst}]$

  By the rule $[APP_{Reg}]$ we get $\Gamma_R \vdash_{Reg} a_i : s_i$, so the binding $[a_i : s_i]$ is contained within Γ_R for those a_i that are not literals. However, because our convention of allowing literals in an environment with their respective type, we shall consider that $\bigoplus_{i=1}^{n}[a_i : s_i] \subseteq \Gamma_R$. By Lemma 3.48, we can discard bindings from Γ_R until we obtain:

$$\Gamma'_R = \Gamma_{0,R} + \bigoplus_{i=1}^{n}[a_i : s_i] + [\overline{r_j : \rho_j}^m] + [f : \forall \overline{\alpha}\ \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s]$$

  whereas:

$$\Gamma_D = \Gamma_{0,D} + \bigoplus_{i=1}^{n}[a_i : m_i] + [f : \overline{m_i}^n \to \mathsf{s}]$$

  and $\Gamma_{0,R}$ is composable with $\Gamma_{0,D}$. Therefore:

$$\Gamma'_R \circ \Gamma_D = (\Gamma_{0,R} \circ \Gamma_{0,D}) + \bigoplus_{i=1}^{n}[a_i : s_i \circ m_i] + [\overline{r_j : \rho_j}^m] + [f : \forall \overline{\alpha}\ \overline{\rho}.\overline{s_i \circ m_i}^n \to \overline{\rho_j}^m \to s]$$

  A given variable has a d mark in Γ_D if and only if it has a condemned type in $\Gamma'_R \circ \Gamma_D$. So, the set domain of the Γ_R environment in the [APP] rule (which we have denoted Γ_0 in this proof, to avoid confusion) is the same as that its counterpart in the $[APP_{Dst}]$ rule. Thus we can apply [APP] in order to get $\Gamma'_R \circ \Gamma_D \vdash e : s$. Since $\Gamma'_R \subseteq \Gamma_R$ and Γ_R is composable with Γ_D, the only bindings that can appear in Γ_R but not in Γ'_R are those of the form $[r : \rho]$ and $[C : \sigma]$. We can extend the environment $\Gamma'_R \circ \Gamma_D$ with the help of the [EXT] rule in order to get $\Gamma_R \circ \Gamma_D \vdash e : s$.

- **Case** $[CONS_{Dst}]$

  The proof is very similar to that of $[APP_{Dst}]$ rule.

- **Case** $[CASE_{Dst}]$

  Let n be the number of case branches. From the premises in the $[CASE_{Dst}]$ and $[CASE_{Reg}]$ rules we obtain, for each $i \in \{1..n\}$:

$$\Gamma_R + [x_{ij} : s_{ij}] \vdash_{Reg} e_i : s$$

$$\Gamma_D + [x_{ij} : m_{ij}] \vdash_{Dst} e_i : \mathsf{s}$$

  Both typing environments are composable, and hence we can apply the induction hypothesis in order to infer, for each $i \in \{1..n\}$: $\Gamma_R \circ \Gamma_D + [x_{ij} : s_{ij} \circ m_{ij}] \vdash e : s$. The *utype*? predicates in the [CASE] rule hold because the s_{ij} in those judgements match the types of their respective constructors, as the $[CASE_{Reg}]$ rule demands. The rest of the premises of the rule [CASE] are direct consequences of their counterparts in $[CASE_{Reg}]$ and $[CASE_{Dst}]$, so the theorem holds by applying [CASE].

- **Case** $[CASE!_{Dst}]$

  Let n be the number of case branches. We obtain $\Gamma_D = \Gamma_{0,D} \sqcup (\Gamma'_D \backslash x) + [x : \mathsf{d}]$ for some Γ'_D. We can also split Γ_R in the same way so that $\Gamma_R = \Gamma_{0,R} \sqcup (\Gamma'_R \backslash x) + [x : T @ \rho]$ such that Γ'_R and Γ'_D are composable. From the premises of each respective rules we obtain:

$$\Gamma'_D \backslash x + [x : m_i] + [\overline{x_{ij} : m_{ij}}^{n_i}] \vdash_{Dst} e_i : \mathsf{s}$$

$$\Gamma_{0,R} \sqcup (\Gamma'_R \backslash x) + [x : T @ \rho] + [\overline{x_{ij} : s_{ij}}^{n_i}] \vdash_{Reg} e_i : s$$

  for each $i \in \{1..n\}$. In case m_i does not occur in the $\vdash_{Dst}$ judgement we can add it via the $[EXT_{Dst}]$ rule. We can also remove the $\Gamma_{0,R}$ from the $\vdash_{Reg}$ judgements by using Lemma 3.48 (since the corresponding variables do not appear in the environment of the $\vdash_{Dst}$ derivations, they do not occur free in the e_i). For each branch, the resulting region environments are composable with the mark environments, so we can apply the induction hypothesis and obtain, for each $i \in \{1..n\}$

$$(\Gamma'_R \circ \Gamma'_D) \backslash x + [x : \tau] + [\overline{x_{ij} : s_{ij} \circ m_{ij}}^{n_i}] \vdash e_i : s$$

  for some τ. The *inh* predicates in rule [CASE!], as well as the rest of the premises in this rule can be easily derived from their counterparts in $[CASE!_{Dst}]$ and $[CASE(!)_{Reg}]$.

□

3.8 Conclusions and related work

In this chapter we have introduced a destruction-aware type system for *Safe* and proved it correct, in the sense that the live part of the heap will never contain dangling pointers. We have shown the expressiveness of the type system by means of several examples. Finally, we have defined two auxiliary systems which allow us to consider regions and explicit destruction separately, and will serve as a basis for the correctness results of the inference algorithms explained in the next two chapters.

Regarding regions, our type system presents obvious similarities with that of Tofte and Talpin [116], since our region model is inspired by the latter. Their correctness proof uses an involved coinductive approach for showing the preservation of consistency under evaluation. Most of the technical difficulty lies in the presence of arrow effects to reflect which regions are being read and written in each sub-expression. These effects are not necessary in *Safe*'s type system, since there is a single local RTV in every function (ρ_{self}). When considering this function from outside, the read and write effects on this RTV are hidden (i.e. are not reflected in the function's type). This considerably simplifies the consistency preservation proof (Theorem 3.26) and the region inference algorithm.

There has been research work aimed at providing a more intuitive understanding of the proofs in [116]. In particular, Boudol [21] extends Tofte and Talpin's calculus with a region deallocation construct which, on the one hand, allows the programmer to break the nested nature of regions, and, on the other hand, allows to simplify the original proof of [116]. A difference with respect our system is the presence of effects (Boudol introduces a new kind of negative effects), and the lack of an inference algorithm.

With regard to explicit destruction, our safety type system has some characteristics of linear types (see [118] as a basic reference). A number of variants of linear types have been developed for years for coping with the related problems of achieving safe in-place updates in functional languages [95] or detecting program sites where values could be safely deallocated [66]. The work closest to our system

is that of Aspinall and Hofmann [10], which proposes a type system for a language explicitly reusing heap cells. They prove that well-typed programs can be safely translated into an imperative language with an explicit deallocation/reusing mechanism. Their typing scheme includes three usage aspects for the variables occurring in the type environment. These are: (1) variable used destructively, (2) variable used read-only, but shares a component with the result, and (3) variable used read-only, and not shared with the result. We summarise here the differences and similarities with our work.

There are non-essential differences such as: (1) they only admit algorithms running in constant heap space, i.e. for each allocation there must exist a previous deallocation; (2) they use at the source level an explicit parameter d representing a pointer to the cell being reused; and (3) they distinguish two different Cartesian products depending on whether there is sharing or not between the tuple components. A major difference with respect to our initial type system of [84] is the use of a total order in their usage aspects, whereas we had defined a partial order in our marks. In this thesis we have improved the work in [84] in order to incorporate a total order in this analysis. In fact, the [WEAK] rule of Figure 3.6 has been inspired by the [DROP] rule of [10]. This change has had a positive impact in our system: on the one hand, it allows the type system to accept more pointer-safe functions, and, on the other hand, it considerably simplifies the inference algorithm (see Section 5.5). Besides this, there are some other essential differences with respect to [10]:

1. Their aspects 2 and 3 (read-only and shared, or just read-only) could be roughly assimilated to our use s (read-only), and their aspect 1 (destructive), to our use d (condemned), both defined in Section 3.3. We add a third use r (in-danger) arising from a sharing analysis based on abstract interpretation [98]. This use allows us to know more precisely which variables are in danger when a DS is destroyed.

2. We make a distinction between the recursive spine of a data structure, and the cells beyond that recursive spine. As a consequence, variables pointing to a non-recursive sub-structure of a destroyed cell can be safely accessed. Konečný introduces in [69] a set of separation assertions for distinguishing between the data and the spine in a DS. However, it is unclear how this system interacts with the different usage aspects of [10].

3. We use a separate analysis [98] for approximating the sharing relations, whereas in [10] sharing is tracked by means of the usage aspect 2.

4. Our typing rule for **let** $x_1 = e_1$ **in** e_2 allows more usage aspects combinations than theirs. Let $i \in \{1,2,3\}$ the aspect assigned to x_1, j the aspect of a variable z in e_1, and k the aspect of the variable z in e_2. We allow the following combinations (i,j,k) that they forbid: $(1,2,2)$, $(1,2,3)$, $(2,2,2)$, $(2,2,3)$. The deep reason is our more precise sharing information and the new in-danger type. In a more recent version of this system [11] combination $(2,2,3)$ is allowed.

   An example of a *Safe* program using the combination $(1,2,3)$ is the following:

   $$\textbf{let } x = z : [\,] \textbf{ in case}! \; x \textbf{ of} \ldots \textbf{case } z \textbf{ of} \ldots$$

   Variable x is destroyed, but a sharing variable z can be read both in the auxiliary and in the main expression. An example of Safe program using the combination $(1,2,2)$ is the following:

   $$\textbf{let } x = z : [\,] \textbf{ in case}! \; x \textbf{ of} \ldots z$$

   Here, the result z shares the destroyed variable x. Both programs take profit from the fact that the

sharing variable z is not a recursive descendant of x. Our type system assigns an s-type to these variables.

Chapter 4

Region Inference

4.1 Introduction

The type system of Chapter 3 establishes the safety of region-annotated *Core-Safe* programs. Given a *Safe* program without regions, our aim in this chapter is to annotate this program with region variables, so that the result is typable by using the $\vdash_{Reg}$ rules of Figure 3.18. We define a region inference algorithm that decorates data types and function definitions by maximizing the number of cells built in the working region *self*. Regarding this, the inference algorithm yields *optimal* programs with respect to the type system. This means that, if the programmer were allowed to annotate *Safe* programs by himself, he would not achieve a better result than the inference algorithm. Any attempt to improve the results of the inference algorithm (for example, by placing data structures in the *self* region, whereas the algorithm places them in an output region) leads to an ill-typed program.

Example 4.1. Let us consider the following function definition implementing the Quicksort algorithm. We show here the *Full-Safe* code, rather than its desugared version, for the sake of clarity:

$$
\begin{array}{lcl}
qsort\ [\,] & = & [\,] \\
qsort\ (x : xs) & = & append\ ls'\ (x : gs') \\
 & & \quad \textbf{where}\ (ls, gs) = partition\ x\ xs \\
 & & \qquad\quad ls' = qsort\ ls \\
 & & \qquad\quad gs' = qsort\ gs
\end{array}
$$

Let us assume that the region inference algorithm has already been applied to the *partition* and *append* functions, obtaining the following type schemes:

$$
\begin{array}{rcl}
partition & :: & Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4 \\
append & :: & [\alpha]@\rho_1 \to [\alpha]@\rho_2 \to \rho_2 \to [\alpha]@\rho_2
\end{array}
$$

The *partition* function receives three region arguments: the first and second one are the regions where the resulting lists containing the elements that are lower or greater than the pivot (first parameter) will be allocated. The last region argument specify the region in which the tuple grouping these results is stored. The *append* function receives a parameter indicating where to build the result, which has to be placed in the same region as the second parameter.

There are several ways in which a given function can be region-annotated. In particular, the follow-

ing region-annotated version of *qsort* would be accepted by the type system:

$$
\begin{array}{lll}
qsort :: [Int]@\rho_0 \to \rho_1 \to \rho_2 \to \rho_3 \to [Int]@\rho_2 & & \\
qsort\ [\,] & @\,r_1\ r_2\ r_3 = & [\,]\,@\,r_2 \\
qsort\ (x : xs) & @\,r_1\ r_2\ r_3 = & append\ ls'\ (x : gs')@r_2\,@\,r_2 \\
 & & \textbf{where}\ (ls, gs) = partition\ x\ xs\,@\,r_1\ r_2\ r_3 \\
 & & \quad ls' = qsort\ ls\,@\,r_1\ r_2\ r_3 \\
 & & \quad gs' = qsort\ gs\,@\,r_1\ r_2\ r_3
\end{array}
$$

The elements of the input list being lower than the pivot are stored in the region corresponding to r_1, whereas r_3 contains the region in which the tuples returned by the *partition* function are built. However, these DS are intermediate results that are not part of the sorted list returned by the *qsort* function. Consequently, the following choice of region annotations is better from the point of view of memory consumption, while still being accepted by the type system:

$$
\begin{array}{lll}
qsort\ [\,] & @\,r_2 = & [\,]\,@\,r_2 \\
qsort\ (x : xs) & @\,r_2 = & append\ ls'\ (x : gs')@r_2\,@\,r_2 \\
 & & \textbf{where}\ (ls, gs) = partition\ x\ xs\,@\,self\ self\ self \\
 & & \quad ls' = qsort\ ls\,@\,self \\
 & & \quad gs' = qsort\ gs\,@\,r_2
\end{array}
$$

Since r_1 and r_3 are not used anymore in the function's body, we can remove them from the list of parameters. Now the result of *partition* resides completely in the working region. This is the version inferred by the algorithm subject of this chapter. If we tried to improve the results by placing the gs' list in the *self* region, then the $(x : gs')$ list would also be foreced to reside in this region. The same would happen with the result of *append* and the empty list being built in the base case. The region r_1 would not be used anymore, so we would get the following region-annotated program:

$$
\begin{array}{lll}
qsort\ [\,] & = & [\,]\,@\,self \\
qsort\ (x : xs) & = & append\ ls'\ ((x : gs')@self)\ @\,self \\
 & & \textbf{where}\ (ls, gs) = partition\ x\ xs\,@\,self\ self\ self \\
 & & \quad ls' = qsort\ ls \\
 & & \quad gs' = qsort\ gs
\end{array}
$$

which is ill-typed, since the type of the result is $[Int]@\rho_{self}$, and the [FUNB] rule of the type system (Figure 3.9) could not be applied, because ρ_{self} occurs in the type of the result. This means that, at runtime, the resulting sorted list would be built in the working region and would be disposed of immediately after the call to *qsort*. □

As it was explained in Section 3.4, *Safe* supports polymorphic recursion on RTVs. This implies that, given a recursive function, the RTVs in the type of the recursive calls need not be the same as the RTVs in the type of the function. This can significantly decrease the memory needs of some functions, as the following example (taken from [116]) shows:

Example 4.2. Assume the following data type for heap integers consisting of a single constructor *HInt* taking an integer as a parameter:

$$\textbf{data}\ HInt\,@\,\rho = HInt\ Int\ @\ \rho$$

The addition function on heap integers is defined as follows:

$$add\ (HInt\ n)\ (HInt\ m) = HInt\ (n + m)$$

Our aim is to annotate the following function for computing Fibonacci numbers:

$$\begin{aligned}
&fib\ (HInt\ 0) = HInt\ 0\\
&fib\ (HInt\ 1) = HInt\ 1\\
&fib\ (HInt\ n) = add\ (fib\ (HInt\ (n-1)))\ (fib\ (HInt\ (n-2)))
\end{aligned}$$

Without allowing polymorphic recursion, the input given to the recursive calls to *fib* would be forced to live in the same region as the input given to the root call. Similarly, the partial results returned by the recursive calls would have to live in the same region as the global result given by the root call. As a consequence, we would obtain the following region-annotated definition:

$$\begin{aligned}
&fib :: HInt\ @\ \rho_1 \to \rho_1 \to \rho_2 \to HInt\ @\ \rho_2\\
&fib\ (HInt\ 0)\ @\ r_1\ r_2 = HInt\ 0\ @\ r_2\\
&fib\ (HInt\ 1)\ @\ r_1\ r_2 = HInt\ 1\ @\ r_2\\
&fib\ (HInt\ n)\ @\ r_1\ r_2 = add\ (fib\ (HInt\ (n-1)\ @\ r_1)\ @\ r_2)\ (fib\ (HInt\ (n-2)\ @\ r_1)\ @\ r_2)\ @\ r_2
\end{aligned}$$

Although the algorithm produces a great amount of intermediate results (namely, the heap integers passed to the recursive calls and the integers returned by those calls), none of these results is stored in the working region. As a consequence, the number of memory cells needed for evaluating the call $fib\ (HInt\ n)$ is proportional to 2^n. If we allow polymorphic recursion, we get the following region-annotated version:

$$\begin{aligned}
&fib :: HInt\ @\ \rho_1 \to \rho_2 \to HInt\ @\ \rho_2\\
&fib\ (HInt\ 0)\ @\ r_2 = HInt\ 0\ @\ r_2\\
&fib\ (HInt\ 1)\ @\ r_2 = HInt\ 1\ @\ r_2\\
&fib\ (HInt\ n)\ @\ r_2 = add\ (fib\ (HInt\ (n-1)\ @\ self)\ @\ self)\ (fib\ (HInt\ (n-2)\ @\ self)\ @\ self)\ @\ r_2
\end{aligned}$$

In this case, the RTVs in the type of the recursive occurrences of *fib* are region instances of the general type scheme inferred for this function:

$$\begin{aligned}
fib\ \text{function} &: \forall \rho_1\ \rho_2\ .\ HInt\ @\ \rho_1 \to \rho_2 \to HInt\ @\ \rho_2\\
\text{Recursive calls} &: \qquad HInt\ @\ \rho_{self} \to \rho_{self} \to HInt\ @\ \rho_{self}
\end{aligned}$$

The intermediate heap integers generated by *fib* are stored in the working region. This results in an algorithm of linear memory complexity, instead of exponential. As an additional advantage, the latter version does not need the region parameter r_1. □

The region inference problem was already addressed by Tofte and Talpin in the context of the MLKit language [116, 113]. Their inference algorithm, described in [112], deals with higher-order programs written in Standard ML. Since (at this moment) *Safe* is a first-order language, region inference can be expected to be simpler and more efficient than that of MLKit. Their algorithm runs in time $\mathcal{O}(n^4)$ in the worst case, where n is the size of the term, including in it the Hindley-Milner type annotations. The explanation of the algorithm and of its correctness arguments [112] needed around 40 pages of dense writing. So, it is not an easy task to incorporate the MLKit ideas into a new language.

$$
\begin{array}{rcll}
\mathbf{Prog}_0 \ni & prog_0 & \rightarrow & \overline{data_{0,i}}; \overline{def_{0,i}} \; ; \epsilon_0 \\
\mathbf{DecData}_0 \ni & data_0 & \rightarrow & \mathbf{data}\ T\ \overline{\alpha_i} = \overline{C_i\ \overline{t_{ij}}^{n_i}} \\
\mathbf{DecFun}_0 \ni & def_0 & \rightarrow & f\ \overline{x_i} = \epsilon_0 \\
\mathbf{Exp}_0 \ni & \epsilon_0 & \rightarrow & a & \{\text{literal } c \text{ or variable } x\} \\
 & & & \mid x@ & \{\text{copy}\} \\
 & & & \mid C\ \overline{a_i} & \{\text{constructor}\} \\
 & & & \mid f\ \overline{a_i} & \{\text{function application}\} \\
 & & & \mid \mathbf{let}\ x_1 = \epsilon_{0,1}\ \mathbf{in}\ \epsilon_{0,2} & \{\mathbf{let}\ \text{declaration}\} \\
 & & & \mid \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow \epsilon_{0,i}} & \{\text{read only pattern matching}\} \\
 & & & \mid \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow \epsilon_{0,i}} & \{\text{destructive pattern matching}\}
\end{array}
$$

Figure 4.1: Input language definition: *Core-Safe* without regions.

$$
\begin{array}{l}
inferProgram :: \mathbf{SafeTypeEnv} \rightarrow \mathbf{Prog}_0 \rightarrow (\mathbf{SafeTypeEnv}, \mathbf{Prog}) \\
inferProgram\ \Gamma\ (\overline{data_{0,i}}; \overline{def_{0,i}}; \epsilon_0) = (\Gamma', \overline{data_i}; \overline{def_i}; e) \\
\quad \mathbf{where}\ \ (\Gamma_d, \overline{data_i}) = \mathsf{mapAccumL}\ inferDecData\ \Gamma\ \overline{data_{0,i}} \\
\qquad\qquad\ \ (\Gamma', \overline{def_i}) = \mathsf{mapAccumL}\ inferDef\ \Gamma_d\ \overline{def_{0,i}} \\
\qquad\qquad\ \ e = inferMainExpression\ \Gamma'\ \epsilon_0
\end{array}
$$

Figure 4.2: A high-level view of the Hindley-Milner and region inference algorithm.

This chapter is an extended version of [85], whose contribution is a simple region inference algorithm for *Safe*. If polymorphic recursion is not inferred, it runs in time $\mathcal{O}(n)$ in the worst case, being n as above, while if polymorphic recursion appears, it needs time $\mathcal{O}(n^2)$ in the worst case. Moreover, the first phase of the algorithm can be directly integrated in the usual Hindley-Milner type inference algorithm, just by considering RTVs as ordinary polymorphic type variables. The second phase involves very simple set operations and the computation of a fixed point. Unlike [112], termination is always guaranteed without special provisions. There, they had to sacrifice principal types in order to ensure termination. Due to its simplicity, we believe that our algorithm can be easily reused in a different first-order functional language featuring Hindley-Milner types.

The algorithm described in this chapter has been implemented as a part of the *Safe* compiler. Although the implementation works at the *Full-Safe* level, the description given here applies only to *Core-Safe*. Its extension to *Full-Safe* is straightforward. In Section 4.2 we give an overview of the inference algorithm. Sections 4.3 and 4.4 deal with the different phases of the algorithm, whereas in Section 4.5 we prove its correctness. Finally, we give some examples in Section 4.6.

4.2 A high level view of the algorithm

The algorithm receives a region-free *Core-Safe* program $prog_0 \in \mathbf{Prog}_0$ and produces a region-annotated *Core-Safe* program $prog \in \mathbf{Prog}$ as a result. The syntax of region-annotated programs has already been described in Section 2.3, whereas the syntax of the region-free programs is shown in Figure 4.1. The only difference between the two syntaxes is the absence of region types and region variables in the latter. We use $data_0$, def_0 and ϵ_0 to denote respectively data definitions, function definitions and expressions in this region-free language.

In Figure 4.2 we show, in a Haskell-like pseudo-code, the definition of the whole process. The function *inferProgram* receives, besides the region-free input program, a typing environment containing the types of every builtin constructor and function (e.g. arithmetic/logic operations and list constructors).

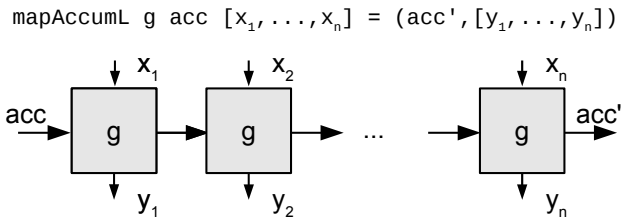


Figure 4.3: Visual description of the mapAccumL function, which behaves like map, but propagating an accumulator. It returns the transformed list and the final accumulator.

In addition to the final region-annotated program, this function returns the extension of the initial Γ with the types of each constructor or function being defined in the program. The function mapAccumL, described in [76], applies a function to each element of a list while passing an accumulator parameter from left to right (see Figure 4.3). Firstly, each **data** declaration is processed by *inferDecData*, which also extends the input environment Γ with the types of the constructors being defined in each data type. After this, the function definitions are processed by the function *inferDef*, which also augments the typing environment resulting from the previous phase with the inferred types for each function definition. Finally, the region annotations of the main expression are inferred.

4.3 Region inference of data declarations

In this phase the algorithm decorates the type and data constructors with region type variables. Each **data** declaration is processed individually. Our aim is to generate as many distinct RTVs as possible for each data constructor, since this implies that, at runtime, the data structures of this type can be distributed among several regions. In this way, we achieve more granularity for deciding which part of a data structure should be place in the *self* region in order to be disposed of as soon as possible.

This phase also generates a typing environment Γ that maps every constructor name being defined to its type signature. We start from an initial environment containing the type of the following built-in constructors:

$$
\begin{array}{lcll}
[\,] & :: & \forall \alpha\, \rho :\ \rho \to [\alpha]@\rho & \{\text{empty list constructor}\} \\
(:) & :: & \forall \alpha\, \rho :\ \alpha \to [\alpha]@\rho \to \rho \to [\alpha]@\rho & \{\text{list constructor (head:tail)}\} \\
(,) & :: & \forall \alpha\, \beta\, \rho :\ \alpha \to \beta \to \rho \to (\alpha, \beta)@\rho & \{\text{pair constructor}\}
\end{array}
$$

In this section we give an informal description of the region inference algorithm for the **data** declarations, together with a running example. Let us consider the following declaration:

$$\textbf{data}\ T\ \alpha\ \beta = C\ (\alpha, \alpha)\ [\beta] \mid D\ \alpha\ (T\ \alpha\ \beta)$$

The steps of the algorithm are as follows:

1. Decorate each non-recursive nested data type with fresh RTVs. All these data types are assumed to have been inferred previously, so we know the exact number of RTVs needed in each occurrence. Basic types *Int* and *Bool* do not require a RTV, as their values do not reside by themselves in the heap. Built-in data types (tuples and lists) require one RTV.

In our example, we obtain:

$$\textbf{data}\ T\ \alpha\ \beta = C\ (\alpha,\alpha)@\rho_1\ [\beta]@\rho_2 \mid D\ \alpha\ (T\ \alpha\ \beta)$$

2. Generate an additional fresh RTV and decorate each constructor with it. This is the *outermost* RTV of the type being defined.

$$\textbf{data}\ T\ \alpha\ \beta = C\ (\alpha,\alpha)@\rho_1\ [\beta]@\rho_2\ @\ \rho_3 |\ D\ \alpha\ (T\ \alpha\ \beta)\ @\ \rho_3$$

3. Add the RTVs generated in the two previous steps as extra region type parameters for the type constructor being defined. The last parameter must correspond to the outermost RTV obtained in step 2.

$$\textbf{data}\ T\ \alpha\ \beta\ @\ \rho_1\ \rho_2\ \rho_3 = C\ (\alpha,\alpha)@\rho_1\ [\beta]@\rho_2\ @\ \rho_3 |\ D\ \alpha\ (T\ \alpha\ \beta)\ @\ \rho_3$$

4. Now that we know which RTVs are associated with the current data type, we can annotate the recursive occurrences of the latter. Polymorphic recursion over region types is supported in function signatures, but not in data type definitions. Therefore the list of RTVs in the recursive occurrences must be the same as in the left-hand side of the definition.

   In our example we get the following result:

$$\textbf{data}\ T\ \alpha\ \beta\ @\ \rho_1\ \rho_2\ \rho_3 = C\ (\alpha,\alpha)@\rho_1\ [\beta]@\rho_2\ @\ \rho_3 |\ D\ \alpha\ (T\ \alpha\ \beta @\ \rho_1\ \rho_2\ \rho_3)\ @\ \rho_3$$

5. Finally, we compute the type signature for the data constructors from the types of their arguments.

$$\begin{array}{lcl} C & :: & \forall\alpha\,\beta\,\rho_1\,\rho_2\,\rho_3 :\ (\alpha,\alpha)@\rho_1 \to [\beta]@\rho_2 \to \rho_3 \to T\ \alpha\ \beta\ @\ \rho_1\ \rho_2\ \rho_3 \\ D & :: & \forall\alpha\,\beta\,\rho_1\,\rho_2\,\rho_3 :\ \alpha \to (T\ \alpha\ \beta @\ \rho_1\ \rho_2\ \rho_3) \to \rho_3 \to T\ \alpha\ \beta\ @\ \rho_1\ \rho_2\ \rho_3 \end{array}$$

   This phase extends the typing environment Γ with the types of these constructors.

Example 4.3. Consider the following data type declarations without regions:

$$\begin{array}{l} \textbf{data}\ \mathit{HInt} = \mathit{HInt}\ \mathit{Int} \\ \textbf{data}\ \mathit{Table} = \mathit{TBL}\ \ [(\mathit{Int}, \mathit{HInt})] \end{array}$$

In the first declaration we have to add a RTV for the data constructor *HInt*:

$$\textbf{data}\ \mathit{HInt}\ @\ \rho = \mathit{HInt}\ \mathit{Int}\ @\ \rho$$

In the second declaration we need four RTVs: one for each nested data type and another one for the data being defined:

$$\textbf{data}\ \mathit{Table}\ @\ \rho_1\ \rho_2\ \rho_3\ \rho_4 = \mathit{TBL}\ \ [(\mathit{Int}, \mathit{HInt}\ @\ \rho_1)\ @\rho_2]\ @\rho_3\ @\ \rho_4$$

□

$$
\begin{array}{l}
\mathit{inferDef} :: \mathbf{SafeTypeEnv} \to \mathbf{DecFun}_0 \to (\mathbf{SafeTypeEnv}, \mathbf{DecFun}) \\
\mathit{inferDef}\ \Gamma\ (f\ \overline{x_i}^n = \epsilon_0) = (\Gamma + [f : \overline{s_i}^n \to \overline{\rho_j}^m \to s], f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e) \\
\quad \textbf{where}\ \ \mathit{fresh}(\alpha_0, \overline{\alpha_i}^n, \overline{r_i}^m) \\
\qquad \Gamma + [f : \alpha_0] + [\overline{x_i : \alpha_i}^n] \vdash_f \epsilon_0 \longrightarrow \epsilon : \alpha \mid E \qquad \text{(A)} \\
\qquad \langle \theta, \overline{\varphi_i}^p \rangle \quad = \quad \mathit{solveEqs}\ E \qquad \text{(B)} \\
\qquad \overline{s_i}^n \to s \quad = \quad \theta(\alpha_0) \qquad \text{(C)} \\
\qquad (R_{arg}, R_{self}, \_) \quad = \quad \mathit{inferRegions}\ (\overline{s_i}^n \to s)\ \theta(\epsilon)\ \overline{\varphi_i}^p \qquad \text{(D)} \\
\qquad \{\overline{\rho_j}^m\} \quad = \quad R_{arg} \qquad \text{(E)} \\
\qquad \mathcal{R} \quad = \quad [\overline{\rho_i \mapsto r_i}^m] + [\rho \mapsto \mathit{self} \mid \rho \in R_{self}] \qquad \text{(F)} \\
\qquad e \quad = \quad \mathit{annotateExp}\ \mathcal{R}\ \epsilon \qquad \text{(G)}
\end{array}
$$

Figure 4.4: Region inference algorithm for function definitions.

4.4 Region inference of function definitions

Our region inference algorithm for function definitions is implemented as an extension of Hindley-Milner inference (HM in the following). In Figure 4.4 we give its definition. The first two steps (A) and (B) deal with HM inference. As usual in many of its presentations [99], we consider two sub-phases: generation of a system of equations between types and its later resolution via unification. This phase produces two results:

1. An intermediate expression ϵ decorated with type information, which will be used in the subsequent phases.
2. A solution $\langle \theta, \overline{\varphi_i}^p \rangle$ to the system of equations, whose meaning will be made precise later.

Section 4.4.1 describes this phase in detail. With the resulting substitution θ we can obtain (step (C)) a preliminary type $\overline{s_i}^n \to s$ (without regions) for the function. The next step (D), which is explained in Section 4.4.2, determines which RTVs have to appear as additional parameters in the function's signature (R_{arg}) and which ones must be unified with ρ_{self} (R_{self}). Once we have computed the corresponding R_{arg} and R_{self}, we build in (F) a correspondence $\mathcal{R}$ between region types and freshly generated region variables. We apply this correspondence to the annotations of the intermediate expression ϵ in order to obtain the final *Core-Safe* expression (step (G)). Besides this, the type of the function is augmented with the variables occuring in R_{arg} and included into the resulting typing environment.

4.4.1 HM Inference

Our implementation of this phase extends the usual HM inference in the following ways:

- **Generation of equations between region types:** Algebraic data types have already been annotated with RTVs by means of the algorithm explained in Section 4.3. The equations being generated in this phase include these region types, which are treated like ordinary polymorphic variables. In particular, some of them may be unified as a consequence of solving the system of equations.
- **Weak forms of unification equations:** In absence of polymorphic recursion, the type of the function being defined must be equal to the type of the function in each call. Since we allow polymorphic recursion over regions, we can relax this requirement, so the type of the function in each call site may be a *region type instance* of the function's type. This means that both types need not be syntactically equal after unification, but there must exist a list $\overline{\varphi_i}$ of substitutions mapping the RTVs

$$
\begin{array}{lcll}
\mathbf{Exp}^1 \ni \epsilon & \to & a & \{\text{literal } c \text{ or variable } x\} \\
& & \mid x@_{[\rho]} & \{\text{copy}\} \\
& & \mid C\ \overline{a_i}_{[\rho]} & \{\text{constructor}\} \\
& & \mid f\ \overline{a_i}_{[\overline{\rho_j}]} & \{\text{nonrecursive function application}\} \\
& & \mid f\ \overline{a_i}_{[k,\overline{t_i}\to t]} & \{\text{recursive function application}\} \\
& & \mid \textbf{let } x_1 = \epsilon_1 \textbf{ in } \epsilon_2 & \{\textbf{let} \text{ declaration}\} \\
& & \mid \textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}} \to \epsilon_i} & \{\text{read only pattern matching}\} \\
& & \mid \textbf{case!} \ x \textbf{ of } \overline{C_i\ \overline{x_{ij}} \to \epsilon_i} & \{\text{destructive pattern matching}\}
\end{array}
$$

Figure 4.5: Intermediate language for expressions.

of the function's type into RTVs of the type of each recursive application. Besides polymorphic recursion, the *Safe* language has a copy expression, in which the types of source and destination DS may differ in the outermost region. For handling these cases, we introduce different kinds of constraints which impose fewer restrictions than unification equations.

- **Decoration of the source expression:** The abstract syntax tree of the input definition has to be decorated with typing information, which will be used in subsequent phases of the algorithm. For this purpose we define the set $\mathbf{Exp}_1$ of intermediate expressions by the grammar in Figure 4.5 and we use $\epsilon, \epsilon_1, \ldots$ to denote such expressions.

  - Copy expressions $x@$ are attached a region type variable ρ, which is the type of the region in which the destination DS will reside at runtime.
  - Constructor applications $C\ \overline{a_i}^n$ are decorated with another variable ρ, which is the outermost region type of the DS being created.
  - Function calls $g\ \overline{a_i}^n$ that are different from the function being defined (i.e. nonrecursive calls) receive a sequence of RTVs $\overline{\rho_j}^m$ as decoration. These types correspond to the region variables that will be added as additional arguments in this function call. Since the regions of these functions have already been inferred, the number of needed RTVs is known by the compiler.
  - Recursive function calls $f\ \overline{a_i}^n$ are decorated with an identifier $k \in \mathbb{N}$ and the concrete instantiation of f's type that is used in this recursive call. If the function being inferred has p recursive calls, we assume that each call has a unique number ranging from 1 to p.

The generation of unification equations is defined by a set of rules with operate on judgements of this form:

$$\Gamma \vdash_f \epsilon^0 \longrightarrow \epsilon : \alpha \mid E \tag{4.1}$$

where Γ is a pair of functions in $(\mathbf{Fun} \cup \mathbf{Cons} \to \mathbf{SafeFunType}) \times (\mathbf{Var} \to \mathbf{TypeVar})$ that map, respectively, function and constructor symbols to safe type schemes σ and program variables x to type variables α. Again, by abuse of notation, we will not make an explicit distinction between these mappings, and both of them will be denoted by Γ. It will be clear from the context which mapping we are referring to, depending on whether we write $\Gamma(f)$, $\Gamma(C)$ or $\Gamma(x)$. The intuitive meaning of (4.1) is that, under the environment Γ, the region-free expression $\epsilon_0 \in \mathbf{Exp}_0$ is transformed into the intermediate expression $\epsilon \in \mathbf{Exp}_1$ of type α. As a side effect, the set E of constraints between types is generated. Each constraint may be of one of these types:

$s_1 = s_2$

$$\frac{c \in \mathbf{Int}}{\Gamma \vdash_f c \longrightarrow c : \alpha \mid \{\alpha = Int\}}\ [\text{LITI}_{\text{Gen}}] \qquad \frac{c \in \mathbf{Bool}}{\Gamma \vdash_f c \longrightarrow c : \alpha \mid \{\alpha = Bool\}}\ [\text{LITB}_{\text{Gen}}]$$

$$\frac{}{\Gamma \vdash_f x \longrightarrow x : \alpha \mid \{\Gamma(x) = \alpha\}}\ [\text{VAR}_{\text{Gen}}]$$

$$\frac{fresh(\rho)}{\Gamma \vdash_f x@ \longrightarrow x@_{[\rho]} : \alpha \mid \{\alpha = isData(\Gamma(x), \rho)\}}\ [\text{COPY}_{\text{Gen}}]$$

$$\frac{fresh(\rho) \qquad \forall i \in \{1 \ldots n\} : \Gamma \vdash_f a_i \longrightarrow a_i : \alpha_i \mid E_i \qquad \overline{s_i}^n \to \rho' \to s \preceq \Gamma(C)}{\Gamma \vdash_f C\ \overline{a_i}^n \longrightarrow C\ \overline{a_i}^n_{[\rho]} : \alpha \mid \{\overline{s_i}^n \to \rho' \to s = \overline{\alpha_i}^n \to \rho \to \alpha\} \cup \bigcup_{i=1}^n E_i}\ [\text{CONS}_{\text{Gen}}]$$

$$\frac{fresh(k) \qquad \forall i \in \{1 \ldots n\} : \Gamma \vdash_f a_i \longrightarrow a_i : \alpha_i \mid E_i}{\Gamma \vdash_f f\ \overline{a_i}^n \longrightarrow f\ \overline{a_i}^n_{[k, \overline{\alpha_i}^n \to \alpha]} : \alpha \mid \{\Gamma(f) \approx_k \overline{\alpha_i}^n \to \alpha\} \cup \bigcup_{i=1}^n E_i}\ [\text{APP-R}_{\text{Gen}}]$$

$$\frac{g \neq f \qquad \forall i \in \{1 \ldots n\} : \Gamma \vdash_f a_i \longrightarrow a_i : \alpha_i \mid E_i \qquad \overline{s_i}^n \to \overline{\rho_j}^m \to s \preceq \Gamma(g)}{\Gamma \vdash_f g\ \overline{a_i}^n \longrightarrow g\ \overline{a_i}^n_{[\overline{\rho_j}^m]} : \alpha \mid \{\overline{s_i}^n \to \overline{\rho_j}^m \to s = \overline{\alpha_i}^n \to \overline{\rho_j}^m \to \alpha\} \cup \bigcup_{i=1}^n E_i}\ [\text{APP-NR}_{\text{Gen}}]$$

$$\frac{\Gamma \vdash_f \epsilon_1^0 \longrightarrow \epsilon_1 : \alpha_1 \mid E_1 \qquad \Gamma \backslash x_1 + [x_1 : \alpha_1] \vdash_f \epsilon_2^0 \longrightarrow \epsilon_2 : \alpha_2 \mid E_2}{\Gamma \vdash_f \mathbf{let}\ x_1 = \epsilon_1^0\ \mathbf{in}\ \epsilon_2^0 \longrightarrow \mathbf{let}\ x_1 = \epsilon_1\ \mathbf{in}\ \epsilon_2 : \alpha \mid E_1 \cup E_2 \cup \{\alpha = \alpha_2\}}\ [\text{LET}_{\text{Gen}}]$$

$$\frac{\Gamma(x) = \alpha_x \qquad \forall i \in \{1 \ldots n\}. \begin{cases} fresh(\overline{\alpha_{ij}}^{n_i}) \ \wedge\ \overline{s_{ij}}^{n_i} \to \rho \to s_i \preceq \Gamma(C_i) \\ \Gamma \backslash \overline{x_{ij}}^{n_i} + \left[\overline{x_{ij} : \alpha_{ij}}^{n_i}\right] \vdash_f \epsilon_i^0 \longrightarrow \epsilon_i : \alpha_i \mid E_i' \\ E_i = E_i' \cup \{\overline{s_{ij}}^{n_i} \to s_i = \overline{\alpha_{ij}}^{n_i} \to \alpha_x\} \cup \{\alpha_i = \alpha\} \end{cases}}{\Gamma \vdash_f \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to \epsilon_i^0}^n \longrightarrow \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to \epsilon_i}^n : \alpha \mid \bigcup_{i=1}^n E_i}\ [\text{CASE}_{\text{Gen}}]$$

Figure 4.6: Rules for generating equations.

It specifies that the types at both sides of the $=$ must be equal after unification.

$s_1 = isData(s_2, \rho)$

This means that the types of s_1 and s_2 may only differ in the outermost region, which must be ρ in the type s_1.

$s_1 \approx_k s_2$

It specifies that s_2 must be a region instance of s_1, but the HM types (i.e. those without considering regions) must be equal. This allows us to have polymorphic recursion over regions. We annotate each constraint of this form with a number k which is the identifier of the recursive call from which this constraint is generated (see below).

The rules that define the equation generation can be found in Figure 4.6. Although not stated explicitly, we assume that α denotes a *fresh* type variable in every rule. Moreover, we assume that each occurrence of $\preceq$ in the rules means that the type on the left-hand side is a *fresh* instance of the type scheme on the right-hand side. Most of the rules are the usual ones in HM inference, with some minor differences. Rule [COPY_{Gen}] generates an *isData* constraint, in order to specify that the outermost region type of the DS being copied may differ from ρ, which is the outermost region type of the result. We make distinction between recursive and non-recursive function applications. In the latter case, the function

being called has been already inferred, so we have to generate as many fresh RTVs as in the function's signature. In recursive function applications we generate an $\approx$ equation and a fresh identifier.

Example 4.4. Given the following function for appending two lists:

$$\begin{array}{lll} \mathit{append}\ \mathit{xs}\ \mathit{ys} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & & \quad [\,] \to \mathit{ys}\ @ \\ & & \quad (x : \mathit{xx}) \to \textbf{let}\ x_1 = \mathit{append}\ \mathit{xx}\ \mathit{ys}\ \textbf{in}\ (x : x_1) \end{array}$$

We start with an environment $\Gamma \supseteq [\mathit{append} : \alpha_0, \mathit{xs} : \alpha_1, \mathit{ys} : \alpha_2]$. The rules of Figure 4.6 generate the following constraints:

$$\begin{array}{rcll} [\alpha_3]@\rho_1 & = & \alpha_1 & \{[\,]\ \text{pattern}\} \\ \alpha_6 \to [\alpha_6]@\rho_2 \to [\alpha_6]@\rho_2 & = & \alpha_4 \to \alpha_5 \to \alpha_1 & \{(x : \mathit{xx})\ \text{pattern}\} \\ \alpha_7 & = & \mathit{isData}(\alpha_2, \rho_3) & \{\mathit{ys}\ @\} \\ \alpha_0 & \approx_1 & \alpha_5 \to \alpha_2 \to \alpha_8 & \{\text{call to } \mathit{append}\} \\ \alpha_9 \to [\alpha_9]@\rho_4 \to \rho_4 \to [\alpha_9]@\rho_4 & = & \alpha_4 \to \alpha_8 \to \rho_5 \to \alpha_{10} & \{(x : x_1)\} \\ \alpha_{11} & = & \alpha_7 & \{\textbf{case}\ \text{branch}\} \\ \alpha_{11} & = & \alpha_{10} & \{\textbf{case}\ \text{branch}\} \\ \alpha_0 & = & \alpha_1 \to \alpha_2 \to \alpha_{11} & \{\text{type of } \mathit{append}\} \end{array}$$

We obtain the following intermediate expression with decorations:

$$\begin{array}{lll} \epsilon_{\mathit{append}} & = & \textbf{case}\ \mathit{xs}\ \textbf{of} \\ & & \quad [\,] \to \mathit{ys}\ @_{[\rho_3]} \\ & & \quad (x : \mathit{xx}) \to \textbf{let}\ x_1 = \mathit{append}\ \mathit{xx}\ \mathit{ys}_{[1, \alpha_5 \to \alpha_2 \to \alpha_8]}\ \textbf{in}\ (x : x_1)_{[\rho_5]} \end{array}$$

The copy expression and the application of the constructor are annotated with the region type variables corresponding to their respective results, whereas the call to *append* is annotated with its concrete type. □

Once we have generated a set E of constraints, we can solve them by unification. The technique we use is a slight variation of original method defined by Robinson [101] with some extensions for handling *isData* constraints and $\approx$-equations, which are treated in this context like latent unifications. These extensions are straightforward to define and implement, and they are shown in Appendix A.

The unification algorithm returns a substitution θ mapping type variables α (resp. RTVs ρ) to types or type signatures (resp. to RTVs). It also yields, for each equation of the form $s_1 \approx_k s_2$, a substitution φ_k between region types which maps the region type variables of the left-hand side to the ones in the right-hand side. If the input function definition has p recursive calls (and hence it involves the generation of p equations of this kind) then the unification method yields a list of p substitutions $\overline{\varphi_i}^p$. In our implementation, both θ and the $\overline{\varphi_i}^p$ are computed simultaneously, but the computation of the $\overline{\varphi_i}^p$ can also be deferred after the unification.

Although we do not explain in this chapter how to obtain a solution via unification, we must specify the meaning of a solution in order to prove the correctness of the algorithm. In the following definition we use the notation $s_1 \equiv s_2$ to denote syntactic equality of types. We also assume that the application of substitutions θ and φ_i to type expressions is defined in the usual way.

Definition 4.5. A pair $\langle \theta, \overline{\varphi_i}^p \rangle$ is a solution to a set of constraints E if and only if the following conditions hold:

1. For each $s_1 = s_2 \in E$, then $\theta(s_1) \equiv \theta(s_2)$. The same applies to equations of the form $sf_1 = sf_2$.
2. For each $s = isData(s', \rho) \in E$ then $\theta(s) \equiv T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m(\theta(\rho))$ and $\theta(s') \equiv T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m\rho'$ for some $\overline{s_i}^n, \overline{\rho_j}^m$ and ρ'.
3. For each $s_1 \approx_k s_2$ then $\varphi_k(\theta(s_1)) \equiv \theta(s_2)$. The same applies to equations of the form $sf_1 \approx_k sf_2$.

Example 4.6. The unification method yields the following substitution for the Example 4.4 above:

$$\theta = \left[\begin{array}{lll} \alpha_1 \mapsto [\alpha_3]@\rho_1 & \alpha_2 \mapsto [\alpha_3]@\rho_6 & \alpha_4 \mapsto \alpha_3 \\ \alpha_5 \mapsto [\alpha_3]@\rho_1 & \alpha_6 \mapsto \alpha_3 & \alpha_7 \mapsto [\alpha_3]@\rho_4 \\ \alpha_8 \mapsto [\alpha_3]@\rho_4 & \alpha_9 \mapsto \alpha_3 & \alpha_{10} \mapsto [\alpha_3]@\rho_4 \\ & \alpha_{11} \mapsto [\alpha_3]@\rho_4 & \\ \rho_2 \mapsto \rho_1 & \rho_3 \mapsto \rho_4 & \rho_5 \mapsto \rho_4 \\ \multicolumn{3}{c}{\alpha_0 \mapsto [\alpha_3]@\rho_1 \to [\alpha_3]@\rho_8 \to [\alpha_3]@\rho_6} \end{array}\right]$$

We compare the type of the function with the type of the recursive application:

$$\begin{array}{rlll} \text{Function:} & append & :: & [a_3]@\rho_1 \to [a_3]@\rho_6 \to [a_3]@\rho_4 \\ \text{Recursive application:} & append & :: & [a_3]@\rho_1 \to [a_3]@\rho_6 \to [a_3]@\rho_4 \end{array}$$

Both types are equal, and hence there is no polymorphic recursion in this example. Therefore, we get the identity region mapping for this recursive call:

$$\varphi_1 = [\rho_1 \mapsto \rho_1, \rho_6 \mapsto \rho_6, \rho_4 \mapsto \rho_4]$$

By applying θ to the intermediate expression we obtain:

$$\begin{array}{rcl} \theta(\epsilon_{append}) & = & \textbf{case}\ xs\ \textbf{of} \\ & & \quad [\,] \to ys\ @_{[\rho_4]} \\ & & \quad (x : xx) \to \textbf{let}\ x_1 = append\ xx\ ys_{[1,[\alpha_3]@\rho_1 \to [a_3]@\rho_6 \to [a_3]@\rho_4]}\ \textbf{in}\ (x : x_1)_{[\rho_4]} \end{array}$$

□

4.4.2 Kernel of the algorithm

Once we have obtained a decorated expression and a solution $\langle \theta, \overline{\varphi_i}^p \rangle$ for the type equations, we apply the substitution θ to the type decorations of the expression, obtaining another decorated expression $\epsilon \in \mathbf{Exp}_1$ as a result. We also apply this substitution to the fresh type variable α_0 assigned to the function being inferred in order to obtain its type $\theta(\alpha_0) \equiv \overline{s_i}^n \to s$ with RTVs. Let us denote by R_{in} the region type variables occurring in the type of the input arguments (i.e. the $\overline{s_i}^n$), and by R_{out} those occurring in the result of the function (i.e. the s). Our next step is to classify these RTVs, along with those that have appeared in the decorations of ϵ. Some of these RTVs must be added to the type of the function as extra parameters; some others will be unified with the region type variable ρ_{self}. Once we have done this classification, the generation of region variables in order to produce the final *Core-Safe* expression is straightforward.

$$
\begin{array}{l}
\mathit{inferRegions}\ (\overline{s_i}^n \to s)\ \epsilon\ \overline{\varphi_j}^p = (R_{arg}, R_{self}, R_{expl}) \\
\quad \textbf{where} \quad \begin{array}{lcl}
R_{out} & = & \mathit{regions}(s) \\
R_{in} & = & \mathit{regions}(\overline{s_i}^n) \\
R_0 & = & \mathit{regions}(\epsilon) \\
(R_{arg}, R_{expl}) & = & \mathit{computeRargFP}\ R_{in}\ R_{out}\ \emptyset\ R_0\ \overline{\varphi_j}^p \\
R_{self} & = & R_{expl} \backslash (R_{in} \cup R_{out})
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{computeRargFP}\ R_{in}\ R_{out}\ R_{arg}\ R_0\ \overline{\varphi_j}^p \\
\quad \begin{array}{lcl}
\mid R_{arg} == R'_{arg} & = & (R'_{arg}, R'_{expl}) \\
\mid \textbf{otherwise} & = & \mathit{computeRargFP}\ R_{in}\ R_{out}\ R'_{arg}\ R_0\ \overline{\varphi_j}^p
\end{array} \\
\quad \textbf{where} \quad \begin{array}{lcl}
R'_{expl} & = & R_0 \cup \bigcup_{j=1}^{p} \{\varphi_j^+(\rho) \mid \rho \in R_{arg}\} \\
R'_{arg} & = & R'_{expl} \cap (R_{in} \cup R_{out})
\end{array}
\end{array}
$$

Figure 4.7: Second phase of the region inference algorithm.

The main phase of the region inference algorithm computes three sets of RTVs:

R_{arg} It contains the types of the region variables that will eventually be added as extra parameters of the function definition. In other words, if $\overline{s_i}^n \to s$ is the type we have obtained for the function in the previous phase, and $R_{arg} = \{\overline{\rho_j}^m\}$, then the function's signature will be extended as $\overline{s_i}^n \to \overline{\rho_j}^m \to s$.

R_{self} It contains the RTVs that will be unified with the ρ_{self} region. This implies that the corresponding region variable in the *Core-Safe* program will be *self*.

R_{expl} It is the set of *explicit* RTVs. A region type variable ρ is considered to be explicit if there exists a region variable r in the resulting *Core-Safe* program with type ρ.

These definitions are far from being constructive, since they all refer to the resulting region-annotated program, which is exactly what we aim for. However, they provide enough intuition in order to characterize some properties of these sets. We have to find R_{arg}, R_{self} and R_{expl} subject to the following restrictions:

1. $R_{expl} \subseteq R_{self} \cup R_{arg}$
2. $R_{self} \cap R_{arg} = \emptyset$
3. $R_{self} \cap (R_{in} \cup R_{out}) = \emptyset$
4. $R_{expl} = \mathit{regions}(\epsilon) \cup \bigcup_{j=1}^{p} \varphi_j^+(R_{arg})$, where $\mathit{regions}(\epsilon)$ is the set of RTVs appearing in the annotations of those subexpressions of ϵ of the form $C\ \overline{a_i}_{[\rho]}$ and $g\ \overline{a_i}_{[\overline{\rho_j}]}$, and $\varphi_j^+(\rho)$ is defined for every $j \in \{1..p\}$ as follows:

$$\varphi_j^+(\rho) = \begin{cases} \varphi_j(\rho) & \text{if } \rho \in \text{dom } \varphi_j \\ \rho & \text{if } \rho \notin \text{dom } \varphi_j \end{cases}$$

We use the notation $\varphi_j^+(R_{arg})$ to denote the set $\{\varphi_j(\rho) \mid \rho \in R_{arg}\}$.

The first one expresses that the region variables occurring in the final *Core-Safe* program (and hence belonging to R_{expl}) must be variables in scope. The types of region variables in scope belong either to R_{arg} or R_{self}, depending on whether they are additional parameters of the function definition, or the identifier *self*. The second and third restrictions are forced by the fact that ρ_{self} cannot appear in the

resulting type signature of the function, as the rule [FUNB] in Figure 3.9 demands. The last restriction specifies that the RTVs appearing in the annotations of ϵ are considered explicit. Moreover, if f is the function being inferred, this restriction also requires that the type of f extended with the region arguments in R_{arg} can produce type instances for typing all the recursive applications of f, each one extended with as many region arguments as the number of elements in R_{arg}.

In Section 4.5 we show that any sets R_{expl}, R_{arg} and R_{self} satisfying these restrictions lead to a region-annotated *Core-Safe* expression which is well-typed according to $\vdash_{Reg}$ rules. Notice that, provided the restriction (4) holds, an algorithm choosing any $R_{arg} \supseteq R_{expl}$ and $R_{self} = \emptyset$ would be correct with respect to this specification, but this solution would be very poor as, on the one hand, no construction would ever be done in the *self* region and, on the other hand, there might be region arguments never used. We look for an optimal solution in two senses: we want R_{arg} to be as small as possible, so that only those regions where data are built are given as arguments, but we also want R_{self} to be as big as possible, so that the maximum amount of memory is deallocated at function termination.

Our algorithm initially computes $R_{arg} = R_0 \cap (R_{in} \cup R_{out})$, by using the set R_0 of initial explicit RTVs, that is, $R_0 \stackrel{\text{def}}{=} regions(\epsilon)$. Then, it starts a fixpoint algorithm *computeRargFP* (Figure 4.7) trying to get the type of f's recursive applications as instances of the type of f extended with the current set R_{arg} of arguments. It may happen that the set R_{expl} may grow when considering different applications (see the Example 4.17). Adding more explicit RTVs to one recursive application will influence the type of the rest. As R_{arg} depends on R_{expl}, it may also grow. So, a fixpoint is used in order to obtain the final R_{arg} and R_{expl} from the initial ones. Due to our solution above, R_{arg} cannot grow greater than $R_{in} \cup R_{out}$, so termination of the fixpoint is guaranteed. We have the following invariant in the algorithm:

$$R_{arg} = R_{expl} \cap (R_{in} \cup R_{out}) \tag{4.2}$$

Once obtained the final R_{arg} and R_{expl}, the set R_{self} is computed as follows:

$$R_{self} = R_{expl} \setminus (R_{in} \cup R_{out}) \tag{4.3}$$

We show in Section 4.5 that these choices maximise the data allocated to the *self* region, which in turn maximises the amount of memory reclaimed at runtime when the corresponding function call finishes. With respect to the remaining DSs not being inferred to live in *self*, they will be allocated to the regions which are parameters to the function being called. It is the *caller* function's responsibility to determine where to put these DSs by passing the suitable arguments. Since the caller function is also inferred by the algorithm, the parameter assignment is done in such a way that the data allocated in the caller's *self* region is also maximized. From a global point of view, every cell not being created in the current topmost region (i.e. the region bound to the *self* identifier) will be created in the highest possible region and hence, will be deallocated at the earliest time allowed by the type system.

Example 4.7. Continuing with the Example 4.6 above, we start iterating with the following sets:

$$R_{in} = \{\rho_1, \rho_6\} \qquad R_{out} = \{\rho_4\} \qquad R_{arg} = \emptyset \qquad R_0 = \{\rho_4\}$$

After the first iteration, we get $R_{expl} = \{\rho_4\}$ and $R_{arg} = \{\rho_4\}$. A further iteration leads to the same solution, which is hence the fixpoint. Finally, we compute $R_{self} = \emptyset$ by using (4.3). This implies that nothing is going to be built in the *self* region during the evaluation of this function. From the RTVs in

R_{arg} we can extend the type signature of *append* as follows:

$$append :: [\alpha_3]@\rho_1 \to [\alpha_3]@\rho_6 \to \rho_4 \to [\alpha_3]@\rho_4$$

□

Example 4.8. Given the following definition:

$$f\ xs\ ys = \textbf{let}\ zs = (1 : xs)\ \textbf{in}\ f\ ys\ zs$$

After the HM inference phase, we obtain the preliminary type $[Int]@\rho_1 \to [Int]@\rho_2$ for f, the region substitution $\varphi_1 = [\rho_1 \mapsto \rho_2, \rho_2 \mapsto \rho_1]$ corresponding to the first recursive call, and the following intermediate definition:

$$f\ xs\ ys = \textbf{let}\ zs = (1 : xs)_{[\rho_1]}\ \textbf{in}\ f\ ys\ zs_{[1,[Int]@\rho_2 \to [Int]@\rho_1]}$$

The initial call to *computeRargFP* is done with the following sets:

$$R_{in} = \{\rho_1\} \qquad R_{out} = \{\rho_2\} \qquad R_{arg} = \emptyset \qquad R_0 = \{\rho_1\}$$

After the first iteration, R_{expl} and R_{arg} are both equal to $\{\rho_1\}$. However, since $\varphi_1(\rho_1) = \rho_2$, the RTV ρ_2 is added to both sets in the second iteration. By iterating a third time we obtain the same result, so we have the following fixed point:

$$R_{arg} = \{\rho_1, \rho_2\} \qquad R_{expl} = \{\rho_1, \rho_2\}$$

□

4.4.3 Annotating function definitions with region variables

Once we know the exact number of RTVs in R_{arg}, we create a fresh region parameter for each RTV in this set. Let us assume that $R_{arg} = \{\overline{\rho_j}^m\}$ and that $\overline{r_j}^m$ are the newly created region parameters. We define a mapping $\mathcal{R} : R_{arg} \cup R_{self} \to \textbf{RegVar}$ as follows:

$$\mathcal{R} \stackrel{\text{def}}{=} [\overline{\rho_j \mapsto r_j}^m] \uplus [\rho \mapsto self \mid \rho \in R_{self}]$$

The last step is to traverse the expression ϵ in order to transform its decorations into region variables, so that we get a *Core-Safe* expression as a result. This transformation is done by the *annotateExp* function, defined in Figure 4.8. Although not explicitly shown, we assume two extra parameters in this function that are propagated downwards to each recursive call: the R_{arg} and $\overline{\varphi_j}^p$ computed in the previous phases. Most cases in the definition of *annotateExp* are straightforward, since we only have to apply the $\mathcal{R}$ mapping to the decorations in ϵ in order to get the actual region annotations in the program. In the case of expressions of the form $f\ \overline{a_i}^n_{[k,\overline{s_i}^n \to s]}$, which correspond to recursive calls, the types of the region variables to be inserted may be distinct from the variables in R_{arg}, since we allow polymorphic recursion. Hence we proceed in two phases: firstly we obtain the concrete region instance correspoding to this call site (this is given by the φ_k^+), and then, we apply the $\mathcal{R}$ mapping which translates each RTV of this instance into a region variable.

$$
\begin{array}{llll}
\mathit{annotateExp} & \mathcal{R} & (c) & = \; c \\
\mathit{annotateExp} & \mathcal{R} & (x) & = \; x \\
\mathit{annotateExp} & \mathcal{R} & (x@_{[\rho]}) & = \; x \;@\; \mathcal{R}(\rho) \\
\mathit{annotateExp} & \mathcal{R} & (C\ \overline{a_i}^{n}_{[\rho]}) & = \; C\ \overline{a_i}^{n} \;@\; \mathcal{R}(\rho) \\
\mathit{annotateExp} & \mathcal{R} & (g\ \overline{a_i}^{n}_{[\overline{\rho_j}^{m}]}) & = \; g\ \overline{a_i}^{n} \;@\; \overline{\mathcal{R}(\rho_j)}^{m} \\
\mathit{annotateExp} & \mathcal{R} & (f\ \overline{a_i}^{n}_{[k,\overline{s_i}^{n}\to s]}) & = \; \mathit{annotateExp}\ \mathcal{R}\ (f\ \overline{a_i}^{n}_{[\overline{\varphi_k^{+}(\rho_j)}^{m}]}) \\
 & & & \quad \textbf{where}\ \{\overline{\rho_j}^{m}\} = R_{arg} \\
\mathit{annotateExp} & \mathcal{R} & (\textbf{let}\ x_1 = \epsilon_1\ \textbf{in}\ \epsilon_2) & = \; \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \\
 & & & \quad \textbf{where}\ e_1 = \mathit{annotateExp}\ \mathcal{R}\ \epsilon_1 \\
 & & & \qquad\ \ e_2 = \mathit{annotateExp}\ \mathcal{R}\ \epsilon_2 \\
\mathit{annotateExp} & \mathcal{R} & (\textbf{case}(!)\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to \epsilon_i}^{n}) & = \; \textbf{case}(!)\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} \\
 & & & \quad \textbf{where}\ \forall i:\ e_i = \mathit{annotateExp}\ \mathcal{R}\ \epsilon_i
\end{array}
$$

Figure 4.8: Definition of *annotateExp*.

Example 4.9. From the results of the Example 4.7 we build the mapping $\mathcal{R} = [\rho_4 \mapsto r]$ and obtain the following *Core-Safe* definition for *append*:

$$
\begin{array}{lll}
\mathit{append}\ xs\ ys\ @\ r & = & \textbf{case}\ xs\ \textbf{of} \\
 & & \quad [\,] \to ys\ @\ r \\
 & & \quad (x : xx) \to \textbf{let}\ x_1 = \mathit{append}\ xx\ ys\ \textbf{in}\ (x : x_1)@r
\end{array}
$$

□

Example 4.10. Given the definition of f in Example 4.8, we apply *annotateExp* with the mapping $\mathcal{R} = [\rho_1 \mapsto r_1, \rho_2 \mapsto r_2]$, with $R_{arg} = \{\rho_1, \rho_2\}$ and $\varphi_1 = [\rho_1 \mapsto \rho_2, \rho_2 \mapsto \rho_1]$:

$$
\begin{array}{ll}
 & \mathit{annotateExp}\ \mathcal{R}\ \left(\textbf{let}\ zs = (1 : xs)_{[\rho_1]}\ \textbf{in}\ f\ ys\ zs_{[1,[Int]@\rho_2\to[Int]@\rho_1]}\right) \\
= & \textbf{let}\ zs = \mathit{annotateExp}\ \mathcal{R}\ \left((1 : xs)_{[\rho_1]}\right)\ \textbf{in}\ \mathit{annotateExp}\ \mathcal{R}\ \left(f\ ys\ zs_{[1,[Int]@\rho_2\to[Int]@\rho_1]}\right) \\
= & \textbf{let}\ zs = (1 : xs)@r_1\ \textbf{in}\ \mathit{annotateExp}\ \mathcal{R}\ \left(f\ ys\ zs_{[\rho_2,\rho_1]}\right) \\
= & \textbf{let}\ zs = (1 : xs)@r_1\ \textbf{in}\ f\ ys\ zs\ @\ r_2\ r_1
\end{array}
$$

We obtain the following region-annotated definition:

$$
f\ xs\ ys\ @\ r_1\ r_2 = \textbf{let}\ zs = (1 : xs)@r_1\ \textbf{in}\ f\ ys\ zs\ @\ r_2\ r_1
$$

□

4.5 Correctness and optimality

The aim of this section is to prove that the final *Core-Safe* program resulting from the algorithm is typable with respect to the $\vdash_{Reg}$ rules in Figure 3.18. This fact can be established in three steps:

1. The HM inference algorithm described in Section 4.4.1 returns a *well-formed* intermediate expression $\epsilon \in \mathbf{Exp}_1$, provided the unification algorithm returns a correct solution to the generated constraints.

$$\frac{}{\Gamma \Vdash_{\overline{\varphi}} c : B}\ [\text{LIT}_{\text{HM}}] \qquad \frac{}{\Gamma + [x : s] \Vdash_{\overline{\varphi}} x : s}\ [\text{VAR}_{\text{HM}}]$$

$$\frac{\Gamma(x) = T\ \overline{s_i}^n @ \overline{\rho_j}^m \rho'}{\Gamma \Vdash_{\overline{\varphi}} x @_{[\rho]} : T\ \overline{s_i}^n @ \overline{\rho_j}^m \rho}\ [\text{COPY}_{\text{HM}}]$$

$$\frac{\overline{s_i}^n \to \rho \to s \trianglelefteq \Gamma(C) \qquad \forall i \in \{1 \ldots n\}.\Gamma \Vdash_{\overline{\varphi}} a_i : s_i}{\Gamma \Vdash_{\overline{\varphi}} C\ \overline{a_i}^n_{[\rho]} : s}\ [\text{CONS}_{\text{HM}}]$$

$$\frac{\varphi_k(\Gamma(f)) = \overline{s_i}^n \to s \qquad \forall i \in \{1 \ldots n\}.\Gamma \Vdash_{\overline{\varphi}} a_i : s_i}{\Gamma \Vdash_{\overline{\varphi}} f\ \overline{a_i}^n_{[k, \overline{s_i}^n \to s]} : s}\ [\text{APP-R}_{\text{HM}}]$$

$$\frac{\overline{s_i}^n \to \overline{\rho_j}^m \to s \trianglelefteq \sigma \qquad \forall i \in \{1 \ldots n\}.\Gamma \Vdash_{\overline{\varphi}} a_i : s_i}{\Gamma + [g : \sigma] \Vdash_{\overline{\varphi}} g\ \overline{a_i}^n_{[\overline{\rho_j}^m]} : s}\ [\text{APP-NR}_{\text{HM}}]$$

$$\frac{\Gamma \Vdash_{\overline{\varphi}} \epsilon_1 : s_1 \qquad \Gamma \backslash x_1 + [x_1 : s_1] \Vdash_{\overline{\varphi}} \epsilon_2 : s}{\Gamma \Vdash_{\overline{\varphi}} \textbf{let}\ x_1 = \epsilon_1\ \textbf{in}\ \epsilon_2 : s}\ [\text{LET}_{\text{HM}}]$$

$$\frac{\Gamma(x) = s_x \qquad \forall i \in \{1..n\} : \left\{ \begin{array}{l} \overline{s_{ij}}^{n_i} \to \rho \to s_x \trianglelefteq \Gamma(C_i) \\ \Gamma \backslash \overline{x_{ij}}^{n_i} + \left[\overline{x_{ij} : s_{ij}}^{n_i}\right] \Vdash_{\overline{\varphi}} \epsilon_i : s \end{array} \right.}{\Gamma \Vdash_{\overline{\varphi}} \textbf{case}(!)\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to \epsilon_i}^n : s}\ [\text{CASE(!)}_{\text{HM}}]$$

Figure 4.9: Rules for $\Vdash$.

2. Given a *well-formed* intermediate expression $\epsilon \in \mathbf{Exp}_1$ and three sets $(R_{arg}, R_{self}, R_{expl})$ such that the four conditions (1), (2), (3) and (4) of Section 4.4.2 hold, the result of *annotateExp* (as described in Section 4.4.3) is a well-typed *Core-Safe* expression.

3. The solution of $(R_{arg}, R_{self}, R_{expl})$ returned by the function *inferRegions* satisfies the above mentioned four conditions.

The link between these steps is not only the intermediate expression ϵ, but also the fact that it is well-formed. We define this notion by means of an intermediate type system, defined by the rules in Figure 4.9. This type system enforces that the annotations in ϵ do not contradict its type. Most rules are similar to those in Section 3.18 and hence they will not be explained here. The only detail worth noting is that there are two different $\Vdash_{\overline{\varphi}}$ rules for dealing with function applications, [APP-R$_{\text{HM}}$] and [APP-NR$_{\text{HM}}$], since recursive and nonrecursive function applications are decorated in different ways. In the case of recursive function applications, the functional type in the decoration is a region instance of the type bound to f within the environment Γ. The concrete instantiation is given by the corresponding φ_k substitution. The list of region substitutions $\overline{\varphi_i}^p$ (abbreviated in the rules as $\overline{\varphi}$) is propagated through the whole derivation.

Given these considerations, we are ready to prove the first step. The proof of the following Lemma is similar to that of Hindley-Milner inference. The main difference is that the $\Vdash_{\overline{\varphi}}$ typing rules involve types with regions.

Lemma 4.11. *For every Γ, ϵ_0, ϵ, α, f and E of their respective types, if $\Gamma \vdash_f \epsilon_0 \longrightarrow \epsilon : \alpha \mid E$ and $\langle \theta, \overline{\varphi_i}^p \rangle$ is a*

solution to E, we can derive the following judgement:

$$\theta(\Gamma) \Vdash_{\overline{\varphi_i}^p} \theta(\epsilon) : \theta(\alpha)$$

where $\theta(\Gamma)$ denotes the application of θ to every type in the range of Γ.

Proof. By induction on the structure of ϵ_0. We distinguish cases:

- **Case** $\epsilon_0 \equiv c$

  We get trivially $\theta(\Gamma) \Vdash_{\overline{\varphi}} c : \mathit{Int}$ or $\theta(\Gamma) \Vdash_{\overline{\varphi}} c : \mathit{Bool}$ depending on whether $c \in \mathbf{Int}$ or $c \in \mathbf{Bool}$.

- **Case** $\epsilon_0 \equiv x$

  We have to prove $\theta(\Gamma) \Vdash_{\overline{\varphi}} x : s$ where $s = \theta(\alpha)$. Since θ is a solution for the equation $\Gamma(x) = \alpha$ we get $s = \theta(\alpha) = \theta(\Gamma(x))$. In order to apply the rule $[\mathrm{VAR_{HM}}]$ we have to show that $x \in \mathrm{dom}(\theta(\Gamma))$, but this follows trivially from the fact that $x \in \mathrm{dom}(\Gamma)$.

- **Case** $\epsilon_0 \equiv x\,@$

  The resulting decorated expression is $\epsilon \equiv x@_{[\rho]}$, where ρ is a fresh region type variable. Since θ is the solution of the generated equation, we get $\theta(\alpha) \equiv T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m \theta(\rho)$. Besides this, since $x \in \mathrm{dom}(\Gamma)$, we get $x \in \mathrm{dom}(\theta(\Gamma))$ and hence $\theta(\Gamma(x)) \equiv T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m \rho'$ which is exactly the premise of the rule $[\mathrm{COPY_{HM}}]$. We can apply this rule in order to obtain $\theta(\Gamma) \Vdash_{\overline{\varphi}} x@_{[\theta(\rho)]} : \theta(\alpha)$.

- **Case** $\epsilon_0 \equiv C\ \overline{a_i}^n$

  In order to apply the rule $[\mathrm{CONS_{HM}}]$ so as to obtain $\theta(\Gamma) \Vdash_{\overline{\varphi}} C\ \overline{a_i}^n_{[\theta(\rho)]} : \theta(\alpha)$ we need to prove its premises: $\overline{\theta(\alpha_i)}^n \to \theta(\rho) \to \theta(\alpha) \trianglelefteq \theta(\Gamma(C))$ and $\forall i \in \{1 \ldots n\} : \theta(\Gamma) \Vdash a_i : \theta(\alpha_i)$. The former follows from the definition of θ being a solution, since $\theta(sf_0) = \overline{\theta(\alpha_i)}^n \to \theta(\rho) \to \theta(\alpha)$ and $\theta(sf_0) \trianglelefteq \Gamma(C)$. In fact, if sf_0 is an instance of $\Gamma(C)$, so it is $\theta(sf_0)$. Moreover, $\Gamma(C)$ does not contain any free variables, so $\theta(\Gamma(C)) = \Gamma(C)$. The latter premise is a direct consequence of applying the induction hypothesis to each a_i.

- **Case** $\epsilon_0 \equiv f\ \overline{a_i}^n$

  We have to find a $\Vdash_{\overline{\varphi}}$ typing for the decorated expression $f\ \overline{a_i}^n_{[k,\theta(\overline{\alpha_i}^n \to \alpha)]}$. Firstly, we can apply the induction hypothesis to each argument in order to get, for all $i \in \{1 \ldots n\}$, $\theta(\Gamma) \Vdash_{\overline{\varphi}} a_i : \theta(\alpha_i)$. Moreover, and because of $\langle \theta, \overline{\varphi_i}^p \rangle$ being a solution to E, we get:

  $$\varphi_k(\theta(\Gamma(f))) = \theta(\overline{\alpha_i}^n \to \alpha)$$

  which is the remaining premise that we need for applying the rule $[\text{APP-R}_{\mathrm{HM}}]$.

- **Case** $\epsilon_0 \equiv g\ \overline{a_i}^n$, with $g \neq f$

  The proof is similar to that of case $\epsilon_0 \equiv C\ \overline{a_i}^n$.

- **Case** $\epsilon_0 \equiv \mathbf{let}\ x_1 = \epsilon_{0,1}\ \mathbf{in}\ \epsilon_{0,2}$

  Since θ is a solution of E, it is also a solution of its components, namely E_1 and E_2. This allows us to apply the induction hypothesis on each subexpression with θ so that we get:

  $$\theta(\Gamma) \Vdash_f \theta(\epsilon_1) : \theta(\alpha_1) \qquad \theta(\Gamma) \backslash x_1 + [x_1 : \theta(\alpha_1)] \Vdash_{\overline{\varphi}} \theta(\epsilon_2) : \theta(\alpha_2)$$

Now we can apply the rule [LET$_{\text{HM}}$] by using these two judgements and the fact that $\theta(\alpha_2) = \theta(\alpha)$, since θ is a solution of $\{\alpha_2 = \alpha\}$.

- **Case** $\epsilon_0 \equiv \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to \epsilon_{0,i}}^{n}$

  Again, the general solution θ is also a solution for each particular E_i, so we apply the induction hypothesis to each $\epsilon_{0,i}$ in order to get $\theta(\Gamma)\backslash\overline{x_{ij}}^{n_i} + [\overline{x_{ij} : \theta(\alpha_{ij})}^{n_i}] \Vdash_{\overline{\varphi}} \theta(\epsilon_i) : \theta(\alpha_i)$, which is equivalent to $\theta(\Gamma)\backslash\overline{x_{ij}}^{n_i} + [\overline{x_{ij} : \theta(\alpha_{ij})}^{n_i}] \Vdash_{\overline{\varphi}} \theta(\epsilon_i) : \theta(\alpha)$ because of θ being a solution to $\{\alpha_i = \alpha\}$. Now we have to prove $\overline{\theta(\alpha_{ij})}^{n_i} \to \rho \to \theta(\alpha_x) \trianglelefteq \theta(\Gamma(C_i))$ for some ρ and for each $i \in \{1 \ldots n\}$. One of the premises of the rule [CASE$_{\text{Gen}}$] tell us that there exist some $\overline{s_{ij}}^{n}, \rho'$ and s_i such that $\overline{s_{ij}}^{n} \to \rho' \to s_i \trianglelefteq \Gamma(C_i)$. We take advantage of the equation $\{\overline{s_{ij}}^{n_i} \to s_i = \overline{\alpha_{ij}}^{n_i} \to \alpha_x\}$, so that we get $\overline{\theta(s_{ij})}^{n_i} = \overline{\theta(\alpha_{ij})}^{n_i}$ for each $j \in \{1 \ldots n_i\}$ and $\theta(s_i) = \theta(\alpha_x)$. Moreover, we know that ρ' belongs to the domain of θ, since the latter is defined for $\theta(s_i)$ and the outermost region of s_i is ρ'. Therefore, we get the required result as follows:

$$\overline{\theta(\alpha_{ij})}^{n_i} \to \theta(\rho') \to \theta(\alpha_x) = \overline{\theta(s_{ij})}^{n_i} \to \theta(\rho') \to \theta(s_i) \trianglelefteq \overline{s_{ij}}^{n_i} \to \rho' \to s_i \trianglelefteq \Gamma(C_i) = \theta(\Gamma(C_i))$$

□

Before embarking on the proof of the second step, we need an auxiliary definition which allows us to change the decorations of ϵ of the form $[k, \overline{s_i}^{n} \to s]$ by decorations of the form $[\overline{\rho_j}^{m}]$, appearing in non-recursive function applications. Of course, we can only do this when the set of additional region type parameters (R_{arg}) is known, since otherwise we could not build the type signature σ of the function. By substituting $[\varphi_k(R_{arg})]$ for each decoration $[k, \overline{s_i}^{n} \to s]$ in ϵ (as it is done indirectly in *annotateExp*), we obtain a expression ϵ' which is also typable via the $\Vdash_{\overline{\varphi}}$ rules.

Example 4.12. If ϵ is the expression at the end of Example 4.6, we substitute $[\varphi_1(\rho_4)] = [\rho_4]$ for the decoration $[1, [\alpha_3]@\rho_1 \to [a_3]@\rho_6 \to [a_3]@\rho_4]$ in order to obtain:

$$\begin{array}{rcl} \epsilon' & = & \mathbf{case}\ xs\ \mathbf{of} \\ & & \quad [\,] \to ys\ @_{[\rho_4]} \\ & & \quad (x : xx) \to \mathbf{let}\ x_1 = append\ xx\ ys_{[\rho_4]}\ \mathbf{in}\ (x : x_1)_{[\rho_4]} \end{array}$$

□

Example 4.13. If ϵ is the intermediate expression obtained in Example 4.8:

$$\epsilon = \mathbf{let}\ zs = (1 : xs)_{[\rho_1]}\ \mathbf{in}\ f\ ys\ zs_{[1,[Int]@\rho_2 \to [Int]@\rho_1]}$$

we get $R_{arg} = \{\rho_1, \rho_2\}$, so $\varphi_1(R_{arg}) = [\rho_2, \rho_1]$ and we transform ϵ as follows:

$$\epsilon' = \mathbf{let}\ zs = (1 : xs)_{[\rho_1]}\ \mathbf{in}\ f\ ys\ zs_{[\rho_2,\rho_1]}$$

□

Now we can prove the correctness of the second part of the algorithm, provided that the choice of R_{arg}, R_{self} and R_{expl} satisfies the conditions enumerated in Section 4.4.2:

Theorem 4.14. *Let us assume* $\Gamma + [f : \overline{s_i}^n \to s] \Vdash_{\overline{\varphi}} e : s'$ *where every annotation in* e *of the form* $[k, sf]$ *occur in a call to* f*. Let* R_{arg}*,* R_{self} *and* R_{expl} *sets of RTVs such that the conditions (1), (2) and (4) hold, where:*

$$\begin{array}{lcl} R_{in} & = & \{regions(s_i) \mid i \in \{1 \ldots n\}\} \\ R_{out} & = & \{regions(s)\} \\ R_{arg} & = & \{\rho_1, \ldots, \rho_m\}, \text{for some } \rho_1, \ldots, \rho_m \end{array}$$

and the e*,* $\overline{\varphi_i}^p$ *occurring in condition (4) are those of the* $\Vdash$ *derivation above. If we define* $\mathcal{R}$ *as follows:*

$$\mathcal{R} = [\overline{\rho_i \mapsto r_i}^m] \uplus [\rho \mapsto self \mid \rho \in R_{self}]$$

Then:

(a) $\mathcal{R}$ *is well-defined, i.e. the* $\uplus$ *operator is applied to region mappings with disjoint definition domains.*

(b) *There exists an* $e' \in \mathbf{Exp}_1$ *not containing annotations of the form* $[k, sf]$*, such that:*

1. $\Gamma + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] \Vdash_{\overline{\varphi}} e' : s'$
2. $R_{expl} = regions(e')$
3. $annotateExp\ \mathcal{R}\ e'\ \overline{\varphi_i}^p\ \{\overline{\rho_j}^m\} = annotateExp\ \mathcal{R}\ e\ \overline{\varphi_i}^p\ \{\overline{\rho_j}^m\}$

(c) *If* $e = annotateExp\ \mathcal{R}\ e\ \overline{\varphi_i}^p\ \{\overline{\rho_j}^m\}$*, then* e *is well-defined and the following judgment can be derived by using the* $\vdash_{Reg}$ *rules:*

$$\Gamma + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] \vdash_{Reg} e : s' \tag{4.4}$$

Proof. **(a)** follows trivially from condition (2). Now we prove that the expression e' that results from replacing every annotation of the form $[k, sf]$ by $[\overline{\varphi_k^+(\rho_j)}^m]$ satisfies the conditions stated in **(b)**:

1. The first condition can be proven by induction on the $\Vdash_{\overline{\varphi}}$ derivation applied to e. Most cases follow trivially from the induction hypothesis, except when the rule [APP-R$_{\text{HM}}$] is applied. In this case e has an annotation of the form $[k, sf]$. This implies, by hypothesis, that e is a call to f, and that sf has the form $\overline{s_i'}^n \to s'$. Our aim is to apply the [APP-NR$_{\text{HM}}$] rule to its counterpart in e', which is a call to f decorated with $[\overline{\varphi_k^+(\rho_j)}^m]$. We have to ensure that $\overline{s_i'}^n \to \overline{\varphi_k(\rho_j)}^m \to s'$ is an instance of $\forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s$, but this follows from the premises of the [APP-R$_{\text{HM}}$] rule:

$$\overline{s_i'}^n \to \overline{\varphi_k(\rho_j)}^m \to s' = \overline{\varphi_k(s_i)}^n \to \overline{\varphi_k^+(\rho_j)}^m \to \varphi_k(s) = \varphi_k^+(\overline{s_i}^n \to \overline{\rho_j}^m \to s) \trianglelefteq \overline{s_i}^n \to \overline{\rho_j}^m \to s$$

2. We know that $regions(e) \subseteq regions(e')$, since the annotations in e of the form $[\rho]$ and $[\overline{\rho_j}]$ are left untouched in e'. The only new RTVs in $regions(e')$ are those that arise from replacing the $[k, sf]$ annotations (which are not taken into account in the *regions* function) with their counterparts in e'. Therefore, we get:

$$regions(e') = regions(e) \cup \bigcup_{j=1}^{p} \{\varphi_j^+(\rho_i) \mid 1 \leq i \leq m\} = regions(e) \cup \bigcup_{j=1}^{p} \varphi_j^+(R_{arg})$$

   which is, by condition (4), equal to R_{expl}.

3. The last condition follows trivially from the definition of *annotateExp*.

Lastly we prove **(c)**. From the results given in **(b)** we can prove the existence of an ϵ' for which **(b)**.1 can be derived without using the [APP-R_{HM}] rule, since ϵ' does not contain annotations of the form $[k, sf]$. Hence we prove the following: given $\Gamma + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] \Vdash_{\overline{\varphi}} \epsilon' : s$ and the sets R_{arg}, R_{self} and R_{expl} such that the conditions (1), (2) and (4) hold, where R_{in} and R_{out} are defined as above, then we can derive (4.4) being $e = annotateExp\ \mathcal{R}\ \epsilon'\ \overline{\varphi_i}^p\ \{\overline{\rho_j}^m\}$, which we know that is equivalent to $annotateExp\ \mathcal{R}\ \epsilon\ \overline{\varphi_i}^p\ \{\overline{\rho_j}^m\}$.We apply structural induction on ϵ'. Let us distinguish cases according to the structure of ϵ':

- **Cases** $\epsilon' \equiv c, x$

  Let us compare the HM-rules that have been applied in the $\Vdash_f$ derivation with the rules we want to apply in order to derive (4.4):

  $$\frac{}{\Gamma \Vdash_{\overline{\varphi}} c : B}\ [\text{LIT}_{\text{HM}}] \qquad \frac{}{\Gamma \vdash_{Reg} c : B}\ [LIT_{Reg}]$$

  $$\frac{}{\Gamma + [x : s'] \Vdash_{\overline{\varphi}} x : s'}\ [\text{VAR}_{\text{HM}}] \qquad \frac{}{\Gamma + [x : s'] \vdash_{Reg} x : s'}\ [VAR_{Reg}]$$

  We can trivially obtain the judgements on the right-hand side from those on the left-hand side. We can extend the environment by applying Lemma 3.48 in order to get (4.4).

- **Case** $\epsilon' \equiv x@_{[\rho]}$

  $$\frac{\Gamma(x) = T@\rho'}{\Gamma \Vdash_{\overline{\varphi}} x@_{[\rho]} : T@\rho}\ [\text{COPY}_{\text{HM}}] \qquad \frac{}{\Gamma + [x : T@\rho', r : \rho] \vdash_{Reg} x@r : T@\rho}\ [COPY_{Reg}]$$

  Since $\rho \in R_{expl}$, by condition (1) we get $\rho \in R_{arg} \cup R_{self}$. We distinguish cases:

  $\rho \in R_{arg}$ Then ρ is one of the ρ_i ($i \in \{1 \dots m\}$), so that $e \equiv x\,@\,r_i$ and we can derive $\Gamma + [x : T@\rho, r_i : \rho_i] \vdash_{Reg} x\,@\,r_i$.

  $\rho \in R_{self}$ We obtain $e \equiv x\,@\,self$ and hence $\Gamma + [x : T@\rho, self : \rho_{self}] \vdash_{Reg} x\,@\,self$.

  In both cases we can extend the environment until we obtain (4.4).

- **Case** $\epsilon' \equiv C\ \overline{a_i}^q_{[\rho]}$

  The following rule has been applied in the $\Vdash_{\overline{\varphi}}$ derivation:

  $$\frac{\overline{s_i}^q \to \rho \to s' \trianglelefteq \Gamma(C) \qquad \forall i \in \{1 \dots q\} : \Gamma \Vdash_{\overline{\varphi}} a_i : s_i}{\Gamma \Vdash_{\overline{\varphi}} C\ \overline{a_i}^q_{[\rho]} : s'}\ [\text{CONS}_{\text{HM}}]$$

  We need to apply this rule to get the desired result:

  $$\frac{\overline{s_i}^q \to \rho \to s' \trianglelefteq \Gamma(C) \qquad \forall i \in \{1 \dots q\} : \Gamma \vdash_{Reg} a_i : s_i}{\Gamma + [r : \rho] \vdash_{Reg} C\ \overline{a_i}^q\ @\ r : s'}\ [CONS_{Reg}]$$

  We know that $annotateExp\ \mathcal{R}\ a_i = a_i$ for every a_i, so we can apply the induction hypothesis on each $a_i$in order to get:

  $$\Gamma + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] \vdash_{Reg} a_i : s_i \quad \text{for } i \in \{1 \dots n\}$$

Now we have to prove that the addition of $[r : \rho]$ to this environment (as required by rule $[CONS_{Reg}]$) results in a well-defined environment. Since $\rho \in R_{expl}$, we get $\rho \in R_{arg} \cup R_{self}$ by condition (1). Again, we distinguish cases:

$\rho \in R_{arg}$ Then $\rho = \rho_i$ for some $i \in \{1 \dots m\}$, so we get $e \equiv C\ \overline{a_i}^n\ @\ r_i$ as our final expression. Rule $[CONS_{Reg}]$ would require us to add the binding $[r_i : \rho_i]$.

$\rho \in R_{self}$ This case leads to $e \equiv C\ \overline{a_i}^n\ @\ self$, so we would have to add $[self : \rho_{self}]$.

In both cases the resulting environment is well-defined and we can apply $[CONS_{Reg}]$ in order to derive (4.4).

- **Case** $\epsilon' \equiv g\ \overline{a_i}^p_{[\overline{\rho'_j}^q]}$

$$\frac{\overline{s_i}^p \to \overline{\rho'_j}^q \to s' \trianglelefteq \sigma \qquad \forall i \in \{1 \dots p\} : \Gamma \Vdash_{\overline{\varphi}} a_i : s_i}{\Gamma + [g : \sigma] \Vdash_{\overline{\varphi}} g\ \overline{a_i}^p_{[\overline{\rho'_j}^q]} : s'} \ [\text{APP-NR}_{\text{HM}}]$$

$$\frac{\overline{s_i}^p \to \overline{\rho'_j}^q \to s' \trianglelefteq \sigma \qquad \forall i \in \{1 \dots p\} : \Gamma \vdash_{Reg} a_i : s_i}{\Gamma + [g : \sigma] + [\overline{r'_j : \rho'_j}^q] \vdash_{Reg} g\ \overline{a_i}^p\ @\ \overline{r'_j}^q : s'} \ [APP_{Reg}]$$

By applying the induction hypothesis in every subexpression we get, for every $i \in \{1 \dots p\}$:

$$\Gamma + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] \vdash_{Reg} a_i : s_i \tag{4.5}$$

If g is distinct from f, since g does not occur in any of the a_i, we can apply Lemma 3.48:

$$\Gamma + [g : \sigma] + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] \vdash_{Reg} a_i : s_i \tag{4.6}$$

The resulting *Core-Safe* expression is $g\ \overline{a_i}^p\ @\ \overline{\mathcal{R}(\rho'_j)}^q$. We denote by r'_j the result of each $\mathcal{R}(\rho'_j)$, as shown in the rule $[APP_{Reg}]$ above. Now we have to show, for each $j \in \{1 \dots q\}$, that the addition of the bindings $[r'_j : \rho'_j]$ does not contradict with the bindings $[\overline{r_i : \rho_i}^m]$. This is very similar to the cases for copy and constructor application: we have $\rho'_j \in R_{expl} \subseteq R_{arg} \cup R_{self}$. On the one hand, if $\rho'_j \in R_{arg}$ then ρ'_j is one of the ρ_i for $i \in \{1 \dots m\}$ and hence $r'_j = r_i$, so the binding $[r'_j : \rho'_j]$ is equivalent to $[r_i : \rho_i]$. On the other hand, if $\rho'_j \in R_{self}$ then $r'_j = self$ and we have the binding $[self : \rho_{self}]$ as a result. In any case, the binding $[r'_j : \rho'_j]$ is compatible with the typing environment above. So, we can apply rule $[APP_{Reg}]$ by using the judgments (4.5) or (4.6) as premises (depending on whether g is equal to f or not) so as to get:

$$\Gamma + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] \vdash_{Reg} g\ \overline{a_i}^p\ @\ \overline{r'_j}^q : s'$$

- **Case** $\epsilon' \equiv \textbf{let}\ x_1 = \epsilon_1\ \textbf{in}\ \epsilon_2$

$$\frac{\Gamma \Vdash_{\overline{\varphi}} \epsilon_1 : s_1 \qquad \Gamma \backslash x_1 + [x_1 : s_1] \Vdash_{\overline{\varphi}} \epsilon_2 : s'}{\Gamma \Vdash_{\overline{\varphi}} \textbf{let}\ x_1 = \epsilon_1\ \textbf{in}\ \epsilon_2 : s'} \ [\text{LET}_{\text{HM}}]$$

$$\frac{\Gamma \vdash_{Reg} e_1 : s_1 \qquad \Gamma \backslash x_1 + [x_1 : s_1] \vdash_{Reg} e_2 : s'}{\Gamma \vdash_{Reg} \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 : s'} \ [LET_{Reg}]$$

We apply the induction hypothesis to each subexpression ϵ_1, ϵ_2 in order to obtain:

$$\Gamma + [f : \forall \overline{\rho}.\overline{t_i}^n \to \overline{\rho_j}^m \to t] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] \vdash_{Reg} e_1 : s_1$$
$$\Gamma \backslash x_1 + [f : \forall \overline{\rho}.\overline{t_i}^n \to \overline{\rho_j}^m \to t] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] + [x_1 : s_1] \vdash_{Reg} e_2 : s'$$

where $e_1 = annotateExp\ \mathcal{R}\ \epsilon_1$ and $e_2 = annotateExp\ \mathcal{R}\ \epsilon_2$. We can apply the rule $[LET_{Reg}]$ in order to prove (4.4).

- **Case** $\epsilon' \equiv \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to \epsilon_i}^n$

  From the premises of the rule $[\text{CASE(!)}_{\text{HM}}]$, the judgment $\Gamma + [\overline{x_{ij} : s_{ij}}^{n_i}] \Vdash_{\overline{\varphi}} \epsilon_i : s'$ holds.

$$\frac{\Gamma(x) = s_x \qquad \forall i \in \{1..n\} : \begin{cases} \overline{s_{ij}}^{n_i} \to \rho \to s_x \trianglelefteq \Gamma(C_i) \\ \Gamma \backslash \overline{x_{ij}}^{n_i} + [\overline{x_{ij} : s_{ij}}^{n_i}] \Vdash_{\overline{\varphi}} \epsilon_i : s' \end{cases}}{\Gamma \Vdash_{\overline{\varphi}} \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to \epsilon_i}^n : s'} \ [\text{CASE(!)}_{\text{HM}}]$$

$$\frac{\begin{array}{c} \forall i \in \{1 \ldots r\} : \Gamma(C_i) = \sigma_i \qquad \forall i \in \{1 \ldots r\} : \overline{s_{ij}}^{n_i} \to \rho \to s_x \trianglelefteq \sigma_i \\ \Gamma(x) = s_x \qquad \forall i \in \{1 \ldots r\} : \Gamma \backslash \overline{x_{ij}}^{n_i} + [\overline{x_{ij} : s_{ij}}^{n_i}] \vdash_{Reg} e_i : s' \end{array}}{\Gamma \vdash_{Reg} \mathbf{case}(!)\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^r : s'} \ [CASE(!)_{Reg}]$$

  By applying the induction hypothesis to each ϵ_i, we get:

$$\Gamma \backslash \overline{x_{ij}}^{n_i} + [f : \forall \overline{\rho}.\overline{t_i}^n \to \overline{\rho_j}^m \to t] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] + [\overline{x_{ij} : s_{ij}}^{n_i}] \vdash_{Reg} e_i : s'$$

  for each $i \in \{1 \ldots r\}$, with $e_i = annotateExp\ \mathcal{R}\ \epsilon_i$. Therefore, by rule $[CASE(!)_{Reg}]$ we can prove (4.4), since the rest of the premises of this rule are the same as their counterparts in $[\text{CASE(!)}_{\text{HM}}]$.

□

Notice that the premises of this theorem do not include condition (3), since it is not necessary in the proof. This theorem guarantees the existence of a $\vdash_{Reg}$ typing for the function's body. In Chapter 5 we put forward an algorithm for dealing with explicit destruction via **case**(!), whose correctness proof guarantees the existence of a $\vdash_{Dst}$ typing. Both $\vdash_{Reg}$ and $\vdash_{Dst}$ typings lead to a $\vdash$ typing for the body of the function, as proven in Section 3.7. However, we need to satisfy an additional constraint in order to type the whole function definition by applying the rule $[FUN]$ of Figure 3.9. This rule specifies that the region type ρ_{self} must not occur in the resulting type signature. However, we can achieve this by assuming the condition (3):

Corollary 4.15. *Under the same conditions of Theorem 4.14 and assuming that (3) in Section 4.4.2 holds, we can derive the following judgement:*

$$\Gamma + [f : \forall \overline{\rho}.\overline{s_i}^n \to \overline{\rho_j}^m \to s] + [\overline{r_i : \rho_i}^m] + [self : \rho_{self}] \vdash_{Reg} e : s'$$

where $\rho_{self} \notin regions(\overline{s_i}^n \to \overline{\rho_j}^m \to s)$.

Proof. It follows trivially from Theorem 4.14 and condition (3). □

Our last step in the correctness proof establishes that the *inferRegions* algorithm returns a solution which satisfies the conditions (1) to (4). The main part of this algorithm is the fixpoint iteration performed by *computeRargFP*. The following invariant determines the origin of every variable occurring in

R_{arg}: either it belongs to the initial set of explicit regions, or it is the image of another variable already present in R_{arg} w.r.t some region mapping φ_i.

Lemma 4.16. *Given the call computeRargFP R_{in} R_{out} R_{arg} R_0 $\overline{\varphi_i}^p$, where $R_{arg} = \emptyset$, the following invariant holds in every subsequent iteration: for every $\rho \in R_{arg}$ there exists a sequence of RTVs $[\rho_0, \ldots, \rho_n] \subseteq R_{arg}$ with $n \geq 0$ and $\rho = \rho_n$ such that:*

1. $\rho_0 \in R_0$
2. For all ρ_i (with $i \in \{1 \ldots n\}$) there exists some φ_k such that $\varphi_k(\rho_{i-1}) = \rho_i$

Proof. The invariant holds trivially in the root call, since $R_{arg} = \emptyset$. Now let us assume that it holds for the input argument R_{arg}. We have to prove the invariant for the R'_{arg} appearing in the recursive call. Let $\rho \in R'_{arg}$. By inspecting the definition in Figure 4.7 we know that $\rho \in R'_{expl}$. So, either $\rho \in R_0$ (and the invariant holds considering the sequence $[\rho]$) or $\rho = \varphi_k(\rho')$ for some $k \in \{1 \ldots p\}$ and $\rho' \in R_{arg}$. In the latter case there exists a sequence $[\rho_0, \ldots \rho_n = \rho']$ which satisfies that required condition, and which we can extend as follows $[\rho_0, \ldots, \rho_n = \rho', \rho]$. □

Example 4.17. Given the following function fragment with a single recursive call:

$$f\ x_1\ x_2\ x_3 = \quad \ldots \quad f\ x_2\ x_3\ x_1 \quad \ldots$$

Let us assume that x_1 has type $[\alpha]@\rho_1$, x_2 has type $[\alpha]@\rho_2$ and x_3 has type $[\alpha]@\rho_3$. Therefore $R_{in} = \{\rho_1, \rho_2, \rho_3\}$. We compare the types of the function and its recursive call:

$$\begin{aligned} \text{Function:} \quad & f \ :: \ [\alpha]@\rho_1 \to [\alpha]@\rho_2 \to [\alpha]@\rho_3 \\ \text{Recursive application:} \quad & f \ :: \ [\alpha]@\rho_2 \to [\alpha]@\rho_3 \to [\alpha]@\rho_1 \end{aligned}$$

Hence we get $\varphi_1 = [\rho_1 \mapsto \rho_2, \rho_2 \mapsto \rho_3, \rho_3 \mapsto \rho_1]$. If $R_0 = \{\rho_1\}$ the fixpoint iteration takes the following steps:

	Initial call	1^{st} iteration	2^{nd} iteration	3^{rd} iteration
R_{arg}	$\emptyset$	$\{\rho_1\}$	$\{\rho_1, \rho_2\}$	$\{\rho_1, \rho_2, \rho_3\}$
R_{expl}	$\{\rho_1\}$	$\{\rho_1\}$	$\{\rho_1, \rho_2\}$	$\{\rho_1, \rho_2, \rho_3\}$

The variable ρ_1 belongs to R_{arg} because it appears initially in R_0. Variable ρ_2 is added to R_{arg} because $\varphi(\rho_1) = \rho_2$ and ρ_1 was already in R_{arg}. Variable ρ_3 is added to R_{arg} because $\varphi(\rho_2) = \rho_3$ and ρ_2 was already in R_{arg}. We have the following sequences for each variable:

$$\rho_1 : [\rho_1] \qquad \rho_2 : [\rho_1, \rho_2] \qquad \rho_3 : [\rho_1, \rho_2, \rho_3]$$

□

This invariant is useful for proving that the solution given by *inferRegions* is optimal with respect to the conditions (1) to (4) of Section 4.4.2. Any other solution satisfying these conditions may only lead to equal or worse results, i.e. a bigger R_{arg}, and hence a smaller R_{self}.

Theorem 4.18. *The algorithm inferRegions in Figure 4.7 returns three sets $(R_{arg}, R_{self}, R_{expl})$ which satisfy the four conditions described in Section 4.4.2. Moreover, for every solution $(R'_{arg}, R'_{self}, R'_{expl})$ satisfying these conditions it holds that $R_{arg} \subseteq R'_{arg}$ and $R_{expl} \subseteq R'_{expl}$.*

Proof. The condition (4) follows trivially from the definition of *computeRargFP*. We can rewrite the remaining conditions by using (4.2) and (4.3):

1. $R_{expl} \subseteq (R_{expl} \backslash (R_{in} \cup R_{out})) \cup (R_{expl} \cap (R_{in} \cup R_{out})$
2. $(R_{expl} \backslash (R_{in} \cup R_{out})) \cap (R_{expl} \cap (R_{in} \cup R_{out}) = \emptyset$
3. $(R_{expl} \backslash (R_{in} \cup R_{out})) \cap (R_{in} \cup R_{out}) = \emptyset$

The three immediately follow from set algebra. Now we prove optimality: let us assume that the four conditions hold for $(R'_{arg}, R'_{self}, R'_{expl})$. First we prove that $R_{arg} \subseteq R'_{arg}$. If $\rho \in R_{arg}$ then there exists (by Lemma 4.16) a sequence $[\rho_0, \ldots, \rho_n = \rho] \subseteq R_{arg}$ with $\rho_0 \in regions(\epsilon)$. Notice that, by the invariant (4.2), all these RTVs must belong to $R_{in} \cup R_{out}$. We show that every ρ_i belongs to R'_{arg} by induction on i.

- **Case** $i = 0$. In this case $\rho_0 \in regions(\epsilon) \subseteq R'_{expl}$. Since $\rho_0 \in R_{in} \cup R_{out}$, then $\rho_0 \notin R'_{self}$ because of condition (3). Hence, by condition (1), we get $\rho_0 \in R'_{arg}$.
- **Case** $i > 0$. In this case $\rho_i = \varphi_k(\rho_{i-1})$ for some k. By induction hypothesis $\rho_{i-1} \in R'_{arg}$ and hence $\rho_i \in R'_{expl}$ by condition (4). Again, ρ_i cannot belong to R'_{self} because it belongs to $R_{in} \cup R_{out}$ and condition (3) forces them to be disjoint. Hence, by condition (1), $\rho_i \in R'_{arg}$.

Therefore, $\rho = \rho_n \in R'_{arg}$ and $R_{arg} \subseteq R'_{arg}$. From this inclusion and the condition (4) it follows that $R_{expl} \subseteq R'_{expl}$.

□

This theorem proves that the resulting R_{arg} is the minimum solution for the conditions (1) to (4). Given a fixed set R_{expl} of explicit variables, it follows from conditions (1) and (2) of Section 4.4.2 that R_{self} is the maximum solution.

4.6 Case studies

In this section we study the results of the region inference algorithm applied to several examples, ranging from simple functions on lists to a compiler running in several phases. Although the algorithm is defined at *Core-Safe* level, all the examples in this section, as well as their region-annotated versions, are written in *Full-Safe* for conciseness.

Example 4.19 (List partition)**.** As a first example, consider the following function definition:

$$
\begin{aligned}
&\mathit{partition}\ y\ [\,] = ([\,], [\,])\\
&\mathit{partition}\ y\ (x : xs)\\
&\qquad\qquad \mid x \leq y = (x : ls, gs)\\
&\qquad\qquad \mid x > y = (ls, x : gs)\\
&\qquad\qquad \textbf{where}\ (ls, gs) = \mathit{partition}\ y\ xs
\end{aligned}
$$

The HM inference phase yields the following decorated program:

$$
\begin{aligned}
&\mathit{partition}\ y\ [\,] = ([\,]_{[\rho_2]}, [\,]_{[\rho_3]})_{[\rho_4]}\\
&\mathit{partition}\ y\ (x : xs)\\
&\qquad\qquad \mid x \leq y = ((x : ls)_{[\rho_2]}, gs)_{[\rho_4]}\\
&\qquad\qquad \mid x > y = (ls, (x : gs)_{[\rho_3]})_{[\rho_4]}\\
&\qquad\qquad \textbf{where}\ (ls, gs) = \mathit{partition}\ y\ xs_{[1, Int \to [Int]@\rho_1 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_9]}
\end{aligned}
$$

and the type $partition :: Int \to [Int]@\rho_1 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4$. By comparing this type with the type of the recursive call we get the following mapping:

$$\varphi_1 = [\rho_1 \mapsto \rho_1, \rho_2 \mapsto \rho_2, \rho_3 \mapsto \rho_3, \rho_4 \mapsto \rho_9]$$

Notice that, in this case, we have polymorphic recusion over the RTV of the pair which is given as a result. We start the fixpoint iteration with these sets:

$$R_{in} = \{\rho_1\} \qquad R_{out} = \{\rho_2, \rho_3, \rho_4\} \qquad R_{arg} = \emptyset \qquad R_0 = \{\rho_2, \rho_3, \rho_4\}$$

After the first iteration, we get:

$$R_{expl} = \{\rho_2, \rho_3, \rho_4\} \qquad R_{arg} = \{\rho_2, \rho_3, \rho_4\}$$

In the next iteration the set R_{expl} grows, since $\varphi_1(\rho_4) = \rho_9$:

$$R_{expl} = \{\rho_2, \rho_3, \rho_4, \rho_9\} \qquad R_{arg} = \{\rho_2, \rho_3, \rho_4\}$$

However, since R_{arg} does not change, we stop the fixpoint iteration and compute $R_{self} = \{\rho_9\}$ by applying (4.3). Finally, we generate fresh region variables r_2, r_3 and r_4 corresponding to the RTVs in R_{arg} and build the following mapping:

$$\mathcal{R} = [\rho_2 \mapsto r_2, \rho_3 \mapsto r_3, \rho_4 \mapsto r_4, \rho_9 \mapsto self]$$

which leads to the following annotated function definition:

$$\begin{array}{l} partition :: Int \to [Int]@\rho_1 \to \rho_2 \to \rho_3 \to \rho_4 \to ([Int]@\rho_2, [Int]@\rho_3)@\rho_4 \\ partition\ y\ [\,]\ @\ r_2\ r_3\ r_4 = ([\,]@r_2, [\,]@r_3)@r_4 \\ partition\ y\ (x : xs)\ @\ r_2\ r_3\ r_4 \\ \qquad \mid x \leq y = ((x : ls)\ @\ r_2, gs)\ @\ r_4 \\ \qquad \mid x > y = (ls, (x : gs)\ @\ r_3)\ @\ r_4 \\ \qquad \textbf{where}\ (ls, gs) = partition\ y\ xs\ @\ r_2\ r_3\ self \end{array}$$

In absence of polymorphic recursion, the recursive call would be $partition\ y\ xs\ @\ r_2\ r_3\ r_4$, which implies that all the tuples returned from the internal calls to *partition* are built in the output region, even when these tuples are temporary and not part of the function's result. □

Example 4.20 (Pascal's triangle)**.** Let us consider the dynamic approach for computing binomial coefficients by using Pascal's triangle. We start from the unit list $[1]$, which is the 0-th row of the triangle. If $[x_0, x_1, \ldots x_{i-1}]$ are the elements of the i-th row, the elements of the $i+1$-th row are given by the list $[1, x_0 + x_1, x_1 + x_2, \ldots, x_{i-1} + x_i, x_i]$. This is shown in Figure 4.10. The binomial coefficient $\binom{n}{m}$ can be obtained from the m-th element in the n-th row of the triangle.

The function *sumList* computes the $i+1$-th row of the triangle from its i-th row. We show directly the intermediate definition:

$$\begin{array}{lcl} sumList\ (x : [\,]) & = & (x : [\,]_{[\rho_2]})_{[\rho_2]} \\ sumList\ (x : xs) & = & ((x + y) : sumList\ xs_{[1, [Int]@\rho_1 \to [Int]@\rho_2]})_{[\rho_2]} \quad \textbf{where}\ (y : \_) = xs \end{array}$$

The fixpoint iteration produces $R_{expl} = R_{arg} = \{\rho_2\}$ and $R_{self} = \emptyset$, which leads to the following

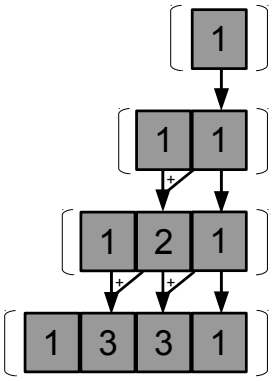


Figure 4.10: Pascal's triangle computation.

definition:

$$\begin{array}{lcl} sumList\ (x : [\,])\ @\ r & = & (x : [\,])\ @\ r \\ sumList\ (x : xs)\ @\ r & = & ((x + y) : sumList\ xs\ @\ r)\ @\ r \quad \textbf{where}\ (y : \_) = xs \end{array}$$

Function *pascal* iterates over the initial list in order to get the row, whose number is given as parameter. Below we show the decorated definition:

$$\begin{array}{lcl} pascal\ 0 & = & (1 : [\,]_{[\rho_1]})_{[\rho_1]} \\ pascal\ n & = & (1 : (sumList\ (pascal\ (n-1))_{[1,Int\to[Int]@\rho_2]})_{[\rho_1]})_{[\rho_1]} \end{array}$$

The type inferred for *pascal* after the HM phase is $Int \to [Int]@\rho_1$. By comparing this type with that of the recursive call we get $\varphi_1 = [\rho_1 \mapsto \rho_2]$. Hence $R_{in} = \emptyset$, $R_{out} = \{\rho_1\}$ and $R_0 = \{\rho_1\}$, which gives us an initial $R_{expl} = R_{arg} = \{\rho_1\}$. After the second iteration ρ_2 is now made explicit and it is added to R_{expl}, which now contains the region variables $\{\rho_1, \rho_2\}$. However, R_{arg} stays the same and hence the fixpoint has been reached. Finally, we get $R_{self} = \{\rho_2\}$ and the program is annotated as follows:

$$\begin{array}{lcl} pascal & :: & Int \to \rho_1 \to [Int]@\rho_1 \\ pascal\ 0\ @\ r & = & (1 : [\,]\ @\ r)@\ r \\ pascal\ n\ @\ r & = & (1 : sumList\ (pascal\ (n-1)\ @\ self)\ @\ r)\ @\ r \end{array}$$

The resulting list from the recursive call to *pascal* will be destroyed once the calling function finishes. Hence a function call *pascal n* has a cost of $\mathcal{O}(n)$ in space. Without polymorphic recursion the result of every recursive call would be built in the output region r, which would imply a cost of $\mathcal{O}(n^2)$ in space (Figure 4.11).

The function definition in charge of computing the binomial coefficient $\binom{n}{m}$ is defined as follows:

$$combNumber\ n\ m = pascal\ n\ !!\ m$$

The !! operator returns the m-th element of the list returned by *pascal*, which is inferred as residing in *self*, since is not part of the function's result. Hence the annotated program is:

$$combNumber\ n\ m = (pascal\ n\ @\ self)\ !!\ m$$

□

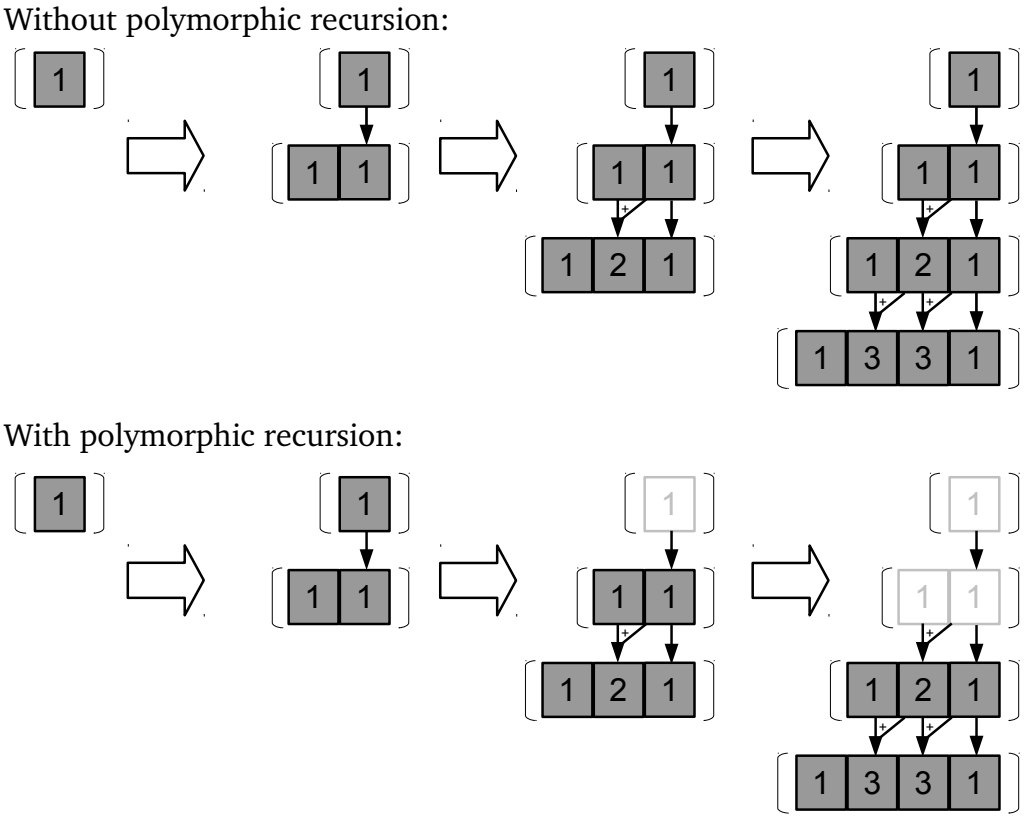


Figure 4.11: Lists being created by the function *pascal* with and without polymorphic recursion. Each column corresponds to the end of a recursive call. White cells are destroyed.

$$
\begin{array}{l}
\textbf{data}\ \mathit{BSTree}\ \alpha\ @\ \rho = \mathit{Empty}\ @\ \rho \mid \mathit{Node}\ (\mathit{BSTree}\ \alpha\ @\ \rho)\ \alpha\ (\mathit{BSTree}\ \alpha\ @\ \rho)\ @\ \rho \\
\\
\mathit{insertT} :: \mathit{Int} \to \mathit{BSTree}\ \mathit{Int}\ @\ \rho_1 \to \rho_1 \to \mathit{BSTree}\ \mathit{Int}\ @\ \rho_1 \\
\mathit{insertT}\ y\ \mathit{Empty}\ @\ r_1 = \mathit{Node}\ (\mathit{Empty}\ @\ r_1)\ y\ (\mathit{Empty}\ @\ r_1)\ @\ r_1 \\
\mathit{insertT}\ y\ (\mathit{Node}\ l\ x\ r)\ @\ r_1 \\
\qquad \mid\ x == y \quad = \mathit{Node}\ l\ x\ r\ @\ r_1 \\
\qquad \mid\ y < x \quad\ = \mathit{Node}\ (\mathit{insertT}\ y\ l\ @\ r_1)\ x\ r\ @\ r_1 \\
\qquad \mid\ y > x \quad\ = \mathit{Node}\ l\ x\ (\mathit{insertT}\ y\ r\ @\ r_1)\ @\ r_1 \\
\\
\mathit{mkTree} :: [\mathit{Int}]@\rho_1 \to \rho_2 \to \mathit{BSTree}\ \mathit{Int}\ @\ \rho_2 \\
\mathit{mkTree}\ [\,]\ @\ r_2 = \mathit{Empty}\ @\ r_2 \\
\mathit{mkTree}\ (x : xs)\ @\ r_2 = \mathit{insertT}\ x\ (\mathit{mkTree}\ xs\ @\ r_2)\ @\ r_2 \\
\\
\mathit{inorder} :: \mathit{BSTree}\ \alpha\ @\ \rho_1 \to \rho_2 \to [\alpha]@\rho_2 \\
\mathit{inorder}\ \mathit{Empty}\ @\ r_2 = [\,]\ @\ r_2 \\
\mathit{inorder}\ (\mathit{Node}\ l\ x\ r)\ @\ r_2 \\
\qquad = \mathit{append}\ (\mathit{inorder}\ l\ @\ \mathit{self})\ (\mathit{append}\ ((x : [\,]@\mathit{self})@\mathit{self})\ (\mathit{inorder}\ r\ @\ r_2)\ @\ r_2)\ @\ r_2 \\
\\
\mathit{treeSort} :: [\mathit{Int}]@\rho_1 \to \rho_2 \to [\mathit{Int}]@\rho_2 \\
\mathit{treeSort}\ xs\ @\ r_2 = \mathit{inorder}\ (\mathit{mkTree}\ xs\ @\ \mathit{self})\ @\ r_2
\end{array}
$$

Figure 4.12: Region inference for the treesort algorithm.

Example 4.21 (Treesort)**.** Let us consider an implementation of the treesort algorithm. Figure 4.12 shows the resulting region-annotated program. Firstly, we define a data type *BSTree* for binary search trees and define a function *insertT* implementing the usual insertion in an ordered tree. From the type signature of *insertT* we deduce that both input and output trees must live in the same region at runtime, since we reuse children of the input tree when constructing the output. The function *mkTree* builds a binary search tree by successive insertions of the elements of a list, whereas *inorder* transforms a tree into an ordered list by means of an inorder traversal. Notice that, in the latter, we use the definition of *append* in which the list passed as second parameter is shared with the result, unlike the definition in Example 4.4. That is why the second recursive call to *inorder* is called with the output region parameter; its result will be shared with the output. The result of the first recursive call is done with the working region, because its result will be copied when being concatenated with the global result. Finally, the function *treeSort* calls these two functions above. The intermediate tree is not needed in the result, so it is built in the working region. □

Region-based memory management is a powerful approach when dealing with programs that work in several phases and produce intermediate data structures between them. Compilers are a typical example of these kind of programs.

Example 4.22. Given a simple imperative **while** language, we want to translate it into bytecode for a *P*-machine. Both source and target languages are defined in Figure 4.13. Notice that some data types are polymorphic on a type variable α. This variable is used as a placeholder for decorations, as we will see below. The translation performs in four phases:

- **Type checking:** In this language we have two types, integers and booleans. This process decorates the AST with type information and checks whether the arguments given to the operators are of a suitable type.

$$\mathit{typeCheck} :: \mathit{Stm}\ \alpha \to \mathit{Stm}\ (\mathit{Maybe}\ \mathit{Type})$$

```
data BinArithOp  =  Add | Sub | Mul | Div                          – binary arithmetic operators
data UnArithOp   =  Neg                                            – unary arithmetic operators
data RelOp       =  Less | Eq                                      – relational operators
data BinBoolOp   =  And | Or                                       – binary logical operators
data UnBoolOp    =  Not                                            – unary logical operators

data Exp α       =  Const Int α                                    – constant
                 |  Var Int α                                      – variable
                 |  AppBinArithOp BinArithOp (Exp α) (Exp α) α     – operator application
                 |  AppUnArithOp UnArithOp (Exp α) α
                 |  AppRelOp RelOp (Exp α) (Exp α) α
                 |  AppBinBoolOp BinBoolOp (Exp α) (Exp α) α
                 |  AppUnBoolOp UnBoolOp (Exp α) α

data Stm α       =  Skip α                                         – does nothing
                 |  Assign Int (Exp α) α                           – variable assignment
                 |  Seq (Stm α) (Stm α) α                          – sequentiation
                 |  If (Exp α) (Stm α) (Stm α) α                   – conditional
                 |  While (Exp α) (Stm α) α                        – loop

data PInst α     =  Ldc Int | Load | Store | Jmp α | JFalse α      – bytecode instructions
                 |  AddP | SubP | MulP | DivP | NegP
                 |  LtP | GtP | EqP | AndP | OrP | NotP
```

Figure 4.13: Source and target language definitions for Example 4.22.

- **Constant folding:** It computes at compile time those operations which involve only constants.

$$constantFold :: Stm\ \alpha \to Stm\ \alpha$$

- **Translation into bytecode:** This phase generates the bytecode with symbolic labels. Branching instructions refer to these labels.

$$translate :: Stm\ \alpha \to [PInst\ Label]$$

- **Patching:** It translates symbolic labels into natural numbers, as well as the arguments of the branching instructions.

$$patch :: [PInst\ Label] \to [PInst\ Int]$$

For the sake of simplicity we do not show the code of these functions here. In order to take advantage of regions, we need some functions *runToZ*, which executes the phase Z and the preceeding ones.

$$\begin{aligned} runToConstantFold\ stm &= \textbf{let}\ stm' = typeCheck\ stm\ \textbf{in}\ constantFold\ stm' \\ runToTranslate\ stm &= \textbf{let}\ stm' = runToConstantFold\ stm\ \textbf{in}\ translate\ stm' \\ runToPatch\ stm &= \textbf{let}\ insts = runToTranslate\ stm\ \textbf{in}\ patch\ insts \end{aligned}$$

We show in Figure 4.14 the number of extra region parameters inferred for data types and function definitions. A more qualitative analysis of the results is depicted in Figure 4.15. Rows represent the working regions corresponding to each phase, while the columns stand for the execution of each phase. The pictograms represent data structures being built in a given region during the execution of a phase.

Data type	Region types
BinArithOp	1
UnArithOp	1
RelOp	1
BinBoolOp	1
UnBoolOp	1
Exp	6
Stm	19
PInst	1

Function	Region parameters
typeCheck	10
constantFold	4
translate	8
patch	2

Figure 4.14: Quantitative results for the compiler example.

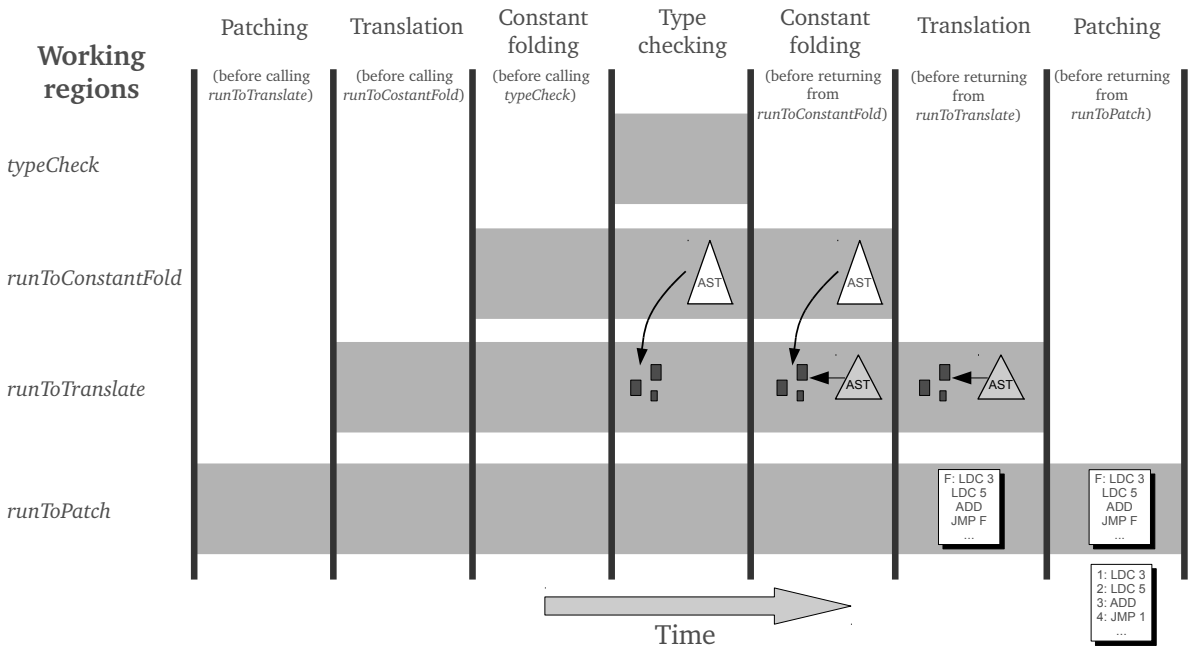


Figure 4.15: Evolution of the heap in each phase. Each gray rectangle represents the working region of a phase, its lifetime and the data structures that are stored into it. Each column stands for the state of the heap at a given time.

The type checking phase creates an AST with typing information. The result is distributed between the working regions of *constantFold* and *translate*, so that those in the latter region can be shared in subsequent phases. The constant folding phase generates another AST which reuses those parts lying in the *translate* working region. The initial AST with types is disposed after the execution of *constantFold*. The next phase creates the list of bytecode instructions in the *patch* working region. After this, the AST is no longer needed and hence its memory is reclaimed. Finally, the patching phase creates the final list of bytecode instructions without symbolic labels. The caller function decides where to locate this list by passing the suitable arguments to *patch*. Notice that, during the whole process, we only have two intermediate data structures simultaneously in the heap: the input and the output of each phase. □

4.7 Conclusions and related work

We have introduced an algorithm for annotating expressions with region variables. The algorithm is correct and optimal with respect the $\vdash_{Reg}$ typing rules. It has a time cost in $\mathcal{O}(n^2)$ in the worst case, and $\mathcal{O}(n)$ in the average case. The completeness of the algorithm can only be guaranteed when the function being analysed has a most general type, according to the $\vdash_{Reg}$ type system. If such a type does not exist, the algorithm fails. The lack of principal types is due to copy expressions. For instance, the function definition $f\ xs\ @\ r = xs\ @\ r$ accepts the types $[Int]@\rho_1 \to \rho_2 \to [Int]@\rho_2$ and $BSTree\ \alpha\ @\ \rho_1 \to \rho_2 \to BSTree\ \alpha\ @\ \rho_2$, but both of them are instances of the type $\alpha \to \rho_2 \to \alpha$, which is not accepted by the $\vdash_{Reg}$ type system, since the $[COPY_{Reg}]$ rule demands an algebraic type in the DS being copied. This lack of principal types can be solved by incorporating a restricted form of polymorphic variable in which the outermost RTV is fixed. For instance, the type $\alpha\ @\ \rho$ stands for an algebraic type whose outermost region is ρ. In this case, the f function would have $\alpha\ @\ \rho_1 \to \rho_2 \to \alpha\ @\ \rho_2$ as its most general type. As a short-term future work we plan to study the integration of this extension.

The pioneer work on region inference is that of Tofte, Talpin, and their colleagues on the MLKit compiler [112] (in what follows, TT). They address a more general problem than we do, since their language is higher-order. The TT algorithm has two phases, respectively called $\mathcal{S}$ and $\mathcal{R}$. The $\mathcal{S}$ algorithm just generates fresh region variables for values and introduces the lexical scope of the regions by using a **letregion** construct. The $\mathcal{R}$ algorithm is responsible for assigning types to recursive functions. It deals with region-polymorphic recursion, and computes a fixed point. Both algorithms use higher-order unification based on directed graphs and on a *UNION-FIND* data structure. If n is the size of the HM-typed term being inferred, the $\mathcal{S}$ algorithm runs in $\mathcal{O}(n^3)$ time, and the $\mathcal{R}$ algorithm in $\mathcal{O}(n^4)$ time in the worst case. The total cost is in $\mathcal{O}(n^4)$. The meaning of a typed expression **letregion** ρ **in** $e : \mu$ is that region ρ does not occur free in type μ, so it can be deallocated upon the evaluation of e. Our algorithm has some resemblances with this part of the inference, in the sense that we decide to unify with ρ_{self} all the region variables not occurring in the result type of a function. They do not claim their algorithm to be optimal, but, in fact, they create as many regions as possible, trying to make local *all* the regions not needed in the final value. Incompleteness in TT comes as a consequence of polymorphic recursion. In our case, it is a consequence of the copy expressions.

The implementation of MLKit in 1998 added a third phase, called *storage mode analysis*, which introduced a *region resetting* action previously to some allocations. In order to take profit of this analysis, the programmer has to introduce *copy* expressions in specific parts of the text. A further work [2] added a constraint and control flow-based analysis after the TT inference. These resulted in modifying the text by delaying as much as possible some region allocations, and by bringing forward some deallocations, without compromising pointer safety. Again, the programmer needs to introduce the *copy* function in

appropriate places, in order to avoid memory leaks. From the programmer's point-of-view, it is far from trivial there to place these *copy* functions. In *Safe*, the programmer can avoid such leaks via explicit deallocation, or by augmenting the number of nested calls in order to make the program more "region-friendly", as it was done in the compiler of Example 4.22.

One problem reported in [114] is that most of the regions inferred in the first versions of the algorithm contained a single value, so region management produced a big overhead at runtime. Later, they added a new analysis to collapse all these regions into a single one local to the invocation (allocated in the stack). In our case, having a single local region *self* per function invocation does not seem to us to be a big drawback if function bodies are small enough. We believe that region-polymorphic recursion has a much bigger impact in avoiding memory leaks than multiplicity of local regions. So, we claim that the results of our algorithm are comparable to those of TT for first-order programs.

A radical deviation from these approaches is [54], which introduces a type system in which region life-times are not necessarily nested. The compiler annotates the program with region variables and supports operations for allocation, releasing, aliasing and renaming. A reference-counting analysis is used in order to decide when a released region should be deallocated. The language is first-order. The inference algorithm [75] can be defined as a global abstract interpretation of the program by following the control flow of the functions in a backwards direction. Although the authors do not give either asymptotic costs or actual benchmarks, it can be deduced that this cost could grow more than quadratically with the program text size in the worst case, as a global fixed point must be computed and a region variable may disappear at each iteration. This lack of modularity could make the approach impractical for large programs.

The main virtue of our design is its simplicity. This comes at the cost of obtaining less granularity, since the above mentioned systems allow to create several nested regions in the same function call. Also, in the case of TT and its derivatives, they support higher-order functions. As a consequence, the inference algorithms are more complex and costly. The techniques shown in this chapter could also be applied to any first-order functional language featuring Hindley-Milner types, since it can be implemented as a slight extension of Hindley-Milner inference. In the near future we plan to extend *Safe* to support higher-order functions and mutually recursive data structures (see Chapter 8). We expect high difficulties in other aspects of the language such as extending dangling pointers safety analyses or memory bounds inference, but not so many to extend the region inference algorithm presented here. However, it is still open whether a higher-order variant of this algorithm could achieve a cost better than the $\mathcal{O}(n^4)$ got by Tofte and Talpin.

Chapter 5

Safe types inference

5.1 Introduction

In this chapter we continue addressing the problem of type reconstruction for the type system of Chapter 3. While the algorithm of the last chapter computes a $\vdash_{Reg}$ derivation for a given program, in this chapter we aim to design an algorithm for finding $\vdash_{Dst}$ derivations. We are no longer concerned about the regions in which a DS lives, but only about whether this DS is safe, condemned, or in-danger. Unlike the region inference algorithm, which annotates the source program with region variables, the algorithm of this chapter leaves the source program untouched: it only checks whether the destructive pattern matching, specified by the programmer, is done in a safe way. A program being accepted by both algorithms is guaranteed to be pointer safe, as explained in Chapter 3.

The inference of safe, condemned, and in-danger marks (*mark inference*, in the following) for finding a $\vdash_{Dst}$ typing derivation is not a trivial task: the $\vdash_{Dst}$ rules are not syntax-directed, and there may be several ways in which a rule can be applied. For instance, in a **let** expression, there are many ways to split the typing environment into the two environments typing its corresponding sub-expressions. The idea of the mark inference algorithm is to do a bottom-up traversal of the abstract syntax tree of the expression being analysed. As this traversal is done, those variables in the scope of the expression being analysed are classified into the set of safe, condemned, and in-danger variables. If more than one mark is applicable to a given variable, the algorithm gives precedence to safe types.

This paper extends the work of [81], whose contribution is a mark inference algorithm for the type system of [84]. Since in Chapter 3 we have introduced an improved version of the latter, in this Chapter we improve the mark inference algorithm of [81] in order to deal with the type system of this thesis. The main advantages of the algorithm of this chapter with respect to that of [81] are the following:

- It is more efficient. Given a function definition of n parameters, if s denotes the size of its abstract syntax tree, the worst-case time complexity of the algorithm in this chapter is $\mathcal{O}(ns^2)$, whereas the algorithm of [81] runs in $\mathcal{O}(ns^3)$ time.

- It is complete. The $\vdash_{Dst}$ rules admit minimal mark signatures for function definitions, and the algorithm is able to find these minimal signatures.

- It is simpler. It consists in a single bottom-up traversal of the abstract syntax tree, whereas the algorithm of [81] combines bottom-up and top-down traversals, which makes its correctness proof considerably more difficult.

In Section 5.2 we explain the inference algorithm in detail. We start from the mark inference of isolated expressions. Then we continue with the inference of function definitions, and, finally, we describe the inference of *Core-Safe* programs. Section 5.3 applies the algorithm to several case studies. In Section 5.4 we prove the algorithm correct and complete with respect to the $\vdash_{Dst}$ rules. Finally, Section 5.5 concludes.

5.2 Mark inference algorithm

The inference algorithm is modular; each function definition is inferred separately. For each function of n parameters a mark signature $\overline{m_i}^n \rightarrow \mathtt{s}$ is inferred. Since the body of a function definition may also contain function applications, there exists an environment Σ containing the mark signatures of the already inferred functions. The inference of a program starts with an empty environment, and adds the mark signature of the function definitions as they are processed by the algorithm.

The inference of a function definition consists in a bottom-up traversal of its abstract syntax tree. For each sub-expression e, the algorithm tries to find an environment Γ such that the judgement $\Gamma \vdash_{Dst} e : \mathtt{s}$ is derivable. The Γ environment must contain the marks of every free variable of e, so the algorithm assigns a mark to the variables it comes across and stores that mark in the mark environment of the sub-expression being processed. There are several situations in which more than one mark may be applicable. For instance, if the expression being inferred is a variable x, the environment could contain the binding $[x : \mathtt{s}]$, $[x : \mathtt{d}]$, or $[x : \mathtt{r}]$, since x is typable under all of these environments. In a similar way, the discriminant of a **case**! may get a `d` or an `r` mark in the context of that expression. In these cases, the algorithm assigns the lowest mark with respect the following order: $\mathtt{s} \leq \mathtt{d} \leq \mathtt{r}$. Thus, in the case of the variable x, it would get a safe mark, whereas in the case of the destructive **case**!, the discriminant would get a condemned mark. The rationale behind this decision is that the lowest mark is the less restrictive when considering the expression e as a sub-expression in a higher context. Let us explain this with an example:

Example 5.1. Given the following *Core-Safe* fragment:

$$\textbf{let } y = x \textbf{ in case } x \textbf{ of } \ldots$$

The x variable in the bound expression may get a safe, condemned, or in-danger mark. However, if it got a condemned or in-danger mark, the **let** expression would not be typeable with the $[LET_{Dst}]$ rule, since there would be a variable with an unsafe mark in the bound expression of a **let** occurring free in its main expression. □

An additional advantage of assigning the lowest possible mark at a given context, is the fact that the $[WEAK_{Dst}]$ rule allows the algorithm to rise that mark in a higher context, when necessary, but there is no rule in the type system for lowering marks.

5.2.1 Inference rules for expressions

The main part of the algorithm is a set of syntax-directed rules that allows us to derive judgements of the form $e \vdash (R, D, S)$, where R, D and S are sets of variables. These sets specify which variables get an in-danger (`r`), condemned (`d`) and safe (`s`) mark in the *minimal* Γ environment typing (w.r.t. the $\vdash_{Dst}$ rules) the expression e. Every free variable in e occurs in one of these three sets, and their union is a subset of the variables in $scope(e)$. The inference rules define the (R, D, S) sets associated with an expression in

$$\frac{}{c \vdash (\emptyset,\emptyset,\emptyset)}\ [LIT_I] \qquad \frac{}{x \vdash (\emptyset,\emptyset,\{x\})}\ [VAR_I] \qquad \frac{}{x\,@\,r \vdash (\emptyset,\emptyset,\{x\})}\ [COPY_I]$$

$$\frac{}{C\ \overline{a_i}^n\ @\ r \vdash (\emptyset,\emptyset,\{\overline{a_i}^n\})}\ [CONS_I]$$

$$\frac{\begin{array}{ll}\Sigma(f) = \overline{m_i}^n \to \mathsf{s} & R \cap S = \emptyset \\ S = \{a_i \mid m_i = \mathsf{s}\} & R \cap D = \emptyset \\ R = \bigcup_{m_i=\mathsf{d}} sharerec(a_i, f\ \overline{a_i}^n\ @\ \overline{r_j}^m) \backslash \{a_i\} & S \cap D = \emptyset \\ \forall i \in \{1..n\}.D_i = \begin{cases}\{a_i\} & \text{if } m_i = \mathsf{d} \\ \emptyset & \text{otherwise}\end{cases} & \forall i,j \in \{1..n\}.i \neq j \Rightarrow D_i \cap D_j = \emptyset \\ D = \bigcup_{i=1}^n D_i & \forall x \in D.isTree(x)\end{array}}{f\ \overline{a_i}^n\ @\ \overline{r_j}^m \vdash (R,D,S)}\ [APP_I]$$

$$\frac{\begin{array}{ll}e_1 \vdash (R_1,D_1,S_1) & (R,D,S) = (R_1,D_1,S_1) \sqcup ((R_2,D_2,S_2)\backslash\{x_1\}) \\ e_2 \vdash (R_2,D_2,S_2) & (R_1 \cup D_1) \cap fv(e_2) = \emptyset\end{array}}{\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2 \vdash (R,D,S)}\ [LET_I]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.P_i = \{\overline{x_{ij}}^n\} \qquad \forall i \in \{1..n\}.e_i \vdash (R_i,D_i,S_i) \\ (R,D,S) = (\bigsqcup_{i=1}^n ((R_i,D_i,S_i)\backslash P_i)) \sqcup (\emptyset,\emptyset,\{x\})\end{array}}{\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash (R,D,S)}\ [CASE_I]$$

$$\frac{\begin{array}{ll}\forall i \in \{1..n\}.e_i \vdash (R_i,D_i,S_i) & \\ \forall i \in \{1..n\}.P_i = \{\overline{x_{ij}}^n\} & (R,D,S) = \bigsqcup_{i=1}^n ((R_i,D_i,S_i)\backslash P_i) \\ \forall i \in \{1..n\}.Rec_i = \{x_{ij} \mid j \in RecPos(C_i)\} & (R',D',S') = ((R,D,S)\backslash\{x\}) \sqcup (R_{SHR},\emptyset,\emptyset) \\ \forall i \in \{1..n\}.(P_i \backslash Rec_i) \cap (D_i \cup R_i) = \emptyset & R_{SHR} = sharerec(x, \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) \backslash \{x\} \\ \forall i \in \{1..n\}.Rec_i \cap R_i = \emptyset & \forall i \in \{1..n\}.(R_{SHR} \cup \{x\}) \cap fv(e_i) = \emptyset\end{array}}{\mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash (R',D' \cup \{x\},S')}\ [CASE!_I]$$

Figure 5.1: Inference rules for expressions.

terms of the triples obtained for each sub-expression. Thus these rules can be implemented by means of a bottom-up traversal of the abstract syntax tree of the expression. We assume the existence of a mark environment $\Sigma : \mathbf{Fun} \to \mathbf{MarkSig}$ that it is implicitly propagated through the $\vdash$ rules, and contains the mark signatures of the functions being called from the corresponding expression. In case of ambiguity, we shall make this signature explicit with the notation $e \vdash_\Sigma (R, D, S)$.

Figure 5.1 shows the inference rules. Rules $[VAR_I]$, $[COPY_I]$, and $[CONS_I]$ assign an s mark to the variables in the corresponding expression. By abuse of notation, we assume that the set $\{\overline{a_i}^n\}$ only contains the atoms a_i that are not literals. In the $[APP_I]$ rule, the mark signature of the function being called determines the marks of the arguments. Those occurring in safe positions get a safe mark, and those occurring in condemned positions get a condemned mark. The disjointness condition between the different D_i sets, together with the condition $S \cap D = \emptyset$ mimic the well-definedness property of the $\oplus$ operator in the typing rules: the same variable can occur in two positions only if both are safe. The conditions $R \cap D = \emptyset$ and $R \cap S = \emptyset$ correspond to the disjoint union of the Γ_R environment with the $\bigoplus_{i=1}^n [a_i : m_i]$ environment.

In the $[LET_I]$, the triples of both auxiliary and main expressions e_1 and e_2 are inferred. The notation $(R, D, S)\backslash M$ abbreviates the triple $(R\backslash M, D\backslash M, S\backslash M)$. The $\sqcup$ operator, which is analogous to the $\sqcup$ operator on typing environments, is defined as follows:

$$(R_1, D_1, S_1) \sqcup (R_2, D_2, S_2) = (R_1 \cup R_2, (D_1 \cup D_2)\backslash(R_1 \cup R_2), (S_1 \cup S_2)\backslash(D_1 \cup D_2 \cup R_1 \cup R_2))$$

It is easy to prove that the $\sqcup$ operator is commutative and associative. The generalization of this operator to n triples can be expressed as follows:

$$\bigsqcup_{i=1}^{n}(R_i, D_i, S_i) = (\bigcup_{i=1}^{n} R_i, \bigcup_{i=1}^{n} D_i\backslash(\bigcup_{i=1}^{n} R_i), \bigcup_{i=1}^{n} S_i\backslash(\bigcup_{i=1}^{n} R_i \cup \bigcup_{i=1}^{n} D_i)) \tag{5.1}$$

The side condition $(R_1 \cup D_1) \cap fv(e_2) = \emptyset$ is equivalent to the side condition of $[LET_{Dst}]$. It ensures that no variable is mentioned in the main expression if a part of it has been destroyed previously.

In the $[CASE_I]$ and $[CASE!_I]$ rules we infer the triple of each alternative separately. This implies that the same variable may get different marks in different **case** branches. The $\sqcup$ operator ensures that this variable gets in the whole **case**(!) expression the lowest (most general) mark common to all the e_i. In terms of typing rules, it represents the least common environment (excluding pattern variables) typing every branch. The least upper bound with $(\emptyset, \emptyset, \{x\})$ obeys the fact that every free variable must belong to one of the sets of condemned, in-danger and safe marks. Thus it is assigned a safe (s) mark if it does not occur in any triple (R_i, D_i, S_i). The definition of R_{SHR} in $[CASE!_I]$, and its associated side condition, are analogous to the R set occurring in $[CASE!_{Dst}]$. The remaining side conditions in $[CASE!_I]$ mimic the *inh* predicate. On the one hand, $(P_i\backslash Rec_i) \cap (D_i \cup R_i) = \emptyset$ demands those pattern variables occurring in non-recursive positions to have a safe mark, or no mark at all, in every branch. On the other hand, $Rec_i \cap R_i = \emptyset$ specifies that, in every branch, recursive pattern variables must have a safe or condemned mark, or remain unmarked in every branch.

5.2.2 Inference of function definitions

When inferring non-recursive function definitions we only have to apply the rules of Figure 5.1 to the body of the function once. We assume that functions being called from the current one have already been inferred, and their mark signatures are already available in Σ. Once we obtain the triple (R, D, S)

$$\mathit{inferMarksDef}\ \Sigma\ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f) = \mathit{inferMarksFP}\ (\Sigma \uplus [f \mapsto (\overline{\mathsf{s}}^n \to \mathsf{s})])\ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f)$$

$$\begin{array}{lll}
\mathit{inferMarksFP}\ \Sigma\ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f) & & \\
\quad \mid\ R \neq \emptyset & = & \mathit{error} \\
\quad \mid\ R = \emptyset \wedge \Sigma(f) = \overline{m_i}^n \to \mathsf{s} & = & \Sigma \\
\quad \mid\ R = \emptyset \wedge \Sigma(f) \neq \overline{m_i}^n \to \mathsf{s} & = & \mathit{inferMarksFP}\ ((\Sigma \backslash f) \uplus [f \mapsto (\overline{m_i}^n \to \mathsf{s})])\ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f) \\
\quad \textbf{where} \quad e_f \vdash_\Sigma (R, D, S) & & \\
\qquad \forall i \in \{1..n\}.m_i = \begin{cases} \mathsf{d} & \text{if } x_i \in D \\ \mathsf{s} & \text{if } x_i \notin D \end{cases} & &
\end{array}$$

Figure 5.2: Inference algorithm applied to function definitions.

for the function, we can reconstruct easily the mark signature of the function as follows: if there is a parameter in the R set, the function is not typeable. If a parameter belongs to the D set (resp. the S set) , it gets a condemned mark d (resp. safe mark s). If it does not belong to any of the three sets, the parameter may have a safe or a condemned mark. Since our goal is to get the lowest type, it is sensible to assign a safe mark s to these parameters.

The inference of recursive function definitions is slightly more involved, because we do not know in advance which mark signature should be assigned to the function being inferred. Since we aim for the most general mark signature, we start assuming that every parameter of the function has a safe mark and apply the rules of Figure 5.1. If every parameter turns out to be safe, our assumption was correct and we return the corresponding updated signature. If some parameter gets a condemned mark, our assumption was wrong and we have to update the signature Σ according to the triple (R, D, S) obtained for the function's body. Parameters belonging to D now get a condemned mark, and those in S (or not occurring in any of the sets) get a safe mark. With these new assumptions (reflected in the updated signature Σ) we apply the inference rules again. We iterate this process until a fixpoint is reached, that is, the obtained triple (R, D, S) corresponds to the assumption under which it is inferred.

These ideas are formalized in the algorithm of Figure 5.2. The *inferMarks* function is given a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ and a signature environment Σ containing the mark signatures of all the functions being called from f (except f itself). This function does the first call to *inferMarksFP* with the initial assumption, in which every parameter has a safe mark. The *inferMarksFP* function does the fixed point iteration. In every iteration, it applies the $\vdash$ rules of Figure 5.1 and obtains a triple (R, D, S). If there is an element in R, then it is one of the parameters, since R only contains variables in scope. In this case, the inference algorithm returns an error, since only safe and condemned types are allowed in function signatures (see Section 3.3 for a discussion on this). If R is empty, we build a mark signature $\overline{m_i}^n \to \mathsf{s}$ which assigns a d mark to those parameters in D and an s mark to those parameters not occurring in D. If the new mark signature matches the one already existing in Σ, a fixed point has been reached and we return Σ. If it does not match, the function updates the signature with the new information and starts another iteration.

Example 5.2. Let us consider the *reverseD* function that reverses the list given as input. It uses an auxiliary function *revAuxD* that destroys its first parameter, while accumulating the result in its second

$$\Sigma = [revAuxD \mapsto (\mathsf{s} \to \mathsf{s} \to \mathsf{s})]$$

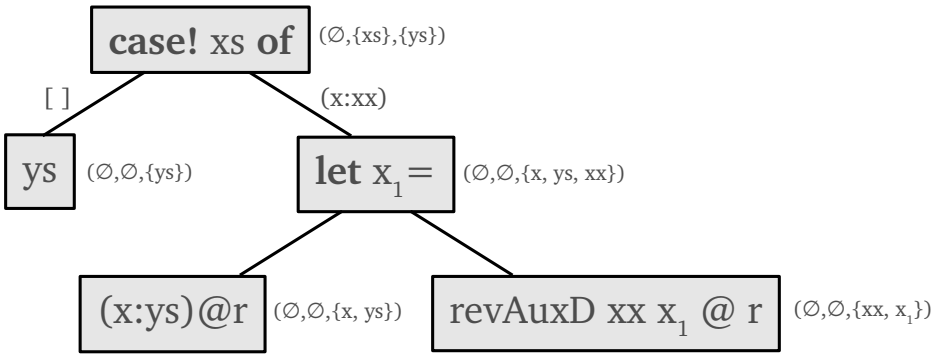


Figure 5.3: Sets inferred after the first iteration of *revAuxD*. The (R, D, S) triple computed for each subexpression is shown to the right of its gray box.

$$\Sigma = [revAuxD \mapsto (\mathsf{d} \to \mathsf{s} \to \mathsf{s})]$$

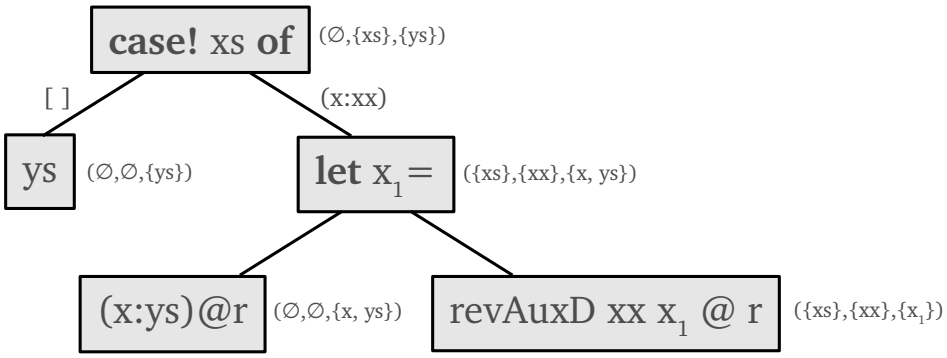


Figure 5.4: Sets inferred after the second iteration of *revAuxD*.

parameter.

$$\begin{aligned} &revAuxD\ xs\ ys\ @\ r \\ &\quad = \mathbf{case}!\ xs\ \mathbf{of} \\ &\qquad [\,] \to ys \\ &\qquad (x : xx) \to \mathbf{let}\ x_1 = (x : ys)@r\ \mathbf{in}\ revAuxD\ xx\ x_1\ @\ r \end{aligned}$$

$$reverseD\ xs\ @\ r = \mathbf{let}\ x_1 = [\,]\,@\,r\ \mathbf{in}\ revAuxD\ xs\ x_1\ @\ r$$

We start applying the inference rules to the body of *revAuxD*. We assume that the mark environment Σ contains the following signature for *revAuxD*: $\mathsf{s} \to \mathsf{s} \to \mathsf{s}$. The results inferred for each sub-expression are shown in Figure 5.3. Notice that xx and x_1 are assigned safe marks, as a consequence of the initial environment Σ assigning a safe type to every parameter. At the end of the iteration we obtain the triple $(\emptyset, \{xs\}, \{ys\})$ for the whole body of *revAuxD*, which corresponds to the signature $\mathsf{d} \to \mathsf{s} \to \mathsf{s}$. Since this signature is different from the one initially assumed, we update the signature environment with the new signature, and apply again the inference rules. The result is shown in Figure 5.4. A difference with respect to the first iteration is the triple of sets inferred in the recursive call: $(\{xs\}, \{xx\}, \{x_1\})$. The xx variable is condemned, since it occurs in a condemned position. Since xs always points to this

$$\Sigma = \left[\begin{array}{lll} \mathit{revAuxD} & \mapsto & (\mathtt{d} \to \mathtt{s} \to \mathtt{s}) \\ \mathit{reverse} & \mapsto & (\mathtt{s} \to \mathtt{s}) \end{array} \right]$$

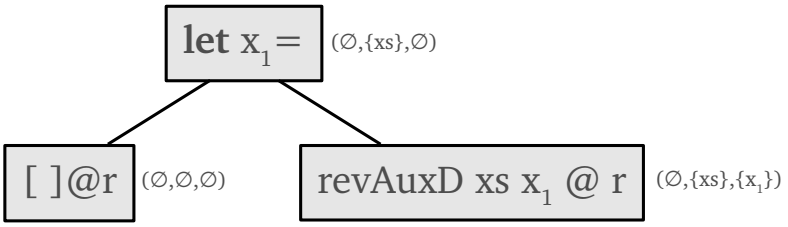


Figure 5.5: Sets inferred after the first iteration of *reverse*.

$$\begin{array}{l} \mathit{inferMarksProg}\ (\overline{\mathit{data}_i}; \overline{\mathit{def}_i}; e) = \textbf{if}\ e \vdash_{\Sigma'} (\emptyset,\emptyset,\emptyset)\ \textbf{then}\ \mathit{OK}\ \textbf{else}\ \mathit{error} \\ \quad \textbf{where}\ \Sigma' = \mathit{foldl}\ \mathit{inferMarksDef}\ [\,]\ \overline{\mathit{def}_i} \end{array}$$

Figure 5.6: Inference algorithm applied to *Core-Safe* programs.

variable, it must occur in $\mathit{sharerec}(xx, \mathit{revAuxD}\ xx\ x_1\ @\ r)$, and is assigned an in-danger mark. However, this mark is turned into a condemned mark as a consequence of the **case**! in the outermost expression. As a result we obtain, again, the mark signature $\mathtt{d} \to \mathtt{s} \to \mathtt{s}$, so we have reached a fixed point. The algorithm stores the binding $[\mathit{revAuxD} \mapsto (\mathtt{d} \to \mathtt{s} \to \mathtt{s})]$ in the signature environment Σ and proceeds with the following definition in the program.

Figure 5.5 shows the (R, D, S) tuples inferred by the algorithm for each sub-expression of *reverse*, assuming $(\mathtt{s} \to \mathtt{s})$ as its mark signature. From the inference rules we get that *xs* gets a condemned mark, which results in the signature $(\mathtt{d} \to \mathtt{s})$. In the second iteration it turns out we have reached a fixed point. The last iteration is unnecessary, as, in this case, the signature of $\Sigma(\mathit{reverse})$ is not relevant to the resulting (R, D, S) tuples. This happens, in general, when inferring the mark signatures of non-recursive function definitions. □

5.2.3 Inference of a *Core-Safe* program

The *inferMarksProg* function shown in Figure 5.6 receives a *Core-Safe* program, and determines whether it is well-typed with respect to the $\vdash_{Dst}$ rules. Firstly, it builds a signature environment Σ' in an incremental way, as function definitions are analysed by the *inferMarksDef* function. After this, it checks whether the judgement $e \vdash (\emptyset,\emptyset,\emptyset)$ is derivable under this environment Σ'. The three components of the resulting triple must be empty sets, since in the context of the main expression of a program there are no variables in scope.

5.3 Case studies

In this section we show how the inference algorithm behaves with several case studies. For the sake of simplicity, we show only the *Full-Safe* code of the programs being analysed, and we show only the results of the algorithm, without any mention to the derivation of the $\vdash$ judgements.

Example 5.3. A *quadtree* is a tree data structure in which each node has four children. It has two different

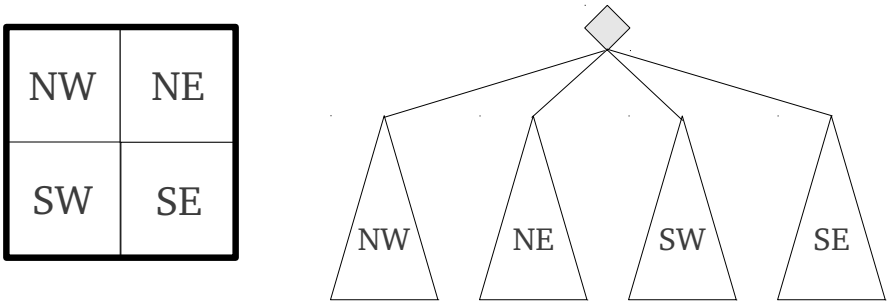


Figure 5.7: Quadtree representation of a bitmap. The whole area is divided into for quadrants. Each quadrant is represented by its own quadtree.

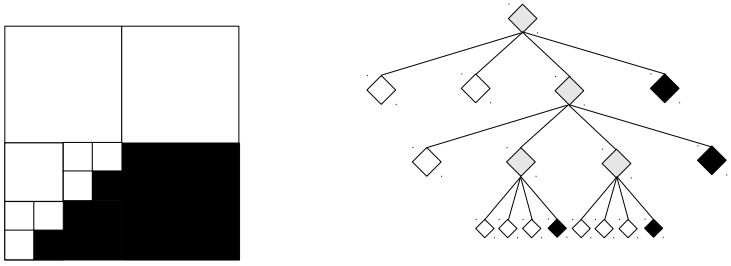


Figure 5.8: Bitmap and its representation as a quadtree.

kinds of leaves: black leaves and white leaves. It is defined as follows:

$$\textbf{data}\ \mathit{QuadTree} = \mathit{Black} \mid \mathit{White} \mid \mathit{Node}\ \mathit{QuadTree}\ \mathit{QuadTree}\ \mathit{QuadTree}\ \mathit{QuadTree}$$

In some situations, quadtrees provide an efficient representation of bitmaps by following a divide-and-conquer strategy. If the bitmap contains only black pixels, it is represented by *Black*. If it contains only white pixels, it is represented by *White*. Otherwise, we divide the bitmap into four quadrants, each of which is represented by its corresponding quadtree. The bitmap is then represented by the *Node* constructor applied to the bitmaps of these quadrants (Figure 5.7). Figure 5.8 shows an example of a bitmap and its quadtree representation.

The *rotateD* function performs a ninety-degree counterclockwise rotation of the bitmap given as parameter (see Figure 5.9). The original bitmap is destroyed, so this function does not need additional heap space:

$$\begin{aligned}
&\mathit{rotateD}\ \mathit{Black}! = \mathit{Black}\\
&\mathit{rotateD}\ \mathit{White}! = \mathit{White}\\
&\mathit{rotateD}\ (\mathit{Node}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se})! =\\
&\qquad \mathit{Node}\ (\mathit{rotateD}\ \mathit{ne})\ (\mathit{rotateD}\ \mathit{se})\ (\mathit{rotateD}\ \mathit{nw})\ (\mathit{rotateD}\ \mathit{sw})
\end{aligned}$$

This function is successfully typed by the algorithm, which infers the following mark signature for *rotateD*: $\mathtt{d} \rightarrow \mathtt{s}$. The *flipHD* function, which mirrors the input image horizontally, works in a similar

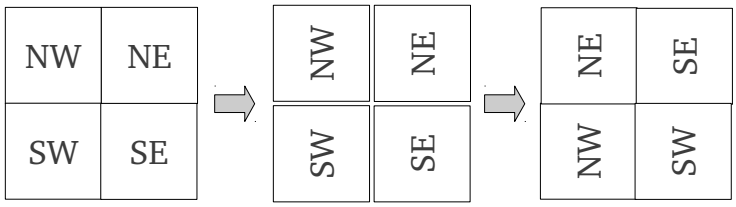


Figure 5.9: Rotation of a bitmap. Firstly, each piece is rotated individually. Then, a global rotation of the four pieces is done.

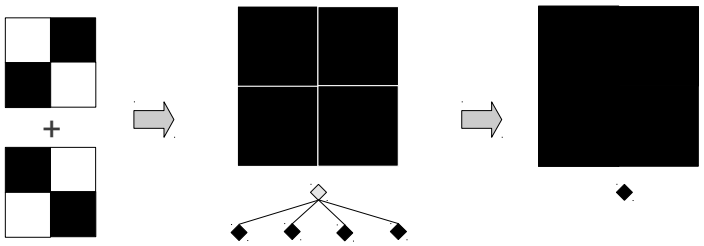


Figure 5.10: When overlying two bitmaps, a simplification of the quadtree may be needed.

fashion:

$$
\begin{aligned}
&\mathit{flipHD}\ \mathit{Black}! = \mathit{Black}\\
&\mathit{flipHD}\ \mathit{White}! = \mathit{White}\\
&\mathit{flipHD}\ (\mathit{Node}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se})! =\\
&\qquad \mathit{Node}\ (\mathit{flipHD}\ \mathit{ne})\ (\mathit{flipHD}\ \mathit{nw})\ (\mathit{flipHD}\ \mathit{se})\ (\mathit{flipHD}\ \mathit{sw})
\end{aligned}
$$

The inference algorithm infers the signature $\mathtt{d} \rightarrow \mathtt{s}$ for this function. Finally, we define a function *overlayD* which merges the black pixels of the two bitmaps given as parameters. Both input pictures are destroyed.

$$
\begin{aligned}
&\mathit{overlayD}\ \mathit{White}!\ q! = q\\
&\mathit{overlayD}\ \mathit{Black}!\ q! = \mathit{Black}\\
&\mathit{overlayD}\ (\mathit{Node}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se})!\ \mathit{White}! = \mathit{Node}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se}\\
&\mathit{overlayD}\ (\mathit{Node}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se})!\ \mathit{Black}! = \mathit{Black}\\
&\mathit{overlayD}\ (\mathit{Node}\ \mathit{nw}_1\ \mathit{ne}_1\ \mathit{sw}_1\ \mathit{se}_1)!\ (\mathit{Node}\ \mathit{nw}_2\ \mathit{ne}_2\ \mathit{sw}_2\ \mathit{se}_2)!\\
&\qquad = \mathit{Node}\ (\mathit{overlayD}\ \mathit{nw}_1\ \mathit{nw}_2)\ (\mathit{overlayD}\ \mathit{ne}_1\ \mathit{ne}_2)\ (\mathit{overlayD}\ \mathit{sw}_1\ \mathit{sw}_2)\ (\mathit{overlayD}\ \mathit{se}_1\ \mathit{se}_2)
\end{aligned}
$$

However, this solution is not as optimal as one would expect, since it might give place to internal nodes whose leaves are all of the same colour (see Figure 5.10). In these cases, a single node of that colour needs less heap space. We can replace the call to the constructor by a wrapper that performs all the necessary simplifications:

$$
\begin{aligned}
&\mathit{overlayD}\ (\mathit{Node}\ \mathit{nw}_1\ \mathit{ne}_1\ \mathit{sw}_1\ \mathit{se}_1)!\ (\mathit{Node}\ \mathit{nw}_2\ \mathit{ne}_2\ \mathit{sw}_2\ \mathit{se}_2)!\\
&\qquad = \mathit{buildNode}\ (\mathit{overlayD}\ \mathit{nw}_1\ \mathit{nw}_2)\ (\mathit{overlayD}\ \mathit{ne}_1\ \mathit{ne}_2)\ (\mathit{overlayD}\ \mathit{sw}_1\ \mathit{sw}_2)\ (\mathit{overlayD}\ \mathit{se}_1\ \mathit{se}_2)
\end{aligned}
$$

So, let us define *buildNode*. A first attempt on implementing this function is as follows:

$$
\begin{aligned}
&\mathit{buildNode}\ \mathit{Black}!\ \mathit{Black}!\ \mathit{Black}!\ \mathit{Black}! = \mathit{Black}\\
&\mathit{buildNode}\ \mathit{White}!\ \mathit{White}!\ \mathit{White}!\ \mathit{White}! = \mathit{White}\\
&\mathit{buildNode}\ \mathit{nw}!\ \mathit{ne}!\ \mathit{sw}!\ \mathit{se}! = \mathit{Node}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se}
\end{aligned}
$$

This function is rejected by the inference algorithm, since it is not correct. When the execution reaches the right-hand side of the third equation it accesses the *nw*, *ne*, *sw*, *se* variables, which have been already destroyed by the pattern matching. In principle we would have to access to the children of each variable, and rebuild the corresponding DSs in the result, as done, for example, in the third equation of *overlayD*. However, that would result in a considerable number of equations, since we would have to distinguish each possible combination of constructors for the four parameters. The following definition, though not very elegant, circumvents this problem:

$$
\begin{aligned}
&\mathit{buildNode}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se}\\
&\quad \mid \mathit{isBlack}\ \mathit{nw}\ \wedge\ \mathit{isBlack}\ \mathit{ne}\ \wedge\ \mathit{isBlack}\ \mathit{sw}\ \wedge\ \mathit{isBlack}\ \mathit{se} =\\
&\qquad\qquad \textbf{let}\ x_1 = \mathit{destroyBlackNodes}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se}\ \textbf{in}\ \mathit{Black}\\
&\quad \mid \mathit{isWhite}\ \mathit{nw}\ \wedge\ \mathit{isWhite}\ \mathit{ne}\ \wedge\ \mathit{isWhite}\ \mathit{sw}\ \wedge\ \mathit{isWhite}\ \mathit{se}\\
&\qquad\qquad \textbf{let}\ x_1 = \mathit{destroyWhiteNodes}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se}\ \textbf{in}\ \mathit{White}\\
&\quad \mid \mathit{otherwise} = \mathit{Node}\ \mathit{nw}\ \mathit{ne}\ \mathit{sw}\ \mathit{se}
\end{aligned}
$$

where the *isXXX* and *destroyXXXNodes* functions are defined as follows:

$$
\begin{aligned}
&\mathit{isBlack}\quad \mathit{Black} &&= \mathit{True}\\
&\mathit{isBlack}\quad \_ &&= \mathit{False}\\
&\\
&\mathit{isWhite}\quad \mathit{White} &&= \mathit{True}\\
&\mathit{isWhite}\quad \_ &&= \mathit{False}\\
&\\
&\mathit{destroyBlackNodes}\quad \mathit{Black}!\ \mathit{Black}!\ \mathit{Black}!\ \mathit{Black}! &&= 0\\
&\\
&\mathit{destroyWhiteNodes}\quad \mathit{White}!\ \mathit{White}!\ \mathit{White}!\ \mathit{White}! &&= 0
\end{aligned}
$$

The *destroyXXXNodes* functions produce only the side-effect of destroying the leaves passed as parameters, but their return value is meaningless, so it is not used in *buildNode*. The inference algorithm accepts all these functions, and returns the following mark signatures:

$$
\begin{aligned}
&\mathit{isBlack} &&:: \mathsf{s} \to \mathsf{s}\\
&\mathit{isWhite} &&:: \mathsf{s} \to \mathsf{s}\\
&\mathit{destroyBlackNodes} &&:: \mathsf{d} \to \mathsf{d} \to \mathsf{d} \to \mathsf{d} \to \mathsf{s}\\
&\mathit{destroyWhiteNodes} &&:: \mathsf{d} \to \mathsf{d} \to \mathsf{d} \to \mathsf{d} \to \mathsf{s}\\
&\mathit{buildNode} &&:: \mathsf{d} \to \mathsf{d} \to \mathsf{d} \to \mathsf{d} \to \mathsf{s}\\
&\mathit{overlayD} &&:: \mathsf{d} \to \mathsf{d} \to \mathsf{s}
\end{aligned}
$$

□

Example 5.4. Let us recall the implementation of priority queues shown in Example 3.13. All functions are accepted by the inference algorithm, which returns the following signatures:

$$
\begin{array}{lcl}
\mathit{cons} & :: & \mathsf{d} \to \mathsf{s} \to \mathsf{d} \to \mathsf{s} \\
\mathit{join} & :: & \mathsf{d} \to \mathsf{d} \to \mathsf{s} \\
\mathit{emptyPQueue} & :: & \mathsf{s} \\
\mathit{addPQueue} & :: & \mathsf{d} \to \mathsf{s} \to \mathsf{s} \\
\mathit{minPQueue} & :: & \mathsf{s} \to \mathsf{s} \\
\mathit{delMinPQueue} & :: & \mathsf{d} \to \mathsf{s}
\end{array}
$$

The first two functions require two iterations of the $\vdash$ rules in order to reach a fixed point. The remaining ones are non-recursive, and hence require only one iteration. □

Example 5.5. Figure 5.11 shows an implementation of the *mergesort* algorithm for sorting a list of integers. The *splitD* function is given a natural number n, and a list xs. It returns a pair with the n first elements of the list in the first component, and the remaining ones in the second component. It also destroys the list given as input. The *mergeD* function coalesces two sorted lists into a single ordered list. It is destructive on its two parameters. As a consequence, the execution of both functions can be done in constant space. Both functions are accepted by the algorithm, which results in the following signatures:

$$
\begin{array}{lcl}
\mathit{splitD} & :: & \mathsf{s} \to \mathsf{d} \to \mathsf{s} \\
\mathit{mergeD} & :: & \mathsf{d} \to \mathsf{d} \to \mathsf{s}
\end{array}
$$

Notice that, in the first equation of *splitD*, the xs parameter occurs with a $(!)$ mark, but its destruction does not take place until the pattern matching is done (second and third equations of *splitD*). That is why xs can be safely returned in the first equation. The same applies to *mergeD*. The *msortD* function, which sorts the input list in constant space, is not accepted by the algorithm. This is due to the information provided by the sharing analysis, which is not accurate enough to infer that the partial sorted lists (i.e. the results of *msortD* xs_1 and *msortD* xs_2) cannot share with each other. As a consequence, the algorithm fails in the function application of *mergeD*, since two mutually sharing lists cannot be passed as two different condemned parameters of *mergeD*. The current implementation of the algorithm allows the programmer to adjust the sharing information manually, in order to deal with these situations. Obviously, the safety of the program can no longer be guaranteed if the programmer modifies the information of the sharing analysis in a wrong way. In our case, is it easy to see that *msortD* builds its result from scratch, and consequently, cannot share its recursive spine with that of the input list. If we remove that over-approximating information, the function is accepted by the algorithm, and we get the following signature:

$$\mathit{msortD} \;::\; \mathsf{d} \to \mathsf{s}$$

Another possibility is to modify the definition of *mergeD* so that it does not destroy any of the input lists. In this case the algorithm also results in the mark signature above. However, the memory costs of the algorithm are no longer constant. □

Figure 5.12 shows the mark signatures returned by the algorithm for some of the examples in last chapters. Some functions have been slightly modified, in order to include destructive pattern matching, instead of non-destructive, or to call the destructive version of a function, instead the non-destructive one. The last column indicates whether it was necessary to modify manually the results of the sharing analysis, in order to make the inference algorithm accept the function. Almost all the examples are accepted by the inference algorithm as is, except the `joinAVL` function, which required a manual refinement of the information given by the sharing analysis.

$$
\begin{array}{lllll}
splitD & 0 & xs! & = & ([\,], xs) \\
splitD & n & [\,]! & = & ([\,], [\,]) \\
splitD & n & (x : xs)! & = & (x : xs_1, xs_2) \\
\multicolumn{5}{l}{\quad \textbf{where}\ (xs_1, xs_2) = splitD\ (n-1)\ xs}
\end{array}
$$

$$
\begin{array}{lllll}
mergeD & [\,]! & ys! & = & ys \\
mergeD & (x : xs)! & [\,]! & = & x : xs \\
mergeD & (x : xs)! & (y : ys)! & & \\
\multicolumn{5}{l}{\quad \mid x \leq y = x : mergeD\ xs\ (y : ys)} \\
\multicolumn{5}{l}{\quad \mid x > y = y : mergeD\ (x : xs)\ ys}
\end{array}
$$

$$
\begin{array}{l}
msortD\ [\,]! = [\,] \\
msortD\ (x : xs)! = \textbf{case}!\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [\,] \to x : [\,] \\
\qquad\qquad\qquad\qquad (y : ys) \to mergeD\ (msortD\ xs_1)\ (msortD\ xs_2) \\
\qquad\quad \textbf{where}\ (xs_1, xs_2) = splitD\ (n\ 'div'\ 2)\ (x : (y : ys)) \\
\qquad\qquad\quad n = length\ (x : (y : ys))
\end{array}
$$

Figure 5.11: Implementation of *mergesort* algorithm.

Function name	Mark signature	Adjust sharing?
`appendD` (Ex. 2.10, pg. 25)	$\mathtt{d} \to \mathtt{s} \to \mathtt{s}$	No
`insertD` (Ex. 2.11, pg. 26)	$\mathtt{s} \to \mathtt{d} \to \mathtt{s}$	No
`inssort` (Ex. 2.11, pg. 26)	$\mathtt{d} \to \mathtt{s}$	No
`insertT` (Ex. 4.21, pg. 154)	$\mathtt{s} \to \mathtt{d} \to \mathtt{s}$	No
`mkTree` (Ex. 4.21, pg. 154)	$\mathtt{d} \to \mathtt{s}$	No
`inorder` (Ex. 4.21, pg. 154)	$\mathtt{d} \to \mathtt{s}$	No
`treesort` (Ex. 4.21, pg. 154)	$\mathtt{d} \to \mathtt{s}$	No
`sJoinAVL` (Ex. 3.14, pg. 92)	$\mathtt{s} \to \mathtt{s} \to \mathtt{s} \to \mathtt{s}$	No
`lJoinAVL` (Ex. 3.14, pg. 92)	$\mathtt{s} \to \mathtt{s} \to \mathtt{d} \to \mathtt{s}$	No
`rJoinAVL` (Ex. 3.14, pg. 92)	$\mathtt{d} \to \mathtt{s} \to \mathtt{s} \to \mathtt{s}$	No
`joinAVL` (Ex. 3.14, pg. 92)	$\mathtt{d} \to \mathtt{s} \to \mathtt{d} \to \mathtt{s}$	Yes
`insertAVL` (Ex. 3.14, pg. 92)	$\mathtt{s} \to \mathtt{d} \to \mathtt{s}$	No
`deleteAVL` (Ex. 3.14, pg. 92)	$\mathtt{s} \to \mathtt{d} \to \mathtt{s}$	No
`partition` (Ex. 4.19, pg. 150)	$\mathtt{s} \to \mathtt{d} \to \mathtt{s}$	No
`sumList` (Ex. 4.20, pg. 151)	$\mathtt{d} \to \mathtt{s}$	No
`pascal` (Ex. 4.20, pg. 151)	$\mathtt{s} \to \mathtt{s}$	No

Figure 5.12: Results of the inference algorithm for some of the examples in this thesis.

5.4 Correctness, completeness, and efficiency

In this section we prove that every function accepted by the mark inference algorithm is typeable w.r.t. the $\vdash_{Dst}$ rules, and that every typeable function w.r.t. the latter rules is accepted by the mark inference algorithm. In case there are several valid signatures for a given function, we explain which one is returned by the algorithm.

5.4.1 Correctness

Before getting in the correctness proof of the $\vdash$ rules, let us introduce some of their invariants. In particular, we prove that a variable cannot get more than a mark in the context of a given expression. That is, the sets returned by the $\vdash$ rules are pairwise disjoint. In addition, the D and S sets may only contain free variables in e.

Lemma 5.6. *Given an expression e such that $e \vdash (R, D, S)$:*

1. *R, D and S are pairwise disjoint.*
2. *$D \cup S \subseteq fv(e)$.*

Proof. By induction on the structure of e. Cases $e \equiv c$, $e \equiv x$, $e \equiv x!$ and $e \equiv C\ \overline{a_i}^n\ @\ \overline{r_j}^m$ are straightforward. In the case of a function application, the lemma is a direct consequence of the side conditions of rule $[APP_I]$. With respect to the remaining cases, from (5.1) it directly follows that the $\sqcup$ operator always returns mutually disjoint sets, so the first conclusion holds. With respect to the second one, we can prove from (5.1) that, if $(R, D, S) = (R_1, D_1, S_1) \sqcup (R_2, D_2, S_2)$ then $D \cup S = D_1 \cup S_1 \cup D_2 \cup S_2$. Together with the fact that $[LET_I]$ removes the bound variable from (R_2, D_2, S_2), and that $[CASE_I]$, $[CASE!_I]$ removes the pattern variables from each (R_i, D_i, S_i), the second conclusion follows trivially from the induction hypothesis applied to each sub-expression. □

With regard to the correctness of the inference rules, we are interested in proving the following fact:

> *If an expression is accepted by the inference algorithm, it is $\vdash_{Dst}$-typeable.*

However, we can take advantage of the information provided by the $\vdash$ rules (namely, the (R, D, S) sets) and be more specific on which mark environment Γ can be used to type the expression w.r.t. the $\vdash_{Dst}$ rules: the variables in the R set (resp. D, S) should occur with a `r` (resp. `d`, `s`) mark in that environment. Hence, we aim to prove the following stronger property:

> *If $e \vdash (R, D, S)$ is derived by the rules of Figure 5.1, then e is $\vdash_{Dst}$-typeable under an environment assigning an in-danger mark to the variables in R, a condemned mark to those in D, and a safe mark to those in S.*

Before proving this, we need an auxiliary result with respect to the $\sqcup$ operator. The definition of this operator, when applied to type environments, has been introduced in in Section 3.3. The result of $\Gamma_1 \sqcup \Gamma_2$, when defined, is another type environment which combines the types of Γ_1 and Γ_2, but giving precedence to in-danger marks over the remaining ones, and condemned marks over safe marks. In this chapter we have given a definition of $\sqcup$ when applied to tuples (R, D, S) (see (5.1)). It is not surprising that this operator behaves in a similar way to its counterpart defined on type environments: the R set takes precedence over D and S, and the D takes precedence over S. This connection between the two meanings of $\sqcup$ is formalized as follows.

Lemma 5.7. *If* $(R, D, S) = (R_1, D_1, S_1) \sqcup (R_2, D_2, S_2)$*, and the following environments are well-defined:*

$$\begin{aligned}
\Gamma_1 &= [x : \mathsf{r} \mid x \in R_1] + [x : \mathsf{d} \mid x \in D_1] + [x : \mathsf{s} \mid x \in S_1] \\
\Gamma_2 &= [x : \mathsf{r} \mid x \in R_2] + [x : \mathsf{d} \mid x \in D_2] + [x : \mathsf{s} \mid x \in S_2] \\
\Gamma &= [x : \mathsf{r} \mid x \in R] + [x : \mathsf{d} \mid x \in D] + [x : \mathsf{s} \mid x \in S]
\end{aligned}$$

then $\Gamma = \Gamma_1 \sqcup \Gamma_2$.

Proof. First we prove that dom Γ = dom $(\Gamma_1 \sqcup \Gamma_2)$. This follows from $R \cup D \cup S$ being equal to $R_1 \cup R_2 \cup S_1 \cup S_2 \cup D_1 \cup D_2$, which can be easily proven from the definition of the $\sqcup$ operator when applied to triples (R, D, S).

Now we prove that, for all $x \in \text{dom } \Gamma$, $(\Gamma_1 \sqcup \Gamma_2)(x) = \mathsf{r}$ if and only if $x \in R$.

$$\begin{aligned}
& (\Gamma_1 \sqcup \Gamma_2)(x) = \mathsf{r} \\
\Leftrightarrow\ & \Gamma_1(x) = \mathsf{r} \ \vee\ \Gamma_2(x) = \mathsf{r} \\
\Leftrightarrow\ & x \in R_1 \ \vee\ x \in R_2 \\
\Leftrightarrow\ & x \in R_1 \cup R_2 \\
\Leftrightarrow\ & x \in R
\end{aligned}$$

Now we prove $(\Gamma_1 \sqcup \Gamma_2)(x) = \mathsf{d}$ if and only if $x \in D$.

$$\begin{aligned}
& (\Gamma_1 \sqcup \Gamma_2)(x) = \mathsf{d} \\
\Leftrightarrow\ & (\Gamma_1(x) = \mathsf{d} \wedge (\Gamma_2(x) \neq \mathsf{r} \vee x \notin \text{dom } \Gamma_2)) \vee (\Gamma_2(x) = \mathsf{d} \wedge (\Gamma_1(x) \neq \mathsf{r} \vee x \notin \text{dom } \Gamma_1)) \\
\Leftrightarrow\ & (x \in D_1 \wedge (x \notin R_2 \vee x \notin R_2 \cup D_2 \cup S_2)) \vee (x \in D_2 \wedge (x \notin R_1 \vee x \notin R_1 \cup D_1 \cup S_1)) \\
\Leftrightarrow\ & (x \in D_1 \wedge \neg(x \in R_2 \wedge x \in R_2 \cup D_2 \cup S_2)) \vee (x \in D_2 \wedge \neg(x \in R_1 \wedge x \in R_1 \cup D_1 \cup S_1)) \\
\Leftrightarrow\ & (x \in D_1 \wedge x \notin (R_2 \cap (R_2 \cup D_2 \cup S_2)) \vee (x \in D_2 \wedge x \notin (R_1 \cap (R_1 \cup D_1 \cup S_1))) \\
\Leftrightarrow\ & (x \in D_1 \wedge x \notin R_2) \vee (x \in D_2 \wedge x \notin R_1) \\
\Leftrightarrow\ & (x \in D_1 \wedge x \notin R_1 \wedge x \notin R_2) \vee (x \in D_2 \wedge x \notin R_1 \wedge x \notin R_2) \\
\Leftrightarrow\ & (x \in D_1 \vee x \in D_2) \wedge x \notin R_1 \wedge x \notin R_2 \\
\Leftrightarrow\ & x \in D_1 \cup D_2 \wedge x \notin R_1 \cup R_2 \\
\Leftrightarrow\ & x \in D
\end{aligned}$$

Finally we prove $(\Gamma_1 \sqcup \Gamma_2)(x) = \mathsf{s}$ if and only if $x \in S$.

$$\begin{aligned}
& (\Gamma_1 \sqcup \Gamma_2)(x) = \mathsf{s} \\
\Leftrightarrow\ & (\Gamma_1 \sqcup \Gamma_2)(x) \neq \mathsf{r} \wedge (\Gamma_1 \sqcup \Gamma_2)(x) \neq \mathsf{d} \\
\Leftrightarrow\ & \Gamma(x) \neq \mathsf{r} \wedge \Gamma(x) \neq \mathsf{d} \\
\Leftrightarrow\ & x \notin R \wedge x \notin D \\
\Leftrightarrow\ & x \in S
\end{aligned}$$

The last step follows because $\Gamma_1 \sqcup \Gamma_2$ and Γ have the same domain: since $x \in$ dom $(\Gamma_1 \sqcup \Gamma_2)$, we get $x \in \text{dom } \Gamma = R \cup D \cup S$, so if a variable does not belong to $R \cup D$, it must occur in S. □

Now we can prove the correctness of the $\vdash$ rules when applied to an expression e. Recall that, although not explicitly indicated in most cases, these rules are parametric on an environment Σ which contains the mark signatures of the functions being called from e. This environment is part of the mark environment Γ, under which e is typed.

Theorem 5.8. *Let us assume that the algorithm infers $e \vdash_\Sigma (R, D, S)$ under some signature Σ. If we define Γ as follows:*

$$\Gamma = \Sigma + [x : \mathsf{r} \mid x \in R] + [x : \mathsf{d} \mid x \in D] + [x : \mathsf{s} \mid x \in S]$$

then Γ is well-defined (i.e. is a function) and $\Gamma \vdash_{Dst} e : \mathsf{s}$.

Proof. Well-definedness follows from the fact that the sets R, D and S are disjoint (Lemma 5.6). Now we prove that $\Gamma \vdash_{Dst} e : \mathsf{s}$ by induction on the structure of e. We distinguish cases:

- **Case** $e \equiv c$

  Since $(R, D, S) = (\emptyset, \emptyset, \emptyset)$, we get $\Gamma = \Sigma$. Rule $[LIT_{Dst}]$ allows us to derive $\emptyset \vdash_{Dst} c : \mathsf{s}$. By using the $[EXT_{Dst}]$ rule we obtain $\Gamma \vdash_{Dst} e : \mathsf{s}$.

- **Cases** $e \in \{x, x\,@\,r\}$

  In these cases $\Gamma = \Sigma + [x : \mathsf{s}]$. By applying the $[VAR_{Dst}]$ or $[COPY_{Dst}]$ rules we obtain $[x : \mathsf{s}] \vdash_{Dst} e : \mathsf{s}$, from which we get $\Sigma + [x : \mathsf{s}] \vdash_{Dst} e : \mathsf{s}$ by using $[EXT_{Dst}]$.

- **Case** $e \equiv C\ \overline{a_i}^n\ @\ r$

  We obtain $\Gamma = \Sigma \cup [\overline{a_i : \mathsf{s}}^n]$. The environment $\bigoplus_{i=1}^n [a_i : \mathsf{s}]$ is well-defined, as it is the union of bindings with s marks. So, by applying $[CONS_{Dst}]$ and $[EXT_{Dst}]$ we get $\Gamma \vdash e : \mathsf{s}$.

- **Case** $e \equiv f\ \overline{a_i}^n\ @\ \overline{r_j}^m$

  Assume that $\Sigma(f) = \overline{m_i}^n \to \mathsf{s}$. Each m_i belongs to the set $\{\mathsf{d}, \mathsf{s}\}$. We get, for every $i \in \{1..n\}$ (assuming $a_i \in$ **Var**)

  $$\Gamma(a_i) = \mathsf{s} \Leftrightarrow a_i \in S \Leftrightarrow m_i = \mathsf{s}$$

  Analogously:

  $$\Gamma(a_i) = \mathsf{d} \Leftrightarrow a_i \in D_i \Leftrightarrow m_i = \mathsf{d}$$

  First we prove that $\bigoplus_{i=1}^n [a_i : m_i]$ is well-defined. If it were not, we would get two different $i, j \in \{1..n\}$ such that $a_i = a_j$, and $m_i = \mathsf{d}$.

  - If $m_j = \mathsf{d}$, then $a_j = a_i \in D_i$ and $a_i = a_j \in D_j$, so D_i and D_j would not be disjoint, which contradicts the assumptions of $[APP_I]$.
  - If $m_j = \mathsf{s}$, then $a_i = a_j \in S$, but $a_j = a_i \in D$. This contradicts the $S \cap D = \emptyset$ assumption of $[APP_I]$.

  If we define $\Gamma_R = [y : \mathsf{r} \mid y \in R]$, the environment $\Gamma_R + \bigoplus_{i=1}^n [a_i : m_i]$ is also well-defined, because of the conditions $R \cap D = \emptyset$ and $R \cap S = \emptyset$. We can apply rule $[APP_{Dst}]$ so as to get $\Gamma_R + [f : \overline{m_i} \to \mathsf{s}] + \bigoplus_{i=1}^n [a_i : m_i] \vdash_{Dst} e : \mathsf{s}$. Since Γ is a superset of this environment, we get $\Gamma \vdash_{Dst} e : \mathsf{s}$ by $[EXT_{Dst}]$.

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

  From the premises in $[LET_I]$ we get $e_1 \vdash (R_1, D_1, S_1)$, and $e_2 \vdash (R_2, D_2, S_2)$. By induction hypothesis, $\Gamma_1 \vdash_{Dst} e_1 : \mathsf{s}$, and $\Gamma_2' \vdash_{Dst} e_2 : \mathsf{s}$, where Γ_1 and Γ_2' are defined as follows:

  $$\begin{aligned} \Gamma_1 &= \Sigma + [x : \mathsf{r} \mid x \in R_1] + [x : \mathsf{d} \mid x \in D_1] + [x : \mathsf{s} \mid x \in S_1] \\ \Gamma_2' &= \Sigma + [x : \mathsf{r} \mid x \in R_2] + [x : \mathsf{d} \mid x \in D_2] + [x : \mathsf{s} \mid x \in S_2] \end{aligned}$$

We define m_1 and Γ_2 as follows:

$$\Gamma_2 = \begin{cases} \Gamma_2' \backslash x_1 & \text{if } x_1 \in \operatorname{dom} \Gamma_2' \\ \Gamma_2' & \text{if } x_1 \notin \operatorname{dom} \Gamma_2' \end{cases} \qquad m_1 = \begin{cases} \Gamma_2'(x_1) & \text{if } x_1 \in \operatorname{dom} \Gamma_2' \\ \mathsf{s} & \text{if } x_1 \notin \operatorname{dom} \Gamma_2' \end{cases}$$

In both cases we can derive $\Gamma_2 + [x_1 : m_1] \vdash_{Dst} e_2 : \mathsf{s}$ (we have to use $[EXT_{Dst}]$ if $x_1 \notin \operatorname{dom} \Gamma_2'$). From Lemma 5.7,

$$\Gamma \backslash (\operatorname{dom} \Sigma) = \Gamma_1 \backslash (\operatorname{dom} \Sigma) \sqcup \Gamma_2 \backslash (\operatorname{dom} \Sigma)$$

and, hence,

$$\Gamma = \Gamma_1 \sqcup \Gamma_2$$

So we can apply $[LET_{Dst}]$ if its side condition holds. From the side condition of $[LET_I]$ we get:

$$(R_1 \cup D_1) \cap fv(e_2) = \emptyset$$

which is equivalent to

$$\begin{array}{ll} & (\{x \in \operatorname{dom} \Gamma_1 \mid \Gamma_1(x) = \mathsf{r}\} \cup \{x \in \operatorname{dom} \Gamma_1 \mid \Gamma_1(x) = \mathsf{d}\}) \cap fv(e_2) = \emptyset \\ \Leftrightarrow & (\{x \in \operatorname{dom} \Gamma_1 \mid \Gamma_1(x) \in \{\mathsf{d}, \mathsf{r}\}\}) \cap fv(e_2) = \emptyset \\ \Leftrightarrow & \forall x \in \operatorname{dom} \Gamma_1 . \Gamma_1(x) \notin \{\mathsf{d}, \mathsf{r}\} \vee x \notin fv(e_2) \\ \Leftrightarrow & \forall x \in \operatorname{dom} \Gamma_1 . \Gamma_1(x) \in \{\mathsf{d}, \mathsf{r}\} \Rightarrow x \notin fv(e_2) \end{array}$$

which proves the result.

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

  For each sub-expression e_i we get $e_i \vdash (D_i, R_i, S_i)$, which implies, by induction hypothesis $\Gamma_i' \vdash_{Dst} e_i : \mathsf{s}$, where Γ_i' is defined as follows:

$$\Gamma_i' = \Sigma + [z : \mathsf{r} \mid z \in R_i] + [z : \mathsf{d} \mid z \in D_i] + [z : \mathsf{s} \mid z \in S_i]$$

  For each i we define $\Gamma_i = \Gamma_i' \backslash \{\overline{x_{ij}}^{n_i}\}$. From the judgement $\Gamma_i' \vdash_{Dst} e_i : \mathsf{s}$ (and by using rule $[EXT_{Dst}]$ if necessary) we can infer $\Gamma_i + [\overline{x_{ij} : m_{ij}}^{n_i}] \vdash_{Dst} e_i : \mathsf{s}$ for some $\{\overline{m_{ij}}^{n_i}\}$. Let $(R', D', S') = \bigsqcup_{i=1}^{n} ((R_i, D_i, S_i) \backslash P_i)$. If we define Γ' as follows:

$$\Gamma' = \Sigma + [z : \mathsf{r} \mid z \in R'] + [z : \mathsf{d} \mid z \in D'] + [z : \mathsf{s} \mid z \in S']$$

  Then, by Lemma 5.7,

$$\Gamma' \backslash (\operatorname{dom} \Sigma) = \bigsqcup_{i=1}^{n} \Gamma_i \backslash (\operatorname{dom} \Sigma)$$

  which implies

$$\Gamma' = \bigsqcup_{i=1}^{n} \Gamma_i$$

  Then, for every $i \in \{1..n\}$ and $z \in \operatorname{dom} \Gamma_i$, $\Gamma_i(z) \leq \Gamma'(z)$. This allows us to apply $[WEAK_{Dst}]$ and/or $[EXT_{Dst}]$ in order to obtain $\Gamma' + [\overline{x_{ij} : m_{ij}}^{n_i}] \vdash_{Dst} e_i : \mathsf{s}$ for each $i \in \{1..n\}$. Now we distinguish cases:

  - $x \in R' \cup D' \cup S'$

Then $R = R'$, $D = D'$ and $S = S'$. Therefore $\Gamma = \Gamma'$ and $\Gamma + [\overline{x_{ij} : m_j}^{n_i}] \vdash_{Dst} e_i : \mathsf{s}$ for each $i \in \{1..n\}$. We apply $[CASE_{Dst}]$ in order to obtain $\Gamma \vdash_{Dst} e : \mathsf{s}$.

– $x \notin R' \cup D' \cup S'$

In this case $R = R'$, $D = D'$, but $S = S' \cup \{x\}$, so $\Gamma = \Gamma' + [x : \mathsf{s}]$. By rule $[EXT_{Dst}]$, $\Gamma + [\overline{x_{ij} : m_{ij}}^{n_i}] \vdash_{Dst} e : \mathsf{s}$ for each $i \in \{1..n\}$. Again, by rule $[CASE_{Dst}]$, we obtain $\Gamma \vdash_{Dst} e : \mathsf{s}$.

- **Case** $e \equiv$ **case!** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

As in the previous case, we get $\Gamma'_i \vdash_{Dst} e_i : \mathsf{s}$ for each $i \in \{1..n\}$ by induction hypothesis, where Γ'_i is defined as follows:

$$\Gamma'_i = \Sigma + [z : \mathsf{r} \mid z \in R_i] + [z : \mathsf{d} \mid z \in D_i] + [z : \mathsf{s} \mid z \in S_i]$$

Let us think about the pattern variables $\overline{x_{ij}}^{n_i}$. By the conditions $Rec_i \cap R_i = \emptyset$ and $(P_i \backslash Rec_i) \cap (D_i \cup R_i) = \emptyset$ occurring in $[CASE!_I]$ we know that, for every $j \in \{1..n_i\}$:

– If $j \in RecPos(C_i)$, then $x_{ij} \in Rec_i$ and x_{ij} occurs in Γ'_i with an s or d mark, or does not occur in Γ'_i at all. In the latter case, we can apply rule $[EXT_{Dst}]$ to make it occur with a d mark. If $\Gamma'_i(x_{ij}) = \mathsf{s}$, we can apply rule $[WEAK_{Dst}]$ to upgrade it to a d mark.

– If $j \notin RecPos(C_i)$, then $x_{ij} \in P_i \backslash Rec_i$, and x_{ij} occurs in Γ'_i with an s mark, or does not occur in Γ'_i at all. Again, in the latter case we can apply $[EXT_{Dst}]$ and add x_{ij} to the environment with an s mark.

Given the above, we can manage to obtain a judgement $\Gamma'_i \backslash \{\overline{x_{ij}}^{n_i}\} + [\overline{x_{ij} : m'_{ij}}^{n_i}] \vdash e_i : \mathsf{s}$, where the $\overline{m'_{ij}}^{n_i}$ are defined as follows:

$$\forall i \in \{1..n\}.\forall j \in \{1..n_i\}.m'_{ij} = \begin{cases} \mathsf{d} & \text{if } j \in RecPos(C_i) \\ \mathsf{s} & \text{if } j \notin RecPos(C_i) \end{cases} \tag{5.2}$$

Let $(R', D', S') = \bigsqcup_{i=1}^{n}((R_i, D_i, S_i) \backslash P_i)$. By defining Γ' in the following way:

$$\Gamma' = \Sigma + [z : \mathsf{r} \mid z \in R'] + [z : \mathsf{d} \mid z \in D'] + [z : \mathsf{s} \mid z \in S']$$

we can prove that $\Gamma' = \bigsqcup_{i=1}^{n} \Gamma'_i \backslash \{x_{ij}\}$ and $\Gamma' + [\overline{x_{ij} : m'_{ij}}^{n_i}] \vdash_{Dst} e_i : \mathsf{s}$. This is done in the same way as in the non-destructive **case**.

If we define $(R'', D'', S'') \stackrel{\text{def}}{=} ((R', D', S') \backslash \{x\}) \sqcup (R_{SHR}, \emptyset, \emptyset)$, where R_{SHR} is defined as in the $[CASE!_I]$ rule, we get, by Lemma 5.7,

$$\Gamma'' \backslash (\text{dom } \Sigma) = \Gamma_R \sqcup (\Gamma' \backslash (\text{dom } \Sigma \cup \{x\})) \tag{5.3}$$

where:

$$\Gamma'' = \Sigma + [z : \mathsf{r} \mid z \in R''] + [z : \mathsf{d} \mid z \in D''] + [z : \mathsf{s} \mid z \in S'']$$

$$\Gamma_R = [z : \mathsf{r} \mid z \in R_{SHR}]$$

By incorporating Σ into both sides of (5.3) we obtain $\Gamma'' = \Gamma_R \sqcup (\Gamma' \backslash x)$. However, notice that

$R'' = R$, $S'' = S$ and $D'' \cup \{x\} = D$. Therefore:

$$\begin{aligned}\Gamma'' + [x : \mathsf{d}] &= \Sigma + [z : \mathsf{r} \mid z \in R''] + [z : \mathsf{d} \mid z \in D''] + [z : \mathsf{s} \mid z \in S''] + [x : \mathsf{d}] \\ &= \Sigma + [z : \mathsf{r} \mid z \in R] + [z : \mathsf{d} \mid z \in D] + [z : \mathsf{s} \mid z \in S] \\ &= \Gamma\end{aligned}$$

So $\Gamma = \Gamma'' + [x : \mathsf{d}] = \Gamma_R \sqcup (\Gamma' \backslash x) + [x : \mathsf{d}]$. We can apply $[CASE!_{Dst}]$ in order to get $\Gamma \vdash_{Dst} e : \mathsf{s}$ if we are able to prove that the side conditions of this rule hold. The condition of $\forall z \in R \cup \{x\}.\forall i \in \{1..n\}.z \notin fv(e_i)$ occurring in $[CASE!_{Dst}]$ follows from its counterpart $\forall i \in \{1..n\}.(R_{SHR} \cup \{x\}) \cap fv(e_i) = \emptyset$ in $[CASE!_I]$. With respect to the *inh* conditions on the $\overline{m'_{ij}}^{n_i}$, they follow immediately from (5.2).

□

The correctness of the *inferMarksDef* function for inferring the mark signatures of function definitions follows immediately from this theorem. Before proving this, we need to define what is a typeable function, since the $\vdash_{Dst}$ rules only apply to expressions. By inspecting the [FUN] rule of Figure 3.9 we can derive the following definition:

Definition 5.9. Given a fixed signature environment Σ not containing f in its domain, a mark signature $\overline{m_i}^n \to \mathsf{s}$ is said to be correct for the function definition $f\ \overline{x_i}^{n'}\ @\ \overline{r_j}^m = e_f$ if and only if these three conditions hold:

1. $n = n'$.
2. $\forall i \in \{1..n\}.m_i \in \{\mathsf{s}, \mathsf{d}\}$.
3. $\Sigma + [f : \overline{m_i}^n \to \mathsf{s}] + [\overline{x_i : m_i}^n] \vdash_{Dst} e_f : \mathsf{s}$.

Given a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, let us assume that *inferMarksFP* has reached a fixed point. This means that, if we apply the inference rules to the body of the expression under a signature $\Sigma = \Sigma' + [f : \overline{m_i}^n \to \mathsf{s}]$ for some $\overline{m_i}^n$, and we obtain $e_f \vdash (R_f, D_f, S_f)$, the following judgement is derivable by Theorem 5.8:

$$\Sigma' + [f : \overline{m_i}^n \to \mathsf{s}] + [x : \mathsf{r} \mid x \in R_f] + [x : \mathsf{d} \mid x \in D_f] + [x : \mathsf{s} \mid x \in S_f] \vdash e_f : \mathsf{s}$$

The set R_f must be empty. Otherwise *inferMarksFP* would have returned an error, and it would have not reached a fixed point. Hence:

$$\Sigma' + [f : \overline{m_i}^n \to \mathsf{s}] + [x : \mathsf{d} \mid x \in D_f] + [x : \mathsf{s} \mid x \in S_f] \vdash e_f : \mathsf{s}$$

We know that $D_f \cup S_f \subseteq fv(e_f) \subseteq \{\overline{x_i}^n\}$, but there could be parameters $\overline{x_i}^n$ that do not appear in any of these sets, so let us apply the $[EXT_{Dst}]$ rule to the previous judgement in order to add those variables to the type environment with a safe mark:

$$\Sigma' + [f : \overline{m_i}^n \to \mathsf{s}] + [x : \mathsf{d} \mid x \in D_f] + [x : \mathsf{s} \mid x \in S_f] + [x : \mathsf{s} \mid x \in \{\overline{x_i}^n\} \backslash (D_f \cup S_f)] \vdash e_f : \mathsf{s}$$

This is equivalent to the following:

$$\Sigma' + [f : \overline{m_i}^n \to \mathsf{s}] + [x : \mathsf{d} \mid x \in D_f] + [x : \mathsf{s} \mid x \in \{\overline{x_i}^n\} \backslash D_f] \vdash e_f : \mathsf{s}$$

For each $i \in \{1..n\}$, let us define:

$$m'_i = \begin{cases} \mathsf{d} & \text{if } x_i \in D_f \\ \mathsf{s} & \text{if } x_i \notin D_f \end{cases}$$

Our typing judgement can be transformed as follows:

$$\Sigma' + [f : \overline{m_i}^n \to \mathsf{s}] + [\overline{x_i : m'_i}^n] \vdash e_f : \mathsf{s}$$

Since we have assumed a fixed point for *inferMarksFP*, we know that $m'_i = m_i$ for each $i \in \{1..n\}$. Therefore:

$$\Sigma' + [f : \overline{m_i}^n \to \mathsf{s}] + [\overline{x_i : m_i}^n] \vdash e_f : \mathsf{s}$$

Now the marks of the parameters $\overline{x_i}^n$ match the marks in the signature of f. If we combine this judgement with its counterpart in the $\vdash_{Reg}$ rules, the rule [FUN] can be applied, and hence the function is well-typed.

5.4.2 Termination

The discussion at the end of the last section only makes sense if the algorithm reaches a fixed point. If it does not, it could be due to three situations:

1. The application of the inference rules results in an error, since no rule can be applied.
2. The application of the inference rules to the body of a function definition results in a triple (R, D, S) in which $R \neq \emptyset$.
3. The *inferMarksFP* function does not terminate.

In this section we prove that the third situation never holds. Assume a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$. The key idea is to define an order between mark signatures, and to prove that each iteration of the $\vdash$ rules to e_f under an environment Σ results in a signature which is greater or equal than $\Sigma(f)$. So, if *inferMarksFP* did not terminate, there would exist an infinitely strictly-increasing chain of mark signatures. However, this cannot happen, since the set of mark signatures associated with a function of n parameters has 2^n elements, and hence is finite.

So, let us start defining an order between signatures:

Definition 5.10. Given a function of n parameters, we define the following order between signatures:

$$\overline{m_i}^n \to \mathsf{s} \leq \overline{m'_i}^n \to \mathsf{s} \overset{def}{\iff} \forall i \in \{1..n\}.m_i \leq m'_i$$

And we extend this order to Σ in the standard way.

From this definition if follows that the set of mark signatures for a given function with this order has the structure of a lattice. Figure 5.13 shows the lattices corresponding to a function of two and three parameters, respectively.

We define a similar order for (R, D, S) tuples. A tuple (R, D, S) is said to be lower than (R', D', S') if the former contains less unsafe variables (i.e. variables in R and D).

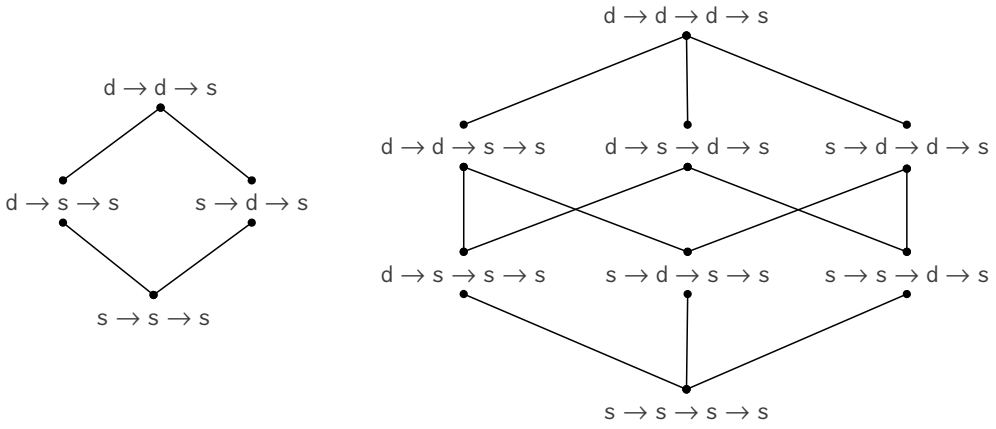


Figure 5.13: Lattice of mark signatures for functions with two and three parameters, respectively.

Definition 5.11. We define the following pre-order between triples (R, D, S):

$$(R, D, S) \sqsubseteq (R', D', S') \Leftrightarrow R \cup D \subseteq R' \cup D'$$

Lemma 5.12. *The $\sqcup$ operator is $\sqsubseteq$-monotonic w.r.t. all its parameters.*

Proof. Let us assume that $(R_1, D_1, S_1) \sqsubseteq (R'_1, D'_1, S'_1)$ and that $(R_2, D_2, S_2) \sqsubseteq (R'_2, D'_2, S'_2)$. If we define,

$$\begin{aligned} (R, D, S) &= (R_1, D_1, S_1) \sqcup (R_2, D_2, S_2) \\ (R', D', S') &= (R'_1, D'_1, S'_1) \sqcup (R'_2, D'_2, S'_2) \end{aligned}$$

we get:

$$\begin{aligned} R \cup D &= (R_1 \cup R_2) \cup ((D_1 \cup D_2) \backslash (R_1 \cup R_2)) \\ &= R_1 \cup R_2 \cup D_1 \cup D_2 \\ &\subseteq R'_1 \cup R'_2 \cup D'_1 \cup D'_2 \\ &= (R'_1 \cup R'_2) \cup ((D'_1 \cup D'_2) \backslash (R'_1 \cup R'_2)) \\ &= R' \cup D' \end{aligned}$$

Hence, $(R, D, S) \sqsubseteq (R', D', S')$. □

Given a fixed expression e, we can consider the application of the $\vdash$ rules as a function which, given a signature environment Σ, returns the result of applying the $\vdash$ rules on e under that environment. The following theorem proves that this function is monotone with respect to the $\leq$ order defined on signature environments.

Theorem 5.13. *Let Σ and Σ' be two signature environments without in-danger marks. If $\Sigma \leq \Sigma'$, $e \vdash_{\Sigma} (R, D, S)$, and $e \vdash_{\Sigma'} (R', D', S')$, then $(R, D, S) \sqsubseteq (R', D', S')$*

Proof. By induction on the structure of e. We distinguish cases:

- **Cases** $e \in \{c, x, x \ @ \ r, C\ \overline{a_i}^n \ @ \ r\}$

Trivial, since $R \cup D = \emptyset = R' \cup D'$.

- **Case** $e \equiv f\ \overline{a_i}^n\ @\ \overline{r_j}^m$

  Let $\Sigma(f) = \overline{m_i} \to \mathsf{s}$ and $\Sigma'(f) = \overline{m'_i} \to \mathsf{s}$. By assumption, $m_i = \mathsf{d}$ implies $m'_i = \mathsf{d}$. Therefore:

$$\begin{array}{ll} & R \cup D \\ = & \bigcup_{m_i = \mathsf{d}} sharerec(a_i, e) \backslash \{a_i\} \cup \bigcup_{i=1}^n D_i \\ = & \bigcup_{m_i = \mathsf{d}} sharerec(a_i, e) \backslash \{a_i\} \cup \{a_i \mid m_i = \mathsf{d}\} \\ \subseteq & \bigcup_{m'_i = \mathsf{d}} sharerec(a_i, e) \backslash \{a_i\} \cup \{a_i \mid m'_i = \mathsf{d}\} \\ = & R' \cup D' \end{array}$$

- **Case** $e \equiv \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2$

  Let us assume:

$$\begin{array}{ll} e_1 \vdash_{\Sigma} (R_1, D_1, S_1) & e_1 \vdash_{\Sigma'} (R'_1, D'_1, S'_1) \\ e_2 \vdash_{\Sigma} (R_2, D_2, S_2) & e_2 \vdash_{\Sigma'} (R'_2, D'_2, S'_2) \end{array}$$

  By induction hypothesis, $(R_1, D_1, S_1) \sqsubseteq (R'_1, D'_1, S'_1)$ and $(R_2, D_2, S_2) \sqsubseteq (R'_2, D'_2, S'_2)$. Since $\sqcup$ is monotonic with respect to both operands, we get:

$$(R, D, S) = (R_1, D_1, S_1) \sqcup ((R_2, D_2, S_2) \backslash \{x_1\}) \sqsubseteq (R'_1, D'_1, S'_1) \sqcup ((R'_2, D'_2, S'_2) \backslash \{x_1\}) = (R', D', S')$$

- **Case** $e \equiv \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n$

  Assume, for every $i \in \{1..n\}$, $e_i \vdash_{\Sigma} (R_i, D_i, S_i)$ and $e_i \vdash_{\Sigma'} (R'_i, D'_i, S'_i)$. By induction hypothesis, $(R_i, D_i, S_i) \sqsubseteq (R'_i, D'_i, S'_i)$ and by monotonicity of $\sqcup$ we get:

$$(R, D, S) = \left(\bigsqcup_{i=1}^{n} ((R_i, D_i, S_i) \backslash P_i) \right) \sqcup (\emptyset, \emptyset, \{x\}) \sqsubseteq \left(\bigsqcup_{i=1}^{n} ((R'_i, D'_i, S'_i) \backslash P_i) \right) \sqcup (\emptyset, \emptyset, \{x\}) = (R', D', S')$$

- **Case** $e \equiv \mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n$

  Again, we assume, for every $i \in \{1..n\}$, $e_i \vdash_{\Sigma} (R_i, D_i, S_i)$ and $e_i \vdash_{\Sigma'} (R'_i, D'_i, S'_i)$. Let us define:

$$(R_a, D_a, S_a) = \bigsqcup_{i=1}^{n} ((R_i, D_i, S_i) \backslash P_i) \qquad (R'_a, D'_a, S'_a) = \bigsqcup_{i=1}^{n} ((R'_i, D'_i, S'_i) \backslash P_i)$$

$$(R_b, D_b, S_b) = ((R_a, D_a, S_a) \backslash \{x\}) \sqcup (R_{SHR}, \emptyset, \emptyset) \qquad (R'_b, D'_b, S'_b) = ((R'_a, D'_a, S'_a) \backslash \{x\}) \sqcup (R_{SHR}, \emptyset, \emptyset)$$

  where R_{SHR} is as defined in $[CASE!_I]$. By monotonicity of $\sqcup$ operator, we get:

$$\forall i. (R_i, D_i, S_i) \sqsubseteq (R'_i, D'_i, S'_i) \Rightarrow (R_a, D_a, S_a) \sqsubseteq (R'_a, D'_a, S'_a) \Rightarrow (R_b, D_b, S_b) \sqsubseteq (R'_b, D'_b, S'_b)$$

  which implies $R_b \cup D_b \subseteq R'_b \cup D'_b$. Therefore:

$$R \cup D = R_b \cup D_b \cup \{x\} \subseteq R'_b \cup D'_b \cup \{x\} = R' \cup D'$$

□

The previous lemma is the key result to prove the termination of *inferMarksDef*. Assume that the latter function is called with a signature environment Σ and a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, and that *inferMarksFP* does not terminate. The first call to *inferMarksFP* is done with a signature Σ_0 in which $\Sigma_0(f) = \overline{\mathsf{s}}^n \to \mathsf{s}$. Let us assume that $e_f \vdash_{\Sigma_0} (D_0, R_0, S_0)$ for some D_0, R_0, and S_0. The second call is done with a signature Σ_1 such that $\Sigma_0(f) < \Sigma_1(f)$, so $\Sigma_0 \leq \Sigma_1$. The inequality is strict, since *inferMarksFP* would have terminated otherwise. Let us assume that $e_f \vdash_{\Sigma_1} (D_1, R_1, S_1)$. By Theorem 5.13 we know that $D_0 \cup R_0 \subseteq D_1 \cup R_1$. Since both R_0 and R_1 are empty (otherwise the algorithm would stop with an error), we get $D_0 \subseteq D_1$, which results in the following call to *inferMarksFP* under an environment Σ_2 such that $\Sigma_1(f) < \Sigma_2(f)$. By repeating this process, we get an infinite ascending chain:

$$\Sigma_0(f) < \Sigma_1(f) < \Sigma_2(f) < \dots$$

This leads to a contradiction, since the set of possible signatures for f is finite.

5.4.3 Completeness

In this section we prove that, if a function is typeable w.r.t. the $\vdash_{Dst}$ rules, the inference algorithm is able to return a mark signature for that function. Throughout this section we will write our type environments as $\Sigma + \Gamma$, where Σ only contains bindings from function and constructor names to mark signatures, and Γ associates variables with marks. We also assume that the mark signatures in Σ do not contain in-danger marks.

The first thing to point out is that there may be more than one correct mark signature for a given function. In principle, we want to prove that our algorithm returns one of these correct signatures. However, we should be more precise and specify *which* signature is given by the algorithm. Recall that the set of mark signatures for a given function is a lattice with respect to the order of Definition 5.10. The completeness proof is done in two steps:

1. Among all the correct mark signatures for a given function, there is a *minimal* correct mark signature.

2. Our algorithm computes this minimal correct mark signature.

The search for the minimal correct mark signature is closely related to the search of the minimal mark environment which types a given expression, so let us prove that such a minimal mark environment exists. Firstly we prove that the set of mark environments with the $\leq$ order defined in Section 3.7 has the structure of a distributive complete lattice:

Proposition 5.14. *The set of mark environments Γ such that dom $\Gamma \subseteq$ **Var** is a distributive complete lattice with respect to the $\leq$ order.*

Proof. The $\sqcup$ and $\sqcap$ operators satisfy the following properties,

$$\text{dom}\ (\Gamma_1 \sqcup \Gamma_2) = \text{dom}\ \Gamma_1 \cup \text{dom}\ \Gamma_2 \tag{5.4}$$

$$\text{dom}\ (\Gamma_1 \sqcap \Gamma_2) = \text{dom}\ \Gamma_1 \cap \text{dom}\ \Gamma_2 \tag{5.5}$$

$\Gamma_1(x)$	$\Gamma_2(x)$	$\Gamma_3(x)$	$((\Gamma_1 \sqcup \Gamma_2) \sqcap \Gamma_3)(x)$	$((\Gamma_1 \sqcap \Gamma_3) \sqcup (\Gamma_2 \sqcap \Gamma_3))(x)$
–	–	s	s	s
–	–	r	$\Gamma_1 \sqcup \Gamma_2$	$\Gamma_1 \sqcup \Gamma_2$
r	–	d	d	d
–	r	d	d	d
s	s	d	s	s
s	d	d	d	d
d	s	d	d	d
d	d	d	d	d

Figure 5.14: This table covers all the possibilities for the values of $\Gamma_1(x)$, $\Gamma_2(x)$ and $\Gamma_3(x)$. The dash symbol (–) stands for any mark. In every case, the desired equality holds.

and they are defined as follows:

$$\forall x \in \operatorname{dom} \Gamma_1 \cup \operatorname{dom} \Gamma_2.(\Gamma_1 \sqcup \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & x \notin \operatorname{dom} \Gamma_2 \\ \Gamma_2(x) & x \notin \operatorname{dom} \Gamma_1 \\ \Gamma_1(x) \sqcup \Gamma_2(x) & \text{otherwise} \end{cases}$$

$$\forall x \in \operatorname{dom} \Gamma_1(x) \cap \operatorname{dom} \Gamma_2.(\Gamma_1 \sqcap \Gamma_2)(x) = \Gamma_1(x) \sqcap \Gamma_2(x)$$

Given a (possibly infinite) family $\mathcal{G}$ of type environments, the result of $\sqcap\mathcal{G}$ is defined as follows:

$$\operatorname{dom} \sqcap \mathcal{G} = \bigcap_{\Gamma \in \mathcal{G}} \operatorname{dom} \Gamma \qquad \forall x \in \operatorname{dom} \sqcap \mathcal{G}.(\sqcap \mathcal{G})(x) = \bigsqcap_{\Gamma \in \mathcal{G}} \Gamma(x)$$

and similarly with $\sqcup\mathcal{G}$. Since the ordered set $(\{\mathtt{r}, \mathtt{d}, \mathtt{s}\}, \leq)$ is a complete lattice, so is the set of mark environments. Finally, we prove distributivity. Let us prove that, for every Γ_1, Γ_2 and Γ_3:

$$(\Gamma_1 \sqcup \Gamma_2) \sqcap \Gamma_3 = (\Gamma_1 \sqcap \Gamma_2) \sqcup (\Gamma_1 \sqcap \Gamma_3) \tag{5.6}$$

The fact that both sides of the equality have the same domain follows from (5.4), (5.5), and set algebra. Let us assume a variable x belonging to that domain. It must hold that $x \in \operatorname{dom} \Gamma_3$, and $x \in \operatorname{dom} \Gamma_1 \cup \operatorname{dom} \Gamma_2$. If $x \notin \operatorname{dom} \Gamma_1$ we get:

$$((\Gamma_1 \sqcup \Gamma_2) \sqcap \Gamma_3)(x) = (\Gamma_2 \sqcap \Gamma_3)(x) = ((\Gamma_1 \sqcap \Gamma_2) \sqcup (\Gamma_2 \sqcap \Gamma_3))(x)$$

since x does not belong to the domain of $\Gamma_1 \sqcap \Gamma_2$. If $x \notin \operatorname{dom} \Gamma_2$ we proceed in a similar way. Finally, if x belongs to the domain of the three environments, we distinguish cases as in Figure 5.14. The equality holds in all these cases, so we have proved (5.6). The fact $(\Gamma_1 \sqcap \Gamma_2) \sqcup \Gamma_3 = (\Gamma_1 \sqcup \Gamma_3) \sqcap (\Gamma_2 \sqcup \Gamma_3)$ follows by duality. □

Given a fixed Σ, the next lemma proves that, if there exists two mark environments typing a given expression, the greatest lower bound of them (which exists, by Proposition 5.14) also types that expression.

Lemma 5.15. *If* $\Sigma + \Gamma_1 \vdash_{Dst} e : \mathsf{s}$ *and* $\Sigma + \Gamma_2 \vdash_{Dst} e : \mathsf{s}$, *then* $\Sigma + (\Gamma_1 \sqcap \Gamma_2) \vdash_{Dst} e : \mathsf{s}$.

Proof. By induction on the structure of e. Without loss of generality, let us assume that the last rule in each of the $\vdash_{Dst}$ derivations is neither $[EXT_{Dst}]$ nor $[WEAK_{Dst}]$. Otherwise we can prove the existence of

some $\Sigma' \subseteq \Sigma$, $\Gamma'_1 \leq \Gamma_1$, and $\Gamma'_2 \leq \Gamma_2$ such that $\Sigma' + \Gamma'_1 \vdash_{Dst} e : \mathsf{s}$ and $\Sigma' + \Gamma'_2 \vdash_{Dst} e : \mathsf{s}$, and this condition holds in both derivations. Lemma 3.49 is useful for ensuring that Σ' is the same in both judgements. Once we have proved that $\Sigma' + (\Gamma'_1 \sqcap \Gamma'_2) \vdash_{Dst} e : \mathsf{s}$ we know that $\Gamma'_1 \sqcap \Gamma'_2 = \Gamma_1 \sqcap \Gamma_2$ and, by the $[EXT_{Dst}]$ rule, $\Sigma + (\Gamma_1 \sqcap \Gamma_2) \vdash_{Dst} e : \mathsf{s}$. Therefore, we can safely rule out the possibility of having $[EXT_{Dst}]$ or $[WEAK_{Dst}]$ as the last rule applied, and we distinguish between the remaining cases:

- **Cases** $[LIT_{Dst}]$**,** $[VAR_{Dst}]$**,** $[COPY_{Dst}]$**, and** $[CONS_{Dst}]$**.**

  All these cases are trivial, since they imply that $\Sigma = \emptyset$ and $\Gamma_1 = \Gamma_2$, so $\Sigma + (\Gamma_1 \sqcap \Gamma_2) = \Gamma_1 = \Gamma_2$.

- **Case** $[APP_{Dst}]$.

  Let $e \equiv f\ \overline{x_i}\ @\ \overline{r_j}$. The value of $\Sigma(f)$ determines the contents of Γ_1 and Γ_2. Since Σ is the same in both judgements, we get $\Gamma_1 = \Gamma_2$, and the lemma holds trivially.

- **Case** $[LET_{Dst}]$**.**

  Let $e \equiv \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2$, and the following derivations:

$$\frac{\Sigma_{11} + \Gamma_{11} \vdash_{Dst} e_1 : \mathsf{s} \qquad \Sigma_{12} + \Gamma_{12} + [x_1 : m_1] \vdash_{Dst} e_2 : \mathsf{s}}{\Sigma + \underbrace{(\Gamma_{11} \sqcup \Gamma_{12})}_{\Gamma_1} \vdash_{Dst} e : \mathsf{s}}$$

$$\frac{\Sigma_{21} + \Gamma_{21} \vdash_{Dst} e_1 : \mathsf{s} \qquad \Sigma_{22} + \Gamma_{22} + [x_1 : m_2] \vdash_{Dst} e_2 : \mathsf{s}}{\Sigma + \underbrace{(\Gamma_{21} \sqcup \Gamma_{22})}_{\Gamma_2} \vdash_{Dst} e : \mathsf{s}}$$

  since each Σ_{ij} is a subset of Σ for each $i, j \in \{1, 2\}$, we can apply $[EXT_{Dst}]$ in each of the premises so as to obtain $\Sigma + \Gamma_{ij}$ in each judgement, possibly followed by $[x_1 : m_i]$ (where $i, j \in \{1, 2\}$). By induction hypothesis, we get:

$$\Sigma + (\Gamma_{11} \sqcap \Gamma_{21}) \vdash_{Dst} e_1 : \mathsf{s} \qquad \Sigma + (\Gamma_{12} \sqcap \Gamma_{22}) + [x_1 : m_1 \sqcap m_2] \vdash_{Dst} e_1 : \mathsf{s}$$

  We can apply $[LET_{Dst}]$, since $(\Gamma_{11} \sqcap \Gamma_{21})(x) \in \{\mathsf{d}, \mathsf{r}\}$ implies $\Gamma_{11}(x) \in \{\mathsf{d}, \mathsf{r}\}$, or $\Gamma_{21} \in \{\mathsf{d}, \mathsf{r}\}$, which implies $x \notin fv(e_2)$. Hence we get,

$$(\Sigma + (\Gamma_{11} \sqcap \Gamma_{21})) \sqcup (\Sigma + (\Gamma_{12} \sqcap \Gamma_{22})) \vdash_{Dst} e : \mathsf{s}$$

  which is equivalent to:

$$\Sigma + ((\Gamma_{11} \sqcap \Gamma_{21}) \sqcup (\Gamma_{12} \sqcap \Gamma_{22})) \vdash_{Dst} e : \mathsf{s}$$

  If we prove that $(\Gamma_{11} \sqcap \Gamma_{21}) \sqcup (\Gamma_{12} \sqcap \Gamma_{22}) \leq (\Gamma_{11} \sqcup \Gamma_{12}) \sqcap (\Gamma_{21} \sqcap \Gamma_{22}) = \Gamma_1 \sqcap \Gamma_2$ we will be able to apply the $[WEAK_{Dst}]$ rule to get the desired result. Thus, we have to prove this inequality. Let us define:

$$\begin{aligned} \Gamma_A &= (\Gamma_{11} \sqcap \Gamma_{21}) \sqcup (\Gamma_{12} \sqcap \Gamma_{22}) \\ \Gamma_B &= (\Gamma_{11} \sqcup \Gamma_{12}) \sqcap (\Gamma_{21} \sqcup \Gamma_{22}) \end{aligned}$$

  Firstly we prove that $\mathrm{dom}\ \Gamma_A \subseteq \mathrm{dom}\ \Gamma_B$. Let $x \in \mathrm{dom}\ \Gamma_A$. We distinguish cases:

  - $x \in \mathrm{dom}\ \Gamma_{11} \ \wedge\ x \in \mathrm{dom}\ \Gamma_{21}$. In this case we get $x \in \mathrm{dom}\ (\Gamma_{11} \sqcup \Gamma_{12})$, and $x \in \mathrm{dom}\ (\Gamma_{21} \sqcup \Gamma_{22})$. Therefore, $x \in \mathrm{dom}\ \Gamma_B$.

– $x \in \text{dom}\ \Gamma_{12} \ \wedge\ x \in \text{dom}\ \Gamma_{22}$. Again, we get $x \in \text{dom}\ (\Gamma_{11} \sqcup \Gamma_{12})$, and $x \in \text{dom}\ (\Gamma_{21} \sqcup \Gamma_{22})$. Therefore, $x \in \text{dom}\ \Gamma_B$.

Now we prove that, for every $x \in \text{dom}\ \Gamma_A$, we get $\Gamma_A(x) \leq \Gamma_B(x)$. If $\Gamma_A(x) = \mathtt{s}$ we are done. If $\Gamma_A(x) = \mathtt{r}$ there are two possibilities:

– $(\Gamma_{11} \sqcap \Gamma_{21})(x) = \mathtt{r}$, which implies $\Gamma_{11}(x) = \mathtt{r}$, and $\Gamma_{21} = \mathtt{r}$. Hence we get $(\Gamma_{11} \sqcup \Gamma_{12})(x) = \mathtt{r}$, $(\Gamma_{21} \sqcap \Gamma_{22})(x) = \mathtt{r}$, and, consequently $\Gamma_B(x) = \mathtt{r}$.

– $(\Gamma_{12} \sqcap \Gamma_{22})(x) = \mathtt{r}$, which implies $\Gamma_{12}(x) = \mathtt{r}$, and $\Gamma_{22} = \mathtt{r}$. Similarly as above, $\Gamma_B(x) = \mathtt{r}$.

Exactly in the same way we can prove that $\Gamma_A(x) = \mathtt{d}$ implies $\Gamma_B \in \{\mathtt{d}, \mathtt{r}\}$. The required result follow from this.

- **Case** $[CASE_{Dst}]$

Let $e \equiv \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$. From the premises of the $[CASE_{Dst}]$ rule we get,

$$\Sigma + (\Gamma_1 + [\overline{x_{ij} : m_{ij}}^{n_i}]) \vdash_{Dst} e_i : \mathtt{s} \wedge \Sigma + (\Gamma_2 + [\overline{x_{ij} : m'_{ij}}^{n_i}]) \vdash_{Dst} e_i : \mathtt{s} \qquad \text{for each } i \in \{1..n\}$$

and, by induction hypothesis,

$$(\Sigma + (\Gamma_1 + [\overline{x_{ij} : m_{ij}}^{n_i}])) \sqcap (\Sigma + (\Gamma_2 + [\overline{x_{ij} : m'_{ij}}^{n_i}])) \vdash_{Dst} e_i : \mathtt{s}$$

which is equivalent to:

$$\Sigma + ((\Gamma_1 \sqcap \Gamma_2) + [\overline{x_{ij} : m_{ij} \sqcap m'_{ij}}^{n_i}]) \vdash_{Dst} e_i : \mathtt{s}$$

for each $i \in \{1..n\}$. The required result follow from applying $[CASE_{Dst}]$ to these judgements.

- **Case** $[CASE!_{Dst}]$

Let $e \equiv \textbf{case}(!)\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$. From $[CASE!_{Dst}]$ we get,

$$\Sigma + (\Gamma_1 + [\overline{x_{ij} : m_{ij}}^{n_i}]) \vdash_{Dst} e_i : \mathtt{s} \wedge \Sigma + (\Gamma_2 + [\overline{x_{ij} : m'_{ij}}^{n_i}]) \vdash_{Dst} e_i : \mathtt{s} \qquad \text{for each } i \in \{1..n\}$$

The *inh* predicates in each judgement force $m_{ij} = m'_{ij}$ for each $i \in \{1..n\}$, $j \in \{1..n_i\}$. Moreover, the R set occurring in the premises of $[CASE!_{Dst}]$ is the same in both judgements, and so is the Γ_R environment. The induction hypothesis gives the following judgement:

$$(\Sigma + (\Gamma_1 + [\overline{x_{ij} : m_{ij}}^{n_i}])) \sqcap (\Sigma + (\Gamma_2 + [\overline{x_{ij} : m_{ij}}^{n_i}])) \vdash_{Dst} e_i : \mathtt{s}$$

which is equivalent to:

$$\Sigma + ((\Gamma_1 \sqcap \Gamma_2) + [\overline{x_{ij} : m_{ij}}^{n_i}]) \vdash_{Dst} e_i : \mathtt{s}$$

The remaining conditions in $[CASE!_{Dst}]$ follow from their counterparts in the initial judgements. Hence we get:

$$\Gamma_R \sqcup ((\Sigma + (\Gamma_1 \sqcap \Gamma_2)) \backslash x) + [x : \mathtt{d}] \vdash_{Dst} e : \mathtt{s}$$

The type environment can be transformed as follows:

$$\begin{aligned}
\Gamma_R \sqcup ((\Sigma + (\Gamma_1 \sqcap \Gamma_2))\backslash x) + [x : \mathsf{d}] &= \Sigma + (\Gamma_R \sqcup ((\Gamma_1 \sqcap \Gamma_2))\backslash x) + [x : \mathsf{d}]) \\
&= \Sigma + (\Gamma_R \sqcup ((\Gamma_1\backslash x) \sqcap (\Gamma_2\backslash x))) + [x : \mathsf{d}]) \\
&= \Sigma + ((\Gamma_R \sqcup (\Gamma_1\backslash x)) \sqcap (\Gamma_R \sqcup (\Gamma_2\backslash x))) + [x : \mathsf{d}]) \\
&= \Sigma + ((\Gamma_R \sqcup (\Gamma_1\backslash x) + [x : \mathsf{d}]) \sqcap (\Gamma_R \sqcup (\Gamma_2\backslash x) + [x : \mathsf{d}]))
\end{aligned}$$

This proves the lemma.

□

The following lemma allows us to strengthen the Σ signature in a typing derivation. The intuitive idea is that, if an expression is typeable under a mark environment Σ, we can replace some of the mark signatures in Σ by some other "less destructive" variants, and the expression is still typeable under the modified environment.

Lemma 5.16. *If $\Sigma + \Gamma \vdash_{Dst} e : \mathsf{s}$ and Σ' is an environment which contains the mark signatures of the functions being called in e, and $\Sigma' \leq \Sigma$, then $\Sigma' + \Gamma \vdash_{Dst} e : \mathsf{s}$.*

Proof. By induction on the size of the typing derivation. Let us distinguish cases on the last rule applied.

- **Case** $[EXT_{Dst}]$

  We can derive $\Sigma'' + \Gamma'' \vdash_{Dst} e : s$, for some $\Sigma'' \subseteq \Sigma$ and $\Gamma'' \subseteq \Gamma$. Let us define Σ'_1 and Σ'_2 as follows:

  $$\begin{aligned}
  \Sigma'_1 &= \Sigma'\backslash\Sigma'' \\
  \Sigma'_2 &= \Sigma' \sqcap \Sigma''
  \end{aligned}$$

  It is obvious that dom $\Sigma' =$ dom $\Sigma'_1 \uplus$ dom Σ'_2. Since $\Sigma'_2 \leq \Sigma''$, and both Σ' and Σ'' contain the signatures of the functions being applied in e, then Σ'_2 also contain those signatures. We can apply the induction hypothesis in order to get:

  $$\Sigma'_2 + \Gamma'' \vdash_{Dst} e : s$$

  By applying $[EXT_{Dst}]$ we get,

  $$\Sigma'_1 \uplus \Sigma'_2 + \Gamma \vdash_{Dst} e : s$$

  which proves the result.

- **Case** $[WEAK_{Dst}]$

  It follows trivially from the induction hypothesis.

- **Cases** $[LIT_{Dst}]$**,** $[VAR_{Dst}]$**,** $[COPY_{Dst}]$**, and** $[CONS_{Dst}]$**.**

  We get $\Sigma = [\,]$, which implies $\Sigma' = [\,]$, and the result holds trivially.

- **Case** $[APP_{Dst}]$

  Let $e \equiv f\ \overline{a_i}^n\ @\ \overline{r_j}^m$. In this case, $\Sigma = [f : \overline{m_i}^n \to \mathsf{s}]$ and $\Sigma' = [f : \overline{m'_i}^n \to \mathsf{s}]$ for some $\overline{m_i}^n$, $\overline{m'_i}^n$ such that $m'_i \leq m_i$ for all $i \in \{1..n\}$. Let us define:

  $$R = \bigcup_{i=1}^{n} \{sharerec(a_i, e) \mid m_i = \mathsf{d}\} \qquad R' = \bigcup_{i=1}^{n} \{sharerec(a_i, e) \mid m'_i = \mathsf{d}\}$$

Since $m'_i = \mathsf{d}$ implies $m_i = \mathsf{d}$ for every $i \in \{1..n\}$ (since the m_i must belong to the set $\{\mathsf{s}, \mathsf{d}\}$), we get $R' \subseteq R$. For the same reason, the condition $\bigwedge_{m'_i=\mathsf{d}} isTree(a_i)$ follows from $\bigwedge_{m_i=\mathsf{d}} isTree(a_i)$, and the well-definedness of $\bigoplus_{i=1}^{n}[a_i : m'_i]$ also follows from that of $\bigoplus_{i=1}^{n}[a_i : m_i]$. We apply $[APP_{Dst}]$:

$$[f : \overline{m'_i}^n \to \mathsf{s}] + [y : \mathsf{r} \mid y \in R'] + \bigoplus_{i=1}^{n}[a_i : m_i] \vdash_{Dst} e : \mathsf{s}$$

Moreover, since $R' \subseteq R$, and none of the $\overline{a_i}$ variables is in R, we can apply the $[EXT_{Dst}]$ rule:

$$\underbrace{[f : \overline{m'_i}^n \to \mathsf{s}]}_{\Sigma'} + \underbrace{[y : \mathsf{r} \mid y \in R] + \bigoplus_{i=1}^{n}[a_i : m_i]}_{\Gamma} \vdash_{Dst} e : \mathsf{s}$$

and the lemma holds.

- **Case** $[LET_{Dst}]$

  By the premises of that rule we get, assuming $e \equiv \textbf{let } x_1 = e_1 \textbf{ in } e_2$:

  $$\Sigma_1 + \Gamma_1 \vdash_{Dst} e_1 : \mathsf{s} \qquad \Sigma_2 + \Gamma_2 + [x_1 : m_1] \vdash_{Dst} e_2 : \mathsf{s}$$

  where $\Sigma \geq \Sigma_1$, $\Sigma \geq \Sigma_2$, and $\Gamma = \Gamma_1 \sqcup \Gamma_2$. Let us define $\Sigma'_1 = \Sigma_1 \sqcap \Sigma'$ and $\Sigma'_2 = \Sigma_2 \sqcap \Sigma'$. Hence $\Sigma' = \Sigma'_1 \cup \Sigma'_2$. Again, Σ' and Σ_1 contain the signatures of the functions being applied in e_1, and, hence, so does Σ'_1. Similarly with Σ'_2. By induction hypothesis:

  $$\Sigma'_1 + \Gamma_1 \vdash_{Dst} e_1 : \mathsf{s} \qquad \Sigma'_2 + \Gamma_2 + [x_1 : m_1] \vdash_{Dst} e_2 : \mathsf{s}$$

  By applying $[LET_{Dst}]$ we get $(\Sigma'_1 + \Sigma'_2) + (\Gamma_1 \sqcup \Gamma_2) \vdash_{Dst} e : \mathsf{s}$, from which the desired result follows.

- **Cases** $[CASE_{Dst}]$ **and** $[CASE!_{Dst}]$

  Both follow trivially from the induction hypothesis.

□

Finally, we use the results of these lemmas in order to show that the set of correct mark signatures for a given function is closed w.r.t. the greatest lower bound operator:

Lemma 5.17. *If Φ is the set of correct signatures for a function f, $\sqcap\Phi$ is also a correct signature for f.*

Proof. Let us assume $\sqcap\Phi = \overline{m'_i}^n \to \mathsf{s}$ for some $\overline{m'_i}^n$. By definition of correct signature, we get:

$$\Sigma + [f : \overline{m_i}^n \to \mathsf{s}] + [\overline{x_i : m_i}^n] \vdash_{Dst} e_f : \mathsf{s} \qquad \forall (\overline{m_i}^n \to \mathsf{s}) \in \Phi$$

By Lemma 5.16:

$$\Sigma + [f : \overline{m'_i}^n \to \mathsf{s}] + [\overline{x_i : m_i}^n] \vdash_{Dst} e_f : \mathsf{s} \qquad \forall (\overline{m_i}^n \to \mathsf{s}) \in \Phi$$

Since the set Φ is finite (because so is the set of signatures for f), we can apply Lemma 5.15 repeatedly so as to get:

$$\Sigma + [f : \overline{m'_i}^n \to \mathsf{s}] + [\overline{x_i : \sqcap_{\overline{m_j} \to \mathsf{s} \in \Phi} m_i}^n] \vdash_{Dst} e_f : \mathsf{s}$$

Finally, since $m'_i = \sqcap_{\overline{m_j} \to \mathsf{s} \in \Phi} m_i$, we get that $\overline{m'_i} \to \mathsf{s}$ is a correct signature for f:

$$\Sigma + [f : \overline{m'_i}^n \to \mathsf{s}] + [\overline{x_i : m'_i}^n] \vdash_{Dst} e_f : \mathsf{s}$$

□

The $\sqcap\Phi$ of the lemma above is, in fact, the *minimal* correct signature for the function definition. We have proved that, if a function is typeable, there exists a minimal mark signature for it. Our next step is to show that the inference algorithm shown in this chapter returns that minimal mark signature. The main part of the proof is a completeness result of the $\vdash$ rules with respect to the $\vdash_{Dst}$ type system. It states that, if an expression is typeable under some environment $\Sigma + \Gamma$, the $\vdash$ rules are able to find some (R, D, S) sets such that the environment they define is lower than Γ.

Theorem 5.18. *If $\Sigma + \Gamma \vdash_{Dst} e : \mathsf{s}$, then we can derive $e \vdash_\Sigma (R, D, S)$ for some sets R, D, and S. Moreover, it holds that:*

$$[x : \mathsf{r} \mid x \in R] + [x : \mathsf{d} \mid x \in D] + [x : \mathsf{s} \mid x \in S] \leq \Gamma$$

Proof. By induction on the structure of e. Let us denote by $\Gamma[R, D, S]$ the environment occurring in the left-hand side of the equation above. Without loss of generality, we assume that the last $\vdash_{Dst}$ rule applied is neither $[EXT_{Dst}]$ nor $[WEAK_{Dst}]$. Otherwise, we can assume the existence of a $\Sigma' \subseteq \Sigma$ and a $\Gamma' \leq \Gamma$ such that $\Sigma' + \Gamma' \vdash_{Dst} e : \mathsf{s}$. If we are able to derive $e \vdash_{\Sigma'} (R, D, S)$, it also holds that $e \vdash_\Sigma (R, D, S)$, and if $\Gamma[R, D, S] \leq \Gamma'$, then $\Gamma[R, D, S] \leq \Gamma$, as we want to prove. So, let us distinguish cases on the structure of the expression e:

- **Cases $e \equiv c$, $e \equiv x$, $e \equiv x \,@\, r$ and $e \equiv C\ \overline{a_i}^n \,@\, r$.**

  The result holds trivially in all these cases. For instance:

  $$C\ \overline{a_i}^n \,@\, r \vdash (\emptyset, \emptyset, \{\overline{a_i}^n\}) \quad \text{and} \quad \Gamma[\emptyset, \emptyset, \{\overline{a_i}^n\}] = [\overline{a_i : \mathsf{s}}^n] = \Gamma$$

- **Case $e \equiv f\ \overline{a_i}^n \,@\, \overline{r_j}^m$.**

  Firstly, let us prove that the side conditions in rule $[APP_I]$ hold. Each one is proven by contradiction:

  $R \cap S = \emptyset$

  Assume that $x \in R \cap S$. Since $x \in S$, there exists an $i \in \{1..n\}$ such that $x = a_i$ and $m_i = \mathsf{s}$, where $\Sigma(f) = \overline{m_i}^n \to \mathsf{s}$. Hence, $x \in \text{dom } \bigoplus_{i=1}^n [a_i : m_i]$. On the other hand, and since $x \in R$, $x \in \text{dom } \Gamma_R$, where Γ_R is defined as $[y : \mathsf{r} \mid y \in R]$. This leads to a contradiction as, according to the $[APP_{Dst}]$ rule, Γ_R and $\bigoplus_{i=1}^n [a_i : m_i]$ have disjoint domains.

  $R \cap D = \emptyset$

  The proof is analogous to that of the previous condition.

  $S \cap D = \emptyset$

  If $x \in S$ and $x \in D_i$ for some $i \in \{1..n\}$, then $x = a_i = a_j$ for some $i, j \in \{1..n\}$ such that $m_i = \mathsf{s}$ and $m_j = \mathsf{d}$ (hence $i \neq j$), but this contradicts the fact that $\bigoplus_{i=1}^n [a_i : m_i]$ is well-defined, as the $[APP_{Dst}]$ rule demands.

  $\forall i, j . i \neq j \Rightarrow D_i \cap D_j = \emptyset$

  Let $x \in D_i$ and $x \in D_j$ for some $i \neq j$. This implies that $x = a_i = a_j$ for some $i, j \in \{1..n\}$ such that $m_i = m_j = \mathsf{d}$. Hence $\bigoplus_{i=1}^n [a_i : m_i]$ in $[APP_{Dst}]$ is not well-defined.

$\forall x \in D.isTree(x)$

If there is some $x \in D$ such that $isTree(x)$ does not hold, then there exists some $i \in \{1..n\}$ such that $x = a_i$ and $m_i = \mathtt{d}$, which contradicts the $\bigwedge_{m_i=\mathtt{d}} isTree(a_i)$ property.

Besides this, $\Sigma = [f : \overline{m_i}^n \to \mathtt{s}]$ and:

$$\begin{aligned}\Gamma[R, D, S] &= [y : \mathtt{r} \mid y \in R] + [a_i : \mathtt{s} \mid m_i = \mathtt{s}] + [a_i : \mathtt{d} \mid m_i = \mathtt{d}] \\ &= \Gamma_R + \bigoplus_{i=1}^{n} [a_i : m_i] \\ &= \Gamma\end{aligned}$$

- **Case** $e \equiv \mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2$

Assume $\Sigma_1 + \Gamma_1 \vdash_{Dst} e_1 : \mathtt{s}$, $\Sigma_2 + \Gamma_2 + [x_1 : m_1] \vdash_{Dst} e_2 : \mathtt{s}$. By induction hypothesis we get $e_1 \vdash_{\Sigma_1} (R_1, D_1, S_1)$ and $e_2 \vdash_{\Sigma_2} (R_2, D_2, S_2)$, which imply their corresponding $\vdash_\Sigma$ judgements. We prove by contradiction that the side condition in $[LET_I]$ holds. Assume $x \in R_1 \cup D_1$ such that $x \in fv(e_2)$. Since $\Gamma[R_1, D_1, S_1] \leq \Gamma_1$ holds by induction hypothesis, we get that $\Gamma_1(x) \in \{\mathtt{r}, \mathtt{d}\}$, and, by $[LET_{Dst}]$, $x \notin fv(e_2)$, which contradicts our assumption. We can apply the $[LET_I]$ rule, so $e \vdash (R, D, S)$.

Now we prove that $\Gamma[R, D, S] \leq \Gamma_1 \sqcup \Gamma_2$. Firstly, $\Gamma[R_2, D_2, S_2] \leq \Gamma_2 + [x_1 : m_1]$ by induction hypothesis. From this, it follows that $\Gamma[(R_2, D_2, S_2) \backslash x_1] \leq \Gamma_2$. On the other hand, we get, for every pair of triples (R, D, S), (R', D', S'):

$$\begin{aligned}\Gamma[R, D, S] \sqcup \Gamma[R', D', S'] &= ([x : \mathtt{r} \mid x \in R] + [x : \mathtt{d} \mid x \in D] + [x : \mathtt{s} \mid x \in S]) \sqcup \\ &\quad ([x : \mathtt{r} \mid x \in R'] + [x : \mathtt{d} \mid x \in D'] + [x : \mathtt{s} \mid x \in S']) \\ &= [x : \mathtt{r} \mid x \in R \cup R'] + [x : \mathtt{d} \mid x \in (D \cup D') \backslash (R \cup R')] + \\ &\quad [x : \mathtt{s} \mid x \in (S \cup S') \backslash (D \cup D' \cup R \cup R')] \\ &= \Gamma[(R, D, S) \sqcup (R', D', S')]\end{aligned}$$

Finally, we obtain:

$$\begin{aligned}\Gamma[R, D, S] &= \Gamma[(R_1, D_1, S_1) \sqcup ((R_2, D_2, S_2) \backslash \{x_1\})] \\ &= \Gamma[R_1, D_1, S_1] \sqcup \Gamma[(R_2, D_2, S_2) \backslash \{x_1\}] \\ &\leq \Gamma_1 \sqcup \Gamma_2\end{aligned}$$

The last step justified by the fact that $\sqcup$ is monotone w.r.t. $\leq$. The proof of this fact is straightforward and it shall be omitted here.

- **Case** $e \equiv \mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}^n$

We know that, for each $i \in \{1..n\}$, $\Sigma + \Gamma' + [\overline{x_{ij} : m_{ij}}] \vdash_{Dst} e_i : \mathtt{s}$ for some Γ' and $\overline{m_{ij}}$ such that $\Gamma = \Gamma' \sqcup [x : \mathtt{s}]$. By induction hypothesis, $e_i \vdash (R_i, D_i, S_i)$, and $\Gamma[R_i, D_i, S_i] \leq \Gamma' + [\overline{x_{ij} : m_{ij}}]$. The latter implies that $\Gamma[(R_i, D_i, S_i) \backslash \{\overline{x_{ij}}\}] \leq \Gamma'$. Hence we can apply $[CASE_I]$ to get $e \vdash (R, D, S)$,

where $(R, D, S) = (\bigsqcup_{i=1}^{n}((R_i, D_i, S_i)\backslash\{\overline{x_{ij}}\}) \sqcup (\emptyset, \emptyset, \{x\})$, and we get:

$$\begin{aligned}
\Gamma[R, D, S] &= \Gamma\left[\bigsqcup_{i=1}^{n}((R_i, D_i, S_i)\backslash\{\overline{x_{ij}}\}) \sqcup (\emptyset, \emptyset, \{x\})\right] \\
&= \bigsqcup_{i=1}^{n}\Gamma[(R_i, D_i, S_i)\backslash\{\overline{x_{ij}}\}] \sqcup \Gamma[\emptyset, \emptyset, \{x\}] \\
&\leq \bigsqcup_{i=1}^{n}\Gamma' \sqcup [x : \mathsf{s}] \\
&= \Gamma' \sqcup [x : \mathsf{s}] \\
&= \Gamma
\end{aligned}$$

and the theorem holds.

- **Case** $e \equiv$ **case**! x **of** $\overline{C_i\ \overline{x_{ij}} \to e_i}^{n}$

  Again, let us assume that $\Sigma + \Gamma' + [\overline{x_{ij} : m_{ij}}] \vdash_{Dst} e_i : \mathsf{s}$, for every $i \in \{1..n\}$, and for some Γ' such that $\Gamma = \Gamma_R \sqcup (\Gamma'\backslash x) + [x : \mathsf{d}]$, where Γ_R is as defined in $[CASE!_{Dst}]$. By induction hypothesis, for every $i \in \{1..n\}$, $e_i \vdash (R_i, D_i, S_i)$ holds for some (R_i, D_i, S_i) such that $\Gamma[R_i, D_i, S_i] \leq \Gamma' + [\overline{x_{ij} : m_{ij}}]$. This implies, for every $i \in \{1..n\}$, $\Gamma[(R_i, D_i, S_i)\backslash\{\overline{x_{ij}}\}] \leq \Gamma'$. Let us define

$$(R', D', S') = \left(\bigsqcup_{i=1}^{n}((R_i, D_i, S_i)\backslash\{\overline{x_{ij}}, x\})\right) \sqcup (R_{SHR}, \emptyset, \emptyset)$$

  where R_{SHR} is as defined in $[CASE!_I]$. We obtain:

$$\Gamma[R', D', S'] = \left(\bigsqcup_{i=1}^{n}\Gamma[(R_i, D_i, S_i)\backslash\{\overline{x_{ij}}, x\}]\right) \sqcup \Gamma[R_{SHR}, \emptyset, \emptyset] \leq \Gamma_R \sqcup \Gamma'\backslash\{x\}$$

  If we are able to apply $[CASE!_I]$, we get $e \vdash (R, D, S)$, where $(R, D, S) = (R', D' \cup \{x\}, S')$, and hence, since (R', D', S') does not contain the discriminant of the **case**:

$$\Gamma[R, D, S] = \Gamma[R', D', S'] + [x : \mathsf{d}] \leq \Gamma_R \sqcup (\Gamma'\backslash\{x\}) + [x : \mathsf{d}] = \Gamma$$

  So, let us prove that the side conditions in $[CASE!_I]$ hold. Again, we prove each of these by contradiction:

  $\forall i \in \{1..n\}.(P_i\backslash Rec_i) \cap (D_i \cup R_i) = \emptyset$

  Assume that $x_{ij} \in P_i\backslash Rec_i$ (that is, $j \notin RecPos(C_i)$) and that $x_{ij} \in D_i \cup R_i$. Since $\Gamma[R_i, D_i, S_i] \leq \Gamma' + [x_{ij} : m_{ij}]$, we get that $m_{ij} \in \{\mathsf{d}, \mathsf{r}\}$. But the condition $inh_{j,C_i}(m_{ij})$ in $[CASE!_{Dst}]$ demands $m_{ij} = \mathsf{s}$.

  $\forall i \in \{1..n\}.Rec_i \cap R_i = \emptyset$

  Assume that $x_{ij} \in Rec_i$ (that is, $j \in RecPos(C_i)$. Similarly as above, $x_{ij} \in R_i$ implies $m_{ij} = \mathsf{r}$, which contradicts the $inh_{j,C_i}(m_{ij})$ condition in $[CASE!_{Dst}]$, which demands $m_{ij} = \mathsf{d}$.

  $\forall i \in \{1..n\}.(R_{SHR} \cup \{x\}) \cap fv(e_i) = \emptyset$

  Assume that $z \in R_{SHR} \cup \{x\}$ and $z \in fv(e_i)$ for some $i \in \{1..n\}$. This implies $z \in R \cup \{x\}$ (being R as defined in $[CASE!_{Dst}]$), since $R = R_{SHR}$. By the latter rule, $z \notin fv(e_i)$, which leads to a contradiction.

□

Given a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$, our algorithm performs several iterations of the $\vdash$ rules until it reaches a mark signature that coincides with the one initially assumed for f. An iteration can be formalized as a function φ_f which, given a mark signature $\overline{m_i}^n \to \mathsf{s}$ returns the mark signature which would be given to the next iteration.

$$\varphi_f(\overline{m_i} \to \mathsf{s}) = \overline{m'_i} \to \mathsf{s}, \text{ where: } \quad e_f \vdash_{\Sigma \uplus [f:\overline{m_i} \to \mathsf{s}]} (\emptyset, D, S) \qquad \forall i \in \{1..n\}. m'_i = \begin{cases} \mathsf{d} & x_i \in D \\ \mathsf{s} & x_i \notin D \end{cases} \tag{5.7}$$

The *inferMarksFP* algorithm stops when the mark signature is given for f is a fixed point of φ_f. A consequence of the previous theorem is that the minimal correct signature of a function is a fixed point of φ_f.

Corollary 5.19. *If Φ is the set of correct signatures for $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$, then $\sqcap\Phi \in Fix(\varphi_f)$.*

Proof. Let $\sqcap\Phi = \overline{m_i} \to \mathsf{s}$ be the minimal signature. By Lemma 5.17 this signature is correct, so we get:

$$\underbrace{\Sigma + [f : \overline{m_i} \to \mathsf{s}]}_{\Sigma'} + [\overline{x_i : m_i}] \vdash_{Dst} e_f : \mathsf{s}$$

where none of the m_i is an r mark. By Theorem 5.18 there exists some (R, D, S) such that $e \vdash_{\Sigma'} (R, D, S)$ and $[x : \mathsf{r} \mid x \in R] + [x : \mathsf{d} \mid x \in D] + [x : \mathsf{s} \mid x \in S] \leq [\overline{x_i : m_i}]$. Since none of the m_i is an in-danger mark, $R = \emptyset$. Moreover, all the variables in the environment in the left-hand side must be one of the x_i. By Theorem 5.8:

$$\Sigma + [f : \overline{m_i} \to \mathsf{s}] + [x_i : \mathsf{d} \mid x_i \in D] + [x_i : \mathsf{s} \mid x_i \in S] \vdash_{Dst} e_f : \mathsf{s}$$

By applying $[EXT_{Dst}]$ rule, we get:

$$\Sigma + [f : \overline{m_i} \to \mathsf{s}] + [x_i : \mathsf{d} \mid x_i \in D] + [x_i : \mathsf{s} \mid x_i \notin D] \vdash_{Dst} e_f : \mathsf{s} \tag{5.8}$$

Let us define, for each $i \in \{1..n\}$:

$$m'_i = \begin{cases} \mathsf{d} & x_i \in D \\ \mathsf{s} & x_i \notin D \end{cases}$$

so that $\varphi_f(\overline{m_i}^n \to \mathsf{s}) = \overline{m_i}' \to \mathsf{s}$. We can transform (5.8) into the following:

$$\Sigma + [f : \overline{m_i} \to \mathsf{s}] + [\overline{x_i : m'_i}] \vdash_{Dst} e_f : \mathsf{s}$$

Let us prove that $m'_i \leq m_i$ for each $i \in \{1..n\}$. If $m'_i = \mathsf{s}$ we are done. If $m'_i = \mathsf{d}$, then $x_i \in D$, and hence $m_i = \mathsf{d}$, since $[x : \mathsf{d} \mid x \in D] + [x : \mathsf{s} \mid x \in S] \leq [\overline{x_i : m_i}]$. Hence, $\overline{m'_i} \to \mathsf{s} \leq \overline{m_i} \to \mathsf{s}$, and we can apply Lemma 5.16 to the previous judgement:

$$\Sigma + [f : \overline{m'_i} \to \mathsf{s}] + [\overline{x_i : m'_i}] \vdash_{Dst} e_f : \mathsf{s}$$

Hence $\overline{m_i}' \to \mathsf{s}$ is a correct signature for f that is lower than $\overline{m_i} \to \mathsf{s}$. However, recall that $\overline{m_i} \to \mathsf{s}$ is the greatest lower bound, so $m'_i = m_i$ for every $i \in \{1..n\}$, and hence $\varphi_f(m_i \to \mathsf{s}) = m_i \to \mathsf{s}$. □

Reciprocally, every fixed point of φ_f is a correct mark signature.

Lemma 5.20. *If Φ is the set of correct signatures for $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$, then $Fix(\varphi_f) \subseteq \Phi$.*

Proof. If $\varphi_f(\overline{m_i} \to \mathsf{s}) = \overline{m_i} \to \mathsf{s}$, we get $e_f \vdash_{\Sigma \uplus [f:\overline{m_i}\to\mathsf{s}]} (\emptyset, D, S)$ for some D and S. By Theorem 5.8, we get:

$$\Sigma + [f : \overline{m_i} \to \mathsf{s}] + [x : \mathsf{d} \mid x \in D] + [x : \mathsf{s} \mid x \in S] \vdash_{Dst} e_f : \mathsf{s}$$

Since $D \cup S \subseteq fv(e_f) \subseteq \{\overline{x_i}\}$, we can apply the $[EXT_{Dst}]$ rule in order to get:

$$\Sigma + [f : \overline{m_i} \to \mathsf{s}] + [x_i : \mathsf{d} \mid x_i \in D] + [x_i : \mathsf{s} \mid x_i \notin D] \vdash_{Dst} e_f : \mathsf{s}$$

By the definition of m_i' in (5.7) we can transform the judgement above into the following:

$$\Sigma + [f : \overline{m_i} \to \mathsf{s}] + [x_i : \mathsf{d} \mid m_i = \mathsf{d}] + [x_i : \mathsf{s} \mid m_i = \mathsf{s}] \vdash_{Dst} e_f : \mathsf{s}$$

which is equivalent to:

$$\Sigma + [f : \overline{m_i} \to \mathsf{s}] + [\overline{x_i : m_i}] \vdash_{Dst} e_f : \mathsf{s}$$

Therefore, $\overline{m_i} \to \mathsf{s}$ is a correct signature for f. □

From the last two results we can prove that the inference algorithm returns the minimal correct type signature. It is straightforward to see that *inferMarksDef* returns the least fixed point of φ_f, since it performs the computation of a Kleene's ascending chain, starting from the bottommost element $\overline{\mathsf{s}} \to \mathsf{s}$. On the one hand, the least fixed point of φ_f (denoted $lfp(\varphi_f)$) is a correct signature by Lemma 5.20, so $lfp(\varphi_f) \geq \sqcap\Phi$, where Φ denotes the set of correct signatures for f. On the other hand, $\sqcap\Phi$ is a fixed point, so $lfp(\varphi_f) \leq \sqcap\Phi$. As a consequence, $lfp(\varphi_f) = \sqcap\Phi$, which proves that the *inferMarksDef* computes the minimal mark signature for f.

5.4.4 Efficiency

Assume we execute the *inferMarksDef* function on a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$. If s denotes the size of the AST corresponding to e_f, a single traversal of the tree has $\mathcal{O}(s)$ complexity if we assume constant-time set operations. However, the complexity of set operations $\cup, \cap, \setminus$ is linear with respect the sizes of the sets to which they are applied. In our case, these sets contain variables occurring in e_f. Since the worst-case scenario assumes a number of variables in the AST proportional to s, a single traversal of the tree has a worst-case $\mathcal{O}(s^2)$ time complexity. Besides this, the algorithm performs, in the worst case, as many iterations as the number of parameters n. As a consequence, the worst-case time complexity of the algorithm for inferring the mark signature of a function definition is in $\mathcal{O}(ns^2)$. Nevertheless, all the examples shown in this chapter only require two iterations: The first one computes the fixed point, whereas the second one checks that the result is a fixed point. Thus we expect that the average-case complexity of our algorithm is $\mathcal{O}(s^2)$.

5.5 Conclusions and related work

The mark inference algorithm introduced in this chapter represents a notable improvement with respect to its predecessor, described in [81]. The main reason behind this fact is the definition of a total order on the different marks, which allows the algorithm to assign the strongest mark to a variable at a given

context, and to weaken that mark in a higher context, if necessary. This results in a simpler algorithm, in which a single bottom-up traversal is done in each iteration. Previous work required some interleaving of bottom-up and top-down traversals with two different set of rules. As a consequence, we get a simpler correctness proof. Moreover, the distinction between strongest and weakest marks allows the $\vdash_{Dst}$ rules to have a minimal typing environment under which a given expression is typeable (Lemma 5.17). This allows us to prove the completeness of our algorithm: if an expression is typeable, the algorithm finds the minimal environment typing that expression.

The examples shown in Section 3.5 have been typed by assuming the most precise analysis satisfying the correctness conditions of Section 3.6.3. On the contrary, in this chapter we have used the sharing analysis of [98]. As we have shown in Section 5.3, the sharing information has to be adjusted in some cases. This motivates the need for a more precise analysis, whose study is subject of future work.

In Section 3.8 we showed the relation between our type system and other approaches related to functional languages. In particular, a detailed comparison was done with Aspinall and Hofmann's type system [11] based on usage aspects (UAPL in the following). An usage aspect inference algorithm for this system is due to Konečný [68]. It is worth highlighting the similarities and differences of his inference algorithm with ours.

- The set of typing rules is, in both cases, non-deterministic. Our inference algorithm is given by a set of syntax-directed rules, whereas UAPL features a deterministic type-checking strategy in which some rules are given more (or less) priority than the remaining ones.

- In UAPL, the (RAISE) rule takes the highest priority. This rule upgrades the usage aspects of the variables in the type environment from 2 (not destroyed, but shares with result) to 3 (not destroyed, does not share with result) in those cases when the result is of a basic (i.e. heap-free) type. This is not necessary in our type system, since the sharing analysis of *Safe* is aware of the fact that values of basic types do not take space in the heap, so they cannot share anything with the remaining variables. Moreover, all basic types have a safe mark in our system.

- The (WEAK) and (DROP) rules of UAPL, which are similar to the [EXT] and [WEAK] rules of *Safe*, are given the lowest priority, and are only used to make the premises in the same rule match each other. In *Safe*, the $\sqcup$ operator on triples has a similar goal.

- The inference of the signature of recursive function definitions follows a similar pattern in both cases. It starts by assuming the strongest signature for the function being inferred, and performs a fixed point iteration. This is not very surprising, as this approach (known as the Kleene's ascending chain) is commonly used when the set of function signatures has the structure of a complete partial order. In particular, the sharing analysis of [98] also works in this way.

Although the intended semantics of safe, condemned, and in-danger types is substantially different from the usage aspects of UAPL, the fact of defining a total order between the marks/usage aspects makes the inference algorithms of UAPL and *Safe* very similar. A contribution of our algorithm with respect to UAPL's is the explicit formalization of the algorithm itself, together with a comprehensive proof of its correctness and completeness.

Chapter 6

Certified absence of dangling pointers

6.1 Introduction

Let us briefly recap the results of last chapters. On the one hand, we have described in Chapter 4 an algorithm for annotating a *Safe* program with region variables, such that the resulting program is well-typed w.r.t. the $\vdash_{Reg}$ rules. On the other hand, we have developed in Chapter 5 an algorithm that checks whether the destruction of cells, managed by the programmer, does not lead to an ill-checked program regarding the $\vdash_{Dst}$ rules. If a *Safe* program is accepted by both algorithms, we can ensure, by the correctness proof given in Chapter 3, that this program is pointer-safe. The aim of this chapter is to formalize this property and its corresponding proof for our specific program, so both things can be mechanically checked. This is an example of a *proof-carrying code* scheme [88], in which the compiler generates, besides the target program, a proof (*certificate*) of the presence of a property which the code consumer expects to hold. Both the program and the certificate are sent to the code consumer, which checks the proof against the program before executing the latter.

We have used the Isabelle proof assistant for encoding all these properties and their proofs. Isabelle provides a meta-logic in which Higher-order logic (HOL) is specified. Both things are commonly referred to with the name *Isabelle/HOL*. This tool provides a framework in which we can specify definitions, theorems, and their proofs. Isabelle/HOL checks whether the theorems actually follow from the given proofs. In this chapter we explain how to generate Isabelle/HOL scripts encoding the above mentioned pointer-safety properties. It is not surprising that most of the proofs and definitions relevant to this chapter are those occurring in Section 3.6, where the pointer-safety of well-typed programs was established. However, the proof rules of this chapter are based on the original *Safe*'s type system described in [84], from which the type system of this thesis stems. As a consequence, there are some slight differences in the certificate w.r.t. the proof shown in this thesis. It is subject of future work to adapt the proofs of this chapter so as to fit those of Section 3.6.

This chapter is based on the work described in [36]. Unlike previous chapters, it is more oriented towards implementation issues. In particular we explain in detail the automatic generation of Isabelle/HOL scripts, since this is the main contribution of the author of this thesis to [36]. More details about the formalization itself can be found in [37], or in de Dios' PhD thesis [35] (the latter in Spanish). However, we briefly describe in Section 6.2 the properties to be proved, and the information that the certificate is expected to contain for a specific program. Section 6.3 deals with the generation of the certificate itself. In Section 6.4 we measure, by means of some case studies, the length of the obtained proofs w.r.t. the original *Core-Safe* code. Finally, Section 6.5 concludes.

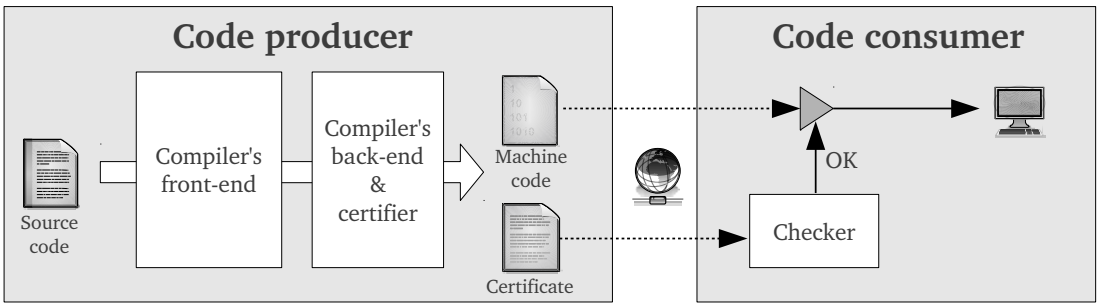


Figure 6.1: Standard PCC paradigm. The certificate refers to the low-level machine code, which is sent to the consumer.

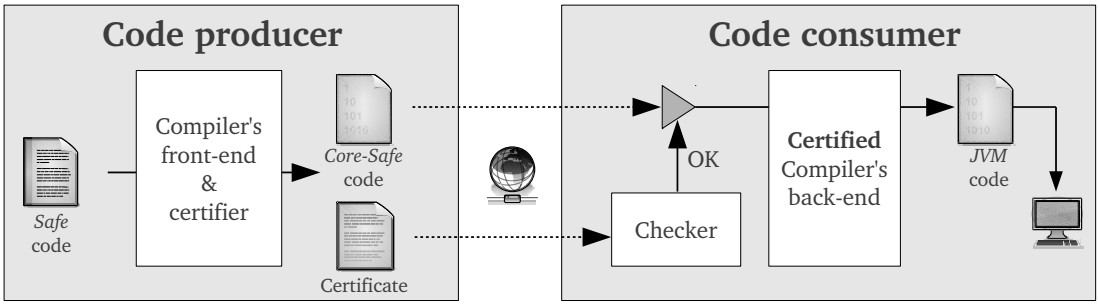


Figure 6.2: Our PCC paradigm.

6.2 Preliminaries

The standard approach to PCC is depicted in Figure 6.1. In this scenario, the compiler attaches a certificate to the resulting machine code, and the proof is performed at machine level. That is, the properties being certified are checked against the machine code. The code consumer receives both low-level code and certificate and checks the validity of the latter. If it is correct, it allows the execution of the code on the target machine. A variation of this scheme advances the generation of the certificate before the translation into machine code, but after the front-end phase, in which the properties of interest are known to hold. In this case, the certificate refers to the intermediate representation resulting from the front-end, and the back-end translates both this intermediate representation and the certificate into a lower-level representation, which is sent to the code consumer. The main drawback of these approaches is that the involved certificates are of considerable size, which results in longer checking times.

Our approach consists in postponing the machine code generation after the certificate is checked (Figure 6.2). The compiler generates a certificate proving the pointer-safety properties of a *Core-Safe* program. After the consumer has checked its validity against the latter, the program is translated into Java bytecode and executed. Since the certificates obtained in this way refer to the properties of a higher-level language (*Core-Safe*) they are smaller and easier to check than their lower-level counterparts. They also save the effort of translating the certificate into a lower-level variant. However, this approach relies on the correctness of the compiler's back-end that translates *Core-Safe* code into Java byte-code. There are two possibilities:

1. Assume the correctness of this translation phase. In this case, the back-end is said to be part of the *trusted code base*.

2. Mechanically prove and certify these translation phases.

In [39] and [38] the authors prove in Isabelle/HOL, respectively, the correctness of the translation from *Core-Safe* into SVM code (see Section 2.6) and its further translation into Java bytecode. The definitions of these translations are specified in Isabelle/ML (a functional, ML-alike language), and a Haskell implementation is automatically extracted from these definitions. Thus, if the Isabelle/HOL definitions are proven correct, so is their corresponding implementation. A drawback of this approach to PCC is that the consumer receives the *Core-Safe* code, which it may be easier to reverse-engineer than its bytecode equivalent. Besides this, part of the compiling process (translation into bytecode) is delegated to the code consumer. This is a non-standard approach in compilers.

Once we know the level at which the certificate is going to be targeted, the next step is to define the contents of the certificate (i.e. *what* to prove). Given a *Core-Safe* program *prog*, the certificate is expected to prove the following fact:

$$\textit{prog} \text{ does not access dangling pointers} \tag{6.1}$$

However, it would be a very poor solution if each certificate contained the full proof of this fact, since several parts of this proof could also be applied to every *Core-Safe* program. Since we have devised a generic way (namely, a type system) to prove a program pointer-safe, it seems sensible to split the proof of (6.1) into two different facts:

$$\text{Well-typed programs do not access dangling pointers} \tag{6.2}$$

$$\textit{prog} \text{ is a well-typed program} \tag{6.3}$$

It is trivial to see that (6.1) follows from these facts. The first one is generic, whereas the second one is program-specific. The idea of the certificate generation is to ask the compiler to deliver the static information inferred during the type inference phase (Chapters 4 and 5) and to use this information for proving (6.3). This can be done with a set of *proof rules*. There exists a proof rule for every syntactical construct of the language. Given an expression of the program, its certificate must ensure that the premises of a proof rule hold before applying that rule. These premises are known as *proof obligations*. The fact (6.2)is proved by relating the static information (types) with the dynamic properties a program is expected to satisfy at runtime (absence of dangling pointers). For every proof rule there exists a lemma establishing this relation. These lemmas are proved once for all, and can be stored in the code consumer side, since they are common to all *Core-Safe* programs. The program-specific part of the certificate consists in establishing the validity of the necessary proof obligations for the given program, and applying the corresponding proof rules. Since there are general theorems relating these proof rules with the safety properties of the language, the specific program can be proved pointer-safe by applying these theorems. This strategy results in small certificates and short checking times, as the total amount of work is linear w.r.t. the size of the program. The heaviest part of the proof (namely, the database of proved proof rules) has been done in advance and is reused by each certified program.

Given the above, our first step is to present these proof rules with their corresponding safety properties. There is some resemblance with the typing rules of Chapter 3, but this resemblance becomes more apparent when comparing the proof rules with those in [84]. In the same way we have considered the inference of regions and marks separately, the certificates also distinguishes the part of pointer safety regarding implicit region deallocation from that related to explicit destruction. Thus we have two sets of proof rules, which are described in next sections.

6.2.1 Rules regarding region deallocation

The aim of this part of the certificate is to prove the preservation of the consistency property through the execution of a program. Recall from Section 3.6.1 the existence of a region instantiation η associating RTVs with actual runtime region numbers. If a DS is inferred to have ρ as the outermost RTV in its type, and this DS lives in region n at runtime, then $\eta(\rho) = n$. Consistency property tells us that the correspondence between the static region types, and the actual regions where the data structures are stored in the heap, do not contradict η. This part of the certificate proves that, if the initial configuration of a program is consistent, so is every intermediate configuration in its execution, and so is the final configuration. Since the execution of a program starts with an empty heap, which is trivially consistent, all the intermediate configurations are also consistent. This part of the certificate deals exclusively with regions, and not with the condemned, safe or in-danger mark of a type, so the proof rules roughly resemble the $\vdash_{Reg}$ typing rules of Section 3.7. Thus we can instantiate the facts (6.2) and (6.3) as follows:

$$\vdash_{Reg}\text{-typed programs preserve consistency along their execution} \tag{6.4}$$

$$\textit{prog} \text{ is a } \vdash_{Reg}\text{-typed program} \tag{6.5}$$

This implies that the deallocation of the topmost region of the heap cannot create dangling pointers at runtime, since η always maps the ρ_{self} region type to the topmost region, and this RTV cannot appear in the signature of a function, so neither the input nor the output of the function will be located in that region.

In Figure 6.3 we show the proof rules. We have typing environments Γ_R mapping program variables to safe types and region arguments to region types. When mechanically checking the certificate, it is convenient to have the types of already certified functions into a separate environment Σ_R, and a global environment Γ_C giving the polymorphic most general type of data constructors. Hence Γ_R only contains variables in scope. In the rules we have judgements of the form $e, \Sigma_R \vdash \Gamma_R \rightsquigarrow s$, where $e \in \textbf{Exp}$ and $s \in \textbf{SafeType}$. They have a similar meaning to the judgement $\Gamma_R \cup \Sigma_R \cup \Gamma_C \vdash_{Reg} e : s$.

In addition to the Γ_R and s, which are provided by the compiler, the certificate needs additional information in function and constructor applications. Recall that the types occurring in these expressions may be instances of the signature of the corresponding function/constructor, so the compiler generates a *type instantiation mapping* μ in these cases. The certificate checks whether this mapping correctly associate the types of the formal parameters with the types of the corresponding actual arguments in the function/constructor application. This motivates the following definition.

Definition 6.1. Given the instantiated types $\overline{s_i}^n$, the instantiated region types $\overline{\rho_j}^m$, the arguments of the application $\overline{a_i}^n$, $\overline{r_j}^m$ and a region typing environment Γ_R, we say that the application is *argument preserving*, denoted $argP(\overline{s_i}^n, \overline{\rho_j}^m, \overline{a_i}^n, \overline{r_j}^m, \Gamma_R)$ iff $\forall i \in \{1..n\}.s_i = \Gamma_R(a_i)$ and $\forall j \in \{1..m\}.\rho_j = \Gamma_R(r_j)$

Thus, given the generic type scheme $\forall \overline{\alpha\rho}.\overline{s_i} \to \overline{\rho_j} \to s$ of a function g, and a specific application $g\ \overline{a_i}\ @\ \overline{r_j}$, the type instantiation mapping μ is valid provided $argP(\overline{\mu(s_i)}, \overline{\mu(\rho_j)}, \overline{a_i}, \overline{r_j}, \Gamma_R)$ holds. This is what rules $[APP_{RegC}]$ and $[CONS_{RegC}]$ ensure.

The certificate incrementally constructs a global environment Σ_R keeping the most general types of the functions already certified. Regarding constructors, the certificate demands a well-formedness condition on their types, namely, that the type of the only region argument of a constructor is the outermost RTV of the type being constructed, and that all the RTVs occurring in the type of the arguments of a constructor also occur in the type of its result.

$$\frac{}{c, \Sigma_R \vdash \Gamma_R \rightsquigarrow B} \; [LIT_{RegC}]$$

$$\frac{}{x, \Sigma_R \vdash \Gamma_R \rightsquigarrow \Gamma_R(x)} \; [VAR_{RegC}]$$

$$\frac{\Gamma_R(x) = T\ \overline{s_i}^m @\ \rho_1 \ldots \rho_l \quad \Gamma_R(r) = \rho'}{x\ @\ r, \Sigma_R \vdash \Gamma_R \rightsquigarrow T\ \overline{s_i}^m @\ \rho_1 \ldots \rho_{l-1}\rho'} \; [COPY_{RegC}]$$

$$\frac{\Gamma_C(C) = \overline{s_i}^n \to \rho \to s_C \quad wellT(\overline{s_i}^n, \rho, s_C) \quad argP(\overline{\mu(s_i)}^n, \mu(\rho), \overline{a_i}^n, r, \Gamma_R) \quad s = \mu(s_C)}{C\ \overline{a_i}^n\ @\ r, \Sigma_R \vdash \Gamma_R \rightsquigarrow s} \; [CONS_{RegC}]$$

$$\frac{\Sigma_R(g) = \overline{s_i}^n \to \overline{\rho_j}^m \to s_g \quad \rho^g_{self} \notin regions(s_g) \quad argP(\overline{\mu(s_i)}^n, \overline{\mu(\rho_j)}^m, \overline{a_i}^n, \overline{r_j}^m, \Gamma_R) \quad s = \mu(s_g)}{g\ \overline{a_i}^n\ @\ \overline{r_j}^m, \Sigma_R \vdash \Gamma_R \rightsquigarrow s} \; [APP_{RegC}]$$

$$\frac{e_1, \Sigma_R \vdash \Gamma_R \rightsquigarrow s_1 \quad e_2, \Sigma_R \vdash \Gamma_R \uplus [x_1 \mapsto s_1] \rightsquigarrow s_2}{\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma_R \vdash \Gamma_R \rightsquigarrow s_2} \; [LET_{RegC}]$$

$$\frac{\begin{array}{c}\forall i.\ \Sigma_R(C_i) = \overline{s_{ij}}^{n_i} \to \rho \to s \qquad \forall i.\ wellT(\overline{s_{ij}}^{n_i}, \rho, s) \\ \forall i.\ e_i, \Sigma_R \vdash \Gamma_R \uplus [\overline{x_{ij} \to \mu(s_{ij})}^{n_i}] \rightsquigarrow s' \qquad \Gamma_R(x) = \mu(s)\end{array}}{\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n, \Sigma_R \vdash \Gamma_R \rightsquigarrow s'} \; [CASE_{RegC}]$$

$$\frac{\begin{array}{c}\forall i.\ \Sigma_R(C_i) = \overline{s_{ij}}^{n_i} \to \rho \to s \qquad \forall i.\ wellT(\overline{s_{ij}}^{n_i}, \rho, s) \\ \forall i.\ e_i, \Sigma_R \vdash \Gamma_R \uplus [\overline{x_{ij} \to \mu(s_{ij})}^{n_i}] \rightsquigarrow s' \qquad \Gamma_R(x) = \mu(s)\end{array}}{\textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n, \Sigma_R \vdash \Gamma_R \rightsquigarrow s'} \; [CASE!_{RegC}]$$

$$\frac{\begin{array}{c}f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f \\ \Gamma_R = [\overline{x_i \mapsto s_i}^n, \overline{r_j \mapsto \rho_j}^m, self \mapsto \rho_{self}] \quad e_f, \Sigma_R \uplus \{f \mapsto \overline{s_i}^n \to \overline{\rho_j}^m \to s_f\} \vdash \Gamma_R \rightsquigarrow s_f\end{array}}{e_f, \Sigma_R \vdash \Gamma_R \rightsquigarrow s_f} \; [REC_{RegC}]$$

Figure 6.3: Proof rules regarding regions.

Definition 6.2. Predicate $wellT(\overline{s_i}^n, \rho, s)$, read *well-typed*, is defined as follows:

$$wellT(\overline{s_i}^n, \rho, T\ \overline{s'_i}\ @\ \overline{\rho_j}) \equiv \rho_m = \rho \wedge \bigcup_{i=1}^{n} regions(s_i) \subseteq regions(T\ \overline{s'_i}\ @\ \overline{\rho_j})$$

The first phase of the region inference algorithm (Section 4.3) generates type schemes for constructors that always satisfy this property. There is an additional rule $[REC_{RegC}]$ for typing the body of a function definition, which is similar to the [FUN] rule of Figure 3.9. This rule states that whenever we are able to prove a judgement $e_f, \Sigma_R \vdash \Gamma_R \rightsquigarrow s_f$ by assuming that Σ_R contains a type scheme for f matching the contents of Γ_R, we can do the same without that assumption. The correctness proof of this rule is technically involved and we refer to [37] for more details on it.

So far for the static concepts aimed at proving (6.5). We move now to the dynamic or runtime ones in order to prove (6.4). First we define the notion of admissible region instantiation.

Definition 6.3. Assume k denotes the topmost region of a given heap. We say that the region instantiation η is admissible, denoted $admissible(\eta, k)$ iff:

$$\rho^f_{self} \in \text{dom } \eta \ \wedge\ \eta(\rho^f_{self}) = k \ \wedge\ \forall \rho \in \text{dom } \eta \backslash \{\rho^f_{self}\}.\eta(\rho) < k$$

Basically, an admissible η maps the type of the *self* region to the topmost region in the heap, and the rest of the RTVs to the regions situated below it. The fact that a region instantiation connects the static information Γ_R with the dynamic one (E, h) is formalised by means of the important notion of consistency, defined as follows:

Definition 6.4. We say that the mappings Γ_R, η, the runtime environment E, and the heap h are *consistent*, denoted $consistent(\Gamma_R, \eta, E, h)$, if:

1. $\forall x \in \text{dom } E\ .\ consistent(\Gamma_R(x), \eta, E(x), h)$ where:

$$\begin{array}{lcl}
consistent(B, \eta, c, h) & = & true \\
consistent(\alpha, \eta, v, h) & = & true \\
consistent(s, \eta, p, h) & \Leftarrow & p \notin \text{dom } h \\
consistent(T\ \overline{s'_i}^m @\ \overline{\rho_j}^l, \eta, p, h) & \Leftarrow & \exists j\ C\ \overline{v_k}^n\ \mu\ \overline{s_{kC}}^n\ \overline{\rho_{jC}}^l\ .\ h(p) = (j, C\ \overline{v_k}^n) \\
 & \wedge & \rho_l \in \text{dom } \eta\ \wedge\ \eta(\rho_l) = j \\
 & \wedge & \Gamma_C(C) = \overline{s_{kC}}^n \rightarrow \rho_{lC} \rightarrow T\ \overline{s'_{iC}}^m @\ \overline{\rho_{jC}}^l \\
 & \wedge & \mu(T\ \overline{s'_{iC}}^m @\ \overline{\rho_{jC}}^l) = T\ \overline{s'_i}^m @\ \overline{\rho_j}^l \\
 & \wedge & \forall k \in \{1..n\}\ .\ consistent(\mu(s_{kC}), \eta, v_k, h))
\end{array}$$

2. $\forall r \in \text{dom } E\ .\ \Gamma_R(r) \in \text{dom } \eta\ \wedge\ E(r) = (\eta \circ \Gamma_R)(r)$

3. $self \in \text{dom } E\ \wedge\ \Gamma_R(self) = \rho^f_{self}$

Notice the similarity with the notion of consistency introduced in Section 3.6.1. In fact, $consistent(\Gamma_R, \eta, E, h)$ holds if and only if η is consistent with $build^*(h, E, \Gamma_R)$.

We are ready to specify (6.4) by means of a *static assertion* relating the static and dynamic properties referred to regions. A judgement of the form $e : [\![\Gamma_R, s]\!]$ defines that, if the expression e is evaluated with an environment E, a heap h with k regions, and an admissible mapping η consistent with Γ_R, then the final heap h' and the final value v are consistent with η. Formally:

Definition 6.5. Let us define the following predicates:

$$
\begin{array}{lcl}
PR1 & \equiv & E \vdash h, k, e \Downarrow h', k, v \\
PR2 & \equiv & \text{dom } E \subseteq \text{dom } \Gamma_R \\
PR3 & \equiv & admissible(\eta, k) \\
PR4 & \equiv & consistent(\Gamma_R, \eta, E, h) \\
PR5 & \equiv & consistent(s, \eta, v, h')
\end{array}
$$

An expression e satisfies the pair (Γ_R, s), denoted $e : [\![\Gamma_R, s]\!]$ iff:

$$\forall E, h, k, h', v, \eta. PR1 \wedge PR2 \wedge PR3 \wedge PR4 \rightarrow PR5$$

Notice that this definition only ensures the consistency of the final configuration assuming the consistency of the initial one, but it does not prevent us from having an non-consistent intermediate configuration. The following theorem states that this does not happen if the initial configuration is admissible and consistent.

Theorem 6.6 (consistency)**.** *Let us assume a $\Downarrow$-judgement:*

$$E \vdash h, k, e \Downarrow h', k, v \tag{6.6}$$

For every Γ_R and s such that $e : [\![\Gamma_R, s]\!]$: if $PR2(E, \Gamma_R)$, $PR3(\eta, k)$, $PR4(\Gamma_R, \eta, E, h)$ hold, then for every judgement $E_i \vdash h_i, k_i, e_i \Downarrow h'_i, k_i, v_i$ in the derivation of (6.6), it holds that $PR3(\eta_i, k_i)$, $PR4(\Gamma_{R,i}, \eta_i, E_i, h_i)$ and $PR5(s_i, \eta_i, v_i, h'_i)$.

Proof. It follows from the proof of Theorem 3.26 (see [35]). □

We know that *PR2*, *PR3* and *PR4* trivially hold for the initial state with the empty heap, the initial runtime environment $[self \mapsto 0]$ and the initial region type environment $[self : \rho_{self}]$, so *PR3*, *PR4* and *PR5* hold across the whole program derivation.

The following theorem relates the proof rules of Figure 6.3 to the assertions defined in Definition 6.5.

Theorem 6.7 (soundness)**.** *If $e, \Sigma_R \vdash \Gamma_R \rightsquigarrow s$, then $e, \Sigma_R : [\![\Gamma_R, s]\!]$.*

Proof. If follows from Theorem 3.26 (see [35]). □

6.2.2 Rules regarding explicit deallocation

This part of the certificate deals with the preservation of the closedness property of the heap in presence of explicit destruction. A heap is said to be closed with respect an expression e if it is not possible to reach a dangling pointer from the variables occurring free in e. In a similar way to the consistency property, the certificate proves that, if the initial heap is closed, so is the final heap and the intermediate heaps taking place in the evaluation of the expression. The initial (empty) heap is always closed w.r.t. the main expression of the program (as the latter does not have free variables), so this fact proves the preservation of closedness during the whole execution of the program. Again, this part of the certificate involves proving the following facts:

$$\text{The heap remains closed as } \vdash_{Dst}\text{-typed programs are executed} \tag{6.7}$$

$$prog \text{ is a } \vdash_{Dst}\text{-typed program} \tag{6.8}$$

The proof rules are shown in Figure 6.4. In this case, the static information consists of a *mark environment* Γ_D assigning each variable in scope a mark in the set $\{\mathtt{d}, \mathtt{r}, \mathtt{s}\}$, as in the $\vdash_{Dst}$ type system. Again,

$$\frac{}{c, \Sigma_D \vdash (\emptyset, \emptyset)} \; [LIT_{DstC}]$$

$$\frac{}{x, \Sigma_D \vdash (\{x\}, \Gamma_D + [x : \mathsf{s}])} \; [VAR_{DstC}]$$

$$\frac{x \in \mathrm{dom}\, \Gamma_D \quad \Gamma_D \text{ well formed}}{x \mathbin{@} r, \Sigma_D \vdash (\{x\}, \Gamma_D)} \; [COPY_{DstC}]$$

$$\frac{\Gamma_D = [\overline{a_i : \mathsf{s}}^n]}{C\ \overline{a_i}^n \mathbin{@} r, \Sigma_D \vdash (\{\overline{a_i}^n\}, \Gamma_D)} \; [CONS_{DstC}]$$

$$\frac{\Sigma_D(g) = \overline{m_i}^n \to \mathsf{s} \quad \Gamma_{D,0} = \bigoplus_{i=1}^n [a_i : m_i] \text{ defined} \quad \Gamma_D \supseteq \Gamma_{D,0} \quad \Gamma_D \text{ well formed}}{g\ \overline{a_i}^n \mathbin{@} \overline{r_j}^m, \Sigma_D \vdash (\{\overline{a_i}^n\}, \Gamma_D)} \; [APP_{DstC}]$$

$$\frac{e_1, \Sigma_D \vdash (L_1, \Gamma_{D,1}) \quad x_1 \notin L_1 \quad e_2, \Sigma_D \vdash (L_2, \Gamma_{D,2} + [x_1 : m]) \quad def(\Gamma_{D,1} \triangleright^{L_2} \Gamma_{D,2}) \quad m \in \{\mathsf{d}, \mathsf{s}\}}{\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2, \Sigma_D \vdash (L_1 \cup (L_2 \backslash \{x_1\}), \Gamma_{D,1} \triangleright^{L_2} \Gamma_{D,2})} \; [LET_{DstC}]$$

$$\frac{\begin{array}{c}\forall i \,.\, (e_i, \Sigma_D \vdash (L_i, \Gamma_{D,i}) \wedge \forall j . \Gamma_{D,i}(x_{ij}) \neq \mathsf{d}) \quad \Gamma_D \text{ well-formed} \\ \Gamma_D \supseteq \bigotimes_i (\Gamma_{D,i} \backslash \{\overline{x_{ij}}\}) \quad x \in \mathrm{dom}\, \Gamma_D \quad L = \{x\} \cup (\bigcup_i (L_i \backslash \{\overline{x_{ij}}\}))\end{array}}{\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \Sigma_D \vdash (L, \Gamma_D)} \; [CASE_{DstC}]$$

$$\frac{\begin{array}{c}\forall i \,.\, (e_i, \Sigma_D \vdash (L_i, \Gamma_{D,i}) \quad \forall j \,.\, \Gamma_{D,i}(x_{ij}) = \mathsf{d} \to j \in RecPos(C_i)) \quad \Gamma_D \text{ well formed} \\ L = \bigcup_i (L_i \backslash \{\overline{x_{ij}}\}) \quad \Gamma_D \supseteq (\bigotimes_i \Gamma_{D,i} \backslash (\{\overline{x_{ij}}\} \cup \{x\})) + [x : \mathsf{d}] \\ \forall z \in \mathrm{dom}\, \Gamma_D \,.\, \Gamma_D(z) \neq \mathsf{s} \to (\forall i \,.\, z \notin L_i)\end{array}}{\mathbf{case!}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}} \to e_i}, \Sigma_D \vdash (L \cup \{x\}, \Gamma_D)} \; [CASE!_{DstC}]$$

$$\frac{f\ \overline{x_i}^n \mathbin{@} \overline{r_j}^m = e_f \quad L = \{\overline{x_i}^n\} \quad \Gamma_D = [\overline{x_i \mapsto m_i}^n] \quad e_f, \Sigma_D \uplus [f \mapsto \overline{m_i}^n \to \mathsf{s}] \vdash (L, \Gamma_D)}{e_f, \Sigma_D \vdash (L, \Gamma_D)} \; [REC_{DstC}]$$

Figure 6.4: Proof rules for explicit deallocation.

we store the mark signatures of functions in a separate environment Σ_D. A judgement $e, \Sigma_D \vdash (L, \Gamma_D)$ means that L is the set of free variables in e, and that $\Sigma_D \cup \Gamma_D \vdash_{Dst} e : \mathsf{s}$. The proof rules of Figure 6.4 slightly deviate from the $\vdash_{Dst}$ rules, since the former are based on the type system of [84]. For instance, in the $[LET_{DstC}]$ rule, the bound variable is not allowed to have an in-danger mark. The $\triangleright$ operator ensures that no variables with unsafe types in Γ_1 occur free in L_2. This is expressed as an additional condition in the $\vdash_{Dst}$ rules. The condition Γ_D *well-formed* specifies that, if a variable x occurs in Γ_D with a condemned mark, all the variables in $sharerec(x, e)$ must also occur in Γ_D with an unsafe mark. The result of the $\otimes$ operator occurring in $[CASE_{DstC}]$ and $[CASE!_{DstC}]$ is defined if the given environments map common variables to the same type, and its result is the union of the bindings of the environments to which it is applied. The $[REC_{DstC}]$ rule plays a similar role as in the rules for region deallocation.

Once we have defined the proof rules, we have to specify how to prove them correct. This amounts to proving (6.7). Again, we do this by relating the static properties of the rules (mark environments) to the dynamic properties of the language (runtime closedness). Given a *Core-Safe* expression, its corresponding static assertion is a judgement of the form $e : [\![L, \Gamma_D]\!]$, where L contains the set of free variables in e. An expression e is said to satisfy this assertion if $fv(e) \subseteq L$ and some others conditions shown below hold. The generic part of the certificate generates a proof different from that described in Section 3.6.4. The definition shown below, which is based on [84], relies on the disjointness of the S and R sets corresponding to a given configuration (L, Γ_D, E, h). These sets are defined as follows:

$$\begin{array}{lcl} S_{L,\Gamma_D,E,h} & \stackrel{\text{def}}{=} & \bigcup_{x \in L, \Gamma_D(x) = \mathsf{s}} \{closure(h, E(x))\} \\ R_{L,\Gamma_D,E,h} & \stackrel{\text{def}}{=} & \bigcup_{x \in L, \Gamma_D(x) = \mathsf{d}} \{p \in live(h, E, L) \mid p \rightarrow_h^* recReach(h, E(x))\} \end{array}$$

In other words, the disjointness of S and R implies that there are no safe variables pointing to recursive children of condemned DSs (otherwise they would be in-danger). Now we give a formal meaning of an expression satisfying a given assertion.

Definition 6.8. Given the following properties:

$$\begin{array}{lcl} PD1 & \equiv & E \vdash h, k, e \Downarrow h', k, v \\ PD2 & \equiv & \text{dom } \Gamma_D \subseteq \text{dom } E \\ PD3 & \equiv & L \subseteq \text{dom } \Gamma_D \\ PD4 & \equiv & fv(e) \subseteq L \\ PD5 & \equiv & \forall x \in \text{dom } E.\, \forall z \in L\,. \\ & & \Gamma_D(z) = \mathsf{d} \wedge recReach(h, E(z)) \cap closure(h, E(x)) \neq \emptyset \rightarrow x \in \text{dom } \Gamma_D \wedge \Gamma_D(x) \neq \mathsf{s} \\ PD6 & \equiv & \forall x \in \text{dom } E\,.\, closure(h, E(x)) \neq closure(h', E(x)) \rightarrow x \in \text{dom } \Gamma_D \wedge \Gamma_D(x) \neq \mathsf{s} \\ PD7 & \equiv & S_{L,\Gamma_D,E,h} \cap R_{L,\Gamma_D,E,h} = \emptyset \\ PD8 & \equiv & closed(h, E, L) \\ PD9 & \equiv & closed(h', v) \end{array}$$

we say that the expression e satisfies the static assertion $[\![L, \Gamma_D]\!]$, denoted $e : [\![L, \Gamma_D]\!]$, iff

$$PD3 \wedge PD4 \wedge (\forall E, h, k, h', v.PD1 \wedge PD2 \rightarrow (PD5 \wedge PD6 \wedge (PD7 \wedge PD8 \rightarrow PD9)))$$

In a similar way as before, we also define a notion of satisfaction relative to the validity of a mark environment Σ_D, denoted $e, \Sigma_D : [\![L, \Gamma_D]\!]$ (see [37] for details).

Property *PD1* defines the runtime evaluation of e. Properties *PD2* to *PD4* just guarantee that each free variable has a type and a value. Properties *PD5* to *PD7* formalise the dynamic meaning of safe and

condemned types: if a variable can share a recursive descendant of a condemned one, or its closure changes during evaluation, it has to occur in the environment with an unsafe type. The key properties are *PD8* and *PD9*. If we prove them to hold for every judgement in the derivation of *PD1*, we guarantee that the live part of the heap remains closed during the execution of e, and hence that there are not dangling pointers at runtime. The following theorem shows that *PD8* is an invariant which is propagated upwards through the evaluation tree, whereas *PD9* is propagated downwards.

Theorem 6.9 (closedness)**.** *Let us assume a $\Downarrow$-judgement:*

$$E \vdash h, k, e \Downarrow h', k, v \tag{6.9}$$

For every L and Γ_D such that $e : [\![L, \Gamma_D]\!]$: if $PD2(\Gamma_D, E)$, $PD7(L, \Gamma_D, E, h)$ and $PD8(E, L, h)$ hold, then for every judgement $E_i \vdash h_i, k_i, e_i \Downarrow h'_i, k_i, v_i$ occurring in the derivation of (6.9) it holds that $PD8(E_i, L_i, h_i)$ and $PD9(v_i, h'_i)$.

Proof. (see [37, 35]). □

Since *PD2*, *PD7* and *PD8* hold trivially for the empty heap, the empty mark environment, and the empty set of free variables, which are the ones corresponding to the initial expression, *PD8* and *PD9* are guaranteed to hold across the whole derivation of the program.

Similarly as in the previous section, the last step is to prove the soundness relation between the proof rules and these static assertions.

Theorem 6.10 (soundness)**.** *If $e, \Sigma_D \vdash (L, \Gamma_D)$, then $e, \Sigma_D : [\![L, \Gamma_D]\!]$.*

Proof. It follows from the proof described in [84] (see [35] for more details). □

6.3 Certificate generation

Once we have established the correctness of the proof rules of Figures 6.3 and 6.4, this section explains how to generate, an Isabelle/HOL script establishing that these proof rules can be applied to the program being certified. We assume that the generic part of the certificate (given by the theorems in the last section) has been proved in advance, and that its corresponding theorems are available in a database of proven facts. This part of the certificate has been proved interactively with Isabelle/HOL [35]. In this section we focus on the program-specific part. Assuming we are given a *Core-Safe* program with the type information inferred in Chapters 4 and 5, the certifier generates a plain text file containing an Isabelle/HOL script. This script contains, for every function definition of the program, the following information:

- An Isabelle/HOL definition of the abstract syntax tree (AST) of the function's body.
- A set of Isabelle/HOL definitions of the static objects inferred by the compiler: sets of free variables, mark environments, region typing environments, type instantiation mappings, etc.
- A set of Isabelle/HOL proof scripts proving a lemma for each expression. Each proof consists in first checking the premises of the proof rule associated to the syntactic form of the expression, and then applying this proof rule.

$$
\begin{array}{lcll}
\textbf{LExp} \ni & \varepsilon & \to & a & \{\text{atom: literal } c \text{ or variable } x\} \\
& & | & x \mathbin{@} r & \{\text{copy}\} \\
& & | & a_1 \oplus a_2 & \{\text{basic operator application}\} \\
& & | & C\ \overline{a_i} \mathbin{@} r & \{\text{constructor application}\} \\
& & | & f\ \overline{a_i} \mathbin{@} \overline{r_j} & \{\text{function application}\} \\
& & | & \textbf{let}\ x_1 = m_1\ \textbf{in}\ m_2 & \{\textbf{let} \text{ declaration: } m_1, m_2 \in \mathbb{N}\} \\
& & | & \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to m_i} & \{\text{read-only pattern matching: } m_i \in \mathbb{N}\} \\
& & | & \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to m_i} & \{\text{destructive pattern matching: } m_i \in \mathbb{N}\}
\end{array}
$$

Figure 6.5: Syntax of labelled *Core-Safe* expressions.

With respect to the latter part, the certificate proves the lemmas of compound expressions in a bottom-up way. For example, when proving that the judgement $\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2, \Sigma_R \vdash \Gamma_R \rightsquigarrow s$ holds, the certificate must have proved the validity of the judgements corresponding to the sub-expressions e_1 and e_2. For this reason, there must be a way to uniquely identify every sub-expression in a given function definition, so each sub-expression can be referred to when proving the lemma of the parent expression. Hence, our first goal is to split the body of the function being certified into a list of its sub-expressions. Each of these is identified by a natural number. When dealing with compound expressions (i.e. **let**, **case** and **case**!), we replace each sub-expression by the number identifying it. As a consequence, we have to slightly adapt the syntax of our *Core-Safe* expressions in order to support these identifiers. The set of *labelled Core-Safe expressions* is defined by the grammar of Figure 6.5. Notice the absence of nested expressions: these have been replaced by their identifier.

Example 6.11. Consider the following function definition:

$$
\begin{array}{l}
\mathit{unshuffle} :: [\alpha]! \to ([\alpha], [\alpha]) \\
\mathit{unshuffle}\ [\,]! \quad\ = ([\,], [\,]) \\
\mathit{unshuffle}\ (x : xs)! = (x : ys2, ys1)\ \textbf{where}\ (ys1, ys2) = \mathit{unshuffle}\ xs
\end{array}
$$

Its translation into *Core-Safe* yields the following result:

$$
\begin{array}{l}
\mathit{unshuffle}\ ::\ [\alpha]!@\rho \to \rho_1 \to \rho_2 \to \rho_3 \to ([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3 \\
\mathit{unshuffle}\ xs\ @\ r_1\ r_2\ r_3 = \\
\qquad \textbf{case!}\ xs\ \textbf{of} \\
\qquad\quad [\,] \qquad \to \quad \textbf{let}\ x_1 = [\,]\ @\ r_1\ \textbf{in} \\
\qquad\qquad\qquad\quad\ \ \textbf{let}\ x_2 = [\,]\ @\ r_2\ \textbf{in}\ (x_1, x_2)@r_3 \\
\qquad\quad (x : xx) \quad \to \quad \textbf{let}\ x_4 = \mathit{unshuffle}\ xx\ @\ r_2\ r_1\ \mathit{self}\ \textbf{in} \\
\qquad\qquad\qquad\quad\ \ \textbf{let}\ x_5 = \textbf{case}\ x_4\ \textbf{of}\ (a, b) \to a\ \textbf{in} \\
\qquad\qquad\qquad\quad\ \ \textbf{let}\ x_6 = \textbf{case}\ x_4\ \textbf{of}\ (c, d) \to d\ \textbf{in} \\
\qquad\qquad\qquad\quad\ \ \textbf{let}\ x_7 = (x : x_6)@r_1\ \textbf{in}\ (x_7, x_5)@r_3
\end{array}
$$

If we split the body of this function into its sub-expressions, we get the list of labelled expressions shown in Figure 6.6. The elements of the list are arranged bottom-up (i.e. from simple to compound expressions), because this is the order required by Isabelle/HOL for applying the proof rules. □

Given the above, the first phase consists in translating the function being certified into a set of tuples containing every sub-expression in its labelled form, together with its related static information. Each

Identifier (m)	**Expression** (ε_m)
1	*unshuffle* xx @ r_2 r_1 *self*
2	a
3	**case** x_4 **of** $(a, b) \rightarrow 2$
4	d
5	**case** x_4 **of** $(c, d) \rightarrow 4$
6	(x_7, x_5) @ r_3
7	$(x : x_6)$ @ r_1
8	**let** $x_7 = 7$ **in** 6
9	**let** $x_6 = 5$ **in** 8
10	**let** $x_5 = 3$ **in** 9
11	**let** $x_4 = 1$ **in** 10
12	(x_1, x_2) @ r_3
13	$[\,]$ @ r_2
14	$[\,]$ @ r_1
15	**let** $x_2 = 13$ **in** 12
16	**let** $x_1 = 14$ **in** 15
17	**case**! xs **of** $\{[\,] \rightarrow 16; (x : xx) \rightarrow 11\}$

Figure 6.6: Labelled expressions resulting from splitting the body of *unshuffle*. The components of **let** and **case** expressions have been replaced by their identifiers.

tuple ϕ_m has the following components:

$$\phi_m \equiv (m,\ \varepsilon_m,\ L_m,\ \Gamma_{D,m},\ \Gamma_{R,m},\ s_m,\ \mu_m) \tag{6.10}$$

where m is the tuple identifier, ε_m is the corresponding sub-expression, L_m is the set of its free variables, $\Gamma_{D,m}$ and $\Gamma_{R,m}$ denote the mark and region environments typing the sub-expression, s_m is its safe type, and μ_m is the type instantiation mapping needed when ε_m is a function or a constructor application.

Example 6.12. Given the *unshuffle* function of Example 6.11 above, in Figure 6.7 we show the components of each tuple. Function and constructor applications also have the extra component μ_m which is the instantiation mappings used in each case, and assuming the following generic type schemes for the list and tuple constructors.

$$\begin{aligned} (:) &:: \alpha \rightarrow [\alpha]@\rho \rightarrow \rho \rightarrow [\alpha]@\rho \\ [\,] &:: \rho \rightarrow [\alpha]@\rho \\ (\cdot,\cdot) &:: \alpha_1 \rightarrow \alpha_2 \rightarrow \rho \rightarrow (\alpha_1, \alpha_2)@\rho \end{aligned}$$

□

There is also a tuple for every function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$. Each one has the following components:

$$\Phi_f \equiv (f,\ \overline{x_i}^n,\ \overline{r_j}^m,\ \overline{m_i}^n,\ \Sigma_{D,f},\ \Sigma_{R,f},\ e_f,\ \overline{\phi_i}^q) \tag{6.11}$$

The first three components are self-explanatory, as well as e_f. The list $\overline{m_i}^n$ denotes the marks inferred for the function parameters. The $\Sigma_{D,f}$ environment contains the mark signatures of the functions that have been certified before f, together with the mark signature of the latter. Thus the signature of every function being called in e_f is assumed to belong to this environment. The $\Sigma_{R,f}$ component has a similar

m	L_m	$\Gamma_{D,m}$	$\Gamma_{R,m}$	s_m
1	$\{xx\}$	$[xx : \mathsf{d}, xs : \mathsf{r}]$	$\Gamma_{R,11}$	$([\alpha]@\rho_2, [\alpha]@\rho_1)@\rho_3$
2	$\{a\}$	$[a : \mathsf{s}]$	$\Gamma_{R,10} + [a : [\alpha]@\rho_2, b : [\alpha]@\rho_1]$	$[\alpha]@\rho_2$
3	$\{x_4\}$	$[x_4 : \mathsf{s}]$	$\Gamma_{R,10}$	$[\alpha]@\rho_2$
4	$\{d\}$	$[d : \mathsf{s}]$	$\Gamma_{R,9} + [c : [\alpha]@\rho_2, d : [\alpha]@\rho_1]$	$[\alpha]@\rho_1$
5	$\{x_4\}$	$[x_4 : \mathsf{s}]$	$\Gamma_{R,9}$	$[\alpha]@\rho_1$
6	$\{x_7, x_5\}$	$[x_7 : \mathsf{s}, x_5 : \mathsf{s}]$	$\Gamma_{R,8} + [x_7 : [\alpha]@\rho_1]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
7	$\{x, x_6\}$	$[x : \mathsf{s}, x_6 : \mathsf{s}]$	$\Gamma_{R,8}$	$[\alpha]@\rho_1$
8	$\{x, x_6, x_5\}$	$[x : \mathsf{s}, x_6 : \mathsf{s}, x_5 : \mathsf{s}]$	$\Gamma_{R,9} + [x_6 : [\alpha]@\rho_1]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
9	$\{x_4, x, x_5\}$	$[x : \mathsf{s}, x_4 : \mathsf{s}, x_5 : \mathsf{s}]$	$\Gamma_{R,10} + [x_5 : [\alpha]@\rho_2]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
10	$\{x_4, x\}$	$[x_4 : \mathsf{s}, x : \mathsf{s}]$	$\Gamma_{R,11} + [x_4 : ([\alpha]@\rho_2, [\alpha]@\rho_1)@\rho_{self}]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
11	$\{xx, x\}$	$[xx : \mathsf{d}, xs : \mathsf{r}, x : \mathsf{s}]$	$\Gamma_{R,17} + [x : \alpha, xx : [\alpha]@\rho]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
12	$\{x_1, x_2\}$	$[x_1 : \mathsf{s}, x_2 : \mathsf{s}]$	$\Gamma_{R,15} + [x_2 : [\alpha]@\rho_2]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
13	$\emptyset$	$[\,]$	$\Gamma_{R,15}$	$[\alpha]@\rho_2$
14	$\emptyset$	$[\,]$	$\Gamma_{R,17}$	$[\alpha]@\rho_1$
15	$\{x_1\}$	$[x_1 : \mathsf{s}]$	$\Gamma_{R,17} + [x_1 : [\alpha]@\rho_1]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
16	$\emptyset$	$[\,]$	$\Gamma_{R,17}$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$
17	$\{xs\}$	$[xs : \mathsf{d}]$	$[xs : [\alpha]@\rho, r_1 : \rho_1, r_2 : \rho_2, r_3 : \rho_3, self : \rho_{self}]$	$([\alpha]@\rho_1, [\alpha]@\rho_2)@\rho_3$

Figure 6.7: Free variables and typing information attached to each labelled expression.

m	μ_m
1	$[\rho_1 \mapsto \rho_2, \rho_2 \mapsto \rho_1, \rho_3 \mapsto \rho_{self}]$
6	$[\alpha_1 \mapsto [\alpha]@\rho_1, \alpha_2 \mapsto [\alpha]@\rho_2, \rho \mapsto \rho_3]$
7	$[\alpha \mapsto \alpha, \rho \mapsto \rho_1]$
12	$[\alpha_1 \mapsto [\alpha]@\rho_1, \alpha_2 \mapsto [\alpha]@\rho_2, \rho \mapsto \rho_3]$
13	$[\alpha \mapsto \alpha, \rho \mapsto \rho_2]$
14	$[\alpha \mapsto \alpha, \rho \mapsto \rho_1]$

Figure 6.8: Type instantiation mappings of the function and constructor applications occurring in *unshuffle*.

meaning, but it contains safe type schemes, instead of mark signatures. Lastly, the $\overline{\phi_i}^q$ component contain the list of tuples into which the body e_f is split.

There also exists a tuple for the whole program. It has the form $(\Gamma_C, \overline{\Phi_{f_i}}, \overline{\phi_j})$ and contains the type schemes of the data constructors being defined in the program, the list of tuples corresponding to each function definition, and the list of tuples into which the main expression of the program is split.

The task of generating all these tuples from the input *Core-Safe* program is straightforward, and it will not be described here. These tuples provide a sequential representation of the expressions of a program which is more amenable to the final step of the certification process: the generation of the Isabelle/HOL script.

Even when we have separated the generic proofs of the certificate from the program-specific ones, a considerable part of the program-specific part is still boilerplate code which can be reused in every certificate with little alteration. In order to ease the programming of the certifier we have used a *template system* to deal with this code. A template is a fragment of Isabelle/HOL code, which is parametric on some parts. These parameters are given as *placeholders*. Templates have the following syntax:

```
%.template [TemplateName]

These are the contents of the template with ${PlaceHolders}

%.end
```

A placeholder is denoted with the syntax `${P}`, being `P` its name. Templates are stored in a plain text file which is read and processed by the compiler. The certifying phase of the compiler substitutes the contents of the tuples ϕ_i shown above for the placeholders in the template. Usually the compiler has to apply a given template several times and join the results. We shall explain this with an example:

Example 6.13. The generation of a mark environment is given by the following templates:

```
%.template EnvBindingsSet

{ ${Bindings} }

%.end
```

```
%.template EnvBinding

"${Variable}" \<mapsto> ${Type}

%.end
```

```
%.template Mark

${Mark}"

%.end
```

Given the environment $\Gamma_D = [x : \mathsf{d}, y : \mathsf{s}, z : \mathsf{r}]$, the compiler firstly applies the EnvBinding template to the binding $x : \mathsf{d}$ as follows: it instantiates the Mark template with the d mark, to get the string d", and then puts this string into the `${Type}` placeholder in EnvBinding. It also substitutes `${Variable}` for x, so as to get `"x" \<mapsto> d"`. If we do the same with the bindings $y : \mathsf{s}$ and $z : \mathsf{r}$, we obtain two additional strings:

```
"y" \<mapsto> s"        "z" \<mapsto> r"
```

Finally, the three strings are joined with a (,) between them. The result is substituted for the `${Bindings}` placeholder in EnvBindingsSet and we obtain:

```
{"x" \<mapsto> d",   "y" \<mapsto> s",  "z" \<mapsto> r"}
```

□

The overall structure of the certificate is given by the template shown in Figure 6.9. The `${ModuleName}` placeholder contains the name of the program being certified. The Γ_C environment containing the signatures of the constructors is stored in `${CSigs}`. The region and mark environments corresponding to every function definition (components $\Sigma_{R,f}$ and $\Sigma_{D,f}$ of the tuple (6.11)) are placed in `${SigmaDefs}`. These environments are generated by the following template:

```
%.template SigmaEntry

constdefs \<Sigma>_${FuncName}_reg ::
  "string \<rightharpoonup>
     TypeExpression list \<times> VarType list \<times> TypeExpression"
  "\<Sigma>_${FuncName}_reg \<equiv> [${SigmaRegEnv}]"

constdefs \<Sigma>_${FuncName}_mark ::
  "string \<rightharpoonup> Mark list"
  "\<Sigma>_${FuncName}_mark \<equiv> [${SigmaMarkEnv}]"

%.end
```

This template is filled in for every function definition appearing in the program. Region signature environments are defined as partial mappings (`\<rightharpoonup>`) from function names (`string`) to triples containing the type of the parameters (`TypeExpression list`), the types of the region parameters (`VarType list`) and the type of the result (`TypeExpression`). The environments themselves are generated as in Example 6.13 and placed in `${SigmaRegEnv}` and `${SigmaMarkEnv}` respectively.

Example 6.14. When applying these templates to our *unshuffle* function (Example 6.11) we get the following Isabelle/HOL definitions:

```
constdefs Σ_unshuffle_reg :: "string ⇀
        TypeExpression list × VarType list × TypeExpression"
 "Σ_unshuffle_reg ≡ [
```

```
%.template Main

(*  Title:      Pointer safety certificate for ${Filename}
    Generated:  ${Timestamp}
    Copyright:  2010 Universidad Complutense de Madrid
*)
header {* Safety certificate for ${Filename} *}
theory ${ModuleName} imports THRegionsDatabase
begin

(***************  CONSTRUCTOR TYPE SIGNATURES  ******************)

defs constructorSignature_def : "constructorSignature \<equiv> ${CSigs}"

(*************** SIGMA ENVIRONMENT DEFINITIONS ******************)

${SigmaDefs}

(*************** REGION ENVIRONMENT DEFINITIONS *****************)

${RegEnvDefs}

(********************** L SET DEFINITIONS  ***********************)

${LDefs}

(***************** MARK ENVIRONMENT DEFINITIONS ******************)

${MarkEnvDefs}

(********************** TYPE DEFINITIONS *************************)

${TDefs}

(************ MU INSTANTIATION MAPPING DEFINITIONS ***************)

${MuDefs}

(******************** PROGRAM EXPRESSIONS ************************)

${ExpDefs}

(******************** REGION SAFETY PROOFS ***********************)

${RegProofs}

(************ EXPLICIT DESTRUCTION SAFETY PROOFS *****************)

${DstProofs}

end

%.end
```

Figure 6.9: Global structure of the program-specific part of the certificate

```
  ''_MOD_'' ↦ ([ConstrT intType [] [], ConstrT intType [] []], [],
               ConstrT intType [] []),
  ''_PLUS_'' ↦ ([ConstrT intType [] [], ConstrT intType [] []], [],
               ConstrT intType [] []),
    ...
  ''unshuffle'' ↦ ([ConstrT ''ListT'' [VarT ''a''] [''rho'']],
               [''rho2'', ''rho3'', ''rho4''],
                ConstrT ''TupleTT_2'' [ConstrT ''ListT'' [VarT ''a''] [''rho2''],
                                       ConstrT ''ListT'' [VarT ''a''] [''rho3'']]
                        [''rho4''])
]"

constdefs Σ_unshuffle_mark :: "string ⇀ Mark list"
 "Σ_unshuffle_mark ≡ [
  ''_MOD_'' ↦ [s'', s''],
  ''_PLUS_'' ↦ [s'', s''],
    ...
  ''unshuffle'' ↦ [d'']
 ]"
```

These initial environments contain the types of built-in operators and some basic functions. It also contains a representation of the type of *unshuffle*. □

The remaining definitions in the certificate are the components of the tuple (6.10). To avoid name clashes in the certificate, the name of each Isabelle/HOL definition contains the name of the function to which it belongs, and the identifier of the tuple. As an example, we show the template for generating the L sets of free variables of each sub-expression.

```
%.template LDef

constdefs L_${FuncName}_${ExpNum} :: "string set"
          "L_${FuncName}_${ExpNum} \<equiv>  {${Vars}}"

%.end
```

An instance of this template is generated for each expression in the program. The `${FuncName}` placeholder contains the name of the context function and `${ExpNum}` contains the identifier of the corresponding tuple.

Example 6.15. Back to our *unshuffle* function, the certifier yields[1] the following definitions for the labelled expression ε_8:

```
constdefs Γ_D_unshuffle_8 :: "TypeEnvironment"
 "Γ_D_unshuffle_8_reg ≡  [''x'' ↦ s'', ''x6'' ↦ s'', ''x5'' ↦ s'']"

constdefs L_unshuffle_8 :: "string set"
 "L_unshuffle_8 ≡  {''x'', ''x6'', ''x5''}"

constdefs Γ_R_var_unshuffle_8 :: "TypeMapping"
 "Γ_R_var_unshuffle_8 ≡ [
```

[1] Variables have been renamed to match those of Figure 6.7, for the sake of clarity.

```
    ''x6'' ↦ ConstrT ''ListT'' [VarT ''a''] [''rho1''],
    ''x5'' ↦ ConstrT ''ListT'' [VarT ''a''] [''rho2''],
    ''x4'' ↦ ConstrT ''TupleTT_2'' [
                ConstrT ''ListT'' [VarT ''a''] [''rho2''],
                ConstrT ''ListT'' [VarT ''a''] [''rho1'']] [''rho_self''],
    ''xx'' ↦ ConstrT ''ListT'' [VarT ''a''] [''rho''],
    ''x'' ↦ VarT ''a'',
    ''xs'' ↦ ConstrT ''ListT'' [VarT ''a''] [''rho'']] "

constdefs Γ_R_reg_unshuffle_8 :: "RegMapping"
 "Γ_R_reg_unshuffle_8 ≡ [''r1'' ↦ ''rho1'' , ''r2'' ↦ ''rho2'' , ''r3'' ↦ ''rho3'',
                         ''self'' ↦ ''rho_self'' ]"

constdefs Type_unshuffle_8 :: "TypeExpression"
 "Type_unshuffle_8 ≡  ConstrT ''TupleTT_2''
                       [ConstrT ''ListT'' [VarT ''a''] [''rho1''],
                        ConstrT ''ListT'' [VarT ''a''] [''rho2'']] [''rho3'']"
```

The certificate splits the environment Γ_R into two: one containing the types of variables (Γ_R_var) and another one containing the types of region variables (Γ_R_reg). □

The last two placeholders of the certificate (${RegProofs} and ${DstProofs}) contain, for every expression e, the proof of the respective judgements $e, \Sigma_D \vdash (L, \Gamma_D)$ and $e, \Sigma_R \vdash \Gamma_R \rightsquigarrow s$, where L is the set of free variables in e, s is its type, and Γ_D and Γ_R are their corresponding environments. Each proof starts with the statement of the lemma to be proved.

```
%.template DstProof

lemma e_${FuncName}_${ExpNum}_DST:
   "def_e_${FuncName}_${ExpNum} : \<lbrace> L_${FuncName}_${ExpNum},
            \<Gamma>_D_${FuncName}_${ExpNum} \<rbrace>"

${ProofBody}

%.end
```

```
%.template RegProof

lemma e_${FuncName}_${ExpNum}_REG:
    "def_e_${FuncName}_${ExpNum} \<turnstile>
  (\<Gamma>_R_var_${FuncName}_${ExpNum},
      \<Gamma>_R_reg_${FuncName}_${ExpNum})
        \<leadsto> Type_${FuncName}_${ExpNum}"

${ProofBody}

%.end
```

The ${ProofBody} placeholder contains the script proving the corresponding lemma. This script depends on the form of the expression to which the lemma refers. Regarding judgements of the form

```
%.template ProofLet1S

apply (unfold def_e_${FuncName}_${ExpNum}_def)
apply (rule_tac ?L1.0="L_${FuncName}_${ExpNum1}" and
          ?\<Gamma>1.0="\<Gamma>_${FuncName}_${ExpNum1}" and
          ?L2.0="L_${FuncName}_${ExpNum2}" and
          ?\<Gamma>2'.0="\<Gamma>_${FuncName}_${ExpNum2}"
              in SafeDA_LET1_s)
apply (rule e_${FuncName}_${ExpNum1})
apply (rule e_${FuncName}_${ExpNum2})
apply (unfold \<Gamma>_${FuncName}_${ExpNum2}_def)
apply (simp add: disjointUnionEnv_def)?
apply (simp add: unionEnv_def)?
apply (rule ext, simp, force)?
apply (simp add: def_disjointUnionEnv_def)
apply (unfold \<Gamma>_${FuncName}_${ExpNum1}_def,
       unfold L_${FuncName}_${ExpNum2}_def,
       simp add: def_pp_def)
apply (unfold unsafe_def, force)?
apply (unfold L_${FuncName}_${ExpNum1}_def, force)
apply (unfold L_${FuncName}_${ExpNum}_def, force)
apply (unfold \<Gamma>_${FuncName}_${ExpNum}_def)
apply (simp add: pp_def)
apply (simp add: disjointUnionEnv_def, simp add: unionEnv_def)?
apply (rule ext,simp)
by (unfold def_e_${${FuncName}}_${ExpNum1}_def, simp)

%.end
```

Figure 6.10: Proof script template for **let** expressions (explicit deallocation).

$e, \Sigma_D : [\![L, \Gamma_D]\!]$, the proof script is generated as a sequence of Isabelle/HOL tactics and rules. For instance, Figure 6.10 shows the script corresponding **let** expressions in which the bound variable gets an s mark in the main expression. The second line in this script consists in applying the $[LET_{DstC}]$ rule. This rule generates several proof obligations: one for each of its premises. The subsequent lines of the script are aimed at proving these proof obligations.

With respect to the judgements of the form $e, \Sigma_R \vdash \Gamma_R \rightsquigarrow s$, de Dios has defined in [35] an Isabelle/HOL tactic for each of the rules in Figure 6.3. So, the proof of each of the lemmas generated by the RegProof template shown above consists in applying the corresponding tactic. Each tactic applies a type rule and discharges its associated proof obligations in a single proof step. This reduces both the size and checking time of the certificate. For instance, Figure 6.11 shows the template which applies the $[LET_{RegC}]$ rule and discharges all its associated proof obligations. It is part of ongoing work to define tactics for the proofs related to explicit destruction.

Example 6.16. By applying these templates to the expression ε_8 above, we get the following proof:

```
lemma e_unshuffle_8_REG:
  "def_e_unshuffle_8 ⊢
```

```
%.template RegProofLet

by (tactic  "SafeDARegion_LET1_tac
   (@{thm def_e_${FuncName}_${ExpNum}_def},
    @{thm \<Gamma>_R_var_${FuncName}_${ExpNum}_def},
    @{thm \<Gamma>_R_reg_${FuncName}_${ExpNum}_def},
    @{thm Type_${FuncName}_${ExpNum}_def},
    @{thm def_e_${FuncName}_${ExpNum1}_def},
    @{thm \<Gamma>_R_var_${FuncName}_${ExpNum1}_def},
    @{thm \<Gamma>_R_reg_${FuncName}_${ExpNum1}_def},
    @{thm Type_${FuncName}_${ExpNum1}_def},
    @{thm e_${FuncName}_${ExpNum1}_REG},
    @{thm def_e_${FuncName}_${ExpNum2}_def},
    @{thm \<Gamma>_R_var_${FuncName}_${ExpNum2}_def},
    @{thm \<Gamma>_R_reg_${FuncName}_${ExpNum2}_def},
    @{thm Type_${FuncName}_${ExpNum2}_def},
    @{thm e_${FuncName}_${ExpNum2}_REG},
    "\<Gamma>_R_var_${FuncName}_${ExpNum1}",
    "\<Gamma>_R_reg_${FuncName}_${ExpNum1}",
    "Type_${FuncName}_${ExpNum1}",
    "\<Gamma>_R_var_${FuncName}_${ExpNum2}",
    "\<Gamma>_R_reg_${FuncName}_${ExpNum2}",
    "Type_${FuncName}_${ExpNum2}")")

%.end
```

Figure 6.11: Proof script template for **let** expressions (region deallocation). There is a single step consisting in applying an user-defined tactic.

Name	*Core-Safe* code	Size of the certificate (lines)				Ratio
	Size (lines)	Definitions	Proofs (regs.)	Proofs (dest.)	Total	
`basic`	306	2667	1634	2218	6619	21.63
`unshuffle`	49	429	271	407	1107	22.59
`mergesort`	189	1748	1134	1753	4635	24.52
`inssort`	67	651	406	616	1673	24.97
`pascal`	106	1105	658	911	2674	25.22
`compiler`	2058	16131	10262	13471	39864	19.37
`AVLTrees`	502	4680	3008	4484	12472	24.84

Figure 6.12: Comparison of sizes between the certificates and the source *Core-Safe* code.

```
    (Γ_R_var_unshuffle_8, Γ_R_reg_unshuffle_8) ⇝  Type_unshuffle_8"
by (tactic "SafeDARegion_LET1_tac
         (@{thm def_e_unshuffle_8_def},
          @{thm Γ_R_var_unshuffle_8_def},
          @{thm Γ_R_reg_unshuffle_8_def},
          @{thm Type_unshuffle_8_def},
          @{thm def_e_unshuffle_7_def},
          @{thm Γ_R_var_unshuffle_7_def},
          @{thm Γ_R_reg_unshuffle_7_def},
          @{thm Type_unshuffle_7_def},
          @{thm e_unshuffle_7_REG},
          @{thm def_e_unshuffle_6_def},
          @{thm Γ_R_var_unshuffle_6_def},
          @{thm Γ_R_reg_unshuffle_6_def},
          @{thm Type_unshuffle_6_def},
          @{thm e_unshuffle_6_REG},
          "Γ_R_var_unshuffle_7",
          "Γ_R_reg_unshuffle_7",
          "Type_unshuffle_7",
          "Γ_R_var_unshuffle_7",
          "Γ_R_reg_unshuffle_7",
          "Type_unshuffle_7"
          )
```

□

6.4 Case studies

In this section we aim to measure the sizes of the resulting certificates in comparison with the size of their respective source code. The table in Figure 6.12 shows the results. For each certificate, we have measured the number of lines generated for the definitions involved in the proofs (such as typing environments, mark environments, instantiation mappings, sets of free variables, etc.) and the number of lines of the proofs themselves. The column *Proofs (regs.)* refers to the part of the certificate related to region deallocation, whereas *Proofs (dest.)* measures the part regarding explicit destruction. The `basic` file contains several example functions on lists and binary search trees (*append*, *appendC*, *insertT*, etc.), most of which were defined in Chapter 2. The `pascal` example was shown in Example 4.20, whereas `compiler` corresponds to the small compiler developed at the end of Chapter 4 (Example 4.22). Lastly, `AVLTrees` contains the code of the functions of a small library for handling AVL Trees, which was shown in Example 3.14.

From these results we can draw the following conclusions:

1. The size of the certificate grows linearly with the size of the input *Core-Safe* program. The expansion factor (last column in the table) ranges between 19 and 25.

2. A considerable part of the certificate (around 40%) is devoted to the definitions of the elements in the tuples shown in (6.10) and (6.11). The proofs of region consistency preservation make up approximately 25% of the certificate, whereas the remaining 35% deals with the proofs related to explicit deallocation. We expect the latter to decrease when we replace the actual proof scripts by the application of used-defined tactics, as it was done with the part related to region consistency.

6.5 Conclusions and related work

In this chapter we have shown how to encode, in Isabelle/HOL, the proof that a *Core-Safe* program is well-typed, so that it can be mechanically checked for that program. The hardest part of the proof (correctness of the type system) is assumed to have been done in advance, and once for all. This results in smallest certificates, and shorter checking times. In spite of that, certificates are approximately twenty times bigger than the program being certified. Another approach would have been to implement a certified type checker, and that the code consumer applies this checker to the input *Core-Safe* program. However, this delegates the type-checking task to the code consumer. It is necessary to find a balance between the size of the certificates, and the amount of work to be done by the code consumer. With regard to this balance, it is worth taking into account that the certification of a program is associated with its deployment in the target machine, but neither with the development of that program, nor with its execution on the target machine. Therefore, deployment is not a so frequent task in software development, as to consider the certificate checking time to be decisive.

Introducing pointers in a Hoare-style assertion logic and using a proof assistant for proving the correctness of programs with pointers goes back to the late seventies [74], where the Stanford Pascal Program Verifier was used. A more recent reference is [20], using the Jape proof editor. A formalisation of Bornat's ideas in Isabelle/HOL was done by Mehta and Nipkow in [78], where they add a complete soundness proof. None of these papers address the problem of automatically generating a certificate.

Connecting the results of a static analysis with the generation of certificates was done from the beginning of the PCC paradigm (see for instance [90]). A more recent work is [15].

Our work is more closely related to [18], where a resource consumption property obtained by Hofmann and Jost's type system [58] is transformed into a certificate. The compiler is able to infer a linear upper bound on heap consumption and to certify this property by emitting an Isabelle/HOL script proving it. Our static assertions have been inspired by their *derived assertions*, used also there to connect static with dynamic properties. However, their heap is simpler to deal with than ours since it essentially consists of a free list of cells, and the only data type available is the list. We must also deal with regions and with any user-defined data type. This results in our complex notion of consistency. The authors of [18] conjecture a set of proof rules and claim they could be used to prove the safety of destruction in Hofmann and Aspinalls's type system [11]. But in fact they do not provide an Isabelle/HOL or a manual proof of these rules. The lessons that can be drawn from [37, 35] about proving the correctness of proof rules of Section 6.2 in a proof assistant is that it is a rather daunting task full of unexpected difficulties which have forced us to frequently modify the proof rules.

Chapter 7

Memory consumption analysis

7.1 Introduction

In this chapter we present a static analysis aimed at inferring upper bounds to the heap and stack consumption made by a *Safe* program. Since the memory needs of a program usually depends on its input, the bounds we obtain in our analysis are functions on the sizes of the inputs. The bounds given by this memory consumption analysis are considered correct if they are equal or greater than the actual worst-case consumptions of the program being analysed.

The inference of the space complexity of a program is a very complex task, even for a first-order language like *Safe*. There are three separate aspects that can be independently studied and solved:

1. Inference of upper bounds to the size of the call-tree deployed at runtime by each recursive *Safe* function.

2. Inference of the sizes of the data structures involved in a *Safe* function.

3. From the results of (1) and (2), inference of upper bounds to the stack and heap space costs of a *Safe* function.

In this chapter we deal exclusively with the latter point. The last two points are subject of ongoing work, and not part of this thesis. The first one is closely related to the termination of a *Safe* program. Although termination is an undecidable program property (as well as the properties given in (2) and (3)), several approaches have been proposed in the literature for approximating it. In particular, we can infer multivariate polynomials of any degree as solutions by using the approach described in [73]. Another possibility is the translation of a *Safe* function into a recurrence relation modelling the size of the call-tree, and use already existing tools (PURRS [13], PUBS [4]) for inferring closed-form expressions to the result of these recurrence relations. This allows us to infer expressions beyond polynomial bounds (in particular, logarithmic and exponential). With regard to (2), in [105] the authors describe how to obtain polynomial, non-monotonic size relations between the input and the output of a function. This approach can also be combined with the PUBS recurrence solver in order to tackle (1), as described in [87]. Another possibility is the approach described in [97], which infers linear size relations between the input and the output of a *Safe* function.

For the purposes of this chapter, we shall assume that the results of (1) and (2) are given by the programmer, so we can concentrate on (3) instead. In principle, the analysis described here can accommodate several complexity classes, as long as their functions are monotonic w.r.t. the input sizes.

Obviously, the complexity class of the result critically depends of the complexity class of the inputs given to the analysis (points (1) and (2) above). In principle, we infer bounds to the costs of *Core-Safe* function definitions without taking destructive pattern matching into account (**case**! expressions are handled as if they were non-destructive), and assuming absence of region-polymorphic recursion. At the end of this chapter we draft some ideas on inferring bounds for full-fledged *Core-Safe* programs.

This chapter is an extended and improved version of [83]. A major difference w.r.t. the latter is that, in this chapter, we present a method for flattening an expression into sequences of basic expressions. This makes the algorithm simpler when considering the base and recursive cases of a function definition. Flattening also makes the algorithm more precise.

The techniques described in this chapter are based on abstract interpretation [33]. Given a function definition of n parameters, the abstract domain is the set of n-ary monotonic functions from real non-negative numbers to a real non-negative result[1]. This domain turns out to be a complete lattice under the usual $\sqsubseteq$ relation on functions. This domain is detailed in Section 7.2. In Section 7.3 we define an interpretation on expressions which is monotonic w.r.t. this domain, and in Section 7.4 we prove this interpretation correct. The interpretation itself allows us to devise a space analysis aimed at non-recursive function definitions. Recursive function definitions are dealt with in Section 7.5. After this, we show in Section 7.6 how the algorithm behaves with several examples and case studies. Finally, in Section 7.7 we sketch some ideas on inferring bounds in presence of destructive pattern matching and polymorphic recursion on regions, and Section 7.8 concludes.

7.2 Function signatures

Let us recall the resource-aware semantics of Section 2.7. There we annotated the evaluation of an expression e with a resource vector (δ, m, s), where δ gives the difference between the number of cells in each region after and before the evaluation of e, and m and s represent the number of heap cells (resp. stack words) needed for evaluating e. A space consumption analysis is expected to compute, given an expression e, upper bounds to each component (δ, m, s) of the resource vector resulting from its evaluation. However, this resource vector does not solely depend on the expression e itself, but also on the initial value environment E and heap h, which contain the *input* to our expression. As a consequence, there are two different ways in which a memory consumption analysis can be designed.

1. The memory consumption analysis is given the input as a parameter, in addition to the expression e to be analysed. In this way, the analysis returns a resource vector (δ', m', s') bounding the actual (δ, m, s) components.

2. We do not assume a fixed input, but the analysis returns a resource vector (δ', m', s') as a *function* of the given input.

The second option seems more reasonable from the programmer's point-of-view, since the input of a program is usually only known at runtime, whereas a memory consumption analysis is done at compile time. Nevertheless, the functions returned by the analysis would be complex and hard to interpret, unless we devise a simpler representation of the input, rather than a pair (E, h). It is more useful, in practice, to abstract the inputs by their *sizes*. For example, we usually say that the mergesort algorithm has $\mathcal{O}(n \log\ n)$ time complexity, being n the *number* of elements of the input list, without regard to the elements themselves. We will apply this abstraction in our analysis.

[1] By allowing real-valued functions instead of integer-valued, we can accommodate size models different from the one explained in this chapter.

Our first step is to define a suitable notion of *size*, so that the memory costs of a *Safe* function can be described as a function of this size. One may be tempted to define the size of a DS as the number of cells needed for storing it. However, this definition excludes the possibility of obtaining useful results for functions that receive an integer as an argument. An integer would always have zero size (since integers do not take heap cells by themselves), which would result in, for instance, a constant function giving the stack consumption of the *fib* function, whereas a linear function is expected. The notion of the size of the input should give an idea on how easy or difficult is to process that input.

Definition 7.1. Given a heap h and a value v, the *size* function is defined as follows:

- If v is a pointer p, *size* returns the number of cells taken by the recursive spine of the DSs which v points to. More formally:

$$size(h[p \mapsto (j, C\ \overline{v_i}^n)], p) = 1 + \sum_{i \in RecPos(C)} size(h, v_i)$$

- If v is a literal $c \in \mathbf{Int}$, then $size(h, c) = c$.
- If v is a literal $c \in \mathbf{Bool}$, then $size(h, c) = 0$.

It is worth noting that, with this definition, the size of a list with n elements is $n + 1$, since it is made of n cells with the $(:)$ constructor plus an additional cell with the $[\,]$ constructor. Analogously, the size of a binary search tree (as defined in Section 2.2) with n elements is always $2n + 1$ (n cells with the *Node* constructor and $n + 1$ cells with the *Empty* constructor).

Our space consumption analysis bounds the m and s components by means of numeric functions that depend on the size of the input. The δ component deserves special attention, since it is not a single number, but a mapping from heap region identifiers (natural numbers) to integers. We cannot know, at compile time, which regions are used to build DSs during the execution of an expression, since this information is only known at runtime by means of the value environment E, which associates region variables with actual region identifiers. As a consequence, we have to abstract the region identifiers by the type ρ of their corresponding region variable. To compute a bound to the δ component, our algorithm will compute a bound to the costs charged to each region type.

Recall that a *Core-Safe* function receives $n + m$ variables as parameters:

$$\begin{aligned} &f :: t_1 \to \cdots \to t_n \to \rho_1 \to \cdots \to \rho_m \to t \\ &f\ x_1 \cdots x_n\ @\ r_1 \cdots r_m = e_f \end{aligned} \tag{7.1}$$

Upper bounds to cost and sizes are represented by numbers in $\mathbb{R}^+_\infty \stackrel{\text{def}}{=} \mathbb{R}^+ \cup \{+\infty\}$. The special value $+\infty$ denotes an absence of upper bounds (either because there are no such bounds, or because the algorithm is not able to infer them). So, memory heap and stack needs are represented as functions of the following set:

$$\mathbb{F} = \{\xi : ((\mathbb{R}^+_\infty)^\perp)^n \to (\mathbb{R}^+_\infty)^\perp \mid \xi \text{ is monotonic and strict}\}$$

The notation $D^\perp$ denotes the set $D \cup \{\perp\}$. In this chapter we use the special value $\perp$ to make undefinedness of functions explicit. If $\xi \in \mathbb{F}$, we say that ξ is undefined for some sizes $\overline{x_i}^n$ if and only if $\xi\ \overline{x_i}^n = \perp$. Hence, the *domain* of a function $\xi \in \mathbb{F}$ (denoted by dom ξ) is defined as follows:

$$\text{dom } \xi = \{\overline{x_i} \in \left(\mathbb{R}^+_\infty\right)^n \mid \xi\ \overline{x_i} \neq \perp\}$$

The intuitive meaning of a cost function returning $\bot$ for a given size is that the function whose cost is being inferred does not evaluate to any value for that size. The strictness condition of space cost functions demands that the result is undefined if at least one of the arguments is undefined.

Example 7.2. Assume the following *Full-Safe* definition:

$$\mathit{dropTail}\ xs \quad = \quad \textbf{case}\ xs\ \textbf{of}\ (x : xx) \to [x]$$

The function application $\mathit{dropTail}\ [\,]$ does not evaluate to any value. Hence the memory needs of this function can be specified as follows:

$$\xi = \lambda xs. \begin{cases} 2 & \text{if } xs \geq 2 \\ \bot & \text{otherwise} \end{cases}$$

□

For the sake of clarity, we shall use curried notation when dealing with functions in $\mathbb{F}$ (in order to avoid an excessive amount of parentheses) and λ-notation when defining these functions (for instance, as in $\lambda x.x + 1$). We use $\xi, \xi_i, \ldots$ to denote a generic element of $\mathbb{F}$. If we use an element of $\mathbb{F}$ for bounding *heap* memory needs (that is, the m component in the resource vector (δ, m, s)) we shall use $\mu, \mu_1, \ldots$ as metavariables, whereas in the context of *stack* memory needs (the s component) we use $\sigma, \sigma_i, \ldots$

The following guard notation will be convenient in the following sections. Given a boolean function G of n variables and a function $\xi \in \mathbb{F}$, the notation $[G \to \xi]$ denotes the following function:

$$[G \to \xi] = \lambda \overline{x_i}. \begin{cases} \xi\ \overline{x_i} & \text{if } G\ \overline{x_i} \text{ holds} \\ \bot & \text{otherwise} \end{cases}$$

By abuse of notation we omit the $\lambda \overline{x_i}.$ prefix in G. For instance, the function ξ in Example 7.2 above can be expressed more succinctly as $[xs \geq 2 \to 2]$.

The usual ordering in $\mathbb{R}^+$ can be extended to $(\mathbb{R}^+_\infty)^\bot$ in the usual way:

$$x \leq y \quad \Leftrightarrow_{\text{def}} \quad x = \bot \vee y = +\infty \vee (x, y \in \mathbb{R}^+ \wedge x \leq y)$$

This order can be further extended to functions in $\mathbb{F}$ as follows:

$$\xi_1 \sqsubseteq \xi_2 \Leftrightarrow_{\text{def}} \forall \overline{x_i} \in \left(\mathbb{R}^+_\infty\right)^n . \xi_1\ \overline{x_i} \sqsubseteq \xi_2\ \overline{x_i}$$

The arithmetic operations $+$ and $*$ can also be extended to $(\mathbb{R}^+_\infty)^\bot$ as usual:

$$\left.\begin{array}{c} (+\infty) + x = x + (+\infty) = +\infty \\ (+\infty) * x = x * (+\infty) = +\infty \end{array}\right\} \text{ for every } x \in \mathbb{R}^+_\infty$$

$$\left.\begin{array}{c} \bot + x = x + \bot = \bot \\ \bot * x = x * \bot = \bot \end{array}\right\} \text{ for every } x \in \left(\mathbb{R}^+_\infty\right)^\bot$$

We have two different kinds of operators for denoting least upper bounds in $(\mathbb{R}^+_\infty)^\bot$, $\sqcup$ and $\biguplus$. The first one ignores undefined values, whereas the second one returns $\bot$ if at least one of the elements to which it is applied is $\bot$. For example, $\sqcup\{2, 5, \bot, 1\} = 5$, but $\biguplus\{2, 5, \bot, 1\} = \bot$.

The $+$, $*$, $\sqcup$ and $\sqcup\!\!\!\sqcup$ operators can be trivially extended to $(\mathbb{R}_\infty^+)^\perp$-valued functions in the usual way. It is easy to see these operators are monotonic on its arguments, and hence the $\mathbb{F}$ set is closed under these operators. Moreover, we obtain the following result:

Proposition 7.3. *Given a fixed function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$, its associated ordered set of space cost functions $(\mathbb{F}, \sqsubseteq)$ is a complete lattice, whose bottommost element is $\lambda\overline{x_i}.\perp$ and the topmost one is $\lambda\overline{x_i}.+\infty$.*

Proof. It follows directly from the fact that $((\mathbb{R}_\infty^+)^\perp, \leq)$ is a complete lattice (see [92, Section A.2]). □

With regard to the δ component of the resource vector we need to take the different region type variables into account. Assuming the function definition given in (7.1), its body can only charge space costs to the regions given as parameters (of types $\rho_1 \dots \rho_m$) and to the working region *self* (of type ρ_{self}). If we denote by R_f the set $\{\rho_1, \dots, \rho_m\}$, the function that gives the memory charges separated by region belong to the following class of functions:

$$\mathbb{D}^* = \{\Delta : ((\mathbb{R}_\infty^+)^\perp)^n \to (R_f \cup \{\rho_{self}\} \to \mathbb{R}_\infty^+)^\perp \mid \Delta \text{ is monotonic and strict}\}$$

and we use the variables $\Delta, \Delta_1, \dots$ to denote functions in this set. The fact that Δ is monotonic should be understood with regard to the following order relation:

$$\begin{gathered}\Delta\ \overline{x_i} \leq \Delta\ \overline{y_i} \Leftrightarrow_{\text{def}} \forall \rho \in R_f \cup \{\rho_{self}\}.\Delta\ \overline{x_i}\ \rho \leq \Delta\ \overline{y_i}\ \rho \\ \perp \leq \Delta\ \overline{y_i} \qquad \text{for every } \overline{y_i}\end{gathered}$$

which works in a componentwise basis.

When considering a function definition from outside, its charges to the *self* region are not visible, since this region is disposed of when the function finishes. In these cases it is more convenient to remove the ρ_{self} region from Δ, so as to get the following set of functions:

$$\mathbb{D} = \{\Delta : (\mathbb{R}_\infty^+)^n \to \left(R_f \to \mathbb{R}_\infty^+\right)^\perp \mid \Delta \text{ is monotonic}\}$$

The elements of $\mathbb{D}$ and $\mathbb{D}^*$ are called *abstract heaps*. Given an abstract heap $\Delta \in \mathbb{D}^*$, we denote by $\lfloor\Delta\rfloor$ the function $\lambda\overline{x_i}.(\Delta\ \overline{x_i})|_{R_f}$, which always belongs to $\mathbb{D}$. In other words, $\lfloor\Delta\rfloor$ discards the information regarding the *self* region.

We define the addition $+$, and the least upper bound $\sqcup$ between abstract heaps in a component-by-component basis, as usual. The same applies to the $\sqsubseteq$ relation. Slightly different is the multiplication operator $*$, which is defined in $\mathbb{F} \times \mathbb{D}$, and not between two abstract heaps, as expected. Given $\xi \in \mathbb{F}$ and $\Delta \in \mathbb{D}$, the result of $\xi * \Delta$ is defined as follows:

$$\xi * \Delta = \lambda\overline{x_i}. \begin{cases} \lambda\rho \in R_f.(\xi\ \overline{x_i}) * (\Delta\ \overline{x_i}\ \rho) & \text{if } \Delta\ \overline{x_i} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

The guarded notation described previously can also be used with the elements of $\mathbb{D}$ to obtain functions such as $[G \to \Delta]$ for some $\Delta \in \mathbb{D}$. Finally, we introduce the $|\cdot|$ operator as a function in $\mathbb{D} \to \mathbb{F}$, defined as follows:

$$|\Delta| = \lambda\overline{x_i}. \begin{cases} \sum_{\rho \in R_f} \Delta\ \overline{x_i}\ \rho & \text{if } \Delta\ \overline{x_i} \neq \perp \\ \perp & \text{otherwise} \end{cases}$$

Intuitively, the $|\cdot|$ operator adds the charges made to all RTVs into a single function, and it resembles the $|\delta|$ notation defined in Section 2.7.

All the operators $+, *, \sqcup, |\cdot|$ and the $\sqsubseteq$ relation can be easily extended to $\mathbb{D}^*$.

Example 7.4. Given the set $R_f = \{\rho_1, \rho_2, \rho_3\}$, let us define:

$$\Delta_1 = \lambda x. \begin{bmatrix} \rho_1 & \mapsto & x+1 \\ \rho_2 & \mapsto & 2 \\ \rho_3 & \mapsto & x^2+3 \end{bmatrix} \qquad \Delta_2 = [x \geq 2 \to \Delta_1] \qquad \xi = \lambda x.2x$$

then:

$$\Delta_1 + \Delta_2 = \lambda x. \left[x \geq 2 \to \begin{bmatrix} \rho_1 & \mapsto & 2x+2 \\ \rho_2 & \mapsto & 4 \\ \rho_3 & \mapsto & 2x^2+6 \end{bmatrix} \right]$$

$$\xi * \Delta_1 = \lambda x. \begin{bmatrix} \rho_1 & \mapsto & 2x^2+2x \\ \rho_2 & \mapsto & 4x \\ \rho_3 & \mapsto & 2x^3+6x \end{bmatrix}$$

$$|\Delta_2| = \lambda x.[x \geq 2 \to x^2 + x + 6]$$

and $\Delta_2 \sqsubseteq \Delta_1$ holds, but not $\Delta_1 \sqsubseteq \Delta_2$. □

Proposition 7.5. *Given a fixed function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$, its associated ordered set $(\mathbb{D}, \sqsubseteq)$ is a complete lattice, whose bottommost element is $\lambda\overline{x_i}.\bot$ and the topmost one is $\lambda\overline{x_i}.\lambda\rho \in R_f. +\infty$.*

Proof. Since $(\mathbb{R}^+_\infty, \leq)$ is a complete lattice, so is $(R_f \to \mathbb{R}^+_\infty, \sqsubseteq)$ with the component-wise ordering [92, Section A.2]. As a consequence, $((R_f \to \mathbb{R}^+_\infty)^\bot, \sqsubseteq)$ is a complete lattice, and so is $\mathbb{D}$, for the same reason. □

These definitions allow us to set up a correspondence between the actual memory consumption of an expression (given by the resource vector (δ, m, s) resulting from its evaluation) and the results of our analysis, which will be triples of the form (Δ, μ, σ), where $\Delta \in \mathbb{D}$ (or $\mathbb{D}^*$, depending on the context) and $\mu, \sigma \in \mathbb{F}$. The abstract heap Δ is meant to be an upper bound to the δ component, whereas μ and σ are upper bounds to the m and s components, respectively. The main difference is that the vector (δ, m, s) represents a *particular* execution of e for a given input size, and (Δ, μ, σ) captures *every possible execution* of the expression for every possible input size. As a consequence, we can abstract the resource behaviour of a function definition by means of its function signature.

Definition 7.6. A function signature for f is a triple (Δ, μ, σ), where Δ belongs to $\mathbb{D}$, and μ, σ belongs to $\mathbb{F}$.

The next step in this section is to precisely define what a *correct* function signature is. Assume the following $\Downarrow$-judgement of the body e_f of the function definition given in (7.1), whose derivation we call the *context derivation*.

$$E_0 \vdash h_0, k_0, e_f \Downarrow h_f, k_0, v_f, (\delta_f, m_f, s_f) \tag{7.2}$$

Intuitively, we expect a correct signature (Δ, μ, σ) to be an upper bound to the actual vector (δ_f, m_f, s_f) for every input size. In order to check that μ is a correct approximation of m_f, we compute the size of the function parameters from the value environment E_0 and the heap h_0, so we define $s_i \stackrel{\text{def}}{=} size(h_0, E_0(x_i))$

for every $i \in \{1..n\}$. If $\mu\ \overline{s_i}^n \geq m_f$, then μ has given a correct result for this particular execution of e_f. If we are able to prove that this holds for every possible execution of e_f, then μ correctly approximates the memory needs of e_f. For the σ we can proceed similarly. However, the case of Δ is more involved, since we have to take into account the correspondence between the RTVs $\rho_1 \ldots \rho_m$ and the actual regions given by $E_0(r_1) \ldots E_0(r_m)$. For this purpose we use the concept of region instantiation which was introduced when proving the correctness of *Safe*'s type system (see Definition 3.15 on page 94). If Γ_0 is the environment typing the expression e_f, then a region instantiation η is consistent with E_0 and Γ_0 it does not contradict the region instantiation defined by from E_0 and Γ_0, i.e. common RTVs are bound to the same actual region:

$$E_0(r) = i \wedge \Gamma_0(r) = \rho \Rightarrow \eta(\rho) = i$$

In case Γ_0 is injective, then $\eta = (E_0 \circ \Gamma_0^{-1})|_{R_f \cup \{\rho_{self}\}}$ is the only region instantiation consistent with E_0 and Γ_0. This is the case for our function bodies, since the region inference algorithm (Chapter 4) assigns different types to different region parameters. Henceforth we will abbreviate $\eta = (E_0 \circ \Gamma_0^{-1})|_{R_f \cup \{\rho_{self}\}}$.

Definition 7.7. Given a sequence of sizes $\overline{s_i}^n$ of the input arguments of the context function f, a number k of regions and a region instantiation η, we say that:

- Δ is an upper bound to δ in the context of $\overline{s_i}^n$, k and η, denoted $\Delta \succeq_{\overline{s_i}^n,k,\eta} \delta$ iff

$$\forall j \in \{0..k\}.\ \sum_{\eta(\rho)=j} \Delta\ \overline{s_i}^n\ \rho \geq \delta(j)$$

- μ is an upper bound to m in the context of $\overline{s_i}^n$, denoted $\mu \succeq_{\overline{s_i}^n} m$, iff $\mu\ \overline{s_i}^n \geq m$.
- σ is an upper bound to s in the context of $\overline{s_i}^n$, denoted $\sigma \succeq_{\overline{s_i}^n} s$, iff $\sigma\ \overline{s_i}^n \geq s$.

Example 7.8. Assume a function definition $f\ x\ y\ @\ r_1\ r_2 = e_f$ and the execution of its body e_f under a environment E and a heap h with three regions (that is, $k = 2$). We also assume that $s_x = size(h, E(x)) = 4$, $s_y = size(h, E(y)) = 2$ and that the function is typable under an environment Γ such that $\Gamma(r_i) = \rho_i$ for $i \in \{1, 2\}$. If $E(r_1) = 0$, $E(r_2) = 0$ and $E(self) = 1$, then we get $\eta = [\rho_1 \mapsto 0, \rho_2 \mapsto 0, \rho_{self} \mapsto 1]$. Let us define Δ as follows:

$$\Delta = \lambda x\, y. \begin{bmatrix} \rho_1 & \mapsto & 2x + y \\ \rho_2 & \mapsto & xy \\ \rho_{self} & \mapsto & 7x + 5 \end{bmatrix}$$

Then Δ is a correct upper bound to $\delta = [0 \mapsto 10, 1 \mapsto 32]$ in the context of s_x, s_y, k and η, since

$$\Delta\ 4\ 2 = \begin{bmatrix} \rho_1 & \mapsto & 10 \\ \rho_2 & \mapsto & 8 \\ \rho_{self} & \mapsto & 32 \end{bmatrix} \text{ and } \begin{cases} 10 + 8 \geq \delta(0) \\ 32 \geq \delta(1) \end{cases}$$

□

The following properties on the $\succeq$ will be useful in the following. In particular, we prove that the $\succeq$ relation on $\mathbb{D}$ (or $\mathbb{D}^*$) is preserved by $+$ and $|\cdot|$ operators.

Lemma 7.9. *If* $\Delta_1 \succeq_{\overline{s_i}^n,k,\eta} \delta_1$ *and* $\Delta_2 \succeq_{\overline{s_i}^n,k,\eta} \delta_2$ *then* $\Delta_1 + \Delta_2 \succeq_{\overline{s_i}^n,k,\eta} \delta_1 + \delta_2$.

Proof. Let $j \in \{0..k\}$. Then:

$$\sum_{\eta(\rho')=j} (\Delta_1 + \Delta_2)\ \overline{s_i}^n\ \rho = \sum_{\eta(\rho')=j} \Delta_1\ \overline{s_i}^n\ \rho + \sum_{\eta(\rho')=j} \Delta_2\ \overline{s_i}^n\ \rho \geq \delta_1(j) + \delta_2(j) = (\delta_1 + \delta_2)(j)$$

□

Lemma 7.10. *If* $\Delta \succeq_{\overline{s_i}^n,k,\eta} \delta$ *then* $|\Delta|\ \overline{s_i}^n \geq |\delta|$

Proof. We know that $\mathrm{dom}\,\eta = R_f$, so for every $\rho \in \mathrm{dom}\ \eta$ there exists some $i \in \{0..k\}$ such that $\eta(\rho) = i$. Thus we get:

$$|\Delta|\ \overline{s_i}^n = \sum_{\rho \in \mathrm{dom}\,\eta} \Delta\ \overline{s_i}^n\ \rho = \sum_{i=0}^{k} \sum_{\eta(\rho)=i} \Delta\ \overline{s_i}^n\ \rho \geq \sum_{i=0}^{k} \delta(i) = |\delta|$$

□

Given these definitions, we are ready to give a formal notion of a correct function signature.

Definition 7.11 (Correct signature)**.** Let (Δ, μ, σ) be the signature of a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ and Γ the type environment inferred for e_f. This signature is said to be correct if for all h, h', k, E_f, $\overline{v_i}^n$, $\overline{i_j}^m$, v, δ, m, s, $\overline{s_i}^n$, η such that:

1. $E_f \vdash h, k+1, e_f \Downarrow h', k+1, v, (\delta, m, s)$ where $E_f = [\overline{x_i \mapsto v_i}^n, \overline{r_j \mapsto i_j}^m, self \mapsto k+1]$
2. For each $i \in \{1..n\}$, $s_i = size(h, v_i)$
3. η is the consistent region instantiation determined by E_f and Γ.

then $\Delta \succeq_{\overline{s_i}^n,k,\eta} \delta|_k$, $\mu \succeq_{\overline{s_i}^n} m$, and $\sigma \succeq_{\overline{s_i}^n} s$.

7.3 Abstract interpretation

Once we have defined *what* a correct signature is, we need to know *how* to infer a correct signature for a given function definition. In this section we provide an abstract interpretation function which addresses this problem for non-recursive function definitions, and also serves as a basis for the inference of recursive function definitions.

Given an expression e , our aim is to find a tuple (Δ, μ, σ) which is an upper bound to its memory consumption. Firstly we have to distinguish between two kinds of *Core-Safe* expressions: literals, variables, copy expressions and function and constructor applications make up the set of *basic expressions*, whereas the set of *compound expressions* include, additionally, **let** and **case** expressions. The motivation for doing this is that basic expressions can be dealt with in an uniform way when inferring each of the components of the tuple (Δ, μ, σ), while the processing of **let** and **case** expressions differs substantially depending on the specific component. Therefore, we redefine the grammar of *Core-Safe* expressions as follows:

$$\begin{array}{rrcl} \textbf{BExp} \ni & be & ::= & c \mid x \mid x\ @\ r \mid a_1 \oplus a_2 \mid C\ \overline{a_i}\ @\ r \mid f\ \overline{a_i}^n\ @\ \overline{r_j}^m \\ \textbf{Exp} \ni & e & ::= & be \mid \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \mid \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \end{array}$$

Let us start defining the abstract interpretation function for basic expressions. This function is defined as a set of rules in Figure 7.1. The interpretation of a basic expression $[\![e]\!]$ receives a signature environment Σ : **Fun** $\to \mathbb{D} \times \mathbb{F} \times \mathbb{F}$ which maps each function name g appearing in e to its memory consumption signature $(\Delta_g, \mu_g, \sigma_g)$, as in Definition 7.6. It also receives a typing environment Γ : **RegVar** $\to$ **RegType**, giving the type of each region variable in scope, and the statically determined length td of the runtime environment when calling a function application. The latter has the same meaning as in the resource-aware semantics of Figure 2.26. The interpretation function is also

$$\overline{[\![c]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1)}$$

$$\overline{[\![x]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 1)}$$

$$\overline{[\![a_1 \oplus a_2]\!]\ \Sigma\ \Gamma\ td = ([\,]_f, 0, 2)}$$

$$\overline{[\![x\ @\ r]\!]\ \Sigma\ \Gamma\ td = ([\Gamma(r) \mapsto |x|]_f, |x|, 2)}$$

$$\overline{[\![C\ \overline{a_i}\ @\ r]\!]\ \Sigma\ \Gamma\ td = ([\Gamma(r) \mapsto 1]_f, 1, 1)}$$

$$\frac{\begin{array}{c}\Sigma(g) = (\Delta_g, \mu_g, \sigma_g) \qquad \theta = unify(\Gamma, g, \overline{r_j}) \\ G = \lambda\overline{x_j}.\left(\Delta_g\ \overline{|a_i|\ \overline{x_j}^{\,l}} \neq \bot \wedge \mu_g\ \overline{|a_i|\ \overline{x_j}^{\,l}} \neq \bot \wedge \sigma_g\ \overline{|a_i|\ \overline{x_j}^{\,l}} \neq \bot\right) \\ \Delta = [G \to \theta \downarrow_{\overline{|a_i|}} \Delta_g] \qquad \mu = [G \to \lambda\overline{x_j}.\mu_g\ (\overline{|a_i|\ \overline{x_j}^{\,l}})] \qquad \sigma = [G \to \lambda\overline{x_j}.\sigma_g\ (\overline{|a_i|\ \overline{x_j}^{\,l}})]\end{array}}{[\![g\ \overline{a_i}^{\,l}\ @\ \overline{r_j}^{\,q}]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sqcup\{l+q, l+q-td+\sigma\})}$$

Figure 7.1: Abstract interpretation function for basic expressions.

parametric on the context function definition f. However, since we assume this definition fixed (and to avoid excessive subscripting) we consider this parameter implicit. The notation $|x|$ denotes the size function $(\mathbb{R}^+_\infty)^n \to \mathbb{R}^+_\infty$ associated with x. These functions are assumed to have been inferred by an external size analysis. If R_f is the set of the context function's region parameters, the $[\,]_f$ notation stands for the abstract heap $\lambda\overline{x_i}.\lambda\rho.0$ where $\rho \in R_f$. Similarly, the binding $[\rho' \mapsto \xi]_f$ denotes the following abstract heap:

$$[\rho' \mapsto \xi]_f \stackrel{\text{def}}{=} \lambda\overline{x_i}.\lambda\rho.\begin{cases}\xi\ \overline{x_i} & \text{if } \rho{=}\rho' \\ 0 & \text{otherwise}\end{cases}$$

The integer values occurring in the rules should be understood as constant functions (that is $n \stackrel{\text{def}}{=} \lambda\overline{x_i}.n$). We omit the $\lambda\overline{x_i}$ for the sake of simplicity.

Notice the similarity between the results of the abstract interpretation in Figure 7.1 and the resource vectors of the semantic rules (Figure 2.26 on page 51). The only rule worth explaining is that of function application. Given a function definition $g\ \overline{y_i}^{\,l}\ @\ \overline{r_j}^{\,m}$, assume we want to infer a particular function application $g\ \overline{a_i}^{\,l}\ @\ \overline{r'_j}^{\,q}$. Firstly we retrieve the signature of g from the signature environment Σ, so let $(\Delta_g, \mu_g, \sigma_g) = \Sigma(g)$. Each component is a function which depends on the sizes of its l parameters $\overline{y_i}^{\,l}$, so we have to pass the sizes of the actual arguments $\overline{|a_i|}^{\,l}$, which, in turn, are functions of the parameters of the caller (that is why we have $\overline{|a_i|\ \overline{x_j}^{\,l}}$). The guard G discards those values $\overline{x_i}$ giving place to sizes $\overline{|a_i|\ \overline{x_i}^{\,l}}$ not belonging to the domain of Δ_g, μ_g or σ_g. Notice that, with regard to the computation of Δ, the type of the arguments in the function application may be instances of the function's type. Hence we have to find a correspondence between the RTVs of g and the RTVs of the particular instance used in the call. That is what the function *unify* does.

Definition 7.12. Given a type environment Γ, a function name $g \in$ **Fun** and a sequence of region variables $\overline{r_j}^{\,q}$, we say that $\theta = unify(\Gamma, g, \overline{r_j}^{\,q})$, iff $\Gamma(g) = \forall\overline{\alpha\rho}.\overline{t_i}^{\,l} \to \overline{\rho_j}^{\,q} \to t$, and $\forall j \in \{1..q\}.\theta(\rho_j) = \Gamma(r_j)$.

If there are several RTVs of g's type being mapped to the same RTV ρ in the function application, then the charges done to the region of type ρ are the sum of the charges made by g to the RTVs ρ' such

that $\theta(\rho') = \rho$. The $\downarrow$ operator does this computation. It is defined as follows:

$$\theta \downarrow_{\overline{|a_i|}} \Delta = \lambda \overline{x_i}.\lambda\rho. \sum_{\rho' \in \theta^{-1}(\rho)} \Delta\ \overline{|a_i|\ \overline{x_i}}^{l}\ \rho'$$

where $\rho \in R_f \cup \{\rho_{self}\}$.

Example 7.13. Assume a function g of type

$$g :: \forall \alpha, \rho_1, \rho_2, \rho_3.[\alpha]\ @\ \rho_1 \to \rho_1 \to \rho_2 \to \rho_3 \to T\ \alpha\ @\ \rho_2\ \rho_3$$

and the function call $g\ x\ @\ self\ r_1\ self$, where x has type $[Int]@\rho_{self}$, r_1 has type ρ, and $self$ has type ρ_{self}. Let us compare the general type of g with the particular instance of the function application:

General type :	$[\alpha]@\rho_1$	$\to$	ρ_1	$\to$	ρ_2	$\to$	ρ_3	$\to$ $T\ \alpha\ @\ \rho_2\ \rho_3$
Function application:	$[Int]@\rho_{self}$	$\to$	ρ_{self}	$\to$	ρ	$\to$	ρ_{self}	$\to$ $T\ Int\ @\ \rho\ \rho_{self}$

Hence we obtain $\theta = unify(\Gamma, g, self\ r_1\ self) = [\rho_1 \mapsto \rho_{self}, \rho_2 \mapsto \rho, \rho_3 \mapsto \rho_{self}]$. If Δ_g is defined as follows:

$$\Delta_g = \lambda y. \begin{bmatrix} \rho_1 & \mapsto & y^2 + 3 \\ \rho_2 & \mapsto & 3y \\ \rho_3 & \mapsto & 2y + 1 \end{bmatrix}$$

and $\xi = \lambda x.2x$, then we get:

$$\theta \downarrow_{\xi} \Delta_g = \lambda x. \begin{bmatrix} \rho_{self} & \mapsto & ((2x)^2 + 3) + (2 \cdot (2x) + 1) \\ \rho & \mapsto & 3 \cdot (2x) \end{bmatrix} = \lambda x. \begin{bmatrix} \rho_{self} & \mapsto & 4x^2 + 4x + 4 \\ \rho & \mapsto & 6x \end{bmatrix}$$

□

So far we have obtained a tuple $(\theta \downarrow_{\overline{|a_i|\ \overline{x_i}}^{l}} \Delta_g, \mu\ \overline{|a_i|\ \overline{x_i}}, \sigma\ \overline{|a_i|\ \overline{x_i}})$, which is an upper bound to the memory costs of the function's *body*. From these we can easily compute the costs of the function application itself: we just have to proceed as in rule [*App*] of Figure 2.26. We use the strict $\sqcup\!\!\downarrow$ operator, since we want the stack cost function of the whole application to be undefined in those values where G does not hold. If we used $\sqcup$ instead we would get in these values $\sqcup\{l + q, l + q - td + \bot\} = l + q$, where an undefined result is expected.

In the following we shall use the notation $[\![be]\!]_\Delta\ \Sigma\ \Gamma\ td$, $[\![be]\!]_\mu\ \Sigma\ \Gamma\ td$ and $[\![be]\!]_\sigma\ \Sigma\ \Gamma\ td$ to refer to the first, second and third components of $[\![be]\!]\ \Sigma\ \Gamma\ td$ respectively. In the first two cases we will omit the td since they are not relevant to the computation of Δ and μ, whereas with the third component we omit the Γ environment for the same reason. Sometimes we shall even leave out entirely the Σ and Γ when they are clear from the context.

An important result is the fact that the results given by the $[\![\cdot]\!]_\Delta$, $[\![\cdot]\!]_\mu$, and $[\![\cdot]\!]_\sigma$ abstract interpretations, when applied to basic expressions, have the same definition domain.

Lemma 7.14. *For every basic expression be and every Γ, Σ, td we get:*

$$dom\ [\![be]\!]_\Delta\ \Sigma\ \Gamma = dom\ [\![be]\!]_\mu\ \Sigma\ \Gamma = dom\ [\![be]\!]_\sigma\ \Sigma\ td$$

Proof. If *be* is not a function application the Lemma follows trivially, since the definition domain is $\mathbb{R}^+$

in these cases. If be is a function application, it holds because the results of $[\![be]\!]_\Delta$, $[\![be]\!]_\mu$ and $[\![be]\!]_\sigma$ are guarded by the same G, and the three are always distinct from $\perp$ if the guard holds. □

For inferring heap space consumption in presence of **let** and **case** expressions we have to apply a *flattening* transformation to the expression being analysed. This works as follows: an expression e is transformed into a set of *sequences* of basic expressions. Each sequence represents a possible execution flow of the expression. The whole set of sequences capture all the possible execution flows that may arise when executing this expression. This transformation destroys the structure of the program: instead of having nested expressions, the result comprises all the basic expressions being executed into a single "level" of nesting (that is why this transformation is called flattening). However, this transformation does not affect heap space consumption, since the latter does not depend on the structure of the expression. The following example provides an intuition on this fact.

Example 7.15. Given the following expressions:

$$e \equiv \textbf{let } x_1 = be_1 \textbf{ in } (\textbf{let } x_2 = be_2 \textbf{ in } be_3)$$

$$e' \equiv \textbf{let } x_1 = (\textbf{let } x_2 = be_1 \textbf{ in } be_2) \textbf{ in } be_3$$

Let (δ_i, m_i, s_i) the resource vector associated to the execution of the expression be_i for $i \in \{1..3\}$. If $x_2 \notin fv(be_3)$ and we can execute e, we can also execute e'. Let (δ, m, s) and (δ', m', s') be the resources of the execution of e and e' respectively. With the δ component, we get:

$$\delta = \delta_1 + (\delta_2 + \delta_3) \qquad \delta' = (\delta_1 + \delta_2) + \delta_3$$

with regard to the m component we obtain:

$$m = \sqcup\{m_1, |\delta_1| + \sqcup\{m_2, |\delta_2| + m_3\}\} \qquad m' = \sqcup\{\sqcup\{m_1, |\delta_1| + m_2\}, |\delta_1 + \delta_2| + m_3\}$$

The equality $\delta = \delta'$ follows trivially. The equality $m = m'$ follows from the fact that $|\delta_1 + \delta_2| = |\delta_1| + |\delta_2|$ and the associativity of the $\sqcup$ operator, so we get:

$$m = m' = \sqcup\{m_1, |\delta_1| + m_2, |\delta_1| + |\delta_2| + m_3\} \tag{7.3}$$

It is easy to show that, in both cases, the expression be_1 is the first to be executed, then be_2 and be_3 comes in the last place. That is why we can represent both expressions as a single sequence $[be_1, be_2, be_3]$. □

When considering sequences of expressions we lose the pattern guards of the **case** expressions. Nevertheless, these guards provide useful information on the size of the DS being matched against. For instance, assume the following expression:

$$\begin{array}{l}\textbf{case } x \textbf{ of}\\ \quad [\,] \to e_1\\ \quad (x : xx) \to e_2\end{array}$$

If the branch e_2 is executed, we know for sure[2] that the size of x must be greater or equal than 2. In general, this size information will be included in the execution sequences as *guards*, specifying under

[2] At this point we assume the absence of dangling pointers in the heap, since the expression is well-typed.

$$
\begin{array}{lcl}
\mathit{seqs}\ \mathit{be} & = & \{[\mathit{be}]\} \\
\mathit{seqs}\ (\mathbf{let}\ x_1 = e_1\ \mathbf{in}\ e_2) & = & \{\mathit{seq}_1 +\!\!+ \mathit{seq}_2 \mid \mathit{seq}_1 \in \mathit{seqs}\ e_1, \mathit{seq}_2 \in \mathit{seqs}\ e_2\} \\
\mathit{seqs}\ (\mathbf{case}\ x\ \mathbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}) & = & \bigcup_{i=1}^{n} (\mathit{seqs}\ e_i \wedge \lambda \overline{x_i}.(|x| \ \overline{x_i} \geq 1 + |\mathit{RecPos}(C_i)|))
\end{array}
$$

Figure 7.2: Definition of *seqs*, which converts an expression into a set of sequences.

which sizes of the **case** discriminant the execution of each sequence may take place. This motivates the following definition:

Definition 7.16. A sequence is a list of basic expressions $be_1, \ldots, be_n$ preceded by a boolean function G. We use the notation $[G \to be_1, ...be_n]$ to denote sequences. The notation $[be_1, \ldots, be_n]$ stands for the sequence $[\mathit{true} \to be_1, \ldots, be_n]$.

We define the concatenation ($+\!\!+$) of sequences as follows:

$$[G_1 \to be_1, \ldots, be_n] +\!\!+ [G_2 \to be'_1, \ldots, be'_m] = [G_1 \wedge G_2 \to be_1, \ldots, be_n, be'_1, \ldots, be'_m]$$

If seq is a sequence, we can strengthen its guard with the notation $\mathit{seq} \wedge G$, which stands for

$$[G_1 \to be_1, \ldots, be_n] \wedge G = [G_1 \wedge G \to be_1, \ldots, be_n]$$

and this notation can be extended to sets of sequences as follows:

$$S \wedge G = \{\mathit{seq} \wedge G \mid \mathit{seq} \in S\}$$

The function *seqs* transforms a *Core-Safe* expression into a set of sequences representing all the possible execution paths. It is defined in Figure 7.2. In order to transform a **let** expression, *seqs* gathers all the sequences of the auxiliary expression e_1 and the main expression e_2 and considers all the combinations. With respect to **case** expressions, it collects the sequences of each branch and adds the corresponding size guard imposed by the recursive positions of the pattern: the size of the discriminant must be one plus the number of recursive positions of its constructor.

Example 7.17. Assume the function *append* for joining two lists:

$$
\begin{array}{lcl}
\mathit{append}\ \mathit{xs}\ \mathit{ys}\ @\ r & = & \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
 & & \quad [\,] \to \mathit{ys} \\
 & & \quad (x : \mathit{xx}) \to \mathbf{let}\ x_1 = \mathit{append}\ \mathit{xx}\ \mathit{ys}\ @\ r\ \mathbf{in}\ (x : x_1)@r
\end{array}
$$

Its body e_{append} can be flattened into two sequences as follows:

$$\mathit{seqs}\ e_{\mathit{append}} = \{[\mathit{xs} \geq 1 \to \mathit{ys}], [\mathit{xs} \geq 2 \to \mathit{append}\ \mathit{xx}\ \mathit{ys}, (x : x_1)@r]\}$$

□

The extension of the abstract interpretation rules to compound expressions will be done in terms of their decomposition into sequences of basic expressions. Therefore our first step is to define the $[\![\cdot]\!]_\Delta$ and $[\![\cdot]\!]_\mu$ interpretations of such sequences. Example 7.15 gives us an intuition on how to define these interpretations. There we had three basic expressions with their associated costs $(\delta_1, m_1), (\delta_2, m_2)$ and (δ_3, m_3). The δ component is additive, in the sense that the δ component of the whole compound

expression is the sum of the δ's of its components. It is reasonable to think that its abstract counterpart (Δ) should be additive, as well. Hence, if (Δ_1, μ_1), (Δ_2, μ_2) and (Δ_3, μ_3) are the inferred costs of the respective basic expressions, we would obtain the following result:

$$[\![[be_1, be_2, be_3]]\!]_\Delta = \Delta_1 + \Delta_2 + \Delta_3 \tag{7.4}$$

With regard to the μ component, from the expression (7.3) we can define an "abstract" counterpart as follows:

$$[\![[be_1, be_2, be_3]]\!]_\mu = \sqcup\{\mu_1, |\Delta_1| + \mu_2, |\Delta_1| + |\Delta_2| + \mu_3\} \tag{7.5}$$

The intuition behind this expression is depicted in Figure 2.30, and its extension to three expressions. The reason of choosing $\sqcup$ instead of $\sqcup$ is that, if the costs of one of the basic expressions return an undefined value for some input sizes, we want to cancel out the whole sequence.

Now we can proceed to the generalization of the expressions given in (7.4) and (7.5) to guarded sequences of an arbitrary number of expressions. This is done as follows:

$$\begin{array}{lcl} [\![[G \to be_1, ..., be_n]]\!]_\Delta\ \Sigma\ \Gamma & = & [G \to \sum_{i=1}^{n} ([\![be_i]\!]_\Delta\ \Sigma\ \Gamma)] \\ [\![[G \to be_1, ..., be_n]]\!]_\mu\ \Sigma\ \Gamma & = & \left[G \to \bigsqcup_{i=1}^{n} \left(\sum_{j=1}^{i-1} |[\![be_j]\!]_\Delta\ \Sigma\ \Gamma| + [\![be_i]\!]_\mu\ \Sigma\ \Gamma\right)\right] \end{array}$$

It is easy to see that Lemma 7.14 can be extended when applying the interpretation to sequences of expressions:

Lemma 7.18. *For every sequence of expressions seq, every Σ and Γ, we get:*

$$dom\ ([\![seq]\!]_\Delta\ \Sigma\ \Gamma) = dom\ ([\![seq]\!]_\mu\ \Sigma\ \Gamma)$$

Proof. Let $seq = [G \to be_1, \ldots, be_n]$. Firstly we prove the $\subseteq$ inclusion. Let $\overline{x_i} \in$ dom $([\![seq]\!]_\Delta\ \Sigma\ \Gamma)$. Then:

1. $G(\overline{x_i})$ holds, and
2. $\overline{x_i} \in$ dom $[\![be_i]\!]_\Delta$ for each $i \in \{1..n\}$, which implies:
   (a) $\overline{x_i} \in$ dom $|[\![be_i]\!]_\Delta|$ for each $i \in \{1..n\}$, by definition of $|\cdot|$ operator.
   (b) $\overline{x_i} \in$ dom $[\![be_i]\!]_\mu$ for each $i \in \{1..n\}$, by Lemma 7.14.

Hence $\overline{x_i} \in$ dom $[\![seq]\!]_\mu$, by (1), (2a) and (2b). The $\supseteq$ inclusion can be proven similarly.

□

Example 7.19. Back to our *append* function, assume a signature Σ in which $\Sigma(append) = (f_\Delta, f_\mu, \lambda xs\, ys.0)$ for some functions $f_\Delta \in \mathbb{D}$ and $f_\mu \in \mathbb{F}$ such that dom $f_\Delta = \{(xs, ys) \mid xs \geq 2\}$ and dom $f_\mu = \{(xs, ys) \mid xs \geq 3\}$. Given the flattening of e_{append} shown in Example 7.17:

$$seqs\ e_{append} = \{\underbrace{[xs \geq 1 \to ys]}_{seq_1}, \underbrace{[xs \geq 2 \to append\ xx\ ys, (x : x_1)@r]}_{seq_2}\}$$

Let us apply the $[\![\cdot]\!]_\Delta$ and $[\![\cdot]\!]_\mu$ interpretations to seq_1 and seq_2 under an environment $\Gamma = [r \mapsto \rho_2]$ and assuming that $|xx| = \lambda xs\ ys.xs - 1$. ·We start with the first sequence:

$$
\begin{aligned}
[\![ys]\!]_\Delta &= \lambda xs\ ys.[\rho_2 \mapsto 0] \\
[\![seq_1]\!]_\Delta &= \lambda xs\ ys.[xs \geq 1 \to [\rho_2 \mapsto 0]]
\end{aligned}
$$

$$
\begin{aligned}
[\![ys]\!]_\mu &= \lambda xs\ ys.0 \\
[\![seq_1]\!]_\mu &= \lambda xs\ ys.[xs \geq 1 \to 0]
\end{aligned}
$$

whereas with the second sequence we get the following guard,

$$
\begin{aligned}
G &= \lambda xs\ ys.f_\Delta(xs) \neq \bot \wedge f_\mu(xs) \neq \bot \\
&= \lambda xs\ ys.xs \geq 2 \ \wedge\ xs \geq 3 \\
&= \lambda xs\ ys.xs \geq 3
\end{aligned}
$$

which is used in the respective interpretations of the *append* function application:

$$
\begin{aligned}
[\![append\ xs\ ys\ @\ r]\!]_\Delta &= \lambda xs\ ys.[G\ (xs-1)\ ys \to f_\Delta\ (xs-1)\ ys] \\
&= \lambda xs\ ys.[xs-1 \geq 3 \to [\rho_2 \mapsto f_\Delta\ (xs-1)\ ys\ \rho_2]] \\
&= \lambda xs\ ys.[xs \geq 4 \to [\rho_2 \mapsto f_\Delta\ (xs-1)\ ys\ \rho_2]]
\end{aligned}
$$

$$
[\![append\ xs\ ys\ @\ r]\!]_\mu = \lambda xs\ ys.[xs \geq 4 \to f_\mu\ (xs-1)\ ys]
$$

$$
[\![(x : x_1)@r]\!]_\Delta = \lambda xs\ ys.[\rho_2 \mapsto 1]
$$

$$
[\![(x : x_1)@r]\!]_\mu = \lambda xs\ ys.1
$$

$$
\begin{aligned}
[\![seq_2]\!]_\Delta &= \lambda xs\ ys.[xs \geq 2 \to [xs \geq 4 \to [\rho_2 \mapsto f_\Delta\ (xs-1)\ ys\ \rho_2]] + [\rho_2 \mapsto 1]] \\
&= \lambda xs\ ys.[xs \geq 2 \to [xs \geq 4 \to [\rho_2 \mapsto 1 + f_\Delta\ (xs-1)\ ys\ \rho_2]]] \\
&= \lambda xs\ ys.[xs \geq 4 \to [\rho_2 \mapsto 1 + f_\Delta\ (xs-1)\ ys\ \rho_2]]
\end{aligned}
$$

$$
\begin{aligned}
[\![seq_2]\!]_\mu &= \lambda xs\ ys.[xs \geq 2 \to \sqcup\{[xs \geq 4 \to f_\mu\ (xs-1)\ ys], [xs \geq 4 \to f_\Delta\ (xs-1)\ ys\ \rho_2 + 1]\}] \\
&= \lambda xs\ ys.[xs \geq 2 \to [xs \geq 4 \to \sqcup\{f_\mu\ (xs-1)\ ys, f_\Delta\ (xs-1)\ ys\ \rho_2 + 1\}]] \\
&\quad \lambda xs\ ys.[xs \geq 4 \to \sqcup\{f_\mu\ (xs-1)\ ys, f_\Delta\ (xs-1)\ ys\ \rho_2 + 1\}]
\end{aligned}
$$

Notice that, for every sequence, both interpretations have the same domain. □

Sometimes it is useful to express the $[\![\cdot]\!]_\Delta$ and $[\![\cdot]\!]_\mu$ interpretations of a sequence in terms of its subsequences. This is given by the following Lemma.

Lemma 7.20. *Given two sequences seq_1 and seq_2, the following facts hold:*

1. $[\![seq_1 +\!\!+ seq_2]\!]_\Delta\ \Sigma\ \Gamma = [\![seq_1]\!]_\Delta\ \Sigma\ \Gamma + [\![seq_2]\!]_\Delta\ \Sigma\ \Gamma$
2. $[\![seq_1 +\!\!+ seq_2]\!]_\mu\ \Sigma\ \Gamma = \sqcup\{[\![seq_1]\!]_\mu\ \Sigma\ \Gamma, |[\![seq_1]\!]_\Delta\ \Sigma\ \Gamma| + [\![seq_2]\!]_\mu\ \Sigma\ \Gamma\}$

Proof. Let us assume $seq_1 = [G_1 \to be_1, \ldots, be_k]$ and $seq_2 = [G_2 \to be'_1, \ldots, be'_m]$. We also define $[be_{k+1}, \ldots, be_{k+m}] = [be'_1, \ldots, be'_m]$. Firstly we prove that the functions occurring in both sides of each equation have the same domain. With regard to the first fact we get

$$
\begin{array}{ll}
& \overline{x_i} \in \mathrm{dom}\ [\![seq_1 +\!\!+ seq_2]\!]_\Delta \\
\Leftrightarrow & G_1\ \overline{x_i} \wedge G_2\ \overline{x_i} \wedge \forall j \in \{1..k+m\}.\overline{x_i} \in \mathrm{dom}\ [\![be_j]\!]_\Delta \\
\Leftrightarrow & G_1\ \overline{x_i} \wedge \forall j \in \{1..k\}.\overline{x_i} \in \mathrm{dom}\ [\![be_j]\!]_\Delta \wedge G_2\ \overline{x_i} \wedge \forall j \in \{k+1..k+m\}.\overline{x_i} \in \mathrm{dom}\ [\![be_j]\!]_\Delta \\
\Leftrightarrow & G_1\ \overline{x_i} \wedge \forall j \in \{1..k\}.\overline{x_i} \in \mathrm{dom}\ [\![be_j]\!]_\Delta \wedge G_2\ \overline{x_i} \wedge \forall l \in \{1..m\}.\overline{x_i} \in \mathrm{dom}\ [\![be'_l]\!]_\Delta \\
\Leftrightarrow & \overline{x_i} \in \mathrm{dom}\ [\![seq_1]\!]_\Delta\ \wedge\ \overline{x_i} \in \mathrm{dom}\ [\![seq_2]\!]_\Delta \\
\Leftrightarrow & \overline{x_i} \in \mathrm{dom}\ ([\![seq_1]\!]_\Delta + [\![seq_2]\!]_\Delta)
\end{array}
$$

whereas in the second one we get:

$$
\begin{array}{ll}
& \overline{x_i} \in \mathrm{dom}\ [\![seq_1 +\!\!+ seq_2]\!]_\mu \\
\Leftrightarrow & G_1\ \overline{x_i} \wedge G_2\ \overline{x_i} \wedge \forall j \in \{1..k+m\}.\overline{x_i} \in \mathrm{dom}\ \sum_{l=1}^{j-1} |[\![be_l]\!]_\Delta| + [\![be_j]\!]_\mu \\
\Leftrightarrow & G_1\ \overline{x_i} \wedge \forall j \in \{1..k\}.\overline{x_i} \in \mathrm{dom}\ \sum_{l=1}^{j-1} |[\![be_l]\!]_\Delta| + [\![be_j]\!]_\mu \\
& \wedge\ G_2\ \overline{x_i} \wedge \forall j \in \{1..m\}.\overline{x_i} \in \mathrm{dom}\ \sum_{l=1}^{j-1} |[\![be'_l]\!]_\Delta| + [\![be'_j]\!]_\mu \\
\Leftrightarrow & G_1\ \overline{x_i} \wedge \forall j \in \{1..k\}.\overline{x_i} \in \mathrm{dom}\ \sum_{l=1}^{j-1} |[\![be_l]\!]_\Delta| + [\![be_j]\!]_\mu \\
& \wedge\ G_2\ \overline{x_i} \wedge \forall j \in \{1..m\}.\overline{x_i} \in \mathrm{dom}\ \sum_{l=1}^{j-1} |[\![be'_l]\!]_\Delta| + [\![be'_j]\!]_\mu \\
& \wedge \forall j \in \{1..k\}.\overline{x_i} \in \mathrm{dom}\ |[\![be_j]\!]_\Delta| \\
\Leftrightarrow & \overline{x_i} \in \mathrm{dom}\ [\![seq_1]\!]_\mu \wedge \overline{x_i} \in \mathrm{dom}\ [\![seq_2]\!]_\mu \wedge \overline{x_i} \in \mathrm{dom}\ |[\![seq_1]\!]_\Delta| \\
\Leftrightarrow & \overline{x_i} \in \mathrm{dom}\ \sqcup\{[\![seq_1]\!]_\mu, |[\![seq_1]\!]_\Delta| + [\![seq_2]\!]_\mu\}
\end{array}
$$

Now let us assume that $D = \mathrm{dom}\ [\![seq_1 +\!\!+ seq_2]\!]_\Delta$ (and hence $D = \mathrm{dom}\ [\![seq_1 +\!\!+ seq_2]\!]_\mu$, by Lemma 7.14). If we restrict ourselves to this definition domain D, we can prove the first fact as follows:

$$
\begin{array}{rcll}
[\![seq_1 +\!\!+ seq_2]\!]_\Delta & = & [\![[be_1, \ldots, be_k, be_{k+1}, \ldots, be_{k+m}]]\!]_\Delta & \\
& = & \sum_{i=1}^{k+m} [\![be_i]\!]_\Delta & \\
& = & \sum_{i=1}^{k} [\![be_i]\!]_\Delta + \sum_{i=k+1}^{k+m} [\![be_i]\!]_\Delta & \\
& = & \sum_{i=1}^{k} [\![be_i]\!]_\Delta + \sum_{j=1}^{m} [\![be_{k+j}]\!]_\Delta & \\
& = & \sum_{i=1}^{k} [\![be_i]\!]_\Delta + \sum_{j=1}^{m} [\![be'_j]\!]_\Delta & \\
& = & [\![seq_1]\!]_\Delta + [\![seq_2]\!]_\Delta & \text{(in } D\text{)}
\end{array}
$$

Under the same conditions, we prove the second fact:

$$
\begin{array}{rcl}
[\![seq_1 +\!\!+ seq_2]\!]_\mu & = & [\![[be_1, \ldots, be_k, be_{k+1}, \ldots, be_{k+m}]]\!]_\mu \\
& = & \sqcup_{i=1}^{k+m} \left(\sum_{j=1}^{i-1} |[\![be_j]\!]_\Delta| + [\![be_i]\!]_\mu \right) \\
& = & \sqcup \left\{ \sqcup_{i=1}^{k} \left(\sum_{j=1}^{i-1} |[\![be_j]\!]_\Delta| + [\![be_i]\!]_\mu \right), \sqcup_{i=k+1}^{k+m} \left(\sum_{j=1}^{i-1} |[\![be_j]\!]_\Delta| + [\![be_i]\!]_\mu \right) \right\} \\
& = & \sqcup \left\{ [\![seq_1]\!]_\mu, \sqcup_{i=k+1}^{k+m} \left(\sum_{j=1}^{k} |[\![be_j]\!]_\Delta| + \sum_{j=k+1}^{i-1} |[\![be_j]\!]_\Delta| + [\![be_i]\!]_\mu \right) \right\} \\
& = & \sqcup \left\{ [\![seq_1]\!]_\mu, \sum_{j=1}^{k} |[\![be_j]\!]_\Delta| + \sqcup_{i=k+1}^{k+m} \left(\sum_{j=k+1}^{i-1} |[\![be_j]\!]_\Delta| + [\![be_i]\!]_\mu \right) \right\} \\
& = & \sqcup \left\{ [\![seq_1]\!]_\mu, \left| \sum_{j=1}^{k} [\![be_j]\!]_\Delta \right| + \sqcup_{l=1}^{m} \left(\sum_{j=k+1}^{k+l-1} |[\![be_j]\!]_\Delta| + [\![be_{k+l}]\!]_\mu \right) \right\} \\
& = & \sqcup \left\{ [\![seq_1]\!]_\mu, |[\![seq_1]\!]_\Delta| + \sqcup_{l=1}^{m} \left(\sum_{p=1}^{l-1} |[\![be_{k+p}]\!]_\Delta| + [\![be'_l]\!]_\mu \right) \right\} \\
& = & \sqcup \left\{ [\![seq_1]\!]_\mu, |[\![seq_1]\!]_\Delta| + \sqcup_{l=1}^{m} \left(\sum_{p=1}^{l-1} |[\![be'_p]\!]_\Delta| + [\![be'_l]\!]_\mu \right) \right\} \\
& = & \sqcup \{ [\![seq_1]\!]_\mu, |[\![seq_1]\!]_\Delta| + [\![seq_2]\!]_\mu \} \qquad \text{(in } D\text{)}
\end{array}
$$

□

In case a compound expression gives place to several sequences, we have to take the least upper bound of all of them. So, the abstract interpretation of a set S of sequences is defined as follows:

$$\begin{aligned}[\![S]\!]_\Delta\ \Sigma\ \Gamma &= \bigsqcup_{seq \in S} [\![seq]\!]_\Delta\ \Sigma\ \Gamma \\ [\![S]\!]_\mu\ \Sigma\ \Gamma &= \bigsqcup_{seq \in S} [\![seq]\!]_\mu\ \Sigma\ \Gamma\end{aligned}$$

Finally, the Δ and μ components of the abstract interpretation of a compound expression are given by the following definition:

$$\begin{aligned}[\![e]\!]_\Delta\ \Sigma\ \Gamma &= [\![seqs\ e]\!]_\Delta\ \Sigma\ \Gamma \\ [\![e]\!]_\mu\ \Sigma\ \Gamma &= [\![seqs\ e]\!]_\mu\ \Sigma\ \Gamma\end{aligned}$$

Example 7.21. Given the same signature and type environments of Example 7.19, we get:

$$\begin{aligned}[\![e_{append}]\!]_\Delta &= \lambda xs\ ys.[xs \geq 1 \to [\rho_2 \mapsto 0]] \sqcup [xs \geq 4 \to [\rho_2 \mapsto 1 + f_\Delta\ (xs-1)\ ys\ \rho_2]] \\ [\![e_{append}]\!]_\mu &= \lambda xs\ ys.[xs \geq 1 \to 0] \sqcup [xs \geq 4 \to \sqcup\{f_\mu\ (xs-1)\ ys, f_\Delta\ (xs-1)\ ys\ \rho_2 + 1\}]\end{aligned}$$

both of which can be expressed with piecewise functions as follows:

$$[\![e_{append}]\!]_\Delta = \lambda xs\ ys. \begin{cases} \bot & xs < 1 \\ [\rho_2 \mapsto 0] & 1 \leq xs < 4 \\ [\rho_2 \mapsto 1 + f_\Delta\ (xs-1)\ ys\ \rho_2] & 4 \leq xs \end{cases}$$

$$[\![e_{append}]\!]_\mu = \lambda xs\ ys. \begin{cases} \bot & xs < 1 \\ 0 & 1 \leq xs < 4 \\ \sqcup\{f_\mu\ (xs-1)\ ys, f_\Delta\ (xs-1)\ ys\ \rho_2 + 1\} & 4 \leq xs \end{cases}$$

□

Regarding the σ component for bounding the stack costs, we cannot apply the flattening-based approach, since this transformation breaks the structure of an expression and, unlike heap costs, the stack costs *do* depend on this structure, as the following example shows.

Example 7.22. Back to the expressions e and e' of Example 7.15, and assuming that s_i represent the stack costs of be_i, for $i \in \{1,2,3\}$. We get for e the following costs:

$$s = \sqcup\{2 + s_1, 1 + \sqcup\{2 + s_2, 1 + s_3\}\} = \sqcup\{2 + s_1, 3 + s_2, 2 + s_3\}$$

whereas with e' we obtain:

$$s' = \sqcup\{2 + \sqcup\{2 + s_1, 1 + s_2\}, 1 + s_3\} = \sqcup\{4 + s_1, 3 + s_2, 1 + s_3\}$$

which is clearly different from s. □

Instead of defining the $[\![\cdot]\!]_\sigma$ interpretation as a function of the sequences originated from a compound expressions, we have to define it in terms of the compound expression itself. The $[\![\cdot]\!]_\sigma$ interpreta-

tion of **let** expressions shown below roughly resembles its counterpart in the resource-aware semantics of Figure 2.26. With respect to **case** expressions, we proceed in a similar way, but we take the least upper bound of all the branches and include the corresponding guard:

$$\begin{array}{lcl} [\![\textbf{let } x_1 = e_1 \textbf{ in } e_2]\!]_\sigma\ \Sigma\ td & = & \sqcup\{2 + [\![e_1]\!]_\sigma\ \Sigma\ 0, 1 + [\![e_2]\!]_\sigma\ \Sigma\ (td+1)\} \\ [\![\textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}]\!]_\sigma\ \Sigma\ td & = & \bigsqcup_{i=1}^{n} [|x| \geq 1 + |RecPos(C_i)| \to n_i + [\![e_i]\!]_\sigma\ \Sigma\ (td + n_i)] \end{array}$$

Example 7.23. The $[\![\cdot]\!]_\sigma$ interpretation of our running example e_{append} leads to the following results with $td = 3$, if we assume that $\Sigma(append) = ([\,]_f, 0, f_\sigma)$ such that dom $f_\sigma = \{(xs, ys) \mid xs \geq 1\}$:

$$\begin{array}{lcl} [\![e_{append}]\!]_\sigma\ \Sigma\ 3 & = & [xs \geq 1 \to [\![ys]\!]_\sigma\ \Sigma\ 3] \sqcup [xs \geq 2 \to 2 + [\![\textbf{let } x_1 = \ldots]\!]_\sigma\ \Sigma\ 5] \\ & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 2 + \sqcup\{2 + [\![append\ xx\ ys\ @\ r]\!]_\sigma\ \Sigma\ 0, 1 + [\![(x : x_1)@r]\!]_\sigma\ \Sigma\ 6\}] \\ & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 2 + \sqcup\{2 + \sqcup\{3, 3 + [xs - 1 \geq 1 \to f_\sigma\ (xs-1)\ ys]\}, 1 + 1\}] \\ & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 2 + \sqcup\{[xs \geq 2 \to \sqcup\{5, 5 + f_\sigma\ (xs-1)\ ys\}], 2\}] \\ & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 2 + \sqcup\{[xs \geq 2 \to 5 + f_\sigma\ (xs-1)\ ys], 2\}] \\ & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to \sqcup\{7 + f_\sigma\ (xs-1)\ ys, 7\}] \\ & = & [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 7 + f_\sigma\ (xs-1)\ ys)] \end{array}$$

□

One may wonder why we have not followed the same approach with the $[\![\cdot]\!]_\Delta$ and $[\![\cdot]\!]_\mu$ interpretations. We could have directly defined these in terms of the **let** and **case** expressions, without any kind of flattening:

$$\begin{array}{lcl} [\![\textbf{let } x_1 = e_1 \textbf{ in } e_2]\!]_\Delta\ \Sigma\ \Gamma & = & [\![e_1]\!]_\Delta\ \Sigma\ \Gamma + [\![e_2]\!]_\Delta\ \Sigma\ \Gamma \\ [\![\textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}]\!]_\Delta\ \Sigma\ \Gamma & = & \bigsqcup_{i=1}^{n} [|x| \geq 1 + |RecPos(C_i)| \to [\![e_i]\!]_\Delta\ \Sigma\ \Gamma] \\ [\![\textbf{let } x_1 = e_1 \textbf{ in } e_2]\!]_\mu\ \Sigma\ \Gamma & = & \sqcup\{[\![e_1]\!]_\mu\ \Sigma\ \Gamma, |[\![e_1]\!]_\Delta\ \Sigma\ \Gamma| + [\![e_2]\!]_\mu\ \Sigma\ \Gamma\} \\ [\![\textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}]\!]_\mu\ \Sigma\ \Gamma & = & \bigsqcup_{i=1}^{n} \left[|x| \geq 1 + |RecPos(C_i)| \to [\![e_i]\!]_\mu\ \Sigma\ \Gamma\right] \end{array} \tag{7.6}$$

However, the interpretation given with these equations (as done in [83]) is, in some cases, less precise than the one given in terms of sequences, as the following example shows:

Example 7.24. Assume the following expression in the context of a function depending on a parameter b:

$$\begin{array}{rl} e \equiv \textbf{let } x_1 = & \textbf{case } b \textbf{ of} \\ & \quad True \to be_1 \\ & \quad False \to be_2 \\ \quad \textbf{in } be_3 & \end{array}$$

and let (Δ_i, μ_i) be the heap costs inferred for the basic expression be_i ($i \in \{1, 2, 3\}$). If we split e into sequences we get:

$$seqs\ e = \{[be_1, be_3], [be_2, be_3]\}$$

With the interpretation applied to sequences we get:

$$\mu = \sqcup \left\{ \begin{array}{l} \sqcup\{\mu_1, |\Delta_1| + \mu_3\} \\ \sqcup\{\mu_2, |\Delta_2| + \mu_3\} \end{array} \right\}$$

whereas with the equations (7.6) we would obtain:

$$\mu' = \sqcup\{\sqcup\{\mu_1, \mu_2\}, |\sqcup\{\Delta_1, \Delta_2\}| + \mu_3\}$$

Then $\mu' \sqsupset \mu$ if and only if $|\sqcup\{\Delta_1, \Delta_2\}| \sqsupset \sqcup\{|\Delta_1|, |\Delta_2|\}$. Assume $R_f = \{\rho_1, \rho_2\}$ and the following values of Δ_1 and Δ_2:

$$\Delta_1 = \lambda b. \begin{bmatrix} \rho_1 & \mapsto & 1 \\ \rho_2 & \mapsto & 2 \end{bmatrix} \qquad \Delta_2 = \lambda b. \begin{bmatrix} \rho_1 & \mapsto & 2 \\ \rho_2 & \mapsto & 1 \end{bmatrix}$$

We would get:

$$|\sqcup\{\Delta_1, \Delta_2\}| = \left|\lambda b. \begin{bmatrix} \rho_1 & \mapsto & 2 \\ \rho_2 & \mapsto & 2 \end{bmatrix}\right| = \lambda b.4, \quad \text{but} \quad \sqcup\{|\Delta_1|, |\Delta_2|\} = \sqcup\{\lambda b.3, \lambda b.3\} = \lambda b.3$$

So, in this particular case, μ' is strictly less precise than μ. □

7.4 Correctness of the abstract interpretation

In this section we aim to prove the following fact:

> *The tuple (Δ, μ, σ) resulting from the abstract interpretation of e is an upper bound to the actual resource vector (δ, m, s) resulting from the execution of e.*

The meaning of (Δ, μ, σ) being an upper bound to (δ, m, s) was already discussed in Section 7.2, in which we defined a $\succeq$ relation. However, this relation is parametric with respect some sizes of the input arguments. Hence, in order to prove the claim above we cannot consider e to be an expression in its own, but contained within a context function definition (7.1) with its set of input parameters. In the same way, we consider that the execution of e, represented as a $\Downarrow$-judgement, belongs to the context derivation specified in (7.2). So, our sizes $\overline{s_i}^n$ occurring in the $\succeq$ relations are obtained from the values of the *parameters* $\overline{x_i}$ in the value environment E_0 and heap h_0 of the context judgement. In other words, for every $i \in \{1..n\}$, $s_i = size(h_0, E_0)$.

We shall assume the admissibility condition (Definition 2.14 on page 34) on the context derivation:

$$\forall j \in \{1..m\}.E_0(r_j) < k_0 \qquad \text{and} \qquad E(self) = k_0$$

By Proposition 2.15 we know that this condition is propagated through the subderivations contained within it. This admissibility property is also propagated to the consistent region instantiation defined by E_0 and the typing environment Γ_0 of e_f, since $\Gamma_0(r_j) = \rho_j$ for all $j \in \{1..m\}$ and $\Gamma_0(self) = \rho_{self}$, hence

$$\forall j \in \{1..m\}.\eta(\rho_j) < k_0 \qquad \text{and} \qquad \eta(\rho_{self}) = k_0$$

The result of the abstract interpretation depends on some elements that are given as parameters to it, namely:

- The function signatures contained within the signature environment Σ passed to $[\![e]\!]$ as a parameter. If we want $[\![e]\!]$ to return correct results we have to ensure that the function signatures in Σ are correct approximations to the costs of their respective functions.
- The size functions $|\cdot|$ used in copy expressions and function applications. Although we do not tackle the problem of size inference, it is necessary to pose a condition of *correct size analysis* in

order to prove that the results of $[\![e]\!]$ are correct. In other words, we have to ensure that the sizes given by the analysis are exact or upper approximations of the actual runtime sizes of its corresponding DSs.

The first problem has been already addressed in Definition 7.11. With respect to the second one, we formulate the following notion of a correct size analysis.

Definition 7.25 (Correct size analysis)**.** Let $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$ be the context function. The size analysis $|\cdot|$ is correct if, given any initial environment E_0 such that the judgement (7.2) is derivable, then for all subexpressions e of e_f such that the judgement:

$$E \vdash h, k_0, td, e \Downarrow h', k_0, v, (\delta, m, s)$$

belongs to the derivation of (7.2) it holds that

$$\forall x \in \text{dom } E : |x|\ \overline{s_i}^n \geq size(h, E(x)) \quad \text{where } s_i = size(h_0, E_0(x_i)) \text{ for each } i \in \{1..n\}$$

In other words, a size function $|x|$ is correct if it is always greater that the size of the data structure pointed to by x at runtime.

The correctness proof of the abstract interpretation relies on the fact that both the signature environment and the size analysis are correct.

Theorem 7.26 (Correctness of the abstract interpretation)**.** *Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ a context function, Γ the inferred global type environment for e_f, Σ containing correct signatures for all the functions called from e_f, an initial environment E_0 and a heap h_0 such that the judgement (7.2) is derivable. For each subexpression e of e_f and E, td, Δ, μ, σ, h, ,h', v, ,t, δ, m, s, S such that:*

1. *$S = seqs\ e$.*
2. *For every $seq \in S$, every occurrence of $|x|$ in the evaluation of $[\![seq]\!]_\Delta\ \Sigma\ \Gamma$ and $[\![seq]\!]_\mu\ \Sigma\ \Gamma$ has been inferred with a correct size analysis.*
3. *$E \vdash h, k_0, td, e \Downarrow h', k_0, v, (\delta, m, s)$ belongs to the derivation of (7.2).*

then:

1. *$[\![e]\!]_\sigma\ \Sigma\ td \succeq_{\overline{s_i}^n} s$, and*
2. *There exists some $seq \in S$ such that $[\![seq]\!]_\Delta\ \Sigma\ \Gamma \succeq_{\overline{s_i}^n, k_0, \eta} \delta$ and $[\![seq]\!]_\mu\ \Sigma\ \Gamma \succeq_{\overline{s_i}^n} m$.*

where $\forall i \in \{1..n\}.s_i = size(h, E_0(x_i))$, and η is the consistent region instantiation determined by E and Γ.

Proof. By induction on the structure of e. In the following we will leave out the $\overline{s_i}^n$ and k_0 subscripts in the $\succeq$ relations for a better readability. We distinguish cases:

- **Cases** $e \equiv c$, $e \equiv x$

  We get a single sequence $[e]$, for which $[\![[e]]\!]_\Delta = \lambda \overline{x_i}^n.\lambda\rho.0$, $[\![[e]]\!]_\mu = \lambda \overline{x_i}^n.0$. Besides this, we get $[\![e]\!]_\sigma = \lambda \overline{x_i}^n.1$. With respect to the execution, we obtain $(\delta, m, s) = ([\,]_{k_0}, 0, 1)$. Thus we get:

  1. $[\![e]\!]_\sigma \succeq s$, since $[\![e]\!]_\sigma\ \overline{s_i}^n = 1 = s$.

2. $[\![[e]]\!]_\Delta \succeq \delta$ and $[\![[e]]\!]_\mu \succeq m$, since for every $j \in \{0..k_0\}$

$$\sum_{\eta(\rho)=j} [\![[e]]\!]_\Delta \, \overline{s_i}^n \, \rho = 0 = \delta(j)$$

$$[\![[e]]\!]_\mu \, \overline{s_i}^n = 0 = m$$

- **Case** $e \equiv a_1 \oplus a_2$

  It is similar to the previous case. The only difference is that $s = 2$ instead of 1, and $[\![e]\!]_\sigma = \lambda \overline{x_i}^n.2$ instead of $\lambda \overline{x_i}^n.1$.

- **Case** $e \equiv x \; @ \; r$

  Assume that $\Gamma(r) = \rho$ and $E(r) = j$. If we denote $m = size(h, E(x))$, the resulting resource vector is $([j \mapsto m]_k, m, 2)$, whereas the result of the analysis is $([\rho \mapsto |x|], |x|, 2)$. Then:

  1. $[\![[e]]\!]_\sigma \succeq s$ holds since $\sigma \, \overline{s_i}^n = 2 = s$.
  2. $[\![[e]]\!]_\Delta \succeq \delta$. Let $i \in \{0..k_0\}$. If $i = j$ then $\eta(\rho) = (E \circ \Gamma^{-1})(\rho) = i$. Hence:

$$\begin{array}{lcll} \sum_{\eta(\rho')=i} \Delta \, \overline{s_i}^n \, \rho' & \geq & \Delta \, \overline{s_i}^n \, \rho & \\ & = & |x| \, \overline{s_i}^n & \\ & \geq & size(h, E(x)) & \{\text{by Definition 7.25}\} \\ & = & \delta(i) & \end{array}$$

  If $i \neq j$ we get:

$$\sum_{\eta(\rho')=i} \Delta \, \overline{s_i}^n \, \rho' \geq 0 = \delta(i)$$

  The proof of $[\![[e]]\!]_\mu \succeq m$ is similar:

$$[\![[e]]\!]_\mu \, \overline{s_i}^n = |x| \, \overline{s_i}^n \geq size(h, E(x)) = m$$

- **Case** $e \equiv C \; \overline{a_i} \; @ \; r$

  Assume $\Gamma(r) = \rho$ and $E(r) = j$. Then $(\delta, m, s) = ([j \mapsto 1]_k, 1, 1)$ and $[\![e]\!] \; \Sigma \; \Gamma \; td = ([\rho \mapsto 1]_f, 1, 1)$. Again, we have a single sequence $seq = [e]$, so we prove:

  1. $[\![[e]]\!]_\sigma \succeq s$. It follows from the fact that $\sigma \, \overline{s_i}^n = 1 = s$.
  2. $[\![[e]]\!]_\Delta \succeq \delta$. Let $i \in \{0..k_0\}$, if $i \neq j$ we get:

$$\sum_{\eta(\rho')=i} \Delta \, \overline{s_i}^n \, \rho' \geq 0 = \delta(i)$$

  and with regard to j, we know that $\eta(\rho) = (E \circ \Gamma^{-1})(\rho) = j$, so

$$\sum_{\eta(\rho')=j} \Delta \, \overline{s_i}^n \, \rho' \geq \Delta \, \overline{s_i}^n \, \rho = 1 = \delta(j)$$

  The proof of $[\![[e]]\!]_\mu \succeq m$ follows from the definitions of μ and m:

$$\mu \, \overline{s_i}^n = 1 = m$$

- **Case** $e \equiv g\ \overline{a_i}^l\ @\ \overline{r'_j}^q$

We assume that $\Sigma\ g \equiv g\ \overline{y_i}^l\ @\ \overline{r''_j}^q = e_g$ and, by using the corresponding rule:

$$E_g \vdash h, k_0+1, l+q, e_g \Downarrow h', k_0+1, v, (\delta_g, m_g, s_g)$$
$$\text{where } E_g = \left[\overline{y_i \mapsto E(a_i)}^l, \overline{r''_j \mapsto E(r'_j)}^q, self \mapsto k_0+1\right]$$

The correctness of the signature $(\Delta_g, \mu_g, \sigma_g)$ follows by assumption. Moreover, g is well-typed and if $\Gamma(g) = \forall \overline{\alpha\rho}.\overline{t_i}^l \to \overline{\rho_j}^q \to t$, the global type Γ_g being inferred for e_g contains the bindings $[\overline{r''_j : \rho_j}^q]$. On the other hand, if $s_{i,g}$ denote the size of the i-th actual argument before evaluating the function's body (i.e. $\forall i \in \{1 \dots l\} : s_{i,g} = size(h, E_g(y_i))$) then, by our definition of correct signature:

$$\Delta_g \succeq_{\overline{s_{i,g}}^l, k_0, \eta'} \delta_g|_{k_0} \qquad \mu_g \succeq_{\overline{s_{i,g}}^l} m_g \qquad \sigma_g \succeq_{\overline{s_{i,g}}^l} s_g$$

where $\eta' = E_g \cdot \Gamma_g^{-1}$. This implies, in particular, that $\Delta_g\ \overline{s_{i,g}}^l \neq \bot$, $\mu_g\ \overline{s_{i,g}}^l \neq \bot$ and $\sigma_g\ \overline{s_{i,g}}^l \neq \bot$. By Definition 7.25 we get, for each $i \in \{1 \dots l\}$

$$|a_i|\ \overline{s_i}^n \geq size(h, E(a_i)) = size(h, E_g(y_i)) = s_{i,g} \tag{7.7}$$

So, by monotonicity of Δ_g, μ_g and σ_g we get $\Delta_g\ (\overline{|a_i|\ \overline{s_i}^n}^l) \neq \bot$, $\mu_g\ (\overline{|a_i|\ \overline{s_i}^n}^l) \neq \bot$ and $\sigma_g\ (\overline{|a_i|\ \overline{s_i}^n}^l) \neq \bot$, from which $G\ (\overline{|a_i|\ \overline{s_i}^n}^l)$ holds. Now we prove:

1. $[\![e]\!]_\sigma \succeq s$. Since σ_g is monotonic:

$$\begin{aligned}
[\![e]\!]_\sigma\ \overline{s_i}^n &= \sqcup\{l+q, \sigma_g\ \overline{(|a_i|\ \overline{s_j}^n)}^l - td + l + q\} \\
&\geq \sqcup\{l+q, \sigma_g\ \overline{s_{j,g}}^l - td + l + q\} \\
&\geq \sqcup\{l+q, s_g - td + l + q\} \\
&= s
\end{aligned}$$

2. $[\![[e]]\!]_\Delta \succeq_{\overline{s_i}^n, k_0, \eta} \delta$. Let $i \in \{0 \dots k_0\}$. Let us define $\Delta = \theta \downarrow_{\overline{|a_i|}} \Delta_g$:

$$\sum_{\eta(\rho)=i} \Delta\ \overline{s_i}^n\ \rho = \sum_{\eta(\rho)=i} \sum_{\theta(\rho')=\rho} \Delta_g\ \overline{|a_i|\ \overline{s_i}^n}^l\ \rho'$$

where $\theta = unify(\Gamma, g, \overline{r'_j}^q)$.

Because of monotonicity of Δ_g:

$$\sum_{\eta(\rho)=i} \Delta\ \overline{s_i}^n\ \rho \geq \sum_{\eta(\rho)=i} \sum_{\theta(\rho')=\rho} \Delta_g\ \overline{s_{i,g}}^l\ \rho' = \sum_{(\eta\circ\theta)(\rho')=i} \Delta_g\ \overline{s_{i,g}}^l\ \rho'$$

By definition of $\Delta_g \succeq_{\overline{s_{i,g}}^l, k_0, (\eta\cdot\theta)} \delta_g|_{k_0}$ and because of the fact that $i \neq k_0+1$, we can get the desired result:

$$\sum_{\eta(\rho)=i} \Delta\ \overline{s_i}^n\ \rho \geq \delta_g|_{k_0}(i) = \delta(i)$$

provided $\eta \circ \theta = \eta'$. However, for each $j \in \{1..q\}$:

$$\begin{array}{rcll} \eta(\theta(\rho_j)) & = & E(\Gamma^{-1}(\Gamma(r'_j))) & \{\text{by definition of } \eta \text{ and } \theta\} \\ & = & E(r'_j) & \\ & = & E_g(r''_j) & \{\text{by definition of } E_g\} \\ & = & E_g(\Gamma_g^{-1}(\rho_j)) & \{\text{by definition of } \Gamma_g\} \\ & = & \eta'(\rho_j) & \{\text{by definition of } \eta'\} \end{array}$$

So $[\![[e]]\!]_\Delta = \Delta \succeq_{\overline{s_i}^n, k_0, \eta} \delta$. Finally, we prove that $[\![[e]]\!]_\mu \succeq_{\overline{s_i}^n} m$:

$$\begin{array}{rcll} \mu\ \overline{s_i}^n & = & \mu_g\ \overline{|a_i|\ \overline{s_i}^n}^l & \\ & \geq & \mu_g\ \overline{s_{i,g}}^l & \{\text{because of (7.7) and monotonicity of } \mu_g\} \\ & \geq & m_g & \{\text{since } \mu_g \succeq_{\overline{s_{i,g}}^l} m_g\} \\ & = & m & \end{array}$$

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

  Let us denote by S_1 and S_2 the results of *seqs* e_1 and *seqs* e_2 respectively. Assume that the execution of e_1 returns (δ_1, m_1, s_1) as a resource vector, and the execution of e_2 returns (δ_2, m_2, s_2). By induction hypothesis we get the following facts:

$$[\![e_1]\!]_\sigma\ \Sigma\ 0 \succeq_{\overline{s_i}^n} s_1 \qquad [\![e_2]\!]_\sigma\ \Sigma\ (td+1) \succeq_{\overline{s_i}^n} s_2 \tag{7.8}$$

$$\exists seq_1 \in S_1 . [\![seq_1]\!]_\Delta \succeq_{\overline{s_i}^n, k, \eta} \delta_1 \wedge [\![seq_1]\!]_\mu \succeq_{\overline{s_i}^n} m_1 \tag{7.9}$$

$$\exists seq_2 \in S_2 . [\![seq_2]\!]_\Delta \succeq_{\overline{s_i}^n, k, \eta} \delta_2 \wedge [\![seq_2]\!]_\mu \succeq_{\overline{s_i}^n} m_2 \tag{7.10}$$

  Now we prove:

  1. $[\![e]\!]_\sigma \succeq_{\overline{s_i}^n} s$. This follows from (7.8) because:

$$\begin{array}{rcl} [\![e]\!]_\sigma\ \Sigma\ td\ \overline{s_i}^n & = & \sqcup\{2 + ([\![e_1]\!]_\sigma\ \Sigma\ 0)\ \overline{s_i}^n, 1 + ([\![e_2]\!]_\sigma\ \Sigma\ (td+1))\ \overline{s_i}^n\} \\ & \geq & \sqcup\{2 + s_1, 1 + s_2\} \\ & = & s \end{array}$$

  2. $[\![seq_1 +\!\!+ seq_2]\!]_\Delta \succeq_{\overline{s_i}^n, k, \eta} \delta$ and $[\![seq_1 +\!\!+ seq_2]\!]_\mu \succeq_{\overline{s_i}^n} m$. We get:

$$[\![seq_1 +\!\!+ seq_2]\!]_\Delta = [\![seq_1]\!]_\Delta + [\![seq_2]\!]_\Delta \succeq_{\overline{s_i}^n, k, \eta} \delta_1 + \delta_2 = \delta$$

$$[\![seq_1 +\!\!+ seq_2]\!]_\mu = \sqcup\{[\![seq_1]\!]_\mu, |[\![seq_1]\!]_\Delta| + [\![seq_2]\!]_\mu\} \succeq_{\overline{s_i}^n} \sqcup\{m_1, |\delta_1| + m_2\} = m$$

     In both cases the first step follows from Lemma 7.20, and the second one follows from (7.9), (7.10), and Lemmas 7.9 and 7.10.

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n$

  Assume the r-th branch being executed ($r \in \{1..n\}$). If (δ_r, m_r, s_r) is the resource vector associated to this branch, we get $\delta = \delta_r$, $m = m_r$ and $s = n_r + s_r$. By induction hypothesis we obtain:

$$[\![e_r]\!]_\sigma\ \Sigma\ (td + n_r) \succeq_{\overline{s_i}^n} s_r \tag{7.11}$$

$$\exists seq \in seqs\ e_r.[\![seq]\!]_\Delta \succeq_{\overline{s_i}^n,k,\eta} \delta_r = \delta \wedge [\![seq]\!]_\mu \succeq_{\overline{s_i}^n} m_r = m \tag{7.12}$$

For each $i \in \{1..n\}$ let us denote by G_i the guard $\lambda\overline{x_i}.|x|\ \overline{x_i} \geq 1 + RecPos(C_i)$. Since the initial configuration is closed, then $size(h, E(x)) \geq 1 + RecPos(C_r)$. By the correctness property of the size analysis, $|x|\ \overline{s_i}^n \geq size(h, E(x)) \geq 1 + RecPos(C_r)$, so $G_r(\overline{s_i}^n)$ holds. We prove below the conclusions of the Theorem:

1. $[\![e]\!]_\sigma \succeq_{\overline{s_i}^n} n_r + s_r$. We get:

$$\begin{aligned}
([\![e]\!]_\sigma\ \Sigma\ td)\ \overline{s_i}^n &= \bigsqcup_{i=1}^{n} [G_i \to n_i + [\![e_i]\!]_\sigma\ \Sigma\ \Gamma\ (td + n_i)]\ \overline{s_i}^n \\
&\geq [G_r \to n_r + [\![e_r]\!]_\sigma\ \Sigma\ \Gamma\ (td + n_r)]\ \overline{s_i}^n \\
&\geq n_r + s_r
\end{aligned}$$

   The last step being justified by (7.11) and the fact that $G_r(\overline{s_i}^n)$ holds.

2. The existence of a sequence $seq \in seqs\ e$ satisfying the conclusions of the theorem follows trivially from (7.12), since $seqs\ e = \bigcup_{i=1}^{n} (seqs\ e_i \wedge G_i)$ and $G_r(\overline{s_i}^n)$ holds.

□

Notice that we have not established that $[\![e]\!]_\Delta$ and $[\![e]\!]_\mu$ are upper bounds to the actual δ and m components, respectively. The previous theorem states a stronger property: there exists a sequence in $seqs\ e$ whose $[\![\cdot]\!]_\Delta$ and $[\![\cdot]\!]_\mu$-interpretations are upper bounds to the δ and m, respectively. However, and since $[\![e]\!]_\Delta$ and $[\![e]\!]_\mu$ are defined as $[\![seqs\ e]\!]_\Delta = \bigsqcup_{seq \in seqs\ e} [\![seq]\!]_\Delta$ and $[\![seqs\ e]\!]_\mu = \bigsqcup_{seq \in seqs\ e} [\![seq]\!]_\mu$, it follows trivially that $[\![e]\!]_\Delta$ and $[\![e]\!]_\mu$ are correct approximations to δ and m.

From this correctness result we can devise a way for inferring upper bounds to the heap and stack consumption of a non-recursive function, assuming that the functions called from it have already been inferred. We just have to apply the abstract interpretation to the body of the function definition.

Example 7.27. Let us recall the *combNumber* function, which, given two integers n and m, returns the result of $\binom{n}{m}$ (see Example 4.20 on page 151). Assuming our signature environment Σ has the following information regarding the stack costs of $(!!)$ and *pascal*:

$$\begin{aligned}
\sigma_{(!!)} &= \lambda xs\ m.[xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 6] \\
\sigma_{pascal} &= \lambda n.15n + 4
\end{aligned}$$

If $e_{combNumber}$ denotes the body of the *combNumber* function, we get:

$$[\![e_{combNumber}]\!]_\sigma\ \Sigma\ 2 = \lambda n\ m.[n \geq 0 \to 15n + 14] \sqcup [n \geq 1 \to 15n + 17]$$

which is a correct bound to the stack costs of *combNumber*, provided $\sigma_{(!!)}$ and σ_{pascal} are correct upper bounds to the stack consumption of their respective functions. □

The inference of recursive function definitions is far more involved, and its study is deferred to the following section.

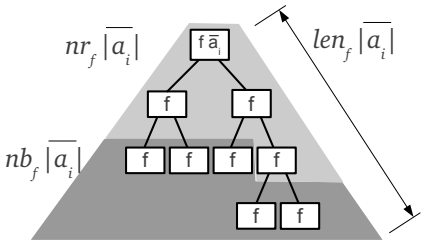


Figure 7.3: Representation of the activation tree of a given function call to f. The result of $nr_f\ \overline{|a_i|}$ and $nb_f\ \overline{|a_i|}$ are the number of internal nodes and leaves, respectively. The value of $len_f\ \overline{|a_i|}$ represents the height of the activation tree.

7.5 Memory consumption of recursive function definitions

The obvious question that arises when applying the abstract interpretation to a recursive function is which signature associated to f must be stored into the signature environment Σ. The result of the abstract interpretation will be correct if this initial signature is correct, but this signature is what we aim to infer. We have to find a correct upper bound to the memory consumptions of a function definition by other means different from the $[\![\cdot]\!]$-interpretations. Once these upper bounds are computed, it still makes sense to apply the abstract interpretation under a signature environment Σ containing those bounds. The new results are also correct, by Theorem 7.26, but they might be more precise than the initial ones, as we will see later.

The rest of this section is devoted to the computation of these *initial approximations*, which will be called Δ_0, μ_0 and σ_0. In order to compute them, we need some information regarding the number of recursive calls produced during the evaluation of a call to f. This information will be given as a function of the input sizes $\overline{x_i}$. We represent these recursive calls by means of activation trees (or *call trees*), as in Figure 7.3. Before computing an upper bound to the memory costs, we assume that the following information is available as elements of $\mathbb{F}$:

nr_f Upper bound to the number of calls to f which invoke f again. This number corresponds to the internal nodes of f's call tree.

nb_f Upper bound to the number of basic calls to f that do not invoke f again. It corresponds to the leaf nodes of f's call tree.

len_f Upper bound to the maximum length of f's call chains. It corresponds to the height of f's call tree.

In general these functions are not independent of each other. For instance, with linear recursion we get $nr_f = len_f - 1$ and $nb_f = 1$. However, we shall not assume a fixed relation between them. The computation of these three functions is closely related to the problem of termination and the computation of *ranking functions*, and we can use the techniques described in [73] for computing them. Another possibility is to give a definition of these components as a recurrence relation and obtain a closed form by using recurrence solving tools, such as PUBS [7, 8], possibly in combination with polynomial interpolation-based techniques [87].

Example 7.28. Assume a call to *append xs ys* where $xs = [x_1, \ldots, x_n]$. We get the following call sequence:

$$\begin{array}{lll} & append\ [x_1, \ldots, x_n] & \text{1st call} \\ \rightarrow & append\ [x_2, \ldots, x_n] & \text{2nd call} \\ \vdots & & \\ \rightarrow & append\ [x_n] & n\text{-th call} \\ \rightarrow & append\ [\,] & (n+1)\text{-th call} \end{array}$$

So we obtain $n+1$ calls to *append*, n of which are recursive. Since the size of a list is its number of elements plus one, we get the following functions:

$$\begin{array}{rcl} nb_{append} & = & \lambda xs\ ys.1 \\ nr_{append} & = & \lambda xs\ ys.xs - 1 \\ len_{append} & = & \lambda xs\ ys.xs \end{array}$$

□

In the following subsections we shall present three algorithms for computing our initial Δ_0, μ_0 and σ_0 (sections 7.5.3, 7.5.4 and 7.5.5, respectively). For each algorithm, we have to prove the following facts:

[CR] The obtained bounds Δ_0, μ_0 and σ_0 are correct bounds to the actual memory needs of the program.

[RD] The results of $[\![e_f]\!]_\Delta$, $[\![e_f]\!]_\mu$ and $[\![e_f]\!]_\sigma$ (being e_f the body of the function definition) under a signature environment mapping f to $(\Delta_0, \mu_0, \sigma_0)$ are equal or more precise than the initial approximations $(\Delta_0, \mu_0, \sigma_0)$.

The latter fact is closely related to the concept of reductive functions over complete lattices. In the next subsection we shall briefly describe all these concepts, including *Tarki's Fixed Point Theorem*, which is of crucial importance for proving that, under some conditions on the given nb_f, nr_f and len_f, **[RD]** implies **[CR]**. In case these conditions do not hold, **[RD]** may not hold, but **[CR]** still holds. Subsection 7.5.6 addresses this situation.

7.5.1 Preliminaries on fixed points in complete lattices

Let $(L, \sqsubseteq)$ be a complete lattice and f a monotonic function on L. Given an element $x \in L$:

- We say that x is a *fixed point* of f iff $f(x) = x$.
- We say that f is *reductive* at x iff $f(x) \sqsubseteq x$.
- We say that f is *extensive* at x iff $f(x) \sqsupseteq x$.

We denote by $Fix(f)$ the set of fixed points of f:

$$Fix(f) = \{x \in L \mid f(x) = x\}$$

Similarly, we denote by $Red(f)$ (resp. $Ext(f)$) the set of points upon which f is reductive (resp. extensive).

$$\begin{array}{rcl} Red(f) & = & \{x \in L \mid f(x) \sqsubseteq x\} \\ Ext(f) & = & \{x \in L \mid f(x) \sqsupseteq x\} \end{array}$$

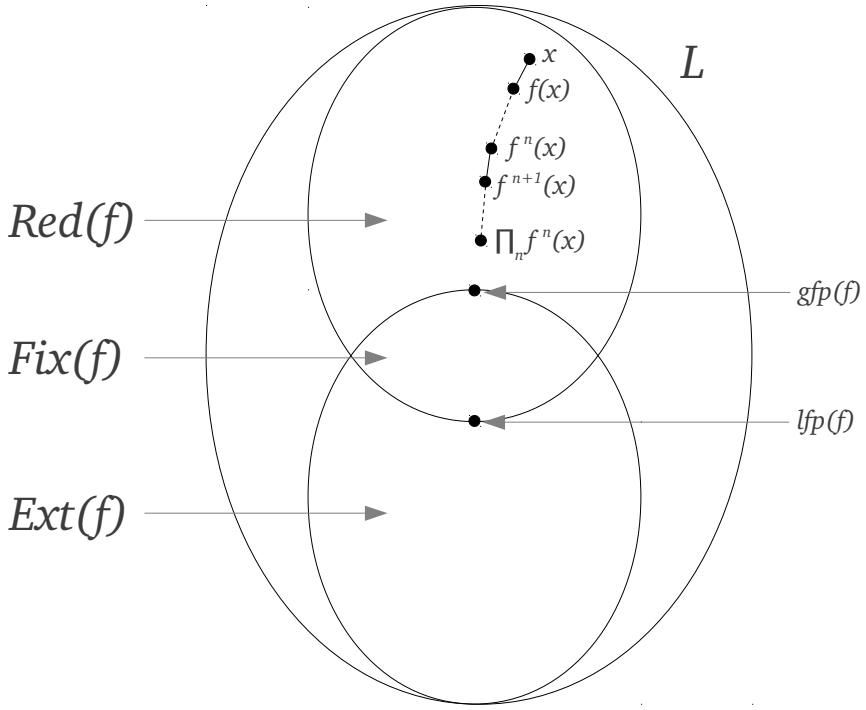


Figure 7.4: Representation of the points upon which a given function is reductive and extensive. The intersection of $Red(f)$ and $Ext(f)$ is the set of fixed points of f. Given an element $x \in Red(f)$, $f^n(x)$ is always above the least fixed point.

Given the fact that L is a complete lattice, the least upper bound and the greatest lower bounds of $Fix(f)$ are both defined and respectively denoted by $lfp(f)$ and $gfp(f)$. Tarski's fixed fixed point theorem [111] establishes the relation between fixed points and the reductivity/extensivity properties.

Theorem 7.29 (Tarski 1955, taken from [92])**.** *Let $(L, \sqsubseteq)$ a complete lattice and $f : L \to L$ a monotone function. Then $lfp(f) = \sqcap Red(f)$ and $gfp(f) = \sqcup Ext(f)$.*

Proof. (see [92, Section A.4]). □

Figure 7.4 depicts the layout of the reductive and extensive elements of L w.r.t. a function f. If x is an element belonging to $Red(f)$, then $f(x) \sqsubseteq x$. By monotonicity of f, we get $f(f(x)) \sqsubseteq f(x)$. By repeating this process we get the following chain,

$$x \sqsupseteq f(x) \sqsupseteq f^2(x) \sqsupseteq \ldots \sqsupseteq f^n(x) \sqsupseteq \ldots$$

whose elements, by Tarski's theorem, are located above the least fixed point.

Now let us apply these concepts to our particular abstract interpretation. From Propositions 7.3 and 7.5 we know that $(\mathbb{F}, \sqsubseteq)$ and $(\mathbb{D}, \sqsubseteq)$ are complete lattices. Given the context function definition f, and some fixed Σ, Γ and td, the iteration of the abstract interpretation can be understood as a function on abstract heaps $\mathbb{D} \to \mathbb{D}$ (resp. on cost functions $\mathbb{F} \to \mathbb{F}$) which, given an input Δ (resp. μ, σ), inserts it into the signature environment Σ, and apply the $[\![e_f]\!]_\Delta$ interpretation (resp. $[\![e_f]\!]_\mu$ and $[\![e_f]\!]_\sigma$) to the body of f.

Definition 7.30. Assume a function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$, and some fixed Σ, Γ, Δ and td, such that

$f \notin \text{dom}\ \Sigma$. The operators $\mathcal{I}_f : \mathbb{D} \to \mathbb{D}$, $\mathcal{M}_{\Delta,f} : \mathbb{F} \to \mathbb{F}$ and $\mathcal{S}_f : \mathbb{F} \to \mathbb{F}$ are defined as follows:

$$\begin{array}{rcl} \mathcal{I}_f(\Delta) & = & \left\lfloor [\![e_f]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma\ td \right\rfloor \\ \mathcal{M}_{\Delta,f}(\mu) & = & [\![e_f]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, \mu, 0)])\ \Gamma\ td \\ \mathcal{S}_f(\sigma) & = & [\![e_f]\!]_\sigma\ (\Sigma \uplus [f \mapsto ([\,]_f, 0, \sigma)])\ \Gamma\ td \end{array}$$

Notice that, strictly speaking, the $\mathcal{I}$, $\mathcal{M}$ and $\mathcal{S}$ operators are also parametric on the given Σ, Γ and td, but we assume all these elements fixed. Now we prove their monotonocity.

Proposition 7.31. *The operators $\mathcal{I}_f$, $\mathcal{M}_{\Delta,f}$ and $\mathcal{S}_f$ are monotonic on their input arguments.*

Proof. From the definition of the abstract interpretation it follows that the result of $\mathcal{I}$, $\mathcal{M}$ and $\mathcal{S}$ is computed by composing size functions with the $\sqcup, \sqcup\!\!\sqcup, +$ operators, and possibly with the substraction of a constant value (td). All these operations are monotone w.r.t. their inputs, and so is the composition of all of them. □

For each of these operators, we can define its set of fixed points, reductive elements and extensive elements in the same way as above.

Example 7.32. Consider the *append* function of Example 7.17. Let Δ be an abstract heap such that $\text{dom}\ \Delta = \{(xs, ys) \mid xs \geq 1\}$. If we assume that $\Gamma = [r_2 : \rho_2]$, our operator $\mathcal{I}_{append}$ is defined as follows:

$$\mathcal{I}_{append}(\Delta)\ xs\ ys\ \rho_2 = [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1 + \Delta(xs - 1, ys)]$$

Let us consider the abstract heaps Δ_1, Δ_2 and Δ_3 defined as follows:

$$\begin{array}{rcl} \Delta_1 & \stackrel{\text{def}}{=} & \lambda xs\ ys.[xs \geq 1 \to [\rho_2 \mapsto 0]] \\ \Delta_2 & \stackrel{\text{def}}{=} & \lambda xs\ ys.[xs \geq 1 \to [\rho_2 \mapsto \lfloor xs - 1 \rfloor]] \\ \Delta_3 & \stackrel{\text{def}}{=} & \lambda xs\ ys.[xs \geq 1 \to [\rho_2 \mapsto 2xs]] \end{array}$$

If we apply the $\mathcal{I}_{append}$ iterator to each of these, we obtain the following results:

$$\begin{array}{rcl} \mathcal{I}_{append}(\Delta_1)\ xs\ ys\ \rho_2 & = & [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1 + 0] \\ & = & [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1] \end{array}$$

$$\begin{array}{rcl} \mathcal{I}_{append}(\Delta_2)\ xs\ ys\ \rho_2 & = & [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1 + \lfloor xs - 2 \rfloor] \\ & = & [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to \lfloor xs - 1 \rfloor] \\ & = & [xs \geq 1 \to \lfloor xs - 1 \rfloor] \end{array}$$

$$\begin{array}{rcl} \mathcal{I}_{append}(\Delta_3)\ xs\ ys\ \rho_2 & = & [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1 + 2(xs - 1)] \\ & = & [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 2xs - 1] \end{array}$$

Therefore, $\Delta_1 \in \mathit{Ext}(\mathcal{I}_{append})$, $\Delta_2 \in \mathit{Fix}(\mathcal{I}_{append})$, and $\Delta_3 \in \mathit{Red}(\mathcal{I}_{append})$. □

An important result is that, if we take a fixed point of each operator, the resulting signature is correct for the given function definition. The precise statement of this fact is rather involved and beyond the scope of this work. For its purposes we consider a simplified version, and refer to Javier de Dios' PhD thesis [35] for more details.

Theorem 7.33. *Let (Δ, μ, σ) be a signature for f. If $\Delta \in Fix(\mathcal{I}_f)$, $\mu \in Fix(\mathcal{M}_{\Delta,f})$ and $\sigma \in Fix(\mathcal{S}_f)$, the signature (Δ, μ, σ) is correct for f.*

Proof. By induction on the maximum number of nested calls to f in the corresponding $\Downarrow$-derivation. The full proof can be found in [35]. □

However, and by Tarski's Theorem (7.29), the reductive elements of each operator are always located above the corresponding least fixed point, so these can also be considered as correct bounds.

Corollary 7.34. *Let (Δ, μ, σ) be a signature for f. If $\Delta \in Red(\mathcal{I}_f)$, $\mu \in Red(\mathcal{M}_{\Delta,f})$ and $\sigma \in Red(\mathcal{S}_f)$, the signature (Δ, μ, σ) is correct for f.*

Proof. By Theorem 7.33, $(lfp(\mathcal{I}_f), lfp(\mathcal{M}_{\Delta,f}), lfp(\mathcal{S}_f))$ is a correct upper bound to f. Let Δ, μ and σ defined as above. By Theorem 7.29 we get $\Delta \sqsupseteq lfp(\mathcal{I}_f) \in Fix(\mathcal{I}_f)$, $\mu \sqsupseteq lfp(\mathcal{M}_{\Delta,f}) \in Fix(\mathcal{M}_{\Delta,f})$, and $\sigma \sqsupseteq lfp(\mathcal{S}_f) \in Fix(\mathcal{S}_f)$. Thus, (Δ, μ, σ) is also a correct bound. □

A consequence of this Corollary is that the **[RD]** condition shown above is sufficient in order to prove the correctness condition **[CR]** of the initial bounds.

7.5.2 Splitting *Core-Safe* sequences

When computing initial approximations to the heap and stack consumption of a function definition, it is useful to separate the costs charged by its base and recursive cases. The result of flattening an expression into a sequence of basic expressions (via the *seqs* function) is quite amenable to this separation: a sequence takes part in the base case if it does not contain recursive calls, and it takes part in the recursive one if it does contain them.

Definition 7.35. Given the context function f whose memory consumption is being inferred, a sequence of basic expressions is said to be a *base sequence* if it does not contain any expression of the form $f\ \overline{a_i}\ @\ \overline{r_j}$ for some $\overline{a_i}$ and $\overline{r_j}$. Otherwise it is said to be *recursive*.

We define the $splitExp_f$ function, which classifies an arbitrary set of sequences into base and recursive sequences.

Definition 7.36. Given a set of sequences S and a context function f, the $splitExp_f$ function is defined as follows:

$$splitExp_f\ S = (S_b, S_r)\ \textbf{where}\ \begin{cases} S_b = \{seq \in S \mid seq \text{ is a base sequence}\} \\ S_r = \{seq \in S \mid seq \text{ is a recursive sequence}\} \end{cases}$$

Obviously, if $splitExp_f\ S = (S_b, S_r)$, then $S = S_b \uplus S_r$.

When computing the charges to the working region, it is also useful to separate the charges done until (and including) the last recursive call. Again, we can do this in terms of sequences. It is easy to see that, if *seq* is a recursive sequence of the body of the context function f, we can split it into two sequences seq_{bef} and seq_{aft} such that $seq = seq_{bef} +\!\!+ seq_{aft}$, the last element of seq_{bef} is of the form $f\ \overline{a_i}\ @\ \overline{r_j}$

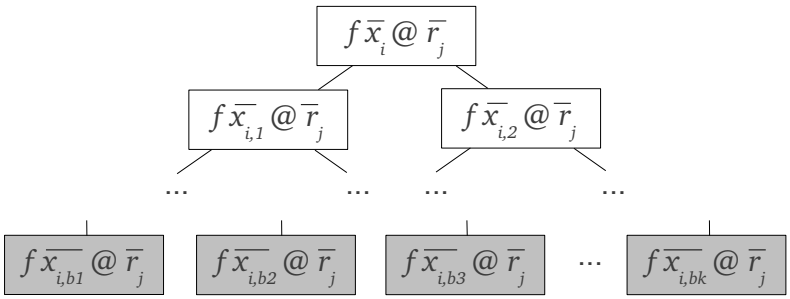


Figure 7.5: Activation tree corresponding to a function call $f\ \overline{x_i}\ @\ \overline{r_j}$. The grey nodes correspond to base calls, whereas the white ones correspond to recursive calls.

whereas seq_{aft} does not contain expressions of this form, and the guard of seq_{bef} and seq_{aft} is the same as that of seq. The uniqueness of this decomposition allows us to define the function $splitBA_f$ which, when applied to sequences, returns the pair (seq_{bef}, seq_{aft}) resulting from this decomposition. This definition can be trivially extended to sets of sequences.

Definition 7.37. Given a set of sequences S and a context function f, the function $splitBA_f$ is defined as follows:

$$splitBA_f\ S = \{splitBA_f\ seq \mid seq \in S\}$$

The $splitBA_f$ function returns a set of pairs. We can join the first components of all these pairs in order to get a single set of *before* fragments. Similarly, the second components of these pairs can also be joined. This gives place to the definition of $splitBA_f^*$ returning a pair of sets:

$$splitBA_f{}^*\ S = (\{seq_{bef} \mid (seq_{bef}, seq_{aft}) \in splitBA_f\ S\}, \{seq_{aft} \mid (seq_{bef}, seq_{aft}) \in splitBA_f\ S\})$$

Example 7.38. In our *append* running example (7.17), if $S = seqs\ e_{append}$ we obtain $splitExp_{append}\ S = (S_b, S_r)$ where:

$$\begin{array}{lcl} S_b & = & \{[xs \geq 1 \rightarrow ys]\} \\ S_r & = & \{[xs \geq 2 \rightarrow append\ xx\ ys, (x : x_1)@r]\} \end{array}$$

by applying $splitBA$ and $splitBA^*$ to the resulting S_r set, we get:

$$\begin{array}{lcl} splitBA_{append}\ S_r & = & \{([xs \geq 2 \rightarrow append\ xx\ ys], [xs \geq 2 \rightarrow (x : x_1)@r])\} \\ splitBA^*_{append}\ S_r & = & (\{[xs \geq 2 \rightarrow append\ xx\ ys]\}, \{[xs \geq 2 \rightarrow (x : x_1)@r]\}) \end{array}$$

□

7.5.3 Algorithm for computing Δ_0

In order to grasp the intuitive meaning of the algorithm, let us consider the activation tree corresponding to a function call $f\ \overline{x_i}\ @\ \overline{r_j}$ (Figure 7.5). We can abstract each node by its charges on memory as follows:

- Base nodes are abstracted by the result of $[\![S_b]\!]_\Delta$, where S_b contains the base sequences of the function's body. We use Δ_b to denote this abstract heap.

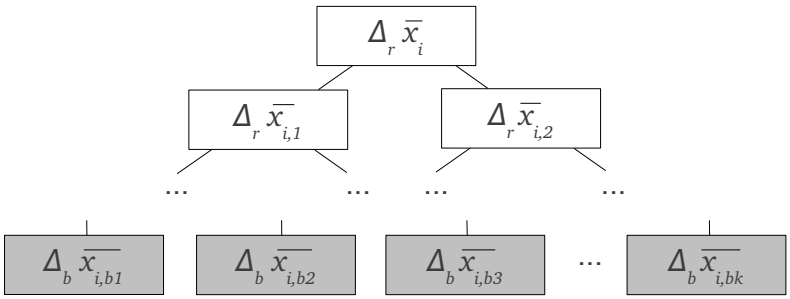


Figure 7.6: Abstraction of the charges done to each call in the activation tree.

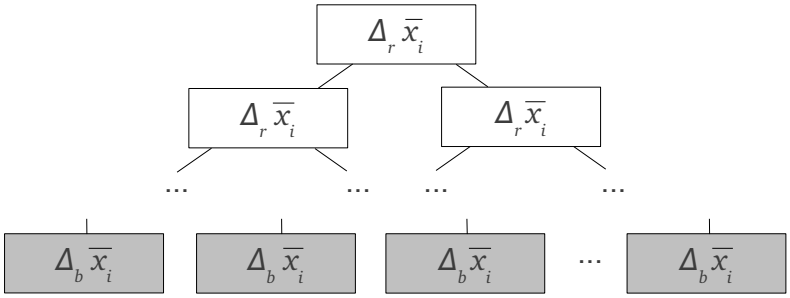


Figure 7.7: The substitution of $\overline{x_i}$ for the parameters in each recursive call leads to an abstraction that approximates the tree shown in Figure 7.6.

- Recursive nodes are abstracted by the result of $[\![S_r]\!]_\Delta$, where S_r contains the recursive sequences of the function's body, and without taking into account the recursive calls occurring in S_r. We use Δ_r to denote this abstract heap.

The result of this abstraction is depicted in Figure 7.6. In the internal nodes of the tree we get $\Delta_r\ \overline{x_{i,j}}$, where j depends on the particular recursive call. In the leafs we obtain $\Delta_b\ \overline{x_{i,bj}}$, where j depends on the particular base call. We can further approximate this abstraction by imposing some conditions on the Δ_b and Δ_r.

Definition 7.39. An abstract heap $\Delta \in \mathbb{D}$ (resp. a space cost function $\xi \in \mathbb{F}$) is said to be *parameter-decreasing* with respect to a function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$ iff, for every recursive call $f\ \overline{a_i}^n\ @\ \overline{r_j}^m$ occurring in e_f, it holds that

$$\Delta\ \overline{|a_i|\ \overline{x_i}} \sqsubseteq \Delta\ \overline{x_i}$$

or, respectively,

$$\xi\ \overline{|a_i|\ \overline{x_i}} \sqsubseteq \xi\ \overline{x_i}$$

Therefore, if we assume that our Δ_r and Δ_b are parameter-decreasing, we can substitute each $\Delta_r\ \overline{x_{i,j}}$ and $\Delta_b\ \overline{x_{i,bj}}$ by their counterparts $\Delta_r\ \overline{x_i}$ and $\Delta_b\ \overline{x_i}$. The new tree we obtain in this way (Figure 7.7) is an upper approximation to the previous one, but now all the recursive nodes charge the same cost, and so do the base nodes. If nb and nr respectively approximate the number of base and recursive nodes, our initial approximation is given by the function $\lfloor\Delta_b\rfloor * nb + \lfloor\Delta_r\rfloor * nr$.

The algorithm for computing the initial approximation Δ_0 is shown in Figure 7.8. The least upper bound with $\lfloor\Delta_b\rfloor$ handles those input sizes in which $\lfloor\Delta_r\rfloor$ becomes undefined.

$$
\begin{array}{lll}
\mathit{computeDelta}\ (f\ \overline{x_i}\ @\ \overline{r_j} = e_f)\ \Sigma\ \Gamma\ nb\ nr = (\lfloor \Delta_b \rfloor * nb + \lfloor \Delta_r \rfloor * nr) \sqcup \lfloor \Delta_b \rfloor & & \\
\quad \textbf{where} \quad S & = & \mathit{seqs}\ e_f \\
(S_b, S_r) & = & \mathit{splitExp}_f\ S \\
\Delta_b & = & [\![S_b]\!]_\Delta\ (\Sigma \uplus [f \mapsto ([\,]_f, 0, 0)])\ \Gamma \\
\Delta_r & = & [\![S_r]\!]_\Delta\ (\Sigma \uplus [f \mapsto ([\,]_f, 0, 0)])\ \Gamma
\end{array}
$$

Figure 7.8: Algorithm for computing Δ_0.

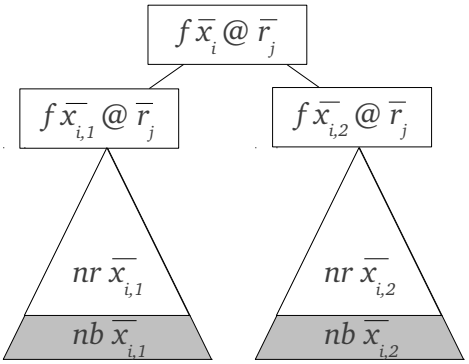


Figure 7.9: Activation tree of a call $f\ \overline{x_i}\ @\ \overline{r_j}$. The triangles represent the activation trees of each child call.

Example 7.40. If we apply *computeDelta* to our *append* example by considering the *nb* and *nr* functions of Example 7.28, we get $\Delta_b\ xs\ ys\ \rho_2 = [xs \geq 1 \to 0]$ and $\Delta_r\ xs\ ys\ \rho_2 = [xs \geq 2 \to 1]$. Therefore we get:

$$\Delta_0\ xs\ ys\ \rho_2 = [xs \geq 2 \to xs - 1] \sqcup [xs \geq 1 \to 0]$$

□

Our next step is to prove that an abstract heap Δ_0 computed by *computeDelta* falls into the reductive area of the lattice $(\mathbb{D}, \sqsubseteq)$ with respect to the iteration of the abstract interpretation $\mathcal{I}_f$. As noted in Section 7.5.1, this also ensures that the result of *computeDelta* is a correct approximation to the actual charges done by f. Reductivity of Δ_0 holds provided some admissibility conditions on the externally given *nb* and *nr* functions hold.

Definition 7.41 (Admissible *nb*)**.** A function *nb* for computing the number of base calls is admissible with respect to a definition $f\ \overline{x_i^n}\ @\ \overline{r_j^m} = e_f$ iff for every $\overline{x_i^n} \in \mathbb{R}^n$ the following conditions hold:

1. $nb\ \overline{x_i^n} \geq 1$
2. $\forall seq \in \mathit{seqs}\ e_f . \sum \left\{ nb\ \overline{|a_i|\ x_i}^n \mid f\ \overline{a_i^n}\ @\ \overline{r_j^m} \in seq \right\} \leq nb\ \overline{x_i^n}$

Definition 7.42 (Admissible *nr*)**.** A function *nr* for computing the number of recursive calls is admissible with respect to a definition $f\ \overline{x_i^n}\ @\ \overline{r_j^m} = e_f$ iff for every $\overline{x_i^n} \in \mathbb{R}^n$ the following conditions hold:

1. $nr\ \overline{x_i^n} \geq 0$
2. $\forall seq \in \mathit{seqs}\ e_f . 1 + \sum \left\{ nr\ \overline{|a_i|\ x_i}^n \mid f\ \overline{a_i^n}\ @\ \overline{r_j^m} \in seq \right\} \leq nr\ \overline{x_i^n}$

In order to get an intuitive idea on these conditions, let us consider the activation tree shown in Figure 7.9. The root of this tree is a call to f giving place to two recursive calls, each one with an

approximation of its number of leafs $nb\ \overline{x_{i,j}}$ and internal nodes $nr\ \overline{x_{i,j}}$, with $j = 1, 2$. Hence, the whole tree has $nb\ \overline{x_{i,1}} + nb\ \overline{x_{i,2}}$ leafs or less. Admissibility on nb holds if the approximation $nb\ \overline{x_i}$ given for the root node is greater or equal than this number. Analogously, nr is admissible if the approximation $nr\ \overline{x_i}$ of the root node is greater or equal than $1 + nr\ \overline{x_{i,1}} + nr\ \overline{x_{i,2}}$.

Given these conditions, we are ready to prove the reductivity of Δ_0:

Theorem 7.43. *Let us define $\Delta_0 = computeDelta\ (f\ \overline{x_i}\ @\ \overline{r_j} = e_f)\ \Sigma\ \Gamma\ nb\ nr$. If nr and nb are admissible, and the Δ_b and Δ_r occurring in the definition of computeDelta are parameter-decreasing, then:*

$$\lfloor [\![seq]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta_0, 0, 0)])\ \Gamma \rfloor \sqsubseteq \Delta_0 \qquad \text{for every } seq \in seqs\ e_f$$

Therefore, $\Delta_0 \in Red(\mathcal{I}_f)$

Proof. Let us denote by Σ' the signature environment $\Sigma \uplus [f \mapsto (\Delta_0, 0, 0)]$, and by Σ_0 the signature environment $\Sigma \uplus [f \mapsto ([\ ]_f, 0, 0)]$. It is easy to see that dom $\Sigma'(f) \subseteq$ dom $\Sigma_0(f)$. Assume $(S_b, S_r) = splitExp\ (seqs\ e_f)$. Then, $seqs\ e_f = S_b \uplus S_r$. Consider a sequence $seq \in seqs\ e_f$. We distinguish cases:

- **Case** $seq \in S_b$. That is, seq is a base sequence. By the definition of Δ_b we get $[\![seq]\!]_\Delta\ \Sigma_0\ \Gamma \sqsubseteq \Delta_b$. The signature associated with f is not relevant, since seq does not contain calls to f. Therefore we get:

$$\lfloor [\![seq]\!]_\Delta\ \Sigma'\ \Gamma \rfloor = \lfloor [\![seq]\!]_\Delta\ \Sigma_0\ \Gamma \rfloor \sqsubseteq \lfloor \Delta_b \rfloor \sqsubseteq \Delta_0$$

- **Case** $seq \in S_r$. That is, seq is a recursive sequence. Let us define $D =$ dom $([\![seq]\!]_\Delta\ \Sigma'\ \Gamma)$. In $\mathbb{R}^n \backslash D$ we get $\lfloor [\![seq]\!]_\Delta\ \Sigma'\ \Gamma \rfloor = \bot \sqsubseteq \Delta_0$ and the Theorem follows trivially. Thus, in the following, we consider the points belonging to D. In those points we get $\lfloor \Delta_r \rfloor \neq \bot$, since $seq \in S_r$ and hence $D \subseteq$ dom $[\![S_r]\!]_\Delta\ \Sigma'\ \Gamma \subseteq$ dom $[\![S_r]\!]_\Delta\ \Sigma_0\ \Gamma =$ dom Δ_r. We split the elements of seq into two sets: the A set contains the elements of the form $f\ \overline{a_i}^n\ @\ \overline{r_j}^m$, and the NA set contains the remaining ones. Let us denote by p the number of elements of A, so we get:

$$A = \{f\ \overline{a_{i,1}}^n\ @\ \overline{r_j}^m, f\ \overline{a_{i,2}}^n\ @\ \overline{r_j}^m, \ldots, f\ \overline{a_{i,p}}^n\ @\ \overline{r_j}^m\}$$

The list of region arguments $\overline{r_j}^m$ is the same as in the function definition of f, as we assume the absence of polymorphic recursion. Since $A \uplus NA$ contains every element of the sequence, we get the following equality in D:

$$\begin{aligned} \lfloor [\![seq]\!]_\Delta\ \Sigma'\ \Gamma \rfloor &= \sum_{be \in NA} \lfloor [\![be]\!]_\Delta\ \Sigma'\ \Gamma \rfloor + \sum_{be \in A} \lfloor [\![be]\!]_\Delta\ \Sigma'\ \Gamma \rfloor \\ &= \sum_{be \in NA} \lfloor [\![be]\!]_\Delta\ \Sigma'\ \Gamma \rfloor + \sum_{k=1}^{p} \lfloor [\![f\ \overline{a_{i,k}}^n\ @\ \overline{r_j}^m]\!]_\Delta\ \Sigma'\ \Gamma \rfloor \qquad \text{(in } D\text{)} \qquad (7.13) \end{aligned}$$

in which, for every $k \in \{1..p\}$

$$[\![f\ \overline{a_{i,k}}^n\ @\ \overline{r_j}^m]\!]_\Delta\ \Sigma'\ \Gamma = \lambda \overline{x_i}. \left[G_k \to \Delta_0\ \overline{|a_{i,k}|\ \overline{x_j}}^n \right] \qquad (7.14)$$

$$\text{where } G_k = \lambda \overline{x_i}. \left(\Delta_0\ \overline{|a_{i,k}|\ \overline{x_j}}^n \neq \bot \right)$$

since $unify(\Gamma, f, \overline{r_j}) = id_{R_f}$, due to the absence of polymorphic recursion. Moreover, G_k holds in those points belonging to dom $(\Delta_0\ \overline{|a_{i,k}|\ \overline{x_j}}^n)$ and only in those points. So, we can rewrite (7.14)

as follows:

$$[\![f\ \overline{a_{i,k}}^n\ @\ \overline{r_j}^m]\!]_\Delta\ \Sigma'\ \Gamma = \lambda \overline{x_i}.\Delta_0\ \overline{|a_{i,k}|\ \overline{x_j}}^n$$

Let us define, for every $k \in \{1..p\}$:

$$\begin{array}{rcl}
\Delta_{b,k} & \stackrel{\text{def}}{=} & \lambda \overline{x_i}^n.\Delta_b\ \overline{|a_{i,k}|\ \overline{x_j}^n}^n \\
\Delta_{r,k} & \stackrel{\text{def}}{=} & \lambda \overline{x_i}^n.\Delta_r\ \overline{|a_{i,k}|\ \overline{x_j}^n}^n \\
nb_k & \stackrel{\text{def}}{=} & \lambda \overline{x_i}^n.nb\ \overline{|a_{i,k}|\ \overline{x_j}^n}^n \\
nr_k & \stackrel{\text{def}}{=} & \lambda \overline{x_i}^n.nr\ \overline{|a_{i,k}|\ \overline{x_j}^n}^n
\end{array}$$

where Δ_b, Δ_r, nb and nr are as defined in *computeDelta*. We get for each $k \in \{1..p\}$:

$$[\![f\ \overline{a_{i,k}}^n\ @\ \overline{r_j}^m]\!]_\Delta\ \Sigma'\ \Gamma = \left(\lfloor\Delta_{b,k}\rfloor * nb_k + \lfloor\Delta_{r,k}\rfloor * nr_k\right) \sqcup \lfloor\Delta_{b,k}\rfloor \qquad (\text{in } D)$$

By substituting this into (7.13) we get:

$$\lfloor[\![seq]\!]_\Delta\ \Sigma'\ \Gamma\rfloor = \sum_{be \in NA} \lfloor[\![be]\!]_\Delta\ \Sigma'\ \Gamma\rfloor + \sum_{k=1}^{p} \left(\left(\lfloor\Delta_{b,k}\rfloor * nb_k + \lfloor\Delta_{r,k}\rfloor * nr_k\right) \sqcup \lfloor\Delta_{b,k}\rfloor\right) \qquad (\text{in } D) \tag{7.15}$$

For each $k \in \{1..p\}$ We denote by $\Delta_{f,k}$ the result of $\left(\lfloor\Delta_{b,k}\rfloor * nb_k + \lfloor\Delta_{r,k}\rfloor * nr_k\right) \sqcup \lfloor\Delta_{b,k}\rfloor$. We prove the following fact:

$$\forall k \in \{1..p\}.\Delta_{f,k} \sqsubseteq \lfloor\Delta_b\rfloor * nb_k + \lfloor\Delta_r\rfloor * nr_k \qquad (\text{in } D) \tag{7.16}$$

Let us consider a $k \in \{1..p\}$ and define $R_k \stackrel{\text{def}}{=} \{\overline{x_i} \in D \mid \lfloor\Delta_{b,k}\rfloor\ \overline{x_i} \geq \left(\lfloor\Delta_{b,k}\rfloor * nb_k + \lfloor\Delta_{r,k}\rfloor * nr_k\right)\ \overline{x_i}\}$. In R_k we obtain:

$$\Delta_{f,k} = \lfloor\Delta_{b,k}\rfloor \sqsubseteq \lfloor\Delta_b\rfloor \sqsubseteq \lfloor\Delta_b\rfloor * nb_k + \lfloor\Delta_r\rfloor * nr_k \qquad (\text{in } R_k)$$

The first $\sqsubseteq$ holds because Δ_b is parameter-decreasing, and the second $\sqsubseteq$ is due to the fact that $nb_k \sqsupseteq 1$, $nr_k \sqsupseteq 0$, and $\lfloor\Delta_r\rfloor \neq \bot$. In $D \backslash R_k$ we get:

$$\Delta_{f,k} = \lfloor\Delta_{b,k}\rfloor * nb_k + \lfloor\Delta_{r,k}\rfloor * nr_k \sqsubseteq \lfloor\Delta_b\rfloor * nb_k + \lfloor\Delta_r\rfloor * nr_k \qquad (\text{in } D \backslash R_k)$$

Again, this holds because both Δ_b and Δ_r are parameter-decreasing, so (7.16) holds for every $k \in \{1..p\}$. By substituting (7.16) into (7.15) we obtain:

$$\lfloor[\![seq]\!]_\Delta\ \Sigma'\ \Gamma\rfloor \sqsubseteq \sum_{be \in NA} \lfloor[\![be]\!]_\Delta\ \Sigma'\ \Gamma\rfloor + \sum_{k=1}^{p} \left(\lfloor\Delta_b\rfloor * nb_k + \lfloor\Delta_r\rfloor * nr_k\right) \qquad (\text{in } D)$$

It is easy to prove that

$$\Delta_r \sqsupseteq \sum_{be \in NA} [\![be]\!]_\Delta\ \Sigma_0\ \Gamma + \sum_{be \in A} [\![be]\!]_\Delta\ \Sigma_0\ \Gamma = \sum_{be \in NA} [\![be]\!]_\Delta\ \Sigma_0\ \Gamma + [\,]_f = \sum_{be \in NA} [\![be]\!]_\Delta\ \Sigma'\ \Gamma$$

in D. The first equality holds since $\Sigma_0(f)$ is defined for every input size, so the guard occurring in

the abstract interpretation rule for function application always holds. Hence:

$$\begin{aligned}\lfloor [\![seq]\!]_\Delta\ \Sigma'\ \Gamma \rfloor &\sqsubseteq \lfloor \Delta_r \rfloor + \sum_{k=1}^{p} (\lfloor \Delta_b \rfloor * nb_k + \lfloor \Delta_r \rfloor * nr_k) \\ &= \lfloor \Delta_r \rfloor + \lfloor \Delta_b \rfloor * \sum_{k=1}^{p} nb_k + \lfloor \Delta_r \rfloor * \sum_{k=1}^{p} nr_k \\ &= \lfloor \Delta_b \rfloor * \sum_{k=1}^{p} nb_k + \lfloor \Delta_r \rfloor * \left(1 + \sum_{k=1}^{p} nr_k\right) \qquad \text{(in } D\text{)}\end{aligned}$$

Finally, by admissibility of nb and nr:

$$\begin{aligned}\lfloor [\![seq]\!]_\Delta\ \Sigma'\ \Gamma \rfloor &\sqsubseteq \lfloor \Delta_b \rfloor * nb + \lfloor \Delta_r \rfloor * nr \\ &\sqsubseteq \Delta_0 \qquad \text{(in } D\text{)}\end{aligned}$$

□

This result not only proves the correctness of our initial approximation Δ_0. It also allows to iterate the abstract interpretation in order to reach more precise bounds by considering the following chain,

$$\Delta_0 \sqsupseteq \mathcal{I}_f(\Delta_0) \sqsupseteq \mathcal{I}_f^2(\Delta_0) \sqsupseteq \cdots \sqsupseteq \mathcal{I}_f^n(\Delta_0) \sqsupseteq \cdots$$

which, if eventually stabilizes, it does in a fixed point. Since the initial approximation to μ depends on the input Δ given (see next section), it is advisable to spent some time iterating $\mathcal{I}_f$ in order to achieve better results.

Example 7.44. Assume the Δ_0 of Example 7.40. By applying $\mathcal{I}_{append}$ on this abstract heap we obtain:

$$\mathcal{I}_{append}(\Delta_0)\ xs\ ys\ \rho_2 = [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1] \sqcup [xs \geq 3 \to xs - 1]$$

which is strictly smaller than Δ_0 when $xs \in [2, 3)$. Another iteration yields the following result:

$$\mathcal{I}^2_{append}(\Delta_0)\ xs\ ys\ \rho_2 = [xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1] \sqcup [xs \geq 3 \to 2] \sqcup [xs \geq 4 \to xs - 1]$$

In general, the i-th iteration results in the following abstract heap:

$$\mathcal{I}^i_{append}(\Delta_0) = \lambda xs\ ys\ \rho_2 \mapsto \begin{cases} \bot & xs < 1 \\ \lfloor xs - 1 \rfloor & 1 \leq xs < i + 2 \\ xs - 1 & i + 2 \leq xs \end{cases}$$

□

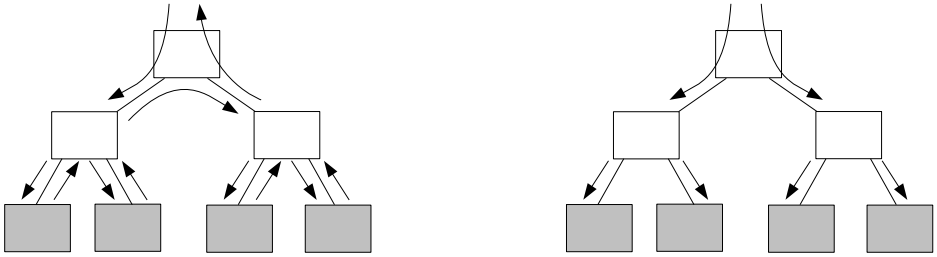
Notice that, in the previous example, the abstract heaps of every iteration are equal if we consider their domains only from a given threshold value. In some applications, it may be enough to obtain an expression $xs - 1$ as an upper-bound to the costs of *append*, even though it is not a fixed point of the corresponding iteration operator. This motivates the following definition:

Definition 7.45. Two abstract heaps $\Delta_1, \Delta_2 \in \mathbb{D}$ are said to be asymptotically equivalent (denoted $\Delta_1 \approx \Delta_2$) if there exists some $\overline{x_{i,0}}$ such that, for every $\overline{x_i}$ such that $x_i \geq x_{i,0}$ for all $i \in \{1..n\}$, $\Delta_1(\overline{x_i}) = \Delta_2(\overline{x_i})$. The definition of two cost functions $\xi_1, \xi_2 \in \mathbb{F}$ being asymptotically equivalent is analogous.

$$computeMu\ (f\ \overline{x_i}\ @\ \overline{r_j} = e_f)\ \Sigma\ \Gamma\ \Delta\ len = \left(\Delta_{self} * (len - 1) + |\Delta_{bef}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\}\right) \sqcup \mu_b$$

$$\begin{array}{lll}
\textbf{where} & S & = seqs\ e_f \\
& (S_b, S_r) & = splitExp_f\ S \\
& (S_{bef}, S_{aft}) & = splitBA_f^*\ S_r \\
& \Delta^*_{bef} & = [\![S_{bef}]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \\
& \Delta_{bef} & = \left\lfloor \Delta^*_{bef} \right\rfloor \\
& \Delta_{self} & = \Delta^*_{bef}\ \rho_{self} \\
& \mu_{bef} & = [\![S_{bef}]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \\
& \mu_{aft} & = [\![S_{aft}]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \\
& \mu_b & = [\![S_b]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma
\end{array}$$

Figure 7.10: Algorithm for computing μ_0.



(a) Regions different from the working region

(b) Working region

Figure 7.11: Growth of the charges done to different regions as the execution progresses.

In our example above, it holds that $\Delta_0 \approx \mathcal{I}_{append}(\Delta_0)$. In this case, we say that Δ_0 is an *asymptotic fixed point* of $\mathcal{I}_{append}$.

7.5.4 Algorithm for computing μ_0

The algorithm for computing a first approximation μ_0 to the heap needs of a function is shown in Figure 7.10. Let us consider the call tree of a given function call f. We make distinction between the charges done to the working regions of the calls in this tree, and the charges done to the remaining regions. The latter are cumulative, in the sense that the cells created in these regions are not removed while the execution of the root call to f progresses. The arrows in Figure 7.11a represent the directions of the execution flow, in which charges to these regions grow. With respect to the charges done to the working

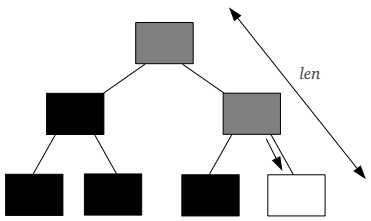


Figure 7.12: Execution point before the control flow reaches the last recursive call. Black nodes represent those calls whose execution is finished. Gray nodes represent those calls whose execution has started, but not finished.

regions of the calls of the activation tree, these only grow from the root call to the base cases, as Figure 7.11b shows. In this case we no longer have arrows pointing upwards in the tree, because all the cells created in the working region are removed when its corresponding function call finishes. Therefore, the only directions in which we know for sure that these charges grow, are the paths from the root call to its recursive children.

Now assume that, during this call to f, the execution flow has reached the point before executing the last base call (Figure 7.12). In the algorithm, $\Delta_{self} * (len - 1)$ represents the charges to the working regions of the gray-marked calls[3], whereas $|\Delta_{bef}|$ stands for the charges done to the remaining regions during the execution of the black- and gray-marked calls. The combination of these two charges gives us $\Delta_{self} * (len - 1) + |\Delta_{bef}|$. Taking this value as a base level, we have to take the following charges into account:

1. Maximum level of occupied memory before the execution flow reaches the last base call, on account of the charges done in memory before the first recursive call, and between the subsequent recursive calls (μ_{bef}).

2. Memory needs of the last base case, which is going to be executed (μ_b). This corresponds to the white-marked call of Figure 7.12.

3. Memory needs of the part of the recursive cases which is still to be executed (μ_{aft}). This corresponds to that part of the gray-marked cells whose execution is pending.

Since none of them is necessarily greater than the other, we take the least upper bound of the three. In the same way as in our Δ_0, we add μ_b in the least upper bound in order to deal with $\bot$ in the remaining components.

Since the $[\![\cdot]\!]_\mu$ interpretation depends on the $[\![\cdot]\!]_\Delta$ interpretation, the *computeMu* function receives an abstract heap Δ, which is bound to f in the signature environment Σ.

Example 7.46. The algorithm applied to *append* yields the following partial results:

$$\begin{aligned} |\Delta_{bef}| &= \lambda xs\ ys.[xs \geq 3 \to xs - 2] \sqcup [xs \geq 2 \to 0] \\ \Delta_{self} &= \lambda xs\ ys.[xs \geq 2 \to 0] \\ \mu_{bef} &= \lambda xs\ ys.[xs \geq 2 \to 0] \\ \mu_{aft} &= \lambda xs\ ys.[xs \geq 2 \to 1] \\ \mu_b &= \lambda xs\ ys.[xs \geq 1 \to 0] \end{aligned}$$

which result in the following initial bound μ_0:

$$\mu_0 = \lambda xs\ ys.[xs \geq 1 \to 0] \sqcup [xs \geq 2 \to 1] \sqcup [xs \geq 3 \to xs - 1]$$

By applying the $\mathcal{M}$ operator to this μ_0, we get an asymptotically equal upper bound. □

Again, the reductivity of the μ_0 computed in this way depends on some admissibility conditions of the *len* function, which bounds the length of longest call chain.

Definition 7.47 (Admissible *len*)**.** A function *len* for computing the maximal length of call chains is admissible with respect to a definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ iff for every $\overline{x_i}^n \in \mathbb{R}^n$ the following conditions hold:

[3] We assume the worst-case execution in which the longest call chain is the one who leads to the last base call.

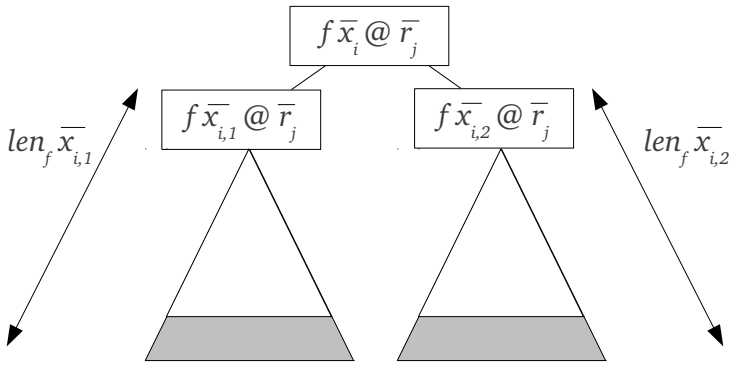


Figure 7.13: Activation tree of a call $f\ \overline{x_i}\ @\ \overline{r_j}$. The triangles represent the activation trees of each child call.

1. $len\ \overline{x_i}^n \geq 1$
2. $\forall seq \in seqs\ e_f . 1 + \bigsqcup \left\{ len\ \overline{|a_i|\ x_i}^n \mid f\ \overline{a_i}^n\ @\ \overline{r_j}^m \in seq \right\} \leq len\ \overline{x_i}^n$
3. If $(S_b, S_r) = splitExp_f\ (seqs\ e_f)$, for every $seq \in S_r$, $len\ \overline{x_i} \geq 2$ whenever $guard(seq)\ \overline{x_i}$ holds.

The first admissibility condition is fairly reasonable. The second one is analogous to its counterparts in *nb* and *nr*. Assume the situation given in Figure 7.13. An upper bound to the height of the whole tree is $1 + \sqcup\{len\ \overline{x_{i,1}}, len\ \overline{x_{i,2}}\}$. The second condition states that the approximated height $len\ \overline{x_i}$ must be greater or equal than this bound. Finally, the third condition states that, in those values $\overline{x_i}$ which may possibly lead to a recursive call, *len* must be greater than two (accounting for the caller and the callee).

Assuming these admissibility conditions, we can prove the reductivity of μ_0:

Theorem 7.48. *Let us define* $\mu_0 = computeMu\ (f\ \overline{x_i}\ @\ \overline{r_j} = e_f)\ \Sigma\ \Gamma\ \Delta\ len$. *If len is admissible, the* Δ_{self}, μ_{bef}, μ_{aft}, μ_b *occurring in computeMu are parameter-decreasing, and* $\lfloor [\![seq]\!]_\Delta\ (\Sigma \uplus [f \mapsto (\Delta, 0, 0)])\ \Gamma \rfloor \sqsubseteq \Delta$ *for every* $seq \in seqs\ e_f$, *then:*

$$[\![seq]\!]_\mu\ (\Sigma \uplus [f \mapsto (\Delta, \mu_0, 0)])\ \Gamma \sqsubseteq \mu_0 \qquad \text{for every } seq \in seqs\ e_f$$

Therefore, $\mu_0 \in Red(\mathcal{M}_{\Delta,f})$.

Proof. We denote by Σ' and Σ_0 the signature environments $\Sigma \uplus [f \mapsto (\Delta, \mu_0, 0)]$ and $\Sigma \uplus [f \mapsto (\Delta, 0, 0)]$ respectively. It is straightforward to prove that $\text{dom}\ \Sigma'(f) \subseteq \text{dom}\ \Sigma_0(f)$. Let $seq \in seqs\ e_f$. If S_b and S_r are as defined in *computeMu*, we know that $seq \in S_b \uplus S_r$, so we distinguish cases:

- **Case** $seq \in S_b$. In other words, *seq* is a base sequence. In this case $[\![seq]\!]_\mu\ \Sigma'\ \Gamma = [\![seq]\!]_\mu\ \Sigma_0\ \Gamma$, as *seq* does not contain calls to f. In this case,

  $$[\![seq]\!]_\mu\ \Sigma'\ \Gamma = [\![seq]\!]_\mu\ \Sigma_0\ \Gamma \sqsubseteq \mu_b \sqsubseteq \mu_0$$

  where μ_b is as defined in *computeMu*.

- **Case** $seq \in S_r$. That is, *seq* is recursive. We define $D = \text{dom}\ ([\![seq]\!]_\mu\ \Sigma'\ \Gamma)$. The Theorem follows trivially in those points $\overline{x_i}$ not belonging to D, so we assume $\overline{x_i} \in D$ in the following. Since our sequence is recursive, we can split it into two subsequences seq_{bef} and seq_{aft} such that

$seq_{bef} +\!\!+ seq_{aft} = seq$, $seq_{bef} \in S_{bef}$ and $seq_{aft} \in S_{aft}$. As a consequence of Lemma 7.20, we know that

$$\begin{array}{rcl} \mathrm{dom}\ (\llbracket seq \rrbracket_\mu\ \Sigma'\ \Gamma) & = & \mathrm{dom}\ (\llbracket seq_{bef} \rrbracket_\mu\ \Sigma'\ \Gamma) \cap \mathrm{dom}\ (\llbracket seq_{aft} \rrbracket_\mu\ \Sigma'\ \Gamma) \\ & \subseteq & \mathrm{dom}\ (\llbracket seq_{bef} \rrbracket_\mu\ \Sigma_0\ \Gamma) \cap \mathrm{dom}\ (\llbracket seq_{aft} \rrbracket_\mu\ \Sigma_0\ \Gamma) \end{array}$$

which implies $\mu_{bef}\ \overline{x_i} \neq \bot$ and $\mu_{aft}\ \overline{x_i} \neq \bot$ for every $\overline{x_i} \in D$. Moreover, and by Lemma 7.18, we get $\Delta^*_{bef}\ \overline{x_i} \neq \bot$ for all $\overline{x_i} \in D$, which implies $|\Delta_{bef}|\ \overline{x_i} \neq \bot$ and $|\Delta_{self}|\ \overline{x_i} \neq \bot$ for every $\overline{x_i} \in D$.

Let us start with the computation of $\llbracket seq \rrbracket_\mu\ \Sigma'\ \Gamma$. By Lemma 7.20:

$$\begin{array}{rcl} \llbracket seq \rrbracket_\mu\ \Sigma'\ \Gamma & = & \llbracket seq_{bef} +\!\!+ seq_{aft} \rrbracket_\mu\ \Sigma'\ \Gamma \\ & = & \sqcup\{\llbracket seq_{bef} \rrbracket_\mu\ \Sigma'\ \Gamma, \llbracket seq_{bef} \rrbracket_\Delta\ \Sigma'\ \Gamma + \llbracket seq_{aft} \rrbracket_\mu\ \Sigma'\ \Gamma\} \end{array}$$

Since the result of $\llbracket seq_{bef} \rrbracket_\Delta$ does not depend on the μ value given by $\Sigma'(f)$, and there are not calls to f in seq_{aft}, we can substitute Σ_0 for Σ' in the previous equation in order to get:

$$\begin{array}{rcl} \llbracket seq \rrbracket_\mu\ \Sigma'\ \Gamma & = & \sqcup\{\llbracket seq_{bef} \rrbracket_\mu\ \Sigma'\ \Gamma, \llbracket seq_{bef} \rrbracket_\Delta\ \Sigma_0\ \Gamma + \llbracket seq_{aft} \rrbracket_\mu\ \Sigma_0\ \Gamma\} \\ & \sqsubseteq & \sqcup\{\llbracket seq_{bef} \rrbracket_\mu\ \Sigma'\ \Gamma, |\Delta^*_{bef}| + \mu_{aft}\} \end{array}$$

The Theorem holds if we can prove that each element of the $\sqcup$ is subsumed by μ_0. Let us start with the second one. Since $len \sqsupseteq 2$ (in D) we get:

$$\begin{array}{rcll} |\Delta^*_{bef}| + \mu_{aft} & = & |\Delta_{bef}| + \Delta_{self} + \mu_{aft} & \\ & \sqsubseteq & |\Delta_{bef}| + \Delta_{self} * (len - 1) + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} & \\ & \sqsubseteq & \mu_0 & \text{(in } D\text{)} \end{array}$$

Now we prove that $\llbracket seq_{bef} \rrbracket_\mu\ \Sigma'\ \Gamma \sqsubseteq \mu_0$ (in D). By unfolding the definition of $\llbracket seq_{bef} \rrbracket_\mu\ \Sigma'\ \Gamma$, this inequality can we rewritten as follows:

$$\lfloor\cdot\rfloor^n_{k=1} \left(\textstyle\sum_{j=1}^{k-1} |\llbracket be_j \rrbracket_\Delta\ \Sigma'\ \Gamma| + \llbracket be_k \rrbracket_\mu\ \Sigma'\ \Gamma \right) \sqsubseteq \mu_0$$

where n is the number of elements in seq_{bef}. It is enough to prove that each element of the $\lfloor\cdot\rfloor$ is dominated by μ_0. We denote by ξ_k the k-th element of the $\lfloor\cdot\rfloor$. Let $k \in \{1..n\}$. We distinguish cases:

- be_k does not have the form $f\ \overline{a_{i,k}}\ @\ \overline{r_j}$.

  In this case $\llbracket be_k \rrbracket_\mu\ \Sigma'\ \Gamma = \llbracket be_k \rrbracket_\mu\ \Sigma_0\ \Gamma$. In addition, $\llbracket be_j \rrbracket_\Delta\ \Sigma'\ \Gamma = \llbracket be_j \rrbracket_\Delta\ \Sigma_0\ \Gamma$ for each $j \in \{1..k-1\}$. Hence we obtain (in D):

  $$\begin{array}{rcll} \xi_k & = & \displaystyle\sum_{j=1}^{k-1} (|\llbracket be_j \rrbracket_\Delta\ \Sigma_0\ \Gamma|) + \llbracket be_k \rrbracket_\mu\ \Sigma_0\ \Gamma & \\ & \sqsubseteq & \mu_{bef} & \\ & \sqsubseteq & \Delta_{self} * (len - 1) + |\Delta_{bef}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} & \\ & \sqsubseteq & \mu_0 & \text{(in } D\text{)} \end{array}$$

  since $len \sqsupseteq 1$ (in D) and $|\Delta_{bef}|$, Δ_{self} are distinct from $\bot$ in D.

- be_k has the form $f\ \overline{a_{i,k}}\ @\ \overline{r_j}$.

Let P be the set of $k \in \{1..n\}$ such that be_k is a function call to f. We define, for each $k \in P$:

$$\begin{array}{rcl}
\mu_{f,k} & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.\mu_0\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n \\
\Delta_{bef,k} & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.\Delta_{bef}\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n \\
\Delta_{self,k} & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.\Delta_{self}\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n \\
\Delta_{f,k} & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.\Delta_0\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n \\
\mu_{bef,k} & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.\mu_{bef}\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n \\
\mu_{aft,k} & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.\mu_{aft}\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n \\
\mu_{b,k} & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.\mu_b\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n \\
len_k & \stackrel{\text{def}}{=} & \lambda\overline{x_i}^n.len\ \overline{|a_{i,k}|\ \overline{x_i}^n}^n
\end{array}$$

We get (in D)

$$\begin{array}{rcl}
\xi_k & = & \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + \mu_{f,k} \\
& = & \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + \Big(\Big(\Delta_{self,k} * (len_k - 1) + |\Delta_{bef,k}| + \sqcup\{\mu_{bef,k}, \mu_{aft,k}, \mu_{b,k}\}\Big) \sqcup \mu_{b,k}\Big)
\end{array}$$

Since Δ_{self}, μ_{bef}, μ_{aft} and μ_b are parameter-decreasing we get:

$$\xi_k \sqsubseteq \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + \Big(\Big(\Delta_{self} * (len_k - 1) + |\Delta_{bef,k}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\}\Big) \sqcup \mu_b\Big)$$

Let R_k be the set of elements $\overline{x_i} \in D$ such that $(\Delta_{self} * (len_k - 1) + |\Delta_{bef,k}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\})\ \overline{x_i} \leq \mu_b\ \overline{x_i}$. We obtain, in R_k

$$\xi_k \sqsubseteq \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + \mu_b \qquad \text{(in } R_k\text{)} \tag{7.17}$$

Again, for each $j \in \{1..k-1\}$ it holds that $[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma = [\![be_j]\!]_\Delta\ \Sigma_0\ \Gamma$. Thus we get:

$$\sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| = \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma_0\ \Gamma| \sqsubseteq \sum_{j=1}^{n} |[\![be_j]\!]_\Delta\ \Sigma_0\ \Gamma| \sqsubseteq |\Delta^*_{bef}| \qquad \text{(in } D\text{)} \tag{7.18}$$

provided $([\![be_j]\!]_\Delta\ \Sigma_0\ \Gamma)\ \overline{x_i} \neq \bot$ for every $\overline{x_i} \in D$ and $j \in \{k..n\}$. However, if we had some $\overline{x_i} \in D$ such that $([\![be_j]\!]_\Delta\ \Sigma_0\ \Gamma)\ \overline{x_i} = \bot$ for some $j \in \{k..n\}$ we would have $([\![seq]\!]_\Delta\ \Sigma_0\ \Gamma)\ \overline{x_i} = \bot$ contradicting the fact that $\overline{x_i} \in D$ since $D = \text{dom}\ [\![seq]\!]_\mu\ \Sigma'\ \Gamma = \text{dom}\ [\![seq]\!]_\Delta\ \Sigma'\ \Gamma \subseteq \text{dom}\ [\![seq]\!]_\Delta\ \Sigma_0\ \Gamma$. Hence we can rewrite the inequality (7.17) above by using (7.18) as follows (in R_k):

$$\begin{array}{rcll}
\xi_k & \sqsubseteq & |\Delta^*_{bef}| + \mu_b & \\
& = & |\Delta_{bef}| + |\Delta_{self}| + \mu_b & \\
& \sqsubseteq & |\Delta_{bef}| + |\Delta_{self}| * (len - 1) + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} & \\
& \sqsubseteq & \mu_0 & \text{(in } R_k\text{)}
\end{array}$$

where the third step holds by admissibility of *len*. On the other hand we get, in $D \backslash R_k$:

$$\begin{aligned}
\xi_k &\sqsubseteq \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + \Delta_{self} * (len_k - 1) + |\Delta_{bef,k}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \\
&\sqsubseteq \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + \Delta_{self} * (\bigsqcup_{p \in P} len_p - 1) \\
&\qquad + |\Delta_{bef,k}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \quad (\text{in } D \backslash R_k)
\end{aligned} \tag{7.19}$$

Now we prove that $|\Delta_{bef,k}| \sqsubseteq |\Delta_{f,k}|$. Let us consider a sequence $seq' \in S_r$ of the form $[G \to be'_1, \ldots, be'_m]$. This sequence can be split into two subsequences seq'_{bef} and seq'_{aft}, each guarded by G, as done by $splitBA_f$. Notice that dom $[\![seq'_{bef}]\!]_\Delta\ \Sigma\ \Gamma \subseteq$ dom $[\![seq'_{aft}]\!]_\Delta\ \Sigma\ \Gamma$ for every Σ and Γ, as seq'_{aft} does not contain calls to f. Therefore:

$$\left\lfloor [\![seq'_{bef}]\!]_\Delta\ \Sigma_0\ \Gamma \right\rfloor \sqsubseteq \left\lfloor [\![seq'_{bef}]\!]_\Delta\ \Sigma_0\ \Gamma \right\rfloor + \left\lfloor [\![seq'_{aft}]\!]_\Delta\ \Sigma_0\ \Gamma \right\rfloor = \lfloor [\![seq']\!]_\Delta\ \Sigma_0\ \Gamma \rfloor = \lfloor [\![seq']\!]_\Delta\ \Sigma'\ \Gamma \rfloor \sqsubseteq \Delta$$

where the last step follows by assumption (reductivity of Δ). Since Δ_{bef} is the least upper bound of all the $\left\lfloor [\![seq'_{bef}]\!]_\Delta\ \Sigma_0\ \Gamma \right\rfloor$, then $\Delta_{bef} \sqsubseteq \Delta$, which implies $|\Delta_{bef}| \sqsubseteq |\Delta|$ and, in turn, $|\Delta_{bef,k}| \sqsubseteq |\Delta_{f,k}|$. We apply this into (7.19) so as to get:

$$\xi_k \sqsubseteq \sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + \Delta_{self} * (\bigsqcup_{p \in P} len_p - 1) + |\Delta_{f,k}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \qquad (\text{in } D \backslash R_k)$$

In a similar way as with the set R_k, in $D \backslash R_k$ it holds that

$$\sum_{j=1}^{k-1} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| + |\Delta_{f,k}| = \sum_{j=1}^{k} |[\![be_j]\!]_\Delta\ \Sigma'\ \Gamma| = \sum_{j=1}^{k} |[\![be_j]\!]_\Delta\ \Sigma_0\ \Gamma| \sqsubseteq \sum_{j=1}^{n} |[\![be_j]\!]_\Delta\ \Sigma_0\ \Gamma| = |\Delta^*_{bef}|$$

hence,

$$\begin{aligned}
\xi_k &\sqsubseteq |\Delta^*_{bef}| + \Delta_{self} * (\bigsqcup_{p \in P} len_p - 1) + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \\
&= |\Delta_{bef}| + \Delta_{self} + \Delta_{self} * (\bigsqcup_{p \in P} len_p - 1) + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \\
&= |\Delta_{bef}| + \Delta_{self} * (\bigsqcup_{p \in P} len_p) + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \\
&\sqsubseteq |\Delta_{bef}| + \Delta_{self} * (len - 1) + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \qquad (\text{in } D \backslash R_k)
\end{aligned}$$

where the last step follows by admissibility of *len*. Hence $\xi_k \sqsubseteq \mu_0$, which proves the theorem.

□

Notice that there is an additional constraint in the assumptions of this theorem: the Δ given as parameter must satisfy the conclusions of Theorem 7.43. This assumption holds, in particular, if Δ is either the initial approximation Δ_0 computed by the algorithm of Section 7.5.3, or the result of applying the abstract interpretation $n > 0$ times to it, $\mathcal{I}^n_f(\Delta_0)$.

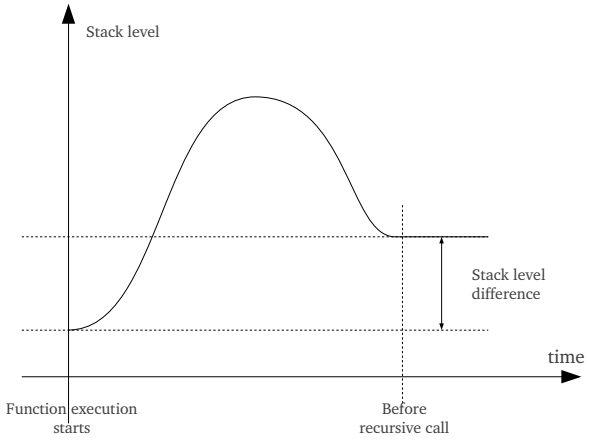


Figure 7.14: The stack level difference value denotes the maximum difference between the stack levels when the function starts executing and before its recursive call is done.

$$
\begin{array}{lcl}
SD_f\ (f\ \overline{a_i}^n\ @\ \overline{r_j}^m)\ td & = & n + m - td \\
SD_f\ be\ td & = & \bot \qquad \text{if } be \text{ is not a call to } f \\
SD_f\ (\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2)\ td & = & \sqcup\{2 + SD_f\ e_1\ 0, 1 + SD_f\ e_2\ (td+1)\} \\
SD_f\ (\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \rightarrow e_i}^{n})\ td & = & \sqcup_{r=1}^{n}(n_r + SD_f\ e_r\ (td + n_r))
\end{array}
$$

Figure 7.15: Definition of SD_f, which computes the stack level difference.

7.5.5 Algorithm for computing σ_0

In order to approximate the stack costs of a function we follow an approach similar to that of μ_0. In this case we do not have cumulative components such as the Δ_{bef} shown before, because the behaviour of the stack, in this sense, is analogous to that of the *self* region in the heap: it grows as the execution flow descends the activation tree. We use the term *stack level* to denote the number of words existing in the stacks at a given execution time. It is useful to obtain the maximum difference between the stack levels in two execution points: when the context function f starts its execution, and when a recursive call to f is going to be done (Figure 7.14). We use the term *stack level difference* to refer to this value, which is not expressed as an element of $\mathbb{F}$, but as an element of $\mathbb{N}^{\perp}$ instead. This is because this value does not depend on the input sizes as the rest of the components we have seen so far.

The *SD* function computes the stack level difference for a given expression in the context of a function definition. If this expression does not contain recursive calls, it returns $\bot$. If e_f is the body of the context function f, with n data parameters and m region parameters, the result of $SD_f\ e_f\ (n+m)$ is the stack level difference of this function. A formal proof of this fact is not necessary at this moment and it will deferred to Appendix B.

The σ_0 function computes an initial approximation to the stack consumption. The intuitive idea that, if we replicate the behaviour shown in Figure 7.14 along a number *len* of nested recursive calls, we get the graph shown in Figure 7.17. At the point in which the last base case is about to

$$
\begin{array}{l}
\mathit{computeSigma}\ (f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f)\ \Sigma\ td\ len = \sqcup\{0, SD_f\ e_f\ (n+m)\} * (len - 1) + \sigma \\
\qquad \textbf{where} \quad \sigma \quad = \quad [\![e_f]\!]_{\sigma}\ (\Sigma \uplus [f \mapsto ([\,]_f, 0, 0)])\ td
\end{array}
$$

Figure 7.16: Computation of an initial approximation σ_0.

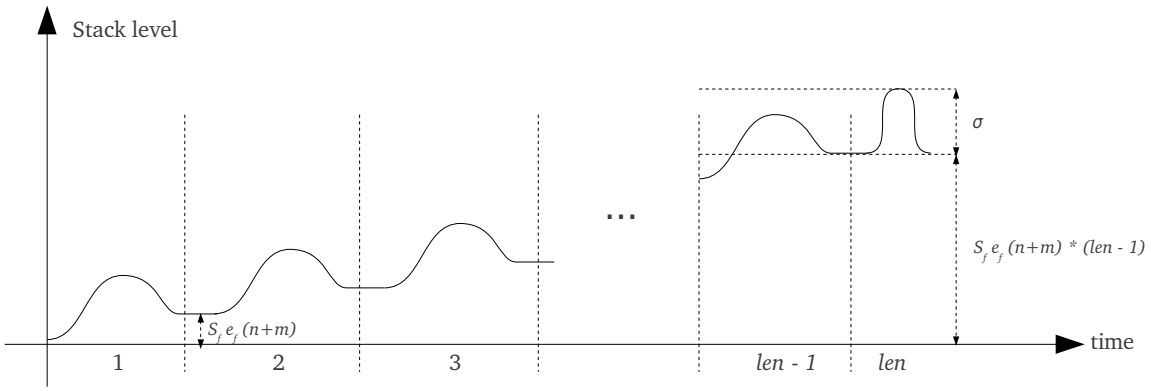


Figure 7.17: Stack level behaviour during the execution of subsequent recursive calls.

be executed, the stack level reaches $SD\ e_f\ (n+m) * (len-1)$ words. Then we have to proceed similarly as in the computation of μ_0: taking the value $SD\ e_f\ (n+m) * (len-1)$ as a base level, we should consider the stack costs on account of the base cases and the part of the recursive cases before and after the last recursive call. If we denote these components by σ_b, σ_{bef} and σ_{aft} respectively, we have to take the least upper bound $\sqcup\{\sigma_b, \sigma_{bef}, \sigma_{aft}\}$. However, and because the absence of cumulative components in the computation of stack needs (unlike the heap needs μ, in which the cumulative Δ component is involved), taking the least upper bound of these three components is equivalent to applying the $[\![\cdot]\!]_\sigma$-interpretation to the whole expression, without taking into account the costs of the subsequent recursive calls. That is what the σ component represents.

Example 7.49. The algorithm *computeSigma*, when applied to the *append* function, yields the following intermediate results,

$$SD_{append}\ (e_{append})\ 3 = 7 \qquad \sigma = [xs \geq 1 \to 1] \sqcup [xs \geq 2 \to 7]$$

leading to $\sigma_0 = [xs \geq 1 \to 7xs - 6] \sqcup [xs \geq 2 \to 7xs]$, which is an asymptotic fixed point. □

As in the previous cases, we have to prove the reductivity of the σ_0 obtained in this way. Before this, we need two auxiliary results: the first one states the minimum value that SD_f can return for a given function f.

Lemma 7.50. *Let $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ be a function definition. Then $SD_f\ e\ td \geq n + m - td$ for all td and every subexpression e of e_f in which there is a recursive call to f.*

Proof. By induction on the structure of e. If e is a recursive call to f the property holds trivially. We consider the remaining cases:

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

  If there is a call to f in e_2 then:

  $$SD_f\ e\ td \geq 1 + SD_f\ e_2\ (td+1) \geq 1 + n + m - (td+1) = n + m - td$$

  If there is no call to f in e_2, there must be at least one call in e_1 and hence:

  $$SD_f\ e\ td \geq 2 + SD_f\ e_1\ 0 \geq 2 + n + m - 0 \geq n + m - td$$

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

  Let us assume that there is a call to f in the r-th branch of the **case**. Then:

$$SD_f\ e\ td \geq n_r + SD_f\ e_r\ (td + n_r) \geq n_r + n + m - (td + n_r) = n + m - td$$

□

As a consequence of this Lemma, it holds that $SD_f\ e\ (n+m) \geq 0$ for every function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$. The second auxiliary lemma allows us to express the result of the $[\![\cdot]\!]_\sigma$-interpretation as the least upper bounds of the stack costs of the function itself (without taking recursive calls into account), and the stack costs of its recursive calls (taking the maximum stack level difference as a base level).

Lemma 7.51. *Let f be the context function, and e a subexpression of its body which contains p recursive calls to f. Let $\sigma' \in \mathbb{F}$ a stack cost function. For each recursive call $k \in \{1..p\}$ we denote by σ'_k the stack cost function $\lambda\overline{x_i}.\sigma'\ (\overline{|a_{ki}|\ \overline{x_i}})$, where $|a_{ki}|$ is the size function of the i-th argument ocurring in the k-th recursive call. If we define:*

$$\begin{array}{rcl} \sigma & = & [\![e]\!]_\sigma\ (\Sigma \uplus [f \mapsto ([\,]_f, 0, \sigma')])\ \Gamma\ td \\ \sigma_r & = & [\![e]\!]_\sigma\ (\Sigma \uplus [f \mapsto ([\,]_f, 0, 0)])\ \Gamma\ td \end{array}$$

then the following relation between σ, σ_r and σ' holds:

$$\sigma \sqsubseteq \sqcup\left(\{\sigma_r\} \cup \left\{SD_f\ e\ td + \sigma'_k \mid k \in \{1 \ldots p\}\right\}\right)$$

Proof. By induction on the structure of e.

- **Case** $e \equiv be$, where be is not a recursive call.

  In this case we get $p = 0$ and the signature of f is not relevant. Thus $\sigma = \sigma_r$, from which the required result follows trivially.

- **Case** $e \equiv f\ \overline{a_i}^n\ @\ \overline{r_j}^m$

  We get $p = 1, \sigma = \sqcup\{n+m, \sigma'_1 - td + n + m\}$ and $\sigma_r = \sqcup\{n+m, n+m-td\} = n+m$ and hence:

$$\sigma = \sqcup\{n+m, n+m-td+\sigma'_1\} = \sqcup\{\sigma_r, SD_f\ e\ td + \sigma'_1\} \sqsubseteq \sqcup\{\sigma_r, SD_f\ e\ td + \sigma'_1\}$$

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

  We assume there are $q \in \{0 \ldots p\}$ calls to f in e_1 and $q - p$ calls to f in e_2. If we denote by σ_1 and σ_2 the stack costs obtained in e_1 and e_2, we can apply the induction hypothesis as follows:

$$\begin{array}{rcl} \sigma_1 & \sqsubseteq & \sqcup\left(\{\sigma_{1,r}\} \cup \{SD_f\ e_1\ 0 + \sigma'_k \mid k \in \{1 \ldots q\}\}\right) \\ \sigma_2 & \sqsubseteq & \sqcup\left(\{\sigma_{2,r}\} \cup \{SD_f\ e_2\ (td+1) + \sigma'_k \mid k \in \{q+1 \ldots p\}\}\right) \end{array}$$

  Consider the set $D = \text{dom}\ \sigma_1 \cap \text{dom}\ \sigma_2$. If $\overline{x_i} \notin D$ then we get:

$$\sigma\ \overline{x_i} = \sqcup\{2 + \sigma_1\ \overline{x_i}, 1 + \sigma_2\ \overline{x_i}\} = \bot$$

and the lemma holds trivially. On the other hand, if $\overline{x_i} \in D$ then $\sigma_1\ \overline{x_i} \neq \bot$ and $\sigma_2\ \overline{x_i} \neq \bot$, which implies, by the definition of $\sigma_{1,r}$ and $\sigma_{2,r}$, $\sigma_{1,r}\ \overline{x_i} \neq \bot$ and $\sigma_{2,r}\ \overline{x_i} \neq \bot$. Therefore we get, in D:

$$
\begin{array}{ll}
 & \sigma \\
= & \quad \{ \text{ by definition of } [\![\cdot]\!]_\sigma \ \} \\
 & \sqcup\{2+\sigma_1, 1+\sigma_2\} \\
\sqsubseteq & \quad \{ \text{ by I.H. } \} \\
 & \sqcup\{2+\sqcup\left(\{\sigma_{1,r}\} \cup \{SD_f\ e_1\ 0+\sigma'_k \mid k \in \{1 \ldots q\}\}\right), \\
 & \quad 1+\sqcup\left(\{\sigma_{2,r}\} \cup \{SD_f\ e_2\ (td+1)+\sigma'_k \mid k \in \{q+1 \ldots p\}\}\right)\} \\
= & \quad \{ \text{ by properties of } \sqcup, \text{ and since } \sigma_1, \sigma_2 \neq \bot \text{ in } D \ \} \\
 & \sqcup\{\sqcup\{2+\sigma_{1,r}, 1+\sigma_{2,r}\}, \sqcup\{2+SD_f\ e_1\ 0+\sigma'_k \mid k \in \{1 \ldots q\}\}, \\
 & \quad \sqcup\{1+SD_f\ e_2\ (td+1)+\sigma'_k \mid k \in \{q+1 \ldots p\}\}\} \\
\sqsubseteq & \quad \{ \text{ by definition of } SD_f \text{ and } \sigma_r, \text{ and since } \sigma_{1,r}, \sigma_{2,r} \neq \bot \text{ in } D \ \} \\
 & \sqcup\{\sigma_r, \sqcup\{SD_f\ e\ td+\sigma'_k \mid k \in \{1 \ldots q\}\}, \\
 & \quad \sqcup\{SD_f\ e\ (td)+\sigma'_k \mid k \in \{q+1 \ldots p\}\}\} \\
= & \quad \{ \text{ by properties of } \sqcup \ \} \\
 & \sqcup\{\sigma_r, \sqcup\{SD_f\ e\ td+\sigma'_k \mid k \in \{1 \ldots p\}\}\} \\
= & \quad \{ \text{ by properties of } \sqcup \ \} \\
 & \sqcup\left(\{\sigma_r\} \cup \{SD_f\ e\ td+\sigma'_k \mid k \in \{1 \ldots p\}\}\right)
\end{array}
$$

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}$

  The p recursive calls are distributed among the e_i, so we assume a sequence $1 = p_0 \leq p_1 \leq \ldots \leq p_n = p$ such that there are $p_j - p_{j-1}$ recursive calls in the j-th branch. If we denote by σ_j ($j \in \{1 \ldots n\}$) the result of applying the abstract interpretation rules to the j-th branch, we get:

$$
\begin{array}{ll}
 & \sigma \\
= & \quad \{ \text{ by definition of } [\![\cdot]\!]_\sigma \ \} \\
 & \sqcup_{j=1}^{n}\{n_j+\sigma_j\} \\
\sqsubseteq & \quad \{ \text{ by I.H. } \} \\
 & \sqcup_{j=1}^{n}\{n_j+\sqcup\left(\{\sigma_{j,r}\} \cup \{SD_f\ e_j\ (td+n_j)+\sigma'_k \mid k \in \{p_{j-1} \ldots p_j\}\}\right)\} \\
= & \quad \{ \text{ by properties of } \sqcup \ \} \\
 & \sqcup\{\sqcup_{j=1}^{n}\{n_j+\sigma_{j,r}\}, \sqcup_{j=1}^{n}\{n_j+SD_f\ e_j\ (td+n_j)+\sigma'_k \mid k \in \{p_{j-1} \ldots p_j\}\}\} \\
\sqsubseteq & \quad \{ \text{ by defs. of } SD_f \text{ and } \sigma_r \ \} \\
 & \sqcup\{\sigma_r, \sqcup_{j=1}^{n}\{SD_f\ e\ td+\sigma'_k \mid k \in \{p_{j-1} \ldots p_j\}\}\}
\end{array}
$$

$$
\begin{array}{ll}
= & \quad \{ \text{ by properties of } \sqcup \ \} \\
 & \sqcup\left(\{\sigma_r\} \cup \{SD_f\ e\ td+\sigma'_k \mid k \in \{1 \ldots p\}\}\right)
\end{array}
$$

□

Given these two properties, we can prove the reductivity of σ_0 provided the *len* function involved in its computation is admissible (see Definition 7.47).

Theorem 7.52. *Let $\sigma_0 =$ computeSigma $(f\ \overline{x_i}\ @\ \overline{r_j} = e_f)\ \Sigma\ td\ len$. If len is admissible and the σ occurring in the definition of computeSigma is parameter-decreasing, then $\sigma_0 \in Red(\mathcal{S}_f)$.*

Proof. If f is not recursive, we get:

$$\mathcal{S}_f(\sigma_0) = [\![e]\!]_\sigma\ \Sigma \uplus [f \mapsto ([\,]_f, 0, \sigma)]\ \Gamma\ (n+m) = [\![e]\!]_\sigma\ \Sigma[f \mapsto ([\,]_f, 0, 0)]\ \Gamma\ (n+m) = \sigma \sqsubseteq \sigma_0$$

and reductivity of σ_0 holds. Now let us assume that there are $p \geq 1$ recursive calls in the function definition. For each $k \in \{1..p\}$ we define $\sigma_{f,k}$, len_k and σ_k as follows.

$$\sigma_{f,k} \stackrel{\text{def}}{=} \lambda \overline{x}^n.\sigma_0\ (\overline{|a_{ki}|\ x_i}) \qquad len_k \stackrel{\text{def}}{=} \lambda \overline{x}^n.len\ (\overline{|a_{ki}|\ x}) \qquad \sigma_k \stackrel{\text{def}}{=} \lambda \overline{x}^n.\sigma\ (\overline{|a_{ki}|\ x})$$

where the $|a_{ki}|$ are size functions previously inferred for each parameter in each recursive call. Then:

$$\begin{array}{ll}
 & \mathcal{S}_f(\sigma_0) \\
\sqsubseteq & \quad \{\text{ by Lemma 7.51 }\} \\
 & \sqcup\Big(\{\sigma\} \cup \{SD_f\ e_f\ (n+m) + \sigma_{f,k} \mid k \in \{1 \ldots p\}\}\Big) \\
= & \quad \{\text{ by Lemma 7.50, } SD_f\ e_f\ (n+m) \geq 0\} \\
 & \sqcup\Big(\{\sigma\} \cup \{\sqcup\{0, SD_f\ e_f\ (n+m)\} + \sigma_{f,k} \mid k \in \{1 \ldots p\}\}\Big) \\
= & \quad \{\text{ by definition of } \sigma_0\} \\
 & \sqcup\Big(\{\sigma\} \cup \{\sqcup\{0, SD_f\ e_f\ (n+m)\} + \sqcup\{0, SD_f\ e_f\ (n+m)\} * (len_k - 1) + \sigma_k \mid k \in \{1 \ldots p\}\}\Big) \\
= & \\
 & \sqcup\Big(\{\sigma\} \cup \{\sqcup\{0, SD_f\ e_f\ (n+m)\} * len_k + \sigma_k \mid k \in \{1 \ldots p\}\}\Big) \\
\sqsubseteq & \quad \{\text{ admissibility of } len\} \\
 & \sqcup\Big(\{\sigma\} \cup \{\sqcup\{0, SD_f\ e_f\ (n+m)\} * (len - 1) + \sigma_k \mid k \in \{1 \ldots p\}\}\Big) \\
\sqsubseteq & \quad \{\text{ since } \sigma_k \sqsubseteq \sigma \text{ for each } k \in \{1 \ldots p\}\} \\
 & \sqcup\Big(\{\sigma\} \cup \{\sqcup\{0, SD_f\ e_f\ (n+m)\} * (len - 1) + \sigma \mid k \in \{1 \ldots p\}\}\Big) \\
= & \quad \{\text{ by properties of } \sqcup, \text{ and because } len \sqsupseteq 1\} \\
 & \sqcup\{0, SD_f\ e_f\ (n+m)\} * (len - 1) + \sigma \\
= & \quad \{\text{ by definition of } \sigma_0\} \\
 & \sigma_0
\end{array}$$

□

7.5.6 Correctness in absence of admissibility conditions

Let us summarize what we have achieved so far: given a function definition f, we have defined three algorithms for computing an initial signature $(\Delta_0, \mu_0, \sigma_0)$, which is a correct approximation of the memory needs of f. Moreover, if we insert this signature into a signature environment Σ and apply the abstract interpretation functions with this updated environment, we get a triple $(\Delta_1, \mu_1, \sigma_1)$, whose components can be equal or more precise than the initial $(\Delta_0, \mu_0, \sigma_0)$. Hence, the initial signature is reductive w.r.t. the abstract interpretation. In case this new signature is strictly more precise than the original one, we can apply the abstract interpretation again and get another triple $(\Delta_2, \mu_2, \sigma_2)$ which can be equal or more precise than $(\Delta_1, \mu_1, \sigma_1)$, and so on.

A key point to notice is that we have only shown the reductivity property of the initial signature, but not (directly) its correctness, since the latter follows from the reductivity property (Corollary 7.34). However, reductivity only holds under some admissibility conditions on the externally given nb, nr,

len functions. If one of these admissibility conditions does not hold, the initial signature may not be reductive. In this case we cannot apply the implication **[RD]⇒[CR]**, since the antecedent does not hold. However, it can be proven that, even in those cases in which these admissibility conditions do not hold, the initial signature $(\Delta_0, \mu_0, \sigma_0)$ is still correct. The proof of this fact is rather technical, and its details are left to Appendix B.

Given the above, let us assume that $(\Delta_0, \mu_0, \sigma_0)$ is correct, but not reductive. If we define, $\Delta_1 = \mathcal{I}_f(\Delta_0)$, $\mu_1 = \mathcal{M}_{\Delta_0,f}(\mu_0)$ and $\sigma_1 = \mathcal{S}_f(\sigma_0)$, we know, by Theorem 7.26, that the new signature $(\Delta_1, \mu_1, \sigma_1)$ is correct, but not necessarily more precise than $(\Delta_0, \mu_0, \sigma_0)$. If $\Delta_1 \sqsupset \Delta_0$ then we can safely discard Δ_1, since our initial approximation Δ_0 is more precise. It could also be the case that Δ_1 and Δ_0 are not comparable, in which case we compute $\Delta_1' = \sqcap\{\Delta_1, \Delta_0\}$, since it is more precise than both Δ_1 and Δ_0, but still correct. The same reasoning applies to the μ_0 and σ_0 components.

Therefore, in absence of the reductivity property we can define the following modified iteration operators as follows:

$$\begin{aligned} \mathcal{I}_f'(\Delta) &= \sqcap\{\Delta, \mathcal{I}_f(\Delta)\} \\ \mathcal{M}_{\Delta,f}'(\mu) &= \sqcap\{\mu, \mathcal{M}_{\Delta,f}(\mu)\} \\ \mathcal{S}_f'(\sigma) &= \sqcap\{\sigma, \mathcal{S}_f(\sigma)\} \end{aligned}$$

for every $\Delta \in \mathbb{D}$, $\mu \in \mathbb{F}$ and $\sigma \in \mathbb{F}$. It is easy to show that, if $(\Delta_0, \mu_0, \sigma_0)$ is our initial signature,

$$\Delta_0 \in \mathit{Red}(\mathcal{I}_f') \quad \wedge \quad \mu_0 \in \mathit{Red}(\mathcal{M}_{\Delta_0,f}') \quad \wedge \quad \sigma_0 \in \mathit{Red}(\mathcal{S}_f')$$

and, for every $n \geq 0$, $((\mathcal{I}_f')^n(\Delta_0), (\mathcal{M}_{\Delta_0,f}')^n(\mu_0), (\mathcal{S}_f')^n(\sigma_0))$ is a correct signature.

7.5.7 Correctness in absence of parameter-decrease conditions.

In addition to the above mentioned admissibility conditions, the reductivity of the initial bounds also depend on the fact that their components are parameter-decreasing, as stated in Definition 7.39. In absence of these conditions, we can ensure neither the reductivity nor the correctness of these initial bounds. Therefore, it is useful to establish some sufficient conditions under which the parameter-decrease property is guaranteed. Given a function definition, the first condition states that if the sizes of the parameters do not increase from the root call to the recursive ones, every abstract heap or space cost function is parameter-decreasing with respect to that definition.

Proposition 7.53. *Assume a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$. If for every recursive call $f\ \overline{a_i}^n\ @\ \overline{r_j}^m$ in its body it holds that $|a_k|\ \overline{x_i}^n \sqsubseteq x_k$ for every $k \in \{1..n\}$, then every $\Delta \in \mathbb{D}$ and $\xi \in \mathbb{F}$ is parameter-decreasing with respect to f.*

Proof. It follows trivially from the fact that the elements of $\mathbb{D}$ and $\mathbb{F}$ are monotone functions. □

It is not usual to obtain function definitions for which this condition does not hold. In particular, those functions having an accumulator parameter whose size increases from the root call to its recursive calls (such as *revAuxD*, in Example 5.2) do not satisfy this condition. Notice, however, that in these kind of functions, the costs do not depend on the sizes of these accumulator functions. Hence we can set out the following weaker sufficient condition:

Proposition 7.54. *Assume a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$. If for every recursive call $f\ \overline{a_i}^n\ @\ \overline{r_j}^m$ in its body it holds that $|a_k|\ \overline{x_i}^n \sqsubseteq x_k$ for every $k \in P \subseteq \{1..n\}$. For every $\Delta \in \mathbb{D}$ (resp. $\xi \in \mathbb{F}$), if Δ (resp. ξ) does not*

depend on the parameters belonging to $\{1..n\}\backslash P$*, then* Δ *(resp.* ξ*) is parameter-decreasing with respect to* f*.*

Proof. It follows trivially from monotonicity of Δ and ξ. □

This criterion is satisfied by every example function definition in this thesis. In absence of the parameter-decrease property, we can still adapt our *computeDelta*, *computeMu* and *computeSigma* algorithms so as to get correct initial approximations. We can follow an approach similar to that of [4]: the key idea is to compute an invariant Ψ bounding the feasible sizes of the parameters as a function of the arguments given to the root call, and then maximize the partial components appearing in each algorithm (for instance, Δ_b and Δ_r in *computeDelta*). If the invariant Ψ is given by a set of linear constraints, we can use linear programming techniques for this maximization. This adaptation is subject of future work.

7.6 Case studies

In this section we apply our space analysis to several examples. Some of the functions have already been introduced in previous chapters. Since, at this moment, our algorithm does not deal with explicit destruction, **case**! expressions are handled by our implementation as non-destructive **case** expressions. The upper bounds to the heap costs of these non-destructive versions are correct approximations to the costs of their destructive counterparts, whereas stack costs are not affected by explicit destruction.

All the algorithms explained in this chapter are currently implemented in *Maple*. The compiler's front-end generates a representation of the abstract syntax tree corresponding to the program being analysed, and *Maple* computes the initial symbolic approximations and perform the necessary simplifications. This system is also used for computing the asymptotic expressions to the obtained bounds.

In order to avoid excessive subscripting, we shall use $\mathcal{I}$, $\mathcal{M}$ and $\mathcal{S}$ to denote the iteration operators, without specifying the name of the function being inferred. The latter can be deduced from the context.

Example 7.55 (Insertion sort)**.** We will compute a bound to the memory needs of *insertD* and *inssortD* from Example 3.12 on page 89. Let us start with the *insertD* function. We use x and ys to denote its input parameters. We start from the following information regarding its activation tree,

$$\begin{aligned} nb &= \lambda x\ ys.1 \\ nr &= \lambda x\ ys.ys - 1 \\ len &= \lambda x\ ys.ys \end{aligned}$$

and the following size information,

$$|yy| = \lambda x\ ys.ys - 1$$

assuming yy is the name of the variable with which the recursive call to *insertD* is done. It is easy to see that the admissibility conditions on *nb*, *nr* and *len* hold, as well as the parameter-decreasingness conditions.

The *computeDelta* algorithm returns the abstract heap $\Delta_0 = \lambda x\ ys.[\rho_1 \mapsto \xi(x, ys)]$, where ξ is defined as follows:

$$\xi(x, ys) = \sqcup \left\{ \begin{array}{l} \overbrace{(ys-1)}^{n_r} * \overbrace{[ys \geq 2 \to 1]}^{\Delta_r} + \overbrace{1}^{n_b} * \overbrace{\sqcup\{[ys \geq 1 \to 2], [ys \geq 2 \to 0], [ys \geq 2 \to 2]\}}^{\Delta_b}, \\ \underbrace{\sqcup\{[ys \geq 1 \to 2], [ys \geq 2 \to 0], [ys \geq 2 \to 2]\}}_{\Delta_b} \end{array} \right\}$$

This result can be converted into a piecewise function, so as to obtain:

$$\Delta_0 = \lambda x\ ys. \left[\rho_1 \mapsto \begin{cases} \bot & ys < 1 \\ 2 & 1 \leq ys < 2 \\ ys + 1 & 2 \leq ys \end{cases} \right]$$

An iteration of the abstract interpretation leads to the following result,

$$\Delta_1 \stackrel{\text{def}}{=} \mathcal{I}(\Delta_0) = \lambda x\ ys. \left[\rho_1 \mapsto \begin{cases} \bot & ys < 1 \\ 2 & 1 \leq ys < 2 \\ 3 & 2 \leq ys < 3 \\ ys + 1 & 3 \leq ys \end{cases} \right]$$

which is equal to Δ_0, except in those points belonging to the interval $[2, 3)$, in which the latter result is more precise. However, Δ_0 is an asymptotic fixed point. If we use this abstract heap as an input to the *computeMu* function we get the following initial approximation:

$$\mu_0 = \lambda x\ ys. \begin{cases} \bot & ys < 1 \\ 2 & 1 \leq ys < 2 \\ 4 & 2 \leq ys < 3 \\ ys + 2 & 3 \leq ys \end{cases}$$

By applying the iterator $\mathcal{M}_{\Delta_0}$ to this result, we obtain,

$$\mu_1 \stackrel{\text{def}}{=} \mathcal{M}_{\Delta_0}(\mu_0) = \lambda x\ ys. \begin{cases} \bot & ys < 1 \\ 2 & 1 \leq ys < 2 \\ 3 & 2 \leq ys < 3 \\ 4 & 3 \leq ys < 4 \\ ys + 1 & 4 \leq ys \end{cases}$$

which is strictly lower than μ_0. A second iteration shows that μ_1 is an asymptotic fixed point of $\mathcal{M}_{\Delta_0}$:

$$\mu_2 \stackrel{\text{def}}{=} \mathcal{M}^2_{\Delta_0}(\mu_0) = \lambda x\ ys. \begin{cases} \bot & ys < 1 \\ 2 & 1 \leq ys < 2 \\ 3 & 2 \leq ys < 3 \\ 4 & 3 \leq ys < 4 \\ 5 & 4 \leq ys < 5 \\ ys + 1 & 5 \leq ys \end{cases}$$

With respect to the stack costs, we obtain

$$\sigma_0 = \lambda x\ ys. \begin{cases} \bot & ys < 1 \\ 9ys - 5 & 1 \leq ys < 2 \\ 9ys & 2 \leq ys \end{cases}$$

which is an asymptotic fixed point, since an application of the iterator $\mathcal{S}$ leads to the following function:

$$\sigma_1 \stackrel{\text{def}}{=} \mathcal{S}(\sigma_0) = \begin{cases} \bot & ys < 1 \\ 9ys - 5 & 1 \leq ys < 3 \\ 9ys & 3 \leq ys \end{cases}$$

Notice that the iteration of $\mathcal{S}$ never reaches its actual fixed point, which is $\lambda x\ ys.[ys \geq 1 \rightarrow \lfloor 9ys - 5 \rfloor]$.

With respect to the *inssortD* definition, and given the following functions:

$$\begin{aligned} nb &= \lambda xs.1 \\ nr &= \lambda xs.xs - 1 \\ len &= \lambda xs.xs \end{aligned}$$

The abstract heap Δ_0 returned by *computeDelta* is the following:

$$\Delta_0 \approx \lambda xs.[\rho_2 \mapsto xs^2 - xs + 1]$$

The successive iterations of the abstract interpretation lead to the following results,

$$\begin{aligned} \Delta_1 = \mathcal{I}(\Delta_0) &\approx \lambda xs.[\rho_2 \mapsto xs^2 - 2xs + 3] \\ \Delta_2 = \mathcal{I}^2(\Delta_0) &\approx \lambda xs.[\rho_2 \mapsto xs^2 - 3xs + 6] \\ \Delta_3 = \mathcal{I}^3(\Delta_0) &\approx \lambda xs.[\rho_2 \mapsto xs^2 - 4xs + 10] \\ \Delta_4 = \mathcal{I}^4(\Delta_0) &\approx \lambda xs.[\rho_2 \mapsto xs^2 - 5xs + 15] \end{aligned}$$

which are depicted in Figure 7.18. None of them is an asymptotic fixed point.

Assume we give Δ_4 as an input to the *computeMu* algorithm. We get,

$$\mu_0 \approx \lambda xs.xs^2 - 6xs + 21$$

from which, by iterating with $\mathcal{M}_{\Delta_4}$ we obtain an asymptotically equivalent bound μ_1. Finally, an initial

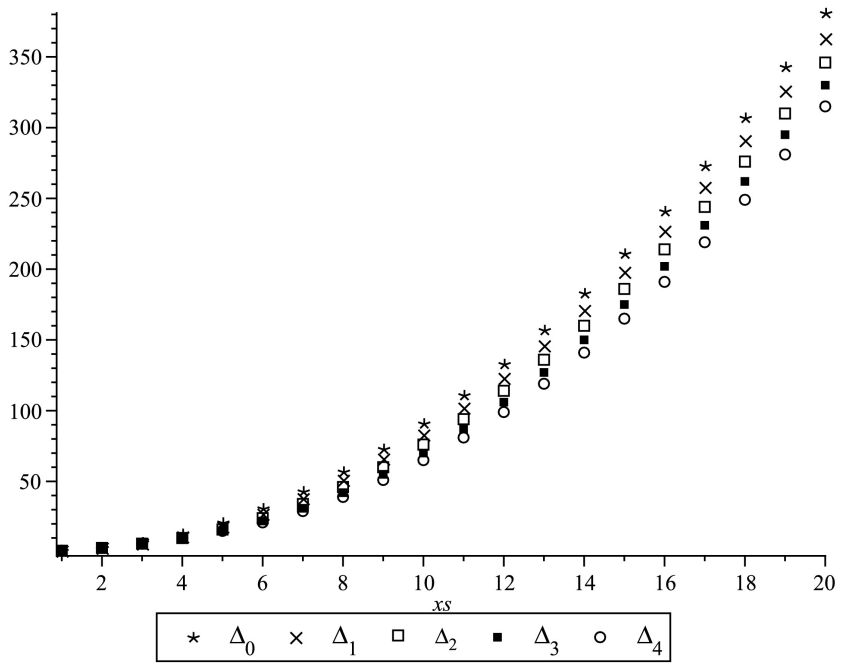


Figure 7.18: Graphical representation of Δ_i *xs* ρ_2 resulting from the i-th iteration.

approximation to the stack costs is given by the following expression:

$$\sigma_0 \approx \lambda xs.15xs - 14$$

and the successive iterations of $\mathcal{S}$ give place to the following functions.

$$\sigma_1 = \mathcal{S}(\sigma_0) \approx \lambda xs.15xs - 23$$
$$\sigma_2 = \mathcal{S}(\sigma_1) \approx \lambda xs.15xs - 32$$

none of which is an asymptotic fixed point. They are shown in Figure 7.19. □

Example 7.56. In Figures 7.20, 7.21 and 7.22 we show the results of our memory consumption analysis for most of the examples appearing in this thesis. We omit the λ-prefixes for a better readability. The values Δ_0, μ_0 and σ_0 represent the intial upper bounds obtained by the *computeDelta*, *computeMu* and *computeSigma* functions. In case these bounds are not fixed points of their corresponding $\mathcal{I}$, $\mathcal{M}$ and $\mathcal{S}$, we represent the results of successive applications of these operators. When $i > 0$, Δ_i denotes the abstract heap $\mathcal{I}^i(\Delta_0)$. The same applies with the μ_i and σ_i. A $(\star)$ mark besides an abstract heap or cost function indicates that it is an asymptotic fixed point of its corresponding operator.

Some functions have been slightly adapted to meet the requirements of the analysis. In particular, polymorphic recursion on regions has been disabled. That is why, in *pascal* function we obtain a quadratic cost function *pascal* instead of linear, as stated in Example 4.20. With regard to the *fib* function, we use a variant of type $Int \to HInt$, instead of the function appearing in Example 4.2, of type $HInt \to HInt$. This is done in order to accommodate the input values to the size model of our analysis. A value of type *HInt* always has size one, whereas the size of an non-negative *Int* is its value.

In some cases, we would get an infinite descending chain $\mu_0 \sqsupset \mu_1 \sqsupset \ldots \sqsupset \mu_i \sqsupset \ldots$ of functions, as in the μ component of *appendC*. However, this does not mean that the limit of this chain when i tends to $+\infty$ is the zero-constant function, since the results shown here are only asymptotic bounds, and the

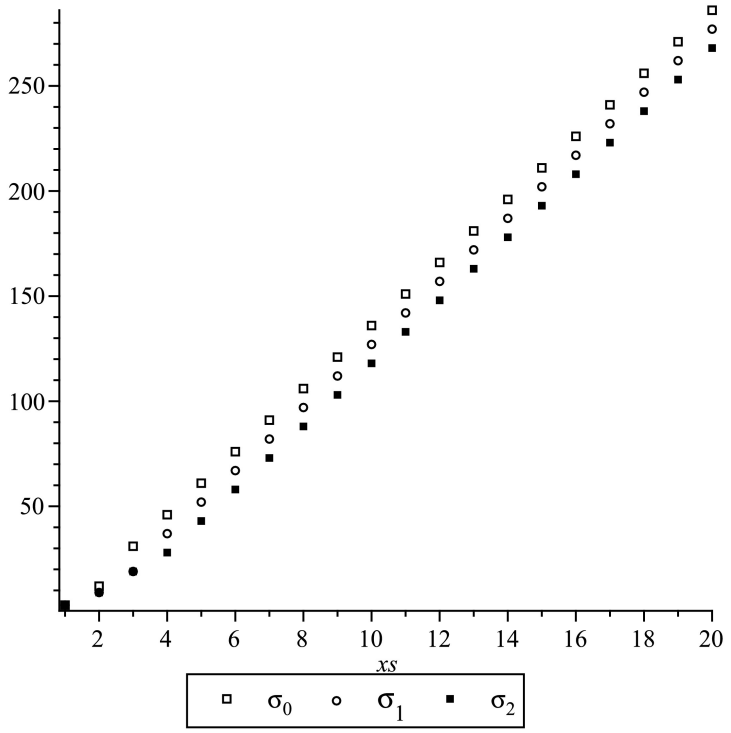


Figure 7.19: Graphical representation of $\sigma_i\ xs$ resulting from the i-th iteration.

limit does not have to coincide asymptotically with the μ_i. In the case of *appendC*, the sequence $\{\mu_i\}_{i\in\mathbb{N}}$ converges pointwise to $\lambda xs\, ys.xs + ys - 1$, which is a fixed point of the $\mathcal{M}$ operator. With the stack costs of *qsort* we get a similar situation.

Normally, the initial bounds are overapproximations of the actual fixed points. In the case of *msort*, whose worst-case space complexity is in $\mathcal{O}(xs \log\ xs)$, we get a quadratic bound. Notice, however, that the xs^2 coefficient keeps decreasing at each iteration.

The *reverse′* function of Figure 7.22 implements a naive algorithm for reversing the elements of a list.

$$\begin{array}{lcl} reverse'\ [\,] & = & [\,] \\ reverse'\ (x : xs) & = & append\ (reverse'\ xs)\ [x] \end{array}$$

This algorithm has quadratic heap space complexity, in contrast to the *revAux* function, of linear heap space complexity. These differences become apparent in the results shown in Figure 7.22. □

Recall from Section 2.8 that the SVM allows us to obtain constant stack costs for tail-recursive functions. Our algorithm *computeSigma* is aware of this, since it can be proven that for every tail-recursive function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$ the result of $SD_f\ e_f\ (n+m)$ is always zero. As a consequence, we get $\sigma_0 = \sigma$, where σ is defined as in *computeSigma*. If the latter does not depend on the input sizes, neither does σ_0.

Example 7.57 (Tail recursive selection sort)**.** Let us consider the implementation of the *selection sort* algorithm shown in Figure 7.23, where every function builds its result in an accumulator parameter. As a consequence, all the functions are tail recursive, and their stack costs are constant (i.e. do not depend on the size of the input). Besides this, the stack upper bounds inferred by *computeSigma* are also

append xs ys @ r (pg. 22) with $\Gamma(r) = \rho_2$

$\Delta_0 \approx [\rho_2 \mapsto xs - 1]^{(\star)}$ $\quad \mu_0 \approx xs - 1^{(\star)}$ $\quad \sigma_0 \approx 7xs^{(\star)}$

---

appendC xs ys @ r (pg. 24) with $\Gamma(r) = \rho_3$

$\Delta_0 \approx [\rho_3 \mapsto xs + ys - 1]^{(\star)}$ $\quad \mu_0 \approx 2ys + xs - 2$ $\quad \sigma_0 \approx 7xs^{(\star)}$

$\mu_1 \approx 2ys + xs - 3$

$\mu_2 \approx 2ys + xs - 4$

---

length xs (pg. 19)

$\Delta_0 \approx [\,]^{(\star)}$ $\quad \mu_0 \approx 0$ $\quad \sigma_0 \approx 5xs^{(\star)}$

---

split n xs @ r₁ r₂ r₃ (pg. 169) with $\Gamma(r_1) = \rho_1, \Gamma_2(r_2) = \rho_2, \Gamma_3(r_3) = \rho_3$

$$\Delta_0 \approx \begin{bmatrix} \rho_1 \mapsto 0 \\ \rho_2 \mapsto \min(n, xs+1) + 1 \\ \rho_3 \mapsto \min(n, xs+1) + 1 \end{bmatrix}^{(\star)}$$

$\mu_0 \approx 6 + 2\min(xs, n-1)$ $\quad \sigma_0 \approx 9\min(n+1, xs+2) + 1^{(\star)}$

$\mu_1 \approx 5 + 2\min(xs, n-1)^{(\star)}$

---

merge xs ys @ r (pg. 169) with $\Gamma(r) = \rho_1$

$\Delta_0 \approx \quad [\rho_1 \mapsto 2xs + 2ys - 3]$ $\quad \mu_0 \approx 2xs + 2ys - 3$ $\quad \sigma_0 \approx 11xs + 11ys - 10$

---

msort xs @ r₁ r₂ (pg. 169) with $\Gamma(r_1) = \rho_1, \Gamma(r_2) = \rho_2$

$\Delta_0 \approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{2}xs^2 + \frac{1}{2}xs - 3 \\ \rho_2 \mapsto 2xs^2 - 3xs \end{bmatrix}$ $\quad \mu_0 \approx \frac{1}{2}xs\log(xs-1) + 2xs + \frac{1}{2}\log(xs-1) + \frac{15}{8}xs^2 - \frac{55}{8}$ $\quad \sigma_0 \approx 14\log(xs-1) + 11xs + 15$

$\Delta_1 \approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{4}xs^2 + \frac{3}{2}xs - \frac{15}{4} \\ \rho_2 \mapsto xs^2 + xs - 3 \end{bmatrix}$ $\quad \mu_1 \approx \frac{5}{4}xs^2 + 3xs - \frac{25}{4}\ ^{(\star)}$ $\quad \sigma_1 \approx 11xs + 1\ ^{(\star)}$

$\Delta_2 \approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{8}xs^2 + \frac{9}{4}xs - \frac{35}{8} \\ \rho_2 \mapsto \frac{1}{2}xs^2 + 4xs - \frac{11}{2} \end{bmatrix}$

$\Delta_3 \approx \begin{bmatrix} \rho_1 \mapsto \frac{1}{16}xs^2 + \frac{23}{8}xs - \frac{79}{16} \\ \rho_2 \mapsto \frac{1}{4}xs^2 + \frac{13}{2}xs - \frac{31}{4} \end{bmatrix}$

---

partition y xs @ r₂ r₃ r₄ (pg. 150) with $\Gamma(r_2) = \rho_2, \Gamma(r_3) = \rho_3, \Gamma(r_4) = \rho_4$

$$\Delta_0 \approx \begin{bmatrix} \rho_2 \mapsto xs \\ \rho_3 \mapsto xs \\ \rho_4 \mapsto xs \end{bmatrix}^{(\star)}$$

$\mu_0 \approx 3xs$ $\quad \sigma_0 \approx 9xs - 2^{(\star)}$

$\mu_1 \approx 3xs - 1^{(\star)}$

Figure 7.20: Results of the space analysis when applied to some example functions on lists. $(\Delta_0, \mu_0, \sigma_0)$ are the initial bounds, and $(\Delta_i, \mu_i, \sigma_i)$ denote the result of the i-th iteration. A $(\star)$ mark indicates that the corresponding bound is an asymptotic fixed point.

qsort xs @ r_1 r_2 (pg. 127) with $\Gamma(r_1) = \rho_1$ and $\Gamma(r_2) = \rho_2$

$\Delta_0 \approx \left[\begin{array}{l} \rho_1 \mapsto 2xs^2 - 4xs + 2 \\ \rho_2 \mapsto xs^2 - xs + 1 \end{array}\right]$	$\mu_0 \approx 7xs^2 - 37xs + 91$	$\sigma_0 \approx 20xs - 9$
$\Delta_1 \approx \left[\begin{array}{l} \rho_1 \mapsto 2xs^2 - 6xs + 6 \\ \rho_2 \mapsto xs^2 - 2xs + 3 \end{array}\right]$	$\mu_1 \approx 7xs^2 - 48xs + 132$	$\sigma_1 \approx 20xs - 19$
$\Delta_2 \approx \left[\begin{array}{l} \rho_1 \mapsto 2xs^2 - 8xs + 12 \\ \rho_2 \mapsto xs^2 - 3xs + 6 \end{array}\right]$	$\mu_2 \approx 7xs^2 - 59xs + 184$	$\sigma_2 \approx 20xs - 29$
$\Delta_3 \approx \left[\begin{array}{l} \rho_1 \mapsto 2xs^2 - 10xs + 20 \\ \rho_2 \mapsto xs^2 - 4xs + 10 \end{array}\right]$	$\mu_3 \approx 7xs^2 - 70xs + 247$	$\sigma_3 \approx 20xs - 39$

sumList xs @ r (pg. 151) with $\Gamma(r) = \rho_2$

$\Delta_0 \approx [\rho_2 \mapsto xs]^{(\star)}$	$\mu_0 \approx xs + 1$ $\mu_1 \approx xs^{(\star)}$	$\sigma_0 \approx 9xs - 9^{(\star)}$

pascal n @ r (pg. 151) with $\Gamma(r) = \rho_1$

$\Delta_0 \approx [\rho_1 \mapsto n^2 + 3n + 2]$	$\mu_0 \approx n^2 - 2n + 17^{(\star)}$	$\sigma_0 \approx 15n + 13$
$\Delta_1 \approx [\rho_1 \mapsto n^2 + 2n + 3]$		$\sigma_1 \approx 15n + 4$
$\Delta_2 \approx [\rho_1 \mapsto n^2 + n + 5]$		$\sigma_2 \approx 15n - 5$
$\Delta_3 \approx [\rho_1 \mapsto n^2 + 8]$		$\sigma_3 \approx 15n - 14$

combNumbers m n (pg. 151)

$\Delta_0 \approx [\,]^{(\star)}$	$\mu_0 \approx m^2 - m + 12^{(\star)}$	$\sigma_0 \approx 15m + 17^{(\star)}$

fib′ n @ r (variation of *fib*, pg. 128) with $\Gamma(r) = \rho_1$

$\Delta_0 \approx [\rho_1 \mapsto 2^n - 1]$	$\mu_0 \approx 4 \cdot 2^{n-5} + 14 \cdot 2^{n-6} + 18 \cdot 2^{n-7} + 10 \cdot 2^{n-8} + 2 \cdot 2^{n-9} - 3$	$\sigma_0 \approx 5n + 7$
$\Delta_1 \approx [\rho_1 \mapsto 2^{n-1} + 2^{n-2} - 1]$	$\mu_1 \approx 2 \cdot 2^{n-5} + 6 \cdot 2^{n-6} + 10 \cdot 2^{n-7} + 16 \cdot 2^{n-8} + 18 \cdot 2^{n-9} + 10 \cdot 2^{n-10} + 2 \cdot 2^{n-11} - 4$	$\sigma_1 \approx 5n + 6$
$\Delta_2 \approx [\rho_1 \mapsto 2^{n-1} + 2^{n-4} - 1]$	$\mu_2 \approx 2 \cdot 2^{n-5} + 8 \cdot 2^{n-6} + 12 \cdot 2^{n-7} + 8 \cdot 2^{n-8} + 2 \cdot 2^{n-9} - 1^{(\star)}$	$\sigma_2 \approx 5n + 5$
$\Delta_3 \approx [\rho_1 \mapsto 2^{n-3} + 3 \cdot 2^{n-4} + 3 \cdot 2^{n-5} + 2^{n-6} - 1]$		$\sigma_3 \approx 5n + 4$

Figure 7.21: Results of the space analysis when applied to some example numeric functions and functions on lists. $(\Delta_0, \mu_0, \sigma_0)$ are the initial bounds, and $(\Delta_i, \mu_i, \sigma_i)$ denote the result of the i-th iteration. A $(\star)$ mark indicates that the corresponding bound is an asymptotic fixed point.

unshuffle xs @ r_2 r_3 (pg. 203) with $\Gamma(r_2) = \rho_2$ and $\Gamma(r_3) = \rho_3$

$$\Delta_0 \approx \left[\begin{array}{l} \rho_2 \mapsto xs + 1 \\ \rho_3 \mapsto xs \end{array}\right]^{(\star)} \qquad \mu_0 \approx 2xs + 2 \qquad \sigma_0 \approx 7xs + 2^{(\star)}$$
$$\mu_1 \approx 2xs + 1^{(\star)}$$

---

reverse′ xs @ r (variation of *reverse* on pg. 163) with $\Gamma(r) = \rho_2$

$\Delta_0 \approx [\rho_2 \mapsto xs^2 - xs + 1]$	$\mu_0 \approx xs^2 - 5xs + 15^{(\star)}$	$\sigma_0 \approx 13xs - 12$
$\Delta_1 \approx [\rho_2 \mapsto xs^2 - 2xs + 3]$		$\sigma_1 \approx 13xs - 19$
$\Delta_2 \approx [\rho_2 \mapsto xs^2 - 2xs + 6]$		$\sigma_2 \approx 13xs - 26$
$\Delta_3 \approx [\rho_2 \mapsto xs^2 - 2xs + 10]$		$\sigma_3 \approx 13xs - 33$

---

revAux xs ys @ r (pg. 163) with $\Gamma(r) = \rho_2$

$$\Delta_0 \approx [\rho_2 \mapsto xs - 1]^{(\star)} \qquad \mu_0 \approx xs^{(\star)} \qquad \sigma_0 \approx 6^{(\star)}$$

---

reverse xs @ r (pg. 163) with $\Gamma(r) = \rho_2$

$$\Delta_0 \approx [\rho_2 \mapsto xs]^{(\star)} \qquad \mu_0 \approx xs + 1^{(\star)} \qquad \sigma_0 \approx 7^{(\star)}$$

---

insertT x t @ r (pg. 154) with $\Gamma(r) = \rho_1$

$$\Delta_0 \approx \left[\rho_1 \mapsto \tfrac{1}{2}t + \tfrac{5}{2}\right]^{(\star)} \qquad \mu_0 \approx \tfrac{1}{2}t + \tfrac{9}{2} \qquad \sigma_0 \approx \tfrac{11}{2}t + \tfrac{11}{2}^{(\star)}$$
$$\mu_1 \approx \tfrac{1}{2}t + \tfrac{7}{2}$$
$$\mu_2 \approx \tfrac{1}{2}t + \tfrac{5}{2}^{(\star)}$$

---

mkTree xs @ r (pg. 154) with $\Gamma(r) = \rho_2$

$\Delta_0 \approx [\rho_2 \mapsto xs^2]$	$\mu_0 \approx xs^2 - 4xs + 14^{(\star)}$	$\sigma_0 \approx 17xs - 16$
$\Delta_1 \approx [\rho_2 \mapsto xs^2 - xs + 2]$		$\sigma_1 \approx 17xs - 27$
$\Delta_2 \approx [\rho_2 \mapsto xs^2 - 2xs + 5]$		$\sigma_2 \approx 17xs - 38$
$\Delta_3 \approx [\rho_2 \mapsto xs^2 - 3xs + 9]$		$\sigma_3 \approx 17xs - 49$

---

inorder t @ r (pg. 154) with $\Gamma(r) = \rho_2$

$\Delta_0 \approx [\rho_2 \mapsto t^2 - \frac{7}{2}t + \frac{7}{2}]$	$\mu_0 \approx 2t^2 - 22t + 79$	$\sigma_0 \approx \frac{39}{2}t - \frac{79}{2}$
$\Delta_1 \approx [\rho_2 \mapsto t^2 - \frac{11}{2}t + \frac{19}{2}]$	$\mu_1 \approx 2t^2 - 30t + 131$	$\sigma_1 \approx \frac{39}{2}t - \frac{143}{2}$
$\Delta_2 \approx [\rho_2 \mapsto t^2 - \frac{15}{2}t + \frac{39}{2}]$	$\mu_2 \approx 2t^2 - 38t + 199$	$\sigma_2 \approx \frac{39}{2}t - \frac{207}{2}$
$\Delta_3 \approx [\rho_2 \mapsto t^2 - \frac{19}{2}t + \frac{67}{2}]$	$\mu_3 \approx 2t^2 - 46t + 283$	$\sigma_3 \approx \frac{39}{2}t - \frac{271}{2}$

---

treesort xs @ r (pg. 154) with $\Gamma(r) = \rho_2$

$$\Delta_0 \approx [\rho_2 \mapsto 4xs^2 - 19xs + 28]^{(\star)} \qquad \mu_0 \approx 9xs^2 - 103xs + 340^{(\star)} \qquad \sigma_0 \approx 39xs - 123^{(\star)}$$

Figure 7.22: Results of the space analysis when applied to some example functions on lists and trees. $(\Delta_0, \mu_0, \sigma_0)$ are the initial bounds, and $(\Delta_i, \mu_i, \sigma_i)$ denote the result of the i-th iteration. A $(\star)$ mark indicates that the corresponding bound is an asymptotic fixed point.

```
min :: Int → Int → Int
min x y | x ≤ y = x
        | x > y = y

minimumAc :: Int → [Int] → Int
minimumAc ac [ ] = ac
minimumAc ac (x : xs) = minimumAc (ac `min` x) xs

removeAc :: Int → [Int] → [Int] → [Int]
removeAc x [ ] ac = ac
removeAc x (y : ys) ac
          | x == y = revAux ys ac
          | x /= y = removeAc x ys (y : ac)

selectSortAc :: [Int] → [Int] → [Int]
selectSortAc [ ] ac = ac
selectSortAc xs ac = selectOrdAc (removeAc y xs [ ]) (y : ac)
          where y = minimumAc (−∞) xs
```

Figure 7.23: Tail-recursive selection sort algorithm

constant. In particular, we obtain:

$$\begin{aligned}
\sigma_{min} &\approx \lambda x\ y.5 \\
\sigma_{minimumAc} &\approx \lambda ac\ xs.11 \\
\sigma_{removeAc} &\approx \lambda x\ ys\ ac.9 \\
\sigma_{selectSortAc} &\approx \lambda xs\ ac.17
\end{aligned}$$

□

7.7 Inference in presence of explicit destruction and polymorphic recursion

The techniques shown in this chapter yield correct upper bounds provided the input function does not have destructive pattern matching, and assuming the absence of region-polymorphic recursion. In this section we briefly sketch how to improve the algorithms of this chapter in order to deal with these language facilities. Notice, however, that none of these extensions is implemented nor formally specified. They are just given here as a proof of the feasibility of these techniques for inferring full-fledged *Safe* definitions.

If we consider region-polymorphic recursive definitions the abstract interpretation described in Section 7.3 would require no changes, since its proof of correctness does not distinguish between recursive and non-recursive function applications. Both of these are handled exactly in the same way. Polymorphic recursion does affect the computation of the initial Δ_0 and μ_0 explained in Section 7.5. Assume a function definition with m region parameters, and that the type of the i-th region parameter is ρ_i, for each $i \in \{1..m\}$. The *computeDelta* algorithm assumes that this mapping between region parameters and RTVs remains constant through the subsequent recursive calls, but this is not true in the case of recursive definitions: the i-th region parameter may be mapped to a different ρ_j ($j \neq i$) in some recursive calls, or it could be mapped to ρ_{self}. As a consequence, we have a finite number $\Gamma_1, \ldots, \Gamma_n$ of typing environments typing the subsequent recursive calls. For every recursive call, the correspondence between

region variables and RTVs is given by one of these environments. Assume we are able to find some $nr_{(i)}$ ($i \in \{1..n\}$) such that:

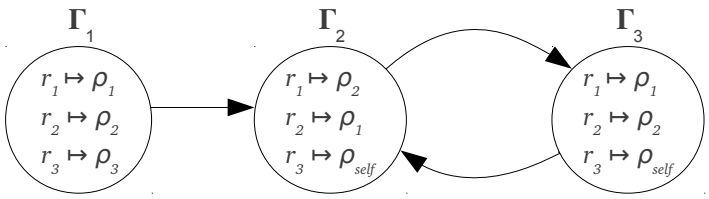
$$nr \sqsubseteq nr_{(1)} + nr_{(2)} + \cdots + nr_{(n)}$$

and each $nr_{(i)}$ bounds the number of recursive calls associated with the environment Γ_i. Then we would be able to consider each $nr_{(i)}$ separately, multiplied by the charges done by the recursive subsequences under the environment Γ_i, in the style of the following example:

Example 7.58. Given the following region-annotated definition:

$$\begin{array}{l} f :: Int \to \rho_1 \to \rho_2 \to \rho_3 \to ([Int]@\rho_1, [Int]@\rho_2)@\rho_3 \\ f\ 0\ @\ r_1\ r_2\ r_3 = ([\,]@r_1, [\,]@r_2)@r_3 \\ f\ n\ @\ r_1\ r_2\ r_3 = (n : xs, n : ys)@r_3 \\ \qquad\qquad \textbf{where}\ (xs, ys) = f\ (n-1)\ @\ r_2\ r_1\ self \end{array}$$

Assume $\Gamma = [r_1 : \rho_1, r_2 : \rho_2, r_3 : \rho_3]$. In the first recursive call, the type of first parameter becomes ρ_2 and the type of the second one becomes ρ_1, whereas no region is mapped to ρ_3. In the second recursive call, the type of the first parameter is ρ_1 again, and the type of the second parameter is ρ_2. In the third recursive call we have the same mapping as in the first one. These changes in the mappings between parameter positions and RTVs can be depicted as follows:



If $nr = \lambda n.n$ is an upper bound to the number of recursive calls, one of these calls is done with the typing environment Γ_1 above, at most $\left\lceil \frac{n-1}{2} \right\rceil$ of these calls correspond to Γ_2, and at most $\left\lceil \frac{n-1}{2} \right\rceil$ are done with Γ_3. Therefore, we have three different functions for modeling the number of recursive calls:

$$nr_{(1)} = \lambda n.1 \qquad nr_{(2)} = \lambda n. \left\lceil \frac{n-1}{2} \right\rceil \qquad nr_{(3)} = \lambda n. \left\lceil \frac{n-1}{2} \right\rceil$$

By applying the abstract interpretation rules with the recursive part of the expression assuming each Γ_i, we obtain the following abstract heaps:

$$\Delta^r_{(1)} = \begin{bmatrix} \rho_1 & \mapsto & 1 \\ \rho_2 & \mapsto & 1 \\ \rho_3 & \mapsto & 1 \end{bmatrix} \qquad \Delta^r_{(2)} = \begin{bmatrix} \rho_1 & \mapsto & 1 \\ \rho_2 & \mapsto & 1 \end{bmatrix} \qquad \Delta^r_{(3)} = \begin{bmatrix} \rho_1 & \mapsto & 1 \\ \rho_2 & \mapsto & 1 \end{bmatrix}$$

We proceed in a similar way with the base sequences of the function's body:

$$\Delta^b_{(1)} = \begin{bmatrix} \rho_1 & \mapsto & 1 \\ \rho_2 & \mapsto & 1 \\ \rho_3 & \mapsto & 1 \end{bmatrix} \qquad \Delta^b_{(2)} = \begin{bmatrix} \rho_1 & \mapsto & 1 \\ \rho_2 & \mapsto & 1 \end{bmatrix} \qquad \Delta^b_{(3)} = \begin{bmatrix} \rho_1 & \mapsto & 1 \\ \rho_2 & \mapsto & 1 \end{bmatrix}$$

Hence we get:

$$\begin{array}{rcl}
\Delta_0\ n\ \rho_1 & = & nr_{(1)}\ n * \Delta^r_{(1)}\ n\ \rho_1 + nr_{(2)}\ n * \Delta^r_{(2)}\ n\ \rho_2 + nr_{(3)}\ n * \Delta_{(3)}\ n\ \rho_1 \\
 & & +nb\ n * (\Delta^b_{(1)}\ n\ \rho_1 \sqcup \Delta^b_{(2)}\ n\ \rho_2 \sqcup \Delta^b_{(3)}\ n\ \rho_1) \\
 & = & 1 + \left\lceil \frac{n-1}{2} \right\rceil + \left\lceil \frac{n-1}{2} \right\rceil + 1 \\
\Delta_0\ n\ \rho_2 & = & nr_{(1)}\ n * \Delta^r_{(1)}\ n\ \rho_2 + nr_{(2)}\ n * \Delta^r_{(2)}\ n\ \rho_1 + nr_{(3)}\ n * \Delta_{(3)}\ n\ \rho_2 \\
 & & +nb\ n * (\Delta^b_{(1)}\ n\ \rho_2 \sqcup \Delta^b_{(2)}\ n\ \rho_1 \sqcup \Delta^b_{(3)}\ n\ \rho_2) \\
 & = & 1 + \left\lceil \frac{n-1}{2} \right\rceil + \left\lceil \frac{n-1}{2} \right\rceil + 1 \\
\Delta_0\ n\ \rho_3 & = & nr_{(1)}\ n * \Delta^r_{(1)}\ n\ \rho_3 + nb\ n * \Delta^b_{(1)}\ n\ \rho_2 \\
 & = & 1
\end{array}$$

□

With regard to destructive pattern matching facility, we need to know which variable is affected by a given **case**!. This variable, which appears explicitly in the discriminant of the **case**!, is lost after flattening. So, our first modification is to keep these variables somewhere in the sequences. Assume we adapt our *seqs* function such that it returns sequences of the form $[G \to be_1, \ldots, be_n \mid D]$, where D is a set containing the variables occurring in the discriminant of a destructive pattern matching. The $[\![\cdot]\!]_\Delta$ interpretation of sequences would be extended as follows:

$$[\![[G \to be_1, ..., be_n \mid D]]\!]_\Delta\ \Sigma\ \Gamma \quad = \quad \left[G \to \lfloor\cdot\rfloor \left\{ [\,]_f, \textstyle\sum_{i=1}^{n} ([\![be_i]\!]_\Delta\ \Sigma\ \Gamma) + \sum_{x \in D} [region(x) \mapsto -1] \right\}\right]$$

where $region(x)$ denotes the outermost RTV of the type of x. This information can be obtained from the typing environment[4]. The least upper bound with the empty abstract heap is done to avoid overall negative charges. With this new definition it can be proven that the $[\![\cdot]\!]_\Delta$ interpretation is correct, and yields a function in $\mathbb{D}$. However, the *computeDelta* algorithm is more problematic: it could return an initial bound Δ_0 not belonging to $\mathbb{D}$, since the monotonicity property might not hold. However, in those examples where Δ_b and Δ_r (being these as defined in *computeDelta*) belong to $\mathbb{D}$, so does the resulting Δ_0. The same applies to μ_0 and σ_0: if their respective components belong to $\mathbb{F}$, so do the results μ_0 and σ_0.

Example 7.59. Assume the *revAuxD* function of Example 5.2.

$$\begin{array}{l}
revAuxD :: [\alpha]@\rho_1 \to [\alpha]@\rho_2 \to \rho_2 \to [\alpha]@\rho_2 \\
revAuxD\ xs\ ys\ @\ r = \textbf{case!}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad [\,] \to ys \\
\qquad\qquad\qquad\qquad (x : xx) \to \textbf{let}\ x_1 = (x : ys)@r\ \textbf{in}\ revAuxD\ xx\ x_1\ @\ r
\end{array}$$

We get the following sequences:

$$\begin{array}{rcl}
seq_1 & = & [xs \geq 1 \to ys \mid xs] \\
seq_2 & = & [xs \geq 2 \to (x : ys)@r, revAuxD\ xx\ x_1 \mid xs]
\end{array}$$

[4]It is worth noting that, with this new definition, the set R_f occurring in the definition of $\mathbb{D}$ would have to be extended with the RTVs occurring in the types of the input parameters, since there could be negative charges in those RTVs.

The first one is a base sequence, from which we get following abstract heap:

$$\lfloor \Delta_b \rfloor = \lambda xs\ ys.[xs \geq 1 \rightarrow [\rho_2 \mapsto 0]]$$

The second sequence is recursive, and by assuming $\Sigma(revAuxD) = ([\,]_f, 0, 0)$ we obtain:

$$\lfloor \Delta_r \rfloor = \lambda xs\ ys.[xs \geq 2 \rightarrow [\rho_2 \mapsto 0]]$$

Both of these heaps belong to $\mathbb{F}$. Since $nr = \lambda xs\ ys.xs - 1$ and $nb = \lambda xs\ ys = 1$, the *computeDelta* algorithm yields the following result:

$$\Delta_0 = \lambda xs\ ys.[xs \geq 1 \rightarrow [\rho_2 \mapsto 0]]$$

□

7.8 Conclusions and related work

We have introduced an abstract interpretation-based analysis for computing memory bounds, which takes heap and stack consumption into account. Our approach consists in finding some initial correct approximations, and applying an abstract interpretation function repeatedly in order to increase their accuracy. The strengths of our approach can be summarized as follows:

1. It scales well to large programs, as each *Safe* function can be separately inferred. The relevant information about the called functions is recorded in the signature environment.
2. It supports arbitrary algebraic data types, provided they do not present mutual recursion.
3. We get upper bounds for the maximum amount of *live* memory, as the inference algorithms take into account the deallocation of dead regions made at function termination.
4. It can accommodate several complexity classes, provided these are monotone with respect to the input sizes.
5. It is, to our knowledge, the first approach in which the upper bounds can be improved just by iterating the inference algorithm.

The latter point should be interpreted cautiously, since a larger amount of iterations implies more accuracy, but also more complexity in the resulting expressions (at least with our current implementation). Notice, however, that we still can get simple *asymptotic* bounds. In those cases where the programmer is only interested in asymptotic bounds, these can be given as input to the next iteration, and the result is also (asymptotically) another asymptotic bound, much simpler to compute than if the iteration is done with a non-asymptotic initial bound.

Notice that, in some cases, it is possible to get an infinite, strictly decreasing sequence of upper bounds without reaching a fixed point. It is an interesting subject of future work to study the convergence of such sequences.

A weak point that still requires more work is the restriction we have imposed to our functions: they must be non-negative and monotone. That is why destructive pattern matching has been omitted from our analysis in a first phase. In Section 7.7 we have given some sufficient conditions in which cell

deallocation can be taken into account without breaking the monotonicity restriction. Notice, however, that these are not necessary conditions: there are some examples not satisfying them, but still resulting in monotone bounds. For instance, the *inssortD* function of Example 7.55 gives a constant space bound in presence of explicit destruction. In spite of this, restricting the abstract domain to monotone functions still excludes some interesting function definitions, such as those that destroy more memory than they consume, or those whose output size decreases as the input size increases.

The first approaches to space consumption analysis were restricted to infer linear memory bounds. Hughes and Pareto developed in [61] a type system and a type-checking algorithm which guarantees safe memory upper bounds in a region-based first-order functional language. Unfortunately, the approach requires the programmer to provide detailed consumption annotations. The first fully automatic technique is due to Hofmann and Jost. In [58] they present a type system and a type inference algorithm which, in case of success, guarantees linear heap upper bounds for a first-order functional language, and it does not require programmer annotations. This type system has been extended in [64] so as to support higher-order functions.

Beyond linear bounds, the pioneer research on memory consumption is that carried out under the AHA project (*Amortised analysis of Heap space Usage*) [43], aimed at inferring amortised costs for heap space. In [105], Shkaravska et al. introduce a variant of sized types, in which the size annotations can be polynomials of any degree. They address two novel problems: polynomials are not necessarily monotonic and they are *exact* bounds, as opposed to approximate upper bounds. These bounds are inferred with a combination of testing and polynomial interpolation-based techniques. In [107] they extend their work to give approximate upper bounds on the output sizes, thus broadening the class of analysable programs. In this case, the size relations are expressed via non-deterministic conditional rewriting systems, from which a closed form is extracted by using polynomial interpolation. A strength of this approach is that, since the inference is testing-based, the function being inferred can be considered as a black box. This allows them to use the same inference techniques in different applications, such as, for example, inferring loop-bounds for Java programs [106]. Unfortunately, polynomial interpolation-based techniques do not necessarily lead to a correct upper bound, and some external mechanism is needed for checking that the result of the analysis is sound. Our analysis always gives correct bounds if the externally given call-tree and size information is correct.

The COSTA system (*COSt and Termination Analyzer for Java Bytecode*) [5, 6] implements a fully mechanical approach to resource analysis for Java bytecode programs. It is based on the classical method of Wegbreit [119]. It consists in the generation of a recurrence relation which captures the cost of the program being analysed, and the computation of a closed form by using a built-in recurrence solver PUBS [3, 4]. Their results go far beyond linear bounds: the system can infer polynomial, logarithmic, and exponential bounds. COSTA also allows a restricted form of non-monotonicity, provided it occurs in the context of linear expressions. The computation of our initial Δ_0 shares some similarities with the way in which PUBS solves recurrence relations. The main difference is that PUBS computes the *nb* and *nr* functions giving the number of calls in a recurrence, whereas these functions are given externally in our system. However, PUBS' approach of computing both *nr* and *nb* from the *len* function may yield imprecise over-approximations in some cases (e.g. Quicksort). A drawback, in comparison to our system, is that COSTA does not support non-linear size relations, even if these were given externally. The reason behind this is that these relations are the guards of the recurrence being generated, and PUBS assumes these guards to be conjunctions of linear constraints.

More recently, the COSTA team has extended their system in order to deal with different models of garbage collection [7]. The new results are very promising. In their work, they claim that their liveness-

based model can accommodate the region-based memory management approach of [27], although this integration is neither described nor formally specified.

Another promising technique for inferring polynomial bounds is due to Hoffmann and Hofmann [56], which extends the work of [58]. In the univariate case, their system is able to infer bounds, expressed as non-negative linear combinations of binomial coefficients. These combinations subsume the class of polynomials with non-negative coefficients, while allowing some polynomials with negative coefficients, such as $x^2 - x$. However, it does not cover some other natural-valued polynomial bounds, such as $3x^2 - 6x + 7$. In a more recent work [55], Hoffmann et al. extend their analysis to the inference of multivariate functions. Unlike our system, they do not handle regions nor explicit destruction, although they claim that the latter can be added with no difficulty. Another drawback is that, to our knowledge, their approach does not deal with arbitrary data types.

Both the original type system of [58], and its extension to polynomial bounds [56] are closely related to the potential method used in the context of amortised analysis [110, 32]. This approach provides an implicit notion of *input size* (the number of elements in a list, usually), but there is no explicit dependence between input sizes and costs, since the latter are given by the potential assigned to each element of the input and output DSs. On the contrary, our space analysis, as well as the COSTA system, are both based on explicit sizes and symbolic manipulation. It is arguable whether one is better than the other. An advantage of amortised analysis is that it allows to express more precisely the memory costs, when they do not depend exclusively on the input size. Moreover, amortised analysis yields more precise results when considering several functions executed in sequence, since it accounts for the overall costs as a whole, instead of just adding the worst-case costs of each function separately. On the contrary, the approaches based on symbolic manipulation can be extended more easily, for instance, by including cost expressions from several complexity classes, as it is done in *Safe*.

Chapter 8

Conclusions and future work

8.1 Conclusions

In this thesis we have designed and implemented several analyses for proving pointer-safety and bounded memory consumption of programs written in a first-order functional language combining region-based memory management with explicit deallocation. Let us summarize the goals of this work, and assess the extent to which they have they been achieved.

1. **Develop an efficient, fully automatic, type-based, static analysis for ensuring pointer safety** [98, 84, 85, 81].

   We have defined a type system which attaches region type variables and marks to standard Hindley-Milner algebraic types. Region type variables reflect all the regions in which a DS may live. They allow the compiler to track the DSs that will be located in the working region at runtime, and ensure that neither the input parameters nor the result of the function are located in this region. Marks indicate whether a given DS is destroyed via destructive pattern matching. An auxiliary analysis is needed for approximating the sharing relations between the different variables. Those relations determine which variables may get corrupted when some of them are disposed of.

   We have provided an inference algorithm for this destruction-aware type system. It consists of two separate parts:

   (a) A region inference algorithm, which annotates the program with region variables, and infers the region type variables in the typing derivation. The fact that our language is first-order, and that our memory model is simpler than in [116], makes our inference algorithm simpler and more efficient than Tofte and Birkedal's algorithm [112].

   (b) A mark inference algorithm, which annotates the types with marks, and ensures that destructive pattern matching is done in a safe way.

   We have tested the inference algorithms with several case studies. A number of algorithms manipulating data structures with constant heap space cost have been successfully typed by the algorithm. In those which have not, we have identified the auxiliary sharing analysis as the main source of accuracy loss.

   The type system has been proven correct with respect to the semantics of *Safe*, whereas the inference algorithms have been shown to be correct with respect to the type system. In addition, the

mark inference algorithm is complete with respect to the latter.

2. **Certify pointer-safety in an automatic way [36].**

   We have shown how the compiler generates an Isabelle/HOL script establishing that the program being certified is well-typed. Together with a set of previously proved theorems guaranteeing pointer-safety of well-typed programs, this certifies that the object program is pointer-safe.

3. **Develop a memory cost model for *Safe* programs [82].**

   From the big-step operational semantics of the language, and in a series of successive refinements, we have derived an imperative abstract machine (called SVM) in which *Safe* programs are run. The translation between *Safe* and the instructions of the SVM has been formalized. This translation has allowed us to set up a connection between each *Safe* expression and its heap and stack memory consumption in the SVM. This connection has been formalised by means of some resource annotations, which have been attached to the big-step operational semantics. These annotations make up a cost model for *Safe*. Finally, these annotations have been proved correct with respect to the actual consumption of *Safe* programs in the SVM.

4. **Develop an abstract interpretation-based analysis for inferring heap and stack memory bounds [83].**

   We have developed an analysis which, given a *Safe* function, returns an expression bounding its stack and heap memory consumption as a function on the sizes of the input parameters. Our abstract domain is the set of monotone functions with the usual $\sqsubseteq$ ordering.

   As a first step, we have defined an abstract interpretation function describing the heap and stack needs of each expression. This interpretation has been proved correct with respect to the cost model developed previously. After this, we have introduced algorithms addressing the inference problem for recursive functions. We have proved that the result of these algorithms is reductive under certain conditions. This implies that we can iterate the abstract interpretation in order to improve the results of these algorithms with each iteration.

   In a first stage we have considered the inference of recursive bounds in absence of polymorphic recursion and explicit destruction, but we have shown the feasibility of the integration of these features in our inference algorithms, provided that the result belongs to the abstract domain.

   The main advantages of our approach are its flexibility for including functions from a broad set of complexity classes, in spite that, at this moment, non-monotonic functions are not considered. A remarkable feature is that the obtained upper bounds can be improved by just iterating the inference algorithm.

8.2 Future work

In this section we describe some possible extensions and improvements that are subject of future work. Regarding our destruction-aware type system, we have the following issues to address:

- **Improved sharing analysis.**

  Some of the case studies of Section 5.3 show the need for a more accurate sharing analysis than the one currently implemented [98], so that the programmer does not have to adjust manually its results in these cases. Our current analysis distinguishes between the recursive and non-recursive

part of a data structure, and keeps track of those variables that may point to each of these parts. However, when approximating the sharing relations in the context of function applications, this distinction has to be lost, in order to keep the analysis sound. A possibility that may increase the accuracy of our analysis is to give more information on which particular descendant of a data structure is being shared by a variable.

- **Principal types.**

  In Section 4.7 we have proved the absence of principal types in the $\vdash_{Reg}$ rules, when considering some pathological cases, such as the function that copies its parameter and returns it as a result. The reason behind this is that the copy operator must be applied to a variable with an *algebraic* type, but it can be *any* algebraic type. The syntax of our types allows us to specify concrete algebraic types, such as $[\alpha]@\rho_1$ or *Tree Int* @ ρ_2, and also polymorphic type variables, which may be instantiated to any type (basic or algebraic). However, there is no mechanism in our type syntax for denoting an arbitrary algebraic data structure whose outermost variable is fixed. A solution to this problem involves adding a new sort of types that allows to specify such algebraic structures. It is subject of short-term future work to integrate this extension in the type system.

- **Mutually recursive data types.**

  The current implementation of *Safe* does not allow the specification of mutually recursive data types. Extending the type system with this broader class of types would have major implications in the sharing analysis, but also in the semantics of the language itself. In absence of mutual recursion, we can easily refer to the recursive spine of a DS just by considering the recursive positions of the data constructors that make up that spine. As a consequence, the recursive children of a given cell are located just one pointer away from this cell. This does not hold if two data types are mutually recursive, as there might be several cells staying in the middle of the path between a given cell and its recursive children. Under these circumstances, our notion of recursive positions would become more sophisticated.

- **Data types with nested recursion.**

  Safe currently allows recursion in data types, but only at the outermost level. Support for nested recursion involves the same implications as having mutually recursive data types. In particular, nested recursion can be translated into a set of mutually recursive data structures.

- **Nested destruction.**

  Although our in-danger types are able to track those DS whose non-recursive part may be corrupted, they give very little information on *which* part. For instance, assume a variable x with type $[[\alpha]@\rho_1]\#@\rho_2$. The fact that x gets an in-danger type implies that:

  - The outermost list pointed to by x may be destroyed, and/or
  - Some of the innermost lists may be destroyed, and/or
  - Some element in some innermost list may be destroyed.

  By extending the type system with nested condemned types, we can increase the accuracy of the type system in determining the level of nesting to which the DS pointed to by x can be destroyed. For instance, $[[\alpha]@\rho_1]!@\rho_2$ only allows the destruction of the outermost list, whereas $[[\alpha]!@\rho_1]!@\rho_2$ allows the programmer to destroy the innermost lists, while ensuring that none of their elements

of type α is destroyed. In order to carry out this improvement, we have to increase the accuracy of our auxiliary sharing analysis, so that it distinguishes between the different layers of a DS. This is more precise than distinguishing only the outermost recursive spine from the rest of the DS.

With respect to our abstract interpretation-based space consumption analysis, we highlight the following aspects to be improved:

- **Support for different kinds of resources.**

  In this thesis we have focused on bounding memory consumption, but the research field of resource analysis is broader; it addresses more kinds of resources. Most of the ideas shown in Chapter 7 can be applied to different classes of resources. A desirable property of a resource bounds analysis is to have some kind of parametrizability on the resource being measured. As an example, the COSTA system [5] provides support for several fixed notions of resources: number of instructions executed, memory consumption in the presence of different models of garbage collection, and number of calls to a given method.

  The work of Aspinall et al. [9] introduces a generic concept of *resource algebras*, providing a general framework for embedding different kinds of resources. A generalization of our abstract interpretation function for arbitrary resource algebras should be straightforward. However, the computation of the initial bounds requires special attention. The algorithm for computing the initial Δ_0 (see Section 7.5.3) can be applied on those kind of resources which are *compositionally additive*, that is, those resources in which the cost of a compound expression is the sum of the costs of its basic sub-expressions. The algorithm for the computation of the initial σ_0 (Section 7.5.5) can be applied on those resources whose consumption may increase and/or decrease along the execution of the program, and we are interested in a maximum consumption level. The algorithm for computing μ_0 also finds a maximum consumption level, but also entails an additive component. A more general resource framework should set certain conditions on resource algebras for determining which algorithm can be used for computing the initial approximation of the bounds to a given resource consumption.

- **Support for different size models.**

  In the same way it is useful to consider different kinds of resources, it is also convenient to consider different kinds of size models. With the term *size* we mean being bound in cost expressions, and that gives an idea on how easy or difficult is to process the input. In this sense, the size of a DS is not necessarily related to the amount of heap memory needed by the DS. For instance, in our system, the size of a natural number is its value, although natural numbers do not take up space in the heap. *Safe* has a fixed notion of size of a DS: the number of cells in the recursive spine. This notion is also fixed in the COSTA system: the size of a DS is the length of the longest reachable chain from its root. The size model may decisively influence the complexity order of the resulting bound. For instance, consider a function for generating a copy of a binary tree. In *Safe* this function is reported to have $\mathcal{O}(t)$ space complexity, whereas, according to COSTA, it has $\mathcal{O}(2^t)$ space complexity. The actual difference is the meaning of the t variable: in *Safe* it stands for the number of nodes in the tree, whereas in COSTA it represents the height of the tree.

  A good starting point is the work of Tamalet et al. [109], which presents a sized type system with arbitrary algebraic data types. Sizes are defined by assigning a weight to each data constructor. The amortised analysis-based approach of Jost et al. [64, 63] can also assign different potentials to each constructor.

- **Non-monotonic upper bounds.**

  As it was shown in Section 7.2, the abstract domain in our system is the set $\mathbb{F}$ of monotonic functions. The monotonicity condition is essential in order to ensure the correctness of the abstract interpretation, but it is very restrictive in presence of explicit deallocation, since the latter can produce negative charges in the heap. There are some interesting functions that cannot be considered with this restriction, such as those destroying a data structure without building anything. In these cases we can observe some sort of duality with respect to our current approach. Assume a function that only destroys a cell in each recursive call, and does nothing else. Regarding heap consumption, the worst-case corresponds to the *minimum* number of recursive calls, as, in that case, less cells are destroyed, which implies less free memory. Hence, when analysing a function whose costs are monotonically decreasing in each call, we have to consider *lower bounds* on the size of the call-tree. Those cost functions which are neither monotonically increasing nor decreasing would have to be considered in a piecewise basis.

- **Convergence of a sequence of cost functions.**

  Our approach allows us to get a decreasing sequence of upper bounds by iterating the abstract interpretation function. In some case studies we have obtained an infinite strictly decreasing sequence of functions, without reaching a fixed point. In many cases we have been able to graphically represent the behaviour of this sequence, and we have been able to (manually) determine the cost function to which this sequence converges, which turned out to be a fixed point of the abstract interpretation. It would be interesting to study the conditions under which we can ensure the convergence of those sequences, and the resulting function that we would obtain when the number of iterations tends to infinity.

Finally, and with respect to the *Safe* language itself, there are two extensions that are worth considering:

- **Higher-order functions.**

  The most challenging (but rewarding, as well) extension which is subject of future work is the inclusion of higher-order functions in *Safe*. This would require a change in almost every aspect of the language: syntax, semantics, virtual machine, region and mark inference, sharing analysis, space consumption analysis, and certification.

  The adaptation of some of the already existing analyses to a higher-order language can be carried out with relatively low difficulty. This includes the region inference algorithm, and the space consumption analysis. With respect to the former, the MLKit compiler [116, 113] already supports region inference for higher-order functions. Since we will maintain the decision of having a single region per call, we expect that our algorithm will remain simpler than that appearing in [112]. With respect to the space consumption analysis for higher-order functions, the cost expressions of functional parameters must appear symbolically in the resulting bound. Only when concrete functions are passed as parameters (and their memory costs have been inferred), these symbols can be replaced by particular cost functions. A different approach, based on amortised analysis, is that of [64], which extends the work of [58] in order to include support for higher-order functions.

- **Alternative target languages.**

  The current implementation of *Safe* generates Java bytecode as a result. A strong point of the Java virtual machine is that it is available in many software and hardware platforms. However, it is somewhat restrictive with regard to memory management. As a consequence of this, the explicit

destruction of a cell is handled by linking that cell to a free list. This cell is reused by the runtime system when a subsequent allocation takes place. Moreover, the statically typed nature of the JVM makes the process of translation awkward. It is convenient to target the bytecode generation to other virtual machines and compilation frameworks which allow more flexibility in memory management. A notable example of this is the LLVM (*Low Level Virtual Machine*) compilation framework [71], which provides a language-independent instruction set. Programs written in this language can be further translated into machine code.

Appendix A

Type constraints solving by unification

The region inference algorithm described in Chapter 4 involves the computation of a solution to a previously generated set of constraints between types. In this appendix we put forward an algorithm for computing this solution. This algorithm is an extension of [12], which, in turn, is based on the original Robinson's unification algorithm [101]. This algorithm is extended to support *isData* constraints and $\approx_k$ equations, which are weaker forms of standard unification (see 4.4.1 for details)

We provide here a description of our algorithm as a state machine. A *configuration* is a pair of the form $(E, \langle \theta, \overline{\varphi_i}^p \rangle)$, where E is the set of constraints to be solved and $\langle \theta, \overline{\varphi_i}^p \rangle$ is the partial solution accumulated from the initial configuration. We assume that the number of elements p in the list $\overline{\varphi_i}^p$ is fixed, and the corresponding superscript will be ommited when it is not relevant. We also define a transition relation $\Longrightarrow$ between configurations, as the minimal relation satisfying the rules in Figure A.1. The reflexive-transitive closure of this relation is denoted by $\Longrightarrow^*$.

Given a set E of constraints to be solved, our algorithm starts from the initial configuration $E, \langle id, \overline{[\,]}^p \rangle$ and applies successively the rules in Figure A.1 in order to reach a configuration in which the set of constraints is empty. The pair $\langle \theta, \overline{\varphi_i} \rangle$ contained within this final configuration is a solution to E, that is, it satisfies the conditions stated in Definition 4.5. If the state machine gets stuck in a configuration in which the set of constraints to be solved is nonempty, the algorithm returns an error.

The $\overline{\varphi_i}$ given as a result is a list of functions on **RegType** $\rightarrow$ **RegType**. However, during the algorithm, some φ_i may map a given RTV to several different RTVs. As a consequence, the $\overline{\varphi_i}$ ocurring in the intermediate states of the $\Longrightarrow^*$ derivation are *relations* on **RegType**, rather than functions. Hence we shall consider our φ_i as a relation on **RegType**. The notation $\varphi_i(\rho) \ni \rho'$ specifies that the pair (ρ, ρ') belongs to φ_i. We extend the notation $\varphi_k(\rho) \ni \rho'$ to arbitrary types as follows:

$$
\begin{array}{lcll}
\varphi_k(B) & \ni & B & \\
\varphi_k(\alpha) & \ni & \alpha & \\
\varphi_k(T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m) & \ni & T\ \overline{s'_i}^n\ @\ \overline{\rho_j}^m & \text{if } \forall i \in \{1..n\}.\varphi_k(s_i) \ni s'_i \ \wedge\ \forall j \in \{1..m\}.\varphi_k(\rho_j) \ni \rho'_j \\
\varphi_k(\overline{s_i}^n \rightarrow \overline{\rho_j}^m \rightarrow s) & \ni & \overline{s'_i}^n \rightarrow \overline{\rho'_j}^m \rightarrow s' & \text{if } \forall i \in \{1..n\}.\varphi_k(s_i) \ni s'_i \ \wedge\ \forall j \in \{1..m\}.\varphi_k(\rho_j) \ni \rho'_j \\
 & & & \wedge\ \varphi_k(s) \ni s'
\end{array}
$$

Notice that, also in this case, there may be more than one s' such that $\varphi_k(s) \ni s'$. Our algorithm ensures that the list of relations $\overline{\varphi_i}$ obtained in the final state is actually a list of functions. Only then we can write $\varphi_i(\rho) = \rho'$. Moreover, if φ_k is a function on regions, so will be its extension to arbitrary types.

Name		Equations	Substitutions	Cond.
$[ID]$		$\{\tau = \tau\} \uplus E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[VAR]$		$\{\alpha = s\} \uplus E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	(1)
	$\Longrightarrow$	$[\alpha \mapsto s](E),$	$\langle [\alpha \mapsto s] \circ \theta, \overline{\varphi_i}^p \rangle$	
$[REG]$		$\{\rho_1 = \rho_2\} \uplus E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	(2)
	$\Longrightarrow$	$[\rho_1 \mapsto \rho_2](\bigcup_{i=1}^{p} E_i \cup E)$	$\langle [\rho_1 \mapsto \rho_2] \circ \theta, \overline{[\rho_1 \mapsto \rho_2](\varphi_i)}^p \rangle$	
$[FUN]$		$\{\overline{s_i}^n \to \overline{\rho_j}^m \to s = \overline{s'_i}^n \to \overline{\rho'_j}^m \to s'\} \uplus E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$\{\overline{s_i = s'_i}^n, \overline{\rho_j = \rho'_j}^m, s = s'\} \cup E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[CONS]$		$\{T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m = T\ \overline{s'_i}^n\ @\ \overline{\rho'_j}^m\} \uplus E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$\{\overline{s_i = s'_i}^n, \overline{\rho_j = \rho'_j}^m\} \cup E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[ISD1]$		$\{T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m = isData(\alpha, \rho)\} \uplus E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	(3)
	$\Longrightarrow$	$\{\alpha = T\ \overline{s_i}^n\ @\ \overline{\rho_j}^{m-1} \rho', \rho_m = \rho\} \cup E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[ISD2]$		$\{\alpha = isData(T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m, \rho)\} \uplus E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$\{\alpha = T\ \overline{s_i}^n\ @\ \overline{\rho_j}^{m-1} \rho\} \cup E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[ISD3]$		$\{T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m = isData(T\ \overline{s'_i}^n\ @\ \overline{\rho'_j}^m, \rho)\} \uplus E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$\{T\ \overline{s_i}^n\ @\ \overline{\rho_j}^{m-1} \rho = T\ \overline{s'_i}^n\ @\ \overline{\rho'_j}^m\} \cup E,$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[QFUN]$		$\{\overline{s_i}^n \to s \approx_k \overline{s'_i}^n \to s'\} \uplus E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$\{\overline{s_i \approx_k s'_i}^n, s \approx_k s'\} \cup E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[QCONS]$		$\{T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m \approx_k T\ \overline{s'_i}^n\ @\ \overline{\rho'_j}^m\} \uplus E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$\{\overline{s_i \approx_k s'_i}^n, \overline{\rho_j \approx_k \rho'_j}^m\} \cup E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[QVAR\text{-}L]$		$\{\alpha \approx_k s\} \uplus E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	(4)
	$\Longrightarrow$	$\{s' \approx_k s\} \cup [\alpha \mapsto s'](E)$	$\langle [\alpha \mapsto s'] \circ \theta, \overline{\varphi_i}^p \rangle$	
$[QVAR\text{-}R]$		$\{s \approx_k \alpha\} \uplus E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	(4)
	$\Longrightarrow$	$\{s \approx_k s'\} \cup [\alpha \mapsto s'](E)$	$\langle [\alpha \mapsto s'] \circ \theta, \overline{\varphi_i}^p \rangle$	
$[QREG1]$		$\{\rho_1 \approx_k \rho_2\} \uplus E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	(5)
	$\Longrightarrow$	$\{\rho_2 = \rho'\} \cup E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
$[QREG2]$		$\{\rho_1 \approx_k \rho_2\} \uplus E$	$\langle \theta, \overline{\varphi_i}^p \rangle$	(6)
	$\Longrightarrow$	E	$\langle \theta, \varphi_1 \cdots \varphi_k \uplus [\rho_1 \mapsto \rho_2] \cdots \varphi_p \rangle$	
$[QEND]$		$\{\overline{\alpha_i \approx_k \alpha'_i}^n\}$	$\langle \theta, \overline{\varphi_i}^p \rangle$	
	$\Longrightarrow$	$\{\overline{\alpha_i = \alpha'_i}^h\}$	$\langle \theta, \overline{\varphi_i}^p \rangle$	

Figure A.1: Rules defining the $\Longrightarrow$ relation between configurations. For each rule, the topmost line stands for the initial configuration and the line below it denotes the final configuration. Some rules may only be applied under certain conditions, which are described in Figure A.2.

(1) $\alpha \notin fv(s)$

(2) $\rho_1 \not\equiv \rho_2 \wedge \forall i \in \{1..p\}.E_i = \begin{cases} \{\rho = \rho'\} & \text{if } (\rho_1,\rho),(\rho_2,\rho') \in \varphi_i \text{ for some } \rho,\rho' \text{ such that } \rho \not\equiv \rho' \\ \emptyset & \text{otherwise} \end{cases}$

(3) $fresh(\rho')$

(4) $s \notin \mathbf{TypeVar}, \alpha \notin fv(s), s' = freshReg(s)$

(5) $(\rho_1,\rho') \in \varphi_k$

(6) $\nexists \rho'.(\rho_1,\rho') \in \varphi_k$

Figure A.2: Conditions under which the unification rules of Figure A.1 may be applied.

Since we have generalized the notion of our $\overline{\varphi_i}^p$ functions to relations, it is necessary to modify the Definition 4.5 as follows:

Definition A.1. A pair $\langle \theta, \overline{\varphi_i} \rangle$ is a solution to a set of constraints E (denoted $\langle \theta, \overline{\varphi_i} \rangle \vDash E$) if and only if the following conditions hold:

1. For each $s_1 = s_2 \in E$, then $\theta(s_1) \equiv \theta(s_2)$. The same applies to equations of the form $sf_1 = sf_2$.
2. For each $s = isData(s', \rho) \in E$ then $\theta(s) \equiv T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m(\theta(\rho))$ and $\theta(s') \equiv T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m\rho'$ for some $\overline{s_i}^n, \overline{\rho_j}^m$ and ρ'.
3. For each $s_1 \approx_k s_2$, then $\varphi_k(\theta(s_1)) \ni \theta(s_2)$. The same applies to equations of the form $sf_1 \approx_k sf_2$.

If the $\overline{\varphi_i}$ are functions, this definition is equivalent to Definition 4.5. In order to ensure that the $\overline{\varphi_i}$ resulting from the algorithm are functions, we have to prove that the following property is an invariant preserved by the $\Longrightarrow$ relation:

If a φ_k relates the same ρ to different RTVs ρ_a, ρ_b, the latter will be eventually unified.

The following definition formalizes these intutitions:

Definition A.2. A machine configuration $E, \langle \theta, \overline{\varphi_i}^p \rangle$ is said to be *good* if, for every $k \in \{1..p\}$ and $\rho, \rho_a, \rho_b \in \mathbf{RegType}$ such that $\varphi_k(\rho) \supseteq \{\rho_a, \rho_b\}$ and $\rho_a \not\equiv \rho_b$, then there exists a list of region type variables $\overline{\rho_i}^n$ (with $n \geq 2$) such that:

$$\rho_a \equiv \rho_1 \wedge \rho_b \equiv \rho_n \wedge \forall j \in \{1..n-1\}.\rho_j = \rho_{j+1} \in E$$

We abbreviate this as follows:

$$\{\rho_a \equiv \rho_1 = \rho_2 = \ldots = \rho_n \equiv \rho_b\} \subseteq E$$

where $\equiv$ denotes syntactic equality, and $=$ denotes belongship to E.

The existence of a chain in E between two different RTVs may be considered as an equivalence relation on **RegType**. Two RTVs belong to the same equivalence class if and only if there exists a chain of equations connecting them in E. As an example, let us assume the following definition of φ_k for some k:

$$\varphi_k = \{(\rho_1,\rho_3),(\rho_1,\rho_5),(\rho_1,\rho_8),(\rho_2,\rho_4),(\rho_2,\rho_7)\} \tag{A.1}$$

The configuration $E, \langle \theta, \overline{\varphi_i} \rangle$ is good if $E = \{\rho_3 = \rho_5, \rho_5 = \rho_8, \rho_4 = \rho_7\}$. There exists a chain of equations in E between ρ_3 and ρ_8, namely $\{\rho_3 = \rho_5 = \rho_8\} \in E$. We have two equivalence classes: $\{\rho_3, \rho_5, \rho_8\}$ and $\{\rho_4, \rho_7\}$.

We now explain the rules of Figure A.1. Rules [*ID*], [*VAR*], [*FUN*] and [*CONS*] are standard and deserve no explanation. The $=$ rules are symmetric, so the equations of the form $s = \alpha$ are deal with by [*VAR*]. Rule [*REG*] is applied when unifying two region types. In addition to generating the corresponding substitution, more equations may be generated in order to preserve the goodness property of the initial configuration. For example, given the definition of φ_k in (A.1), let us assume that the equation $\rho_1 = \rho_2$ is being processed. We apply the substitution $[\rho_1 \mapsto \rho_2]$ to this relation, and we obtain:

$$[\rho_1 \mapsto \rho_2](\varphi_k) = \{(\rho_2, \rho_3), (\rho_2, \rho_5), (\rho_2, \rho_8), (\rho_2, \rho_4), (\rho_2, \rho_7)\}$$

In order to preserve the invariant, every RTV in the set $\{\rho_3, \rho_5, \rho_8\}$ has to be unified with the RTVs of the set $\{\rho_4, \rho_7\}$, as the goodness property demands. This is done by selecting a representative of each equivalence class and generating an equation between these two representatives. In our example, we could choose $\rho_3 = \rho_4$. With this new equation the two equivalence classes would be merged into a single equivalence class. If we denote by E' the set $E \cup \{\rho_3 = \rho_4\}$, we can always find a chain in E' between two given variables of this class. For instance, between ρ_5 and ρ_7: $\{\rho_5 = \rho_3 = \rho_4 = \rho_7\} \subseteq E'$.

Rules [*ISD1*], [*ISD2*], [*ISD3*] deal with *isData* constraints. These are relaxed forms of standard unification, since they allow different outermost regions in each side of the equation. Notice that there is no rule for processing equations of the form $\alpha_1 = \mathit{isData}(\alpha_2, \rho)$, which are not addressed until one of the type variables is replaced by an algebraic type. If this substitution never takes place, the algorithm fails.

The rest of the rules involve $\approx$-equations. In this case, the right-hand side of the equations must be a region instance of the left-hand side. The corresponding φ_k may be updated when processing equations of the form $\rho_1 \approx_k \rho_2$. This equation would add the pair (ρ_1, ρ_2) to φ_k if ρ_1 is not related to any RTV in φ_k (rule [*QREG2*]). Otherwise we would have to unify the ρ_2 with the RTV already bound to ρ_1, so that the goodness property is preserved (rule [*QREG1*]). Finally, the rule [*QEND*] ensures that the equations of the form $\alpha_1 \approx_k \alpha_2$ are the last ones to be processed, and they will processed as standard unification equations ($\alpha_1 = \alpha_2$). By delaying the processing of these equations we achieve more general types, since one of involved variables may be instantiated to a more concrete type, so that the variable occurring in the other side of the $\approx_k$-equation may get the same type with different RTVs. On the contrary, if we unify both variables with an equation of the form $\alpha_1 = \alpha_2$, they will be forced to have the same region types, if they are instantiated later.

Now we prove that this property is preserved by the $\Longrightarrow$ rules of Figure A.1.

Lemma A.3. *Given a transition $E, \langle \theta, \overline{\varphi_i} \rangle \Longrightarrow E', \langle \theta', \overline{\varphi_i'} \rangle$, if the initial configuration is good, then so is the final configuration.*

Proof. By case distinction on the $\Longrightarrow$ rule applied. Most cases are trivial, except the following ones:

- **Case** [*ID*]

  In this case $E = E' \uplus \{\rho = \rho\}$ for some ρ and $\overline{\varphi_i} = \overline{\varphi_i'}$. So, if $\varphi_k'(\rho') \ni \{\rho_a, \rho_b\}$ with $\rho_a \not\equiv \rho_b$ for some ρ', then $\varphi_k(\rho') \ni \{\rho_a, \rho_b\}$. Since the initial configuration is good, we have the following chain:

  $$\{\rho_a \equiv \rho_1 = \rho_2 = \ldots = \rho_{i-1} = \rho_i = \rho_{i+1} = \ldots = \rho_n \equiv \rho_b\} \subseteq E$$

  Assume that there exists an i such that $\rho_i \equiv \rho$ and $\rho_{i+1} \equiv \rho$. We can shorten the chain as follows:

  $$\{\rho_a \equiv \rho_1 = \rho_2 = \ldots = \rho_{i-1} = \rho_{i+1} = \ldots = \rho_n \equiv \rho_b\} \subseteq E$$

If we repeat this until we obtain a chain in E not containing the $\rho = \rho$ equation, the result is a chain in E', which proves the lemma.

- **Case** $[REG]$

  Assume k such that $\varphi'_k(\rho') \supseteq \{\rho'_a, \rho'_b\}$, where $\rho'_a \not\equiv \rho'_b$. There must exist some RTVs $\rho_x, \rho_y, \rho_a, \rho_b$ such that $\rho_a \not\equiv \rho_b$ and:

  $$\begin{aligned} [\rho_1 \mapsto \rho_2](\rho_x) &= \rho' \\ [\rho_1 \mapsto \rho_2](\rho_y) &= \rho' \\ [\rho_1 \mapsto \rho_2](\rho_a) &= \rho'_a \\ [\rho_1 \mapsto \rho_2](\rho_b) &= \rho'_b \end{aligned}$$

  and $\varphi_k(\rho_x) \ni \rho_a, \varphi_k(\rho_y) \ni \rho_b$. We distinguish cases:

  1. $\rho_x \equiv \rho_y$

     Since the initial configuration is good, there exists a chain:

     $$\{\rho_a \equiv \rho'_1 = \ldots = \rho'_n \equiv \rho_b\} \subseteq E$$

     By applying the $[\rho_1 \mapsto \rho_2]$ substitution to each element, we get:

     $$\{[\rho_1 \mapsto \rho_2](\rho_a) \equiv [\rho_1 \mapsto \rho_2](\rho'_1) = \ldots = [\rho_1 \mapsto \rho_2](\rho'_n) \equiv [\rho_1 \mapsto \rho_2](\rho_b)\} \subseteq [\rho_1 \mapsto \rho_2](E)$$

     which is equivalent to:

     $$\{\rho'_a \equiv [\rho_1 \mapsto \rho_2](\rho'_1) = \ldots = [\rho_1 \mapsto \rho_2](\rho'_n) \equiv \rho'_b\} \subseteq [\rho_1 \mapsto \rho_2](\bigcup_{i=1}^{n} E_i \cup E)$$

     which proves the required result.

  2. $\rho_x \not\equiv \rho_y$

     In this case one of the ρ_x, ρ_y must be equal to ρ_1. Otherwise we would get:

     $$\rho_x \equiv [\rho_1 \mapsto \rho_2](\rho_x) \equiv \rho' \equiv [\rho_1 \mapsto \rho_2](\rho_y) \equiv \rho_y$$

     which contradicts the fact that ρ_x and ρ_y are distinct. Without loss of generality, let us assume that $\rho_x \equiv \rho_1$ (the case $\rho_x \equiv \rho_2$ is symmetrical). Then $\rho' \equiv \rho_2$, and ρ_y must be either ρ_1 or ρ_2 (otherwise $\rho' \equiv \rho_2 \not\equiv [\rho_1 \mapsto \rho_2](\rho_y)$, which leads to a contradiction). Since ρ_y is distinct from ρ_x (which is equal to ρ_1), the only possibility is $\rho_y \equiv \rho_2$. Summarizing, we get:

     $$\varphi_k(\rho_x) \ni \rho_a, \text{ which is equivalent to } \varphi_k(\rho_1) \ni \rho_a$$

     $$\varphi_k(\rho_y) \ni \rho_b, \text{ which is equivalent to } \varphi_k(\rho_2) \ni \rho_b$$

     By one of the premises of the $[REG]$ rule, there exist some ρ_c, ρ_d with $\varphi_k(\rho_1) \ni \rho_c$ and $\varphi_k(\rho_2) \ni \rho_d$ such that the equation $[\rho_1 \mapsto \rho_2](\rho_c = \rho_d)$ belongs to E'. Notice that these ρ_c, ρ_d may or may not be equal to the region types ρ_a, ρ_b. Thus there are four possibilities:

– $\rho_c \equiv \rho_a$ and $\rho_d \equiv \rho_b$. We build the chain:

$$\{\rho'_a \equiv [\rho_1 \mapsto \rho_2](\rho_a) = [\rho_1 \mapsto \rho_2](\rho_b) \equiv \rho'_b\} \subseteq E'$$

– $\rho_c \not\equiv \rho_a$ and $\rho_d \equiv \rho_b$. In this case $\varphi_k(\rho_1) \supseteq \{\rho_a, \rho_c\}$. Since the initial state is good, there exists a chain:

$$\{\rho_a \equiv \rho_1^{(0)} = \ldots = \rho_{n_1}^{(0)} \equiv \rho_c\} \subseteq E$$

to which the substitution $[\rho_1 \mapsto \rho_2]$ can be applied:

$$\{\rho'_a \equiv [\rho_1 \mapsto \rho_2](\rho_1^{(0)}) = \ldots = [\rho_1 \mapsto \rho_2](\rho_{n_1}^{(0)}) \equiv [\rho_1 \mapsto \rho_2]\rho_c\} \subseteq [\rho_1 \mapsto \rho_2](E)$$

By using the equation $[\rho_1 \mapsto \rho_2](\rho_c = \rho_d)$, we get the required result:

$$\{\rho'_a \equiv [\rho_1 \mapsto \rho_2](\rho_1^{(0)}) = \ldots = [\rho_1 \mapsto \rho_2](\rho_c) = [\rho_1 \mapsto \rho_2](\rho_d)\} \subseteq E' \tag{A.2}$$

and the lemma follows since $[\rho_1 \mapsto \rho_2](\rho_d) = [\rho_1 \mapsto \rho_2](\rho_b) = \rho'_b$.

– $\rho_c \equiv \rho_a$ and $\rho_d \not\equiv \rho_d$. Similarly to the previous case, we obtain $\varphi_k(\rho_2) \supseteq \{\rho_b, \rho_d\}$ and the following chain:

$$\{\rho_d \equiv \rho_1^{(1)} = \ldots = \rho_{n_2}^{(1)} \equiv \rho_b\} \subseteq E$$

By applying the above mentioned substitution and equation, we get:

$$\{[\rho_1 \mapsto \rho_2](\rho_c) = [\rho_1 \mapsto \rho_2](\rho_d) = \ldots = [\rho_1 \mapsto \rho_2](\rho_{n_2}^{(1)}) \equiv \rho'_b\} \subseteq E' \tag{A.3}$$

and the lemma follows since $[\rho_1 \mapsto \rho_2](\rho_c) = [\rho_1 \mapsto \rho_2](\rho_a) = \rho'_a$.

– $\rho_c \not\equiv \rho_a$ and $\rho_b \not\equiv \rho_d$. We obtain the desired result by joining the chains (A.2) and (A.3).

□

The next step is to prove that a $\Longrightarrow^*$ derivation builds incrementally a solution to the system of constraints given in the initial configuration. Before this, we need the following auxiliary results, whose proof is a direct consequence of Definition A.1.

Lemma A.4. *The $\vDash$ relation satisfies the following properties:*

1. $\langle \theta, \overline{\varphi_i} \rangle \vDash \theta'(E) \Leftrightarrow \langle \theta \circ \theta', \overline{\varphi_i} \rangle \vDash E$
2. $\langle \theta, \overline{\varphi_i} \rangle \vDash E \Rightarrow \langle \theta, \overline{\varphi'_i \cup \varphi_i} \rangle \vDash E$

Proof. Both facts follow trivially from the definition of $\vDash$. □

Lemma A.5. *If $E, \langle \theta_0, \overline{\varphi_{i,0}}^p \rangle \Longrightarrow E', \langle \theta, \overline{\varphi_i}^p \rangle$ then, for some θ':*

1. $\theta = \theta' \circ \theta_0$
2. $\theta'(\varphi_{i,0}) \subseteq \varphi_i$ *for all* $i \in \{1..p\}$.

Proof. By simple inspection of the $\Longrightarrow$ rules. □

Lemma A.6. *Given the following transition:*

$$E, \langle \theta_0, \overline{\varphi_{i,0}} \rangle \Longrightarrow E', \langle \theta \circ \theta_0, \overline{\varphi_i} \rangle$$

If $\langle \theta', \overline{\varphi_i'} \rangle$ *is a solution to* E'*, then* $\langle \theta' \circ \theta, \overline{\varphi_i' \cup \theta'(\varphi_i)} \rangle$ *is a solution to* E*.*

Proof. By case distinction on the $\Longrightarrow$ rule applied. We show only the most relevant cases. The remaining ones are either straightforward or similar.

- **Case** $[QVAR\text{-}L]$

  Let us assume $s' = \mathit{freshReg}(s)$. By hypothesis, $\langle \theta', \overline{\varphi_i'} \rangle \vDash \{s' \approx_k s\} \cup [\alpha \mapsto s'](E)$. Therefore:

$$
\begin{array}{ll}
 & \varphi_k'(\theta'(s')) \ni \theta'(s) \ \wedge \ \langle \theta', \overline{\varphi_i'} \rangle \vDash [\alpha \mapsto s'](E) \\
\Rightarrow & \quad \{ \text{ since } \alpha \notin \mathit{fv}(s) \} \\
 & \varphi_k'((\theta' \circ [\alpha \mapsto s'])(\alpha)) \ni (\theta' \circ [\alpha \mapsto s'])(s) \ \wedge \ \langle \theta', \overline{\varphi_i'} \rangle \vDash [\alpha \mapsto s'](E) \\
\Rightarrow & \quad \{ \text{ by Lemma A.4 (1) } \} \\
 & \varphi_k'((\theta' \circ [\alpha \mapsto s'])(\alpha)) \equiv (\theta' \circ [\alpha \mapsto s'])(s) \ \wedge \ \langle \theta' \circ [\alpha \mapsto s'], \overline{\varphi_i'} \rangle \vDash E \\
\Rightarrow & \quad \{ \text{ by definition of } \vDash \} \\
 & \langle \theta' \circ [\alpha \mapsto s'], \overline{\varphi_i'} \rangle \vDash \{\alpha \approx_k s\} \cup E \\
\Rightarrow & \quad \{ \text{ by Lemma A.4 (2) } \} \\
 & \langle \theta' \circ [\alpha \mapsto s'], \overline{\varphi_i' \cup \theta'(\varphi_i)} \rangle \vDash \{\alpha \approx_k s\} \cup E
\end{array}
$$

- **Case** $[QREG1]$

  Again, we start from $\langle \theta', \varphi_i' \rangle \vDash \{\rho_2 = \rho'\} \cup E$ and the precondition $\varphi_k(\rho_1) = \varphi_{k,0}(\rho_1) \ni \rho'$ of the $[QREG1]$ rule.

$$
\begin{array}{ll}
 & \theta'(\rho_2) \equiv \theta'(\rho') \ \wedge \ \langle \theta', \overline{\varphi_i'} \rangle \vDash E \ \wedge \ \varphi_k(\rho_1) \ni \rho' \\
\Rightarrow & \quad \{ \text{ by properties of substitutions } \} \\
 & \theta'(\rho_2) \equiv \theta'(\rho') \ \wedge \ \langle \theta', \overline{\varphi_i'} \rangle \vDash E \ \wedge \ (\theta'(\varphi_k))(\theta'(\rho_1)) \ni \theta'(\rho') \\
\Rightarrow & \quad \{ \text{ by the first equality } \} \\
 & \langle \theta', \overline{\varphi_i'} \rangle \vDash E \ \wedge \ (\theta'(\varphi_k))(\theta'(\rho_1)) \ni \theta'(\rho_2) \\
\Rightarrow & \quad \{ \text{ by definition of } \vDash \} \\
 & \langle \theta', \overline{\varphi_i'} \rangle \vDash E \ \wedge \ \langle \theta', \overline{\theta'(\varphi_i)} \rangle \vDash \rho_1 \approx_k \rho_2 \\
\Rightarrow & \quad \{ \text{ by Lemma A.4 (2) and definition of } \vDash \} \\
 & \langle \theta', \overline{\varphi_i' \cup \theta'(\varphi_i)} \rangle \vDash \{\rho_1 \approx_k \rho_2\} \cup E
\end{array}
$$

□

The following lemmata generalize these results to $\Longrightarrow^*$ derivations.

Lemma A.7. *Assume the derivation* $E, \langle \theta, \overline{\varphi_i} \rangle \Longrightarrow^* E', \langle \theta', \overline{\varphi_i'} \rangle$*. If the initial configuration is good, then so is the final configuration.*

Proof. By induction on the length of the $\Longrightarrow^*$ derivation, and by Lemma A.3. □

Lemma A.8. *If* $E, \langle \theta_0, \overline{\varphi_{i,0}}^p \rangle \Longrightarrow^* E', \langle \theta, \overline{\varphi_i}^p \rangle$ *then, for some* θ_1*:*

1. $\theta = \theta_1 \circ \theta_0$
2. $\theta_1(\varphi_{i,0}) \subseteq \varphi_i$ *for all* $i \in \{1..p\}$*.*

Proof. By induction on the length of the $\Longrightarrow^*$ derivation. If the length is zero we can prove the lemma by choosing $\theta_1 = \mathit{id}$. If the $\Longrightarrow^*$ derivation is of length greater than zero, we can split it as follows:

$$E, \langle \theta_0, \overline{\varphi_{i,0}}^p \rangle \Longrightarrow E_1, \langle \theta_1, \overline{\varphi_{i,1}}^p \rangle \Longrightarrow^* E', \langle \theta, \overline{\varphi_i}^p \rangle$$

By induction hypothesis we obtain:

$$\theta = \theta' \circ \theta_1 \quad \wedge \quad \overline{\theta'(\varphi_{i,1}) \subseteq \varphi_i}^{\,p} \quad \text{for some } \theta'$$

and, by applying Lemma A.5 in the first step:

$$\theta_1 = \theta'' \circ \theta_0 \quad \wedge \quad \overline{\theta''(\varphi_{i,0}) \subseteq \varphi_{i,1}}^{\,p} \quad \text{for some } \theta''$$

Hence we get:

$$\theta = (\theta' \circ \theta'') \circ \theta_1 \quad \wedge \quad \overline{(\theta' \circ \theta'')(\varphi_{i,1}) \subseteq \theta'(\varphi_{i,1}) \subseteq \varphi_i}^{\,p}$$

which proves the lemma. □

Lemma A.9. *Given the following derivation:*

$$E, \langle \theta_0, \overline{\varphi_{i,0}} \rangle \Longrightarrow^* E', \langle \theta \circ \theta_0, \overline{\varphi_i} \rangle$$

If $\langle \theta', \overline{\varphi'_i} \rangle$ *is a solution to* E'*, then* $\langle \theta' \circ \theta, \overline{\varphi'_i \cup \theta'(\varphi_i)} \rangle$ *is a solution to* E*.*

Proof. By induction on the length n of the $\Longrightarrow^*$ derivation.

- **Case** $n = 0$

  We get $\theta = id$ and, by hypothesis, $\langle \theta', \overline{\varphi'_i} \rangle \vDash E$, which implies $\langle \theta', \overline{\varphi'_i \cup \theta'(\varphi_i)} \rangle \vDash E$ by Lemma A.4 (2).

- **Case** $n > 0$

  Let us split the derivation as follows:

$$E, \langle \theta_0, \overline{\varphi_{i,0}} \rangle \Longrightarrow E_1, \langle \theta_1 \circ \theta_0, \overline{\varphi_{i,1}} \rangle \Longrightarrow^* E', \langle \underbrace{\theta_2 \circ \theta_1}_{\theta} \circ \theta_0, \overline{\varphi_i} \rangle$$

  We obtain:

$$\begin{array}{ll} & \langle \theta', \overline{\varphi'_i} \rangle \vDash E' \\ \Rightarrow & \quad \{ \text{ by i.h. } \} \\ & \langle \theta' \circ \theta_2, \overline{\varphi'_i \cup \theta'(\varphi_i)} \rangle \vDash E_1 \\ \Rightarrow & \quad \{ \text{ by applying Lemma A.6 on the first step } \} \\ & \langle \theta' \circ \theta_2 \circ \theta_1, \overline{\varphi'_i \cup \theta'(\varphi_i) \cup (\theta' \circ \theta_2)(\varphi_{i,1})} \rangle \vDash E \\ \Rightarrow & \quad \{ \text{ by Lemma A.8, } \forall i \in \{1..p\}.\theta_2(\varphi_{i,1}) \subseteq \varphi_i \} \\ & \langle \theta' \circ \theta_2 \circ \theta_1, \overline{\varphi'_i \cup \theta'(\varphi_i)} \rangle \vDash E \end{array}$$

□

Finally, we prove that if we start applying the $\Longrightarrow^*$ rules on a set of constraints E with the empty solution and we reach a configuration with no equations to be solved, the final result is a solution to E, not only in the sense of Definition A.1, but also in the sense of Definition 4.5.

Corollary A.10 (Soundness)**.** *Assume a set* E *of constraints and* $\theta, \overline{\varphi_i}^p$ *such that the following judgment can be derived from the rules of Figure A.1:*

$$E, \langle id, \overline{[\,]}^p \rangle \Longrightarrow^* \emptyset, \langle \theta, \overline{\varphi_i}^p \rangle$$

Then:

1. *For each $i \in \{1..p\}$, φ_i is a function from RTVs to RTVs.*
2. *$\langle \theta, \overline{\varphi_i}^p \rangle$ is a solution to E.*

Proof. The first configuration is trivially good. By Lemma A.7, the final configuration is also good. Now let us prove (1) by contradiction: assume that there exists some φ_k not being a function. Then there exist ρ, ρ_a, ρ_b such that $\varphi(\rho) \ni \{\rho_a, \rho_b\}$ and $\rho_a \not\equiv \rho_b$. Since the final configuration is good, there must exist a chain of the form:

$$\{\rho_a \equiv \rho_1 = \ldots = \rho_m \equiv \rho_b\} \subseteq \emptyset$$

Which is contradictory, since the chain must have at least one equation. Therefore our assumption was false, and every $\overline{\varphi_i}$ is a function. The result (2) is a particular instance of Lemma A.9. □

Lemma A.11 (Termination). *Given a finite configuration $E, \langle \theta, \overline{\varphi_i} \rangle$, there does not exist an infinite $\Longrightarrow^*$ derivation:*

$$E, \langle \theta, \overline{\varphi_i} \rangle \Longrightarrow E_1, \langle \theta_1, \overline{\varphi_{1,i}} \rangle \Longrightarrow \ldots \Longrightarrow E_n, \langle \theta_n, \overline{\varphi_{n,i}} \rangle \Longrightarrow \ldots$$

Proof. Let us define the *size* function $\| \cdot \| : \mathbf{Type} \to \mathbb{N}$ as follows:

$$\begin{array}{l} \|\alpha\| = \|\rho\| = \|B\| = 1 \\ \|T\ \overline{s_i}^n\ @\ \overline{\rho_j}^m\| = \|T\ \overline{s_i}^n\ !@\ \overline{\rho_j}^m\| = \|T\ \overline{s_i}^n\ \#@\ \overline{\rho_j}^m\| = 1 + \sum_{i=1}^n \|s_i\| + \sum_{j=1}^m \|\rho_j\| \\ \|\overline{t_i}^n \to \overline{\rho_j}^m \to s\| = 1 + \sum_{i=1}^n \|t_i\| + \sum_{j=1}^m \|\rho_j\| \end{array}$$

The domain of this function can be easily extended to constraints:

$$\begin{array}{l} \|\tau_1 = \tau_2\| = \|\tau_1\| + \|\tau_2\| \\ \|s_1 = \mathit{isData}(s_2, \rho)\| = \|s_1\| + \|s_2\| + \|\rho\| \\ \|\tau_1 \approx_k \tau_2\| = \|\tau_1\| + \|\tau_2\| \end{array}$$

Let Φ be a function mapping a configuration $E, \langle \theta, \overline{\varphi_i} \rangle$ to a tuple $(N_\alpha, N_{\mathit{isData}}, N_\rho, s, N_\approx)$ such that:

- N_α is the number of *distinct* type variables occurring in E.
- N_{isData} is the number of *isData* constraints in E.
- N_ρ is the number of *distinct* RTVs occurring in E and in the $\overline{\varphi_i}$.
- s is the sum of the sizes of the constraints in E.
- $N_\approx$ is the number of $\approx_k$-constraints in E.

Then, if $E, \langle \theta, \overline{\varphi_i} \rangle \Longrightarrow E', \langle \theta', \overline{\varphi_i'} \rangle$, Figure A.3 shows that $\Phi(E, \langle \theta, \overline{\varphi_i} \rangle) > \Phi(E', \langle \theta', \overline{\varphi_i'} \rangle)$, where $>$ is the lexicographical order in $\mathbb{N}^5$. If we apply Φ to every configuration in a infinite $\Longrightarrow^*$ derivation we would obtain a infinite $>$ chain in $\mathbb{N}^5$, which is not possible since $(\mathbb{N}^5, \leq)$ is a well-founded set.

□

Rule	N_α	N_{isData}	N_ρ	s	$N_\approx$
$[ID]$	$\leq$	$=$	$\leq$	$<$	$=$
$[VAR]$	$<$	$=$	$=$	$\geq$	$=$
$[REG]$	$=$	$=$	$<$	$\geq$	$=$
$[FUN]$	$=$	$=$	$=$	$<$	$=$
$[CONS]$	$=$	$=$	$=$	$<$	$=$
$[ISD1]$	$=$	$<$	$>$	$>$	$=$
$[ISD2]$	$=$	$<$	$=$	$<$	$=$
$[ISD3]$	$=$	$<$	$=$	$<$	$=$
$[QFUN]$	$=$	$=$	$=$	$<$	$>$
$[QCONS]$	$=$	$=$	$=$	$<$	$>$
$[QVAR\text{-}L]$	$<$	$=$	$>$	$\geq$	$=$
$[QVAR\text{-}R]$	$<$	$=$	$>$	$\geq$	$=$
$[QREG1]$	$=$	$=$	$=$	$=$	$<$
$[QREG2]$	$=$	$=$	$=$	$=$	$<$
$[QEND]$	$=$	$=$	$=$	$=$	$<$

Figure A.3: Variation of each component in the Φ function when applying each unification rule. A $<$ sign in a cell means that the value of the corresponding component in the size of the final configuration is strictly lower than in the size of the initial one. Similarly with $\leq$, $=$, $\geq$ and $>$. Notice that, considering the lexicographical order $(N_\alpha, N_{isData}, N_\rho, s, N_\approx)$, the $\Longrightarrow$ relation always gives a configuration of lower size than the size of the initial one.

Appendix B

Correctness of the initial bounds

In Section 7.5 we introduced three algorithms for computing an initial signature (Δ, μ, σ) in order to bound the stack and memory needs of a given recursive function definition. In that section we were concerned about the reductivity of each of the components of the signature, but we did not state explicitly that these components are correct bounds to the actual runtime figures (δ, m, s). This statement was intentionally left out, since we had shown in Section 7.5.1 the fact of an upper bound being reductive was a sufficient condition to its correctness.

Nevertheless, reductivity is not a *necessary* condition to obtain correct bounds. It was pointed up in Section 7.5.6 that reductivity can only be guaranteed under some admissibility conditions on the externally given bounds to the size of the call tree (*nb*, *nr*, and *len*). These conditions were formalised in Definitions 7.41, 7.42 (page 245), and 7.47 (page 250). When these conditions do not hold, the results of the algorithms computing the initial bounds may not be reductive, but they still would be useful if we are able to show their correctness. This shows the need for an *explicit* proof of the correctness properties of the results of *computeDelta*, *computeMu*, and *computeSigma*, without basing our proof on reductivity arguments.

In this Appendix we aim to show the following fact: If $(\Delta_0, \mu_0, \sigma_0)$ are initial approximations to the memory needs of f, they have been computed with the algorithms of Section 7.5, and the *nb*, *nr*, and *len* functions are *correct* approximations to the corresponding runtime figures of the call tree, then $(\Delta_0, \mu_0, \sigma_0)$ is a correct signature with respect to f. The proof of this statement is very technical, since it requires several extensions to the resource-aware big-step semantics defined in Figure B.1.

Remark. Throughout this section we assume the absence of explicit destruction via **case**! expressions. We also assume, by convention, that $\max\ \emptyset = 0$.

B.1 Before/after semantics with call tree counters

In this section we will make precise the idea of the *nb*, *nr*, and *len* functions being correct approximations to the actual parameters of the call tree deployed at runtime. This is done by extending our semantic judgements with a triple (N_b, N_r, L) of natural numbers representing the number of base calls, recursive calls, and the maximum depth of the call tree. Obviously, it makes little sense to talk about recursive calls, if we do not specify *which* function do these numbers refer to. Therefore, the name of this function will be attached to the arrow of these judgements, as in $\Downarrow_f$.

Besides this, it turns out to be useful to distinguish between the charges done in memory before the last recursive call in the current context, and those done after this call. For this reason, we shall split

each of the δ and m components of our resource vector into two subcomponents: their *before* part, (δ_b and m_b), and their *after* part (δ_a and m_a). The former takes into account the expressions being executed before (and including) the last recursive call in the current call context, and the latter takes into account the expressions being executed after (and excluding) that recursive call. By convention, we assume that, if the evaluation of an expression does not contain any recursive call, all its charges are stored in the after part, and the before part is left with no charges.

Given the above, our extended semantics defines the derivation of judgements of the following form:

$$E \vdash h, k, e \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)$$

We omit the s component of the resource vector, since no distinction between the before and after part is necessary for that component, as we shall see later. The semantic rules are shown in Figure B.1. The $[Basic_{NC}]$ rule is used when the expression being evaluated does not contain recursive calls to f. In this case the evaluation can be described by our usual $\Downarrow$ rules of Figure 2.26. We follow our convention mentioned above, and we consider their charges in the after part. The opposite situation is that of rule $[AppR_{NC}]$, where all the charges are added to the before component. The number of recursive calls N_r, and the length of the call-tree L are incremented accordingly. When evaluating a **let** expression we have to distinguish whether a recursive call to f is done during the evaluation of e_2. This can be done with the help of the L_2 component of the call tree counter returned by the evaluation of e_2. If there are no recursive calls in e_2 we obtain $L_2 = 1$ (i.e. the depth of the call tree is 1), and we apply $[Let1_{NC}]$. In this case, the before part corresponds to the before part of the evaluation of e_1, whereas the after part comprises the after part of e_1 and the whole evaluation of e_2. The $[Let2_{NC}]$ rule is used when there are recursive calls in e_2 (that is, $L_2 > 1$). The before part includes the evaluation of e_1 and the before part of e_2, whereas the after part only contains the after part of e_2. In this case, the formalization of the (N_r, N_b, L) requires further case distinction, and we define in Figure B.2 a separate operator $\oplus$ for this purpose. The $[Case_{NC}]$ rule just propagates the before/after information, and the call tree counters given by the branch being executed.

From these semantic rules it is easy to show that the new components introduced are just counters, and they do not influence the evaluation of the expression. As a consequence, if we can execute an expression e under the $\Downarrow$ semantics for a given environment E and heap h with k regions, we can do the same under the $\Downarrow_f$ semantics (for any f), and we will get the same normal form v and final heap h'. A little less obvious is the relation between the $(\delta_b/\delta_a, m_b/m_a)$ components of the $\Downarrow_f$ semantics, and the original resource vector (δ, m, s) given by the corresponding $\Downarrow$-judgement. The whole δ component is equivalent to the (region-wise) addition of the before and after parts. With respect to the m component, an intuitive idea of its relation with the m_b and m_a is given by Figure 2.30. The following lemma states these results formally.

Lemma B.1. *Let us assume an evaluation $E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ under a signature environment Σ. Given a function $f \in dom\ \Sigma$, there exists a single tuple (N_b, N_r, L) and a single tuple $(\delta_b/\delta_a, m_b/m_a)$ such that $E \vdash h, k, e \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)$. Moreover, it holds that:*

1. $N_b \geq 1$, $N_r \geq 0$, *and* $L \geq 1$.
2. *If* $L = 1$, *then* $\delta_b = [\,]_k$ *and* $m_b = 0$.
3. $\delta = \delta_b + \delta_a$.
4. $m = \max\{m_b, |\delta_b| + m_a\}$.

$$\frac{\begin{array}{c}E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s) \\ e \text{ is of the form } c, x, a_1 \oplus a_2, x\,@\,r, C\,\overline{a_i}\,@\,r \text{ or } g\,\overline{a_i}\,@\,\overline{r_j} \text{ with } g \neq f\end{array}}{E \vdash h, k, e \Downarrow_f h', k, v, (1,0,1), ([\,]_k/\delta, 0/m)}\ [Basic_{NC}]$$

$$\frac{\begin{array}{c}(f\ \overline{y_i}^n\ @\ \overline{r_j}^l = e_f) \in \Sigma \\ [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^l, self \mapsto k+1] \vdash h, k+1, e_f \Downarrow_f h', k+1, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)\end{array}}{E \vdash h, k, f\ \overline{a_i}^n\ @\ \overline{r_j}^l \Downarrow_f h'\ |_k, k, v, (N_b, N_r+1, L+1), ((\delta_b+\delta_a)|_k/[\,]_k, \max\{m_b, |\delta_b|+m_a\}/0)}\ [AppR_{NC}]$$

$$\frac{\begin{array}{c}L_2 = 1 \\ E \vdash h, k, e_1 \Downarrow_f h', k, v_1, (N_{b,1}, N_{r,1}, L_1), (\delta_{b,1}/\delta_{a,1}, m_{b,1}/m_{a,1}) \\ E \cup [x_1 \mapsto v_1] \vdash h', k, e_2 \Downarrow_f h'', k, v, (N_{b,2}, N_{r,2}, L_2), (\delta_{b,2}/\delta_{a,2}, m_{b,2}/m_{a,2})\end{array}}{E \vdash h, k, \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \Downarrow_f h'', k, v, (N_{b,1}, N_{r,1}, \max\{L_1, L_2\}), (\delta_{b,1}/\delta_{a,1} + \delta_{a,2}, m_{b,1}/\max\{m_{a,1}, |\delta_{a,1}| + m_{a,2}\})}\ [Let1_{NC}]$$

$$\frac{\begin{array}{c}L_2 \neq 1 \qquad m_b = \max\{m_{b,1}, |\delta_{b,1}| + m_{a,1}, |\delta_{b,1}| + |\delta_{a,1}| + m_{b,2}\} \\ (N_b, N_r, L) = (N_{b,1}, N_{r,1}, L_1) \otimes (N_{b,2}, N_{r,2}, L_2) \\ E \vdash h, k, e_1 \Downarrow_f h', k, v_1, (N_{b,1}, N_{r,1}, L_1), (\delta_{b,1}/\delta_{a,1}, m_{b,1}/m_{a,1}) \\ E \cup [x_1 \mapsto v_1] \vdash h', k, e_2 \Downarrow_f h'', k, v, (N_{b,2}, N_{r,2}, L_2), (\delta_{b,2}/\delta_{a,2}, m_{b,2}/m_{a,2})\end{array}}{E \vdash h, k, \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \Downarrow_f h'', k, v, (N_b, N_r, L), (\delta_{b,1} + \delta_{a,1} + \delta_{b,2}/\delta_{a,2}, m_b/m_{a,2})}\ [Let2_{NC}]$$

$$\frac{\begin{array}{c}C = C_r \qquad E(x) = p \\ E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, e_r \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)\end{array}}{E \vdash h[p \mapsto (j, C\ \overline{v_i}^{n_r})], k, \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n} \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)}\ [Case]$$

Figure B.1: Big-step semantics enriched with components for counting the number of recursive calls, and for distinguishing the charges done before and after the last recursive call.

$$(N_{b,1}, N_{r,1}, L_1) \otimes (N_{b,2}, N_{r,2}, L_2) = (N_b, N_r, \max\{L_1, L_2\})$$

$$\textbf{where } (N_b, N_r) = \begin{cases} (N_{b,2}, N_{r,2}) & \text{if } N_{r,1} = 0 \wedge N_{r,2} \neq 0 \\ (N_{b,1}, N_{r,1}) & \text{if } N_{r,2} = 0 \\ (N_{b,1} + N_{b,2}, N_{r,1} + N_{r,2} - 1) & \text{otherwise} \end{cases}$$

Figure B.2: Definition of $\otimes$ operator on (N_b, N_r, L) tuples.

Proof. The existence and uniqueness of the respective tuples can be easily shown by induction on the size of the $\Downarrow$ derivation. They are direct consequence of the semantic rules being deterministic. Hence, let us prove (1), (2), (3), and (4) by induction on the $\Downarrow$ derivation. Let us distinguish cases according to the last $\Downarrow$-rule applied. Cases [*Lit*], [*Var*], [*PrimOp*], [*Copy*], [*Cons*], and [*App*] when the function being applied is different from f, are trivial. In the case of rules [*Case*], and [*App*] when the function being applied is f, the result follows trivially from the induction hypothesis. The only subtlety when proving (3) is related to the restriction of the domain of δ occurring in [*App*], but we get:

$$\delta|_k = (\delta_a + \delta_b)|_k = \delta_a|_k + \delta_b|_k$$

The [*Let*] case is more involved. We assume the existence of the tuples $(N_{b,1}, N_{r,1}, L_1)$ and $(\delta_{b,1}/\delta_{a,1}, m_{b,1}/m_{a,1})$ corresponding to the $\Downarrow$-evaluation of e_1, and the tuples $(N_{b,2}, N_{r,2}, L_2)$ and $(\delta_{b,2}/\delta_{a,2}, m_{b,2}/m_{a,2})$ regarding the $\Downarrow$-evaluation of e_2. By induction hypothesis, we get:

$$N_{b,i} \geq 1 \qquad N_{r,i} \geq 0 \qquad L_i \geq 1 \quad \text{for } i \in \{1,2\} \tag{B.1}$$

$$\delta_i = \delta_{b,i} + \delta_{a,i} \quad \text{for } i \in \{1,2\} \tag{B.2}$$

$$m_i = \max\{m_{b,i}, |\delta_{b,i}| + m_{a,i}\} \quad \text{for } i \in \{1,2\} \tag{B.3}$$

where δ_i and m_i are the figures of the resource vectors occurring in each $\Downarrow$-subderivation. Now we distinguish cases:

- **Case** $L_2 = 1$. Then $N_b = N_{b,1}$, $N_r = N_{r,1}$, and $L = \max\{L_1, L_2\}$. The fact (1) follows from equation (B.1). With regard to (2), if $L = 1$, then $L_1 = 1$, which implies $\delta_{b,1} = [\,]|_k$ and $m_{b,1} = 0$, so that property holds. Moreover, by induction hypothesis, we get $\delta_{b,2} = [\,]_k$ and $m_{b,2} = 0$, since $L_2 = 1$. Therefore:

$$\begin{array}{lll} & \delta_b + \delta_a & \\ = & \delta_{b,1} + \delta_{a,1} + \delta_{a,2} & \\ = & \delta_{b,1} + \delta_{a,1} + \delta_{b,2} + \delta_{a,2} & \{ \text{ since } \delta_{b,2} = [\,]_k \} \\ = & \delta_1 + \delta_2 & \{ \text{ by (B.2) } \} \\ = & \delta & \end{array}$$

$$\begin{array}{lll} & \max\{m_b, |\delta_b| + m_a\} & \\ = & \max\{m_{b,1}, |\delta_{b,1}| + \max\{m_{a,1}, |\delta_{a,1}| + m_{a,2}\}\} & \\ = & \max\{m_{b,1}, |\delta_{b,1}| + m_{a,1}, |\delta_{b,1}| + |\delta_{a,1}| + m_{a,2}\} & \{ \text{ since } \delta_{b,2} = [\,]_k \} \\ = & \max\{m_1, |\delta_1| + m_{a,2}\} & \{ \text{ by (B.2) and (B.3) } \} \\ = & \max\{m_1, |\delta_1| + \max\{m_{b,2}, |\delta_{b,2}| + m_{a,2}\}\} & \{ \text{ since } m_{a,2} \geq 0, m_{b,2} = 0 \text{and } |\delta_{b,2}| = 0 \} \\ = & \max\{m_1, |\delta_1| + m_2\} & \{ \text{ by (B.3) } \} \end{array}$$

- **Case** $L_2 > 1$. We get $(N_b, N_r, L) = (N_{b,1}, N_{r,1}, L_1) \otimes (N_{b,2}, N_{r,2}, L_2)$. From induction hypothesis

it follows that $N_b \geq 1$ and $L \geq 1$. With respect to N_r, the only caution to take is the case when $N_r = N_{r,1} + N_{r,2} - 1$, but this only holds when both $N_{r,1}$ and $N_{r,2}$ are strictly positive, so $N_r \geq 1$, and (1) holds. Property (2) holds vacuously, since $L_2 > 1$ implies $L > 1$. Now we prove (3) and (4):

$$\begin{array}{lll} & \delta_b + \delta_a & \\ = & \delta_{b,1} + \delta_{a,1} + \delta_{b,2} + \delta_{a,2} & \\ = & \delta_1 + \delta_2 & \{ \text{ by (B.2) } \} \\ = & \delta & \end{array}$$

$$\begin{array}{lll} & \max\{m_b, |\delta_b| + m_a\} & \\ = & \max\{m_{b,1}, |\delta_{b,1}| + m_{a,1}, |\delta_{b,1}| + |\delta_{a,1}| + m_{b,2}, |\delta_{b,1}| + |\delta_{a,1}| + |\delta_{b,2}| + m_{a,2}\} & \\ = & \max\{m_1, |\delta_1| + m_{b,2}, |\delta_1| + |\delta_{b,2}| + m_{a,2}\} & \{ \text{ by (B.2) and (B.3) } \} \\ = & \max\{m_1, |\delta_1| + \max\{m_{b,2}, |\delta_{b,2}| + m_{a,2}\}\} & \\ = & \max\{m_1, |\delta_1| + m_2\} & \end{array}$$

□

In the following we shall use the letters φ, ψ, χ, etc. to denote $\Downarrow$-judgements. Given one of these:

$$\varphi \equiv E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$$

we have shown that, given a function f, there exist unique numbers N_b, N_r, L, m_b, and m_a, and unique mappings δ_a, δ_b such that $E \vdash h, k, e \Downarrow_f h', k, v, (N_b, N_r, L), (\delta_b/\delta_a, m_b/m_a)$ can be derived. All these components are determined by the judgement φ, so we can use the notation $N_b^f(\varphi)$, $N_r^f(\varphi)$, $L^f(\varphi)$, $\delta_b^f(\varphi)$, and $\delta_a^f(\varphi)$ to refer to these components. By abuse of notation, we use $\delta(\varphi)$, $m(\varphi)$, $s(\varphi)$ for denoting, respectively, the δ, m, and s components of the resource vector occurring in the judgement φ. Moreover, we use $Exp(\varphi)$ for denoting the expression being evaluated in φ.

Our abstract interpretation functions $[\![\cdot]\!]_\Delta$ and $[\![\cdot]\!]_\mu$ defined in Section 7.3 were defined in terms of sequences of basic expressions. The following lemma establishes the relation of these sequences to the before and after parts of the evaluation of an expression. It also presents some useful properties of the call tree.

Lemma B.2. *Assume the following judgement:*

$$\varphi \equiv E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$$

There exists a sequence $[G \to be_1, \ldots, be_n] \in seqs\ e$ and some $\overline{\varphi_i}^n$, $\overline{E_i}^n$, $\overline{h_i}^n$, $\overline{td_i}^n$, $\overline{h'_i}^n$, $\overline{v_i}^n$, $\overline{\delta_i}^n$, $\overline{m_i}^n$, $\overline{s_i}^n$ such that:

1. *For every $i \in \{1..n\}$, $\varphi_i \equiv E_i \vdash h_i, k, td_i, be_i \Downarrow h'_i, k, v_i, (\delta_i, m_i, s_i)$, and φ_i belongs to the derivation of φ.*

2. *Let $I \subseteq \{1..n\}$ be the set of indices i such that be_i has the form $f\ \overline{a_i}$ @ $\overline{r_j}$ for some f, $\overline{a_i}$ and $\overline{r_j}$ of their respective types. Then:*

$$N_b^f(\varphi) = \begin{cases} 1 & \text{if } I = \emptyset \\ \sum_{i \in I} N_b^f(\varphi_i) & \text{otherwise} \end{cases}$$

$$N_r^f(\varphi) = \begin{cases} 0 & \text{if } I = \emptyset \\ 1 + \sum_{i \in I} (N_r^f(\varphi_i) - 1) & \text{otherwise} \end{cases}$$

$$L^f(\varphi) = \begin{cases} 1 & \text{if } I = \emptyset \\ 1 + \max_{i \in I}\{L^f(\varphi_i) - 1\} & \text{otherwise} \end{cases}$$

3. *Being I defined as above, let $p = \max\ I$, that is, the index of the last recursive call (if any; otherwise take $p = 0$). Then:*

$$\delta_b^f(\varphi) = \sum_{i=1}^{p} \delta_i \qquad \delta_a^f(\varphi) = \sum_{i=p+1}^{n} \delta_i$$

$$m_b^f(\varphi) = \max_{i \in \{1..p\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\} \quad m_a^f(\varphi) = \max_{i \in \{p+1..n\}} \left\{ \sum_{j=p+1}^{i-1} |\delta_j| + m_i \right\}$$

$$\delta = \sum_{i=1}^{n} \delta_i \qquad m = \max_{i \in \{1..n\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\} \tag{B.4}$$

Proof. Let us start with the last equation, that is, (B.4), by assuming that the remaining facts of the third conclusion have been proved, and by using Lemma B.1:

$$\delta = \delta_b^f(\varphi) + \delta_a^f(\varphi) = \sum_{i=1}^{p} \delta_i + \sum_{i=p+1}^{n} \delta_i = \sum_{i=1}^{n} \delta_i$$

$$\begin{aligned} m &= \max\{m_b^f(\varphi), |\delta_b^f(\varphi)| + m_a^f(\varphi)\} \\ &= \max\left\{ \max_{i \in \{1..p\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\}, \sum_{i=1}^{k} \delta_i + \max_{i \in \{p+1..n\}} \left\{ \sum_{j=k+1}^{i-1} |\delta_j| + m_i \right\} \right\} \\ &= \max\left\{ \max_{i \in \{1..k\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\}, \max_{i \in \{k+1..n\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\} \right\} \\ &= \max_{i \in \{1..n\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\} \end{aligned}$$

The rest of the lemma is proven by induction on the size of the $\Downarrow$- derivation. We distinguish cases depending on the structure of e.

- **Cases** $e \equiv c$, $e \equiv x$, $e \equiv a_1 \oplus a_2$, $e \equiv x \ @\ r$, $e \equiv C\ \overline{a_i}\ @\ \overline{r}$, and $e \equiv g\ \overline{a_i}\ @\ \overline{r_j}$ with $g \neq f$

  In all these cases we get *seqs* $e = \{[e]\}$, and (1) holds trivially by taking $\varphi_1 \equiv \varphi$. Moreover, we can apply the $[Basic_{NC}]$ rule, so $N_b^f(\varphi) = 1$, $N_r^f(\varphi) = 0$, $L^f(\varphi) = 1$, and (2) follows from the fact that $I = \emptyset$ (we have a single expression in the sequence, which is not a call to f). Finally, since $p = 0$, and, again, by the $[Basic_{NC}]$ rule:

$$\delta_b^f(\varphi) = [\,]_k = \sum_{i=1}^{0} \delta_i \qquad \delta_a^f(\varphi) = \delta = \delta_1 = \sum_{i=1}^{1} \delta_i$$

$$m_b^f = 0 = \max_{i \in \emptyset} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\} \qquad m_a^f(\varphi) = m = m_1 = \max_{i \in \{1\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\}$$

  Therefore, (3) holds.

- **Case** $e \equiv f\ \overline{a_i}\ @\ \overline{r_j}$

Again, we get a sequence with a single element ($n = 1$), so (1) holds with $\varphi_1 \equiv \varphi$. In this case it holds that $I = \{1\}$, so (2) is proved as follows:

$$N_b^f(\varphi) = N_b^f(\varphi_1) \qquad N_r^f(\varphi) = 1 + (N_r^f(\varphi_1) - 1) \qquad L^f = 1 + (L^f(\varphi_1) - 1)$$

With respect to (3), we denote by ψ the judgment corresponding to the evaluation of the body of the function f, which is just above φ in the derivation tree of the latter, and by (δ_f, m_f, s_f) the resource vector resulting from its evaluation. By applying the $[AppR_{NC}]$ rule and Lemma B.1 we get:

$$\delta_b^f(\varphi) = (\delta_b^f(\psi) + \delta_a^f(\psi))|_k = \delta_f|_k = \delta_1 = \sum_{i=1}^{1} \delta_i$$

$$\delta_a^f(\varphi) = [\,]_k = \sum_{i=1}^{0} \delta_i$$

$$m_b^f = \max\{m_b^f(\psi), |\delta_b^f(\psi)| + m_a^f(\psi)\} = m_f = m_1 = \max_{i \in \{1\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\}$$

$$m_a^f(\varphi) = 0 = \max_{i \in \emptyset} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\}$$

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

  We get the following judgements belonging to φ:

$$\psi_A \equiv E \vdash h, k, 0, e_1 \Downarrow h_A, k, v_A, (\delta_A, m_A, s_A)$$

$$\psi_B \equiv E \cup [x_1 \mapsto v_A] \vdash h_A, k, td + 1, e_2 \Downarrow h_B, k, v, (\delta_B, m_B, s_B)$$

  By applying induction hypothesis, there exist two sequences $[G_1 \to be_1, \ldots, be_n] \in seqs\ e_1$ and $[G_2 \to be_1, \ldots, be_m]$ satisfying the lemma. By definition of *seqs*, the sequence

$$[G_1 \wedge G_2 \to be_1, \ldots, be_n, be'_1, \ldots, be'_m]$$

  belongs to $seqs\ e$, so let us prove that the conclusions of the lemma hold for this sequence. The first condition follows from the induction hypothesis. If we denote by $\overline{\varphi_i}^n$ the judgements belonging to ψ_A, and by $\overline{\varphi'_i}^m$ the judgements of ψ_B (the existence of both lists is justified by induction hypothesis), it is obvious that both the $\overline{\varphi_i}^n$ and $\overline{\varphi'_i}^m$ belong to the main judgement φ. Now we move on to (2). Before this, let us define, for each $j \in \{n+1, ..., n+m\}$, $be_j = be'_{j-n}$, $\varphi_j = \varphi'_i$. Let I_A be the set of indices i such that be_i is a function application, and let I'_B be defined similarly, but with the be'_i. We define $I_B = \{n + j \mid j \in I'_B\}$, and $I = I_A \cup I_B$. In this way we ensure that, for every $i \in \{1..n+m\}$, be_i is a function application if and only if $i \in I$. Now we distinguish cases in order to prove (2):

  - $I_A = \emptyset$, $I_B = \emptyset$

    This implies that both I_A and I_B (and hence I'_B) are empty. By induction hypothesis, this means that:

$$N_b^f(\psi_A) = 1 \qquad N_r^f(\psi_A) = 0 \qquad L^f(\psi_A) = 1 \tag{B.5}$$

$$N_b^f(\psi_B) = 1 \qquad N_r^f(\psi_B) = 0 \qquad L^f(\psi_B) = 1 \tag{B.6}$$

Therefore, we get, by rule $[Let1_{NC}]$:

$$N_b^f(\varphi) = N_b^f(\psi_A) = 1 \qquad N_r^f(\varphi) = N_r^f(\psi_A) = 0 \qquad L^f(\varphi) = \max\{L^f(\psi_A), L^f(\psi_B)\} = 1$$

and (2) holds.

– $I_A \neq \emptyset$, $I_B = \emptyset$

Then $I = I_A$, but the facts (B.6) also hold in this case. Again, by $[Let2_{NC}]$:

$$N_b^f(\varphi) = N_b^f(\psi_A) = \sum_{i \in I_A} N_b^f(\varphi_i) = \sum_{i \in I} N_b^f(\varphi_i)$$

$$N_r^f(\varphi) = N_r^f(\psi_A) = 1 + \sum_{i \in I_A} (N_r^f(\varphi_i) - 1) = 1 + \sum_{i \in I} (N_r^f(\varphi_i) - 1)$$

$$L^f(\varphi) = \max\{L^f(\psi_A), 1\} = L^f(\psi_A) = 1 + \max_{i \in I_A}\{L^f(\varphi_i) - 1\} = 1 + \max_{i \in I}\{L^f(\varphi_i) - 1\}$$

where, in each case, the second step is justified by induction hypothesis. Thus (2) holds.

– $I_A = \emptyset$, $I_B \neq \emptyset$

In this case we apply $[Let2_{NC}]$ in order to obtain:

$$N_b^f(\varphi) = N_b^f(\psi_B) = \sum_{i \in I'_B} N_b^f(\varphi'_i) = \sum_{i \in I_B} N_b^f(\varphi_i) = \sum_{i \in I} N_b^f(\varphi_i)$$

and similarly with $N_r^f(\varphi)$ and $L^f(\varphi)$, so (2) holds.

– $I_A \neq \emptyset$, $I_B \neq \emptyset$

Again, we apply $[Let2_{NC}]$ and get:

$$\begin{aligned} N_b^f(\varphi) &= N_b^f(\psi_A) + N_b^f(\psi_B) \\ &= \sum_{i \in I_A} N_b^f(\varphi_i) + \sum_{i \in I'_B} N_b^f(\varphi'_i) \\ &= \sum_{i \in I_A} N_b^f(\varphi_i) + \sum_{i \in I_B} N_b^f(\varphi_i) \\ &= \sum_{i \in I} N_b^f(\varphi_i) \end{aligned}$$

$$\begin{aligned} N_r^f(\varphi) &= N_r^f(\psi_A) + N_r^f(\psi_B) - 1 \\ &= 1 + \sum_{i \in I_A} (N_r^f(\varphi_i) - 1) + 1 + \sum_{i \in I'_B} (N_r^f(\varphi'_i) - 1) - 1 \\ &= 1 + \sum_{i \in I_A} (N_r^f(\varphi_i) - 1) + 1 + \sum_{i \in I_B} (N_r^f(\varphi_i) - 1) - 1 \\ &= 1 + \sum_{i \in I} (N_r^f(\varphi_i) - 1) \end{aligned}$$

$$\begin{aligned} L^f(\varphi) &= \max\{L^f(\psi_A), L^f(\psi_B)\} \\ &= \max\{1 + \max_{i\in I_A}\{L^f(\varphi_i) - 1\}, 1 + \max_{i\in I'_B}\{L^f(\varphi'_i) - 1\}\} \\ &= 1 + \max\{\max_{i\in I_A}\{L^f(\varphi_i) - 1\}, \max_{i\in I_B}\{L^f(\varphi_i) - 1\}\} \\ &= 1 + \max_{i\in I}\{L^f(\varphi_i) - 1\} \end{aligned}$$

Finally, let us prove (3). For every $i \in \{1..n+m\}$, let (δ_i, m_i, s_i) be the resource vector occurring in φ_i. For every $i \in \{1..n\}$, let (δ'_i, m'_i, s'_i) be the resource vector of φ'_i. With I_A, I'_B, and I_B defined as above, we define $p_A = \max\ I_A$ and $p_B = \max\ I'_B$, so $n + p_B = \max\ I_B$. Let us distinguish cases:

- $I_B = \emptyset$

  In this case $p_B = 0$, and $L^f(\psi_B) = 1$. Therefore, $p = \max\,(I_A \cup I_B) = p_A$. We obtain, by $[Let1_{NC}]$, and the induction hypothesis:

$$\delta^f_b(\varphi) = \delta^f_b(\psi_A) = \sum_{i=1}^{p_A} \delta_i = \sum_{i=1}^{p} \delta_i$$

$$\begin{aligned} \delta^f_a(\varphi) &= \delta^f_a(\psi_A) + \delta^f_a(\psi_B) = \sum_{i=p_A+1}^{n} \delta_i + \sum_{i=p_B+1}^{m} \delta'_i = \sum_{i=p+1}^{n} \delta_i + \sum_{i=1}^{m} \delta'_i \\ &= \sum_{i=p+1}^{n} \delta_i + \sum_{i=n+1}^{n+m} \delta_i = \sum_{i=p+1}^{n+m} \delta_i \end{aligned}$$

$$m^f_b(\varphi) = m^f_b(\psi_A) = \max_{i\in\{1..p_A\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\} = \max_{i\in\{1..p\}} \left\{ \sum_{j=1}^{i-1} |\delta_j| + m_i \right\}$$

$$\begin{aligned} m^f_a(\varphi) &= \max\{m^f_a(\psi_A), |\delta^f_a(\psi_A)| + m^f_a(\psi_B)\} \\ &= \max\left\{ \max_{i\in\{k_A+1..n\}} \left\{ \sum_{j=k_A+1}^{i-1} |\delta_j| + m_i \right\}, \sum_{i=k_A+1}^{n} |\delta_i| + \max_{i\in\{k_B+1..m\}} \left\{ \sum_{j=1}^{i-1} |\delta'_j| + m'_i \right\} \right\} \\ &= \max\left\{ \max_{i\in\{k+1..n\}} \left\{ \sum_{j=k+1}^{i-1} |\delta_j| + m_i \right\}, \sum_{i=k+1}^{n} |\delta_i| + \max_{i\in\{1..m\}} \left\{ \sum_{j=1}^{i-1} |\delta'_j| + m'_i \right\} \right\} \\ &= \max\left\{ \max_{i\in\{k+1..n\}} \left\{ \sum_{j=k+1}^{i-1} |\delta_j| + m_i \right\}, \sum_{i=k+1}^{n} |\delta_i| + \max_{i\in\{1..m\}} \left\{ \sum_{j=n+1}^{n+i-1} |\delta_j| + m'_i \right\} \right\} \\ &= \max\left\{ \max_{i\in\{k+1..n\}} \left\{ \sum_{j=k+1}^{i-1} |\delta_j| + m_i \right\}, \sum_{i=k+1}^{n} |\delta_i| + \max_{i\in\{n+1..n+m\}} \left\{ \sum_{j=n+1}^{i-1} |\delta_j| + m_i \right\} \right\} \\ &= \max\left\{ \max_{i\in\{k+1..n\}} \left\{ \sum_{j=k+1}^{i-1} |\delta_j| + m_i \right\}, \max_{i\in\{n+1..n+m\}} \left\{ \sum_{j=k+1}^{i-1} |\delta_j| + m_i \right\} \right\} \\ &= \max_{i\in\{k+1..n+m\}} \left\{ \sum_{j=k+1}^{i-1} |\delta_j| + m_i \right\} \end{aligned}$$

- $I_B \neq \emptyset$

  We obtain, in this case $p = \max\ I_B = n + p_B$. That is, the last call to f in the whole sequence is the last call to f in the sub-sequence $[G_2 \to be'_1, \ldots, be'_n]$. Moreover, we know that $L^f(\psi_B) \neq 1$

in this case, so we have to apply $[Let2_{NC}]$, instead of $[Let1_{NC}]$. We obtain:

$$\begin{aligned}\delta_b^f(\varphi) &= \delta_b^f(\psi_A) + \delta_a^f(\psi_A) + \delta_b^f(\psi_B) = \sum_{i=1}^{p_A}\delta_i + \sum_{i=p_A+1}^{n}\delta_i + \sum_{i=1}^{p_B}\delta'_i \\ &= \sum_{i=1}^{n}\delta_i + \sum_{i=n}^{n+p_B}\delta_i = \sum_{i=1}^{n+p_B}\delta_i = \sum_{i=1}^{p}\delta_i\end{aligned}$$

$$\delta_a^f(\varphi) = \delta_a^f(\psi_B) = \sum_{i=p_B+1}^{m}\delta'_i = \sum_{i=n+p_B+1}^{n+m}\delta_i = \sum_{i=p+1}^{n+m}\delta_i$$

$$\begin{aligned}m_b^f(\varphi) &= \max\left\{m_b^f(\psi_A), |\delta_b^f(\psi_A)| + m_a^f(\psi_A), |\delta_b^f(\psi_A)| + |\delta_a^f(\psi_A)| + m_b^f(\psi_B)\right\} \\ &= \max\left\{\max_{i\in\{1..k_A\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\}, \sum_{i=1}^{k_A}|\delta_i| + \max_{i\in\{k_A+1..n\}}\left\{\sum_{j=k_A+1}^{i-1}|\delta_j| + m_i\right\},\right. \\ &\qquad \left.\sum_{i=1}^{k_A}|\delta_i| + \sum_{i=k_A+1}^{n}|\delta_i| + \max_{i\in\{1..k_B\}}\left\{\sum_{j=1}^{i-1}|\delta'_j| + m'_i\right\}\right\} \\ &= \max\left\{\max_{i\in\{1..k_A\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\}, \max_{i\in\{k_A+1..n\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\},\right. \\ &\qquad \left.\sum_{i=1}^{n}|\delta_i| + \max_{i\in\{n+1..n+k_B\}}\left\{\sum_{j=n+1}^{i-1}|\delta_j| + m_i\right\}\right\} \\ &= \max\left\{\max_{i\in\{1..k_A\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\}, \max_{i\in\{k_A+1..n\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\},\right. \\ &\qquad \left.\max_{i\in\{n+1..n+k_B\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\}\right\} \\ &= \max_{i\in\{1..n+k_B\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\} \\ &= \max_{i\in\{1..k\}}\left\{\sum_{j=1}^{i-1}|\delta_j| + m_i\right\}\end{aligned}$$

$$\begin{aligned}m_a^f(\varphi) &= m_a^f(\psi_B) = \max_{i=\{k_B+1..m\}}\left\{\sum_{j=k_B+1}^{i-1}|\delta'_j| + m'_i\right\} \\ &= \max_{i=\{k_B+1..m\}}\left\{\sum_{j=n+k_B+1}^{n+i-1}|\delta_j| + m'_i\right\} \\ &= \max_{i=\{n+k_B+1..n+m\}}\left\{\sum_{j=n+k_B+1}^{i-1}|\delta_j| + m_i\right\} \\ &= = \max_{i=\{k+1..n+m\}}\left\{\sum_{j=k+1}^{i-1}|\delta_j| + m_i\right\}\end{aligned}$$

Therefore, (3) holds in both cases.

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}} \to e_i}$

Since the values $N_b^f(\varphi)$, $N_r^f(\varphi)$, $L^f(\varphi)$, $\delta_b^f(\varphi)$, $\delta_a^f(\varphi)$, $m_b^f(\varphi)$, $m_a^f(\varphi)$ of the φ judgement are exactly the same as their counterparts in the judgement of the e_i being executed, the lemma follows trivially from the induction hypothesis applied to the latter judgement.

□

Now we are ready to give a formal definition of a function *nb*, *nr* and *len* being correct.

Definition B.3. Let us assume a function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$. We say that *nr* (resp. *nb* and *len*) are correct approximations of the number of recursive calls (resp. base calls and height of the call tree), iff for every φ, E_f, h, k, td, h', v, δ, m, s, $\overline{v_i}^n$, $\overline{i_j}^m$, $\overline{s_i}^n$ such that:

1. $\varphi \equiv E_f \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$, where $E_f = [\overline{x_i \mapsto v_i}^n, \overline{r_j \mapsto i_j}^m, self \mapsto k+1]$.
2. For each $i \in \{1..n\}$, $s_i = size(h, v_i)$

Then it holds that $nr\ \overline{s_i}^n \geq N_r^f(\varphi)$ (resp. $nb\ \overline{s_i}^n \geq N_b^f(\varphi)$ and $len\ \overline{s_i}^n \geq L^f(\varphi)$).

B.2 Correctness of the initial Δ_0

The first step for proving the correctness of *computeDelta* is to express our δ component resulting from the evaluation of a given function in terms of the charges done by the base and recursive cases, and the number of base and recursive calls done during that evaluation. The intuition behind this idea is the same as in Section 7.5.3. In order to compute the charges done by a recursive call, we need to be able to isolate the charges done by the call itself from the charges done by its subsequent recursive calls. This can be done with the help of an additional syntactic construct **dmask**, which resets the δ component of a given expression. Its semantics are given by the following rule:

$$\frac{E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)}{E \vdash h, k, td, \mathbf{dmask}\ e \Downarrow h', k, v, ([\,]_k, m, s)}$$

It is easy to see that if we replace a function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f$ in a signature Σ by another masked function definition $f\ \overline{x_i}\ @\ \overline{r_j} = \mathbf{dmask}\ e_f$ we will be able to obtain the same judgements as with our initial signature, but we will obtain a possibly different δ component in the resource vector. The next theorem uses this function to define the charges done by a recursive call. The notation $\Phi(\varphi)$ denotes the set of judgements belonging to the derivation of φ (including φ itself).

Theorem B.4. *Assume a function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f \in \Sigma$ such that the following execution takes place:*

$$\varphi \equiv E \vdash h, k+1, td, e_f \Downarrow_\Sigma h', k+1, v, (\delta, m, s)$$

Let us define $\Sigma' = (\Sigma \backslash f) \uplus [f \mapsto f\ \overline{x_i}\ @\ \overline{r_j} = \mathbf{dmask}\ e_f]$, and assume the following execution under Σ',

$$\varphi' \equiv E \vdash h, k+1, td, e_f \Downarrow_{\Sigma'} h, k+1, v, (\delta', m, s)$$

which is derivable by using the $\Downarrow$-rules. Given the following definitions:

$$\begin{aligned} \delta_{base} &= \max\{\delta(\psi) \mid \psi \in \Phi(\varphi), Exp(\psi) = e_f, L^f(\psi) = 1\} \\ \delta_{rec} &= \max\{\delta(\psi) \mid \psi \in \Phi(\varphi'), Exp(\psi) = e_f, L^f(\psi) > 1\} \end{aligned}$$

Then we get:

$$\delta|_k \leq \delta_{base}|_k * N_b^f(\varphi) + \delta_{rec}|_k * N_r^f(\varphi) \tag{B.7}$$

Proof. By induction on $L^f(\varphi)$. Let us distinguish cases:

- **Case** $L^f(\varphi) = 1$

  In this case $\delta_{base} \geq \delta$. Since we assume absence of destruction, we get $\delta_{rec}(i) \geq 0$ for each $i \in \{1..n\}$. Moreover, $N_b^f(\varphi) \geq 1$ and $N_r^f(\varphi) \geq 0$, so:

  $$\delta|_k \leq \delta_{base}|_k \leq \delta_{base}|_k * N_b^f(\varphi) \leq \delta_{base}|_k * N_b^f(\varphi) + \delta_{rec}|_k * N_r^f(\varphi)$$

- **Case** $L^f(\varphi) > 1$

  By Lemma B.2, there exists a sequence $[G \to be_1, \ldots, be_n]$ in *seqs* e_f and some $\overline{E_i}^n$, $\overline{h_i}^n$, $\overline{td_i}^n$, $\overline{h_i}^n$, $\overline{v_i}^n$, $\overline{\delta_i}^n$, $\overline{m_i}^n$, $\overline{s_i}^n$ of their respective types such that:

  $$\varphi_i \equiv E_i \vdash h_i, k, td_i, be_i \Downarrow_\Sigma h_i', k, v_i, (\delta_i, m_i, s_i) \text{ belongs to the derivation of } \varphi$$

  For each $i \in \{1..n\}$, since the derivation of φ' is the same as that of φ, but with different δ components in the resource vectors, the following judgements belong to the derivation of φ',

  $$\varphi_i' \equiv E_i \vdash h_i, k, td_i, be_i \Downarrow_{\Sigma'} h_i', k, v_i, (\delta_i', m_i, s_i)$$

  with $\delta' = \sum_{i=1}^n \delta_i'$. Now we define the set I of indices such that the corresponding be_i is a function application to f. For every $i \in \{1..n\}$, if $i \notin I$ we can execute be_i under Σ to obtain a judgement similar to φ_i', since the latter does not contain calls to f, and the value of $\Sigma(f)$ is not relevant to this execution. Thus we get:

  $$E_i \vdash h_i, k, td_i, be_i \Downarrow_\Sigma h_i', k, v_i, (\delta_i', m_i, s_i)$$

  But, since φ_i holds, and the computation of the resource vector is deterministic, we get $\delta_i = \delta_i'$. If $i \in I$, we know that there is an evaluation of **dmask** e_f "above" the judgement φ_i', so we know that $\delta_i' = [\,]_k$ in these cases. Thus we get:

  $$\sum_{i \notin I} \delta_i = \sum_{i \notin I} \delta_i' = \sum_{i \notin I} \delta_i' + \sum_{i \in I} \delta_i' = \sum_{i=1}^n \delta_i' = \delta' \leq \delta_{rec} \tag{B.8}$$

  Now assume an index $i \in I$, and consider again the corresponding φ_i. Above this judgement, we can find the execution of the body of the function , whose judgement we denote by χ_i:

  $$\chi_i \equiv E_{i,f} \vdash h_{i,f}, k+2, td_{i,f}, e_f \Downarrow_\Sigma h_{i,f}', k+2, v_{i,f}, (\delta_{i,f}, m_{i,f}, s_{i,f})$$

  There exists a counterpart χ_i', defined as follows:

  $$\chi_i \equiv E_{i,f} \vdash h_{i,f}, k+2, td_{i,f}, e_f \Downarrow_{\Sigma'} h_{i,f}', k+2, v_{i,f}, (\delta_{i,f}', m_{i,f}, s_{i,f})$$

  By induction hypothesis, we get, for each $i \in I$:

  $$\delta_{i,f}|_{k+1} \leq \delta_{base,i}|_{k+1} * N_b^f(\chi_i) + \delta_{rec,i}|_{k+1} * N_r^f(\chi_i)$$

where $\delta_{base,i}$ and $\delta_{rec,i}$ are defined as follows:

$$\begin{aligned}\delta_{base,i} &= \max\{\delta(\psi) \mid \psi \in \Phi(\chi_i), Exp(\psi) = e_f, L^f(\psi) = 1\}\\ \delta_{rec,i} &= \max\{\delta(\psi) \mid \psi \in \Phi(\chi'_i), Exp(\psi) = e_f, L^f(\psi) > 1\}\end{aligned}$$

Since $\Phi(\chi_i) \subseteq \Phi(\varphi)$ and $\Phi(\chi'_i) \subseteq \Phi(\varphi')$, we get $\delta_{base,i} \leq \delta_{base}$ and $\delta_{rec,i} \leq \delta_{rec}$. Hence:

$$\delta_i|_k \leq \delta_{base}|_k * N_b^f(\varphi_i) + \delta_{rec}|_k * (N_r^f(\varphi_i) - 1) \tag{B.9}$$

Therefore, we can prove the required result:

$$\begin{aligned}\delta|_k &= \textstyle\sum_{i=1}^n \delta_i|_k && \{\text{ by Lemma B.2 }\}\\ &= \textstyle\sum_{i\notin I} \delta_i|_k + \sum_{i\in I} \delta_i|_k\\ &\leq \delta_{rec}|_k + \textstyle\sum_{i\in I} \delta_i|_k && \{\text{ by (B.8) }\}\\ &\leq \delta_{rec}|_k + \textstyle\sum_{i\in I}(\delta_{base}|_k * N_b^f(\varphi_i) + \delta_{rec}|_k * (N_r^f(\varphi_i) - 1)) && \{\text{ by (B.9) }\}\\ &= \delta_{base}|_k * \textstyle\sum_{i\in I} N_b^f(\varphi_i) + \delta_{rec}|_k * (1 + \sum_{i\in I}(N_r^f(\varphi_i) - 1))\\ &= \delta_{base}|_k * N_b^f(\varphi) + \delta_{rec}|_k * N_r^f(\varphi) && \{\text{ by Lemma B.2 }\}\end{aligned}$$

□

Notice the similarity between the expression (B.7) and that occurring in the definition of *computeDelta*. The latter can be considered an "abstract" version of the former. The correctness of the result of *computeDelta* follows from this theorem. Before this, we need some technical result on the domain of the result given by the abstract interpretation function. In particular, it states the conditions under which this result is defined.

Lemma B.5. *Assume an execution* $\varphi \equiv E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)$ *belonging to the derivation of the following context judgement:*

$$E_0 \vdash h_0, k, td_0, e_f \Downarrow h'_0, k, v_0, (\delta_0, m_0, s_0)$$

corresponding to the function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f$*. Let us define, for each* $i \in \{1..n\}$, $s_i = size(h, E_0(x_i))$*, and* $[\![e]\!]\ \Sigma\ \Gamma\ td = (\Delta, \mu, \sigma)$*, where* Σ *is an environment* Σ *containing correct signatures for the functions being called from* f*, and* Γ *is an environment typing* e_f*. If* $closed(h, E_0(x_i))$ *for each* $i \in \{1..n\}$*, then* $\Delta\ \overline{s_i}^n \neq \bot$*,* $\mu\ \overline{s_i}^n \neq \bot$*, and* $\sigma\ \overline{s_i}^n \neq \bot$*.*

Proof. (Sketch) By induction on the size of the $\Downarrow$ derivation. All cases are straightforward. In the case of function applications, the guards of the resulting triple (Δ, μ, σ) hold because Σ is a correct signature. In **case** expressions, the guards are satisfied because the initial heap is closed, and closedness is propagated through the execution of a well-typed function. □

Theorem B.6. *Let* $\Delta = computeDelta\ (f\ \overline{x_i}\ @\ \overline{r_j} = e_f)\ \Sigma\ \Gamma\ nb\ nr$*. If the following conditions hold:*

1. Σ *is a correct signature for all the functions being called from* f*.*
2. $\Gamma \vdash e : s$ *for some* $s \in$ **SafeType**.
3. *nb and nr are correct approximations of the number of base and recursive calls of* f*, respectively.*
4. *The abstract heaps* Δ_b *and* Δ_r *occurring in the definition of computeDelta are parameter-decreasing.*

Then Δ *is a correct abstract heap for* f*.*

Proof. (Sketch) It is a consequence of Theorem B.4, and the Δ_b and Δ_r being upper bounds of the δ_{base} and δ_{rec}, respectively. The abstract heap Δ_b is a correct bound of δ_{base} because of Theorem 7.26, whereas Δ_r is a correct bound for δ_{rec}, because $[\,]_f$ is a correct bound to the heap charges of the function definition $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = \mathbf{dmask}\ e_f$. Assume a judgement:

$$\varphi \equiv E \vdash h, k, td, e_f \Downarrow h', k, v, (\delta, m, s)$$

Let us define, for each $i \in \{1..n\}$, $s_i = size(h, E(x_i))$, and let us denote by η the region instantiation consistent with E and Γ. We get:

$$\Delta \sqsupseteq \lfloor \Delta_b \rfloor * nb + \lfloor \Delta_r \rfloor * nr \succeq_{\overline{s_i}^n, k, \eta} \delta_{base} * N_b^f(\varphi) + \delta_{rec} * N_r^f(\varphi) \geq \delta$$

□

B.3 Correctness of the initial μ_0

For proving the correctness of the *computeMu* algorithm we follow a similar approach as in the previous section. In this case we need an additional construct **mmask**, which resets the m component of the resource vector:

$$\frac{E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)}{E \vdash h, k, td, \mathbf{mmask}\ e \Downarrow h', k, v, (\delta, 0, s)}$$

Similarly as above, the following theorem gives the memory needs as an expression which resembles the cost expression given by *computeDelta*. The difference is that it uses elements from the concrete domain, rather than the abstract one.

Theorem B.7. *Assume a function definition $f\ \overline{x_i}\ @\ \overline{r_j} = e_f \in \Sigma$ and the following judgement:*

$$\varphi \equiv E \vdash h, k+1, td, e_f \Downarrow_\Sigma h', k+1, v, (\delta, m, s)$$

Let us define $\Sigma' = (\Sigma \backslash f) \uplus [f \mapsto f\ \overline{x_i}\ @\ \overline{r_j} = \mathbf{mmask}\ e_f]$, and assume the following execution under Σ',

$$\varphi' \equiv E \vdash h, k+1, td, e_f \Downarrow_{\Sigma'} h, k+1, v, (\delta, m', s)$$

which is derivable by using the $\Downarrow$-rules. Given the following definitions:

$$\begin{aligned}
\delta_{self} &= \max\{\delta_b^f(\psi)(k+1) \mid \psi \in \Phi(\varphi'), Exp(\psi) = e_f, L^f(\psi) > 1\} \\
m_{bef} &= \max\{m_b^f(\psi) \mid \psi \in \Phi(\varphi'), Exp(\psi) = e_f, L^f(\psi) > 1\} \\
m_{aft} &= \max\{m_a^f(\psi) \mid \psi \in \Phi(\varphi'), Exp(\psi) = e_f, L^f(\psi) > 1\} \\
m_{base} &= \max\{m(\psi) \mid \psi \in \Phi(\varphi), Exp(\psi) = e_f, L^f(\psi) = 1\}
\end{aligned}$$

We get:

$$m \leq |\delta_b^f(\varphi')|_k| + \delta_{self} * (L^f(\varphi) - 1) + \max\{m_{bef}, m_{aft}, m_{base}\} \tag{B.10}$$

Proof. By induction on the maximum number of nested calls, $L^f(\varphi)$. We distinguish cases:

- **Case $L^f(\varphi) = 1$.**

Then $m \leq m_{base}$. Moreover, $|\delta_b^f(\varphi')|_k| \geq 0$, since we are assuming absence of explicit destruction. Hence we get:

$$m \leq m_{base} \leq |\delta_b^f(\varphi')|_k| + \delta_{self} * 0 + \max\{m_{bef}, m_{aft}, m_{base}\}$$

- **Case** $L^f(\varphi) > 1$.

By Lemma B.2, there exists a sequence $[G \to be_1, \ldots, be_n]$ such that φ_i belongs to $\Phi(\varphi)$, where φ_i is defined as follows:

$$\varphi_i \equiv E_i \vdash h_i, k, td_i, be_i \Downarrow_\Sigma h_i', k, v_i, (\delta_i, m_i, s_i)$$

for every $i \in \{1..n\}$, and for some E_i, h_i, td_i, h_i', v_i, δ_i, m_i and s_i of their respective types. Since the only difference between Σ and Σ' is that the latter resets the m component of each recursive call to f, there exists, for each of these φ_i judgements, a counterpart φ_i' belonging to φ',

$$\varphi_i' \equiv E_i \vdash h_i, k, td_i, be_i \Downarrow_{\Sigma'} h_i', k, v_i, (\delta_i, m_i', s_i)$$

for some m_i'. Let us denote by I the set of indices i such that be_i has the form $f\ \overline{a_i}\ @\ \overline{r_j}$ for some $\overline{a_i}$ and $\overline{r_j}$. The difference between Σ and Σ' is not relevant in the judgements φ_i' when $i \notin I$, since there are no applications of f in their derivations. Then we can substitute Σ for Σ' in those φ_i', and we would get the same resource vector (δ_i, m_i', s_i) as a result. Since the computation of this vector is deterministic, and because φ_i holds, we get $m_i = m_i'$ whenever $i \notin I$.

Let us define $p = \max\ I$. By applying (B.1) we know that

$$m = \max\{m_b^f(\varphi), |\delta_b^f(\varphi)| + m_a^f(\varphi)\}$$

We have to prove that each one of the expressions in the $\max\{\ldots\}$ is lower than the right-hand side of (B.10). Let us start with the second one. By Lemma (B.2) it holds that

$$\delta_b^f(\varphi) = \sum_{i=1}^{p} \delta_i = \delta_b^f(\varphi') \tag{B.11}$$

and, similarly,

$$m_a^f(\varphi) = \max_{i \in \{p+1..n\}} \left\{ \sum_{j=p+1}^{i-1} |\delta_j| + m_i \right\} = \max_{i \in \{p+1..n\}} \left\{ \sum_{j=p+1}^{i-1} |\delta_j| + m_i' \right\} = m_a^f(\varphi') \tag{B.12}$$

The second step can be done because $i \notin I$ for every $i \in \{p+1..n\}$, by definition of I. Besides this, we know that $\text{dom}\ \delta_b^f(\varphi') = \{0..k+1\}$, so we can split $|\delta_b^f(\varphi')|$ as follows:

$$|\delta_b^f(\varphi')| = |\delta_b^f(\varphi')|_k| + \delta_b^f(\varphi')(k+1) \tag{B.13}$$

Hence we get:

$$
\begin{array}{lll}
& |\delta_b^f(\varphi)| + m_a^f(\varphi) & \\
= & |\delta_b^f(\varphi')| + m_a^f(\varphi') & \{ \text{ by (B.11) and (B.12) } \} \\
= & |\delta_b^f(\varphi')|_k| + \delta_b(\varphi')(k+1) + m_a^f(\varphi') & \{ \text{ by (B.13) } \} \\
\leq & |\delta_b^f(\varphi')|_k| + \delta_{self} + m_{aft} & \{ \text{ by definition of } \delta_{self} \text{and } m_{aft} \} \\
\leq & |\delta_b^f(\varphi')|_k| + \delta_{self} * (L^f(\varphi) - 1) + m_{aft} & \{ \text{ since } L^f(\varphi) > 1 \} \\
\leq & |\delta_b^f(\varphi')|_k| + \delta_{self} * (L^f(\varphi) - 1) + \max\{m_{bef}, m_{aft}, m_{base}\} &
\end{array}
$$

and we are done. Now let us prove that $m_b^f(\varphi)$ is lower than the right-hand side of (B.10). We can unfold, by Lemma B.2 the value of $m_b^f(\varphi)$ as follows:

$$m_b^f(\varphi) = \max_{i \in \{1..p\}} \{M_i\}, \text{ where } \forall i \in \{1..p\}. M_i \stackrel{\text{def}}{=} \sum_{j=1}^{i-1} |\delta_j| + m_i$$

Thus, the required result amounts to proving that M_i is lower or equal than the right-hand side of (B.10), for each $M_i \in \{1..p\}$. So, let us assume $i \in \{1..p\}$ and distinguish cases:

– $i \notin I$

In this case it holds that:

$$M_i = \sum_{j=1}^{i-1} |\delta_j| + m_i = \sum_{j=1}^{i-1} |\delta_j| + m_i' \leq m_b^f(\varphi') \leq m_{bef}$$

and, hence,

$$M_i \leq m_{bef} \leq |\delta_b^f(\varphi')|_k| + \delta_{self} * (L^f(\varphi) - 1) + \max\{m_{bef}, m_{aft}, m_{base}\}$$

since $\delta_b^f(\varphi')(j) \geq 0$ for each $j \in \{1..k\}$, $\delta_{self} \geq 0$, and $L^f(\varphi) > 1$.

– $i \in I$

In this case, be_i is a function application. We get that the following judgement:

$$\chi_i \equiv E_{i,f} \vdash h_{i,f}, k+2, td_{i,f}, e_f \Downarrow_\Sigma h'_{i,f}, k+2, v_{i,f}, (\delta_{i,f}, m_{i,f}, s_{i,f})$$

belongs to the derivation of φ_i, and a analogous one χ_i' belongs to φ_i':

$$\chi_i' \equiv E_{i,f} \vdash h_{i,f}, k+2, td_{i,f}, e_f \Downarrow_{\Sigma'} h'_{i,f}, k+2, v_{i,f}, (\delta_{i,f}, m'_{i,f}, s_{i,f})$$

By induction hypothesis:

$$m_{i,f} \leq |\delta_b^f(\chi_i')|_{k+1}| + \delta_{self,i} * (L^f(\chi_i) - 1) + \max\{m_{bef,i}, m_{aft,i}, m_{base,i}\} \tag{B.14}$$

where

$$\begin{aligned}
\delta_{self,i} &= \max\{\delta_b^f(\psi)(k+1) \mid \psi \in \Phi(\chi_i'), Exp(\psi) = e_f, L^f(\psi) > 1\} \\
m_{bef,i} &= \max\{m_b^f(\psi) \mid \psi \in \Phi(\chi_i'), Exp(\psi) = e_f, L^f(\psi) > 1\} \\
m_{aft,i} &= \max\{m_a^f(\psi) \mid \psi \in \Phi(\chi_i'), Exp(\psi) = e_f, L^f(\psi) > 1\} \\
m_{base,i} &= \max\{m(\psi) \mid \psi \in \Phi(\chi_i), Exp(\psi) = e_f, L^f(\psi) = 1\}
\end{aligned}$$

By Lemma B.1 (3) we know that $\delta_b^f(\chi_i') \leq \delta_{i,f}$. Moreover, by the $[App]$ rule, $\delta_{i,f}|_{k+1} = \delta_i$. Therefore:

$$|\delta_b^f(\chi_i')|_{k+1}| \leq |\delta_{i,f}|_{k+1}| \leq |\delta_i| \tag{B.15}$$

Besides this, the judgement χ_i belongs to the derivation of φ, and χ_i' belongs to the derivation of φ'. Thus $\Phi(\chi_i) \subseteq \Phi(\varphi)$ and $\Phi(\chi_i) \subseteq \Phi(\varphi')$, which implies:

$$\delta_{self,i} \leq \delta_{self} \qquad m_{bef,i} \leq m_{bef} \qquad m_{aft,i} \leq m_{aft} \qquad m_{base,i} \leq m_{base} \tag{B.16}$$

Finally, by the $[App_{NC}]$ rule, it holds that $L^f(\varphi_i) = L^f(\chi_i) + 1$. By combining (B.14), (B.15), (B.16) with this equality and the fact that $m_i = m_{i,j}$, we obtain:

$$m_i = m_{i,j} \leq |\delta_i| + \delta_{self} * (L^f(\varphi_i) - 2) + \max\{m_{bef}, m_{aft}, m_{base}\}$$

Therefore:

$$\begin{aligned}
M_i &= \textstyle\sum_{j=1}^{i-1} |\delta_j| + m_i \\
&\leq \textstyle\sum_{j=1}^{i-1} |\delta_j| + |\delta_i| + \delta_{self} * (L^f(\varphi_i) - 2) + \max\{m_{bef}, m_{aft}, m_{base}\} \\
&= \textstyle\sum_{j=1}^{i} |\delta_i| + \delta_{self} * (L^f(\varphi_i) - 2) + \max\{m_{bef}, m_{aft}, m_{base}\} \\
&\leq \textstyle\sum_{j=1}^{p} |\delta_i| + \delta_{self} * (L^f(\varphi_i) - 2) + \max\{m_{bef}, m_{aft}, m_{base}\} \qquad \{\text{ since } i \leq p\}
\end{aligned}$$

The last step is justified by the fact that $|\delta_i| \geq 0$ for each $i \in \{1..p\}$. By Lemma B.2 we know that $\delta_b^f(\varphi') = \sum_{i=1}^{p} \delta_i$, and, since dom $\delta_b^f(\varphi') = \{0..k+1\}$ we get:

$$\begin{aligned}
M_i &\leq |\delta_b^f(\varphi')|_k| + \delta_b^f(\varphi')(k+1) + \delta_{self} * (L^f(\varphi_i) - 2) + \max\{m_{bef}, m_{aft}, m_{base}\} \\
&\leq |\delta_b^f(\varphi')|_k| + \delta_{self} + \delta_{self} * (L^f(\varphi_i) - 2) + \max\{m_{bef}, m_{aft}, m_{base}\} \\
&= |\delta_b^f(\varphi')|_k| + \delta_{self} * (1 + (L^f(\varphi_i) - 1) - 1) + \max\{m_{bef}, m_{aft}, m_{base}\} \\
&\leq |\delta_b^f(\varphi')|_k| + \delta_{self} * (L^f(\varphi) - 1) + \max\{m_{bef}, m_{aft}, m_{base}\}
\end{aligned}$$

□

The correctness of *computeMu* is established by connecting the elements of the abstract domain appearing in its result with the elements of the concrete domain occurring in the previous theorem.

Theorem B.8. *Let μ = computeMu $(f\ \overline{x_i}\ @\ \overline{r_j} = e_f)\ \Sigma\ \Gamma\ \Delta$ len. If the following conditions hold:*

1. *Σ is a correct signature for all the functions being called from f.*
2. *$\Gamma \vdash e : s$ for some $s \in$ **SafeType**.*
3. *len is a correct approximation of the maximum length of the call tree of f.*
4. *Δ is a correct abstract heap for f.*

5. *The space cost functions Δ_{self}, μ_{aft}, and μ_{bef} occurring in the definition of computeMu are parameter-decreasing.*

Then μ is correct with respect to f.

Proof. (Sketch) The proof proceeds in a similar way as Corollary B.6. In this case, it follows from Theorem B.7. If we have a judgement:

$$\varphi \equiv E \vdash h, k, td, e_f \Downarrow h', k, v, (\delta, m, s)$$

and denote, for each $i \in \{1..n\}$, $s_i = size(h, E(x_i))$, we obtain:

$$\mu \sqsupseteq \Delta_{self} * (len - 1) + |\Delta_{bef}| + \sqcup\{\mu_{bef}, \mu_{aft}, \mu_b\} \succeq_{\overline{s_i}^n} \delta_{self} * (L^f(\varphi) - 1) + |\delta_b^f(\varphi)|_k| + \max\{m_{bef}, m_{aft}, m_{base}\} \geq m$$

□

B.4 Correctness of the initial σ_0

The computation of the initial approximation to the stack costs relies on the concept of stack level, defined in Section 7.5.5. This level stands for the number of words existing in the stack at a given time. Neither the semantic rules of Figure 2.26 nor the rules we have defined in Figure B.1 is able to capture this information. Recall that the *td* component only counts the number of words from the top of the stack to the topmost continuation, whereas we are interested in the total number of words in the stack including those lying below that continuation. For this reason, we have to extend again the semantic rules of Figure 2.26 in order to include this information. Our judgements will have the following form:

$$E \vdash h, k, (td, s_0), e \Downarrow h', k, v, (\delta, m, s)$$

where s_0 is the new component that counts the number of words in the stack. In Figure B.3 we show the rules defining this extension. Most of them are self-explaining. The s_0 components mimics the *td* component in almost every expression, except when evaluating the bound expression of a **let**, where the *td* component becomes zero (as a new continuation has been pushed), and the s_0 component is incremented by two (the number of stack words taken by a continuation).

We are particularly interested in the difference between the stack levels between two points: when a function starts, and when a recursive call is going to be done. That is what the SD_f function defined in Figure 7.15 computes. The following lemma shows how we can compute the stack costs of the execution of a given expression from these differences between stack levels, and the stack costs of each recursive call. Before this, we need to introduce the semantics of the **smask** expressions, which play a similar role as **dmask** and **mmask**.

$$\frac{E \vdash h, k, td, e \Downarrow h', k, v, (\delta, m, s)}{E \vdash h, k, td, \mathbf{smask}\ e \Downarrow h', k, v, (\delta, m, 0)}$$

Lemma B.9. *Let us assume the execution of an expression under an environment Σ, and that there are p direct calls to f in the corresponding execution (and hence p judgements evaluating e_f). For each $i \in \{1 \dots p\}$, we*

$$\frac{}{E \vdash h, k, (td, s_0), c \Downarrow h, k, c, ([\,]_k, 0, 1)}\ [Lit_{ST}]$$

$$\frac{}{E[x \mapsto v] \vdash h, k, (td, s_0), x \Downarrow h, k, v, ([\,]_k, 0, 1)}\ [Var_{ST}]$$

$$\frac{}{E \vdash h, k, (td, s_0), a_1 \oplus a_2 \Downarrow h, k, E(a_1) \oplus E(a_2), ([\,]_k, 0, 2)}\ [PrimOp_{ST}]$$

$$\frac{j \leq k \qquad (h', p') = copy(h, p, j) \qquad m = size(h, p)}{E[x \mapsto p, r \mapsto j] \vdash h, k, (td, s_0), x\ @\ r \Downarrow h', k, p', ([j \mapsto m]_k, m, 2)}\ [Copy_{ST}]$$

$$\frac{\begin{array}{c}(g\ \overline{y_i}^n\ @\ \overline{r'_j}^l = e_g) \in \Sigma \\ [\overline{y_i \mapsto E(a_i)}^n, \overline{r'_j \mapsto E(r_j)}^l, self \mapsto k+1] \vdash h, k+1, (n+l, s_0+n+l-td), e \Downarrow h', k+1, v, (\delta, m, s)\end{array}}{E \vdash h, k, (td, s_0), g\ \overline{a_i}^n\ @\ \overline{r_j}^l \Downarrow h' \mid_k, k, v, (\delta \mid_k, m, \max\{n+l, s+n+l-td\})}\ [App_{ST}]$$

$$\frac{E(r) = j \qquad j \leq k \qquad fresh_h(p)}{E \vdash h, k, (td, s_0), C\ \overline{a_i}^n\ @\ r \Downarrow h \uplus [p \mapsto (j, C\ \overline{E(a_i)}^n)], k, v, ([j \mapsto 1]_k, 1, 1)}\ [Cons_{ST}]$$

$$\frac{\begin{array}{c}E \vdash h, k, (0, s_0+2), e_1 \Downarrow h', k, v_1, (\delta_1, m_1, s_1) \\ E \cup [x_1 \mapsto v_1] \vdash h', k, (td+1, s_0+1), e_2 \Downarrow h'', k, v, (\delta_2, m_2, s_2)\end{array}}{E \vdash h, k, (td, s_0), \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \Downarrow h'', k, v, (\delta_1 + \delta_2, \max\{m_1, |\delta_1| + m_2\}, \max\{2 + s_1, 1 + s_2\})}\ [Let_{ST}]$$

$$\frac{C = C_r \qquad E(x) = p \qquad E \uplus [\overline{x_{rj} \mapsto v_j}^{n_r}] \vdash h, k, (td + n_r, s_0 + n_r), e_r \Downarrow h', k, v, (\delta, m, s)}{E \vdash h[p \mapsto (j, C\ \overline{v_i}^{n_r})], k, (td, s_0), \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \Downarrow h', k, v, (\delta, m, s + n_r)}\ [Case_{ST}]$$

Figure B.3: Operational semantics of *Core-Safe* expressions with stack level information.

denote the initial stack size (resp. stack cost) of the i-th derivation by $s_0^{(i)}$ (resp. $s^{(i)}$):

$$\dfrac{\dfrac{\ldots\,(td^{(1)}, s_0^{(1)}), e_f \Downarrow \ldots\,(\delta^{(1)}, m^{(1)}, s^{(1)})}{\vdots} \quad \ldots \quad \dfrac{\ldots\,(td^{(p)}, s_0^{(p)}), e_f \Downarrow \ldots\,(\delta^{(p)}, m^{(p)}, s^{(p)})}{\vdots}}{\ldots\,(td, s_0), e \Downarrow \ldots\,(\delta, m, s)}$$

In addition, we assume the execution of the same expression e under the environment $\Sigma \backslash f \uplus [f \mapsto f\ \overline{x_i}\ @\ \overline{r_j} = \textbf{smask}\ e_f]$, obtaining s' as the stack cost. Then the following relation between s and s' holds:

$$s = \max\left(\{s'\} \cup \{s_0^{(i)} - s_0 + s^{(i)} \mid i = 1 \ldots p\}\right)$$

Proof. By induction on the structure of e.

- **Cases $e \equiv c$, $e \equiv x$, $e \equiv a_1 \oplus a_2$, $e \equiv C\ \overline{a_i}\ @\ r$, and $e \equiv g\ \overline{a_i}\ @\ \overline{r_j}$ with $g \neq f$.**

  This is trivial, since $p = 0$ and both executions with Σ and with Σ' produce the same stack costs. Hence $s = s' = \max\{s'\}$

- **Case $e \equiv f\ \overline{a_i}^n\ @\ \overline{r'_j}^m$**

  There is a single recursive call ($p = 1$):

  $$\dfrac{\ldots\,(n + m, s_0 + n + m - td), [\textbf{smask}]\ e_f \Downarrow \ldots\,(\ldots, \ldots, s^{(1)})}{\ldots\,(td, s_0), f\ \overline{a_i}^n\ @\ \overline{r'_j}^m \Downarrow \ldots\,(\ldots, \ldots, \max\{n + m, s^{(1)} + n + m - td\})}$$

  Under Σ' we get $s^{(1)} = 0$ and hence $s' = \max\{n + m, 0 + n + m - td\} = n + m$. Since $s_0^{(1)} - s_0 = (s_0 + n + m - td) - s_0 = n + m - td$, we get:

  $$s = \max\{n + m, s^{(1)} + n + m - td\} = \max\{s', s_0^{(1)} - s_0 + s^{(1)}\}$$

- **Case $e \equiv \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2$**

  We assume that there are q calls to f in the execution of e_1 and $p - q$ calls in the execution of e_2, with $0 \leq q \leq p$. We have got the following situation:

  $$\dfrac{\dfrac{(q\text{calls to } f)}{\ldots, (0, s_0 + 2), e_1 \Downarrow \ldots\,(\delta_1, m_1, s_1)} \quad \dfrac{(q - p\text{calls to } f)}{\ldots, (td + 1, s_0 + 1), e_2 \Downarrow \ldots\,(\delta_2, m_2, s_2)}}{\ldots, (td, s_0), \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \Downarrow \ldots\,(\delta, m, s)}$$

  If we denote by s' (resp. s'_1, s'_2) the costs of e (resp. e_1, e_2) when evaluated under Σ', we get:

  $$\begin{array}{rcl}
  s & = & \max\{2 + s_1, 1 + s_2\} \\
  & = & \max\Big\{2 + \max\left(\{s'_1\} \cup \{s_0^{(i)} - (s_0 + 2) + s^{(i)} \mid i \in \{1 \ldots q\}\}\right), \\
  & & \quad 1 + \max\left(\{s'_2\} \cup \{s_0^{(j)} - (s_0 + 1) + s^{(j)} \mid j \in \{q + 1 \ldots p\}\}\right)\Big\} \quad \{\text{ by i.h. }\} \\
  & = & \max\Big\{\max\{2 + s'_1, 1 + s'_2\}, \max\{s_0^{(i)} - s_0 + s^{(i)} \mid i \in \{1 \ldots q\}\}, \\
  & & \quad \max\{s_0^{(j)} - s_0 + s^{(j)} \mid j \in \{q + 1 \ldots p\}\}\Big\} \\
  & = & \max\left(\{s'\} \cup \{s_0^{(i)} - s_0 + s^{(i)} \mid i \in \{1 \ldots p\}\}\right)
  \end{array}$$

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}$

  We assume that the r-th branch is executed and we denote by s_r and s'_r the costs associated to the derivations with Σ and Σ'.

$$\begin{array}{rcll} s & = & n_r + s_r & \\ & = & n_r + \max\left(\{s'_r\} \cup \{s_0^{(i)} - (s_0 + n_r) + s^{(i)} \mid i \in \{1 \dots p\}\}\right) & \{\text{ by i.h. }\} \\ & = & \max\left(\{n_r + s'_r\} \cup \{s_0^{(i)} - s_0 + s^{(i)} \mid i \in \{1 \dots p\}\}\right) & \\ & = & \max\left(\{s'\} \cup \{s_0^{(i)} - s_0 + s^{(i)} \mid i \in \{1 \dots p\}\}\right) & \end{array}$$

□

The following lemma shows that the SD_f function returns an upper bound to the above mentioned stack difference.

Lemma B.10 (Correctness lemma for SD_f)**.** *We assume the execution $E \vdash h, k, (td, s_0), e \Downarrow h', k, v, (\delta, m, s)$, in which there are $p \geq 1$ direct calls to a function f. For a given $i \in \{1 \dots p\}$, we denote by $s_0^{(i)}$ the stack size before executing the e_f body corresponding to the i-th call of f:*

$$\dots \quad \dfrac{\dfrac{E' \vdash h_i, k+1, (td_i, s_0^{(i)}), e_f \Downarrow h'_i, k+1, v_i, (\delta_i, m_i, s_i)}{\vdots}\ \dots}{E \vdash h, k, (td, s_0), e \Downarrow h', k, v, (\delta, m, s)}$$

Then: $SD_f\ e\ td \geq s_0^{(i)} - s_0$

Proof. By induction on the size of the $\Downarrow$-derivation. Since the derivation contains a call to f, we assume that e contains a sub-expression $f\ \overline{a_i}\ @\ \overline{r_j}$. This rules out the cases $e \equiv c, x, a_1 \oplus a_2, C\ \overline{a_i}^n\ @\ r$, and $g\ \overline{b_i}^q\ @\ \overline{s_j}^q$ with $g \neq f$. We distinguish the remaining cases:

- **Case** $e \equiv f\ \overline{a_i}^n\ @\ \overline{r_j}^m$

  We get $s_0^{(i)} = s_0 + n + m - td$, and hence, $SD_f\ e\ td = n + m - td = s_0^{(i)} - s_0$

- **Case** $e \equiv$ **let** $x_1 = e_1$ **in** e_2

  If the i-th call to f is in e_2 we get:

$$\begin{array}{rcll} SD_f\ e\ td & \geq & 1 + (SD_f\ e_2\ (td+1)) & \{\text{ by definition of } SD_f\} \\ & \geq & 1 + s_0^{(i)} - (s_0 + 1) & \{\text{ by i.h. }\} \\ & = & s_0^{(i)} - s_0 & \end{array}$$

  If the i-th call to f is in e_1 we get:

$$\begin{array}{rcll} SD_f\ e\ td & \geq & 2 + (SD_f\ e_1\ 0) & \{\text{ by definition of } SD_f\} \\ & \geq & 2 + s_0^{(i)} - (s_0 + 2) & \{\text{ by i.h. }\} \\ & = & s_0^{(i)} - s_0 & \end{array}$$

- **Case** $e \equiv$ **case** x **of** $\overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n$

We assume the r-th branch being executed ($1 \leq r \leq n$). Therefore:

$$\begin{array}{lcll} SD_f\ e\ td & \geq & n_r + (SD_f\ e_r\ (td + n_r)) & \{\text{ definition of } SD_f\} \\ & \geq & n_r + s_0^{(i)} - (s_0 + n_r) & \{\text{ by i.h. }\} \\ & = & s_0^{(i)} - s_0 & \end{array}$$

□

Given these results, we are ready to proceed as in previous sections: we find an expression for s that resembles its abstract counterpart given by *computeSigma*.

Theorem B.11. *Assume a function definition* $f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f \in \Sigma$ *such that the following execution takes place:*

$$\varphi \equiv E \vdash h, k+1, td, e_f \Downarrow_\Sigma h', k+1, v, (\delta, m, s)$$

Let us define $\Sigma' = (\Sigma \backslash f) \uplus [f \mapsto f\ \overline{x_i}\ @\ \overline{r_j} = \textbf{smask}\ e_f]$*, and assume the following execution under* Σ'*:*

$$\varphi' \equiv E \vdash h, k+1, td, e_f \Downarrow_{\Sigma'} h', k+1, v, (\delta, m, s')$$

Given the following definition,

$$s_{max} \quad = \quad \max\{s(\psi) \mid \psi \in \Phi(\varphi')\}$$

we get,

$$s \leq \max\{0, SD\ e_f\ (n+m)\} * (L^f(\varphi) - 1) + s_{max} \tag{B.17}$$

Proof. By induction on $L^f(\varphi)$. We distinguish cases:

- **Case** $L^f(\varphi) = 1$

  There are no function applications to f in the derivation of φ'. Hence the value of $\Sigma'(f)$ is not relevant in that judgement, and we can substitute Σ for Σ' in φ', so as to get:

  $$E \vdash h, k+1, td, e_f \Downarrow_\Sigma h', k+1, v, (\delta, m, s')$$

  But, because of Lemma 2.19, we get $s = s'$, which implies $s \leq s_{max}$. Therefore:

  $$s \leq s_{max} \leq \max\{0, SD_f\ e_f\ (n+m)\} * 0 + s_{max}$$

- **Case** $L^f(\varphi) > 1$

  Let us assume there are p calls to f in the derivation of φ. We know that $p \geq 1$ (otherwise, $L^f(\varphi)$ would not be greater than 1). For each $i \in \{1..p\}$ there exists a judgement

  $$\chi_i \equiv E_i \vdash h_i, k+2, (td_i, s_{0,i}), e_f \Downarrow h_i', k+2, v_i, (\delta_i, m_i, s_i)$$

  in the derivation φ. Similarly, there exists another judgement

  $$\chi_i' \equiv E_i \vdash h_i, k+2, (td_i, s_{0,i}), e_f \Downarrow h_i', k+2, v_i, (\delta_i, m_i, s_i')$$

  in the derivation of φ'. Let us denote by φ_i (resp. φ_i') the judgements situated "below" χ_i (resp. χ_i') in the corresponding derivation tree. These judgements must represent the execution of a function

application of f. By Lemma B.9 we get:

$$s = \max\{\{s_{0,i} - s_0 + s_i \mid i \in \{1..p\}\} \cup \{s'\}\} \tag{B.18}$$

But we know that $s' \leq s_{max}$ by definition of s_{max}. By Lemma B.10 we get $SD_f\ e_f\ (n+m) \geq s_{0,i} - s_0$ for each $i \in \{1..p\}$. Moreover, by induction hypothesis, we get, for each $i \in \{1..p\}$:

$$s_i \quad \leq \quad \max\{0, SD_f\ e_f\ (n+m)\} * (L^f(\chi_i) - 1) + s_{max,i}$$

where:

$$s_{max,i} \quad = \quad \max\ \{s \mid \psi \equiv E \vdash h, k, td, e_f \Downarrow h', k, v, (\delta, m, s) \in \Phi(\chi_i')\}$$

Since $\Phi(\chi_i') \subseteq \Phi(\varphi')$, we get $s_{max,i} \leq s_{max}$. Besides this, $L^f(\varphi_i) = 1 + L^f(\chi_i)$. Therefore:

$$s_i \quad \leq \quad \max\{0, SD_f\ e_f\ (n+m)\} * (L^f(\varphi_i) - 2) + s_{max}$$

Finally, we get $1 + (L^f(\varphi_i) - 1) \leq L^f(\varphi)$. Thus:

$$s_i \quad \leq \quad \max\{0, SD_f\ e_f\ (n+m)\} * (L^f(\varphi) - 2) + s_{max}$$

We can rewrite (B.18) as follows:

$$\begin{array}{rcl} s & \leq & \max\{\{SD_f\ e_f\ (n+m) + \max\{0, SD_f\ e_f\ (n+m)\} * (L^f(\varphi) - 2) + s_{max}\} \cup \{s_{max}\} \\ & = & \max\{0, SD_f\ e_f\ (n+m)\} * (L^f(\varphi) - 1) + s_{max} \end{array}$$

which proves the Lemma.

□

Finally, the required result follows from this theorem. We have to prove that σ actually approximates s_{max}.

Theorem B.12. *Let σ = computeSigma $(f\ \overline{x_i}^n\ @\ \overline{r_j}^m = e_f)\ \Sigma\ (n+m)$ len. If the following conditions hold:*

1. *Σ is a correct signature for all the functions being called from f.*
2. *len is a correct approximation of the maximum length of the call tree of f.*
3. *The space cost functions σ occurring in the definition of computeSigma is parameter-decreasing.*

Then σ is correct with respect to f.

Proof. (Sketch) Similarly to Corollaries B.6 and B.8, it follows from Theorem B.11. Now σ is a correct bound to s_{max}. By assuming a judgement:

$$\varphi \equiv E \vdash h, k, td, e_f \Downarrow h', k, v, (\delta, m, s)$$

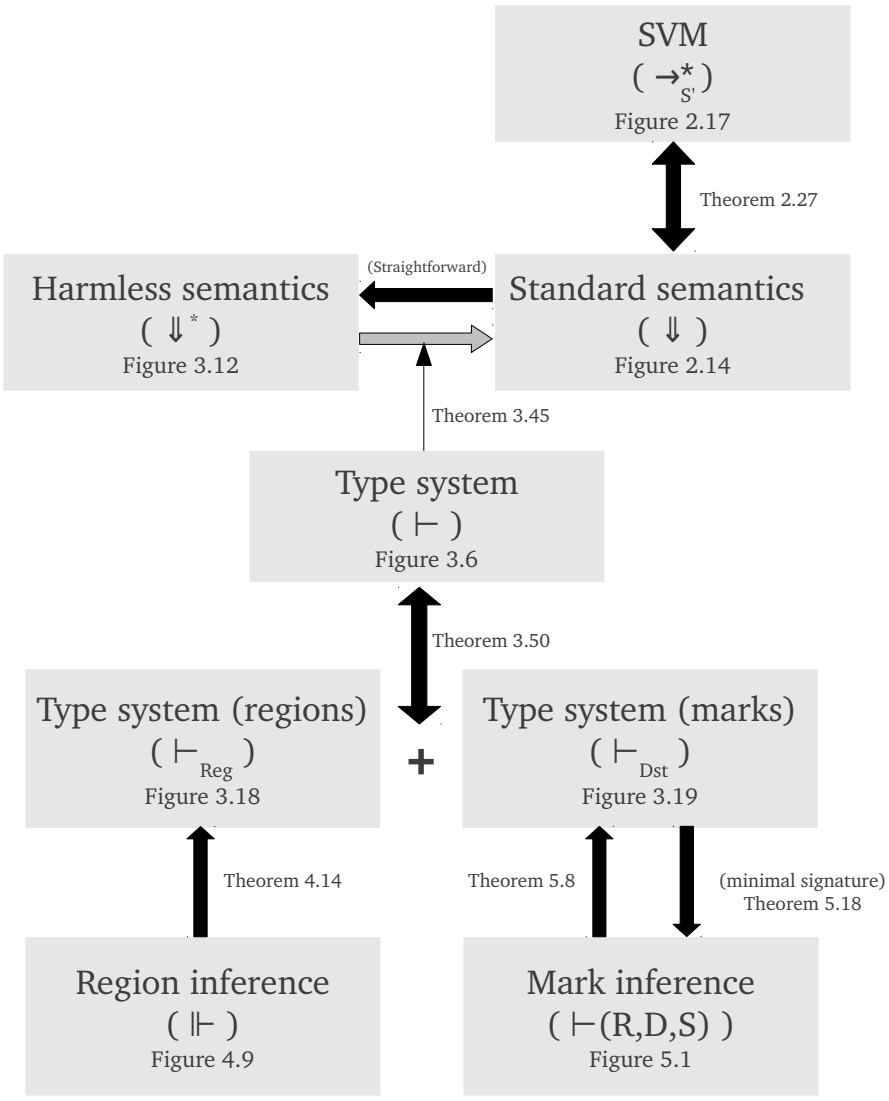
and defining, for each $i \in \{1..n\}$, $s_i = size(h, E(x_i))$, we obtain:

$$\begin{array}{rcl} \sigma & \sqsupseteq & \sqcup\{0, SD_f\ e_f\ (n+m)\} * (len-1) + \sigma \\ & \succeq_{\overline{s_i}^n} & \max\{0, SD_f\ e_f\ (n+m)\} * L^f(\varphi) + s_{max} \\ & \geq & s \end{array}$$

□

Appendix C

Map of semantic definitions and type systems



Bibliography

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP'03*, pages 8–19. ACM, 2003.

[2] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, PLDI'95*, pages 174–185. ACM, 1995.

[3] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In Springer Berlin / Heidelberg, editor, *Static Analysis: 15th International Symposium, SAS 2008.*, pages 221–237, 2008.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

[5] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost analysis of Java bytecode. In *16th European Symposium on Programming*, volume 4421, pages 157–172. Lecture Notes in Computer Science. Springer, March 2007.

[6] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Costa: Design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects, FMCO 2008*, volume 5382, pages 113–132. Springer, 2008.

[7] Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In *Proceedings of the 2010 international symposium on Memory management, ISMM'10*, pages 121–130. ACM Press, 2010.

[8] Elvira Albert, Samir Genaim, and Abu Naser Masud. More precise yet widely applicable cost analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI'11*, pages 38–53. Springer, 2011.

[9] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resources. *Theoretical Computer Science*, 389:411–445, 2007.

[10] David Aspinall and Martin Hofmann. Another type system for in-place update. In *11th European Symposium on Programming, ESOP 2002*, pages 36–52. Springer, 2002.

[11] David Aspinall, Martin Hofmann, and Michal Konečný. A type system with usage aspects. *Journal of Functional Programming*, 18(2):141–178, 2008.

[12] Franz Baader and Tobias Nipkow. *Term rewriting and all that*, chapter 4, pages 73–79. Cambridge University Press, 1998.

[13] Roberto Bagnara, Alessandro Zaccagnini, and Tatiana Zolo. The automatic solution of recurrence relations I: Linear recurrences of finite order with constant coefficients. Quaderno 334, Dipartimento di Matematica, Università di Parma, 2003.

[14] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. Mobius: Mobility, ubiquity, security. In *Trustworthy Global Computing*, pages 10–29. Springer-Verlag, 2007.

[15] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. *ACM Transactions on Programming Languages and Systems*, 31(5):18/1–18/45, 2009.

[16] Florence Benoy and Andy King. Inferring argument size relationships with CLP($\mathcal{R}$). In *Proceedings of the 6th International Workshop on Logic Programming Synthesis and Transformation, LOPSTR'96*, pages 204–223. Springer, 1997.

[17] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '02*, pages 1–12. ACM, 2002.

[18] Lennart Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *Proceedings of 11th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'04*, pages 347–362. Springer, 2005.

[19] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'96*, pages 171–183. ACM, 1996.

[20] Richard Bornat. Proving pointer programs in hoare logic. In *Proceedings of the 5th International Conference on Mathematics of Program Construction, MPC'00*, pages 102–126. Springer, 2000.

[21] Gérard Boudol. Typing safe deallocation. In *Proceedings of the 17th European Symposium on Programming, ESOP 2008*, pages 116–130. Springer, 2008.

[22] Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[23] Christopher W. Brown. QEPCAD. quantifier elimination by partial cylindrical algebraic decomposition. http://www.usna.edu/Users/cs/qepcad/B/QEPCAD.html.

[24] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Space invading systems code. In *Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08*, pages 1–3. Springer, 2009.

[25] Brian Campbell. Prediction of linear memory usage for first-order functional programs. In *Trends in Functional Programming. Volume 9. Selected papers of the 9th Symposium on Trends in Functional Programming, TFP'08*. Intellect, 2008.

[26] Brian Campbell. *Type-based amortised stack memory prediction*. PhD thesis, Laboratory for Foundations of Computer Science. School of Informatics. University of Edinburgh, 2008.

[27] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the 4th International Symposium on Memory Management, ISMM'04*, pages 85–96. ACM, 2004.

[28] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. *ACM SIGPLAN Notices*, 39(6):243–254, 2004. Extended version of PLDI '04.

[29] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher Order and Symbolic Computation*, 14(2–3):261–300, 2001.

[30] Christopher Colby, Peter Lee, George C. Necula, Fred Blau, Mark Plesko, and Kenneth Cline. A certifying compiler for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Languages Design and Implementation, PLDI'00*, pages 95–107. ACM, 2000.

[31] Jesús Conesa, Ricardo López, and Ángel Lozano. Desarrollo de un compilador para un lenguaje funcional con gestión explícita de memoria. Master's thesis, Universidad Complutense de Madrid, 2006.

[32] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 17, pages 451–463. MIT Press, third edition, 2009.

[33] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'77*, pages 238–252. ACM Press, 1977.

[34] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '82*, pages 207–212. ACM Press, 1982.

[35] Javier de Dios. *Certificación de propiedades en un lenguaje funcional impaciente*. PhD thesis, Universidad Complutense de Madrid, 2011.

[36] Javier de Dios, Manuel Montenegro, and Ricardo Peña. Certified absence of dangling pointers in a language with explicit deallocation. In *Proceedings of the 8th International Conference on Integrated Formal Methods, IFM'10*, pages 305–319. Springer, 2010.

[37] Javier de Dios, Manuel Montenegro, and Ricardo Peña. Certified absence of dangling pointers in a language with explicit deallocation. Technical report, Dpto. de Sistemas Informáticos y Computación. Universidad Complutense de Madrid., 2010. Available at: http://dalila.sip.ucm.es/safe/papers/ifm10_extended.pdf.

[38] Javier de Dios and Ricardo Peña. A certified implementation on top of the Java virtual machine. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems, FMICS'09*, pages 181–196. Springer, 2009.

[39] Javier de Dios and Ricardo Peña. Formal certification of a resource-aware implementation. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOL'09*, pages 196–211. Springer, 2009.

[40] Javier de Dios and Ricardo Peña. Certification of safe polynomial memory bounds. In *Proceedings of the 17th International Symposium on Formal Methods, FM 2011*. Springer, 2011. To appear.

[41] Alberto de la Encina and Ricardo Peña. From natural semantics to C: A formal derivation of two STG machines. *Journal of Functional Programming*, 2008. To appear.

[42] Chris Dornan, Isaac Jones, and Simon Marlow. *Alex User Guide*, 2005.

[43] Marko van Eekelen, Olha Shkaravska, Ron van Kesteren, Bart Jacobs, Erik Poll, and Sjaak Smetsers. AHA: Amortized space usage analysis. In *Trends in Functional Programming. Volume 8. Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'07*, pages 36–53. Intellect, 2008.

[44] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, PLDI'93*, pages 237–247. ACM, 1993.

[45] Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *15th European Symposium on Programming, ESOP 2006*, pages 7–21. Springer, 2006.

[46] D. Gaertner and W. E. Kluge. π-RED$^+$ – an interactive compiling graph reduction system for an applied λ-calculus. *Journal of Functional Programming*, 6(5):723–757, 1996.

[47] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated termination tools with AProVE. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications, RTA'04*, pages 210–220. Springer, 2004.

[48] Gudmund Grov, Greg Michaelson, Christoph Herrmann, Hans-Wolfgang Loidl, Steffen Jost, and Kevin Hammond. Hume cost analyses for imperative programs. In *Proceedings of International Conference on Software Engineering Theory and Practice, SETP 2009*, 2009.

[49] Sumit Gulwani. SPEED: Symbolic complexity bound analysis. invited talk. In *Proceedings of the 21st International Conference on Computer Aided Verification, CAD 2009*, pages 51–62. Springer, 2009.

[50] Kevin Hammond and Greg Michaelson. Hume: A domain-specific language for real-time embedded systems. In Frank Pfenning and Yannis Smaragdakis, editors, *Proceedings of Generative Programming and Component Engineering, Second International Conference (GPCE 2003)*, volume 2830, pages 37–56. Springer, 2003.

[51] Kevin Hammond and Greg Michaelson. Predictable space behaviour in FSM-Hume. In *Implementation of Functional Languages 2002*, pages 1–16, 2003.

[52] John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[53] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253, 1993.

[54] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 175–186. ACM, 2001.

[55] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL'11*, pages 357–370. ACM, 2011.

[56] Jan Hoffmann and Martin Hofmann. Amortized resource analysis with polynomial potential. In *Proceedings of the 19th European Symposium on Programming, ESOP 2010*, pages 287–306. Springer, 2010.

[57] Martin Hofmann. A type system for bounded space and functional in-place update–extended abstract. *Nordic Journal of Computing*, 7(4):258–289, Autumn 2000.

[58] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–197, 2003.

[59] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *Proceedings of the 15th European Symposium on Programming, ESOP 2006*, pages 22–37. Springer, 2006.

[60] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages, HOPL III*, pages (12–1)–(12–55), 2007.

[61] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ML programming. In *Proceedings of the 4th ACM SIGPLAN international conference on Functional programming, ICFP'99*, pages 70–81. ACM, 1999.

[62] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'96*, pages 410–423. ACM Press, 1996.

[63] Steffen Jost. *Automated Amortised Analysis*. PhD thesis, Fakultät für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München, August 2010.

[64] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'10*, pages 223–236. ACM, 2010.

[65] Werner Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer Texts in Theoretical Computer Science, 2005.

[66] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL'99*, pages 29–42. ACM, 1999.

[67] Michal Konečný. LFPL with types for deep sharing. Technical Report EDI-INF-RR-157, LFCS, Division of Informatics, University of Edinburgh, October 2002.

[68] Michal Konečný. Typing with conditions and guarantees for functional in-place update. In *Selected papers from the 2nd International Workshop on Types for Proofs and Programs, TYPES 2002*, pages 182–199. Springer, 2002.

[69] Michal Konečný. Functional in-place update with layered datatype sharing. In *Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications, TLCA'03*, pages 195–210. Springer, 2003.

[70] Peter Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[71] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization, CGO'04*, pages 75–87. ACM, 2004.

[72] Salvador Lucas. MU-TERM: A tool for proving termination of context-sensitive rewriting. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications, RTA'04*, pages 200–209. Springer, 2004.

[73] Salvador Lucas and Ricardo Peña. Rewriting techniques for analysing termination and complexity bounds of Safe programs. In *Draft Proceedings of 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08*, pages 43–57, 2008.

[74] David C. Luckham and Norihisa Suzuki. Verification of array, record, and pointer operations in pascal. *ACM Transactions on Programming Languages and Systems*, 1(2):226–244, 1979.

[75] Henning Makholm. *A language-independent framework for region inference*. PhD thesis, University of Copenhagen. Departament of Computer Science, August 2003.

[76] Simon Marlow et al. *Haskell 2010 Language Report*, 2010.

[77] Simon Marlow and Andy Gill. *Happy User Guide*, 2001.

[78] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Compututation*, 199(1-2):200–227, 2005.

[79] Manuel Montenegro. Inferencia de tipos seguros en un lenguaje funcional con destrucción explícita de memoria. Master's thesis, Universidad Complutense de Madrid, September 2007.

[80] Manuel Montenegro, Ricado Peña, and Clara Segura. Experiences in developing a compiler for safe using Haskell. In *Actas del I Taller de Programación Funcional, TPF'09*, pages 31–46, 2009.

[81] Manuel Montenegro, Ricardo Peña, and Clara Segura. An inference algorithm for guaranteeing safe destruction. In *Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR'08*, pages 135–151. Springer, 2008.

[82] Manuel Montenegro, Ricardo Peña, and Clara Segura. A resource-aware semantics and abstract machine for a functional language with explicit deallocation. In *Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming, WFLP'08*, pages 167–182. Elsevier, 2008.

[83] Manuel Montenegro, Ricardo Peña, and Clara Segura. A space consumption analysis by abstract interpretation. In *Proceedings of the 1st International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA'09*, pages 34–50. Springer, 2010.

[84] Manuel Montenegro, Ricardo Peña, and Clara Segura. A type system for safe memory management and its proof of correctness. In *Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, PPDP'08*, pages 152–162. ACM Press, 2008.

[85] Manuel Montenegro, Ricardo Peña, and Clara Segura. A simple region inference algorithm for a first-order functional language. In *Proceedings of the 18th International Workshop on Functional and (Constraint) Logic Programming, WFLP'09*, pages 145–161. Springer, 2009.

[86] Manuel Montenegro, Ricardo Peña, and Clara Segura. A resource-aware semantics and abstract machine for Safe. A functional language with regions and explicit deallocation, 2011. Submitted.

[87] Manuel Montenegro, Olha Shkaravska, Marko van Eekelen, and Ricardo Peña. Interpolation-based height analysis for improving a recurrence solver. In *Draft Proceedings of the 2nd International Workshop on Foundational Practical Aspects of Resource Analysis (FOPARA 2011). Technical Report SIC-08/11. Dpto. de Sistemas Informáticos y Computación. Universidad Complutense de Madrid*, pages 95–110, 2011.

[88] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'97*, pages 106–119, 1997.

[89] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998.

[90] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the second USENIX symposium on Operating Systems, Design and Implementation, OSDI'96*, pages 229–243. ACM, 1996.

[91] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI'98*, pages 333–344. ACM, 1998.

[92] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[93] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[94] Manuel Núñez, Pedro Palao, and Ricardo Peña. A second year course on data structures based on functional programming. In *First international symposium on Functional Programming Languages in Education, FPLE'95*, pages 65–84. Springer Verlag, 1995.

[95] Martin Odersky. Observers for linear types. In *Proceedings of the 4th European Symposium on Programming, ESOP'92*, pages 390–407. Springer, 1992.

[96] Lars Pareto. Sized types, 1998. Licentiate thesis, Chalmers University of Technology.

[97] Ricardo Peña and Agustin D. Delgado. Size invariant and ranking function synthesis in a functional language. In *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming, WFLP 2011*, pages 52–67. Springer, 2011.

[98] Ricardo Peña, Clara Segura, and Manuel Montenegro. A sharing analysis for Safe. In *Trends in Functional Programming (Volume 7). Selected Papers of the Seventh Symposium on Trends in Functional Programming, TFP'06*, pages 109–128. Intellect, 2007.

[99] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[100] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS'02*, pages 55–74. ACM, 2002.

[101] John A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[102] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL'88*, pages 285–293. ACM, 1988.

[103] Donald Sannella, Martin Hofmann, David Aspinall, Stephen Gilmore, Ian Stark, Lennart Beringer, Hans-Wolfgang Loidl, Kenneth MacKenzie, Alberto Momigliano, and Olha Shkaravska. Mobile resource guarantees. In *Trends In Functional Programming, volume 6. Selected papers of the 6th International Symposium on Trends in Functional Programming, TFP 2005*, pages 211–226. Intellect, 2005.

[104] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

[105] Olha Shkaravska, Marko van Eekelen, and Ron van Kesteren. Polynomial size analysis of first-order shapely functions. *Logical Methods in Computer Science*, 5(2):1–35, 2009. Special Issue with Selected Papers from TLCA 2007.

[106] Olha Shkaravska, Rody Kersten, and Marko van Eekelen. Test-based inference of polynomial loop-bound functions. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ 2010*. ACM, 2010.

[107] Olha Shkaravska, Marko van Eekelen, and Alejandro Tamalet. Collected size semantics for functional programs over lists. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, pages 1–21. Springer, 2008.

[108] Fausto Spoto, Patricia M. Hill, and Étienne Payet. Path-length analysis for object-oriented programs. In *EAAI'06: First International Workshop on Emerging Applications of Abstract Interpretation*, 2006.

[109] Alejandro Tamalet, Olha Shkaravska, and Marko van Eekelen. Size analysis of algebraic data types. In *Trends in Functional Programming Volume 9. Selected papers of the 9th Symposium on Trends in Functional Programming, TFP'08*, pages 33–48. Intellect, 2009.

[110] Robert E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6:306–318, 1985.

[111] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[112] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(5):724–767, 1998.

[113] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions for the MLKit. Technical report, University of Copenhagen, 2006.

[114] Mads Tofte and Niels Hallenberg. Region-based memory management in perspective. In Henning Makholm Fritz Henglein, John Hughes and Henning Niss, editors, *Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE 2001 (Invited Talk)*, pages 23–30, 2001.

[115] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, POPL'94*, pages 188–201. ACM, 1994.

[116] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[117] Pedro B. Vasconcelos and Kevin Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proceedings of the 15th International Workshop on Implementation of Functional Languages, IFL'03*, pages 86–101. Springer, 2004.

[118] Philip Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.

[119] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.