Cómo eliminar evidencia resolviendo restricciones para producir automáticamente programas tipados

Hernán Badenes

Trabajo final para obtener el grado de

Licenciado en Informática

de la

Facultad de Informática,
Universidad Nacional de La Plata,
Argentina

Director: M.Sc. Pablo E. Martínez López
Co-director: Prof. Gabriel A. Baum

La Plata, Agosto de 2003

Índice general

Agradecimientos

Mientras miro las nuevas olas,
yo ya soy parte del mar...

Charly García – Serú Girán

Siendo este trabajo la culminación de una importante etapa de una carrera, y esta sección precisamente el lugar donde mencionar los agradecimientos, es difícil abarcar a todas aquellas personas que tomaron parte en mi formación y de una forma u otra ayudaron para que este trabajo esté ahora escrito.

Intentando no olvidar a ninguno, en una lista donde cada uno podría aparecer o no aparecer, me quedo con la opción de no mencionar a nadie explícitamente y dejar que cada uno adivine si fue en realidad nombrado, y en qué medida... dando incluso la oportunidad a quien lo quiera de suponer que está en la lista.

Aunque mucho anterior a mi estadía en la facultad, y fuera del ámbito académico, primero está mi familia; quienes me hicieron crecer sin que nunca sintiera la falta de nada, apoyándome e impulsándome a cualquiera de mis iniciativas.

También influyeron en mi formación, aunque nunca haya podido decírselos, los maestros que año tras año siguen trabajando en la OMA, despertando la matemática en muchos que siempre les estaremos agradecidos.

Obtuve mucho de aquellos con quienes pude tomar cursos fuera de la facultad. En Río IV, la UBA, en Tandil; científicos y profesores (nacionales e importados) que transmitían y transmiten su ciencia con la satisfacción de encontrar oídos en este país, y contagiándome por un tiempo su entusiasmo por sus trabajos. En muchos casos, argentinos casi exiliados por nuestra política (o carencia de ella) hacia la ciencia nacional estudian y ejercen fuera del país pero siguen volviendo para poder transmitir sus experiencias.

En la facultad no faltaron tampoco profesores que me dieron mucho a cambio de poco; tuve la oportunidad de elegir ejemplos a seguir (también contraejemplos).

Dentro del LIFIA, al que empecé a entrar casi al mismo tiempo que a la carrera, hay una variedad de personas a las que agradecer; sobre todos aquellos

que se dedican a pensar sobre sus objetivos: ciencia y docencia, y se dedican por completo a ejercerlas o permitir que lo sean. Casi en el mismo ámbito, a mis compañeros de Programación Funcional, los de hace años y los de ahora, de quienes mucho pude aprender.

Desde aquel momento en mi primer año en la facultad, hasta ahora en que firma este trabajo, agradezco a mi director, colega y amigo por su constante apoyo, la contagiosa e incansable pasión que pone en su trabajo, su inacabable paciencia, y también (cómo olvidarlo) a su sello negro; no sólo conmigo sino con todo aquél cuyo camino se cruza con él.

También, y no hubiera sido lo mismo sin ellos (tal vez ni siquiera hubiera sido), a mis amigos y compañeros. A aquél por quien entré (como persiguiéndolo) a esta carrera, a todos ellos que corrieron conmigo y a quienes vi correr queriendo poder imitarlos. Por todas esas interminables horas de estudio, por los kilos de apuntes compartidos (casi nunca escritos por mí), por las miles de líneas de código y esas viciosas redes maratónicas.

También agradezco a los jurados anónimos que con sus críticas contribuyeron a pulir la publicación de estos resultados.

Y finalmente (sólo por guardarme lo mejor para este momento) a mis soles, por quienes tuve razones de sobra para demorar tanto en este trabajo y por quienes ahora lo termino. Por quienes vivo.

Abstract

Computer science is no more about computers
than astronomy is about telescopes.

Edsger Dijkstra

This is an undergraduate thesis to obtain the degree of Licentiate in Computer Science in University of La Plata, Argentina. Since by standing rules of the institution this work must be presented in Spanish, the author would like to cross-refer the reader to a shorter English version of this work [Martínez López and Badenes, 2003], that is being published in the Proceedings of the VII Argentinian Workshop on Theoretical Computer Science (WAIT 2003, Buenos Aires, September 2003).
There, the core chapters of this work (Simplification, chapter 2, and Solving, chapter 3) are included (with a lower level of detail).
In any case, we give an English introduction for better understanding the scope of this work.

While computers are more and more present each day in every aspect and detail of our lives, and those who lead this technology speak largely about how evolved their discipline is, it is also true that we are still behindhand in some way with respect to the methodology in which one of the main tasks of this area is performed, viz. programming. Programs, an essential component of every software, written to be interpreted for computers (actually the only thing intelligible by computers) can not, paradoxically, be written by computers themselves. John Hughes, who is the author of many of the ideas contained in this work, stated it (in a course of program specialisation in Montevideo, during 1997) in a brief and accurate phrase: "Programming is a Handraft!".

In the few years of computers' history a variety of programming techniques were discovered, explored, developed and sometimes discarded. But, in essence, it is the same programmer, the *craftsman*, who builds every program to be executed by a computer. More and more 'advanced' tools have been invented and

developed to specify, model, design, implement, analyze, and debug programs. However, the analogy to a 'pipeline' to build general-purpose programs has not been invented yet. Computing has become and reinforced as a science in a few years, but with respect to programming, it has not reached its 'industrial revolution'.

However, many advances in the area allows to mechanize more tasks every time. Due to the exact nature of the computers' behavior, these tasks are not only studied in practice but also, and perhaps more stressed, in the theoretical field, building the basis for the creation of tools and their supporting theories. Theoretical studies, as the search for properties, formal specifications, computability and complexity, correctness, semantics, etc. has shown to be not only a mathematicians' yearning but an essential tool for the progress of computer science.

My thesis is developed in this frame, contributing with a small sand grain to the formal study of the way of automatic program production.

And automatic program production is not a useless nor utopistic idea. Cost, in money and time, of programming any piece of software, and the higher probability of human mistakes introduced as errors (and whose cost is measurable and represents a significative portion of software developing budget) justifies research towards this automatization. And it is not absurd to imagine a scenario in which computers program themselves: nowadays there are a limited number of areas in which this is already performed (as domain specific languages or user interface generation), starting from a proper specification.

One of the ways of automatically producing programs is the approach known as *program specialisation*. This area, as a technique for manipulating programs, is studied in the formal field, existing several approaches. In this work we shall mention some of them and study in depth one of the newest: *type specialisation*. On this approach we shall study and develop in a formal way some relations, properties and algorithms, contributing to the field with a language in which some open problems can be discussed.

This thesis is structured in six chapters. In the first place, basic notions and problems motivating this work are described. Three areas are briefly visited:

1. type specialisation, introducing the concepts of program specialisation and the common problems coming from the existence of 'inherited limits', summarizing the work of [Hughes, 1996];

2. the qualified types theory, the work of Mark Jones [Jones, 1994a] that defines a general framework to design type systems in which types are generalized with conditions (predicates) that restrict the range of type variables; and

3. principal type specialisation, a reformulation of Hughes' work developed by Martínez López and himself [Martínez López and Hughes, 2002] aiming to solve the problem of principality of solutions, and to which this work contributes with a postprocessing phase.

These three main items are the subject of Chapter 1, as a summary of main bibliography and with the minimal level of detail to continue reading the rest of this work.

The next two chapters are the core of this work. Chapter 2 defines a relation among predicates of qualified types, in order to determine when a predicate assignment is 'more general' than other or, in other words, when a simplification can be done in an specialisation judgment. This *simplification* removes predicates created during specialisation replacing type variables and properly modifying terms. This simplification relation is defined in terms of an important relation of the underlying qualified theory: entailment. A minimalistic simplification relation is built, to be used as a base for any other simplification for a given language. Also the soundness of this process w.r.t. specialisation is proved, adding a new rule to the set of principal specialisation rules.

The next chapter, the most important and which gives the title to this work, uses the simplification relation presented to define and give a solution to the *constraint solving* problem. This problem consists of determining the way in which scheme variables contained in residual types can be replaced by their final values (solutions), a task that specialisation only formulates in terms of the creation of predicates. The concept of constraint solving is formally defined and a proof of soundness with respect to the principal specialisation system is given in the same way as done in the previous chapter. Then a simple algorithm to solve constraints is implemented, as a simple example appointed to solve the predicates generated during the specialisation of the language studied.

In both chapters every decision is properly justified and different alternatives are considered, also relating this work with other works in the area.

Chapter 4 is about an application of the notions studied in previous chapters. It discusses the problem of *evidence elimination*, a postprocessing task necessary after principal specialisation. The objective of this phase is to 'clean' terms of residual expressions coming from the representation of specialisation, namely evidence used to prove predicates. Programs are in this way more legible and similar to those that would be obtained if manually written, taking them to the same language of that of original specialisation system. This chapter can be seen too as an example of a use of the constraint solving system, that allows to obtain 'better' programs as an added-value, by a simple instance of the constraint solving framework.

Then final chapters are reached. Chapter 5 remarks in the first place the implementation job of all previous tasks. They are implemented as modules in the prototype of principal type specialisation of Martínez López, modifying his original implementations (implemented at first without formal specifications). Being the practical edge of this work, it allows also to observe the studied notions as they work, embedded in a whole implementation of type specialisation. As this is an open field, by no means depleted, most of this work is devoted to mention different extensions and future tasks to be made after this work. Finally, chapter 6 gives the conclusions of having done this whole work, and the contributions to the area.

Additionally, an appendix with the proofs of all theorems is included. This is separately presented to leave previous chapters clearer; its reading is optional but very advisable for those who want to understand every detail of this work in order to make further contributions.

Introducción

Computer science is no more about computers
than astronomy is about telescopes.

Edsger Dijkstra

Mientras las computadoras están cada vez más presentes en cualquier aspecto de la vida cotidiana y aquellos que lideran esta tecnología se jactan de lo evolucionada que está esta disciplina, es también cierto que aún estamos (es difícil saber por cuánto tiempo) atrasados en cierta forma respecto a la metodología de una las tareas fundamentales para su uso: la programación. Paradójicamente los programas, la componente esencial de todo software, escritos sólo para ser entendidos por computadoras (y siendo la única cosa inteligible por ellas) no pueden ser fabricados por ellas mismas. John Hughes, autor de parte de las ideas contenidas en esta tesis, lo expuso muy concisamente en un curso de especialización de programas en Montevideo, durante 1997: *Programming is a handcraft!*

En pocos años se desarrollaron, descubrieron, exploraron y en ocasiones descartaron multitudes de técnicas de programacíón. Pero en esencia, sigue siendo el programador el *artesano* que construye todo programa que vaya a ser ejecutado por una computadora. Se han inventado y construido herramientas cada vez más 'avanzadas' para especificar, modelar, diseñar, implementar, analizar y depurar, como parte de la tarea de programar. Mas no se ha inventado la línea de montaje, la fábrica en la que se produzcan los programas de uso general. La computación se ha consolidado como ciencia en sus pocos años de vida, pero en materia de programación, aún no ha llegado a su 'revolución industrial'.

No obstante, los avances logrados permiten un número cada vez mayor de tareas automatizables. Por la naturaleza exacta del funcionamiento de una computadora y la ejecución de los programas que interpreta, estas tareas no se estudian sólo en la práctica sino también, y con igual o mayor importancia, en el campo formal, sentando las bases para la creación de herramientas y teorías que las soporten. El estudio teórico, como la búsqueda de propiedades, especificaciones formales, la computabilidad y complejidad, la correctitud y semántica, etc. ha demostrado no ser un capricho de matemáticos sino una herramienta fundamental para el avance de la ciencia de la computación.

En este marco se desarrolla esta tesis, aportando un grano de arena al estudio formal de la manera de producir programas automáticamente.

La generación automática de programas no es ni una idea inútil ni una utopía. El costo, en tiempo y dinero, de la programación de cualquier pieza de software, sumado a la probabilidad de errores humanos que puedan deslizarse, en forma de *bugs* (y cuyo costo es mesurable y representa una porción no despreciable del total destinado a la producción de software) justifican la investigación en pos de esta automatización. Y no es descabellado imaginar un escenario en donde las propias computadoras se programen a sí mismas; ya en un contado número de áreas (como lenguajes de dominio específico o la generación de interfaces), algunos programas se generan automáticamente a partir de una correcta especificación.

Una de las formas de producir programas automáticamente es la técnica que se conoce como *especialización de programas*. Esta tarea, como técnica de manipulación de programas cuya semántica está definida en forma exacta, se estudia en el ámbito formal, existiendo diversos enfoques. En este trabajo mencionaremos algunos de ellos y estudiaremos en profundidad una de las formas más novedosas en el campo: la *especialización de tipos*. Sobre esta última forma de manipulación de programas estudiaremos y desarrollaremos en un marco formal algunas relaciones, propiedades y algoritmos, aportando al área un lenguaje de discusión sobre el que atacar problemas que se encuentran aún abiertos.

Esta tesis se organiza en seis capítulos. En primer lugar se definen las nociones básicas y los problemas que motivan este trabajo. Se visitan brevemente tres áreas principales:

1. especialización de tipos, introduciendo los conceptos de especialización de programas y los problemas comunes dados por la existencia de límites heredados e introduciendo el trabajo de John Hughes [Hughes, 1996];

2. la teoría de tipos calificados, el trabajo de Mark Jones [Jones, 1994a] que define un marco general de trabajo para diseñar sistemas de tipos generalizados con condiciones o predicados que restringen el rango de variables de tipo; y

3. la especialización principal de tipos, una reformulación del trabajo de John Hughes realizada en conjunto con Pablo E. Martínez López [Martínez López and Hughes, 2002] con el objetivo de resolver el problema de principalidad de especializaciones, y a la cual este trabajo aporta el estudio formal de una etapa de postprocesamiento.

Estos tres temas principales componen el capítulo 1, a modo de resumen de la bibliografía principal y con el nivel de detalle mínimo necesario para continuar con el resto del trabajo.

Los siguientes dos capítulos componen el corazón de este trabajo. El capítulo 2 define una relación entre predicados de los tipos calificados, en pos de especificar cuándo un contexto es más específico que otro, o en otras palabras, cuándo y cómo puede simplificarse un juicio de especialización - esta *simplificación* elimina predicados creados durante la especialización reemplazando variables de tipo y convirtiendo términos en forma acorde. Se define esta relación de simplificación en términos de las relaciones de la teoría de tipos calificados subyacente y se construye una relación general como base a cualquier simplificación de un lenguaje especializado, junto con algunos refinamientos para el lenguaje utilizado como ejemplo. Además se prueba la consistencia de esta tarea respecto a la de especialización principal, agregando una nueva regla al sistema que la define.

El siguiente capítulo, el más importante y que da el título a este trabajo, se basa en la simplificación presentada para definir y resolver el problema de *resolución de restricciones*. Este problema consiste en determinar la manera en que las variables de esquema contenidas en los tipos pueden ser reemplazadas por sus valores finales (soluciones), tarea que la especialización en sí sólo formula en términos de la creación de predicados. Se define formalmente el concepto de solución y se prueba su consistencia respecto del sistema de especialización principal, en forma análoga al trabajo de simplificación del capítulo anterior. A continuación se desarrolla un algoritmo simple de resolución de restricciones como ejemplo minimal destinado a resolver los predicados generados para el lenguaje de especialización en estudio.

En ambos capítulos se justifica cada decisión y se contemplan diferentes alternativas a las tomadas, además de relacionar este trabajo con otros existentes.

El capítulo 4 trata sobre una aplicación de las nociones estudiadas en los capítulos anteriores. Este capítulo aborda el problema de *eliminación de evidencia*, una tarea de postprocesamiento necesaria luego de la especialización principal. El objetivo de esta tarea es 'limpiar' los términos obtenidos de expresiones residuales de la simplificación de los predicados. Los programas de esta forma quedan más legibles y similares a la forma que tendrían de ser programados manualmente, llevándose además al mismo lenguaje residual del trabajo original de especialización de tipos. Este capítulo puede verse a la vez como una ejemplo de aplicación del concepto de resolución de restricciones, que permite obtener 'mejores' programas como valor agregado, simplemente como instancia de una relación de resolución particular.

Seguidamente se llega a los capítulos finales. El capítulo 5 comenta en primer lugar el trabajo de implementación de todas las tareas anteriores. Las mismas se implementan como módulos del prototipo de especializador principal de Martínez López, cambiando las implementaciones originales (programadas

originalmente sin una especificación formal). Al ser el lado más práctico de este trabajo, permite además observar las nociones definidas en funcionamiento, embebidas en una implementación completa de un especializador. Como éste es un tema abierto, y de ninguna forma agotado, la mayor parte de este capítulo se dedica exclusivamente a mencionar diferentes extensiones que amplíen este trabajo, alternativas a las decisiones tomadas que pueden ser estudiadas en mayor profundidad y tareas a realizar a futuro que lo continúen. Finalmente en el capítulo 6 se dan las conclusiones que se obtienen de la realización del trabajo completo, y los aportes realizados sobre el área.

En forma adicional, se da un apéndice con las pruebas de todos los teoremas formulados en el trabajo – éstas se dan en forma separada para dejar los capítulos anteriores más limpios; su lectura puede verse como opcional para quienes deseen comprender todos los detalles técnicos como base para continuar el trabajo.

Capítulo 1

Especialización

-El problema contigo, Shev, es que nunca dices nada hasta que has amontonado toda una carga de malditos argumentos, pesados como ladrillos, y entonces los largas todos de golpe, y nunca se te ocurre mirar el cuerpo ensangrentado y maltrecho bajo el montón...

Los Desposeídos (1974)
Úrsula K. Le Guin

A lo largo de este capítulo se resumen los conceptos necesarios para definir este trabajo, y los trabajos previos en que nos basamos.

La sección 1.1 es un resumen de algunos conceptos y resultados que motivan el trabajo en el campo. En 1.1.3 se introduce la noción de *especialización de tipos* como alternativa a los enfoques tradicionales de especialización de programas. La sección 1.2 introduce la *teoría de tipos calificados*, que estudia en forma abstracta propiedades de sistemas de tipos utilizadas informalmente en otros trabajos y que se utilizan en la sección 1.3, *especialización principal de tipos*, para resolver los problemas del sistema de especialización original.

Hay que destacar que todo lo que se muestra en este capítulo se hace a vuelo de pájaro; para una comprensión más profunda se remite al lector principalmente a [Hughes, 1996; Jones, 1994a; Martínez López and Hughes, 2002] además de las referencias adicionales indicadas en el texto.

1.1 Especialización de Programas

Este trabajo se centra proveer herramientas para el estudio de una forma de producir programas automáticamente. La *especialización de programas* es una de las formas de generación automática de programas más estudiada y conocida. En ella, un programa (llamado programa *fuente*) es tomado como entrada, y se producen uno o más programas como salida (programas *residuales*), cada uno especializado para datos particulares.

La motivación de este enfoque proviene de una forma común (y en general deseable) de programación: frente a un requisito, un programador suele realizar una solución que resuelve un problema más general, y luego obtiene la solución deseada aplicando su programa a ciertos datos particulares. El ejemplo clásico es la función recursiva *power* que calcula x^n:

$$\begin{aligned} power\, x\, n \;=\; & \textbf{if } n == 1 \textbf{ then} \\ & \quad x \\ & \textbf{else} \\ & \quad x * power\, (n-1)\, x \end{aligned}$$

Suponiendo que se desea una función que eleve (por ejemplo) al cubo un número dado, la función $power\,3$ contiene cierto número de comparaciones y llamadas recursivas superfluas, pudiendo simplificarse a

$$power_3\, x \;=\; x * (x * x)$$

para este ejemplo. Es claro que esta versión es más eficiente que utilizar *power* aplicada a 3 y un valor x cualquiera.

1.1.1 Enfoques tradicionales

El enfoque más estudiado en el campo de la especialización de programas es la *evaluación parcial* [N.D. Jones and Sestoft, 1993; Consel and Danvy, 1993], en la cual se producen los programas residuales mediante una forma generalizada de reducción: las subexpresiones cuyo valor depende sólo de datos estáticos (conocidos en tiempo de compilación) son reemplazadas por sus evaluaciones, dejando intactas el resto de las expresiones, cuya computación no puede ser realizada todavía. De esta forma el evaluador, tomando el texto de un programa fuente, fija algunos de los datos de entrada (datos *estáticos*) y mezcla evaluación y generación código para producir el nuevo programa (el programa residual).

Esta técnica se aplica exitosamente en el área de producción de compiladores: la compilación se obtiene especializando un intérprete con un programa escrito en el lenguaje sobre el que trabaja. El intérprete juega el papel de programa fuente, y el programa al que se lo aplica conforma los datos estáticos de la especialización. Dando una vuelta más a la tuerca, si el evaluador parcial está escrito

en el mismo lenguaje que especializa, entonces es posible aplicarlo a sí mismo: el código del evaluador es el programa fuente, el interprete sus datos estáticos, y el programa residual hace especializaciones del intérprete mencionado; es decir, es un compilador! Este resultado es conocido como la segunda proyección de Futamura [Futamura, 1971].

1.1.2 Límites heredados

Una noción importante al trabajar con especialización de programas es la de *límites heredados* [Mogensen, 1996; Mogensen, 1998]. Un límite heredado es una limitación en el programa residual impuesta por la estructura del programa fuente y el método de especialización. Es decir, la 'forma' de los programas obtenibles está condicionada a la forma de aquellos especializados. Por ejemplo, si cada función en un programa es especializada por separado (*especialización monovariante*), el número de funciones se convierte en un límite heredado. Otros ejemplos de límites que se pueden encontrar en especializadores son el tamaño de un programa, el número de variables o de tipos, la profundidad en el anidamiento de ciertas estructuras, etc.

Una manera práctica de determinar la presencia o ausencia de límites en un método es especializar un auto-intérprete y comparar los programas residuales con el fuente: si son esencialmente los mismos, se puede decir que no existen límites (ya que no hay rastros del intérprete en ellos). En ese caso, se dice que la especialización es *óptima* (*Jones-optimal*, por el trabajo de Neil Jones [Jones, 1988]).

La evaluación parcial, para auto-intérpretes escritos en lenguajes no tipados, puede llegar a ser óptima. Pero el caso de lenguajes tipados es diferente: dado que la evaluación parcial se realiza por medio de reducciones, el tipo del programa residual es el mismo o una función de aquel del programa fuente; por lo tanto el código residual está restringido por el tipo del programa fuente (puede ser el mismo o un subtérmino, por ejemplo); por lo tanto un contendrá información de tipos proveniente de la representación de programas en el intérprete: no es posible obtener optimalidad. Este problema fue expuesto por Neil Jones en 1987 como uno de los problemas abiertos de la evaluación parcial, y el que quiso resolver John Hughes con su trabajo, comentado en la siguiente sección.

Aunque un trabajo reciente sugiere que ciertas formas de ver esta situación evitan este problema [Danvy and Martínez López, 2003], en este trabajo tomaremos el enfoque tradicional.

1.1.3 Especialización de Tipos

'Especialización de tipos', la -pobre- traducción que utilizaremos para *Type Specialisation*, es un enfoque de especialización de programas introducido por John

Hughes en 1996 [Hughes, 1996]. La idea principal es especializar el programa fuente *junto con su tipo* a un programa y un tipo residuales. La especialización se hace mediante una forma generalizada de inferencia de tipos – en el tipo residual se agregan construcciones más expresivas para capturar las características de la especialización como información estática y dinámica.

Lenguaje fuente

Como lenguaje fuente se utiliza λ-cálculo enriquecido con constantes aritméticas y booleanas, operaciones entre ellas, definiciones locales (*let*) y tuplas, todas ellas con anotaciones $\_^S$ o $\_^D$ para diferenciar la información estática de la dinámica, más tres nuevas anotaciones (**lift**, **poly**, **spec**) cuyo efecto se explicará más adelante. Los términos del lenguaje fuente, denotados e, se definen por la siguiente gramática:

$$\begin{array}{lllll} e ::= & x & \mid n^D & \mid e +^D e \\ & \mid \textbf{lift}\ e & \mid n^S & \mid e +^S e \\ & \mid \lambda^D x.e & \mid e\ @^D\ e & \mid \textbf{let}^D\ x = e\ \textbf{in}\ e \\ & \mid (e, \ldots, e)^D & \mid \pi^D_{n,n}\ e & \\ & \mid \textbf{poly}\ e & \mid \textbf{spec}\ e & \end{array}$$

Donde n es un número natural, $(e, \ldots, e)^D$ es una tupla finita para cada posible aridad, y x varía sobre un conjunto infinito numerable de variables.

Antes de continuar, una aclaración acerca de la notación usada en todo este trabajo. En cualquier caso en donde se utilicen variables sobre cualquier conjunto que describimos, por ejemplo el lenguaje dado por la gramática anterior, el nombre de las variables es importante: la letra con que se nombra cada variable indica su *categoría sintáctica*, denotando el conjunto sobre el cual toma su valor. Por ejemplo, en la gramática de términos anterior, la letra e se uso para nombrar términos, y así será en el resto del trabajo: cualquier variable nombrada e, e_0, e_1, e_i, etc. denotará un término de este lenguaje. Asimismo en la gramática anterior se introducen las x como variables sobre un conjunto infinito de objetos del lenguaje fuente y n como un número natural. Oportunamente se seguirán introduciendo nuevas letras para diferentes conjuntos.

En este lenguaje los términos aparecen anotados con $\_^S$ o $\_^D$ indicando su naturaleza. Se espera que los términos estáticos sean eliminados por el especializador, y su información *movida a su tipo*, mientras que los dinámicos deben perdurar en el programa residual. Tales anotaciones son provistas por el programador, y tomadas como entrada por el especializador gobernarán la forma en que un término se especializa.

La construcción **lift** es una coerción de enteros estáticos a dinámicos; **poly** anota una expresión como *polivariante*, esto es, que puede tomar diferentes formas en el programa residual (de acuerdo al tipo); y la anotación **spec** se aplica

a términos polivariantes para producir las diferentes especializaciones de los mismos.

El lenguaje de tipos fuente también refleja la naturaleza estática o dinámica de las expresiones. El tipo de constantes, funciones y operadores debe ser consistente con la anotación de la expresión. Las expresiones de tipo τ se definen como:

$$\tau ::= Int^D \mid Int^S \mid (\tau, \ldots, \tau)^D \mid \tau \rightarrow^D \tau \mid \textbf{poly}\ \tau$$

(Notar que, de acuerdo a las categorías sintácticas introducidas en los párrafos anteriores, a partir de ahora τ denotará expresiones de tipos del lenguaje fuente. En adelante se omitirán este tipo de comentarios.)

El lenguaje fuente presentado aquí es un subconjunto del original de [Hughes, 1996], pero contiene suficientes construcciones para capturar los problemas que apuntamos a resolver en este trabajo (como se verá en la ejemplo 1.4 uno de los problemas que resolveremos se introducen por el uso de polivarianza). En la sección 5.2 se mencionarán algunas extensiones para trabajar con construcciones adicionales como recursión dinámica y estática, funciones estáticas y *datatypes*.

Lenguaje residual

El lenguaje residual contiene todas las construcciones y tipos correspondientes a las expresiones dinámicas del lenguaje fuente, más algunas adicionales para representar el resultado de especializar construcciones estáticas.

En su formulación original, los términos e' del lenguaje residual contienen construcciones correspondientes a expresiones dinámicas, la constante nula "$\bullet$" (*void*) proveniente de la especialización de una expresión estática, y tuplas y proyecciones generadas por la polivarianza y su especialización. (Hughes no distingue en su formulación entre el residuo de las tuplas dinámicas y el de las expresiones polivariantes mediante tuplas estáticas y dinámicas, por lo que expresiones de cualquiera de los dos tipos pueden ser eliminadas por la fase de postprocesamiento llamada *arity raising*.)

$$\begin{array}{rl} e' ::= & x' \mid n \mid e' + e' \mid \bullet \\ & \mid \lambda x'.e' \mid e'@e' \mid \textbf{let}\ x' = e'\ \textbf{in}\ e' \\ & \mid (e'_1, \ldots, e'_n) \mid \pi_{n,n}\, e' \end{array}$$

Los tipos del lenguaje residual, cuyos elementos se denotan τ', son elementos del lenguaje representado por:

$$\tau' ::= Int \mid \hat{n} \mid \tau' \rightarrow \tau' \mid (\tau', \ldots, \tau')$$

La novedad es el uso de infinitos tipos unitarios o *singletons*: $\hat{n}$ es el tipo residual de una expresión con valor estático conocido n.

Especializaciones

En este sistema se especifica la especialización mediante una forma general de inferencia de tipos, con juicios de la forma

$$\Gamma \vdash e : \tau \hookrightarrow e' : \tau'$$

que indican que el programa fuente e de tipo τ se puede especializar a un programa residual e' de tipo τ', bajo las hipótesis de la asignación Γ (que contiene suposiciones de la forma $x : \tau \hookrightarrow x' : \tau'$ para especializar las variables libres de e).

En lugar de presentar el sistema de reglas que especifican el sistema original de especialización, ilustraremos mediante ejemplos su capacidad y limitaciones, que motivan la creación de un sistema con especializaciones *principales*, en el que se basa nuestro trabajo.

Ejemplo 1.1. En este sistema se pueden derivar los siguientes juicios de especialización.

1. $\vdash 42^D : Int^D \hookrightarrow 42 : Int$
2. $\vdash 42^S : Int^S \hookrightarrow \bullet : \hat{42}$
3. $\vdash (2^S +^S 1^S) +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$
4. $\vdash \mathbf{lift}\ (2^S +^S 1^S) +^D 1^D : Int^D \hookrightarrow 3 + 1 : Int$

Observar que las expresiones dinámicas se conservan en los términos residuales, mientras que las estáticas fueron reemplazadas por $\bullet$ y su información trasladada al tipo. También se muestra el uso de **lift**, que en este caso recuperó del tipo la información estática de su expresión, que especializa a $\bullet : \hat{3}$, para construir una expresión dinámica con ese valor.

Ejemplo 1.2. Las asunciones proveen información sobre variables libres, y permiten especializar funciones:

1. $x : Int^S \hookrightarrow \bullet : \hat{3} \vdash x +^S 1^S : Int^S \hookrightarrow \bullet : \hat{4}$
2. $\vdash (\lambda^D x.x +^S 1^S)\ @^D\ (2^S +^S 1^S) : Int^S \hookrightarrow (\lambda x'.\bullet)@\bullet : \hat{4}$
3. $\vdash (\lambda^D f.f\ @^D\ 42^S)\ @^D\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) : Int^D$
   $\hookrightarrow (\lambda f.f@\bullet)@(\lambda x.42 + 1) : Int$

En el último caso, además, se movió información estática del cuerpo de una función de alto orden hasta el lugar donde **lift** la reinserta en el término residual.

Ejemplo 1.3. La expresión

$$\begin{array}{l}\mathbf{let}^D\ f = \lambda^D x.\mathbf{lift}\ x +^D 1^D \\ \mathbf{in}\ (f\ @^D\ 42^S, f\ @^D\ 17^S)^D : (Int^D, Int^D)^D .\end{array}$$

no puede ser especializada: la función f toma un entero estático, y este valor será expresado en el término especializado como un tipo *singleton* en su tipo residual. El valor del argumento debe fijarse entonces al especializar el término, y no puede instanciarse a dos valores diferentes luego de esta decisión. Para especializar este tipo de expresiones, se anota la función como polivariante con la construcción **poly**, que resulta en diferentes especializaciones de la misma expresión según los argumentos que se le apliquen:

1. $\vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)$
   $\mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D : (Int^D, Int^D)^D \hookrightarrow$
   $\mathbf{let}\ f' = (\lambda x'.42 + 1, \lambda x'.17 + 1)$
   $\mathbf{in}\ (\mathbf{fst}\ f'@\bullet, \mathbf{snd}\ f'@\bullet) : (Int, Int)$
2. $\vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D)$
   $\mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 42^S, \mathbf{spec}\ f\ @^D\ 17^S)^D : (Int^D, Int^D)^D \hookrightarrow$
   $\mathbf{let}\ f' = (\lambda x'.17 + 1, \lambda x'.55 + 1, \lambda x'.42 + 1)$
   $\mathbf{in}\ (\pi_{3,3}\ f'@\bullet, \pi_{1,3}\ f'@\bullet) : (Int, Int)$

Observar que no se restringe el tamaño, ni el orden, ni la forma de especializar cada uno de los elementos en la tupla residual, en tanto que haya al menos una componente de cada uno de los tipos que cada **spec** requiera. De esta forma, se pueden obtener varias –infinitas– especializaciones distintas (sin relación unas con otras, es decir que ninguna es 'mejor' que otra) a partir de un mismo programa fuente.

En el siguiente ejemplo se ilustra el problema que se atacará con el enfoque de Especialización de Tipos Principal de [Martínez López and Hughes, 2002].

Ejemplo 1.4. Obsérvese que en todos los casos falta información estática:

1. $\lambda^D x.x +^S 1^S : Int^S \to^D Int^S$
2. $\mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) : \mathbf{poly}\ (Int^S \to^D Int^D)$
3. $\lambda^D f.\mathbf{spec}\ f\ @^D\ 13^S : \mathbf{poly}\ (Int^S \to^D Int^D) \to^D Int^D$

Estas expresiones tienen varias especializaciones cada una, sin relación unas con otras. La primera puede especializar a $\lambda x'.\bullet : \hat{n} \to \hat{n}'$ para cualquier n y n' tales que $n' = n + 1$. Si esta expresión apareciera en un módulo, aplicada a otra, entonces la especialización debería esperar hasta obtener el valor de n antes de decidir el tipo residual. Un problema similar sucede en los otros casos, con respecto a la polivarianza. La generación de la tupla estática o la proyección adecuada debe diferirse hasta que se obtenga toda la información acerca de los usos de la expresión.

El problema se denomina falta de *principalidad*, ya que no es expresable en este sistema una especialización 'principal' a partir de la cual cualquier otra pueda ser obtenida mediante alguna operación (cierta forma de instanciación)

La falta de principalidad fuerza a las implementaciones de la especialización a realizarse en forma monolítica, sin capacidad de especializar módulos en forma separada.

1.2 Tipos Calificados

La teoría de Tipos Calificados (*Qualified Types*) es un framework creado por Mark Jones [Jones, 1994a] para la creación de sistemas de tipos en un nivel intermedio entre las disciplinas de monomorfismo y polimorfismo.

El trabajo de [Martínez López and Hughes, 2002] sobre el que nos basamos utiliza profundamente este sistema reformulando la especialización de [Hughes, 1996] mediante tipos calificados; y en particular el presente trabajo trata de la manipulación de tales tipos para completar la especialización, por lo que nos extenderemos en su explicación.

Los tipos calificados pueden ser vistos como una forma restringida de polimorfismo o como una extensión del monomorfismo (comúnmente llamada sobrecarga -*overloading*-, en la que una función puede tener diferentes interpretaciones de acuerdo a sus argumentos). Los tipos contienen variables de tipo, y una serie de *predicados* que restringen los posibles valores que tales variables pueden tomar.

La teoría explica cómo enriquecer los tipos con estos predicados y realizar la tarea de inferencia, y las propiedades mínimas que los predicados deben satisfacer para obtener resultados similares a un sistema Hindley-Milner [Milner, 1978] (en particular, se prueba que cualquier programa tipado tiene un *tipo principal* que puede ser calculado con una versión extendida del algoritmo de Milner).

1.2.1 Predicados e implicación

Polimorfismo es la capacidad de tratar algunos términos como si tuvieran diferentes tipos. La forma clásica de expresarlo es mediante *esquemas de tipos* [Damas and Milner, 1982], usando cuantificación universal sobre las variables que representan aquellas partes donde el tipo puede variar. De esta forma, si $f(t)$ es un tipo para cada posible valor de la variable t, asignar el esquema de tipos $\forall t.f(t)$ a un término significa que éste puede recibir cualquier tipo del conjunto

$$\{f(\tau) : \tau \text{ es un tipo}\}$$

En algunos casos tal cuantificación captura más tipos de los que se desea, ya que t puede tomar cualquier valor en el conjunto de los tipos. En estos casos se usa una forma restringida de cuantificación: si $\delta(t)$ es un predicado sobre tipos, el esquema de tipos calificado $\forall t.\delta(t) \Rightarrow f(t)$ representa el conjunto de tipos

$$\{f(\tau) : \tau \text{ es un tipo que satisface } \delta(t)\}$$

La característica esencial de esta teoría es el uso de *predicados* δ para describir conjuntos de tipos (o, en general, relaciones entre tipos) El conjunto exacto de predicados posibles varía según el lenguaje, y los que usaremos en nuestro caso se presentarán en la sección 1.3.1.

En este punto cabe una aclaración respecto a la notación utilizada. Jones 'sobrecarga' el significado de algunos símbolos de forma de simplificar la notación [Jones, 1994a], confundiendo (en el buen sentido) cuando es conveniente elementos con listas de elementos y pares de ellos, y operaciones aplicadas a cualquiera de los mismos. De esta forma, se toman las siguiente convenciones:

- Sean L y L' cualquier tipo de listas o conjuntos (finitos), y l un elemento de ellos. Escribimos L, L' como el resultado de concatenar ambas listas (resp. unir ambos conjuntos). Con l, L representamos el resultado de la inserción de l en la lista (resp. conjunto) L. Y denotando con $\emptyset$ la lista (resp. conjunto) vacía, asumimos $\emptyset, L = L = L, \emptyset$. (Notar que de esta forma l puede denotar tanto al elemento en sí como a la lista (conjunto) $l, \emptyset$, lo que usualmente es determinado por el contexto.)

- También es conveniente, cuando se trabaja con lenguajes de diferentes términos construidos en base a agregar calificadores sobre otros, el confundirlos a ambos. Por ejemplo, tomemos el caso de los tipos calificados, que se definen como

$$\rho ::= \delta \Rightarrow \rho \mid \tau$$

  Siendo Δ la lista de predicados $\delta_1, ..., \delta_n$, se abrevia $\delta_1 \Rightarrow ... \Rightarrow \delta_n \Rightarrow \tau$ como $\Delta \Rightarrow \tau$. Más aún, en el caso de no haber calificadores, se identifica $\emptyset \Rightarrow \tau$ con τ.
  Siendo una lista de predicados $\Delta = \delta_1, \cdots, \delta_m$, una lista de variables $\alpha = \alpha_1, \cdots, \alpha_m$ una lista de variables de evidencia $h = h_1, \cdots, h_m$ y uns lista de expresiones de evidencia $v = v_1, \cdots, v_m$, utilizaremos las abreviaturas (la definición formal de cada una de estas expresiones se hará mas adelante en esta sección):

Objeto	Expresión	Abreviación
Tipo Calificado	$\delta_1 \Rightarrow \cdots \delta_m \Rightarrow \tau'$	$\Delta \Rightarrow \tau'$
Esquema de tipos	$\forall\alpha_1.\cdots\forall\alpha_m.\rho$	$\forall\alpha.\rho$
Abstracción de evidencia	$\Lambda h_1.\cdots\Lambda h_m.e'$	$\Lambda h.e'$
Aplicación de evidencia	$((e'(\!(v_1)\!))\cdots)(\!(v_m)\!)$	$e'(\!(v)\!)$

- Siendo $h = h_1, \ldots, h_n$ y $\Delta = \delta_1, \ldots, \delta_n$, la lista (o conjunto) de pares $h_1 : \delta_1, \ldots, h_n : \delta_n$ se abrevia $h : \Delta$ (o inclusive Δ, cuando la otra componente no es relevante). La concatenación (unión) de listas (conjuntos) de pares puede ser denotada tanto $h : \Delta, h' : \Delta'$ como $h, h' : \Delta, \Delta'$.

Fuera de la razonable confusión inicial de un lector casual al encontrar esta notación, estas convenciones pueden dominarse sin mayor trabajo y prueban ser realmente útiles en la práctica.

Las propiedades mínimas que se requieren de los predicados son capturadas en la relación de implicación (*entailment*) $\Vdash$, entre conjuntos finitos de predicados. Con $\Delta_1 \Vdash \Delta_2$ se denota que los predicados de el predicado Δ_1 permiten inferir, o implican, los predicados de Δ_2.

Las propiedades básicas que se requieren de la relación $\Vdash$ (que puede especificarse con un sistema de reglas como el de la figura 1.1, sin la evidencia) son:

- *Monotonicidad:* $\Delta \Vdash \Delta'$ siempre que $\Delta \supseteq \Delta'$
- *Transitividad:* si $\Delta \Vdash \Delta'$ y $\Delta' \Vdash \Delta''$, entonces $\Delta \Vdash \Delta''$
- *Clausura:* si $\Delta \Vdash \Delta'$, entonces $S\Delta \Vdash S\Delta'$

Siguiendo la convención sintáctica descripta arriba, con $\Delta \Vdash \delta$ se abrevia $\Delta \Vdash \{\delta\}$. Notar además puede si $\delta \in \Delta$, entonces la monotonicidad de $\Vdash$ asegura que $\Delta \Vdash \delta$.

1.2.2 Inferencia de tipos calificados

En esta teoría, el lenguaje de tipos y esquemas de tipos se estratifica en forma similar al sistema de Hindley-Milner, siendo la restricción más importante que los tipos calificados o polimórficos no pueden ser argumento de funciones. Los tipos se denotan con τ, y son definidos por una gramática con (al menos) las producciones $\tau ::= t \mid \tau \rightarrow \tau$ (siendo t un elemento de un conjunto infinito de variables de tipo), más construcciones para constantes y otros elementos de cada lenguaje particular. Sobre ellos se definen los tipos calificados $\rho ::= \Delta \Rightarrow \tau$, y finalmente los esquemas de tipo de la forma $\sigma ::= \forall\{\alpha_i\}.\rho$. De esta forma, y de acuerdo a las convenciones de la sección anterior, cualquier esquema de tipos se puede escribir de la forma $\forall\alpha_i.\Delta \Rightarrow \tau$, representando el conjunto de tipos calificados

$$\{\Delta[\tau_i/\alpha_i] \Rightarrow \tau[\tau_i/\alpha_i] : \tau_i \text{ es un tipo}\}$$

El lenguaje de términos es llamado OML (*Overloaded ML*) y está basado en λ-cálculo; denotaremos a sus términos con e. La inferencia de tipos usa juicios extendidos con un contexto de predicados: $\Delta \mid \Gamma \vdash e : \sigma$, representando el hecho de que cuando los predicados de Δ sean satisfechos, y los tipos de las variables libres los determinados por Γ, entonces e tiene tipo σ. El sistema de reglas de inferencia es esencialmente el mismo que para un sistema Hindley-Milner, con la adición de las reglas (GEN) e (INST) para generalizar e instanciar variables en esquemas, y reglas (QIN) y (QOUT) para el manejo de predicados:

$$\text{(Fst)} \quad h : \Delta, h' : \Delta' \Vdash h : \Delta$$

$$\text{(Snd)} \quad h : \Delta, h' : \Delta' \Vdash h' : \Delta'$$

$$\text{(Univ)} \quad \frac{h : \Delta \Vdash v' : \Delta' \qquad h : \Delta \Vdash v'' : \Delta''}{h : \Delta \Vdash v' : \Delta', v'' : \Delta''}$$

$$\text{(Trans)} \quad \frac{h : \Delta \Vdash v' : \Delta' \qquad h' : \Delta' \Vdash v'' : \Delta''}{h : \Delta \Vdash v''[v'/h'] : \Delta''}$$

$$\text{(Close)} \quad \frac{h : \Delta \Vdash v' : \Delta'}{h : S\,\Delta \Vdash v' : S\,\Delta'}$$

Figura 1.1: Leyes estructurales de implicación

$$\text{(QIN)} \quad \frac{\Delta, \delta \mid \Gamma \vdash e : \rho}{\Delta \mid \Gamma \vdash e : \delta \Rightarrow \rho}$$

$$\text{(QOUT)} \quad \frac{\Delta \mid \Gamma \vdash e : \delta \Rightarrow \rho \qquad \Delta \Vdash \delta}{\Delta \mid \Gamma \vdash e : \rho}$$

Para determinar las formas en que un término e puede ser usado bajo un Γ dado, se trabaja con conjuntos de la forma

$$\{(\Delta \mid \sigma) : \Delta \mid \Gamma \vdash e : \sigma\}$$

donde $(\Delta \mid \sigma)$ denota un par con una lista de predicados y un esquemas de tipos como el anterior. A estos pares se los llama *esquema de tipos restringido.*

La herramienta principal para tratar tales conjuntos es un preorden $\geq$ ('*más general que*'), definido entre esquemas de tipos restringidos. Para definirlo formalmente se utiliza la noción de *instancia genérica*: un tipo calificado $\Delta_\tau \Rightarrow \tau$ es una instancia genérica de $(\Delta \mid \forall\alpha_i.\Delta' \Rightarrow \tau')$ si existen tipos τ_i tales que

$$\Delta_\tau \Vdash \Delta, \Delta'[\tau_i/\alpha_i] \quad \text{y} \quad \tau = \tau'[\tau_i/\alpha_i]$$

En particular, un tipo calificado $\Delta \Rightarrow \tau$ es instancia de otro $\Delta' \Rightarrow \tau'$ si y sólo si $\Delta \Vdash \Delta'$ y $\tau = \tau'$.

Utilizando la noción de instancia genérica se define el orden de *'más general que'*: $(\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ si cada instancia genérica de $(\Delta' \mid \sigma')$ es también una instancia genérica de $(\Delta \mid \sigma)$. Este concepto será esencial para la especialización principal de tipos sobre la que trabajaremos.

1.2.3 Evidencia y Coherencia

La forma de Jones de dar semántica a los términos del sistema es mediante la noción de *evidencia*, y dando una traducción del lenguaje OML original a uno

que manipula evidencia explícitamente, llamado OP (*Overloaded Polymorphic λ-calculus*). La idea esencial es que un objeto de tipo $\Delta \Rightarrow \tau$ sólo puede usarse si se provee evidencia adecuada de que los predicados en Δ realmente se cumplen. El tratamiento de la evidencia casi no tiene efectos en el algoritmo de tipado, pero es esencial para proveer *coherencia*: que la semántica de un término no dependa de la forma en que es tipado. Las propiedades de la relación de implicación $\Vdash$ se extienden para manejar asignaciones de predicados y expresiones de evidencia (las reglas de la figura 1.1 ya expresan esta extensión). Se denotan con h las variable de evidencia, cuyo rango es sobre un lenguaje de *expresiones de evidencia*, denotadas v; y las asignaciones de evidencia se denotan (de acuerdo a las convenciones sintáticas) como $h : \Delta$, representando a $h_1 : \delta_1, \ldots, h_n : \delta_n$ (en forma similar, $v : \Delta$).

El lenguaje de términos de OP es extendido con variables de evidencia h, y con las construcciones $\Lambda h.e'$ y $e'((v))$ para abstraer y proveer evidencia, respectivamente. Las reglas de tipado de OP permiten introducir y eliminar estas construcciones. También permiten el uso de polimorfismo irrestricto, y aunque este hecho hace indecidible la inferencia de tipos, el lenguaje OP sólo se utiliza como imagen de la traducción para términos OML, por lo que no es un problema. Las reglas de traducción de OML a OP reflejan la extensión homomórfica de las construcciones sintácticas, pudiendo agregar construcciones de evidencia (abstracción y apicación) en presencia de tipos calificados:

$$\text{(QIN)} \quad \frac{\Delta, h : \delta \mid \Gamma \vdash e \hookrightarrow e' : \rho}{\Delta \mid \Gamma \vdash e \hookrightarrow \Lambda h.e' : \delta \Rightarrow \rho}$$

$$\text{(QOUT)} \quad \frac{\Delta \mid \Gamma \vdash e \hookrightarrow e' : \delta \Rightarrow \rho \qquad \Delta \Vdash v : \delta}{\Delta \mid \Gamma \vdash e \hookrightarrow e'((v)) : \rho}$$

La interacción presente en estas reglas entre implicación de predicados, variables de evidencia, términos de evidencia, su abstracción y aplicación jugará un rol esencial en nuestro trabajo.

La traducción de algunos términos de OML puede dar lugar a diferentes términos OP, no equivalentes, lo que muestra que la semántica de los términos OML depende de la forma en que son tipados. Para caracterizar aquellos términos cuyo significado es único, se define una equivalencia entre términos de OP, utilizando reducción, y la noción de conversión. Una *conversión* de un esquema σ a otro σ' es un término que permite transformar cualquier término OP de tipo σ a uno de tipo σ' manipulando su evidencia. Esta noción extiende la definición de $\geq$ con el tratamiento de evidencia.

Siendo $\sigma = \forall \alpha_i.\Delta_\tau \Rightarrow \tau$ y $\sigma' = \forall \beta_i.\Delta'_\tau \Rightarrow \tau'$ dos esquemas de tipos, y suponiendo que ninguna de las β_i aparece libre en σ, Δ, o Δ', una *conversión* de $(\Delta \mid \sigma)$ a $(\Delta' \mid \sigma')$ es un término cerrado C de OP con tipo $(\Delta \mid \sigma) \rightarrow (\Delta' \mid \sigma')$ -denotado $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$- si existen tipos τ_i, variables de evidencia h' y h'_τ, y expresiones de evidencia v y v' tales que:

- $\tau' = \tau[\alpha_i/\tau_i]$
- $h' : \Delta', h'_\tau : \Delta'_\tau \Vdash v : \Delta, v' : \Delta_\tau[\alpha_i/\tau_i]$, y
- $C = (\lambda x.\Lambda h', h'_\tau.x((v))((v')))$

Las conversiones se usan en esta teoría para relacionar diferentes traducciones del mismo término. En especialización principal tendrán un uso similar, como veremos en la siguiente sección.

Para finalizar, mencionaremos que Jones define las nociones de *simplificación* y *mejora* sobre conjuntos de predicados, como forma de producir esquemas de tipos legibles y de tamaño razonable. Utilizaremos profundamente este concepto en el capítulo 2, que servirá de base para el trabajo del capítulo 3.

1.3 Especialización Principal

Como se ejemplificó en 1.1.3, el sistema de [Hughes, 1996] no especializa correctamente algunos términos a menos que toda la información del contexto sea conocida. El problema de tener diferentes –incomparables– especializaciones de un mismo término (de las cuales ninguna es mejor que otra, a menos que se la use en algún contexto) tiene una analogía en el λ-cálculo *simply-typed*: se pueden asignar infinitas expresiones de tipo a $\lambda x.x$, pero ninguna que abarque a todas ellas. En λ-cálculo, el problema anterior se resuelve con tipos polimórficos (con variables de tipo cuantificadas universalmente y una noción de instanciación).

Para atacar el problema en especialización de tipos, el sistema de [Martínez López and Hughes, 2002] es capaz de encontrar *especializaciones principales* para cada término, tales que cualquier otra especialización es una 'instancia' de la inferida por el sistema.

De esta forma la especialización de un término se puede hacer en forma aislada, sin información del contexto. El primer paso es agregar variables de tipo y cuantificación para diferir la especializaciones: pero esto no sería suficiente ya que relaciones como la que hay entre n y n' en el ejemplo 1.4-1 no son expresables. Como solución se utilizan tipos calificados (sección 1.2) para restingir el alcance de la cuantificación.

Esta sección resume el trabajo de especialización principal de tipos, el cual completaremos en el resto de nuestro trabajo.

1.3.1 Lenguaje Residual

Como se dijo, la idea fundamental es la existencia de predicados en el tipo de una expresión residual, para expresar las restricciones impuestas por el contexto. El lenguaje es también extendido de forma de manipular evidencia: las nuevas construcciones son aquellas 'estructurales' (tomadas de la teoría de tipos calificados) y construcciones para expresar características particulares de la especialización.

$$(\beta_v) \quad (\Lambda h.e'_1)((v)) \ \triangleright \ e'_1[v/h]$$

$$(\eta_v) \quad \Lambda h.e'_1((h)) \ \triangleright \ e'_1 \quad \left(h \notin EV(e'_1)\right)$$

$$(\mathbf{let}_v) \quad \mathbf{let}_v \ x = e'_1 \ \mathbf{in} \ e'_2 \ \triangleright \ e'_2[e'_1/x]$$

$$(\circ_v) \quad (v_1 \circ v_2)[e'] \ \triangleright \ v_1[v_2[e']]$$

Figura 1.2: Reglas de reducción de términos residuales

Siguiendo a [Jones, 1994a], el lenguaje de tipos residuales se expresa mediante tipos (τ') conteniendo variables de tipo (t), tipos calificados (ρ), esquemas cuantificados (σ) y predicados (δ). La innovación más importante es el uso de una nueva construcción, **poly** σ, y el uso de variables de esquema s:

$$\begin{array}{rcl}
\tau' & ::= & t \mid \mathit{Int} \mid \hat{n} \mid \tau' \to \tau' \mid (\tau', \ldots, \tau') \mid \mathbf{poly}\ \sigma \\
\rho & ::= & \delta \Rightarrow \rho \mid \tau' \\
\sigma & ::= & s \mid \forall s.\sigma \mid \forall t.\sigma \mid \rho \\
\delta & ::= & \mathrm{IsInt}\ \tau' \mid \tau' := \tau' + \tau' \mid \mathrm{IsMG}\ \sigma\ \sigma
\end{array}$$

El lenguaje de términos se extiende para manipular evidencia (v), incluyendo variables de evidencia (h), abstracciones ($\Lambda h.e'$) y aplicaciones de evidencia ($e'((v))$). La evidencia es esencial para abstraer las diferencias entre distintos términos residuales de una misma expresión fuente. Se utilizan dos tipos de evidencia: números (como evidencia a predicados IsInt y $\_ := \_ + \_$) y conversiones (como evidencia para predicados IsMG). Las conversiones, C, se definen como elementos separados del lenguaje, y son contextos en vez de (familias de) términos. El lenguaje de expresiones residuales e' queda entonces definido por la siguiente gramática:

$$\begin{array}{rcl}
e' & ::= & x' \mid \lambda x'.e' \mid e'@e' \mid n \mid e' + e' \mid \bullet \\
 & \mid & \pi_{n,n}\, e' \mid (e'_1, \ldots, e'_n) \mid \mathbf{let}\ x' = e'\ \mathbf{in}\ e' \mid h \\
 & \mid & v[e'] \mid \Lambda h.e' \mid e'((v)) \mid \mathbf{let}_v\ x = e'\ \mathbf{in}\ e' \\
v & ::= & h \mid n \mid C \mid v \circ v \\
C & ::= & [] \mid \Lambda h.C \mid C((v)) \mid \mathbf{let}_v\ x = C\ \mathbf{in}\ e'
\end{array}$$

Se trabaja siempre bajo la equivalencia $=$ sobre términos residuales, definida como la mínima relación entre expresiones conteniendo α-conversión para abstracciones (de términos y evidencia) y las reglas de la figura 1.2. La equivalencia se extiende para conversiones, definiendo $C = C'$ si para toda expresión e', $C[e'] = C'[e']$.

$$\text{(IsInt)} \quad \Delta \Vdash n : \text{IsInt } \hat{n}$$

$$\text{(IsOp)} \quad h : \Delta \Vdash n : \hat{n} := \hat{n}_1 + \hat{n}_2 \quad \left(\textit{si } n{=}n_1{+}n_2\right)$$

$$\text{(IsOpIsInt)} \quad \Delta, h : \tau' := \tau'_1 + \tau'_2, \Delta' \Vdash h : \text{IsInt } \tau'$$

$$\text{(IsMG)} \quad \frac{C : (\Delta \mid \sigma') \geq (\Delta \mid \sigma)}{\Delta \Vdash C : \text{IsMG } \sigma' \; \sigma}$$

$$\text{(Comp)} \quad \frac{\Delta \Vdash v : \text{IsMG } \sigma_1 \; \sigma_2 \qquad \Delta \Vdash v' : \text{IsMG } \sigma_2 \; \sigma_3}{\Delta \Vdash v' \circ v : \text{IsMG } \sigma_1 \; \sigma_3}$$

Figura 1.3: Construcción de evidencia por la relación de implicación

El significado de la relación de implicación $\Vdash$, cuyas propiedades estructurales se expresan en la figura 1.1, se completa con reglas de construcción de evidencia de la figura 1.3. El predicado IsInt correspondiente a un tipo unitario, se prueba proveyendo el número que lo representa. Un predicado $\_ := \_ + \_$ es demostrable si los tres argumentos son los correspondientes números a los tres tipos unitarios. El predicado IsMG internaliza el orden $\geq$ (incorporándolo al lenguaje con variables de esquema), y su evidencia es la conversión que convierte términos del tipo más general al más específico. La composición de evidencia utilizada en esta regla es la definida en la figura 1.2.

1.3.2 Tipos residuales

Como se mencionó en la sección 1.2.3 para la teoría de tipos calificados, la relación entre distintos tipos y esquemas de tipos se realiza mediante la relación 'más general que'. Las conversiones ahora están definidas como contextos en lugar de términos del lenguaje residual, dado que serán usadas como evidencia (conformarán la evidencia que probará cierto tipo de predicados cuya semántica es muy cercana a la relacion $\geq$) y se aplicarán a términos cuando se estén probando los predicados. Estos cambios se reflejan en la nueva definición de $\geq$.

Definición 1.5 (Conversión, $\geq$). Sean $\sigma = \forall \alpha_i.\Delta_\tau \Rightarrow \tau$ y $\sigma' = \forall \beta_i.\Delta'_\tau \Rightarrow \tau'$ dos esquemas de tipo, y supongamos que ninguna de las β_i aparece libre en σ, $h : \Delta$, ó $h' : \Delta'$. Se llama a C una *conversión* de $(\Delta \mid \sigma)$ a $(\Delta' \mid \sigma')$, denotado $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$, si y sólo si existen tipos τ_i, variables de evidencia h_τ y h'_τ, y expresiones de evidencia v y v' tales que:

- $\tau' = \tau[\alpha_i/\tau_i]$
- $h' : \Delta', h'_\tau : \Delta'_\tau \Vdash v : \Delta, v' : \Delta_\tau[\alpha_i/\tau_i]$, y

- $C = (\mathbf{let}_v\ x = \Lambda h.[]\ \mathbf{in}\ \Lambda h'_\tau.x((v))((v')))$

La propiedad más importante de las conversiones es que, aplicadas sobre un objeto e' de tipo σ bajo un conteto Δ, lo transforman en un elemento de tipo σ' bajo una asignación de predicados Δ', cambiando sólo la evidencia más externa (expresada en el término por sus variables de evidencia).

Proposición 1.6. Las propiedades de reflexividad y transitividad se cumplen para $\geq$, establecidas por las siguientes conversiones:

1. $[] : (\Delta \mid \sigma) \geq (\Delta \mid \sigma)$
2. Si $C : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ y $C' : (\Delta' \mid \sigma') \geq (\Delta'' \mid \sigma'')$ entonces $C' \circ C : (\Delta \mid \sigma) \geq (\Delta'' \mid \sigma'')$

Ejemplo 1.7. Las conversiones se utilizan para ajustar la evidencia dentro de los términos de acuerdo a lo que requieran sus esquemas de tipos. Para todo Δ se cumple que:

1. $[]((42)) : (\Delta \mid \forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int) \geq (\Delta \mid \hat{42} \to Int)$
2. $C : (\Delta \mid \forall t_1, t_2.\mathrm{IsInt}\ t_1, \mathrm{IsInt}\ t_2 \Rightarrow t_1 \to t_2) \geq (\Delta \mid \forall t.\mathrm{IsInt}\ t \Rightarrow t \to t)$ donde $C = \Lambda h.[]((h))((h))$
3. $\Lambda h.[] : (\Delta \mid \hat{42} \to Int) \geq (\Delta \mid \forall t.\mathrm{IsInt}\ t \Rightarrow \hat{42} \to Int)$

En [Martínez López and Hughes, 2002] también se presenta un sistema de inferencia de tipos para el lenguaje residual (probando luego que la especialización es consistente respecto a ambos sistemas de tipos, fuente y residual). Este sistema no está hecho para *obtener* los tipos de expresiones residuales, sino sólo para *verificarlos*: las expresiones residuales no serán dadas por el programador, sino obtenidas como producto de la especialización, y como tales serán obtenidas junto con su tipo. (Por esta razón se puede proveer el polimorfismo de alto orden, en la construcción **poly**.)

1.3.3 Sistema de Especialización Principal

La especialización se especifica mediante dos sistemas de reglas.

El primero relaciona tipos fuente con tipos residuales, expresando qué tipos residuales pueden ser obtenidos por especialización de uno fuente; sus reglas se presentan la figura 1.4. La utilización de la relación entre tipos fuente y residuales se utiliza al especializar λ-abstracciones y especializar expresiones polivariantes, en donde sin la restricción impuesta (ver figura 1.5) podrían obtenerse más especializaciones de las deseadas, como por ejemplo. $\vdash \lambda^D x.x : Int^S \to^D Int^S \hookrightarrow \lambda x'.x' : Bool \to Bool$.

El segundo sistema de reglas especifica la especialización en sí misma, presentado en la figura 1.5. Los juicios tiene la forma $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$,

$$\text{(SR-DINT)} \quad \Delta \vdash_{\text{SR}} Int^D \hookrightarrow Int$$

$$\text{(SR-SINT)} \quad \frac{\Delta \Vdash \text{IsInt } \tau'}{\Delta \vdash_{\text{SR}} Int^S \hookrightarrow \tau'}$$

$$\text{(SR-DFUN)} \quad \frac{\Delta \vdash_{\text{SR}} \tau_1 \hookrightarrow \tau_1' \qquad \Delta \vdash_{\text{SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta \vdash_{\text{SR}} \tau_2 \to^D \tau_1 \hookrightarrow \tau_2' \to \tau_1'}$$

$$\text{(SR-TUPLE)} \quad \frac{(\Delta \vdash_{\text{SR}} \tau_i \hookrightarrow \tau_i')_{i=1,..,n}}{\Delta \vdash_{\text{SR}} (\tau_1, \ldots, \tau_n)^D \hookrightarrow (\tau_1', \ldots, \tau_n')}$$

$$\text{(SR-POLY)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma' \qquad \Delta \Vdash \text{IsMG } \sigma' \, \sigma}{\Delta \vdash_{\text{SR}} \textbf{poly } \tau \hookrightarrow \textbf{poly } \sigma}$$

$$\text{(SR-QIN)} \quad \frac{\Delta, \delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho}$$

$$\text{(SR-QOUT)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \delta \Rightarrow \rho \qquad \Delta \Vdash \delta}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \rho}$$

$$\text{(SR-GEN)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \sigma}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \forall \alpha.\sigma} \quad \left(\alpha \notin FV(\Delta)\right)$$

$$\text{(SR-INST)} \quad \frac{\Delta \vdash_{\text{SR}} \tau \hookrightarrow \forall \alpha.\sigma}{\Delta \vdash_{\text{SR}} \tau \hookrightarrow S\sigma} \quad \left(\text{dom}(S)=\alpha\right)$$

Figura 1.4: Relación entre tipos fuente y residuales

extendiendo las nociones explicadas en la sección 1.1.3 con el uso de tipos calificados (y asignaciones de predicados como información contextual).

Las reglas (QIN) y (QOUT) incorporan la noción de evidencia introducida en la sección 1.2 y permiten mover información desde el contexto (en forma de predicados) hacia los términos (introduciendo los predicados en el tipo y abstrayendo la evidencia) y a la inversa. (Notar que estas reglas son inherentemente no 'dirigidas a sintaxis' -syntax directed-, y que cada una es la inversa de la otra, pudiendo ser aplicadas para anular el efecto de la contraria.)

Para finalizar el resumen de este sistema de especialización, utilizaremos las reglas de la figura 1.5 con el fin de obtener varias especializaciones principales. Revisitaremos los ejemplos de la sección 1.1.3 ilustrando algunas especializaciones en el sistema con principalidad.

$$\text{(VAR)}\quad \Delta \mid \Gamma, x : \tau \hookrightarrow x' : \tau', \Gamma' \vdash_{\mathrm{P}} x : \tau \hookrightarrow x' : \tau'$$

$$\text{(DINT)}\quad \Delta \mid \Gamma \vdash_{\mathrm{P}} n^{D} : \mathit{Int}^{D} \hookrightarrow n : \mathit{Int}$$

$$\text{(D+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \mathit{Int}^{D} \hookrightarrow e_i' : \mathit{Int})_{i=1,2}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 +^{D} e_2 : \mathit{Int}^{D} \hookrightarrow e_1' + e_2' : \mathit{Int}}$$

$$\text{(LIFT)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \mathit{Int}^{S} \hookrightarrow e' : \tau' \qquad \Delta \Vdash v : \mathrm{IsInt}\ \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{lift}\ e : \mathit{Int}^{D} \hookrightarrow v : \mathit{Int}}$$

$$\text{(SINT)}\quad \Delta \mid \Gamma \vdash_{\mathrm{P}} n^{S} : \mathit{Int}^{S} \hookrightarrow \bullet : \hat{n}$$

$$\text{(S+)}\quad \frac{(\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \mathit{Int}^{S} \hookrightarrow e_i' : \tau_i')_{i=1,2} \qquad \Delta \Vdash v : \tau' := \tau_1' + \tau_2'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 +^{S} e_2 : \mathit{Int}^{S} \hookrightarrow \bullet : \tau'}$$

$$\text{(DTUPLE)}\quad \frac{(\Delta \mid \Gamma \vdash_{\mathrm{P}} e_i : \tau_i \hookrightarrow e_i' : \tau_i')_{i=1,..,n}}{\Delta \mid \Gamma \vdash_{\mathrm{P}} (e_1, ..., e_n)^{D} : (\tau_1, ..., \tau_n)^{D} \hookrightarrow (e_1', ..., e_n') : (\tau_1', ..., \tau_n')}$$

$$\text{(DPRJ)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : (\tau_1, \ldots, \tau_n)^{D} \hookrightarrow e' : (\tau_1', \ldots, \tau_n')}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \pi^{D}_{i,n}\, e : \tau_i \hookrightarrow \pi_{i,n}\, e' : \tau_i'}$$

$$\text{(DLAM)}\quad \frac{\Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau_2' \vdash_{\mathrm{P}} e : \tau_1 \hookrightarrow e' : \tau_1' \qquad \Delta \vdash_{\mathrm{SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \lambda^{D} x.e : \tau_2 \to^{D} \tau_1 \hookrightarrow \lambda x'.e' : \tau_2' \to \tau_1'} \quad (x' \text{ nueva})$$

Figura 1.5: Reglas de especialización (primera parte)

Ejemplo 1.8. El primer término del ejemplo 1.4-1 no podía especializarse porque faltaba información en el contexto para determinar el valor de la variable estática x. En este sistema, el término se especializa de la siguiente forma:

$$\begin{aligned} &\vdash_{\mathrm{P}} \lambda^{D} x.x +^{S} 1^{S} : \mathit{Int}^{S} \to^{D} \mathit{Int}^{S} \\ &\hookrightarrow \Lambda h, h'.\lambda x.\bullet : \forall t, t'.h : \mathrm{IsInt}\ t, h' : t' := t + \hat{1} \Rightarrow t \to t' \end{aligned}$$

Observar cómo se introdujo un predicado para obligar expresar que la variable x debe tener un valor entero, y otro para forzar el valor del resultado de la suma, expresando con estos predicados la información que aún falta obtener del contexto en que se pueda usar este término.

Ejemplo 1.9. La expresión del ejemplo 1.4-2 es un término polivariante que no puede ser especializado en el sistema original de especialización. La especial-

$$\text{(DAPP)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 : \tau_2 \to^D \tau_1 \hookrightarrow e_1' : \tau_2' \to \tau_1' \quad \Delta \mid \Gamma \vdash_{\mathrm{P}} e_2 : \tau_2 \hookrightarrow e_2' : \tau_2'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_1 \mathbin{@^D} e_2 : \tau_1 \hookrightarrow e_1' @ e_2' : \tau_1'}$$

$$\text{(DLET)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e_2 : \tau_2 \hookrightarrow e_2' : \tau_2' \qquad \Delta \mid \Gamma, x : \tau_2 \hookrightarrow x' : \tau_2' \vdash_{\mathrm{P}} e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{let}^D\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \hookrightarrow \mathbf{let}\ x' = e_2'\ \mathbf{in}\ e_1' : \tau_1'} \quad (x' \text{ nueva})$$

$$\text{(POLY)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma' \qquad \Delta \Vdash v : \mathrm{IsMG}\ \sigma'\ \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow v[e'] : \mathbf{poly}\ \sigma}$$

$$\text{(SPEC)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \mathbf{poly}\,\tau \hookrightarrow e' : \mathbf{poly}\,\sigma \quad \Delta \Vdash v : \mathrm{IsMG}\ \sigma\ \tau' \quad \Delta \vdash_{\mathrm{SR}} \tau \hookrightarrow \tau'}{\Delta \mid \Gamma \vdash_{\mathrm{P}} \mathbf{spec}\ e : \tau \hookrightarrow v[e'] : \tau'}$$

$$\text{(QIN)}\quad \frac{\Delta, h : \delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \rho}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow \Lambda h.e' : \delta \Rightarrow \rho}$$

$$\text{(QOUT)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \delta \Rightarrow \rho \qquad \Delta \Vdash v : \delta}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'((v)) : \rho}$$

$$\text{(GEN)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \forall \alpha.\sigma} \quad (\alpha \notin FV(\Delta) \cup FV(\Gamma))$$

$$\text{(INST)}\quad \frac{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \forall \alpha.\sigma}{\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\,\sigma} \quad (\mathrm{dom}(S) = \alpha)$$

Figura 1.5: Reglas de especialización (continuación)

ización de esta función en el sistema principal es:

$$\begin{aligned}
&\vdash_{\mathrm{P}} \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x +^D 1^D) \\
&\quad : \mathbf{poly}\ (Int^S \to^D Int^D) \\
&\hookrightarrow \Lambda h.h[\Lambda h_x.\lambda^D x'.h_x + 1] \\
&\quad : \forall s.\mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int)\ s \Rightarrow \mathbf{poly}\ s
\end{aligned}$$

En este ejemplo la polivarianza de la función (que dependerá de los valores actuales de su argumento estático) se abstrae mediante la conversión h, que abstrae la evidencia de que el tipo $\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int$ es efectivamente más general que las instancias que se encuentren para s. A su vez, el valor real de x en cada una de las expresiones que se puedan obtener de esta función polivariante se abstrae mediante la variable de evidencia h_x.

Ejemplo 1.10. En este ejemplo, la misma función polivariante del ejemplo

anterior se instancia de dos formas diferentes:

$$\begin{array}{l}\vdash_{\mathrm{P}} \mathbf{let}^{D}\ f = \mathbf{poly}\ (\lambda^{D}x.\mathbf{lift}\ x +^{D} 1^{D}) \\ \quad \mathbf{in}\ (\mathbf{spec}\ f\ @^{D}\, 42^{S}, \mathbf{spec}\ f\ @^{D}\, 17^{S})^{D} : (Int^{D}, Int^{D})^{D} \\ \hookrightarrow \mathbf{let}\ f' = \Lambda h.\lambda x'.h + 1 \\ \quad \mathbf{in}\ (f'((42))@\bullet, f'((17))@\bullet) : (Int, Int)^{D}\end{array}$$

Notar la interacción entre las anotaciones **poly** y **spec**, que introducen la abstracción y aplicación de la 'evidencia' de que la variable estática x tiene como valor un número entero (como lo requiere **lift**).

Ejemplo 1.11. Finalmente ilustraremos la forma en que se especializa el término del ejemplo 1.4-3:

$$\begin{array}{l}\vdash_{\mathrm{P}} \lambda^{D}f.\mathbf{spec}\ f\ @^{D}\, 13^{S} : \mathbf{poly}\ (Int^{S} \to^{D} Int^{D}) \to^{D} Int^{D} \\ \hookrightarrow \Lambda h_5.\lambda f.h_5[f]@\bullet \\ : \forall s.h_2 : \mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int)\ s, \\ \quad h_5 : \mathrm{IsMG}\ s\ (\hat{13} \to Int) \Rightarrow \mathbf{poly}\ s \to Int\end{array}$$

Este ejemplo muestra una función de alto orden que recibe una función (polivariante) y la especializa para aplicarla a un valor estático específico. En el tipo residual se expresa que la función debe tener un tipo que sea cualquier instancia de $\forall t.\mathrm{IsInt}\ t \Rightarrow t \to Int$, pero que al menos debe ser tan general como $\hat{13} \to Int$. Estas condiciones a su vez se representan en el término con variables de evidencia, que lo modificarán en forma acorde cuando se conozcan las soluciones que prueben los predicados.

El sistema $\vdash_{\mathrm{P}}$ es estable bajo sustituciones, propiedad esencial para la prueba de principalidad.

Proposición 1.12. Para cualquier juicio de especialización $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ y sustitucion S se cumple que $S\,\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\,\sigma$.

Tambien se establece que la especialización respeta la relación $\Vdash$ de implicación de conjuntos de predicados:

Proposición 1.13. Para todo juicio $h : \Delta \mid \Gamma \vdash_{\mathrm{P}} \mathrm{e} : \tau \hookrightarrow e' : \sigma$, si se cumple la implicación $h' : \Delta' \Vdash v : \Delta$ entonces $h' : \Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'[v/h] : \sigma$.

1.3.4 Algoritmo

Además de las reglas de especificación de la figura 1.5, [Martínez López and Hughes, 2002] presentan un algoritmo de especialización principal, mediante un sistema de reglas *syntax-directed* similar a las de traducción de Mark Jones en su teoría de tipos calificados [Jones, 1994a]. Este algoritmo se basa en el clásico algoritmo W de Milner [Milner, 1978], y las reglas pueden ser interpretadas como una gramática de atributos [Rèmy, 1989]. El sistema de reglas se presenta en la figura 1.6. El algoritmo utiliza además algunos algoritmos auxiliares:

- Unificación: Variación del algoritmo clásico de Robinson [Robinson, 1965], modificado para trabajar con sustituciones bajo cuantificaciones universales (esto es, bajo tipos residuales polivariantes). Se utiliza '*skolemización*' de las variables cuantificadas para sustituirlas por variables denotadas c que varían sobre un conjunto infinito enumerable sin intersección con otro conjunto de variables. El algoritmo deriva juicios de la forma $\sigma_c \sim^U \sigma_c$, siendo U una sustitución. El algoritmo se presenta en la figura 1.8 (interpretado como una gramática de atributos, los tipos residuales son atributos heredados y la sustitución es uno sintetizado).
- Implicación: La idea del algoritmo es calcular el conjunto de predicados que deberían ser agregados a una asignación Δ para que implique a un predicado δ. El algoritmo toma una asignación de predicados Δ y un predicado δ a implicar, y devuelve el conjunto de predicados a agregar y la evidencia que pruebe δ. De esta forma la única regla necesaria es

$$h : \delta \mid \Delta \Vdash_{\mathrm{W}} h : \delta \quad (h \text{ nueva})$$

  es decir, genera una variable nueva de evidencia y agrega $h : \delta$ al conjunto de predicados.

  Versiones más refinadas del algoritmo podrían manejar eficientemente predicados *ground* como IsInt $\hat{n}$ o la aparición de predicados duplicados; pero, como se verá, las etapas de simplificación (capítulo 2) y resolución (capítulo 3) pueden manejar estas situaciones eficientemente.
- Relación fuente-residual: El algoritmo que calcula la relación entre tipos fuente y residual especificada en la figura 1.4 se implementa también con una gramática de atributos, con juicios de la forma $\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'$, tomando τ como entrada (atributo heredado) y sintetizando Δ y τ'. Las reglas se dan en la figura 1.7. El algoritmo se utiliza en las reglas (W-DLAM) y (W-SPEC), para asegurar que el tipo residual elegido (para la variable en el primer caso y para la expresión **spec** en el segundo) sea un residuo posible del tipo original.

Finalmente, enunciaremos la propiedad más importante y que motiva este sistema, la principalidad:

Teorema 1.14. Sean Γ, e, y τ tales que $\Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$ para algún Δ, e', y σ. Entonces existen e'_p y σ_p tales que $\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'_p : \sigma_p$ y que para todo Δ'', e'', y σ'' tales que $\Delta'' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'' : \sigma''$ existen una conversión C y una sustitución R tales que $C : (\mid R\sigma_p) \geq (\Delta'' \mid \sigma')$ y $C[e'_p] = e''$.

Esta propiedad asegura que para cualquier término (junto con su tipo) que se especialice existe una especialización (llamada *especialización principal*) que

obtiene un término y tipo residuales a partir de los cuales toda otra especialización del término se puede obtener mediante una instanciación adecuada (la aplicación de una conversión que convierte términos del tipo hallado al tipo deseado). Los detalles de esta prueba pueden encontrarse en [Martínez López and Hughes, 2002].

La principalidad permite especializar programas en forma modular, especializando cada fragmento independientemente del contexto en que se usa para luego ensamblar las partes especializadas. En el área de compiladores existe una noción análoga, y tan importante como ésta: compilación modular.

1.3.5 Extensiones

El lenguaje fuente definido en este capítulo es en realidad un subconjunto del soportado por el sistema de [Martínez López and Hughes, 2002]. El subconjunto tomado contiene las expresiones principales de un λ-cálculo (variables, abstracciones, y aplicaciones), enriquecido con constantes y operaciones numéricas, y extendido con las construcciones para utilización de polivarianza.

Entre las posibles y más simples extensiones del lenguaje aquí utilizado se pueden mencionar:

- Booleanos u otro tipo de datos 'básico', junto con sus operaciones primitivas (tal como los enteros se incluyen en este trabajo).
- Tipos de datos algebraicos en general, con constructores y operaciones de destrucción (pattern matching) como parte de instrucciones de selección **case**.

Estas extensiones pueden ser especializadas sin mayores complicaciones, extendiendo el sistema de reglas de especialización de manera directa. Además de las mencionadas existen otras como

- recursión,
- funciones estáticas,
- polimorfismo

cuyo tratamiento para lograr especialización principal es complejo.

Consideramos que el lenguaje utilizado en este trabajo presenta un conjunto de problemas representativo de los que queremos definir y resolver, y que puede ser extendido para soportar extensiones como las anteriormente descriptas.

$$\text{(W-VAR)}\quad \frac{x : \tau \hookrightarrow x' : \tau' \in \Gamma}{\emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} x : \tau \;\hookrightarrow\; x' : \tau'}$$

$$\text{(W-DINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^D : Int^D \;\hookrightarrow\; n : Int$$

$$\text{(W-D+)}\quad \frac{\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : Int^D \hookrightarrow e_1' : Int \qquad \Delta_2 \mid S_2\,S_1\,\Gamma \vdash_{\mathrm{W}} e_2 : Int^D \hookrightarrow e_2' : Int}{S_2\,\Delta_1, \Delta_2 \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^D e_2 : Int^D \;\hookrightarrow\; e_1' + e_2' : Int}$$

$$\text{(W-LIFT)}\quad \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : Int^S \;\hookrightarrow\; e' : \tau' \qquad \Delta' \mid \Delta \Vdash_{\mathrm{W}} v : \mathrm{IsInt}\ \tau'}{\Delta', \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{lift}\ e : Int^D \;\hookrightarrow\; v : Int}$$

$$\text{(W-SINT)}\quad \emptyset \mid \mathrm{Id}\,\Gamma \vdash_{\mathrm{W}} n^S : Int^S \;\hookrightarrow\; \bullet : \hat{n}$$

$$\text{(W-S+)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : Int^S \;\hookrightarrow\; e_1' : \tau_1' \\ \Delta_2 \mid S_2\,S_1\,\Gamma \vdash_{\mathrm{W}} e_2 : Int^S \;\hookrightarrow\; e_2' : \tau_2' \\ \Delta \mid S_2\,\Delta_1, \Delta_2 \Vdash_{\mathrm{W}} v : t := S_2\,\tau_1' + \tau_2'\end{array}}{S_2\,\Delta_1, \Delta_2, \Delta \mid S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1 +^S e_2 : Int^S \;\hookrightarrow\; \bullet : t} \quad (t \text{ nueva})$$

$$\text{(W-DLAM)}\quad \frac{\begin{array}{c}\Delta \vdash_{\text{W-SR}} \tau_2 \;\hookrightarrow\; \tau_2' \\ \Delta' \mid S\,(\Gamma, x : \tau_2 \hookrightarrow x' : \tau_2') \vdash_{\mathrm{W}} e : \tau_1 \;\hookrightarrow\; e' : \tau_1'\end{array}}{\Delta', S\,\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} \lambda^D x.e : \tau_2 \to^D \tau_1 \;\hookrightarrow\; \lambda x'.e' : S\,\tau_2' \to \tau_1'} \quad (x' \text{ nueva})$$

$$\text{(W-DAPP)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_2 \to^D \tau_1 \;\hookrightarrow\; e_1' : \tau_2' \to \tau_1' \\ \Delta_2 \mid S_2\,S_1\,\Gamma \vdash_{\mathrm{W}} e_2 : \tau_2 \;\hookrightarrow\; e_2' : \tau_2'' \qquad S_2\,(\tau_2' \to \tau_1') \sim^U \tau_2'' \to t\end{array}}{\Delta_1, \Delta_2 \mid U S_2 S_1\,\Gamma \vdash_{\mathrm{W}} e_1\ @^D\ e_2 : \tau_1 \;\hookrightarrow\; e_1' @ e_2' : U\,t} \quad (t \text{ nueva})$$

$$\text{(W-POLY)}\quad \frac{\begin{array}{c}h : \Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \tau \;\hookrightarrow\; e' : \tau' \\ \Delta' \mid \emptyset \Vdash_{\mathrm{W}} v : \mathrm{IsMG}\ (\mathrm{Gen}_{S\,\Gamma}(\Delta \Rightarrow \tau'))\ s\end{array}}{\Delta' \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \;\hookrightarrow\; v[\Lambda h.e'] : \mathbf{poly}\ s} \quad (s \text{ nueva})$$

$$\text{(W-SPEC)}\quad \frac{\begin{array}{c}\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : \mathbf{poly}\ \tau \;\hookrightarrow\; e' : \mathbf{poly}\ \sigma \\ \Delta' \vdash_{\text{W-SR}} \tau \;\hookrightarrow\; \tau' \qquad \Delta'' \mid \Delta, \Delta' \Vdash_{\mathrm{W}} v : \mathrm{IsMG}\ \sigma\ \tau'\end{array}}{\Delta, \Delta', \Delta'' \mid S\,\Gamma \vdash_{\mathrm{W}} \mathbf{spec}\ e : \tau \;\hookrightarrow\; v[e'] : \tau'}$$

Figura 1.6: Algoritmo de especialización principal (primera parte)

$$\text{(W-DTUPLE)}\quad \frac{\begin{array}{c}\Delta_1 \mid S_1\,\Gamma \vdash_{\mathrm{W}} e_1 : \tau_1 \hookrightarrow e_1' : \tau_1' \\ \cdots \quad \Delta_n \mid S_n\,S_{n-1}\ldots S_1\,\Gamma \vdash_{\mathrm{W}} e_n : \tau_n \hookrightarrow e_n' : \tau_n'\end{array}}{\begin{array}{l}S_{n-1}\ldots S_1\,\Delta_1,\ldots,\Delta_n \mid S_n\ldots S_1\,\Gamma \vdash_{\mathrm{W}} (e_1,\ldots,e_n)^D \\ \quad : (\tau_1,\ldots,\tau_n)^D \hookrightarrow (e_1',\ldots,e_n') : (S_n\ldots S_2\,\tau_1', S_n\ldots S_3\,\tau_2',\ldots,\tau_n')\end{array}}$$

$$\text{(W-DPRJ)}\quad \frac{\Delta \mid S\,\Gamma \vdash_{\mathrm{W}} e : (\tau_1,\ldots,\tau_n)^D \hookrightarrow e' : \tau' \quad \tau' \sim^U (t_1,\ldots,t_n)}{\Delta \mid US\,\Gamma \vdash_{\mathrm{W}} \pi^D_{i,n}\,e : \tau_i \hookrightarrow \pi_{i,n}\,e' : U\,t_i} \quad \left(t_1,\ldots,t_n \text{ nueva}\right)$$

$$\text{(W-DLET)}\quad \frac{\begin{array}{c}\Delta_2 \mid S_2\,\Gamma \vdash_{\mathrm{W}} e_2 : \tau_2 \hookrightarrow e_2' : \tau_2' \\ \Delta_1 \mid S_1\,S_2\,\Gamma, x : \tau_2 \hookrightarrow x' : \tau_2' \vdash_{\mathrm{W}} e_1 : \tau_1 \hookrightarrow e_1' : \tau_1'\end{array}}{\begin{array}{l}S_1\,\Delta_2, \Delta_1 \mid S_1 S_2\,\Gamma \vdash_{\mathrm{W}} \mathbf{let}^D\ x = e_2\ \mathbf{in}\ e_1 : \tau_1 \\ \qquad\qquad \hookrightarrow \mathbf{let}\ x' = e_2'\ \mathbf{in}\ e_1' : \tau_1'\end{array}} \quad \left(x' \text{ nueva}\right)$$

Figura 1.6: Algoritmo de especialización principal (continuación)

$$\mathrm{IsInt}\ t \vdash_{\text{W-SR}} Int^S \hookrightarrow t \quad \left(t \text{ nueva}\right)$$

$$\emptyset \vdash_{\text{W-SR}} Int^D \hookrightarrow Int$$

$$\frac{\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1' \qquad \Delta_2 \vdash_{\text{W-SR}} \tau_2 \hookrightarrow \tau_2'}{\Delta_1, \Delta_2 \vdash_{\text{W-SR}} \tau_2 \rightarrow^D \tau_1 \hookrightarrow \tau_2' \rightarrow \tau_1'}$$

$$\frac{(\Delta_1 \vdash_{\text{W-SR}} \tau_1 \hookrightarrow \tau_1')_{i=1,\ldots,n}}{\Delta_1,\ldots,\Delta_n \vdash_{\text{W-SR}} (\tau_1,\ldots,\tau_n)^D \hookrightarrow (\tau_1',\ldots,\tau_n')}$$

$$\frac{\Delta \vdash_{\text{W-SR}} \tau \hookrightarrow \tau'}{\mathrm{IsMG}\ \sigma\ s \vdash_{\text{W-SR}} \mathbf{poly}\ \tau \hookrightarrow \mathbf{poly}\ s} \quad \left(\sigma = \mathrm{Gen}_{\emptyset}(\Delta \Rightarrow \tau')\ y\ s \text{ nueva}\right)$$

Figura 1.7: Algoritmo para calcular la relación entre tipos fuente-residual

$$c \sim^{\mathrm{Id}} c$$

$$\hat{n} \sim^{\mathrm{Id}} \hat{n}$$

$$Int \sim^{\mathrm{Id}} Int$$

$$\frac{\alpha \notin FV(\sigma)}{\alpha \sim^{[\sigma/\alpha]} \sigma}$$

$$\frac{\alpha \notin FV(\sigma)}{\sigma \sim^{[\sigma/\alpha]} \alpha}$$

$$\frac{\tau_1' \sim^{T} \tau_2' \qquad T\,\tau_1'' \sim^{U} T\,\tau_2''}{\tau_1' \to \tau_2' \sim^{UT} \tau_1'' \to \tau_2''}$$

$$\frac{\tau_{11}' \sim^{T_1} \tau_{21}' \quad T_1\,\tau_{12}' \sim^{T_2} T_1\,\tau_{22}' \quad \ldots \quad T_{n-1}\ldots T_1\,\tau_{1n}' \sim^{T_n} T_{n-1}\ldots T_1\,\tau_{2n}'}{(\tau_{11}', \ldots, \tau_{1n}') \sim^{T_n \ldots T_1} (\tau_{21}', \ldots, \tau_{2n}')}$$

$$\frac{\sigma \sim^{U} \sigma'}{\mathbf{poly}\ \sigma \sim^{U} \mathbf{poly}\ \sigma'}$$

$$\frac{\sigma[\alpha/c] \sim^{U} \sigma'[\alpha'/c]}{\forall\alpha.\sigma \sim^{U} \forall\alpha'.\sigma'} \quad (c \text{ nueva})$$

$$\frac{\delta \sim^{U} \delta' \qquad \rho \sim^{U} \rho'}{\delta \Rightarrow \rho \sim^{U} \delta' \Rightarrow \rho'}$$

$$\frac{\tau \sim^{U} \tau'}{\mathrm{IsInt}\ \tau \sim^{U} \mathrm{IsInt}\ \tau'}$$

$$\frac{\tau \sim^{T} \tau' \qquad T\,\tau_1 \sim^{U} T\,\tau_1' \qquad UT\,\tau_2 \sim^{V} UT\,\tau_2'}{\tau := \tau_1 + \tau_2 \sim^{VUT} \tau' := \tau_1' + \tau_2'}$$

$$\frac{\sigma_1 \sim^{T} \sigma_2 \qquad T\,\sigma_1' \sim^{U} T\,\sigma_2'}{\mathrm{IsMG}\ \sigma_1\ \sigma_1' \sim^{UT} \mathrm{IsMG}\ \sigma_2\ \sigma_2'}$$

Figura 1.8: Algoritmo de Unificación

Capítulo 2

Simplificación

La verdad raras veces es tan simple, aunque no muchas verdades acaban siendo tan complicadas como al fin lo fue la mía.

Un pescador del Mar Interior (1994)
Úrsula K. Le Guin

Basado en el capítulo anterior, se trabaja sobre el problema de simplificación de los conjuntos de predicados obtenidos y utilizados durante la especialización de tipos de [Martínez López and Hughes, 2002] (capítulo 1), utilizando las ideas de 'simplificación' y 'mejora' de la teoría de tipos calificados de [Jones, 1994b].

El objetivo de este capítulo es formalizar el proceso de simplificación para el sistema de especialización, problema que se describe y motiva en la sección 2.1. El enfoque consistirá en definir en primer lugar el concepto de relación de simplificación (sección 2.2), para luego en las secciones 2.3 y 2.4 especificar una relación de simplificación adecuada para nuestro lenguaje (que será luego implementada en la sección 5.1). La incorporación del proceso de simplificación a la especialización de [Martínez López and Hughes, 2002] se hace en la sección 2.5, y en la sección 2.6 se relaciona este trabajo con el de [Jones, 1994b] expandiendo nuestra simplificación con algunas reglas ilustrativas. Finalmente, en la sección 2.8 se dan las conclusiones finales a la vez que se introduce el trabajo del capítulo 3.

2.1 Motivación

Con el modelo propuesto en [Martínez López and Hughes, 2002] la especialización se realiza en etapas. En la primer etapa, la especialización en sí misma, se toma un término en el lenguaje fuente junto con su tipo y se lo especializa a un término y un tipo del lenguaje residual. El sistema de tipos residual contiene *tipos calificados*, como propone [Jones, 1994a], que contienen *predicados* difiriendo la toma de ciertas decisiones mediante la inclusión explícita de condiciones que se deben cumplir para finalizar la especialización. La forma de completar la especialización es "resolver" estos predicados: se provee la evidencia necesaria que los "pruebe", y posteriormente esa evidencia es eliminada fundiéndose con el término. Estas últimas etapas son explicadas en los capítulos 3 y 4.

Dado que durante la especialización se generan predicados por la aplicación de algunas reglas, es útil agregar a esta etapa un proceso de *simplificación de predicados*, en donde se trata de reducir el número de los mismos o traducirlos a formas más simples, con el doble objetivo de ahorrar trabajo computacional a las siguientes etapas y mejorar la legibilidad de los términos obtenidos.

La generación de predicados se realiza mediante la aplicación de reglas del algoritmo de especialización. Cada aplicación puede crear un nuevo predicado, en muchos casos similar o relacionado (por alguna forma de implicación) con los ya producidos para el término. Por ejemplo, al especializar el siguiente término utilizando el algoritmo W especificado en la reglas de la figura 1.6:

$$\lambda^D x.\mathbf{lift}\ ((x +^S 1^S) +^S (x +^S 1^S)) : Int^S \rightarrow^D Int^D$$

se obtiene

$$\begin{aligned} \Lambda h_t\ h_a\ h_b\ h_c.\lambda x.h_c : \forall t\ t'\ t''.\quad & h_t : \mathrm{IsInt}\ t, \\ & h_a : t' := t + 1, \\ & h_b : t'' := t + 1, \\ & h_c : t''' := t'' + t' \Rightarrow t \rightarrow Int \end{aligned}$$

donde se puede observar la clase de predicados que pueden generarse a partir de la especialización de un término fuente: en algunos casos los mismos son redundantes, como puede ser cuando un predicado es 'implicado' por otros – en este caso, el predicado más débil podría eliminarse sin cambiar el significado del conjunto, ya que cualquiera tipo que satisfaga al nuevo conjunto también debe satisfacer al conjunto completo.

Mediante simplificación, este término podría convertirse a

$$\Lambda h, h'.\lambda x.h' : \forall t, t', t'''.h : t' := t + 1, h' : t''' := t' + t' \Rightarrow t \rightarrow Int$$

que es 'más simple' que el original, y a su vez es equivalente (bajo una equivalencia que se definirá formalmente en este capítulo).

De lo expuesto se puede ver que es necesaria la simplificación de los predicados producidos durante la especialización, tarea desarrollada en las siguientes secciones.

2.2 Especificación

El primer paso del trabajo será establecer formalmente la noción de simplificación, detallando las propiedades que se esperan de cualquier relación que se desee usar a tal efecto. Es importante notar que cualquier relación que cumpla las condiciones de la definición de esta sección (no solamente la implementada en las siguientes secciones) podría ser utilizada como simplificación en un prototipo. De esta forma, se estudiarán las propiedades de la simplificación independientemente de cómo pueda ser implementada.

Notación. Dado que en simplificación se utilizan mayormente conversiones de la forma $(\Lambda h.[])((v))$ y composición de las mismas, se utilizará por comodidad "listas de reemplazo de evidencia", que no son más que una forma restringida de conversiones. De esta forma, usaremos $h{\leftarrow}v$ para denotar la conversión anterior, y compondremos reemplazos con $h{\leftarrow}v \cdot C$ para denotar $(\Lambda h.C)((v))$ (por convención, utilizaremos a $\cdot$ con asociatividad a derecha). De esta forma, $h_1{\leftarrow}v_1 \cdot \ldots \cdot h_n{\leftarrow}v_n$ denotará la conversión $(\Lambda h_n \ldots h_1.[])((v_n \ldots v_1))$. Será útil saber que el operador $\cdot$ de composición de reemplazos cumple la siguiente propiedad:

Lema 2.1. Las conversiones $h{\leftarrow}h$ son neutras para $\cdot$ (notar que la conversion $[]$ es un caso particular de estas conversiones): O sea, para cualquier conversión C y variables de evidencia h se cumple que $h{\leftarrow}h \cdot C = C = C \cdot h{\leftarrow}h$.

Siendo la anterior la única notación nueva y necesaria pasamos a la principal definición de este capítulo, la de relación de simplificación.

Definición 2.2 (Simplificación). Una relación $S; C \mid h : \Delta \trianglerighteq h' : \Delta'$ es una relación de *simplificación* si la conversión es de la forma $C = h{\leftarrow}v$ y se cumple que:

$$\begin{array}{ll} \text{(i)} & h' : \Delta' \Vdash v : S\Delta \\ \text{(ii)} & S\Delta \Vdash \Delta' \end{array}$$

Las dos condiciones de la definición establecen que los predicados son equivalentes en cuanto a satisfactibilidad (bajo la sustitución S), y la forma de la conversión C garantiza que puede ser usada para convertir un término de cierto tipo que asume predicados de Δ en otro que asume los predicados de Δ', haciendo los cambios necesarios sobre las evidencias (recordar que la igualdad de conversiones que aparece en esta condición no es estructural, sino la equivalencia de conversiones definida en [Martínez López and Hughes, 2002]).

Ejemplo 2.3. Dados

$$\begin{aligned}\Delta_1 =& h_1 : \text{IsInt } \hat{9},\\ & h_2 : \text{IsInt } t''',\\ & h_3 : t := \hat{1} + \hat{2},\\ & h_4 : t' := t + \hat{3},\\ & h_5 : t''' := t'' + t'\\ \Delta_2 =& h_5 : t''' := t'' + \hat{6}\end{aligned}$$

esperamos que nuestra implementación de $\trianglerighteq$ cumpla que

$$S; C \mid \Delta_1 \trianglerighteq \Delta_2$$

donde $S = [t/\hat{3}][t'/\hat{6}]$ y $C = h_1{\leftarrow}9 \cdot h_2{\leftarrow}h_5 \cdot h_3{\leftarrow}3 \cdot h_4{\leftarrow}6$; justificado por:

- $h_1 : \text{IsInt } \hat{9}$ se simplifica trivialmente, y 9 es su evidencia.
- $h_2 : \text{IsInt } t'''$ es implicado por el quinto predicado.
- $h_3 : t := \hat{1} + \hat{2}$ se simplifica calculando el resultado de $1 + 2$, y generando la sustitución que cambia t por $\hat{3}$ en el cuarto predicado.
- $h_4 : t' := t + \hat{3}$, tiene resultado conocido para t' una vez que se conoce el de t (que proviene del predicado anterior).

Sin embargo, la especificación no fuerza a la relación a cumplir la simplificación del ejemplo anterior: en la definición cabe perfectamente una relación $\trianglerighteq$ tal que relacione sólo todas las listas de predicados con sí mismas, con la sustitución identidad y la conversión [] como testigos de esa 'simplificación'. Por supuesto, cualquier simplificación con alguna utilidad práctica, como la especificada más adelante en este capítulo, será capaz de realizar más simplificaciones que sólo las obtenidas por la identidad.

Concluimos la sección con una propiedad de clausura de la relación de simplificación respecto a las sustituciones, que permite determinar que si una lista de predicados simplifica a otra entonces dos instancias de ellas mantienen esta relación. Dado que la definición de $\trianglerighteq$ incluye una sustitución en sí misma, la propiedad no es cierta en el caso general, sino sólo para sustituciones que se 'comporten bien' entre sí.

Definición 2.4. Dos sustituciones S y T se dicen *compatibles* respecto a un tipo τ, denotado $S \leftrightarrow_\tau T$, si $TS\tau = ST\tau$. La noción se extiende en forma natural a esquemas de tipos σ, predicados δ y listas de predicados Δ.

Lema 2.5. Sea la simplificación $T; C \mid \Delta \trianglerighteq \Delta'$. Si S y T son compatibles sobre Δ, es decir $S \leftrightarrow_\Delta T$, entonces T es también una simplificación de $S\Delta$, o sea $T; C \mid S\Delta \trianglerighteq S\Delta'$.

Esta propiedad será de particular utilidad en el capítulo 3, donde habremos de combinar resoluciones con simplificaciones.

$$\text{(SimEntl)}\ \frac{h : \Delta \Vdash v_\delta : \delta}{\mathrm{Id}; h_\delta \leftarrow v_\delta \mid h : \Delta, h_\delta : \delta \trianglerighteq h : \Delta}$$

$$\text{(SimComp)}\ \frac{S; C \mid h : \Delta \trianglerighteq h' : \Delta' \qquad S'; C' \mid h' : \Delta' \trianglerighteq h'' : \Delta''}{S'S; C' \circ C \mid h : \Delta \trianglerighteq h'' : \Delta''}$$

$$\text{(SimCtx)}\ \frac{S; C \mid h_1 : \Delta_1 \trianglerighteq h_2 : \Delta_2}{S; C \mid h_1 : \Delta_1, h' : \Delta' \trianglerighteq h_2 : \Delta_2, h' : S\Delta'}$$

$$\text{(SimPerm)}\ \frac{S; h_1, h_2 \leftarrow v_1, v_2 \mid h_1 : \Delta_1, h_2 : \Delta_2 \trianglerighteq h_1' : \Delta_1', h_2' : \Delta_2'}{S; h_2, h_1 \leftarrow v_2, v_1[h_2/v_2] \mid h_2 : \Delta_2, h_1 : \Delta_1 \trianglerighteq h_2' : \Delta_2', h_1' : \Delta_1'}$$

Figura 2.1: Reglas estructurales de simplificación

2.3 Implementación de una simplificación

Cualquier relación que cumpla las condiciones de la definición 2.2 puede ser utilizada como relación de simplificación. En esta sección se especificará mediante un sistema de reglas de deducción natural una relación básica de especificación, que será extendida más adelante en la sección 2.4. Estas reglas serán implementadas en el prototipo de la sección 5.1.

Para el lenguaje de especialización que utilizamos hay ciertas simplificaciones que pueden ser consideradas básicas y son inherentes al lenguaje y sus construcciones principales; sus reglas aparecen en la figura 2.1. Estas no son las únicas reglas posibles; el conjunto de reglas variará de acuerdo a las extensiones del lenguaje que se considere y a los aspectos puntuales que se deseen optimizar en los conjuntos de predicados.

En primer lugar, (SimEntl) incorpora al sistema de reglas de simplificación la semántica contenida en la relación $\Vdash$ de implicación de listas de predicados. Esta regla se usará para eliminar predicados redundantes: predicados que son, según $\Vdash$, más 'débiles' que otros de la misma lista.

La segunda regla, (SimComp), dota a la relación de simplificación de transitividad sobre las asignaciones de predicados: con esta regla se pueden componer derivaciones de una simplificación, construyendo la sustitución y la conversión como composiciones de las obtenidas en las premisas. Esta composición se realiza en forma de preservar la consistencia de la relación simplificación definida, en este caso basándose en la propiedad análoga de la relación de implicación de asignaciones de predicados.

La regla (SimCtx) permite extender una simplificación dada para trabajar en asignaciones de predicados más amplias, agregando una lista de predicados arbitraria a ambos lados de la ecuación. Notar que es necesario del lado derecho, 'simplificado', la aplicación de la sustitución S: de otra manera variables de tipo

que son simplificadas por ella podrían quedar libres en la expresión simplificada.

Finalmente, la regla (SimPerm) clausura la relación de simplificación para tratar a las listas de predicados como listas no ordenadas. Esta regla es imprescindible para simplificar cualquier lista de predicados, sin preocuparse del orden en que se introducen los elementos de una asignación de predicados. El objetivo de la regla es que, dada una simplificación conocida, tanto la asignación de predicados original como la simplificada puedan permutarse, preservando la relación de simplificación entre ambas asignaciones.

En esta última regla es necesario modificar la conversión dada en la premisa, ya que en general no es lo mismo hacer el reemplazo $h_1, h_2 \leftarrow v_1, v_2$ que $h_2, h_1 \leftarrow v_2, v_1$, porque al abstraerse y aplicarse en diferente orden las variables podrían ligarse de diferente forma o quedar libre alguna que no lo estuviese – en general los ejemplos tratarán a la conversión de la premisa como igual a la conversión obtenida por la regla, por la forma en que serán elegidas las variables de evidencia, pero la condición es necesaria para la consistencia de la regla. (Aunque normalmente estas dos conversiones suelen ser iguales en la práctica, porque no hay colisiones entre variables de evidencia.)

Las dos últimas reglas, (SimCtx) y (SimPerm), son complementarias y necesarias para utilizar simplificaciones en cualquier ambiente. Normalmente, en donde es necesaria aplicar una de ellas para adaptar una asignación de predicados a un contexto dado también se aplicará la otra. Por esta razón definimos la siguiente regla *derivada* (SimCHAM), como fusión de las dos anteriores:

$$\text{(SimCHAM)} \quad \frac{\Delta_1' \approx \Delta_1 \quad S; C \mid \Delta_1 \trianglerighteq \Delta_2 \quad \Delta_2 \approx \Delta_2'}{S; C^{\approx} \mid \Delta_1', \Delta \trianglerighteq \Delta_2', S\Delta}$$

Esta regla se inspira originalmente en la idea de la *Chemical Abstract Machine* en [Berry and Boudol, 1992]. En la regla, la equivalencia $\approx$ se define como la menor congruencia entre listas de predicados que contiene a $\Delta, \Delta' \approx \Delta', \Delta$, es decir, se considera congruentes a dos listas si son obtenibles una de la otra por permutación. Siendo $\alpha\Delta_1 = \Delta_1'$ la permutación involucrada, es fácil definir $C^{\approx}$ que reemplace las evidencias en el orden correspondiente a la permutación α, sustituyendo las siguientes por aquellas que ya fueron aplicadas previamente, generalizando el caso de la regla (SimPerm). De esta forma se trabaja sobre estas listas como si fueran no ordenadas; y las reglas, especificadas 'localmente', se pueden aplicar en cualquier contexto.

La siguiente propiedad se prueba en forma directa sobre la definición de $\Vdash$, y será necesaria para la prueba de consistencia de (SimPerm).

Lema 2.6. La relación $\Vdash$ está clausurada por permutaciones en las asignaciones de predicados de ambos operandos. Es decir, si $h_1 : \Delta_1, h_2 : \Delta_2 \Vdash v_1 : \Delta_1', v_2 : \Delta_2'$ entonces $h_2 : \Delta_2, h_1 : \Delta_1 \Vdash v_2 : \Delta_2', v_1 : \Delta_1'$.

Cabe señalar que la 'simplificación nula' mencionada en la sección 2.2 está incluida en la relación especificada por estas reglas: en particular, la regla (SimEntl) permite derivarla sabiendo que $h : \Delta \Vdash \emptyset$ es una regla estructural de $\Vdash$. (Notar que el lado derecho de $\Vdash$ esta la regla, $v_\delta : \delta$, denota una lista unitaria de predicados y no un predicado, de acuerdo a las convenciones de notación de la sección 1.2.1.)

Es importante notar que el orden de los predicados es irrelevante sólo si se está hablando de un contexto, pero una vez que esta información se introduce en un tipo este orden pasa a ser fundamental, ya que la ligadura (antes explícita, por nombre) de las variables de evidencia con sus respectivos predicados se hace implícita basada en el orden en que aparecen en un tipo calificado $\delta_1 \Rightarrow \ldots \Rightarrow \delta_n \Rightarrow \tau'$.

Para que las reglas básicas expuestas en la figura 2.1 definan realmente una relación de especificación se deben cumplir las propiedades requeridas de la definición 2.2, lo cual se demuestra en el siguiente teorema.

Teorema 2.7. Las reglas (SimEntl), (SimComp), (SimCtx) y (SimPerm) (figura 2.1) definen una relación de simplificación, y la regla derivada (SimCHAM) es consistente con ella.

La eliminación de predicados redundantes de un conjunto se realiza aplicando la regla (SimEntl). Esto incluye tanto a predicados redundantes respecto a otro conjunto de predicados con el que coexisten (tal como aparece en la regla) como a predicados que siempre se cumplen y pueden ser descartados en cualquier contexto. Por ejemplo, se cumple que $\emptyset \Vdash \mathrm{IsInt}\ \hat{n}$ para todo n, por lo que un predicado como $\mathrm{IsInt}\ \hat{3}$ es eliminado en el residuo de

$$(\lambda^D x.x)\ @^D\ 3^S$$

Este tipo de simplificación se basa en la semántica que haya sido expresada por la relación $\Vdash$, que para nuestro lenguaje incluye predicados como $\mathrm{IsInt}\ \hat{n}$ para cualquier entero n.

Además de predicados como $\mathrm{IsInt}\ \hat{n}$, la misma regla (SimEntl) es capaz de eliminar predicados $\mathrm{IsMG}\ \sigma_1\ \sigma_2$ para casos particulares donde ninguno de los σ_i es una variable de esquema, y existe una conversión que prueba $C : \sigma_1 \geq \sigma_2$. Esta propiedad es esencial para la implementación de la resolución de restricciones, como se verá en el capítulo 3.

Como comentario acerca de la regla (SimEntl), notar que la conversión utilizada en esta regla es estructuralmente diferente a la requerida por la definición 2.2. Según la definición, la conversión debería ser de la forma $h, h_\delta \leftarrow v, v'$ para alguna elección adecuada de v y v'; es decir, como la asignación de predicados contiene

$$\text{(SimOp}_{res}\text{)}\quad \frac{t \sim^S \hat{n}}{S; h_\delta{\leftarrow}n \mid h_\delta : t := \hat{n_1} \otimes \hat{n_2} \trianglerighteq \emptyset}\ \left(n{=}n_1 \otimes n_2\right)$$

$$\text{(SimMG}_U\text{)}\quad \frac{C : \sigma_2 \geq \sigma_1}{\mathrm{Id}; h_2{\leftarrow}h_1 \circ C \mid h_1 : \mathrm{IsMG}\ \sigma_1\ s, h_2 : \mathrm{IsMG}\ \sigma_2\ s \trianglerighteq h_1 : \mathrm{IsMG}\ \sigma_1\ s}$$

Figura 2.2: Reglas básicas de simplificación

a h y h_δ la conversión debería reemplazar todas ellas. Sin embargo, en esta regla (así como en cualquier otro lado en este trabajo) se trabaja bajo la igualdad de conversiones definida por su semántica, y justificado por el lema 2.1 sabemos que $h_\delta{\leftarrow}v_\delta = h{\leftarrow}h \cdot h_\delta{\leftarrow}v_\delta$, tal como se requiere en 2.2.

Un comentario similar cabe para la conversión utilizada en la regla (SimComp): siendo $C = h{\leftarrow}v$ y $C' = h'{\leftarrow}v'$ con las condiciones de las reglas, $C' \circ C = h{\leftarrow}v[v'/h']$, de forma que la conversión reemplaza exactamente las variables de evidencia de $h : \Delta$. La justificación formal de esta última aserción aparece en las demostraciones de consistencia de estas reglas (en el apéndice A).

2.4 Reglas básicas

Además de las reglas estructurales presentadas en 2.3, existirán diferentes reglas de acuerdo al lenguaje específico y a los objetivos de la simplificación. En el lenguaje sobre el que se trabaja en [Martínez López and Hughes, 2002], las reglas de especialización al especializar expresiones estáticas y polivariantes producen predicados como IsMG y $\_ := \_ \otimes \_$. En algunos casos, los predicados generados pueden ser simplificados; algunas reglas para su tratamiento aparecen en la figura 2.2.

La primer regla, (SimOp$_{res}$), internaliza la semántica de los operadores aritméticos binarios del lenguaje como suma o resta Esta regla generaliza la noción clásica de *constant folding* en la especialización, por lo que vale la pena un análisis detallado. El residuo de la especialización de una expresión aritmética estática es siempre $\bullet$, habiéndose trasladado toda su información al tipo, en forma de predicados. Estos predicados tendrán básicamente la misma forma que la expresión, utilizando variables de tipo para representar los resultados intermedios (sus subexpresiones). Cuando se conoce el valor de todos los operandos de uno de estos predicados, el mismo puede simplificarse reemplazando cada ocurrencia de su variable de tipo por el resultado de la operación. De esta forma, la sustitución utilizada en (SimEntl) es la que reemplaza esta variable por el valor del resultado, y la conversión provee el mismo resultado como evidencia. Reglas similares existirán en la implementación para operadores unarios y para otros

tipos como los booleanos.

Las otra regla, (SimMG$_U$), elimina usos redundantes de predicados IsMG. Cuando dos predicados son cota superior de una misma variable, y las cotas además son comparables (esto es, una es más general que la otra), el predicado cuya cota es más general no está agregando información al tipo. Así, se elimina el predicado y se provee como evidencia la conversión que lleva de un tipo a otro compuesta con la variable de evidencia que se resolverá cuando pueda probarse el IsMG restante. Esta regla tiene una particular importancia respecto al objetivo de ganar eficiencia con la simplificación: durante la resolución de los predicados (capítulo 3) la complejidad se verá afectada por el número y naturaleza de predicados IsMG involucrados.

Al igual que para las reglas básicas, su uso se justifica en el siguiente teorema:

Teorema 2.8. La adición de las reglas (SimOp$_{res}$) y (SimMG$_U$) de la figura 2.2 define una relación de simplificación.

En forma similar a (SimMG$_U$), se podría pensar en incluir una regla para eliminar cotas inferiores redundantes:

$$\text{(SimMG}_L\text{)}\quad \frac{C : \sigma_2 \geq \sigma_1}{\mathrm{Id}; h_1 \leftarrow C \circ h_2 \mid h_1 : \mathrm{IsMG}\ s\ \sigma_1, h_2 : \mathrm{IsMG}\ s\ \sigma_2 \trianglerighteq h_2 : \mathrm{IsMG}\ \sigma_2\ s}$$

Sin embargo, aunque la regla no es inconsistente con las demás ni conlleva a contradicciones, esta regla no es de utilidad en el sistema de especialización en que trabajamos, ya que las cotas inferiores que se introducen como predicados corresponden a expresiones **spec**, y sus tipos son siempre *ground* (no contienen variables): por lo tanto no pueden ser comparables salvo que sean idénticos (y la regla (SimEntl) puede eliminar entre otras cosas predicados duplicados, por propiedad de $\vdash$). Además, para poder completar el proceso de Eliminación de Evidencia (capítulo 4) se requiere que la evidencia de todas las cotas inferiores sea una variable hasta el momento en que decida eliminarse (sección 4.4.2).

Las reglas presentadas en este capítulo (más las reglas para las extensiones de la sección 5.2) pueden parecer restringidas, poco generales. Su objetivo es el de simplificar exactamente las construcciones generadas por el algoritmo de especialización, y están diseñadas para poder trabajar sobre su salida (un ejemplo es la omisión de la regla recién comentada). Si bien podría intentarse hacer reglas de uso más general, se perdería su aplicabilidad en casos puntuales como los producidos por este algoritmo y se lograrían menos simplificaciones. En el diseño de este sistema no puede dejarse de lado la consideración de generalidad (reglas más elegantes pero menos efectivas para este lenguaje y algoritmo) vs. particularidad (reglas ad hoc, con más efectividad simplificando los predicados que aparecen en la práctica). En un extremo se obtendría una simplificación

tan general que resultaría incapaz de simplificar ningún predicado dependiente del lenguaje; en el otro extremo se podría especificar una relación que resultara indecidible o intratable al intentar llevarla a la forma de algoritmo [Jones, 1994b].

2.5 Simplificación de especializaciones

Las reglas dadas arriba, y en general cualquier relación de simplificación especificada, sólo determinan qué conjuntos de predicados pueden ser reemplazados por otros con la misma semántica (y posiblemente más simples, de acuerdo a la definición de $\trianglerighteq$). Pero nada se dice en ellas de la forma de utilizar la simplificación: cuándo utilizarla, sobre qué predicados y por qué reemplazarlos.

Siguiendo a [Jones, 1994a] especificaremos su uso agregando este concepto a la relación de especialización, para luego modificar el algoritmo que la implementa.

Primero agregaremos al sistema de especialización P (especificado en la figura 1.5) una nueva regla, (SIMP):

$$\text{(SIMP)}\quad \frac{h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma \qquad S; C \mid h : \Delta \trianglerighteq h' : \Delta'}{h' : \Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow C[e'] : S\sigma}$$

Esta regla incorpora la simplificación al proceso de especialización. Para preservar las propiedades del sistema P, debe ser probado que esta regla es consistente con las demás, lo cual es consecuencia directa del siguiente teorema.

Teorema 2.9 (Consistencia de (SIMP)). Se pueden realizar simplificaciones en el sistema de especialización: Dada la especialización

$$h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma,$$

si $S; C \mid h : \Delta \trianglerighteq h' : \Delta'$ entonces

$$h' : \Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow C[e'] : S\sigma.$$

La adición de la regla (SIMP) convierte al sistema de especialización en "no determinístico" (no es *syntax-directed*, en la terminología usual). Este hecho no causa problemas en la práctica. Primero, siempre se pueden reducir dos aplicaciones consecutivas de (SIMP) a una sola, por transitividad de $\Vdash$. Además, dado que el único lugar en que las asignaciones de predicados son realmente usadas en los términos es en la regla que especializa expresiones *poly*, éste es el único punto donde es beneficioso usar la regla (SIMP) durante una especialización: inmediatamente antes de (W-POLY).

Otro punto que no está determinado por (SIMP) es la elección de v, Δ', S y C. Dado que la relación de simplificación no es necesariamente funcional, el nuevo

conjunto de predicados, la sustitución y la conversión pueden no ser únicos. Una implementación adecuada de un algoritmo de especialización utilizaría en lugar de la relación de simplificación a alguna función *simplify* tal que $simplify(h, \Delta)$ devuelva una terna $\langle v : \Delta', S, C \rangle$ tal que no sólo se cumpla $S; C \mid h : \Delta \trianglerighteq v : \Delta'$, sino que su elección corresponda a la versión más simplificada (de acuerdo a algún criterio) del conjunto de predicados. Esta técnica de incorporar la simplificación a la versión algorítmica del (en nuestro caso) especializador se basa en el trabajo de Jones [Jones, 1994a], en donde incorpora su noción de simplificación al algoritmo de tipado de tipos calificados.

Continuando con su idea, la regla (W-POLY) del algoritmo de especialización se puede reemplazar por:

$$\text{(W-POLY)}\quad \frac{h : \Delta \mid S\,\Gamma \vdash_{\mathsf{W}} e : \tau \hookrightarrow e' : \tau'}{h'' : \mathrm{IsMG}\ \sigma\ s \mid TS\,\Gamma \vdash_{\mathsf{W}} \mathbf{poly}\ e : \mathbf{poly}\ \tau \hookrightarrow e'' : \mathbf{poly}\ s}$$

donde:

$$\begin{aligned} &e'' = h''[\Lambda h.C[e']] \\ &(h' : \Delta', T, C) = simplify(h : \Delta) \\ &\sigma = Gen_{TS\Gamma}(\Delta' \Rightarrow T\tau') \\ &s \text{ y } h'' \text{ son variables frescas} \end{aligned}$$

Con esta versión del algoritmo, los predicados se simplifican antes de ser introducidos en los esquemas de tipos de **poly**s. Como se nombró en la sección 1.3.4, el algoritmo de implicación de [Martínez López and Hughes, 2002] agrega al contexto todo predicado que necesite ser implicado, aún si ya fuera implicado por el mismo: es en este punto entonces donde se realizan las simplificaciones que justifican esta elección para la implicación.

Es importante notar que, sin embargo, la simplificación no puede eliminar todos los predicados antes de "empaquetarlos" dentro de un IsMG: algunos predicados contendrán por ejemplo variables libres, que aún no pudieron resolverse (por ejemplo un predicado IsInt t), y esos predicados se transfieren al tipo sin modificación alguna. En algún momento posterior durante la especialización, tales variables de tipo tomarán su valor final, y los predicados que las contienen podrán ser simplificados una vez que se los extraiga del IsMG: luego de la especialización de cada **spec** que utilice la expresión polivariante.

Incorporando estas reglas a los sistemas de especialización de las figuras 1.5 y 1.6 se completa el objetivo de agregar simplificación al proceso de especialización, cuya consistencia queda justificada por el teorema 2.9. En las secciones siguientes discutiremos algunas reglas y características adicionales, para luego llegar a las conclusiones finales.

2.6 Simplificación y mejora

Observando las reglas definidas de simplificación puede notarse que la mayoría de las reglas de las figuras 2.3 y 2.4, excepto (SimOp_{rcs}), no introducen sustitución alguna (salvo la identidad) sino que acarrean o componen sustituciones ya armadas. Este hecho se basa en que estas sustituciones trabajan principalmente sobre predicados redundantes, por sí solos o en conjunto con los demás, que pueden ser eliminados simplemente proveyendo la evidencia (dada por la conversión) necesaria para transformar los términos, pero no necesitan tomar decisiones acerca de los valores de variables de tipo.

Sin embargo hay más predicados que pueden ser simplificados, decidiendo valores de variables de tipo. Las reglas que implementen estas simplificaciones estarán basadas en la semántica de cada predicado, hallando los valores para las variables de tipo cuando sea posible y generando de esta manera sustituciones. Este tipo de simplificaciones son las que llamamos *mejoras*, por su analogía con trabajo previo.

2.6.1 Noción de mejora

El caso de (SimOp_{rcs}) es un caso particular que resuelve el valor de una variable de tipo t que depende *funcionalmente* del valor de otros tipos ya conocidos. Mientras que las demás reglas hablan de relaciones de 'implicación' entre predicados, (SimOp_{rcs}) se refiere a la satisfactibilidad de un predicado cuya forma es particular (al igual que otras reglas que se verán más adelante). La clave es analizar la pregunta ¿cuáles de los posibles t cumplen $t := \hat{n}_1 \otimes \hat{n}_2$? Claramente, sólo uno: el que representa a $n_1 \otimes n_2$. Esta información es usada para *resolver* la variable t a ese valor, y eliminar el predicado.

Los dos casos de simplificación mencionados se corresponden a las nociones de 'simplificación' y 'mejora' (*simplification & improving*) de [Jones, 1994b], que en nuestro sistema se fusionan en una misma relación de simplificación. A pesar de no diferenciarlas debemos tener en cuenta que reglas como (SimOp_{rcs}) toman ciertas decisiones que modifican los tipos. Por ello es que las reglas de simplificación en sí (en el sentido de [Jones, 1994b]) pueden aplicarse a cualquier subtérmino o predicado, mientras que las que resuelven valores están limitadas a ciertos contextos.

Para ilustrar esta restricción, utilizaremos un ejemplo en que se especializan valores booleanos y expresiones **if-then-else** estáticas. La especialización de estas expresiones, en [Martínez López and Hughes, 2002], se hace extendiendo los lenguajes de tipos y expresiones fuente:

$$\begin{aligned} e &::= \ldots \mid \mathit{True}^S \mid \mathit{False}^S \mid \mathbf{if}^S\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid e ==^S e \\ \tau &::= \ldots \mid \mathit{Bool}^S \end{aligned}$$

y se introducen principalmente dos predicados nuevos, con los tipos y valores de evidencia necesarios:

$$\begin{array}{rcl} v & ::= & \ldots \mid \mathit{True} \mid \mathit{False} \\ \tau' & ::= & \ldots \mid \hat{\mathit{True}} \mid \hat{\mathit{False}} \\ \delta & ::= & \ldots \mid \tau' \ ?\ \delta \mid !\tau' \ ?\ \delta \end{array}$$

(se omiten algunas construcciones por simplicidad en la exposición, los detalles están en el trabajo original). Entre otras, las reglas para probar los predicados condicionales son:

$$\frac{\Delta \Vdash v : \delta}{\Delta \Vdash v : \hat{\mathit{True}} \ ?\ \delta} \qquad \Delta \Vdash \bullet : \hat{\mathit{False}} \ ?\ \delta$$

De esta manera los predicados con una guarda falsa pueden simplificarse trivialmente; y en otro caso es necesario probar el predicado interior.

Las expresiones **if** estáticas se especializan entonces 'envolviendo' con guardas diferentes a los predicados de ambas ramas, de acuerdo a la variable que representa el valor de la expresión condicional estática. Con estas herramientas presentaremos con un ejemplo la situación en que la aplicación de una regla de simplificación queda restringida a su tipo: de simplificación pura o de mejora.

Ejemplo 2.10. En la especialización del siguiente término se ilustra la dificultad de aplicar *mejora* sobre cualquier variable.

$$\begin{array}{l} \Delta \mid \emptyset \vdash_{\mathrm{P}} \lambda^D b.\lambda^D x.\, \mathbf{if}^S\, b \\ \qquad\qquad \mathbf{then}\ \ \mathbf{if}^D\, \mathit{True} \\ \qquad\qquad\qquad\ \ \mathbf{then}\ x^S +^S (5^S +^S 7^S) \\ \qquad\qquad\qquad\ \ \mathbf{else}\ 13^S \\ \qquad\qquad \mathbf{else}\ x \\ \qquad\quad : \mathit{Bool}^S \to^D \mathit{Int}^S \to^D \mathit{Int}^S \\ \hookrightarrow\ \ \lambda b.\lambda x.\, \mathbf{if}_v\ h_b \\ \qquad\qquad \mathbf{then}\ \ \mathbf{if}\ \mathit{True}\ \mathbf{then}\ \bullet\ \mathbf{else}\ \bullet \\ \qquad\qquad \mathbf{else}\ x \\ \qquad\quad : t_b \to t_x \to t_r \end{array}$$

donde

$$\begin{array}{l} \Delta = h_x : \mathrm{IsInt}\ t_x, h_b : \mathrm{IsBool}\ t_b, h_r : \mathrm{IsInt}\ t_r, \\ \quad h_1 : t_r := \mathrm{if}\ t_b\ \mathrm{then}\ t_{13}\ \mathrm{else}\ t_x, \\ \quad h_2 : t_b\ ?\ t_{12} := \hat{5} + \hat{7}, \\ \quad h_3 : t_b\ ?\ t_{13} := t_x + t_{12}, \\ \quad h_4 : t_b\ ?\ t_{13} \sim \hat{13} \end{array}$$

En lugar de finalizar la especialización introduciendo los predicados en el tipo y clausurando universalmente todas sus variables (reglas (QIN), (GEN)), podría intentarse extender las reglas presentadas para ser aplicables dentro del cuerpo de los predicados guardados.

En este caso, se aplicaría (SimOp_{rcs}) al predicado $t_b \ ?\ t_{12} := \hat{5} + \hat{7}$, decidiendo el valor de t_{12} a ser $\hat{12}$. También se unificaría el valor de t_{13} con $\hat{13}$, quedando finalmente sólo un predicado guardado: $\hat{13} := t_x + \hat{12}$. Es claro que este predicado fuerza a t_x a valer $\hat{1}$ (la regla que infiere esta simplificación se presenta en la figura 2.3, a la que se agregan reglas de simplificación de unificaciones en forma directa). Finalmente el tipo de la función (aparte de algunos predicados que no llegan a simplificarse — aquellos que dependen del valor de b) resulta en $t_b \to \hat{1} \to t_r$: la función sólo puede aplicarse con un 1 en el segundo argumento. Observando nuevamente el término inicial a especializar, se aprecia que si b es *False*, entonces la función es la identidad *para cualquier número*, no sólo para el 1.

La inconsistencia se produjo al *decidir* el valor de t_x. Cualquier decisión que se tome bajo una guarda está sujeta a que el valor de la variable guardada habilite el predicado guardado. Por lo tanto, si los efectos de la simplificación trascienden a ese predicado y se reflejan en el contexto, puede llegarse a una inconsistencia. Las simplificaciones que salen 'hacia afuera' de las guardas son exactamente aquellas que introducen sustituciones; esta idea coincide con el concepto de *mejora* de [Jones, 1994b]. En nuestro caso hay libertad de aplicar simplificaciones sin mejora bajo guardas, pero sólo se pueden aplicar mejoras si sus sustituciones no afectan al contexto: es decir, si las variables cuyo valor se decide no ocurren fuera de la guarda (como el caso de t_{12} en el ejemplo). (Incidentalmente, los dos casos válidos son caras de la misma moneda: la sustitución identidad nunca modificará el contexto, mientras que una que decida el valor de una variable no lo modificará sólo si esta variable no ocurre fuera del mismo.)

En el ejemplo, la máxima simplificación que se puede lograr –al menos hasta que se provea más información sobre la variable b– es el contexto

$$\begin{aligned}\Delta' = {} & h_x : \text{IsInt } t_x, h_b : \text{IsBool } t_b, h_r : \text{IsInt } t_r, \\ & h_1 : t_r := \text{if } t_b \text{ then } \hat{13} \text{ else } t_x, \\ & h_3 : t_b \ ?\ \hat{13} := t_x + \hat{12},\end{aligned}$$

Mantendremos la formulación de nuestras relaciones de simplificación de la forma más simple, y no agregaremos complejidad contemplando el tratamiento de predicados guardados (que requerirían el chequeo de ocurrencias de cada variable fuera de la guarda en que se encuentre).

$$\text{(SimOP}_{inv}\text{)}\quad \frac{t \sim^S \hat{n}}{S; h{\leftarrow}n_1 \mid h : \hat{n_1} := t \otimes \hat{n_2} \trianglerighteq \emptyset} \quad \left(\exists \text{ único } n : n_1{=}n{\otimes}n_2\right)$$

$$\text{(SimOp}_{neu}\text{)}\quad \frac{t \sim^S \hat{k}}{S; [] \mid h : t' := t \otimes t' \trianglerighteq h : \text{IsInt } t'} \quad \left(k \text{ neutro de } \otimes\right)$$

$$\text{(SimCase}_1\text{)}$$

$$\text{Id}; h{\leftarrow} \bullet \mid h : t := \text{IsCase } \tau_v \text{ of } \tau_p \rightarrow \tau_b \trianglerighteq h : \tau_v \sim \tau_p, h' : t \sim \tau_b \quad \left(h', h'' \text{ nueva}\right)$$

Figura 2.3: Reglas de simplificación y mejora *backwards*

2.6.2 Reglas de mejora

La misma idea de 'mejora' basada en observar la satisfactibilidad de ciertos predicados nos lleva a definir nuevas reglas, que aparecen en la figura 2.3. Sobre la misma idea estará basada tambien la *resolución de restricciones*, tratada a partir del capítulo 3.

La regla (SimOP$_{inv}$) resuelve el valor de una variable como operando de una operación aritmética inyectiva (como la suma o resta) cuyo resultado y el otro operando son conocidos. La simplificación elimina el predicado, sustituye la variable por su único valor posible, y genera la conversión que provee el valor del resultado de la operación como evidencia (valor que ya era conocido).

La segunda regla, (SimOp$_{neu}$), se utiliza en el caso de que uno de los operandos y el resultado de una operación son la misma variable: aunque su valor sea desconocido, su aparición determina que el valor del otro operando debe ser el neutro de la operación (si éste existe y es único). En nuestro lenguaje, provee el tipo $\hat{0}$ en las sumas. La sustitución generada reemplaza la variable resuelta. El predicado se cambia por otro que sólo dice de la incógnita que es un número entero, y la conversión generada es la identidad porque no se provee evidencia alguna.

Reglas similares a (SimOP$_{inv}$) y (SimOp$_{neu}$) existirán para otro orden de los operandos, y para otro tipo de operaciones (booleanas, relacionales, etc.).

La última regla, (SimCase$_1$), es algo más compleja y su uso menos evidente, pero demuestra también la capacidad de inferencia de información de un buen sistema de simplificación. Esta regla simplifica los predicados generados por una construcción **case** con solo un *pattern*. Tal expresión sólo puede evaluarse en caso de que el valor siendo comparado coincida con el dado en el patrón, por lo que se justifica basar la simplificación en esta aserción. El predicado se elimina, junto con su evidencia (la conversión la reemplaza por $\bullet$), y dos nuevos predicados son creados: uno fuerza a que el tipo de la expresión se identifique con el del patrón, y el otro a que el del resultado coincida con el del cuerpo de ese patrón.

Si bien esta optimización puede parecer excesiva, demasiado particular, la especialización del intérprete de λ-cálculo de [Martínez López and Hughes, 2002] construye numerosas ocurrencias de términos *case* con patrones únicos (por ejemplo, cuando se sabe que en una aplicación uno de los términos tiene que tener tipo funcional, y se extrae su tipo con un único pattern *Fun*). Sin este tipo de regla, tales predicados permanecerían inalterados sin poder simplificarse.

Este caso ilustra la necesidad de reglas *ad-hoc* para la simplificación de casos particulares, motivados por la aparición de ciertos predicados en la especialización de un término específico, o por el uso de una regla particular de especialziación.

2.7 Discusión

En esta sección se discuten algunas características de las reglas presentadas que, si bien pueden ser pasadas por alto, capturan diferentes ideas y objetivos presentes en las reglas de simplificación.

2.7.1 Simplificación incremental

En la regla (SimCase$_1$) aparecen predicados no utilizados hasta el momento: los de unificación. La regla podría haberse presentado requieriendo que los tipos sean unificados en las premisas, y devolviendo el unificador como sustitución:

$$\frac{(\tau_v, t) \sim^S (\tau_p, \tau_b)}{S; h \leftarrow \bullet \mid h : t := \text{IsCase } \tau_v \text{ of } \tau_p \rightarrow \tau_b \trianglerighteq \emptyset}$$

En esta regla, el predicado se elimina y se provee $\bullet$ como evidencia –al igual que en la regla original– pero la unificación de los tipos se realiza mediante una sola sustitución. De esta manera, los valores de todas las variables son decididos por la misma sustitución, en una simplificación atómica.

De acuerdo a lo discutido en 2.6.1, esta regla es una regla de mejora, y por lo tanto no podría ser aplicada en cualquier contexto. Esto permitiría seguir simplificando, aún bajo guardas, a todos o parte de los nuevos predicados (al menos aquellos cuyas variables no afectan al contexto de la guarda).

Una posible pregunta sería si las demás reglas que utilizan sustituciones, como (SimOp$_{res}$) o (SimOP$_{inv}$), puede enunciarse de forma que no produzcan sustituciones. La respuesta es negativa, y radica en que en los predicados simplificados por estas dos últimas reglas el valor de la evidencia es significativo para el término (es el resultado de la operación) mientras que en (SimCase$_1$) el valor es $\bullet$ sin importar el resultado de las unificaciones.

Por estas razones, en nuestro caso preferimos utilizar reglas con unificaciones incrementales en lugar de monolíticas, siempre que fue posible.

2.7.2 Inversión del flujo de información

Aunque hayan sido motivadas por la idea de mejora, no todas las reglas presentadas en la figura 2.3 lo son — (SimCase_1) no entra en esta definición.

La razón de haberlas agrupado de esta manera es que comparten otra particularidad: el sentido en que se propaga la información es el inverso a las demás reglas. Este tipo de propagación no es común en las especializaciones y simplificaciones clásicas.

En las demás reglas, la información se utiliza propagándola 'hacia arriba' en el árbol sintáctico del término: por ejemplo, el conocer el valor todos los argumentos (subárboles) de una operación estática lleva a conocer el valor del resultado (el nodo que los contiene). En el caso de estas reglas, la información puede 'bajar' en este árbol, si se conoce suficiente como para propagarse en este sentido (como puede ser cuando el resultado de una operación estática y todos los operandos excepto uno son conocidos).

Con ello se muestra la capacidad de optimización que puede tener un sistema de simplificación correctamente diseñado para un lenguaje particular (recordemos que estas reglas están fuertemente atadas a la semántica de las construcciones que simplifican).

2.7.3 Eliminación de modelos

A lo largo de este capítulo se nombró el hecho de que las simplificaciones no agregan información ni toman decisiones sobre una asignación de predicados, salvo aquellas implicadas por el mismo. Analizaremos ahora en detalle la veracidad de esta afirmación.

Consideremos la siguiente especialización:

$$h : t := \hat{1} + \hat{1} \mid \emptyset \vdash_{\mathrm{p}} 1^S +^S 1^S : Int^S \hookrightarrow \bullet : t$$

Es claro que el contexto $h : t := \hat{1} + \hat{1}$ puede ser simplificado, y que su solución es el tipo $\hat{2}$. Pero consideremos la simplificación (absurda pero correcta de acuerdo a la definición):

$$S; h \leftarrow h \mid h : t := \hat{1} + \hat{1} \trianglerighteq h : \hat{8} := \hat{1} + \hat{1}$$

siendo $S = [\hat{8}/t]$. Es fácil verificar que se cumplen las condiciones necesarias para que esa sea una simplificación. Utilizando la regla (SIMP) sobre la especialización, se obtiene

$$h : \hat{8} := \hat{1} + \hat{1} \mid \emptyset \vdash_{\mathrm{p}} 1^S +^S 1^S : Int^S \hookrightarrow \bullet : \hat{8}$$

La inconsistencia se hace más evidente aún si se mueve la información estática de vuelta a la expresión, mediante un **lift**:

$$h : \hat{8} := \hat{1} + \hat{1} \mid \emptyset \vdash_{\mathrm{p}} \mathbf{lift}\ (1^S +^S 1^S) : Int^D \hookrightarrow 8 : Int$$

ya que $h : \hat{8} := \hat{1} + \hat{1} \Vdash h : \text{IsInt } \hat{8}$ (regla (IsOpIsInt), figura 1.3). Resulta obvio que el término residual 8 no es una buena especialización de **lift** $(1^S +^S 1^S)$, pero no es tan claro dónde se encuentra la inconsistencia.

El problema no radica en nuestra definición de la relación de simplificación. De hecho, no hay inconsistencia en la derivación ni en el juicio de especialización obtenido: éste puede leerse como "si h es evidencia de que $1 + 1 = 8$, entonces el término **lift** (...) especializa al término residual 8". Como nunca se encontrará una tal evidencia para reemplazar h, el juicio de especialización carece de significado.

¿Por qué se permite entonces esa especialización? Tanto el sistema de especialización como la relación de simplificación que le agregamos se basan en la semántica de cada predicado. Y tal semántica es la que relaciona un predicado con sus formas de probarlo, es decir, la evidencia que lo prueba. Las reglas de construcción de evidencia dependen del lenguaje (ya que dependen de los predicados que se usen para ese lenguaje) y en nuestro caso son aquellas -sólo aquellas- de la figura 1.3.

Si no existe evidencia para un predicado, decimos que el predicado "es no satisfactible", no hay un modelo para el mismo y no puede formar parte en un juicio que represente una especialización completa (en que los contextos se vacían).

Este es un caso conocido en la teoría de tipos calificados. Por ejemplo, una de las analogías en un entorno completamente diferente es el caso de la introducción de predicados que no pueden eliminarse en el lenguaje funcional Haskell [Peyton Jones and Hughes (editors), 1999], que utiliza un sistema de clases basado en esta teoría de tipos calificados. Hay casos en que una función es perfectamente tipable, pero los predicados que contiene su tipo nunca podrán ser resueltos, de forma que no podrán instanciarse sus variables de tipo sobre las clases en donde sus predicados lo requieren.

Por parte del algoritmo de especialización (cuyo prototipo se trata en el capítulo 5) se utilizan también predicados insatisfactibles, "Fail", para representar una rama de una expresión que no puede ser especializada (pero que dentro de un contexto en donde no se necesita especializarla, no hacen fallar al término completo).

La conclusión es que no se requiere un tratamiento especial para este tipo de predicados, y que la relación de simplificación no es incorrecta por poder manipularlos. Simplemente se debe evitar que el *algoritmo* de simplificación, aquel que decida los valores reales de las variables de tipo, elija valores que conserven la satisfactibilidad de los predicados.

2.8 Conclusiones

Es importante notar que las reglas de simplifición presentadas hasta este punto no son necesarias u obligatorias, así como tampoco es exhaustiva la lista: podrían adicionarse o removerse más reglas de acuerdo a los objetivos específicos de la simplificación a implementarse, o a medida que se agreguen construcciones al lenguaje de especialización.

También es importante resaltar que la definición de la relación de simplificación no es el único punto a considerar para definir el comportamiento de la relación. Como se nombró en la sección 2.3, la regla (SimEntl) traslada la semántica de $\Vdash$ a la relación de simplificación, permitiendo la eliminación de algunos predicados espúreos que inevitablemente aparecen en especializaciones. Por esta razón, tanto la definición de $\Vdash$ como la de $\trianglerighteq$ deben tenerse en cuenta en el diseño de un módulo de simplificación para especialización. La implementación de ambas relaciones es tema de la sección 5.1.

A pesar de que cualquier resultado obtenible utilizando simplificación es también obtenible incluyendo esta tarea, hay dos motivos principales para llevarla a cabo:

- *Legibilidad:* Aún en la especialización de programas pequeños la cantidad de predicados generados afecta la legibilidad de los términos obtenidos. Utilizando simplificación, la mayor parte de éstos puede ser descartada, dejando a la vista sólo aquellos significativos.

- *Complejidad:* La cantidad y complejidad de los predicados de un tipo especializado tiene impacto directo en las etapas de resolución de restricciones y eliminación de evidencia (capítulos 3 y 4), en donde predicados innecesarios o demasiado complejos se reflejan en computaciones extra, agregando tiempo de computación a estas tareas.

Finalmente, este capítulo aporta al trabajo las definiciones básicas, conceptos y un lenguaje formal de discurso sobre el que se basarán las ideas detrás de la implementación de la resolución de restricciones y posteriormente eliminación de evidencia, en los capítulos 3 y 4. Como veremos, es sobre la definición de simplificación y sus propiedades que se define la resolución de restricciones, el tema principal de este trabajo.

Capítulo 3

Resolución

The significant problems we face cannot be solved at the same level of thinking we were at when we created them.

Albert Einstein (1879–1955)

Como parte central de este trabajo, en este capítulo describiremos el proceso de resolución, planteado como forma de resolver decisiones diferidas durante la especialización. Este capítulo se basa principalmente en los resultados obtenidos en el estudio de la noción de *simplificación*, tema del capítulo anterior.

Se comienza describiendo el problema y motivando la necesidad de resolución en la sección 3.1, junto con las ventajas de este enfoque. En la sección 3.2 se formaliza el problema utilizando los conceptos ya vistos de simplificación, y en 3.3 se incorpora esta noción a la especialización. La sección 3.4 discute algunas nociones a tener en cuenta que guiarán en la búsqueda de soluciones, y la sección 3.5 describe los algoritmos que la implementan. Finalmente, en 3.6 se discuten algunas alternativas y extensiones del trabajo, y en 3.7 se dan las conclusiones.

3.1 Motivación

Como se vio en el capítulo 1, durante la especialización de expresiones polivariantes (y su uso, construcciones **poly** y **spec**) la forma final no se decide sino que se genera un conjunto de restricciones, expresadas en forma de predicados, que la representan. Estas restricciones, una vez satisfechas, darán lugar a conversiones y sustituciones que transformarán la expresión residual y su tipo a su forma final.

La resolución de restricciones no puede realizarse arbitrariamente durante la especialización debido a que, en el caso general, necesita información global que no es representada por las reglas de especialización - de hecho, la principal motivación del trabajo de [Martínez López and Hughes, 2002] es la producción de especializaciones *principales*, que lleguen a su forma final posiblemente cuando se integren a un contexto mayor, pero sin tener que volver a especializarse.

De esta forma en este enfoque [Aiken, 1999]:

- Se ve a la especialización como un análisis estático del programa, especificado en forma más natural que resolviendo el problema completo: cada pieza de un programa es analizada *localmente*, aislada del resto del programa. La conjunción de todas las restricciones generadas en esta etapa captura las propiedades globales del programa analizado.

- Se separa el proceso en dos partes bien diferenciadas y analizables por separado: una de *especificación*, la forma de encontrar y describir un problema (etapa de especialización); y otra de *implementación* de su solución (resolución de predicados).

- Posibilita la aplicación de ténicas clásicas de *resolución* para el tratamiento de los predicados. En nuestro caso, en donde la resolución de los predicados generados no es un tema profundamente estudiado, provee de un lenguaje en el que poder expresar estos problemas para luego resolverlos.

Ejemplo 3.1. Uno de los ejemplos más simples que ilustran la necesidad de la resolución de restricciones es la siguiente especialización:

$$
\begin{array}{l}
\vdash \mathbf{let}^{D}\ f = \mathbf{poly}\ (\lambda^{D}x.\mathbf{lift}\ x)\ \mathbf{in}\ \mathbf{spec}\ f\ @^{D}\ 13^{S} : Int^{D} \\
\hookrightarrow \Lambda h^{u}, h^{\ell}.\mathbf{let}\ f = h^{u}[\Lambda h_{t}.\lambda x.h_{t}]\ \mathbf{in}\ h^{\ell}[f]@\bullet \\
\quad : \forall s.h^{u} : \mathrm{IsMG}\ (\forall t.h_{t} : \mathrm{IsInt}\ t \Rightarrow t \rightarrow Int)\ s, \\
\qquad h^{\ell} : \mathrm{IsMG}\ s\ (\hat{13} \rightarrow Int) \\
\qquad \Rightarrow Int
\end{array}
$$

En este caso hay una función anotada como polivariante, y un solo uso de la misma. La especialización produce dos restricciones para un esquema de tipo s, pero no decide su valor final. Incidentalmente, ya que el único uso de la variable

s es dentro de la expresión mostrada, su valor puede ser determinado por los lugares donde ocurre: los predicados IsMG.

Como convención, utilizaremos en este capítulo y el siguiente la notación usada en el ejemplo anterior para variables de evidencia. Tanto h_1, h_2, ..., h_x, etc. como h^u, h^ℓ, h_i^u, etc. denotan variables de evidencia: en los primeros casos no se hace distinción del origen de la variable o el predicado que abstrae, mientras que en el segundo denota (como convención) con un superíndice ^{u} a las que marcan cotas superiores de una variable de esquema, y con $^\ell$ a las inferiores.

En este capítulo investigaremos la forma de reconocer cuándo una variable o un predicado pueden ser resueltos, llegando a una especificación formal de esta noción, para luego abocarnos a la tarea de su resolución en forma algorítmica.

3.2 Definición

De forma similar a lo expuesto para la simplificación en el capítulo 2, se definirá la noción de resolución de un conjunto de predicados como una relación y determinando las condiciones que debe cumplir, para luego implementar una resolución particular a nuestro lenguaje de especialización.

Definición 3.2. Una *resolución* de un conjunto de predicados Δ_1 hacia otro conjunto Δ_2, requiriendo los predicados de Δ', es una relación denotada

$$S\colon T; C \mid \Delta_1 + \Delta' \rhd_V \Delta_2$$

donde S y T son sustituciones, C una conversión y V un conjunto de variables de tipo, tales que

$$\begin{array}{ll} \text{(i)} & T; C \mid S\Delta_1, \Delta' \trianglerighteq \Delta_2 \\ \text{(ii)} & \mathrm{dom}(S) \cap (V \cup FTV(\Delta')) = \emptyset \end{array}$$

En este caso diremos que S es una *solución* a las variables que reemplaza ($\mathrm{dom}(S)$), y que V *restringe* la aplicación de S (ya que la sustitución no puede reemplazar variables de V ni variables libres, FTV, de Δ'). El resto de los componentes de la relación aparecen también en la definición de la simplificación, y garantizan las condiciones esperadas de una solución (que se sigan implicando los mismos predicados) mientras que a la vez proveen la forma de convertir términos y tipos para la nueva asignación de predicados (mediante la conversión C y la sustitución T).

A primera vista la resolución parece un caso especial de simplificación, aunque como veremos sus consecuencias son mayores. La solución S presente en la definición hace que las asignaciones Δ_1 y Δ' no sean equivalentes a Δ_2 en el sentido mencionado para la simplificación en el capítulo 2. En el caso particular de que $S = \mathrm{Id}$, se encuentra un caso de simplificación (con la única condición

de que está restringida a V). Pero en cualquier otro caso la resolución es en general irreversible, porque S "decide" los valores de las variables que resuelve, y de esta forma no sólo los predicados originales no implican a los resueltos sino que de hecho pueden existir diferentes soluciones, no comparables, para una misma variable en una misma asignación de predicados. En la sección 3.5 se determinará una forma de calcular soluciones, justificada por el teorema de la sección 4.5.

Ejemplo 3.3. En el ejemplo 3.1, para el término obtenido por especialización existen dos soluciones posibles para s:

- Si se elige $S = [\hat{1}3 \to \mathit{Int}/s]$ como solución, reemplazando s y simplificando el resultado se obtiene:

$$\begin{aligned} &\vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x)\ \mathbf{in\ spec}\ f\ @^D\ 13^S : \mathit{Int}^D \\ &\hookrightarrow \mathbf{let}\ f = \lambda x.13\ \mathbf{in}\ f@\bullet : \mathit{Int} \end{aligned}$$

- Si en cambio $S = [\forall t.h_t : \mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int}/s]$, reemplazando y simplificando se obtiene:

$$\begin{aligned} &\vdash \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x)\ \mathbf{in\ spec}\ f\ @^D\ 13^S : \mathit{Int}^D \\ &\hookrightarrow \mathbf{let}\ f = \Lambda h_t.\lambda x.h_t\ \mathbf{in}\ f((13))@\bullet : \mathit{Int} \end{aligned}$$

Aunque la primera parece una mejor resolución, la segunda corresponde a una noción más general de resolución y sería la elegida por nuestro algoritmo, como se justifica en las secciones siguientes.

La utilización de las conversiones utilizadas en la resolución anterior se explica con más detalle en el ejemplo siguiente.

Ejemplo 3.4. Tomemos el clásico ejemplo donde una función polivariante se especializa de dos formas diferentes:

$$\begin{aligned} &\mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x) \\ &\ \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 13^S, \mathbf{spec}\ f\ @^D\ 17^S) : (\mathit{Int}^D, \mathit{Int}^D) \end{aligned}$$

Al especializar el término, antes de introducir los predicados generados y clausurar el tipo universalmente, se obtiene el término residual

$$\mathbf{let}\ f = h^u[\Lambda h_t.\lambda x.h_t]\ \mathbf{in}\ (h^\ell_{13}[f]@\bullet, h^\ell_{17}[f]@\bullet) : (\mathit{Int}, \mathit{Int})$$

y un contexto con la asignación de predicados

$$\begin{aligned} &h^u : \mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \to \mathit{Int})\ s, \\ &h^\ell_{13} : \mathrm{IsMG}\ s\ (\hat{1}3 \to \mathit{Int}), \\ &h^\ell_{17} : \mathrm{IsMG}\ s\ (\hat{1}7 \to \mathit{Int}) \end{aligned}$$

La simplificación implementada en el capítulo 2 no alcanza para eliminar estos predicados. Pero éstos indican que el valor de s debe ser una instancia del tipo $(\forall t.\text{IsInt } t \Rightarrow t \rightarrow Int)$, más general que $(\hat{1}3 \rightarrow Int)$ y que $(\hat{1}7 \rightarrow Int)$, lo que intuitivamente debería llevar a alguna decisión sobre el valor de s. Incidentalmente, en este caso el único valor que cumple las restricciones es $(\forall t.\text{IsInt } t \Rightarrow t \rightarrow Int)$, probado por las conversiones

$$[](\!(13)\!) : (\forall t.\text{IsInt } t \Rightarrow t \rightarrow Int) \geq (\hat{1}3 \rightarrow Int)$$
$$[](\!(17)\!) : (\forall t.\text{IsInt } t \Rightarrow t \rightarrow Int) \geq (\hat{1}7 \rightarrow Int)$$

Utilizando estas dos conversiones como evidencia para h^{ℓ}_{13} y h^{ℓ}_{17}, y la conversión $[]$ para h^{u}, se podrían eliminar los predicados y obtener

$$\textbf{let } f = \Lambda h_t.\lambda x.h_t \textbf{ in } (f(\!(13)\!)@\bullet, f(\!(17)\!)@\bullet) : (Int, Int)$$

que es, informalmente, una especialización del término original "completamente resuelta" (ya que no contiene decisiones diferidas expresadas en predicados). Utilizando *eliminación de evidencia* (capítulos 1 y 4) el término puede traducirse a una versión en el mismo lenguaje residual de [Hughes, 1996] (en particular, el mismo término que se obtiene con su algoritmo):

$$\textbf{let } f = (\lambda x.13, \lambda x.17)^{S} \textbf{ in } (\pi^{S}_{1,2}\, f, \pi^{S}_{2,2}\, f) : (Int, Int)$$

en donde la función polivariante f tomó dos formas diferentes, de acuerdo a los dos usos que se le da en el programa.

En las secciones siguientes justificaremos el uso de resolución en la especialización, y daremos un algoritmo para efectivamente calcular tales soluciones.

3.3 Especialización

Siguiendo el mismo esquema que para la simplificación en el capítulo 2, se especificará la introducción del proceso de resolución mediante el agregado de una nueva regla, (SOLV), al sistema de especialización:

$$\text{(SOLV)}\quad \frac{\Delta_1 \mid \Gamma \vdash_{\text{P}} e : \tau \hookrightarrow e' : \sigma \qquad S\colon T; C \mid \Delta_1 + \Delta' \rhd_{FTV(\Gamma,\sigma)} \Delta_2}{\Delta_2 \mid TS\Gamma \vdash_{\text{P}} e : \tau \hookrightarrow C[e'] : TS\sigma}$$

Los componentes de la relación de resolución ya se explicaron en la sección 3.2, pero hay una condición que aparece en la regla que no ha sido vista hasta el momento: la restricción $FTV(\Gamma, \sigma)$. Este conjunto de variables impone como condición a la aplicación de la regla que las variables que se resuelvan no ocurran en el tipo del término, σ, ni en el contexto en el que fue tipado, Γ (cuya información pasará en última instancia a σ). Restringe la aplicación de (SOLV)

a variables que contengan en el contexto del término toda la información pertinente sobre ellas: la ocurrencia de esas variables en el tipo significa que más información sobre las mismas puede 'llegar' de otros términos que se especialicen junto con el actual, como se ilustra en el ejemplo siguiente.

Ejemplo 3.5. Sea el término

$$\begin{array}{l}\mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\lambda^D y.(\mathbf{lift}\ x, \mathbf{lift}\ y)) \\ \ \mathbf{in}\ (\mathbf{spec}\ f\ @^D 1^S\ @^D 2^S, f) \\ : ((Int^D, Int^D), \mathbf{poly}\ (Int^S \to Int^S \to (Int^D, Int^D)))\end{array}$$

y su especialización principal.

$$\begin{array}{l}\Lambda h_f^u, h_f^\ell.\mathbf{let}\ f = h_f^u[\Lambda h_x.\Lambda h_y.\lambda x.\lambda y.(h_x, h_y)]\ \mathbf{in}\ (h_f^\ell[f]@ \bullet @\bullet, f) \\ : \forall s.\ \mathrm{IsMG}\ (\forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t \to t' \to (Int, Int))\ s, \\ \quad \mathrm{IsMG}\ s\ (\hat{1} \to \hat{2} \to (Int, Int)) \\ \ \Rightarrow ((Int, Int), \mathbf{poly}\ s)\end{array}$$

De no existir la restricción de las variables a sustituir en la regla de resolución para especialización, la variable s sería candidata a ser resuelta. Resolviendo antes de la aplicación de (QIN) y (GEN) resultaría en el término:

$$\begin{array}{l}\mathbf{let}\ f = \Lambda h_x.\Lambda h_y.\lambda x.\lambda y.(h_x, h_y)\ \mathbf{in}\ (f(\!(1)\!)(\!(2)\!)@ \bullet @\bullet, f) \\ : ((Int, Int), \mathbf{poly}\ (\forall t, t'.\mathrm{IsInt}\ t, \mathrm{IsInt}\ t' \Rightarrow t \to t' \to (Int, Int)))\end{array}$$

Que a primera vista no parece una especialización incorrecta. Sin embargo, al resolver la variable s se ha perdido información. Si el mismo término se encuentra dentro del siguiente programa,

$$\begin{array}{l}\mathbf{let}^D\ e = \mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\lambda^D y.(\mathbf{lift}\ x, \mathbf{lift}\ y)) \\ \qquad\qquad \mathbf{in}\ (\mathbf{spec}\ f\ @^D 1^S\ @^D 2^S, f) \\ \ \mathbf{in}\ \mathbf{let}^D\ id = \lambda^D x.x \\ \ \mathbf{in}\ \mathbf{let}^D\ g = \mathbf{poly}\ (\lambda^D x.\lambda^D y.(\mathbf{lift}\ (id\ @^D x), \mathbf{lift}\ (id\ @^D y))) \\ \ \mathbf{in}\ \mathbf{spec}\ (\mathbf{if}\ True\ \mathbf{then}\ g\ \mathbf{else}\ \mathbf{snd}\ e)\ @^D 4^S\ @^D 4^S\ : (Int, Int)\end{array}$$

el mismo se especializa a

$$\begin{array}{l}\mathbf{let}\ e = \mathbf{let}\ f = h_f^u[\Lambda h_x, h_y.\lambda x.\lambda y.(h_x, h_y)] \\ \qquad\qquad \mathbf{in}\ (h_f^\ell[f]@ \bullet @\bullet, f) \\ \ \mathbf{in}\ \mathbf{let}\ id = \lambda x.x \\ \ \mathbf{in}\ \mathbf{let}\ g = h_g^u[\Lambda h_{xy}.\lambda x.\lambda y.(h_{xy}, h_{xy})] \\ \ \mathbf{in}\ h_e^\ell[\mathbf{if}\ True\ \mathbf{then}\ g\ \mathbf{else}\ \mathbf{snd}\ e]@ \bullet @\bullet \\ : (Int, Int)\end{array}$$

bajo el contexto

$$h_f^u : \text{IsMG } (\forall t', t''.\text{IsInt } t', \text{IsInt } t'' \Rightarrow t' \to t'' \to (Int, Int))\ s,$$
$$h_g^u : \text{IsMG } (\text{IsInt } t \Rightarrow t \to t \to (Int, Int))\ s,$$
$$h_f^\ell : \text{IsMG } s\ (\hat{1} \to \hat{2} \to (Int, Int)),$$
$$h_e^\ell : \text{IsMG } s\ (\hat{4} \to \hat{4} \to (Int, Int)),$$
$$h_t : \text{IsInt } t$$

¡cuyos predicados no pueden resolverse! El problema proviene de haber decidido prematuramente el valor de s, antes de conocer toda la información posible de la misma. La situación se refleja en el hecho de que la variable de esquema s ocurra libre en el esquema σ (o en las asunciones de Γ, que en última instancia se incorporarán a σ). Este hecho puede interpretarse como que no están dadas las condiciones acerca de s: cualquier uso de f sobre otros argumentos, o dentro de otros subtérminos del término en donde e esté contenido, podría hacer 'llegar' información adicional para decidir el valor del esquema s; y cerrar prematuramente el valor de la variable sin conocer el contexto completo (relativo a esa variable) acarrea el problema que la condición de esta regla evita.

Finalmente, la incorporación de la regla (SOLV) al sistema de reglas de especialización se justifica probando su consistencia respecto de las demás reglas.

Teorema 3.6 (Consistencia de (SOLV)). La resolución de restricciones puede realizarse durante la derivación de especializaciones. Para cualquier jucio de especialización $\Delta_1 \mid \Gamma \vdash_P e : \tau \hookrightarrow e' : \sigma$, si se cumple que

$$S : T; C \mid \Delta_1 + \Delta' \rhd_{FTV(\Gamma,\sigma)} \Delta_2$$

entonces también es cierto que $\Delta_2 \mid TS\Gamma \vdash_P e : \tau \hookrightarrow C[e'] : TS\sigma$.

Hasta este punto hemos especificado qué variables de entre las existentes en un término residual puede ser cambiadas por un valor, y las condiciones que deben cumplir estos valores para ser considerados una solución. En las siguientes secciones veremos cómo hallar valores que cumplan con estas condiciones, y de haber más de una solución cuál será elegida y por qué.

3.4 Elección de soluciones

En la sección anterior se definió el concepto de solución a los predicados que incluyen ciertas variables, pero no se describió la forma de resolverlos ni se examinó la clase de predicados que deben ser resueltos.

De la misma forma que en los ejemplos vistos en las secciones anteriores, todos las variables creadas durante la especialización y cuyo valor necesita resolverse se encuentran en predicados IsMG, que provienen de construcciones

poly y **spec**. Estas restricciones son una forma general de desigualdades que deben resolverse, de acuerdo a un orden -parcial- entre esquemas de tipo regido por conversiones (capítulo 1).

Diferentes construcciones del lenguaje como recursión o funciones estáticas, que no son tratadas en este trabajo, generan otro tipo de restricciones que deben ser propiamente estudiadas. En este trabajo nos centraremos en la resolución de predicados IsMG que se generan para nuestro lenguaje.

3.4.1 Soluciones múltiples

Como se dijo, los predicados que se generan restringiendo los valores de una variable tienen la forma de desigualdades bajo un orden parcial de esquemas de tipos.

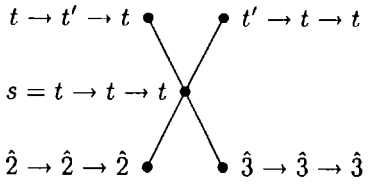
En este capítulo hallaremos soluciones para variables de esquema en ciertas asignaciones de predicados. Nos limitaremos a aquellas asignaciones en que el conjunto de predicados puede ser particionado en tres subconjuntos: (i) predicados IsMG que la incluyen como primer argumento (el tipo 'mayor') (ii) predicados IsMG que la incluyen como los que incluyen a la variable como segundo argumento, y (iii) predicados en donde no ocurre esa variable. De esta manera podremos identificar las 'cotas' que limitan el valor de la variable por $\geq$ a la izquierda o derecha (casos (i) y (ii)) e ignorar el resto de los predicados (caso (iii)).

La primer clase de predicados son aquellos que fuerzan a que el valor de la variable sea 'al menos tan general como' cierto esquema, y provienen cada uno de una construcción **spec** aplicada a un argumento polivariante. La segunda clase se corresponde a las construcciones **poly**, y especifica que el valor que se decida para esa variable debe poder ser el tipo del término bajo cada **poly**, es decir, ser una instancia del esquema.

Viendo a tales clases de predicados como desigualdades, los denominaremos *cotas inferiores* y *cotas superiores*, respectivamente. Dado que el orden parcial da lugar a un reticulado de esquemas de tipos, podemos visualizar las restricciones gráficamente como se ejemplifica en la figura 3.1. En la figura aparecen dos gráficos con los predicados que los generan. En un caso hay dos cotas superiores y dos inferiores, mientras que el segundo tiene sólo una superior y una inferior.

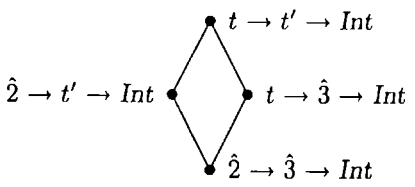
Para poder satisfacer esos predicados, se debe encontrar un valor para s tal que cumpla con esas restricciones. Pero como en el caso de la figura 3.1, en general puede suceder que exista una solución única, que no haya solución alguna o que existan muchas -potencialmente infinitas- soluciones. Entonces aparece el problema de decidir cuál de las soluciones elegir en una implementación de la resolución de restricciones.

Afortunadamente cualquier valor que cumpla las restricciones es una buena solución. En 3.5 la solución particular que elegiremos estará basada en la recolec-



IsMG $(\forall t, t'.\text{IsInt } t, \text{IsInt } t' \Rightarrow t \to t' \to t)\ s$
IsMG $(\forall t, t'.\text{IsInt } t, \text{IsInt } t' \Rightarrow t' \to t \to t)\ s$
IsMG $s\ (\hat{2} \to \hat{2} \to \hat{2})$
IsMG $s\ (\hat{3} \to \hat{3} \to \hat{3})$

(a) Solución única



IsMG $(\forall t, t'.\text{IsInt } t, \text{IsInt } t' \Rightarrow t \to t' \to Int)\ s$
IsMG $s\ (\hat{2} \to \hat{3} \to Int)$

(b) Múltiples soluciones

Figura 3.1: Reticulados de soluciones

ción de cotas superiores, y obteniendo una cota inferior de ellas. Diferiremos la justificación de elección para el capítulo 4, donde se formalizarán las nociones que permitirán probarlo.

3.4.2 Elección de variables

En una implementación práctica del proceso de resolución hay decisiones adicionales que tomar además de definir los valores particulares para las variables. Por ejemplo, esta especificación no determina si las variables deben ser resueltas ser resueltas todas a la vez, o en algún orden particular.

Si bien con extensiones del lenguaje como recursión (en particular, recursión mutua) se pueden generar restricciones en donde dos o más variables dependan mutuamente unas de otras, en nuestro lenguaje se da el caso especial de que de haber soluciones, siempre se puede encontrar alguna *unitaria* (cuyo rango es una única variable).

De esta forma, las soluciones que elegiremos en la práctica resolverán una única variable por vez, y componiéndolas se obtendrá una solución general (si existe) para todas ellas. Justificamos tal composición de soluciones con el siguiente lema:

Lema 3.7. La composición de resoluciones es una resolución. Es decir, si se cumple que $S_2 \leftrightarrow_{S\Delta_1,\Delta'} T_1$ entonces las resoluciones

$$S_1 : T_1; C_1 \mid \Delta_1 + \Delta' \rhd_V \Delta_2 \quad \text{y} \quad S_2 : T_2; C_2 \mid \Delta_2 + \Delta'' \rhd_V \Delta_3$$

implican que

$$S_2S_1 : T_2T_1; C_2 \circ C_1 \mid \Delta_1 + (S_2\Delta', \Delta'') \rhd_V \Delta_3$$

En el ejemplo 3.12 se muestra un caso donde aparecen dos variables a ser resueltas. En este caso el orden parece estar determinado si se quieren realizar mediante soluciones unitarias, ya que la solución de una de las variables depende del valor que se elija para la otra. En el caso general puede haber más de una variable que pueda ser elegida para resolverse en una misma asignación de predicados; en este caso no importará el orden y se iterará el proceso componiendo las soluciones de cualquier variable que se elija.

La propiedad demostrada es de utilidad para nuestra técnica de resolución, que resuelve una variable a la vez, eligiéndolas sin un orden particular. El algoritmo que desarrollamos no permite resolver el caso de dos o más variables mutuamente dependientes. En este caso, y suponiendo que se desarrollara un algoritmo capaz de resolverlas, en vez de hablar de variables únicas y su orden de elección se trataría el caso de componentes fuertemente conexas (del grafo de dependencias de variables), pudiendo seleccionarse y resolverse estas componentes en un orden indistinto como una analogía al caso aquí presentado.

3.4.3 Ambigüedad

Hay otra situación en donde un contexto puede parecer no resoluble, pero para la que veremos que se llega a una solución satisfactoria sin la necesidad de tratamiento adicional. Se trata de los esquemas de tipos ambiguos, tal como se tratan en [Jones, 1994a].

Definición 3.8 (Ambigüedad). Un esquema de tipo $\sigma = \forall\alpha_i.\Delta \Rightarrow \tau$ es *ambiguo* si

$$(FTV(\Delta) \cup \{\alpha_i\}) \setminus FTV(\tau) \neq \emptyset$$

Es decir, un esquema de tipos es ambiguo si predica sobre variables que no aparecen en el tipo. El problema que traen estos esquemas es que la aparición de alguna variable en los predicados implica que se debe proveer evidencia para probarlos; pero si el contexto es incapaz de proveerla (porque clausura a esta variable universalmente) entonces no es posible en el caso general elegir la evidencia a usar — sin importar cuánto se avance o cuánta información se encuentre mediante resolución.

Ejemplo 3.9. El siguiente término se especializa con un tipo ambiguo:

$$\begin{aligned} &\vdash \textbf{let } f = \lambda x.\textbf{lift } x \textbf{ in } 2 : \mathit{Int} \\ &\hookrightarrow \Lambda h.\textbf{let } f = \lambda x.h \textbf{ in } 2 : \forall t.h : \mathrm{IsInt}\ t \Rightarrow \mathit{Int} \end{aligned}$$

En el término aparece entonces una variable de evidencia (correspondiente al predicado irresoluble IsInt t) que no puede eliminarse, por la imposibilidad de elegir –con justificación– una evidencia particular.

Analizando la resolución en un tipo ambiguo, se llega a la conclusión de que su efecto es el de *propagar* las ambigüedades. Por ejemplo, si en el contexto hubiera dos cotas superiores para una variable de esquema:

$$\forall t_1.\text{IsInt } t_1 \Rightarrow Int \qquad \forall t_2.\text{IsInt } t_2 \Rightarrow Int$$

Dado que ambas son cotas superiores, se debe hallar tipo más específico que ambas (bajo la relación $\geq$). Una elección adecuada (que será la utilizada en los algoritmos de la siguiente sección) es elegir la *máxima cota inferior*, comúnmente llamada 'glb', eligiendo en este caso:

$$\forall t_1, t_2.\text{IsInt } t_1, \text{IsInt } t_2 \Rightarrow Int$$

generando un tipo más complejo pero igualmente ambiguo.

Podría sugerirse que se falle *localmente* durante la especialización tan pronto como se encuentra un tipo ambiguo, pero esta optimización no sería consistente. Si bien un subtérmino cuyo tipo es ambiguo haría fallar al resto del término si fuera utilizado, en el caso de estar clausurado bajo una forma polivariante el término completo puede especializarse.

Ejemplo 3.10. En este caso la misma función del ejemplo 3.9 se encuentra dentro del ámbito de un **poly** que nunca llega a utilizarse (en un **spec**). El término especializa a

$$\begin{aligned} &\vdash \textbf{let } g = \textbf{poly } (\lambda x.\textbf{lift } x) \textbf{ in } 2 : Int^D \\ &\hookrightarrow \textbf{let } g = h'[\Lambda h.\textbf{let } f = \lambda x.h \textbf{ in } 2] \textbf{ in } 3 : Int \end{aligned}$$

bajo el contexto

$$h' : \text{IsMG } (\forall t.h : \text{IsInt } t \Rightarrow Int)\ s$$

que se resuelve trivialmente eligiendo como valor para s el mismo que su única –y ambigua– cota superior. Realizando también eliminación de evidencia (capítulo 4) el término tomaría la forma final **let** $g = \bullet$ **in** $3 : Int$.

La conclusión es que no se necesitará un cuidado especial para tratar con tipos ambiguos en el diseño del algoritmo de resolución.

3.5 Algoritmo

En secciones anteriores se definió la noción de resolución y se discutieron algunos aspectos acerca de las posibles soluciones y de la forma de hallarlas. En esta sección daremos una implementación algorítmica de una heurística que permite

hallar, en los casos en que sea posible, una resolución de todos los predicados contenidos en un juicio de especialización.

La intención de esta sección es dar la forma en que las soluciones pueden ser halladas y cómo aplicarlas, y no dar una implementación ejecutable del algoritmo, que utilizando numerosas estructuras de datos y pasos intermedios contendría demasiado detalle para los efectos de esta presentación – algunos detalles del prototipo implementado pueden encontrarse en el capítulo 5. De esta manera presentaremos una serie de funciones que implementan este algoritmo, expresadas en pseudocódigo funcional.

El algoritmo de resolución se implementa con la función **stepSolve**, que dada una asignación de predicados y una variable a resolver, halla su solución y simplifica el resto de los predicados.

La variable a resolver puede ser cualquiera que las que están en condiciones de ser resueltas, como se justificó en la sección anterior, de forma que no se da una función que halle una variable en estas condiciones (ésta no haría mas que verificar condiciones sobre variables libres en los predicados del contexto). Asimismo la obtención del algoritmo final no es más que la composición consigo misma de la función **stepSolve**, iterando sobre cada variable resolubles; por lo que esta última será la única función definida.

Informalmente, se denotará con notación vectorial de elementos de un tipo. Por ejemplo, $\overrightarrow{\sigma}$ representa una lista de esquemas de tipo, mientras que con σ_i se denota cada uno de sus elementos. En la sección 5.1 se darán más detalles de su implementación en el prototipo.

La función **stepSolve** se define en términos de varias funciones que se detallan a continuación (de la más primitiva a la más compleja).

- **glb:** (*greatest lower bound*) Calcula la máxima cota inferior de un conjunto de esquemas de tipos calificados.
  Por simplicidad, describiremos el algoritmo para trabajar sobre *dos* esquemas de tipos, ya que su generalización a cualquier número de esquemas es simple (ya sea iterando el cálculo de a pares o generalizando el algoritmo que se describe).
  Dados $\sigma_1 = \forall \alpha_i.\Delta_1 \Rightarrow \tau_1$ y $\sigma_2 = \forall \beta_j.\Delta_2 \Rightarrow \tau_2$, asumiendo $\alpha_i \cap \beta_j = \emptyset$ (de lo contrario se haría una α-conversión previa), el algoritmo:

  1. Elimina la cuantificación de las variables α_i y β_j obteniendo los tipos calificados $\Delta_1 \Rightarrow \tau_1$ y $\Delta_2 \Rightarrow \tau_2$.
  2. Calcula el unificador más general U que identifica ambos tipos, $\tau_1 \sim^U \tau_2$.
  3. El resultado final es la generalización del tipo unificado, calificado con la instanciación de ambas listas de predicados mediante el unificador ya calculado: $glb(\sigma_1, \sigma_2) = \mathrm{Gen}_{\alpha_i\beta_j}(U\Delta_1, U\Delta_2 \Rightarrow U\tau_1)$

- **conversion:** Dados dos esquemas de tipos σ y σ', devuelve (de existir) una conversión C tal que $C : \sigma \geq \sigma'$.
- **makeMoreGeneral:** Es similar a *conversion*, pero 'fuerza' a los esquemas a ser comparables, en caso de que no lo sean, agregando predicados donde sea necesario. Dados dos esquemas de tipos σ y σ', devuelve una conversión C y una asignación de predicados Δ tal que $C : \sigma \geq (\Delta \mid \sigma')$.
- **findSolution:** Implementa búsqueda de una solución para el lenguaje en el que trabajamos, dadas las cotas superiores e inferiores para una variable de esquema.

$$
\begin{array}{lll}
\mathit{findSolution}\ \vec{\sigma}^u\ \vec{\sigma}^\ell\ s\ = & \textbf{let} & \\
 & \quad \sigma & = \mathit{glb}\ \vec{\sigma}^u \\
 & \quad \vec{C}^u & = \mathit{conversion}\ (\vec{\sigma}^u,\ \sigma) \\
 & \quad (\vec{C}^\ell,\ \vec{\Delta}_f) & = \mathit{makeMoreGeneral}\ (\sigma,\ \vec{\sigma}^u) \\
 & \quad \Delta_f & = \mathit{concat}\ \vec{\Delta}_f \\
 & \textbf{in} & \\
 & \quad (\vec{C}^u,\ \vec{C}^\ell,\ \Delta_f,\ \sigma) &
\end{array}
$$

- **stepSolve:** Es, finalmente, la implementación de un paso de nuestro algoritmo de resolución. Recibiendo un contexto de la forma

$$\Delta = \{h_{u_i} : \text{IsMG}\ \sigma_{u_i}\ s\}, \{h_{l_j} : \text{IsMG}\ s\ \sigma_{l_j}\}, \Delta_s$$

y una variable s, con $s \notin FTV(\Delta_s, \sigma_{u_i}, \sigma_{l_j})$, devuelve una sustitución S, una conversión C, dos listas de predicados Δ_f y Δ' tales que es cierta la resolución $S\colon T; C \mid \Delta + \Delta_f \rhd_V \Delta'$ para cualquier V que no contenga a s. Se implementa de la forma:

$$
\begin{array}{lll}
\mathit{stepSolve}\ \mathit{chooseEv}\ s\ (\{h_{u_i} : \text{IsMG}\ \sigma_{u_i}\ s\},\ \{h_{l_j} : \text{IsMG}\ s\ \sigma_{l_j}\},\ \Delta_s)\ = & & \\
\quad \textbf{let} & & \\
\quad\quad (\vec{C}^u, \vec{C}^\ell, \Delta_f,\ \sigma) & = & \mathit{findSolution}\ (\vec{\sigma}^u, \vec{\sigma}^\ell, s) \\
\quad\quad (\vec{C}_2^u, \vec{C}_2^\ell) & = & \mathit{chooseEv}\ (\vec{C}^u, \vec{C}^\ell) \\
\quad\quad (T, C, \Delta') & = & \mathit{simplify}\ (\Delta_s, \Delta_f) \\
\quad \textbf{in} & & \\
\quad\quad (\ [\sigma/s],\ T,\ C \circ (\ \vec{h}^u \leftarrow \vec{C}_2^u\ \cdot\ \vec{h}^\ell \leftarrow \vec{C}_2^\ell\ ),\ \Delta_f,\ \Delta'\ ) &
\end{array}
$$

El argumento *chooseEv* es una función que recibe y devuelve un par de listas de conversiones; será utilizada en capítulo 4 para eliminar la evidencia del término. Por el momento usaremos la función *id*.

La función *stepSolve* presentada representa un paso del algoritmo de resolución. Como se justificó en 3.4.2, la resolución de los contextos de nuestro

lenguaje puede descomponerse en sucesivas resoluciones unitarias de forma tal que su composición sea una solución al contexto original.

En una implementación real, el algoritmo que busca una variable resoluble e itera *stepSolve* sería el utilizado hasta agotar las elecciones de variables. Asimismo este proceso iterativo puede ser optimizado "recordando" relaciones entre variables y predicados (por ejemplo, puede saberse que un predicado no podrá ser resuelto antes de resolver otro, y esta relación puede usarse para forzar este orden de resolución evitando chequeos en futuras iteraciones). Existen otras optimizaciones de este algoritmo, que no nos detendremos a mencionar. Como implementación prototípica, que es el propósito de esta presentación, los algoritmos dados son suficientes.

Lema 3.11. El algoritmo presentado es correcto respecto a la definición de la relación de resolución de restricciones. Expresamos este hecho en las siguientes proposiciones:

1. Si $\sigma = glb(\sigma_1, \sigma_2)$ entonces existen conversiones C_1, C_2 tales que $C_i : \sigma_i \geq \sigma$, y para cualquier σ' tal que $\sigma_i \geq \sigma'$ se cumplirá que $\sigma \geq \sigma'$.

2. *findSolution* encuentra efectivamente una solución posible para s para las cotas dadas. Es decir, las conversiones para las cotas superiores cumplen que $C_i^u : \sigma_i^u \geq \sigma$ y de forma análoga para las inferiores, $C_i^\ell : \sigma_i^\ell \geq (\Delta_f^\ell \mid \sigma)$.

3. Si $(S, T, C, \Delta_f, \Delta') = \mathit{stepSolve}\ id\ s\ \Delta$ y $s \notin V$ entonces

$$S\colon T; C \mid \Delta + \Delta_f \rhd_V \Delta'.$$

Dejaremos este lema sin demostración.

Es importante notar que el algoritmo *solve* no está definido para todo su dominio, entre ellas las que no tienen forma de separar el conjunto de predicados en tres partes: (i) las cotas superiores, (ii) las cotas inferiores, y (iii) los predicados que no contienen a la variable a simplificar. Tales contextos no son generados por la especialización de ningún término del lenguaje con que trabajamos, por lo que *solve* es capaz de resolver en la práctica cualquier contexto. Sin embargo, y como se verá en la sección 5.2, nuevas construcciones del lenguaje introducen nuevas formas de predicados, como por ejemplo, IsMG s ($\forall s' \ldots$ IsMG ($\ldots s \ldots$) $\ldots$) para los cuales determinar todas las cotas superiores o inferiores de una variable (y por lo tanto, su resolución) pueden depender de resolver la misma variable. Por esto decimos que el algoritmo dado es una *heurística*, que para nuestros propósitos es suficiente para resolver cualquier contexto.

Ejemplo 3.12. En este ejemplo se justifica la aparición de Δ' en la regla (SOLV), a la vez que se motiva la utilización de alguna forma de iteratividad para realizar

las resoluciones. También se ilustra en detalle la utilización de resolución y las transformaciones intermedias de los términos y predicados. Dada la complejidad de la especialización (comparado con otros ejemplos dados anteriormente, pero no contra especializaciones de cualquier programa real) nos extenderemos sobre esta especialización mostrando cada detalle del proceso.
Sea la siguiente especialización principal:

$$\begin{array}{l}\textbf{let}^D\ f = \textbf{poly}\ (\lambda^D x.\textbf{lift}\ (x +^S 2^S)) \\ \ \textbf{in let}^D\ g = \textbf{poly}\ (\lambda^D y.\textbf{spec}\ f\ @^D\ (y +^S 1^S))\ \textbf{in spec}\ g\ @^D\ 2^S \\ \quad \hookrightarrow \\ \textbf{let}\ f = h_f^u[\Lambda h_x.\Lambda h_x''.\lambda x.h_x''] \\ \ \textbf{in let}\ g = h_g^u[\Lambda h_y.\Lambda h_f^\ell.\Lambda h_y'.\lambda y.h_f^\ell[f]@\bullet]\ \textbf{in}\ h_g^\ell[g]@\bullet : \mathit{Int}\end{array}$$

en el contexto de la asignación de predicados

$$\begin{array}{l}h_g^u : \text{IsMG}\ (\forall t_y, t_y'.\text{IsInt}\ t_y, \text{IsMG}\ s_f\ (t_y' \to \mathit{Int}), t_y' := t_y + \hat{1} \Rightarrow t_y \to \mathit{Int})\ s_g, \\ h_g^\ell : \text{IsMG}\ s_g\ (\hat{2} \to \mathit{Int}), \\ h_f^u : \text{IsMG}\ (\forall t_x, t_x''.\text{IsInt}\ t_x, t_x'' := t_x + \hat{2} \Rightarrow t_x \to \mathit{Int})\ s_f,\end{array}$$

En la expresión aparecen dos funciones polivariantes, cada una con un único uso. Pero la aplicación de f esta dentro del cuerpo de g, por lo que no se conocerán lor argumentos a los que f puede aplicarse hasta que no se produzcan todas las formas necesarias de g (incidentalmente, si g no tuviera aplicaciones daría lugar a la tupla vacía, y el mismo código para f). Si durante la especialización, antes de descargar los predicados sobre el tipo de la expresión se decide resolver las variables s_g y s_f (lo cual es posible respecto a la restricción justificada en el ejemplo 3.5), se tiene que en principio sólo se puede hallar la solución para s_g - de forma que algoritmo continúa resolviendo esta variable.
Observando que tiene una única cota superior y una inferior, puede tomarse esta cota como valor final de s_g: se genera la sustitución que reemplaza s_g por el tipo de la cota superior. El valor de t_y se unifica entonces con $\hat{2}$.
Los predicados del esquema de tipos de la cota superior de s_g se agregan al contexto con variables frescas (aquí es donde se ve la asignación nombrada Δ'' en la regla (SOLV)), a saber:

$$\begin{array}{lll}(1) & h_1^* : \text{IsInt}\ \hat{2} & = (\text{IsInt}\ t_y)[\hat{2}/t_y] \\ (2) & h_2^* : \text{IsMG}\ s_f\ (t_y' \to \mathit{Int}) & \\ (3) & h_3^* : t_y' := \hat{2} + \hat{1} & = (t_y' := t_y + \hat{1})[\hat{2}/t_y]\end{array}$$

En estos predicados es posible eliminar el primero trivialmente, y el tercero resolviendo la suma y asignando el valor $\hat{3}$ a la variable t_y' (estos dos predicados se eliminan con una simplificación). El segundo predicado es el único que permanece en el contexto.
Volviendo al término, al realizar la resolución se utilizaron las siguientes expresiones de evidencia:

$$\begin{array}{ll} [] & \text{para } h_g^u \\ []((2))((h_2^*))((3)) & \text{para } h_g^\ell \end{array}$$

La especialización es en este punto, entonces (habiendo aplicado estas dos conversiones):

$$\begin{array}{l} \textbf{let } f = h_f^u[\Lambda h_x, h_x''.\lambda x.h_x''] \\ \textbf{ in let } g = \Lambda h_y, h_f^\ell, h_y'.\lambda y.h_f^\ell[f]@\bullet \textbf{ in } g((2))((h_2^*))((3))@\bullet : \mathit{Int} \end{array}$$

con la siguiente asignación de predicados en el contexto:

$$\begin{array}{l} h_f^u : \text{IsMG } (\forall t_x, t_x''.\text{IsInt } t_x, t_x'' := t_x + \hat{2} \Rightarrow t_x \rightarrow \mathit{Int})\ s_f, \\ h_2^* : \text{IsMG } s_f\ (\hat{3} \rightarrow \mathit{Int}) \end{array}$$

Finalmente, el trabajo que resta es más simple: sólo hay una variable de esquema para resolver, y ésta ocurre en dos predicados (como cotas superior e inferior). Siguiendo con la misma política que arriba, se elige el esquema que es única cota superior como valor solución para s_f. Se utiliza [] como evidencia de para h_f^u (ya que es un IsMGde dos esquemas de tipo iguales). Para simplificar el predicado restante,

$$h_2^* : \text{IsMG } (\forall t_x, t_x''.\text{IsInt } t_x, t_x'' := t_x + \hat{2} \Rightarrow t_x \rightarrow \mathit{Int})\ (\hat{3} \rightarrow \mathit{Int})$$

se unifica t_x con $\hat{3}$ y se resuelve t_x'' a $\hat{5}$; y por lo tanto se utiliza como $[]((3))((5))$ evidencia para h_2^*. La resolución termina, resultando en el término

$$\begin{array}{l} \textbf{let } f = \Lambda h_x, h_x''.\lambda x.h_x'' \\ \textbf{ in let } g = \Lambda h_y, h_f^\ell, h_y'.\lambda y.h_f^\ell[f]@\bullet \textbf{ in } g((2))(([])((3))((5)))((3))@\bullet : \mathit{Int} \end{array}$$

que es el resultado final de la resolución.
Es interesante notar que, conociendo los lugares exactos de las definiciones de las abstracciones f y g, se puede encontrar un término equivalente al obtenido por resolución, en donde las definiciones locales (**let**) de f y g ya contienen la toda la información estática:

$$\begin{array}{l} \textbf{let } f = \lambda x.5 \\ \textbf{ in let } g = \lambda y.f@\bullet \textbf{ in } g@\bullet : \mathit{Int} \end{array}$$

Para llegar a este término, realizando donde fuera necesario aplicaciones de evidencia (β_v), fue esencial saber que las aplicaciones presentes en el término eran las únicas posibles (que no podrían aparecer otras). En el capítulo 4 se utilizará un proceso similar, basado en resolución, para obtener un término sin abstracciones ni aplicación de evidencia.

El ejemplo muestra paso por paso una aplicación del algoritmo de resolución mostrado arriba, motivando:

1. La resolución iterada de las diferentes variables de esquema, al proporcionar la solución de una la información necesaria para resolver la siguiente.

2. El movimiento de predicados entre el tipo y la asignación del contexto durante la especialización (desde y hacia los **polys**).

3. El flujo de información de evidencia entre el término y su tipo, por medio de las variables de evidencia y conversiones.

A la vez se puede ver que el resultado final de una resolución puede ser innecesariamente 'complicado', repleto de expresiones de evidencia (abstracciones y aplicaciones) que son un subproducto de la representación interna del proceso de especialización y no necesariamente deseados por un usuario. En el siguiente capítulo se tratará una forma de eliminar estas expresiones.

3.6 Discusión

A lo largo de este capítulo se especificó la relación de resolución, y se desarrolló un algoritmo capaz de realizar resoluciones. En esta sección trataremos algunos aspectos contenidos en tarea de resolución discutiendo alternativas y las relaciones entre esta noción con los sistemas de especialización y la simplificación.

3.6.1 Resolución en especialización

La regla (SOLV) de la sección 3.3 permite realizar resolución durante un proceso de especialización. Los algoritmos de la sección 3.5 permiten resolver el valor de aquellas variables cuya información haya sido completamente provista por el contexto.

Dada esta versión algorítmica de resolución, podría agregarse la resolución a la versión algorítmica del especializador. Al igual que en el caso de la simplificación, es en el uso de **polys** y **specs** donde se justifica este proceso: cuando la información de predicados de se 'desempaqueta' de un tipo al aplicar la regla (W-SPEC), puede iterarse el algoritmo de resolución hasta que todas las variables posibles hayan sido resueltas.

A nivel implementación, se intentaría hacer el proceso de forma más eficiente con información adicional que indique de qué depende la información a proveer para cada variable de esquema, o cuáles variables de esquema cambian su condición de no resolubles o eligibles a ser resueltas cuando esta información cambie. Estos temas se cubren en la sección 5.1.

3.6.2 Resolución vs. Simplificación

Puede observarse que la definición de las nociones de simplificación y resolución son similares. Sin embargo, hay un aspecto en que son radicalmente diferentes, y tiene que ver con el tipo de decisiones que se toma en cada caso.

En las simplificaciones, dada la 'doble implicación' por la relación $\vdash$ de las asignaciones de predicados involucradas, sólo se deciden valores que son únicos (y deducibles de su contexto en que aparecen, con respecto a la relación de *entailment*): su reemplazo no aporta información, y puede volverse hacia atrás sin cambiar nada (salvo en casos en que se deciden valores 'incorrectamente', ver sección 2.7.3).

En la resolución de restricciones, en cambio, los valores que se reemplazan no son una consecuencia directa de los predicados que lo rodean; más aún, pueden existir diferentes soluciones para una misma variable (indistinguibles desde el punto de vista de la implicación) y durante su resolución se selecciona arbitrariamente una de ellas.

Es por esta razón que para realizar las resoluciones se debe esperar a obtener la mayor cantidad posible de información sobre las variables que se resuelven (como en el caso de las cotas superiores, como se vio en la sección 3.5), para no tomar decisiones incorrectas o incompletas (que pudieran, en el caso de un algoritmo, requerir *backtracking*).

3.7 Conclusiones

La resolución de restricciones es el corazón de este trabajo, y la tarea principal a realizarse luego del proceso de especialización. La simplificación vista en el capítulo 2 proporciona las herramientas básicas para especificar las condiciones que debe cumplir cualquier solución; y en este capítulo se describió la forma de caracterizar y posteriormente encontrar dichas soluciones. Y es precisamente esa tarea, la *descripción* de las soluciones, el aporte principal de este trabajo: la creación de un lenguaje en el cual poder aislar y tratar a la resolución de las restricciones creadas por la especialización de términos con polivarianza.

Si bien la resolución de restricciones puede ser vista como similar a la simplificación (hecho al que se agrega la similitud entre la construcción de ambas relaciones, desarrollado en los capítulos 2 y 3), hay una característica fundamental que los diferencia:

- La simplificación trabaja sólo sobre contextos (independientemente del término que esté siendo especializado). Remueve de los mismos los predicados redundantes, o los expresa de otras formas más simples. Para esto se basa o bien en reemplazos triviales o en implicaciones dadas por el resto

de los predicados contenidos en el mismo contexto. En cierta forma, como se dijo en la sección 2.7.3, estas transformaciones no eliminan los posibles modelos de un término o contexto, es decir, cualquiera los posibles términos que pueden tener un tipo calificado por los predicados de una asignación de predicados también pueden tener el tipo de la asignación de predicados resultante de realizar la simplificación, y viceversa (módulo algunas transformaciones dadas por las conversiones construidas durante la simplificación).

- La resolución de restricciones, en cambio, trabaja tanto con el tipo (los predicados) como con el término, transformándolo de acuerdo a las variables de esquema que se resuelven en su contexto. Para realizar este proceso *toma decisiones* sobre valores que no necesariamente únicos; decisiones que pueden eliminar posibles instancias de un término general. De esta forma la tarea principal de este capítulo fue determinar qué se considera una solución para una variable. Posteriormente, conociendo el conjunto de posibles soluciones (al menos habiéndolo definido por comprensión mediante la relación de resolución), se definió un algoritmo que elige, para el caso de este capítulo, la solución más general de las posibles (la máxima cota inferior de las presentes en los predicados IsMG).

Las decisiones que se toman durante la resolución representan las respuestas a los problemas que se encuentran durante la especialización y que no pueden resolverse localmente. De esta forma los predicados a resolver dependen esencialmente de las construcciones presentes en los términos que se especializan y el método por el cual son especializados. Por ejemplo, en nuestro trabajo la implementación del algoritmo de resolución (que es una relación definida en términos generales, abstrayéndose de los predicados que se utilicen) dado en este capítulo se centra en manipular 'cotas' dadas por los predicados IsMG, que son el subproducto de la especialización de expresiones anotadas como polivariantes. En lenguajes más complejos –y más reales– el algoritmo de resolución deberá manipular y resolver predicados provenientes de la especialización de recursión o tipos de datos algebraicos, por ejemplo (ver capítulo 5).

La utilidad de las soluciones elegidas, dado que en general habrá múltiples elecciones para hacer, dependerá en general del propósito de la resolución que se lleve a cabo y de las propiedades que se esperen del término especializado.

En este capítulo se describió un algoritmo simple que plasma las ideas descriptas, manipulando asignaciones de predicados para elegir las variables a ser resueltas, hallar una de sus soluciones posibles, reemplazar la variable por su solución y finalmente reducir la asignación de predicados mediante simplificación (convirtiendo el término en forma acorde). El algoritmo se describió con pseudocódigo, y su traducción a código funcional ejecutable es directa.

El algoritmo resuelve un subconjunto de las posibles asignaciones: aquellas provenientes de expresiones polivariantes y que además contienen toda la información del contexto para asegurarse que no pueden cambiar las relaciones que se conocen entre los tipos abstraídos por las variables de esquema. La resolución de todos los posibles términos no es un objetivo de nuestro algoritmo, y está fuera del alcance de este trabajo: el objetivo de este algoritmo es mostrar una forma de llevar a la práctica en forma consistente las relaciones descriptas formalmente. Además, se espera que el algoritmo sirva como base para el desarrollo de algoritmos más complejos, útiles para resolver los predicados que se presenten durante la especialización de lenguajes de uso general.

La especificación de resoluciones tal como se vio en este capítulo sirve no sólo para la implementación del algoritmo de resolución visto, sino para otro tipo de resoluciones según el objetivo que se tenga en mente. Con sólo elegir diferentes soluciones se puede cambiar drásticamente la forma de los términos obtenidos. La relación de resolución definida en este capítulo se utilizará con un fin diferente en el siguiente capítulo, de forma de resolver asignaciones de predicado dejando a los términos libres de expresiones de evidencia (abstracciones y aplicaciones). Este hecho mostrará la ventaja de haber definido de la forma más abstracta y general posible la relación de resolución de restricciones.

Capítulo 4

Eliminación de Evidencia

All translations are equal, but some are more equal than others.

in *Coherence for Qualified Types* (1993),
Mark P. Jones misquoting George Orwell's *Translation Farm*

Durante la simplificación y resolución de predicados (capítulos 2 y 3) se introducen conversiones en los términos residuales. Estas conversiones no pertenecen al lenguaje original de especialización de [Hughes, 1996]. En este capítulo describimos una forma de remover las pruebas de estos predicados: la etapa llamada *Eliminación de Evidencia* [Martínez López and Hughes, 2002].

La sección 4.1 describe en detalle qué tipo de expresiones se intenta eliminar de los términos residuales. A continuación, en la sección 4.2 se introducen las extensiones necesarias al lenguaje de evidencia tratado hasta este momento, y en 4.3 se las utiliza para eliminar evidencia usando el mismo algoritmo de resolución del capítulo 3. La siguiente sección (4.4) discute la forma en que el código se replica dentro del término, y cuáles son las limitaciones que tiene nuestra implementación del proceso de eliminación de evidencia. En la sección 4.5 se da una base formal a ciertas elecciones tomadas en el capítulo anterior, justificadas por los resultados de este capítulo. Finalmente, en 4.6 se dan las conclusiones sobre la realización de este proceso.

4.1 Evidencia visible

En el trabajo de *Especialización Principal de Tipos* [Martínez López and Hughes, 2002], se definió una versión del proceso de especialización original de [Hughes, 1996] dotado de la propiedad de *principalidad*. En los capítulos 2 y 3 de este trabajo se describieron tratamientos a aplicar a los predicados resultantes de una especialización (principal): primero para lograr 'mejores' versiones de los mismos y luego como la toma de decisiones diferidas durante el proceso de especialización y que completan la misma. Sin embargo, los términos resultantes de una resolución exitosa no son los mismos, en general, que aquellos provenientes de la especialización original - de hecho, pertenecen a un lenguaje más amplio que incluye también expresiones de evidencia.

Con el proceso de *Eliminación de Evidencia* se intenta eliminar todas las construcciones (abstracciones, aplicaciones) de evidencia posibles en un término obtenido por especialización principal, para llevar a los términos al mismo lenguaje que el especializador de [Hughes, 1996]. Esto completará, inicialmente (aún se debe extender el lenguaje con características como recursión) el camino de [Martínez López and Hughes, 2002] desde los términos del lenguaje fuente a aquellos del lenguaje residual, pero con las ventajas de este enfoque mencionadas en el capítulo 1 (principalidad es la propiedad más destacada).

Aún si no se tuviera como objetivo el obtener términos similares o en el mismo lenguaje que [Hughes, 1996], los términos obtenidos por [Martínez López and Hughes, 2002] tienen cierta desventaja respecto a los originales, que debería ser reparada como postprocesamiento: la repetida abstracción y aplicación de evidencia produce un *overhead* de cómputo que es optimizable en tiempo de compilación. La eliminación de evidencia será el proceso que permita resolver este problema.

Ejemplo 4.1. Para mencionar una última característica, utilizaremos la siguiente especialización. Dado el término

$$\begin{array}{l}\mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x)\ \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 13^S, \mathbf{spec}\ f\ @^D\ 17^S) \\ : (Int^D, Int^D)\end{array}$$

su especialización principal es

$$\begin{array}{l}\Lambda h^u, h_1^\ell, h_2^\ell.\mathbf{let}\ f = h^u[\Lambda h_x.\lambda x.h_x]\ \mathbf{in}\ (h_1^\ell[f]@\bullet, h_2^\ell[f]@\bullet) \\ \quad : \forall s.\,\mathrm{IsMG}\ (\forall t_x.\mathrm{IsInt}\ t_x \Rightarrow t_x \rightarrow Int)\ s, \\ \qquad\quad \mathrm{IsMG}\ s\ (\hat{13} \rightarrow Int), \\ \qquad\quad \mathrm{IsMG}\ s\ (\hat{17} \rightarrow Int) \\ \qquad\quad \Rightarrow (Int, Int)\end{array}$$

Aplicando resolución obtendríamos

$$\mathbf{let}\ f = \Lambda h_x.\lambda x.h_x\ \mathbf{in}\ (f(\hat{(}13))@\bullet, f(\hat{(}17))@\bullet) : (Int, Int)$$

En el ejemplo la función f está anotada como polivariante. Como se dijo en el capítulo 1, esto es una indicación de que se desean *distintas* especializaciones de la misma función, de acuerdo a los distintos argumentos que pueda recibir. En el cuerpo del **let** hay dos especializaciones de la misma, para los parámetros 13 y 17. Sin embargo, la salida de una especialización principal (con el método de resolución visto en el capítulo 3) crea *una sola* versión de la función f, a la que se le aplican los dos argumentos en forma de evidencia – de forma que no es tan evidente la especialización que debería haber computado estáticamente estas versiones. Uno de los posibles términos residuales a los que se podría llevar el término para obtener realmente dos especializaciones distintas de f sería:

$$\textbf{let } f = (\lambda x.13, \lambda x.17)^S \textbf{ in } (\pi^S_{1,2}\, f@\bullet, \pi^S_{2,2}\, f@\bullet) : (Int, Int) \qquad (4.1)$$

En donde la función f fue reemplazada por un par de funciones, cada una de las instancias obtenidas por polivarianza. (Cabe aclarar que el término obtenido puede refinarse aún más, aunque este procesamiento no corresponde a la etapa de eliminación de evidencia; ver sección 4.4.4.)

De un proceso de eliminación de evidencia esperaremos que permita obtener términos como el anterior (4.1), partiendo de términos como el del ejemplo 4.1. Además será deseable que sea óptimo en algun sentido: por ejemplo, que no cree elementos que no se usen en las tuplas estáticas, o que no repita elementos para varias especializaciones con los mismos argumentos.

Veremos que el marco de trabajo definido en los capítulos anteriores es suficientemente expresivo como para capturar tal transformación sin trabajo adicional. Durante el resto del capítulo nos concentraremos en formalizar el proceso.

4.2 Extensiones al lenguaje de evidencia

Continuando con la idea de mantener el código de las funciones en el mismo lugar donde fueron definidas, y esencialmente producir la misma salida que el especializador de [Hughes, 1996], extenderemos el lenguaje de evidencia con dos nuevas construcciones para el manejo de tuplas. Estas construcciones, utilizadas junto con las conversiones usadas en capítulos anteriores y combinadas con la noción de resolución permitirán lograr este efecto.

El lenguaje residual y el de conversiones se extienden con las siguientes construcciones:

$$\begin{aligned} e' &= \ldots \mid (e', \ldots, e')^S \mid \pi^S_{n,n}\, e' \\ C &= \ldots \mid (C, \ldots, C)^c \mid \pi^c_{n,n}\, [] \end{aligned}$$

La primer construcción agregada es el constructor de tuplas estáticas de cualquier aridad, siendo entonces $(e'_1, \ldots, e'_n)^S$ un término residual para cualquier n natural. La construcción $\pi^S_{i,n}\, e'$ proyecta la componente i-ésima de una tupla e' de n componentes, para cualquier i y n naturales con $i \leq n$.

La razón de que estas nuevas construcciones contengan anotaciones ^{S} es que la eliminación de evidencia no es verdaderamente la etapa final de la especialización, si se compara el trabajo con el de [Hughes, 1996]: posteriormente se realizarán las etapas llamadas *arity raising* y *void erasure* (que puede ser vista como un caso particular de la primera), en donde se removerán tanto como sea posible las tuplas estáticas (y por lo tanto los usos de $\bullet$). Esta etapa se comentará en la sección 4.4.4.

El caso especial de la tupla vacía, $()^S$, también es un término válido. Se lo puede ver como el constructor de un tipo sin información, similar a $\bullet$ (de hecho, como se verá más adelante, puede ser conveniente no distinguirlos).

Las nuevas conversiones son análogas a las expresiones agregadas. Proporcionando un término a $\pi^c_{i,n}\,[]$, se transforma en la proyección de tuplas estáticas correspondiente. Para el constructor de tuplas $(C, \ldots, C)^c$, se replica el término para aplicarlo a cada una de las conversiones que contiene. La semántica de las nuevas conversiones es entonces:

$$\begin{aligned} \pi^c_{i,n}[e'] &= \pi^S_{i,n}\, e' \\ (C_1, \ldots, C_n)^c\,[e'] &= (C_1[e'], \ldots, C_n[e'])^S \end{aligned}$$

Estas construcciones se anotan con ^{c} para diferenciarlas de sus paralelos en el lenguaje residual – esta etiqueta es sólo una anotación y no representa un elemento de algún conjunto.

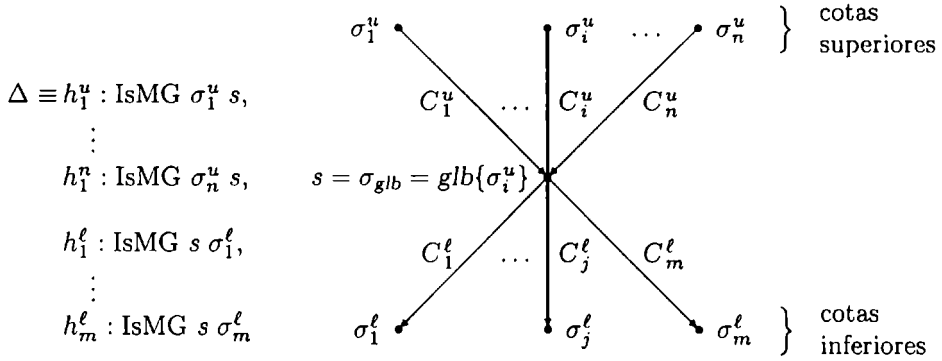
Notar que el lenguaje fuente se mantiene sin cambios: las construcciones agregadas serán otro residuo posible de las mismas construcciones (luego de aplicar resolución), como por ejemplo **poly**.

4.3 Eliminación de evidencia mediante resolución

Como se explicó en 4.1, se eliminará la abstracción y aplicación de evidencia replicando el código de las expresiones polivariantes, y luego reduciendo las expresiones de evidencia (principalmente por β_v). Para tal efecto se usarán las conversiones definidas en la sección 4.2: el constructor de tupla estáticas será usado en lugar de las expresiones polivariantes, y las correspondientes proyecciones en sus puntos de uso, es decir, en sus **specs**.

4.3.1 Soluciones

Para eliminar evidencia utilizaremos una forma diferente de construir conversiones a partir de las obtenidas de las cotas superiores e inferiores por las funciones utilizadas en el algoritmo de resolución del capítulo 3. Describiremos en detalle este proceso, mostrando la similitud que tiene con el utilizado para resolución de restricciones.



La conversión entre cualesquiera σ_i^u y σ_j^ℓ se obtiene construyendo $C_j^\ell \circ C_i^u$

Figura 4.1: Construcción de conversiones para eliminar evidencia

Las conversiones para eliminar evidencia se construyen examinando el conjunto de cotas superiores e inferiores de la variable de esquema a ser resuelta. Explicaremos en detalle este proceso, utilizando como guía la representación gráfica de la figura 4.1:

- Sea s la variable de esquema seleccionada para resolución. Esta variable debe cumplir las condiciones requeridas por el algoritmo de resolución (sección 3.5) pueda resolverse (es decir, sólo debe ocurrir como cota en predicados IsMG).

- Sean σ_i^u y σ_j^ℓ los conjuntos de cotas superiores e inferiores respectivamente (los esquemas contenidos en los predicados IsMG en los que s es una de las cotas).

Se cumplen dos condiciones: todas las n cotas superiores son más generales (de acuerdo a la relación $\geq$) que cualquiera de las cotas inferiores, y además existe una cota inferior común a todas ellas (su máxima cota inferior, glb). Este valor, $\sigma_{glb} = glb(\{\sigma_i^u\}_{i=1..n})$, que es además más general que cualquier de las cotas inferiores σ_j^ℓ, es elegido por el algoritmo de resolución como valor para s.

Recordemos las conversiones utilizadas en el algoritmo del capítulo 3. La construcción de la evidencia para la cualquier variable h^u que abstrae una cota superior se eligió en este algoritmo como la conversión C_i^u que conviernte una expresión de tipo σ_i^u al tipo σ_{glb}. Similarmente, se utilizó C_j^ℓ como evidencia para h_j^ℓ, convirtiendo términos de tipo σ_{glb} al tipo requerido σ_j^ℓ. El motivo de esta elección fue que en cualquier punto de uso (**spec**) de una expresion polivariante de tipo s se necesita convertir al tipo σ_j^ℓ una expresion de tipo σ_i^u; y al aplicar las conversiones correspondientes, como una composición (primero la

conversión C_i^u y a su resultado C_j^ℓ), se logra convertir el término entre los tipos mencionados.

En este punto la solución con eliminación de evidencia difiere de la elegida en capítulo 3: en lugar de convertir cada una de las expresiones que forman las cotas superiores al tipo σ_{glb} en donde están definidas, y aplicar la conversión que las transforme a los tipos requeridos en los puntos de uso (**specs**), se instanciarán las cotas superiores *de todas las formas necesarias* en su definición (**poly**) y se seleccionará la instancia adecuada en cada uso. La representación de estas posibilidades se hará con las tuplas estáticas introducidas en 4.2, y la selección se hará con la proyección adecuada.

De esta forma, en lugar de utilizar C_i^u ó C_j^ℓ como evidencia para las cotas superiores e inferiores i y j respectivamente, se utilizarán

$$\begin{array}{rl} (C_1^\ell \circ C_i^u, \ldots, C_m^\ell \circ C_i^u)^c & \text{para IsMG } \sigma_i^u\ s \\ \pi_{j,m}^c\ [] & \text{para IsMG } s\ \sigma_j^\ell \end{array}$$

La justificación es que, en cada 'punto de uso' (cota inferior j-ésima) donde se utilice a la i-ésima 'definición' (cota superior), la composición de la conversiones $(C_j^\ell \circ C_i^u)^c$ y $\pi_{j,m}^c\ []$ es

$$\pi_{j,m}^c\ [] \circ (C_1^\ell \circ C_i^u, \ldots, C_m^\ell \circ C_i^u)^c = C_j^\ell \circ C_i^u$$

que es equivalente (es la misma, bajo la igualdad de conversiones) a la conversión utilizada en la resolución del capítulo 3.

4.3.2 Implementación

El enfoque de resolución del capítulo 3 fue definido en forma suficientemente general como para dejar cierta libertad a la hora de elegir la evidencia con la que transformar al término. Esta evidencia se construye utilizando las cotas calculadas en el mismo algoritmo de resolución. Algunas funciones son inherentes a la heurística de resolución utilizada (*glb*, *makeMoreGeneral*, *findSolution*), mientras que sólo en la última de ellas se aplica concretamente la evidencia construida (las conversiones, inicialmente obtenidas a partir de los precicados IsMG) para la construcción del término con la variable resuelta. Esta función está a su vez parametrizada con una función que manipula un par de listas de conversiones para devolver otro par del mismo tipo, pero posiblemente diferente. En la resolución presentada en el capítulo 3 fue suficiente utilizar la función identidad, ya que no se deseaba ninguna forma especial en las construcciones construidas.

Para eliminar la evidencia utilizando la idea descripta en esta sección, se implementa una función *eliminateEv* que simplemente compondrá las conversiones construidas por la función *findSolution* de la sección 3.5:

$$
\begin{array}{l}
\mathit{eliminateEv}\ (\overrightarrow{C}^{u}_{i=1..n},\ \overrightarrow{C}^{\ell}_{j=1..m}) = \\
\quad \textbf{let} \\
\quad\quad \mathit{ubs} = \{\ (C^{\ell}_{1} \circ C^{u}_{i}, \ldots, C^{\ell}_{m} \circ C^{u}_{i})^{c}\ \}_{i=1..n} \\
\quad\quad \mathit{lbs} = \{\ \pi^{c}_{j,m}\ []\ \}_{j=1..m} \\
\quad \textbf{in} \\
\quad (\mathit{ubs},\ \mathit{lbs})
\end{array}
$$

De esta forma, la función

$$\mathit{stepSolve}\ \mathit{eliminateEv}$$

implementa el proceso de eliminación de evidencia descripto.

Ejemplo 4.2. Ilustraremos con la siguiente especialización principal el uso del algoritmo descripto y mostraremos que la elección de una solución hecha por el procedimiento descripto en el capítulo 3 no pierde información.

$$
\begin{array}{l}
\textbf{let}^{D}\ f = \textbf{poly}\ (\lambda^{D} y.\textbf{lift}\ y)\ \textbf{in spec}\ f\ @^{D}\ 7^{S} : \mathit{Int} \\
\quad \hookrightarrow \\
\textbf{let}\ f = h^{u}[\Lambda h_y.\lambda y.h_y]\ \textbf{in}\ h^{\ell}[f]@\bullet : \mathit{Int}
\end{array}
$$

en el contexto

$$
\begin{array}{l}
h^{u} : \mathrm{IsMG}\ (\forall t.\mathrm{IsInt}\ t \Rightarrow t \rightarrow \mathit{Int})\ s, \\
h^{\ell} : \mathrm{IsMG}\ s\ (\hat{7} \rightarrow \mathit{Int})
\end{array}
$$

Para esos predicados hay al menos dos soluciones: darle a s el valor de la cota superior o el de la inferior. En el primer caso, se utilizarían [] como evidencia para h^{u} y [](7)) para h^{ℓ}, que por resolución (del capítulo 3) resultaría en el término:

$$\textbf{let}\ f = \Lambda h_y.\lambda y.h_y\ \textbf{in}\ f((7))@\bullet : \mathit{Int}$$

que contiene la abstracción y aplicación de evidencia que se desea eliminar. En el segundo caso, eligiendo la cota inferior como solución se construye [](7)) como evidencia para h^{u} y [] para h^{ℓ}, obteniendo, si se resolviera directamente:

$$\textbf{let}\ f = \lambda y.7\ \textbf{in}\ f@\bullet : \mathit{Int}$$

Sin embargo, el resultado de eliminar evidencia para cualquiera de las soluciones es el mismo:

$$\textbf{let}\ f = (\lambda y.7)^{S}\ \textbf{in}\ \pi^{S}_{1,1}\ f@\bullet : \mathit{Int}$$

En el ejemplo vimos un caso en que el resultado luego de eliminar evidencia fue el mismo sin importar cuál haya sido el valor elegido para la variable de esquema reemplazada por el proceso de resolución. En la sección 4.5 se dará una justificación de esta propiedad, mostrando que la elección de soluciones descripta en la sección 3.5 es adecuada.

4.4 Limitaciones y postprocesamiento

En el capítulo 1 se describió a **poly** como una anotación que indica que se deben crear diferentes versiones de una misma expresión (polivarianza). La creación de esas versiones de una misma función requiere, en el caso general, la replicación de las partes dinámicas de la expresión polivariante. Sin embargo, las reglas que implementan el proceso de especialización no replican código fuente, así como tampoco lo hacen las conversiones vistas hasta el capítulo 3. Es recién durante la eliminación de evidencia donde se realiza esta replicación, más específicamente con la conversión $(\ldots)^c$ descripta en este capítulo. Esta conversión aplica una copia del término que convierte a cada una de las conversiones que contiene.

En las siguientes secciones se discuten algunas decisiones acerca de la manera de replicar código de nuestro algoritmo de eliminación de evidencia.

4.4.1 Identificación de cotas inferiores

Uno de los objetivos de la eliminación de evidencia es producir efectivamente las diferentes versiones de las expresiones polivariantes. Esto se logra replicando el código de las funciones marcadas con anotaciones **poly** dentro de tuplas estáticas, cada una especializada para el argumento que se le aplica.

La decisión más importante al momento de replicar ese código es la cantidad de copias a crear. La decisión se basa, obviamente, en los argumentos a los que la función es aplicada. Pero no es claro cómo calcular la cantidad y naturaleza precisa de las componentes de la nueva tupla: desde el caso extremo –y no deseable– de un elemento por cada argumento aplicado, a no producir código redundante en la forma final del término (luego de aplicadas las conversiones), optimización que podría llegar a ser demasiado costosa.

Hay diferentes criterios que inicialmente podrían ser utilizados para la decisión, basados en la identificación de algunos parámetros durante la eliminación de evidencia:

- Por cada *uso*: Una conversión distinta por cada ocurrencia de un **spec**. Dado que puede haber distintos **specs** con los mismos argumentos (ejemplo 4.3-1), se generarían normalmente elementos iguales en la misma tupla, una redundancia indeseable.
- Por el *tipo* de los argumentos: Sin importar el valor de los argumentos a los que se especializa una función polivariante, las conversiones necesarias para convertir el término a uno de tipo especializado sólo dependen del tipo de los argumentos. Por ende, un elemento en la tupla por cada tipo residual a que se especializa es un enfoque viable; aunque también puede generar elementos iguales en la misma tupla.

- Por las *conversiones (sintácticamente)*: Cada predicado IsMG de las cotas inferiores se 'prueba' durante resolución con una conversión; y hay casos en donde dos predicados diferentes (diferentes cotas inferiores) pueden probarse con la misma conversión (se encuentran conversiones sintácticamente iguales para ambos predicados), y por lo tanto resultar en el mismo término especializado. En caso de encontrar que dos predicados se prueban con conversiones exactamente iguales (que no es demasiado costoso) se puede optar por identificarlos en una única componente de la tupla resultante. Este enfoque sería probablemente adecuado para manejar polimorfismo, en donde varias utilizaciones de una expresión polimórfica sobre diferentes tipos generarían comúnmente la misma conversión.

- Por las *conversiones (extensionalmente)*: Similarmente al caso anterior, hay ocasiones en donde dos conversiones que prueban diferentes predicados pueden ser iguales. En este caso nos referimos a igualdad semántica (la definida mediante extensionalidad en el capítulo 1). Sin embargo, el chequeo de esta igualdad de conversiones es poco factible computacionalmente por lo que este caso es difícilmente implementable.

- Por el *código* resultante: Se crearía una componente en la tupla sólo si el resultado de aplicar la composición de conversiones del algoritmo de eliminación de evidencia resultaría en diferente código residual. Pero para este caso se deben tener en cuenta, para identificar dos cotas inferiores, el resultado de convertir cada una de las cotas superiores; tampoco es viable computacionalmente.

De las alternativas presentadas, la más razonable para implementarse es la segunda, en donde las cotas inferiores se identifican por su tipo.

Esta alternativa tiene la ventaja de eliminar algunas redundancias comunes (argumentos iguales tendrán siempre el mismo tipo), y además es muy conveniente para el algoritmo. Si antes de eliminar evidencia se realiza la simplificación de los predicados del contexto, una de las simplificaciones (dada por (SimEntl)) elimina predicados idénticos (como pasaría con el caso de los generados por cotas del mismo tipo) identificando sus variables de evidencia, por lo que al proveer cualquier conversión para probar al predicado que se mantiene se estarían utilizando las mismas conversiones para el término.

Este método no es óptimo, en el sentido de que puede quedar en última instancia código duplicado en una tupla (sección 4.4.3), pero consideramos que conserva un buen balance entre resultados y costo computacional. Esta versión es la que se utiliza en la implementación de eliminación en el prototipo.

4.4.2 Evidencia que no se elimina

El objetivo de esta fase es eliminar la mayor cantidad de evidencia, llegando de ser posible a dejar términos sin evidencia. Las chances de lograr este objetivo dependen fuertemente de la forma en que se haya abstraído (y modificado) la evidencia antes de llegar a la etapa de eliminación: mediante simplificación o resolución.

Ya se mencionó que la eliminación reemplaza en cierta forma a la resolución (se implementa con una versión modificada de su algoritmo), pero no se trataron las consecuencias de la forma en que se simplifique. En la sección 2.4 se mencionó que no es necesaria una regla que elimine cotas inferiores de una misma variable de esquema. Veamos por qué se tomó esta decisión.

Si alguna construcción **spec** construyera cotas inferiores de una variable que sean comparables (por $\geq$) entre sí, entonces podría eliminarse una de ellas utilizando como evidencia la composición de evidencia entre la conversión de una cota a otra y la variable aún abstraída por el otro predicado. Por ejemplo, si $C : \sigma_2 \geq \sigma_1$, entonces

$$h_1 : \text{IsMG } s\ \sigma_1, h_2 : \text{IsMG } s\ \sigma_2$$

se podría simplificar a

$$h_2 : \text{IsMG } s\ \sigma_2$$

convirtiendo el término con la conversión $h_1 \leftarrow C \circ h_2$. De esta manera se eliminó una variable de evidencia en las cotas inferiores, lo que llevaría posiblemente a una componente menos en la tupla estática a crearse.

Pero también se introdujo una conversión en el término, y esta conversión puede contener aplicaciones o abstracciones de evidencia que no podrían ser luego eliminadas.

Por esta razón, para que no queden expresiones que no puedan ser removidas por el proceso de eliminación de evidencia, elegimos no realizar ese tipo de simplificaciones.

4.4.3 Limitaciones y redundancia

El algoritmo de eliminación de evidencia descripto en este capítulo es capaz de resolver razonablemente el problema planteado pero de una forma que no siempre es la óptima.

Si bien la eliminación de evidencia por resolución –combinada con una adecuada simplificación– obtiene la menor cantidad posible de elementos en las tuplas cuando toda la información de las cotas inferiores es conocida, si la información pudiese variar (por ejemplo, por haber variables de tipo no resueltas) la solución podría no ser óptima: se podrían llegar a generar elementos repetidos en las tuplas estáticas. Esta situación se ilustra en el siguiente ejemplo.

Ejemplo 4.3. Las siguientes especializaciones del mismo término ilustran la situación en que pueden repetirse elementos en las tuplas luego de la eliminación de evidencia. Sea el término original:

$$\begin{array}{l}\lambda^D y.\mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x)\\ \quad \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 13^S, \mathbf{spec}\ f\ @^D\ 13^S, \mathbf{spec}\ f\ @^D\ y)\\ : Int^S \to^D (Int^D, Int^D, Int^D)\end{array}$$

Se pueden obtener las siguiente especializaciones:

1. Especialización de [Hughes, 1996] (si se especializa aislado; el resultado podría ser diferente según el contexto):

   $$\begin{array}{l}\lambda y.\,\mathbf{let}\ f = (\lambda x.13)^S\\ \quad \mathbf{in}\ (\pi^S_{1,2}\, f@\bullet, \pi^S_{1,2}\, f@\bullet, \pi^S_{1,2}\, f@y) : \hat{13} \to (Int, Int, Int)\end{array}$$

2. Especialización principal (aplicando simplificación):

   $$\begin{array}{l}\Lambda h^u, h^\ell_{13}, h_y, h^\ell_y.\lambda y.\mathbf{let}\ f = h^u[\Lambda h_x.\lambda x.h_x]\\ \qquad\qquad \mathbf{in}\ (h^\ell_{13}[f]@\bullet, h^\ell_{13}[f]@\bullet, h_y[f]@\bullet)\\ : \forall t_y, s.\mathrm{IsMG}\ (\forall t_x.\mathrm{IsInt}\ t_x \Rightarrow t_x \to Int)\ s,\\ \quad \mathrm{IsMG}\ s\ (\hat{13} \to Int),\\ \quad \mathrm{IsInt}\ t_y,\\ \quad \mathrm{IsMG}\ s\ (t_y \to Int) \Rightarrow t_y \to (Int, Int, Int)\end{array}$$

3. Especialización principal + resolución:

   $$\begin{array}{l}\Lambda h_y.\lambda y.\mathbf{let}\ f = \Lambda h_x.\lambda x.h_x\\ \qquad \mathbf{in}\ (f((13))@\bullet, f((13))@\bullet, f((h_y))@\bullet)\\ : \forall t.\mathrm{IsInt}\ t \Rightarrow t \to (Int, Int, Int)\end{array}$$

4. Resolución con eliminación de evidencia:

   $$\begin{array}{l}\Lambda h_y.\lambda y.\mathbf{let}\ f = (\lambda x.13, \lambda x.h_y)^S\\ \qquad \mathbf{in}\ (\pi^S_{1,2}\, f@\bullet, \pi^S_{1,2}\, f@\bullet, \pi^S_{2,2}\, f@\bullet)\\ : \forall t.\mathrm{IsInt}\ t_y \Rightarrow t_y \to (Int, Int, Int)\end{array}$$

En el término fuente dos especializaciones de f se aplican a un argumento del mismo tipo residual ($\hat{13}$), y otra especialización a un valor desconocido. Al especializar, se generan dos predicados IsMG idénticos, uno de los cuales es eliminado por simplificación por lo que se utiliza la misma evidencia (abstraída por h^ℓ_{13}), por lo que no se generarán componentes diferentes para estos dos **specs**.

Sin embargo, el caso de **spec** $f\ @^D\ y$ es más problemático. Tal como está presentado el término, no se conoce si el valor de y será 13 ó un número diferente. En el primer caso, debería utilizarse la misma componente de la tupla; en cualquier otro caso una nueva componente debería crearse.

En el caso de la especialización de [Hughes, 1996] (4.3-1), la función f especializa a una tupla 'abierta' con un único elemento. Si al continuar especializando

el contexto (el proceso de especialización es monolítico, no modular) se aplica un valor diferente de 13, la tupla se expande con el elemento correspondiente y se continúa especializando. Al finalizar la especialización, se finaliza la tupla 'cerrando' la información que falte: en este caso, se decidió que el valor de y sería 13, en ausencia de más información. Esta decisión, algorítmica, no es la que deseamos en nuestra eliminación de evidencia.

La especialización principal (4.3-2) es la forma más general y en ella se expresan todas las soluciones posibles, pero aún no se ha llegado a la forma final que se desea en el término: no se ha decidido el valor de la variable de esquema s.

En el término con resolución (4.3-3) obtenido como se describe en el capítulo 3, es claro que no hay código replicado innecesariamente, pero existe evidencia que no puede ser eliminada (y además tal evidencia está en sí repetida en cada **spec**), y el término no tiene la forma que se desea, como se explicó en este capítulo.

Finalmente, en (4.3-4) aparece el término que se obtuvo por eliminación de evidencia. Al haber dos predicados como cotas inferiores (luego de simplificación, los idénticos colapsan a uno solo), se crean dos elementos en la tupla. Si posteriormente se aplicara este término con un argumento de valor 13^S, la segunda componente de la tupla sería igual a la primera, pero no podría ser eliminada.

Una forma de resolver este último problema sería esperar a que se tenga toda la información para resolver con eliminación de evidencia; pero esto contradice nuestros objetivos de modularidad -principalidad-. Un enfoque más adecuado sería formalizar una noción de tuplas extensibles, de forma que en el ejemplo se crearía una sola componente (la estrictamente necesaria) y se dejaría la tupla 'abierta' para poder extenderse si nueva información del contexto lo requiriera. Se continuará con estas ideas en la sección 5.2.

4.4.4 Postprocesamiento

Para describir un postprocesamiento que puede hacerse luego de eliminar evidencia volveremos al ejemplo 4.1, en el que se introdujo la necesidad de remover las expresiones de evidencia resultantes de probar los predicados introducidos durante la especialización.

En el ejemplo se muestra la especialización principal de

$$\begin{array}{l}\mathbf{let}^D\ f = \mathbf{poly}\ (\lambda^D x.\mathbf{lift}\ x)\ \mathbf{in}\ (\mathbf{spec}\ f\ @^D\ 13^S, \mathbf{spec}\ f\ @^D\ 17^S) \\ : (Int^D, Int^D)\end{array}$$

y el resultado de eliminar evidencia sobre el mismo, resultando en el término etiquetado (4.1):

$$\mathbf{let}\ f = (\lambda x.13, \lambda x.17)^S\ \mathbf{in}\ (\pi^S_{1,2}\, f@\bullet, \pi^S_{2,2}\, f@\bullet) : (Int, Int)$$

En esta especialización, la función f, anotada como polivariante, se reemplazó por una tupla estática (un par) de funciones. No se declararon localmente dos funciones por separado, como en realidad se deseaba con la anotación original. Sin embargo esta última salida se puede lograr mediante un proceso llamado *arity raising*, obteniendo el término:

$$\textbf{let } f_1 = \lambda x.13; f_2 = \lambda x.17 \textbf{ in } (f_1@\bullet, f_2@\bullet) : (Int, Int) \tag{4.2}$$

El proceso de *arity raising*, que no es tratado en este trabajo, es capaz de lograr esa transformación e inclusive eliminar estáticamente las aplicaciones de los *voids* '•', cambiando cada una de las f_i del ejemplo de una abstracción a un valor constante. En este proceso también se identifican las tuplas vacías $()^S$ con • (como se sugirió en la sección 4.2), logrando eliminar también estas construcciones.

El trabajo de *arity raising* que se requiere para trabajar con polimorfismo (que es una de las aplicaciones de nuestro trabajo) no figura en la literatura tradicional [Hannan and Hicks, 1998]; las bases de este trabajo están siendo estudiadas en un trabajo en curso [de la Canal, 2003].

Esta etapa de postprocesamiento es disjunta con la resolución de restricciones o la eliminación de evidencia, y se estudia mediante técnicas diferentes. Dado que para su aplicación es suficiente construir términos como el 4.1, en nuestro proceso de eliminación de evidencia nos basta con obtener estas construcciones: pasamos por alto el *overhead* dado por construcción y destrucción de tuplas estáticas o aplicación de elementos • sin información, que serán eliminados luego en *arity raising*.

4.5 Coherencia de soluciones

En el ejemplo 4.2 se vio que aún tomando diferentes soluciones para una misma variable se puede llegar al mismo término utilizando eliminación de evidencia. De cumplirse, la independencia de la elección de cuál de las soluciones se selecciona durante la resolución de restricciones respecto al término final luego de eliminación de evidencia sería una propiedad importante: permitiría justificar la decisión (arbitraria) de seleccionar los valores descriptos como soluciones en el capítulo 3. En esta sección justificaremos nuestra hipótesis de que esta propiedad se cumple.

Existe un paralelo de esta propiedad en la teoría de tipos calificados [Jones, 1994a], nombrada con el término *coherencia*. En ese trabajo esta noción (originalmente acuñada en [Breazu-Tannen *et al.*, 1989]) se utiliza para describir que 'el significado de un término no depende de la forma en que se realiza el chequeo de tipos' [Jones, 1993].

En nuestro caso lo utilizaremos para decir que 'la forma del término residual no depende de la forma en que se realiza la especialización'. En las nociones introducidas en este trabajo hay un punto muy marcado donde se tomaron decisiones que, como se dijo, pueden parecer arbitrarias: al tener varias soluciones posibles para una variable de esquema, se eligió la máxima cota inferior. La coherencia de soluciones, en el marco de este trabajo, sería una justificación de esta elección.

Siguiendo las líneas de Jones en su trabajo, nuestra propuesta para demostrar la coherencia de las soluciones obtenidas se basa en asegurar, en primer lugar, que la evidencia posible para un predicado dado es única. Este hecho es cierto en el caso en que las pruebas son valores del lenguaje (enteros, booleanos) pero no en el caso general en presencia de conversiones. Sin embargo, creemos que para las construcciones que producen la introducción de predicados que se resuelven en este trabajo (que son un conjunto limitado de los predicados posibles) esta propiedad de unicidad se cumple. Enunciamos este hecho como conjetura.

Conjetura 4.4 (Igualdad de conversiones). La conversión entre dos esquemas de tipos restringidos dados es única. Es decir, si $C_1, C_2 : (\Delta \mid \sigma) \geq (\Delta' \mid \sigma')$ y σ no es ambiguo entonces $C_1 = C_2$.

Si bien tenemos borradores de una demostración de la conjetura, a la fecha no se ha finalizado la demostración. En el caso del trabajo de tipos calificados de Jones, la propiedad se cumple trivialmente en los ejemplos dados. Pero en nuestro caso la definición mutuamente recursiva de la relación de implicación y las conversiones como expresiones de evidencia para predicados IsMG (figura 1.3 y definición 1.5) hace que no sea cierta en todos los casos.

Utilizando esa conjetura, se puede probar el teorema principal que asegura la unicidad de los términos obtenibles luego de eliminación de evidencia.

Teorema 4.5 (Coherencia de soluciones – basado en conjetura 4.4). El término obtenido luego de resolver eliminando evidencia sobre una variable de esquema es siempre el mismo, independientemente de la solución que se elija para la misma.

Con este teorema, y suponiendo cierta la conjetura anterior, podríamos afirmar que las elecciones tomadas por el algoritmo de resolución son adecuadas: en particular, cualquier solución que se elija para una variable de esquema resultaría en el mismo término luego de eliminación de evidencia (como sucede en el ejemplo 4.2).

4.6 Conclusiones

En este capítulo se implementó el proceso de eliminación de evidencia, que permite remover de los términos residuales de especializaciones las abstracciones y

aplicaciones de evidencia. Estas expresiones son particulares de la representación e implementación del sistema de especialización de [Martínez López and Hughes, 2002], pero no es deseable que sean parte de los términos finales ya que no son consecuencia de las expresiones del término original ni las anotaciones del mismo.

El proceso de eliminación de evidencia se implementó como un caso particular de resolución de restricciones, eligiendo de una forma conveniente las conversiones aplicadas por la resolución. Este hecho, además de permitir eliminar evidencia, muestra que la resolución de restricciones especificada en el capítulo 3 es tan general como para además poder restringir la forma de los términos que se desean obtener con ella. La eliminación de evidencia se obtiene entonces como un valor agregado casi sin esfuerzo adicional luego de especificar las relaciones de simplificación y resolución de restricciones.

La evidencia removida, además, no forma parte del lenguaje original de especialización de Hughes [Hughes, 1996], de forma que con este trabajo se relaciona el trabajo de especialización principal con el trabajo original, llevando los términos especializados al mismo lenguaje (siempre que es posible) al lenguaje de este último. La obtención de los mismos términos que los obtenidos por el especializador de Hughes puede lograrse aplicando una etapa adicional, *arity raising*, a los términos obtenidos luego de eliminar evidencia; nuestros términos quedan listos para esta etapa de postprocesamiento (que se estudia en forma separada).

Además se planteó una justificación (basada en una conjetura que aún no demostramos) de por qué cualquier elección de solución durante la resolución de restricciones, por ejemplo aquellas elegidas en el capítulo 3, es buena y lleva finalmente al mismo término residual, lo que llamamos coherencia de soluciones.

Capítulo 5

Implementación y extensiones

"Cheshire Puss", comenzó ella algo tímidamente... "Me dirías por favor, qué camino debería tomar para ir desde aquí?"
"Eso depende mucho de dónde quiera usted ir" dijo el gato.
"Poco me preocupa dónde ir", contestó Alicia.
"Entonces, nada importa qué camino tome", replicó el gato.

Alice's Adventures in Wonderland (1865)
Lewis Carroll

En este capítulo se describe en primer lugar el trabajo realizado sobre la implementación prototipo del especializador, en la cual se experimentaron algoritmos y se implementaron las relaciones descriptas en capítulos anteriores.

En segundo lugar, se describen las líneas principales de trabajo futuro, que consisten esencialmente de toda extensión al lenguaje que se especializa en este trabajo que tienda a llevarlo a ser un lenguaje de programación "real": por ejemplo, recursión, especialización de funciones estáticas y tipos algebraicos.

5.1 Prototipo

En esta sección describiremos el trabajo de implementación de un prototipo del algoritmo de especialización principal con simplificación y resolución de restricciones, como se describió en los capítulos 2 y 3. Describiremos en primer lugar el prototipo de [Martínez López and Hughes, 2002] usado como base, para luego describir los módulos implementados, algunos aspectos de implementación, y nuèstra contribución a este trabajo.

5.1.1 Lenguaje

La implementación del prototipo del especializador se realizó en el lenguaje funcional *Haskell* [Peyton Jones and Hughes (editors), 1999]. Este prototipo se ha usado durante la implementación, sobre su intérprete *Hugs* en sus versiones para Unix y Win32; y para obtener eficiencia el compilador `ghc`.

Trabajar sobre el paradigma funcional permite aprovechar al máximo la capacidad de trabajar sobre estructuras de datos de la misma forma en que se piensa sobre ellas, lo que es especialmente adecuado para la representación de las construcciones de nuestro lenguaje y el proceso de especialización.

Además, la transparencia referencial y la programación mediante ecuaciones facilita el definir formalmente pre- y post-condiciones de cada proceso (tarea que se dificulta en un lenguaje imperativo o en el paradigma de orientación a objetos, donde la semántica de cada construcción depende de un estado global y no sólo de sus parámetros). De esta forma, se pudieron asegurar algunas propiedades esenciales sobre las funciones utilizadas.

Respecto al estilo de programación, el especializador está basado en una mónada [Wadler, 1993] de estado que acarrea la información referente al estado de la especialización (variables, sustituciones, etc.). El estilo monádico permite abstraerse de la información de estado acarreada y efectos producidos, para centrarse en las tareas que cada función realiza.

5.1.2 Trabajo previo

Se utilizó como base el prototipo implementado por [Martínez López and Hughes, 2002], que contiene estructuras de datos para representar términos y tipos fuente y residuales, esquemas, predicados, asignaciones, y demás elementos de los lenguajes manipulados durante la especialización. La funcionalidad implementada previamente contenía al núcleo del especializador principal, que realizaba especializaciones de acuerdo al algoritmo W especificado mediante las reglas de la figura 1.6. Esta implementación realizaba ciertas simplificaciones simples, como forma de facilitar la visualización de los resultados. Sin embargo, se debían estudiar simplificaciones más elaboradas y formalizar todo el conjunto para demostrar la consistencia de este procedimiento.

5.1.3 Detalles de implementación

Como se vio en los capítulos 2 y 3, la resolución de restricciones se formula en base a simplificación (con el agregado, si se quiere una versión algorítmica, de decidir las variables a ser simplificadas y hallar sus cotas superiores e inferiores para encontrar el valor la solución). La simplificación, a su vez, se basa en la definición de implicación definida por la relación $\Vdash$.

Por este motivo, parte del desarrollo de estos módulos incluye la implementación mediante funciones de esta relación, para determinar cuándo un predicado es implicado por otros, o qué predicados son necesarios para implicar a uno dado.

5.1.4 Trabajo realizado y futuro

Lejos de ser una implementación final, el prototipo descripto implementa las relaciones y algoritmos descriptos en los capítulos anteriores como forma de experimentar sobre los mismos.

Actualmente, el prototipo tiene la capacidad de:

- *parsing* de un lenguaje fuente y su representación en un arbol sintático adecuado, junto con la representación del lenguaje residual y los sistemas de tipos de ambos lenguajes, y las funciones de impresión de todos ellos;

- *obtención y chequeo de tipos* de ambos lenguajes, además del chequeo de la relación de tipos fuente-residual (relación SR, figura 1.4) necesario para especialización;

- *especialización principal* de términos del lenguaje fuente (implementando el algoritmo de la figura 1.6), obteniendo programas en el lenguaje residual con expresiones de tipos calificados (con predicados) y evidencia;

- *simplificación* como etapa intermedia a la especialización, necesaria para la eficiencia en el tratamiento de los predicados agregados por cada regla, implementando las reglas del sistema descripto en el capítulo 2 y algunas extensiones;

- *resolución de restricciones*, con la heurística descripta por el algoritmo de la sección 3.5, permitiendo la decisión de los valores de la evidencia de los predicados introducidos durante la especialización;

- *eliminación de evidencia* (sección 4.3), como etapa final (y opcional), como forma de obtener programas sin expresiones de evidencia y, en los casos en que se hace posible, programas idénticos al sistema original de especialización de [Hughes, 1996].

En cada etapa se definen además las estructuras de datos necesarias para representar sustituciones, tipos calificados, generalización e instanciación de los mismos, abstracción y aplicación de evidencia, etc.

Como parte de este trabajo se implementaron las funciones de simplificación y resolución de restricciones, con las estructuras de datos para representar la información que manipulan (sustituciones, variables libres, *pool* de variables frescas, estado, etc.), y una instancia adicional de la resolución de restricciones que toma la forma de eliminación de evidencia (como se describe en el capítulo 4).

Como trabajo futuro, se pueden mencionar varias tareas. La primera y más inmediata es la optimización de los algoritmos ya implementados: la eficiencia de los algoritmos del prototipo no es la deseable, y en muchos casos especiales (en donde sabemos elegir los términos que atacan los casos más débiles en este respecto) la tarea completa desde especialización hasta eliminación de evidencia tarda desde varios minutos a inclusive algunas horas. Esta tarea incluye la de *profiling* de ejecuciones de los algoritmos para buscar puntos que puedan agilizarse por ejemplo recordando valores sin recalcularlos (cosa que en la actualidad no se hace a sabiendas), búsqueda de puntos en donde el agregado de sentencias que hagan las funciones estrictas (eliminando la evaluación *lazy*) para obtener mayor eficiencia espacial, o inclusive la reimplementación en un lenguaje más de rápida ejecución, sea en el paradigma funcional con anotaciones especiales (e.g. Clean), efectos laterales (e.g. ML) o en otro paradigma.

Otra tarea posible es la extensión de las simplificaciones y resoluciones para dar mayor poder al algoritmo hoy implementado como heurística. Este trabajo será un subproducto de la extensión de las nociones teóricas y reglas adicionadas a las relaciones de los capítulos 2 y 3, y permitirá resolver más términos especializados, en el mejor caso cualquier término de los posibles.

Finalmente, otra tarea que tiene más que ver con extensiones de este sistema es la de agregar construcciones al lenguaje fuente (y como consecuencia al residual) para especializar términos más cercanos a los utilizados diariamente al programar, achicando la brecha entre el prototipo como juguete de laboratorio y algoritmo de especialización como herramienta de uso diaria en algún dominio.

5.1.5 Conclusiones de esta implementación

Un aporte que tuvo la fase de implementación del prototipo sobre el resto del trabajo fue la constante 'puesta en práctica' de los conceptos que se desarrollaban en teoría. Se lo utilizó innumerables veces para experimentar con la especialización de términos para obtener asignaciones de formas deseadas para especializar.

La ejecución del prototipo fue un elemento clave para visualizar el resultado de las simplificaciones y el algoritmo de resolución. Con estos resultados prácticos, se podían refinar o corregir las definiciones y algoritmos especificados

en forma teórica. En algunos casos se encontraron inconvenientes en las definiciones, especificadas en un mundo imaginario, que eran incompatibles con el resto del trabajo; este hecho se probaba al tratar de incorporar las mismas a las partes anteriores del prototipo. Esto llevaba a un ciclo de volver sobre las especificaciones y reformularlas, para luego corregir las implementaciones y observar nuevamente los resultados de su ejecución.

Un caso destacable de la valiosa información provista durante la implementación es la inclusión de la variable Δ'' en la definición de resolución (definición 3.2): en nuestra especificación original, no existía el concepto de "predicados forzados" y estos surgieron como necesidad al ver que los algoritmos de implicación no lograban hacer que un esquema de tipos fuera más general que el otro, debido a los predicados que contenía y se debían agregar al contexto para que la implicación se cumpliera.

Con estas apreciaciones podemos ver el valor de llevar a la práctica un trabajo teórico de estas características.

5.2 Extensiones e ideas a explorar

En el capítulo 3 se logró formalizar noción de Resolución de Restricciones basándose en la de Simplificación, y dando a la vez un algoritmo muy básico que logra resolver las ocurrencias de algunos predicados provenientes de la etapa de especialización.

Sin embargo, el dominio del algoritmo de resolución, como ha sido presentado, no abarca al conjunto completo de predicados sino sólo aquellos en que las variables a resolver son o bien la cota inferior o la cota superior de un predicado IsMG, y además *no aparece* como parte de la otra cota en el mismo predicado ni ocurre (propiamente) dentro de ninguno de ellos. Una variable que ocurra en las dos cotas de un predicado,

$$\text{IsMG}\ (\ldots s \ldots)\ (\ldots s \ldots)$$

no puede ser resuelta con nuestra heurística.

Este tipo de predicados es introducido normalmente al tratar con recursión (como se verá en el ejemplo de la siguiente sección). Otros predicados pueden provenir de enriquecer al lenguaje (actualmente podado para conformar sólo un lenguaje representativo y sobre el que estudiar los problemas que trata este trabajo) con nuevas construcciones.

Entre las construcciones posibles a agregar se encuentran todas aquellas presentes el lenguajes de programación de uso general, ya que este trabajo apunta a largo plazo a ser adaptado para usarse en un ámbito fuera del laboratorio. Mencionaremos algunas en las siguientes secciones.

Las extensiones que se proponen entonces a este respecto es estudiar posibles algoritmos o heurísticas de resolución, aptas para resolver situaciones como las presentadas a continuación.

5.2.1 Recursión

Es probablemente la primer extensión para pensar. En cualquier lenguaje con un mínimo de poder expresivo se querrá utilizar recursión como forma de definición de funciones para resolver problemas de tamaños variables.

Es de notar que en nuestro lenguaje no se incluyeron adrede constructores para representar recursión. Su inclusión en la forma usual (por ejemplo con un **let** recursivo) acarrea la implementación de reglas de especialización análogas a las de tipado de un sistema Hindley-Milner [Milner, 1978] para recursión. En el trabajo de [Martínez López and Hughes, 2002] las reglas de recursión se implementan con un constructor **fix**, que retorna el mínimo punto fijo de una función, en la forma usual de agregar recursión a λ-cálculo.

Describiremos informalmente el tratamiento de estas construcciones con un ejemplo. Se aclara que el ejemplo es necesariamente complejo; los problemas que se desean ilustrar aparecen en ejemplos que fácilmente contienen decenas de predicados.

Ejemplo 5.1. Sea el término que representa a la función recursiva `power`, que eleva a la potencia n su argumento x. La implementación contiene la construcción **fix** además de las anotaciones necesarias para realizar estáticamente la eliminación del parámetro actual que reemplace a n:

$$\begin{aligned}\textbf{fix}\ (\lambda f \rightarrow \textbf{poly}\ (\lambda n\ x \rightarrow\ & \textbf{if}^S\ n ==^S 1^S \\ & \textbf{then}\ x \\ & \textbf{else}\ x * \textbf{spec}\ f\ @^D\ (n -^S 1^S)\ @^D\ x))\end{aligned}$$

Su especialización principal es:

$$\begin{aligned}\Lambda h_{16}.\textbf{fix}\ (\lambda f.h_{16}[\Lambda h_4\ h_6\ h_7\ h_5\ h_8\ h_9\ h_{10}\ h_{11}\ h_{12}.\lambda n\ x. \\ \textbf{if}_v\ h_4\ \textbf{then}\ x\ \textbf{else}\ x * h_7[f]@ \bullet @x \\ ])\end{aligned}$$

del siguiente tipo residual:

$$
\begin{array}{l}
\forall s.h_2 : \text{IsMG } (\forall t.h' : \text{IsInt } t \Rightarrow t \rightarrow \mathit{Int} \rightarrow \mathit{Int})\ s, \\
\quad h_{16} : \boxed{\text{IsMG}}\ (\forall t\, t'\, t_2\, t_3\, t_4\, t_5\, s'. \\
\qquad\qquad h_3 : \text{IsInt } t, \\
\qquad\qquad h_4 : (t' := t == \hat{1}), \\
\qquad\qquad h_6 : !t'\ ?\ \text{IsInt } t_2, \\
\qquad\qquad h_7 : !t'\ ?\ \text{IsMG } s'\ (t_2 \rightarrow \mathit{Int} \rightarrow \mathit{Int}), \\
\qquad\qquad \boxed{h_5 : !t'\ ?\ \mathbf{poly}\ s \sim \mathbf{poly}\ s'}, \\
\qquad\qquad h_8 : !t'\ ?\ t_3 := t - \hat{1}, \\
\qquad\qquad h_9 : !t'\ ?\ (t_2 \rightarrow \mathit{Int} \rightarrow \mathit{Int}) \sim (t_3 \rightarrow t_4), \\
\qquad\qquad h_{10} : !t'\ ?\ t_4 \sim (\mathit{Int} \rightarrow t_5), \\
\qquad\qquad h_{11} : !t'\ ?\ \mathit{Int} \sim \mathit{Int}, \\
\qquad\qquad h_{12} : !t'\ ?\ (t_5 \sim \mathit{Int}) \\
\qquad\qquad \Rightarrow t \rightarrow \mathit{Int} \rightarrow \mathit{Int})\boxed{s} \\
\Rightarrow \mathbf{poly}\ s
\end{array}
$$

Sin entrar en demasiados detalles (este análisis es un ejercicio interesante) ilustraremos brevemente la forma del tipo residual y la razón por la que nuestra heurística no puede resolverlo.
El tipo residual contiene dos predicados. El primero corresponde a la cota superior (etiquetado por h_2), dada por la construcción **poly**.
El segundo, que a su vez contiene un tipo calificado por varios predicados, corresponde al **spec** de la función recursiva f. Este predicado es un IsMG cuya cota inferior es s, pero dentro de su cota superior está la misma variable (resaltados con recuadros). La razón de esta ocurrencia es que la solución requiere hacer *unfolding* de este predicado, unificando con s' en cada paso, hasta que todos los predicados se resuelvan, que sucederá cuando t' sea verdadera, es decir, t valga $\hat{1}$. Este proceso equivale a hallar el punto fijo del tipo.
Notar además que, como se dijo en la seccion 2.6, hay predicados que pueden ser simplificados (por ejemplo el etiquetado con h_{11}), pero al estar bajo una guarda ($!t'$?...) esta simplificación no se ha realizado.

Este ejemplo es uno de los más simples que ilustran la ocurrencia de una misma varible a ambos lados de un IsMG, utilizando recursión. El tratamiento general de predicados con este patrón requiere la realización de pasos de *unfolding* del tipo, llevando a realizar estáticamente (durante la especialización de estas construcciones anotadas como estáticas) y pudiendo acarrear la no-terminación al especializador.

5.2.2 Funciones estáticas

La anotación de funciones como estáticas hace que abstracciones y aplicaciones sean eliminadas por el especializador.

En [Martínez López, 2003], esta extensión se logra agregando abstracciones y aplicaciones con la correspondiente anotación ^{S} en el lenguaje fuente (y el constructor de tipos funcionales $\rightarrow^S$).

Es en el lenguaje residual donde la situación es más complicada. El sistema de tipos residual se extiende con una construcción para representar clausuras:

$$\tau' ::= \ldots \mid \mathbf{clos}(e' : \sigma')$$

y los predicados con un constructor para relacionar clausuras:

$$\delta ::= \ldots \mid \text{IsFunS } \tau' \ \tau'$$

(utilizando en formal similar a IsMG en el caso de polivarianza), que será usado para posponer la decisión de los valores de un tipo funcional hasta que se provea toda la información del contexto. Además se agrega un nuevo tipo de expresión residual, la aplicación de evidencia

$$e' ::= \ldots \mid v@_v e'$$

que es eliminada como parte del postprocesamiento de la eliminación de evidencia.

El cuerpo de la función estática se especializa, y la especialización principal se almacena como una clausura dentro de un predicado IsFunS. Lluego, al especializar la aplicación estática de una expresión, se utiliza la evidencia utilizada para probar el predicado IsFunS proveyendo las especializaciones de las variables libres del cuerpo de la función (que se dejan sin especializar en la regla de la abstracción) y la especialización del argumento de la función.

En lugar de mostrar las reglas de especialización de estas expresiones daremos dos ejemplos que ilustran su uso.

Ejemplo 5.2. En este primer ejemplo se muestra como el cuerpo de la función estática es movido dentro de la cota superior de un IsFunS, y sus variables libres quedan como residuo de su especialización.

$$\begin{array}{l}
\vdash_{\mathrm{P}} \mathbf{let}^D \ x = (5^S, 6^D)^D \\
\quad \mathbf{in}\ \lambda^S y.\mathbf{lift}\ (\mathbf{fst}^D\ x +^S y) +^D \mathbf{snd}^D\ x \\
\quad\quad : \mathit{Int}^S \rightarrow^S \mathit{Int}^D \\
\quad \hookrightarrow \Lambda h_f.\mathbf{let}\ x = (\bullet, 6)\ \mathbf{in}\ x \\
\quad\quad : \forall t.\text{IsFunS } (\mathbf{clos}(\Lambda h_y, h_r.\lambda f'.\lambda y.h_r + \mathbf{snd}\ f' \\
\quad\quad\quad\quad\quad\quad : \forall t_y, t_r.\text{IsInt } t_y, t_r := \hat{5} + t_y \Rightarrow t_y \rightarrow \mathit{Int})) \ t \Rightarrow t
\end{array}$$

Ejemplo 5.3. En este segundo ejemplo, la misma función estática del ejemplo anterior se aplica estáticamente a un argumento conocido, 2^S, y el tipo pasa a contener otro predicado IsFunS correspondiente a esta aplicación.

$$
\begin{array}{l}
\vdash_{\mathsf{P}} \mathbf{let}^D\ f = \mathbf{let}^D\ x = (5^S, 6^D)^D \\
\qquad\qquad \mathbf{in}\ \lambda^S y.\mathbf{lift}\ (\mathbf{fst}^D\ x +^S y) +^D \mathbf{snd}^D\ x \\
\quad \mathbf{in}\ f\ @^S\ 2^S \\
\quad : Int^D \\
\hookrightarrow \Lambda h_f, h_a.\mathbf{let}\ f = (\mathbf{let}\ x = (\bullet, 6)\ \mathbf{in}\ x) \\
\qquad\qquad \mathbf{in}\ h_a @_v f @_v () \\
\quad : \forall t, t'.\mathrm{IsFunS}\ (\mathbf{clos}(\Lambda h_y, h_r.\lambda f'.\lambda y.h_r + \mathbf{snd}\ f' \\
\qquad\qquad\qquad\qquad : \forall t_y, t_r.\mathrm{IsInt}\ t_y, t_r := \hat{5} + t_y \Rightarrow t_y \rightarrow Int))\ t, \\
\qquad \mathrm{IsFunS}\ t\ \mathbf{clos}(t' : \hat{2} \rightarrow Int) \Rightarrow Int
\end{array}
$$

Durante la resolución de estos predicados, se calcularía el término $\lambda f'.\lambda y.7 + \mathbf{snd}\ f'$ como evidencia del predicado IsFunS t $\mathbf{clos}(t' : \hat{2} \rightarrow Int)$; esta evidencia se asignaría a h_a, y la reducción de evidencia (la regla $(@_v)$para $@_v$) produciría el resultado final: $7 + \mathbf{snd}\ f$.

Los ejemplos ilustran la manera en que en [Martínez López, 2003] se elige especializar funciones estáticas, e informalmente una idea de la forma de realizar resolución de restricciones en ellas. Esta es similar al tratamiento de polivarianza (creando predicados con cotas superiores e inferiores hasta que se obtenga toda la información del contexto), pero con complicaciones adicionales. Las funciones estáticas son entonces otra extensión posible a nuestro trabajo.

5.2.3 Tipos algebraicos

Los tipos alebraicos (sumas disjuntas de tipos) son otra característica común en lenguajes funcionales y que no es tratada en este trabajo. En el trabajo de [Martínez López, 2003] también se tratan los tipos algebraicos, con constructores nombrados y construcciones de destrucción por *pattern matching* **case**.

Su tratamiento acarrea en ese trabajo el agregado de nuevas reglas de especialización, para constructores y la instrucción **case**, dos nuevos tipos de predicados (nombrados IsConstrOf e IsCase), y reglas de reducción para construcciones **case** de evidencia.

Los constructores (y sus argumentos) pueden ser anotadas además como estáticas o dinámicas, por lo que el tratamiento contiene toda la complejidad de las funciones estáticas descriptas en la sección anterior.

Se remite al lector a [Martínez López, 2003] para la lectura completa de esta extensión, que queda para nosotros como trabajo futuro.

$$
\begin{array}{l}
\vdash_{\mathrm{P}} \mathbf{let}^{D}\ f = \mathbf{fix}^{S}\ (\lambda^{S} f.\lambda^{S} x.Cons^{S}\ 1^{D}\ (f\ @^{S}\ x)) \\
\quad \mathbf{in\ case}^{S}\ f\ @^{S}\ ()^{S}\ \mathbf{of} \\
\qquad Cons\ x\ xs\ \rightarrow\ x \\
\quad : Int^{D} \\
\quad \hookrightarrow \Lambda h_U, h_L, h_{y_s}.\mathbf{let}\ f = \bullet \\
\qquad\qquad\qquad\qquad \mathbf{in\ fst}^{S}\ (h_L @_v f @_v ()^{S}) \\
\qquad : \forall t, t_{y_s}, t_e.\mathrm{IsFixS}\ \mathrm{cl}_{x_s}\ t \\
\qquad\qquad\qquad \mathrm{IsFunS}\ t \\
\qquad\qquad\qquad\quad \mathbf{clos}(t_e : ()^{S} \rightarrow Cons\ Int\ t_{y_s}), \\
\qquad\qquad\qquad \mathrm{IsConstrOf}\ (List^{S}\ Int^{D})\ t_{y_s}, \\
\qquad\qquad\qquad \Rightarrow Int \\
\text{where} \\
\mathrm{cl}_{x_s} = \\
\quad \mathbf{clos}(\Lambda h_f, h_r.\lambda f_s.\lambda f.(f)^{S} \\
\qquad : \forall t_f, t_r, t_e.\mathrm{IsFunS}\ \mathrm{cl}_f(t_e)\ t_f, \\
\qquad\qquad\qquad \mathrm{IsFunS}\ \mathrm{cl}_r(t_f)\ t_r \\
\qquad\qquad\qquad \Rightarrow t_f \rightarrow t_r) \\
\mathrm{cl}_f(t_e) = \\
\quad \mathbf{clos}(t_e : \forall t_{x_s}.\mathrm{IsConstrOf}\ (List^{S}\ Int^{D})\ t_{x_s} \\
\qquad\qquad\qquad \Rightarrow ()^{S} \rightarrow t_{x_s}) \\
\mathrm{cl}_r(t_f) = \\
\quad \mathbf{clos}(\Lambda h'_{y_s}, h'_L.\lambda^{S} f'_s.\lambda^{S} x.(1, h'_L @_v \pi_{1,1}\ f'_s @_v x)^{S} \\
\qquad : \forall t'_{y_s}, t'_e.\mathrm{IsConstrOf}\ (List^{S}\ Int^{D})\ t'_{y_s} \\
\qquad\qquad\qquad \mathrm{IsFunS}\ t_f\ \mathbf{clos}(t'_e : ()^{S} \rightarrow t'_{y_s}) \\
\qquad\qquad\qquad \Rightarrow ()^{S} \rightarrow Cons\ Int\ t'_{y_s})
\end{array}
$$

Figura 5.1: Ejemplo con terminación bajo lazy evaluation

5.2.4 Arity Raising y Lazy Evaluation

Otro tema de posible trabajo futuro es el estudio de la especialización de funciones estáticas bajo *lazy evaluation*. Ilustraremos la motivación de este trabajo con un ejemplo tomado de 1.3 - este ejemplo utiliza construcciones que no estudiamos formalmente en este trabajo y sólo se introdujeron con ejemplos en secciones anteriores.

En la figura 5.1 se muestra la especialización de una función que dado un argumento cualquiera (que no utiliza) construye una lista infinita de valores iguales a 1. Esta función no termina de evaluarse si se quiere obtener la lista completa, pero sí pueden obtenerse tantos términos como se desee de la misma bajo *lazy evaluation*. En el ejemplo, el constructor creado en la aplicación de la

función a un valor nulo se destruye para obtener la cabeza y la cola de la lista, y se devuelve el primero de estos elementos como valor final. De esta forma no se evalua el resto de la lista.

En enfoques tradicionales, especialización de funciones estáticas implica la evaluación de las expresiones especializadas a forma normal para luego utilizar los valores en la expresión residual. Inclusive en nuestro enfoque de resolución, si se quisieran resolver los predicados construidos por esta recursión se produciría un *unfolding* infinito de predicados iguales.

Sin embargo, una alternativa en nuestro mismo marco de trabajo es tratar a la resolución con más cuidado de forma de tener en cuenta la estrategia de evaluación (por ejemplo *lazy evaluation*).

Una idea prometedora es realizar resolución de restricciones *bajo demanda* del proceso de *arity raising* (ver sección 4.4.4): resolver variables y predicados cada vez que se realmente se necesite su valor en la forma final del término (y no porque los predicados simplemente existan en el contexto).

Este enfoque, sugerido en [Martínez López and Hughes, 2002] y ahora más cercana de ser estudiada por el marco que aporta este trabajo, es una de las ideas más prometedoras para estudio futuro.

Capítulo 6

Conclusiones y trabajo futuro

Cuando Ogión despertó, con el frío del alba, Ged había desaparecido. Sólo había dejado, a la manera de los hechiceros, trazado en runas de plata sobre la piedra del hogar, un mensaje que se desvaneció a medida que Ogión lo leía: "maestro, salgo de caza".

Un Mago de Terramar (1968)
Úrsula K. Le Guin

En este trabajo se estudió la especialización de tipos, creada por John Hughes [Hughes, 1996] como un enfoque alternativo y novedoso para evitar ciertos límites heredados de los enfoques de especialización de programas tradicionales. Utilizando la base mencionada, en [Martínez López and Hughes, 2002] se continúa este trabajo creando un lenguaje para capturar la noción de principalidad, es decir, la capacidad de encontrar una especialización 'más general' que cualquier otra, a partir de la cual se puedan lograr las demás mediante instanciación. En ese trabajo se utiliza un sistema de predicados para representar condiciones encontradas durante el proceso de especializiación, que los términos y sus tipos deben cumplir.

Se puede resumir el aporte principal de este trabajo como la formalización de las nociones de simplificación y resolución de restricciones, completando el sistema de especialización de [Martínez López and Hughes, 2002].

La simplificación, estudiada en el capítulo 2, se definió en términos de la relación de implicación de predicados. Así, se especificó en forma abstracta determinando las condiciones mínimas que una relación debe cumplir para ser considerada una simplificación. Se incorporó esta noción al sistema P de especialización y a su versión algorítmica, junto con una prueba de consistencia de esta regla. Se implementó además una relación de simplificación con las propiedades estudiadas, apta para simplificar los predicados provenientes de la especialización del lenguaje en estudio. Adicionalmente se estudiaron diferentes aspectos a ser tenidos en cuenta para el diseño de cualquier relación de simplificación, entre otros relacionando a este trabajo con el tratamiento de tipos calificados original de Jones [Jones, 1994a].

La resolución de restricciones (tema del capítulo capítulo 3) no estaba presente en el trabajo original [Hughes, 1996] sino que las decisiones se tomaban cuando era posible en el mismo algoritmo de especialización, llevando a realizar retrocesos (backtracking) o forzar a soluciones en casos no únicos. En [Martínez López and Hughes, 2002] se separa la tarea en una primer fase de especialización del término, obteniendo una *especialización principal* que contiene las condiciones que cualquier instancia cumplirá expresadas en términos de predicados. En este trabajo se especificó formalmente lo que es una resolución de restricciones en una especialización, definiéndola en base a las relaciones de simplificación antes estudiadas. Además se incorporó esta noción al sistema de reglas que especifica la especialización, y al algoritmo que la implementa, probando la consistencia de tal adición. Finalmente, se implementó como ejemplo de uso de la especificación un algoritmo simple de resolución capaz de resolver las restricciones dejadas para un subconjunto del lenguaje sobre el que se define la especialización; sentando las bases para algoritmos de resolución de casos más complejos (como recursión o tipos de datos algebraicos).

El lenguaje residual de especialización principal no es el mismo (ya que es más general) que aquél del el trabajo de John Hughes. En este trabajo se implementó ademas una forma de realizar eliminación de evidencia (capítulo 4), como una forma de postprocesamiento que lleva (cuando es posible) los términos especializados con el sistema de especialización principal al mismo lenguaje del sistema original, completando el circuito que relaciona el trabajo de especialización principal con el original. Esta eliminación de evidencia fue implemetnada como una forma particular de resolución de restricciones, lo que muestra que esa relación fue especificada con suficiente generalidad como para capturar diferentes tipos de resoluciones.

Se implementaron además las relaciones anteriores en el prototipo de especializador de Martínez López, dando una forma de visualizar las relaciones descriptas y experimentar con ellas, obteniendo además una implementación de un especializador basada en funciones y relaciones estudiadas y especificadas formalmente, lo que da una base más sólida a los algoritmos.

El aporte principal de este trabajo es el estudio en un marco formal de las relaciones antedichas. La separación de estas etapas y su especificación proporciona un lenguaje en el que poder estudiar los problemas aún no resueltos de la especialización de tipos.

Los problemas sin resolver siguen siendo muchos, pero como se dijo la intención de este trabajo era crear un marco de trabajo en el que estudiarlos. Estos problemas quedan como trabajo futuro, y constan de toda extensión al lenguaje tratado en este trabajo que sirva para aproximarlo a un lenguaje cotidiando, como aquellos estudiados en el capítulo 5: recursión, funciones estáticas, datatypes, etc.

Consideramos que conceptos como los aquí estudiados pueden aplicarse en el futuro en áreas como el desarrollo de compiladores o aplicaciones sobre lenguajes de dominio específico (DSLs), y la base teórica puede ser el mejor pilar para su aplicación práctica, como se ha demostrado en experiencias pasadas.

Apéndice A

Demostraciones

How do you know I'm mad? -said Alice.
You must be -said the cat- or you wouldn't have come here.

Alice's Adventures in Wonderland
Lewis Carroll

En este capítulo se presentan las demostraciones de todos los teoremas enunciados en capítulos anteriores. Si bien su lectura es recomendable para la comprensión en profundidad de algunos detalles de este trabajo, se aclara al lector que su lectura es opcional y puede ser más difícil que el resto del trabajo.

Lema 2.1. Las conversiones $h{\leftarrow}h$ son neutras para $\cdot$ (notar que la conversion [] es un caso particular de estas conversiones): O sea, para cualquier conversión C y variables de evidencia h se cumple que $h{\leftarrow}h \cdot C = C = C \cdot h{\leftarrow}h$.

Demostración. Para todo e',

$$(h{\leftarrow}h \cdot C)(\!(e')\!) = (\Lambda h.C)(\!(h)\!)[e']$$

que es igual a $(\Lambda h.C[e'])(\!(h)\!)$. A su vez, este término es equivalente bajo β_v a

$$C[e'][h/h] = C[e']$$

(el caso $C = C \cdot h{\leftarrow}h$ es aún más simple). □

Lema 2.5. Sea la simplificación $T; C \mid \Delta \trianglerighteq \Delta'$. Si S y T son compatibles sobre Δ, es decir $S \leftrightarrow_\Delta T$, entonces T es también una simplificación de $S\Delta$, o sea $T; C \mid S\Delta \trianglerighteq S\Delta'$.

Demostración. Si $T; C \mid \Delta \trianglerighteq \Delta'$, entonces $h' : \Delta' \Vdash v : T\Delta$ y $T\Delta \Vdash \Delta$. Aplicando $\Vdash_{Clos}$ se obtiene $h' : S\Delta' \Vdash v : ST\Delta$ y $ST\Delta \Vdash S\Delta$. Como $S \leftrightarrow_\Delta T$, esto equivale a $h' : S\Delta' \Vdash v : TS\Delta$ y $TS\Delta \Vdash S\Delta$, que es lo mismo que $T; C \mid S\Delta \trianglerighteq S\Delta'$. □

Teorema 2.7. Las reglas (SimEntl), (SimComp), (SimCtx) y (SimPerm) (figura 2.1) definen una relación de simplificación, y la regla derivada (SimCHAM) es consistente con ella.

Demostración. Se probará por inducción en la estructura de las derivaciones, siendo cada caso el correspondiente a una regla:

- (SimEntl): Utilizando la hipótesis de que $h : \Delta \Vdash v_\delta : \delta$, se sigue por $\Vdash_{Univ}$ que (i) $h : \Delta \Vdash h : \Delta, v_\delta : \delta$. En forma similar, también es cierto que (ii) $h : \Delta, h_\delta : \delta \Vdash v_\delta : \delta$. Además, la conversión cumple con las condiciones de la definición, ya que (iii) $h_\delta{\leftarrow}\delta = h{\leftarrow}h \cdot h_\delta{\leftarrow}\delta$, como se probó en el lema 2.1.

- (SimComp): Las hipótesis inductivas son las correspondientes a la definición de simplificación sobre las dos premisas de la regla: (i) $h' : \Delta' \Vdash v : S\Delta$ y $h'' : \Delta'' \Vdash v' : S'\Delta'$; (ii) $S\Delta \Vdash \Delta'$ y $S'\Delta' \Vdash \Delta''$; (iii) $C = h{\leftarrow}v$ y $C' = h'{\leftarrow}v'$. Se obtendrá la evidencia $v^* = v[v'/h']$.
  Para obtener (i) se utilizarán las dos hipótesis inductivas de la primer condición. Aplicando la clausura $\Vdash_{Clos}$ (ver figura 1.1) con la sustitución S' sobre la primer hipótesis, se obtiene que $h' : S'\Delta' \Vdash v : S'S\Delta$. Luego, utilizando la regla de transitividad $\Vdash_{Trans}$ entre la implicación obtenida y la segunda hipótesis, se llega a $h'' : \Delta'' \Vdash v[v'/h'] : S'S\Delta$, tal como se necesitaba (notar que la evidencia obtenida es $v[v'/h'] = v^*$).
  Para (ii) se considerarán las dos hipótesis inductivas correspondientes. Aplicando las mismas reglas que en el caso anterior se obtiene inmediatamente que $S'S\Delta \Vdash \Delta''$.
  Finalmente, (iii) se cumple inmediatamente observando la composición se las conversiones de las hipótesis: $(C' \circ C)[e'] = e'[v[v'/h']/h] = (h{\leftarrow}v^*)[e']$, luego $C' \circ C = h{\leftarrow}v^*$ como se necesitaba.

- (SimCtx): Las hipótesis inductivas establecen que $h_2 : \Delta_2 \Vdash v : S\Delta_1$ y $S\Delta_1 \Vdash \Delta_2$. Enriqueciendo a ambos lados la primera de ellas con $h' : S\Delta'$ y la segunda con $S\Delta'$ se obtienen las condiciones (i) y (ii) . Notar que la conversión original, $h_1{\leftarrow}v_1$, es igual bajo la congruencia utilizada a la conversión necesaria para la regla: $h_1, h'{\leftarrow}v, h'$

- (SimPerm): La regla es una extensión de la propiedad clausura por permutaciones en las asignaciones de predicados de la relación de implicación. La demostración sólo se dificulta en que la conversión C^* puede no ser la misma (ya que el reemplazo por abstracción y aplicación puede dar diferentes resultados si se altera el orden) y en la manipulación de las listas de predicados.
  La condición (i) se obtiene inmediatamente de la hipótesis inductiva correspondiente.

Dada la hipótesis inductiva de que

$$h'_1 : \Delta'_1, h'_2 : \Delta'_2 \vdash v_1 : S\Delta_1, v_2 : S\Delta_2,$$

obtendremos la condición (ii) de la siguiente forma: permutando una implicación de $h'_1 : \Delta'_1$, $h'_2 : \Delta'_2$ con sí mismo por el lema 2.6 y enriqueciendo el consecuente con parte de la hipótesis (la que construye la evidencia $v_2 : \Delta_2$), sabemos que

$$h'_2 : \Delta'_2, h'_1 : \Delta'_1 \vdash h'_1 : \Delta'_1, h'_2 : \Delta'_2, v_2 : S\Delta_2.$$

Utilizando una vez más la hipótesis original, esta vez para la evidencia $v_1 : S\Delta_1$, y enriqueciendo el antecedente con una hipótesis que no agrega información ($h_2 : \Delta_2$), se obtiene que

$$h'_1 : \Delta'_1, h'_2 : \Delta'_2, h_2 : S\Delta_2 \vdash v_1 : S\Delta_1.$$

Notar que las últimas dos implicaciones pueden ser compuestas por $\vdash_{Trans}$ obteniendo entonces

$$h'_2 : \Delta'_2, h'_1 : \Delta'_1 \vdash v_1[h'_1/h'_1][h'_2/h'_2][v_2/h_2] : S\Delta_1$$

que es equivalente a $h'_2 : \Delta'_2, h'_1 : \Delta'_1 \vdash v_1[v_2/h_2] : S\Delta_1$. Como además de la hipótesis inductiva implica que $h'_2 : \Delta'_2, h'_1 : \Delta'_1 \vdash v_2 : S\Delta_2$, por $\vdash_{Univ}$ se llega a que

$$h'_2 : \Delta'_2, h'_1 : \Delta'_1 \vdash v_2 : S\Delta_2, v_1[h_2/v_2] : S\Delta_1$$

Obteniendo la condición (ii) .

Queda entonces demostrada que las reglas de la figura 2.1 definen una relación de simplificación.
Finalmente mostraremos la obtención de la regla (SimCHAM): Si $\Delta_1 \approx \Delta'_1$ y $\Delta_2 \approx \Delta'_2$, Δ_1 puede obtenerse en un número finito de intercambios entre sublistas de Δ'_1, de la forma en que se permite aplicar la regla (SimPerm); análogamente para Δ_2 y Δ'_2. De esta forma, si $S; C \mid \Delta_1 \trianglerighteq \Delta_2$ entonces $S; C^{\approx} \mid \Delta'_1 \trianglerighteq \Delta'_2$ para la conversión adecuada (tal como se explica en la definición de la regla). Una aplicación final de (SimCtx) obtiene $S; C^{\approx} \mid \Delta'_1, \Delta \trianglerighteq \Delta'_2, S\Delta$, como se deseaba. □

Teorema 2.8. La adición de las reglas ($SimOp_{rcs}$) y ($SimMG_U$) de la figura 2.2 define una relación de simplificación.

Demostración. La prueba es, al igual que el caso de las reglas básicas, por inducción y casos sobre cada regla:

- (SimOp$_{res}$): (i) Es cierto que $\emptyset \Vdash n : \hat{n} := \hat{n_1} + \hat{n_2}$, y también (ii) $\hat{n} := \hat{n_1} + \hat{n_2} \Vdash \emptyset$. La conversión es exactamente la requerida.
- (SimMG$_U$): (i) Como $C : \sigma_2 \geq \sigma_1$, por $\Vdash_{IsMG}$ se cumple que IsMG σ_2 σ_1. Además h_1 : IsMG σ_1 s, por lo que aplicando $\Vdash_{Trans}$ se llega a $h_1 \circ C$: IsMG σ_2 s. (ii) es inmediato, y la conversión es la requerida: $C = h_2 \leftarrow (h_1 \circ C) = h_1 \leftarrow h_2 \cdot h_2 \leftarrow (h_1 \circ C)$. □

Teorema 2.9 (Consistencia de (SIMP)). Se pueden realizar simplificaciones en el sistema de especialización: Dada la especialización

$$h : \Delta \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma,$$

si $S; C \mid h : \Delta \trianglerighteq h' : \Delta'$ entonces

$$h' : \Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow C[e'] : S\sigma.$$

Demostración. Por prop. 1.12, aplicando la sustitución a la hipótesis se obtiene $h : S\Delta \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\sigma$. Además, por definición de $\trianglerighteq$, se sumple que $h' : \Delta' \Vdash v : S\Delta$. Luego (prop 1.13),

$$h' : \Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e'[v/h] : S\sigma.$$

Y el término convertido es $e'[v/h] = (\Lambda h.[])((v))[e'] = C[e']$, como el propuesto en la tesis. □

Teorema 3.6 (Consistencia de (SOLV)). La resolución de restricciones puede realizarse durante la derivación de especializaciones. Para cualquier jucio de especialización $\Delta_1 \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma$, si se cumple que

$$S\colon T; C \mid \Delta_1 + \Delta' \rhd_{FTV(\Gamma,\sigma)} \Delta_2$$

entonces también es cierto que $\Delta_2 \mid TS\Gamma \vdash_{\mathrm{P}} \mathrm{e} : \tau \hookrightarrow C[e'] : TS\sigma$.

Demostración. La premisa $S\colon T; C \mid \Delta_1 + \Delta' \rhd_{FTV(\Gamma,\sigma)} \Delta_2$ implica, por definición de resolución, que es cierta la simplificación:

$$T; C \mid S\Delta_1, \Delta' \trianglerighteq \Delta_2$$

en donde $\mathrm{dom}(S) \cap (FTV(\Gamma, \sigma) \cup FTV(\Delta')) = \emptyset$.
Se toma el juicio dado como hipótesis $\Delta_1 \mid \Gamma \vdash_{\mathrm{P}} \mathrm{e} : \tau \hookrightarrow e' : \sigma$ y se enriquece el contexto con Δ' (por Th. A.12 de [Martínez López and Hughes, 2001]) obteniendo

$$\Delta_1, \Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma.$$

Luego, aplicando la sustitución S a todo el juicio (Th. A.13 de [Martínez López and Hughes, 2001]), se llega a que

$$S\Delta_1, S\Delta' \mid S\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : S\sigma.$$

Pero por las restricciones sobre el dominio de S de la hipótesis, se sabe que $S\Delta' = \Delta'$, $S\Gamma = \Delta'$ y $S\sigma = \Delta'$, por lo que esta derivación es igual a

$$S\Delta_1, \Delta' \mid \Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow e' : \sigma.$$

Finalmente, aplicando la regla (SIMP) a esa derivación, dada por la premisa de que $T; C \mid S\Delta_1, \Delta' \trianglerighteq \Delta_2$, se obtiene

$$\Delta_2 \mid T\Gamma \vdash_{\mathrm{P}} e : \tau \hookrightarrow C[e'] : T\sigma$$

como se deseaba. □

Lema 3.7. La composición de resoluciones es una resolución. Es decir, si se cumple que $S_2 \hookleftarrow_{S\Delta_1, \Delta'} T_1$ entonces las resoluciones

$$S_1 : T_1; C_1 \mid \Delta_1 + \Delta' \triangleright_V \Delta_2 \quad \text{y} \quad S_2 : T_2; C_2 \mid \Delta_2 + \Delta'' \triangleright_V \Delta_3$$

implican que

$$S_2S_1 : T_2T_1; C_2 \circ C_1 \mid \Delta_1 + (S_2\Delta', \Delta'') \triangleright_V \Delta_3$$

Demostración. Por hipótesis en la primer resolución, se tiene que

$$T_1; C_1 \mid S_1\Delta_1, \Delta' \trianglerighteq \Delta_2.$$

Utilizando la compatibilidad entre S_2 y T_1 (por lema 2.5), se puede aplicar S_2 a esta simplificación para obtener

$$T_1; C_1 \mid S_2S_1\Delta_1, S_2\Delta' \trianglerighteq S_2\Delta_2.$$

Enriqueciendo ambas listas de predicados, por la regla (SimCHAM), se obtiene

$$T_1; C_1 \mid S_2S_1\Delta_1, S_2\Delta', \Delta'' \trianglerighteq S_2\Delta_2, \Delta'',$$

Además, la primer resolución asegura que

$$T_2; C_2 \mid S_2\Delta_2, \Delta'' \trianglerighteq \Delta_3,$$

y componiendo estas dos últimas simplificaciones (regla (SimComp)), dadas también las restricciones sobre las sustituciones aseguradas por las hipótesis, se sabe que

$$S_2S_1 : T_2T_1; C_2 \circ C_1 \mid \Delta_1 + (S_2\Delta', \Delta'') \triangleright_V \Delta_3$$ □

Teorema 4.5 (Coherencia de soluciones – basado en conjetura 4.4). El término obtenido luego de resolver eliminando evidencia sobre una variable de esquema es siempre el mismo, independientemente de la solución que se elija para la misma.

Demostración. Utilizaremos el hecho de que

$$\Delta \vdash C : \text{IsMG } \sigma\ \sigma' \text{ implica } C : \sigma \geq \sigma' \tag{A.1}$$

Cuya demostración es directa por inducción en la definición de $\vdash$ (figura 1.3): hay sólo dos reglas que prueban predicados IsMG, la primera ($\vdash_{IsMG}$) es el caso base, y la segunda ($\vdash_{Comp}$) es el caso inductivo, que se prueba por transitividad de $\geq$ (propiedad 1.6).
Sean $\delta^u = \text{IsMG } \sigma\ s$ y $\delta^\ell = \text{IsMG } s\ \sigma$, σ_1, σ_2 dos soluciones para s, y S la sustitución que las representa ($S = [\sigma_1/s]$). Para la primer solución se obtienen dos conversiones C_1^u y C_1^ℓ, con $C_1^u : S\delta^u$ y $C_1^\ell : S\delta^\ell$.
Por (A.1), se cumple que $C_1^u : \sigma^u \geq \sigma_1$ y $C_2^\ell : \sigma^\ell \geq \sigma_1$. Luego, por la transitividad de $\geq$ se sabe que $C_1^\ell \circ C_1^u : \sigma^u \geq \sigma^\ell$. Con un razonamiento análogo sobre σ_2, se justifica que $C_2^\ell \circ C_2^u : \sigma^u \geq \sigma^\ell$.
Finalmente, la conjetura 4.4 (unicidad de conversiones) aseguraría que $C_1^\ell \circ C_1^u = C_2^\ell \circ C_2^u$, con lo que se concluye que el término que se obtenga por eliminación de evidencia será independiente de la solución que se haya elegido. □

Bibliografía

The web has been likened to the world's largest library
— with the books piled in the middle of the floor.

Funding a Revolution: Government Support for Computing Research.
Computer Science and Telecommunications Board,
National Academy Press, Washington, 1999

[Aiken, 1999] Alexander Aiken. Introduction to set constraint-based program analysis. In *Science of Computer Programming*, volume 35, pages 79–111, 1999.

[Berry and Boudol, 1992] G. Berry and G. Boudol. The chemical abstract machine. In *Theoretical Computer Science*, volume 96, pages 217–248, 1992.

[Breazu-Tannen *et al.*, 1989] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and coercion. In *Proceedings of the fourth annual symposium on logic in computer science*, 1989.

[Consel and Danvy, 1993] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of 20th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages (POPL '93)*, pages 493–501, Charleston, South Carolina, USA, January 1993. ACM Press.

[Damas and Milner, 1982] Luis Damas and Robin Milner. Principal type-schemes for functional languages. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[Danvy and Martínez López, 2003] Olivier Danvy and Pablo E. Martínez López. Tagging, encoding and Jones optimality. In *Programming Languages and Systems. 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 335–347, April 2003.

[Danvy *et al.*, 1996] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Selected papers of the International Seminar "Partial Evaluation"*, volume 1110 of *Lecture Notes in Computer Science*, Dagstuhl, Germany, February 1996. Springer-Verlag, Heidelberg, Germany.

[de la Canal, 2003] Esteban de la Canal. Obteniendo eficiencia y legibilidad en programas generados automáticamente. Tesis de grado de la Licenciatura en Informática en la Fac. de Informática, Universidad Nacional de La Plata, 2003.

[Futamura, 1971] Yoshihiko Futamura. Partial evaluation of computation process - An approach to a compiler-compiler. *Computer, Systems, Controls*, 2(5):45–50, 1971.

[Hannan and Hicks, 1998] John Hannan and Patrick Hicks. Higher-order arity raising. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 27–38. ACM Press, 1998.

[Hughes, 1996] John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [1996], pages 183–215.

[Jones, 1988] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14, North-Holland, 1988. IFIP World Congress Proceedings, Elsevier Science Publishers B.V.

[Jones, 1993] Mark P. Jones. Coherence for qualified types, September 1993. Research Report YALEU/DCS/RR-989, Yale University.

[Jones, 1994a] Mark P. Jones. *Qualified Types: Theory and Practice.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Jones, 1994b] Mark P. Jones. Simplifying and improving qualified types, June 1994. Research Report YALEU/DCS/RR-1040, Yale University.

[Martínez López and Badenes, 2003] Pablo E. Martínez López and Hernán Badenes. Simplifying and solving qualified types for principal type specialisation. In *Proceedings del 7mo. Workshop Argentino de Informática Teórica (WAIT 2003)*, September 2003.

[Martínez López and Hughes, 2001] Pablo E. Martínez López and John Hughes. Towards principal type specialisation. Technical report, University of Buenos Aires, 2001. URL:
`http://www-lifia.info.unlp.edu.ar/~fidel/Works/towardsPTS.dvi.tgz`.

[Martínez López and Hughes, 2002] Pablo E. Martínez López and John Hughes. Principal type specialisation. In *Proceedings of Asian Symposium on Partial Evaluation and Semantic-Based Program Manipulation (ASIA-PEPM)*. ACM Press, September 2002.

[Martínez López, 2003] Pablo E. Martínez López. *Type Specialisation of Polymorphic Languages.* PhD thesis, University of Buenos Aires, 2003. (In preparation).

[Milner, 1978] Robin Milner. A theory of type polymorphism in programming. In *Journal of Computer and System Sciences*, volume 17 of *3*, 1978.

[Mogensen, 1996] Torben Æ. Mogensen. Evolution of partial evaluators: Removing inherited limits. In Danvy et al. [1996], pages 303–321.

[Mogensen, 1998] Torben Æ. Mogensen. Inherited limits. *ACM Computing Surveys*, 30(3), September 1998.

[N.D. Jones and Sestoft, 1993] C.K. Gomard N.D. Jones and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.
URL: `http://www.dina.dk/~sestoft/pebook/pebook.html`.

[Peyton Jones and Hughes (editors), 1999] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language, February 1999. URL: `http://www.haskell.org/onlinereport/`.

[Rèmy, 1989] Didier Rèmy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, January 1989. Austin, Texas.

[Robinson, 1965] J. A. Robinson. A machine oriented logic based on the resolution principle. In *Journal of the Association for Computer Machinery*, volume 12, 1965.

[Wadler, 1993] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School.* Springer-Verlag, 1993.