Una aproximación *offline* a la evaluación parcial dirigida por *narrowing*

J. Guadalupe Ramos Díaz
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Memoria presentada para optar al título de:

Doctor en Informática

Director:

Dr. Germán F. Vidal Oriola

Tribunal de Lectura:

Presidente:	Dra. María Alpuente Frasnedo	U.P. Valencia
Vocales:	Dr. Ginés Moreno Valverde	U. Castilla-La Mancha
	Dr. Pascual Julián Iranzo	U. Castilla-La Mancha
	Dr. Germán Puebla Sánchez	U.P. Madrid
Secretario:	Dr. Francisco Javier López Fraguas	U.C. Madrid

Valencia, Mayo de 2007.

Resumen

La evaluación parcial dirigida por *narrowing* (NPE: *Narrowing-driven Partial Evaluation*) es una técnica potente para la especialización de sistemas de reescritura [AV02], i.e., para el componente de primer orden de muchos lenguajes declarativos (lógico) funcionales como Haskell [PJ03], Curry [Han03] o Toy [LFSH99].

Los evaluadores parciales se clasifican en dos grandes categorías: *online* y *offline*, de acuerdo al momento temporal en que se consideran los aspectos de terminación del proceso de especialización. Los evaluadores parciales *online* son usualmente más precisos ya que tienen más información disponible. En particular, el esquema original NPE, el cual sigue la aproximación *online*, considera una variante del teorema de Kruskal (*Kruskal's Tree Theorem*) llamada *"subsumción homeomórfica"* [Leu02], para asegurar la terminación del proceso: si un término *subsume* algún término previo en la misma computación de *narrowing*, se aplica alguna forma de generalización—usualmente el operador de generalización más específica—y la evaluación parcial se reinicia con los términos generalizados [AFV98]. Sin embargo, las pruebas para el test de subsumción, junto con las generalizaciones asociadas, hacen que el esquema NPE *online* sea muy costoso (en términos de tiempo y espacio), por lo que no se adapta adecuadamente a problemas realistas como la especialización de intérpretes.

Los evaluadores parciales *offline* proceden comúnmente en dos etapas; la primera etapa procesa un programa (e.g., para identificar aquellas llamadas a función que se pueden desplegar sin riesgo de no terminación) e incluye anotaciones para guiar las computaciones parciales; entonces, una segunda etapa, la de evaluación parcial propiamente dicha, sólo tiene que obedecer las anotaciones y por tanto el especializador es mucho más rápido que en la aproximación *online*.

En esta tesis se presenta un nuevo esquema de evaluación parcial dirigido por *narrowing*, más eficiente y que asegura la terminación siguiendo el estilo *offline*. Para ello, identificamos una caracterización de programas cuasi-terminantes a los que llamamos "no crecientes". En tales programas, las computaciones por *narrowing* necesario presentan sólo un conjunto finito de términos diferentes (módulo renombramiento

de variables). La propiedad de la *cuasi-terminación* es importante toda vez que su presencia es regularmente una condición suficiente para la terminación del proceso [Hol91] de especialización.

Sin embargo, la clase de programas cuasi-terminantes es muy restrictiva, por lo que introducimos un algoritmo que acepta programas *inductivamente secuenciales*—una clase mucho más amplia sobre la que está definido el *narrowing* necesario—y anota aquellas partes que violan la caracterización de programas no crecientes. Para procesar de manera adecuada los nuevos términos anotados extendemos la relación de *narrowing*, a la que llamamos: *narrowing* necesario generalizante.

Una vez definido el esquema, desarrollamos un prototipo para la evaluación parcial *offline* dirigida por *narrowing*. La validación experimental arroja resultados que muestran una mejora significativa en el tiempo de especialización *offline* con respecto al esquema *online*.

Recientemente, se ha formulado un nuevo principio para el análisis de terminación de los programas basado en el cambio de tamaño de los argumentos de las llamadas a función [LJBA01, GJ05]. Con la finalidad de mejorar la precisión de la fase de anotación propia del nuevo esquema, hemos adaptado los grafos de cambio de tamaño (grafos *size-change*), introducidos originalmente para los lenguajes funcionales, a los lenguajes lógico funcionales. En particular los grafos nos son útiles para determinar una forma específica de cuasi-terminación, que finalmente utilizamos para realizar anotaciones al estilo de nuestra primera propuesta *offline*.

Los lenguajes empotrados de dominio específico (DSELs) generan código ineficiente debido a la sobrecarga de interpretación, por lo que [Hud98, SCK04, Chr03, HM04] entre otros, plantean el uso de las técnicas de evaluación parcial como un medio para remediar este inconveniente. En este punto, seleccionamos el dominio de los *routers*, en particular el modelo de *routers software Click* [KMC+00] y desarrollamos un lenguaje de especificación para *routers* denominado *Rose*, empotrado en el lenguaje Curry.

Los DSELs aportan una vía de prueba interesante en dos aspectos: primero, han permitido ejercitar los conceptos de la programación lógico funcional en la generación de lenguajes de dominio específico; y segundo, nos han proporcionado ejemplos para la especialización de programas mediante las técnicas NPE *offline* desarrolladas.

Abstract

Narrowing-driven partial evaluation (NPE) is a powerful specialization technique for rewrite systems [AV02], i.e., for the first-order component of many functional (logic) languages like Haskell [PJ03], Curry [Han03] or Toy [LFSH99].

Partial evaluators fall in two main categories, *online* and *offline*, according to the time when termination issues are addressed. Online partial evaluators are usually more precise since more information is available. For instance, the original NPE scheme (which follows the online approach) considers a variant of the Kruskal tree condition called "homeomorphic embedding" [Leu02] to ensure the termination of the process [AFV98]: if a term embeds some previous term in the same narrowing computation, some form of generalization—usually the *most specific generalization* operator—is applied and partial evaluation is restarted with the generalized terms. However, this extra precision comes at a cost: the homeomorphic embedding tests, together with the associated generalizations, make NPE very expensive and, thus, it does not scale up well to realistic problems like interpreter specialization.

Offline partial evaluators usually proceed in two stages: the first stage returns a program that includes annotations to guide the partial computations (e.g., to identify those function calls that can be safely unfolded); then, the second stage—the proper partial evaluation—only needs to obey the annotations and, thus, it is generally much faster than online partial evaluators.

In this thesis we present a more efficient narrowing-driven partial evaluation scheme that ensures termination by following the offline approach. For this, we identify a class of quasi-terminating rewrite systems, called *nonincreasing*. In such programs, only finitely many different terms—modulo variable renaming—are computed by needed narrowing. This is an important property since quasi-termination of programs is frequently a sufficient condition to ensure the termination of the specialization process [Hol91].

Unfortunately, this class is too restrictive and, thus, we also introduce an algorithm that takes an *inductively sequential* program (a much broader class of programs

for which NPE is originally defined) and annotates those expressions that violate nonincreasing characterization. Then, we define an extended needed narrowing relation: *generalizing needed narrowing*, in which annotated subterms are generalized.

Once the new partial evaluation method is defined, we develop an offline narrowing-driven partial evaluator prototype. Its experimental validation demonstrates that the offline scheme is faster than the online approach.

Recently, a new principle for termination analysis of programs was formulated. The principle is based on the size change of arguments at function calls: *size-change graphs* [LJBA01, GJ05]. Since size-change graphs were originally introduced for functional languages, we adapt the formalism to functional logic programs. Size-change graphs are useful to detect a specific form of quasi-termination, which is used to make annotations analogously to our first offline approximation. This allows us to improve the precision of the annotation phase.

One important partial evaluation challenge is the specialization of realistic applications. Domain Specific Embedded Languages (DSELs) produce inefficient code due to the interpretation overhead. This is the reason why [Hud98, SCK04, Chr03, HM04] propose partial evaluation in order to overcome this drawback. We select the *routers* domain, in particular we choose the *Click software router model* [KMC+00] and develop a language for router specification embedded in Curry: *Rose*.

Finally, the development of Rose was also useful to test the pros and cons of Curry in the DSEL area. It was also useful to check the power of the partial evaluation scheme in order to compile programs by using the NPE technique.

Resum

L'avaluació parcial dirigida per *narrowing* (NPE: *Narrowing-driven Partial Evaluation*) és una tècnica potent per a l'especialització de sistemes de reescriptura [AV02], i.e., per al component de primer ordre de molts llenguatges declaratius (lògic) funcionals com Haskell [PJ03], Curry [Han03] o Toy [LFSH99].

Els avaluadors parcials es classifiquen en dos grans categories: *online* i *offline*, d'acord al moment temporal que es consideren els aspectes de terminació del procés d'especialització. Els avaluadors parcials *online* són usualment més precisos ja que tenen més informació disponible. En particular, l'esquema original NPE, el qual segueix l'aproximació *online*, considera una variant del teorema de Kruskal (*Kruskal's Tree Theorem*) anomenada "subsumpció homeomòrfica" [Leu02], per a assegurar la terminació del procés: si un terme subsumeix algun terme previ en la mateixa computació de *narrowing*, s'aplica alguna forma de generalització—usualment l'operador de generalització més específica—i l'avaluació parcial es reinicia amb els termes generalitzats [AFV98]. No obstant això, les proves per al test de subsumpció, juntament amb les generalitzacions associades, fan que l'esquema NPE *online* siga molt costós (en termes de temps i espai), pel que no s'adapta adequadament a problemes realistes com l'especialització d'intèrprets.

Els avaluadors parcials *offline* procedeixen comunament en dues etapes; la primera etapa processa un programa (e.g., per a identificar aquelles crides a funció que es poden desplegar sense risc de no terminació) i inclou anotacions per a guiar les computacions parcials; llavors, una segona etapa, la d'avaluació parcial pròpiament dita, només ha d'obeir les anotacions i per tant el especialitzador és molt més ràpid que en l'aproximació *online*.

En aquesta tesi es presenta un nou esquema d'avaluació parcial dirigit per *narrowing*, més eficient i que assegura la terminació seguint l'estil *offline*. Per a això, identifiquem una caracterització de programes quasi-terminants als quals anomenem "no creixents". En tals programes, les computacions per *narrowing* necessari presenten només un conjunt finit de termes diferents (mòdul reanomenament de variables). La

propietat de la quasi-terminació és important atès que la seva presència és regularment una condició suficient per a la terminació del procés [Hol91] d'especialització.

No obstant això, la classe de programes quasi-terminants és molt restrictiva, pel que introduïm un algorisme que accepta programes inductivament seqüencials—una classe molt més àmplia sobre la qual està definit el *narrowing* necessari—i anota aquelles parts que violen la caracterització de programes no creixents. Per a processar de manera adequada els nous termes anotats estenem la relació de *narrowing*, a la qual anomenem: *narrowing* necessari generalitzant.

Una vegada definit l'esquema, desenvolupem un prototip per a l'avaluació parcial *offline* dirigida per *narrowing*. La validació experimental llança resultats que mostren una millora significativa en el temps d'especialització *offline* pel que fa a l'esquema *online*.

Recentment, s'ha formulat un nou principi per a l'anàlisi de terminació dels programes basat en el canvi de grandària dels arguments de les crides a funció [LJBA01, GJ05]. Amb la finalitat de millorar la precisió de la fase d'anotació pròpia del nou esquema, hem adaptat els grafs de canvi de grandària (grafs *size-change*), introduïts originalment per als llenguatges funcionals, als llenguatges lògic funcionals. En particular els grafs ens són útils per a determinar una forma específica de quasi-terminació, que finalment utilitzem per a realitzar anotacions a l'estil de la nostra primera proposta *offline*.

Els llenguatges encastats de domini específic (DSELs) generen codi ineficient a causa de la sobrecàrrega d'interpretació, pel que [Hud98, SCK04, Chr03, HM04] entre uns altres, plantegen l'ús de les tècniques d'avaluació parcial com un mitjà per a remeiar aquest inconvenient. En aquest punt, seleccionem el domini dels *routers*, en particular el model de *routers software Click* [KMC+00] i desenvolupem un llenguatge d'especificació per a *routers* denominat *Rose*, encastat en el llenguatge Curry.

Els DSELs aporten una via de prova interessant en dos aspectes: primer, han permès exercitar els conceptes de la programació lògic funcional en la generació de llenguatges de domini específic; i segon, ens han proporcionat exemples per a l'especialització de programes mitjançant les tècniques NPE *offline* desenvolupades.

Agradecimientos

Para el desarrollo de un proyecto tan importante como lo es una tesis de doctorado, hace falta el esfuerzo y el apoyo de muchas personas e Instituciones, mismas que deseo enumerar aquí, con la esperanza de no omitir a nadie al momento de manifestar mi agradecimiento.

De manera muy especial a mi director de tesis, el Dr. Germán Vidal, quien me brindó un apoyo total más allá de su responsabilidad. Inclusive en los momentos difíciles cuando me encontraba lejos de mi familia. A Josep Silva, por sus consejos técnicos, por su ejemplo al afrontar los problemas con serenidad pero de manera brillante y sin duda, por los momentos fraternos compartidos en la UPV.

A nuestros profesores de la primera etapa del doctorado: Isidro Ramos, Matilde Celma, Ma. José Ramírez, Oscar Pastor, María Alpuente, Salvador Lucas y al mismo Germán Vidal, gracias por ser guías en cuanto a profesionalismo, tenacidad y calidad científica.

Gracias también a mis amigos del DSIC: Cèsar Ferri, José Hernández, Vicent Estruch, Santiago Escobar, Alicia Villanueva, etc, por todos los momentos compartidos, por sus enseñanzas y consideraciones recibidas.

No puedo olvidar a mis compatriotas Leopoldo, Clemente, Ricardo Quintero y en especial a los compañeros de cuitas y vericuetos: Rogelio, Ricardo Blanco y Gustavo Arroyo.

Mención singular merece mi familia. En estos tiempos modernos no es fácil sacrificar la compañía de la familia por la superación personal. Han sido momentos muy duros, no sólo por la lejanía sino por las pruebas que va descubriendo la vida. Gracias pues, a mi esposa Isela, a mi pequeña Jessica y a José Vicente que se unió al grupo mientras nos encontrabamos en Valencia, y en recuerdo a ello su nombre es. Gracias a mis hermanos Maria, Isabel, Alfonso, Martín, José, Hermelinda, Carmen, Luz, Gustavo y Florencio q.e.p.d., también a mis padres porque han sido inspiración y añoranza en todo momento. A Dios gracias, porque el hombre, para subsistir, necesita reconocer su condición temporal.

A las autoridades del Instituto Tecnológico de La Piedad: Juan Manuel Salazar, José Carlos Paz, Juan Carlos Solorio, Rubén García y Fernando Sánchez, les agradezco todo su apoyo personal y de gestión.

Este proyecto recibió ayuda económica de diferentes Instituciones a las que agradezco y cito a continuación:

- En México: Convenio SES-Asociación Nacional de Universidades e Instituciones de Educación Superior, DGEST-SEP e ITLP y,
- En España: Vicerrectorado de Cooperación y Proyectos de Desarrollo de la UPV, DSIC, grupo ELP y el grupo MIST.

A los revisores gracias por sus consejos y correcciones que indudablemente contribuyen en una mejor calidad del presente documento.

Gracias al grupo MIST y al grupo ELP por haber sido mis anfitriones en Valencia.

Índice general

1. Introducción **1**

1.1. Lenguajes de dominio específico 2

1.1.1. Lenguajes empotrados 4

1.1.2. Eficiencia de los lenguajes empotrados 5

1.2. Lenguajes declarativos lógico funcionales 6

1.2.1. Antecedentes 7

1.2.2. El lenguaje lógico funcional Curry 8

1.2.3. El DSEL Rose 11

1.3. Evaluación parcial de programas 12

1.4. Evaluación parcial de programas lógico funcionales 14

1.4.1. Antecedentes 15

1.4.2. Evaluación parcial dirigida por *narrowing* 20

1.4.3. El problema de la terminación 22

1.5. Objetivos de la tesis 25

1.5.1. Planteamiento 25

1.5.2. Objetivo general 25

1.5.3. Contribuciones 26

1.6. Organización de la memoria 27

2. Preliminares **29**

2.1. Signaturas y términos 29

2.2. Sustituciones 30

2.3. Sistemas de reescritura de términos 31

2.4. Programación lógico funcional 32

2.4.1. *Narrowing* y estrategia de *narrowing* 34

2.4.2. *Narrowing* necesario 36

3. Evaluación parcial dirigida por *narrowing* **45**
3.1. Antecedentes . . . 45
3.2. Resultantes y pre–evaluación parcial . . . 46
3.3. El cierre de los programas especializados . . . 49
3.4. Renombramiento . . . 50
3.5. Evaluación parcial . . . 52
3.6. Aspectos de control . . . 56
3.6.1. La relación de subsumción homeomórfica . . . 57
3.6.2. El algoritmo básico . . . 57
3.6.3. El control local y el control global . . . 58
3.7. Conclusiones . . . 60

4. Evaluación parcial *offline* dirigida por *narrowing* **61**
4.1. Antecedentes . . . 61
4.2. Cuasi-terminación de las computaciones de *narrowing* necesario . . . 62
4.3. Del esquema NPE *online* al esquema NPE *offline* . . . 70
4.4. El método de evaluación parcial *offline* dirigido por *narrowing* . . . 75
4.4.1. Descripción del método NPE *offline* . . . 75
4.4.2. Aspectos de control . . . 78
4.4.3. Ejemplos . . . 79
4.4.4. Evaluación experimental . . . 84
4.5. Conclusiones . . . 86

5. Implementación del evaluador parcial *offline* **89**
5.1. La representación abstracta de programas . . . 89
5.2. Facilidades de meta-programación en Curry . . . 91
5.3. La herramienta de evaluación parcial . . . 93
5.3.1. La semántica de residualización . . . 93
5.3.2. Fases del evaluador parcial . . . 97
5.3.3. Fase de anotación . . . 98
5.3.4. Fase de especialización . . . 99
5.4. Construcción del árbol de *narrowing* . . . 102
5.4.1. Estructura de datos del árbol de búsqueda de *narrowing* . . . 103
5.4.2. RLNT . . . 104
5.4.3. Aspectos de control . . . 105
5.5. El evaluador parcial *offline* en la práctica . . . 107
5.6. Conclusiones . . . 111

6. Análisis por grafos *size-change* **113**
6.1. Antecedentes . . . 113
6.2. Grafos *size-change* . . . 114
6.3. Cuasi-terminación basada en grafos *size-change* . . . 118
6.3.1. EP-terminación . . . 118
6.3.2. Aspectos de control . . . 119
6.3.3. Cuasi-terminación por EP-terminación . . . 120
6.4. Un análisis BTA para programas Curry . . . 124
6.5. Nuevo esquema de anotación . . . 129
6.6. Trabajo relacionado . . . 132
6.7. Conclusiones . . . 132

7. El lenguaje empotrado de dominio específico Rose **135**
7.1. Antecedentes . . . 135
7.2. El Lenguaje Click . . . 137
7.3. El lenguaje de especificación Rose . . . 139
7.3.1. Operadores de composición . . . 145
7.3.2. Operadores de flujo . . . 146
7.3.3. Implementación . . . 148
7.4. Trabajo relacionado . . . 149
7.5. Conclusiones . . . 150

8. Aplicación del evaluador parcial **151**
8.1. Antecedentes . . . 151
8.2. Compilación por evaluación parcial . . . 152
8.3. El orden superior y NPE . . . 154
8.4. Compilación . . . 156
8.4.1. Compilando Rose . . . 156
8.4.2. Un intérprete funcional simple . . . 160
8.5. Trabajo relacionado y conclusiones . . . 162

9. Conclusiones y líneas de trabajo futuro **163**
9.1. Contribuciones . . . 163
9.2. Trabajo futuro . . . 165

Bibliografía **167**

Capítulo 1

Introducción

La investigación que se describe en esta memoria, se relaciona fundamentalmente con la evaluación parcial de programas, una técnica formal para la *especialización* y optimización de programas. Un evaluador parcial toma un programa y parte de sus datos de entrada (los llamados datos *estáticos*) e intenta llevar a cabo todas las computaciones que sean posibles a partir de tales datos. El evaluador parcial devuelve un programa nuevo, denominado programa *residual*, el cual se ejecuta generalmente de manera más eficiente que el programa original, ya que las computaciones que dependen de los datos estáticos se han realizado en la fase de evaluación parcial de una vez y para siempre.

Un aspecto clave para asegurar la terminación del proceso de especialización es la adecuada selección de las partes del programa que deberían computarse en tiempo de especialización [GJ05]. La atención se debe centrar en aquellas partes (normalmente llamadas a función) que podrían producir computaciones infinitas. Para evitarlas, se aplica a menudo alguna forma de *generalización*. La decisión acerca de qué llamadas a función se deberían generalizar puede tomarse de modo *online* u *offline*. Los evaluadores parciales *online* deciden qué llamadas a función se deben generalizar y cuáles no durante el proceso de especialización, mientras que los evaluadores parciales *offline* proceden en dos etapas: la primera hace un análisis estático del programa y "anota" aquellas llamadas a función que se deberían generalizar; la segunda (la etapa de especialización propiamente dicha) sólo sigue las anotaciones y de esa manera el evaluador parcial *offline* es normalmente más rápido (pero generalmente menos preciso).

El esquema NPE de evaluación parcial dirigida por *narrowing*[1] (NPE: *Narrowing-driven Partial Evaluation*) es una técnica potente para la especialización (*online*)

[1] Para ser acordes con la literatura usaremos la acepción inglesa "*narrowing*" en lugar de su traducción al castellano: estrechamiento.

de sistemas de reescritura [AV02], i.e., el componente de primer orden de diversos lenguajes (lógico) funcionales como Haskell [PJ03] o Curry [Han03]. Sin embargo, este esquema no se adapta bien a la especialización de programas realistas (como la especialización de lenguajes de dominio específico) debido a que sigue el estilo *online* de evaluación parcial. Esto significa que durante el proceso de especialización se tendrán que llevar a cabo costosos análisis para determinar las funciones que pueden desplegarse sin riesgo de no terminación. El cómputo asociado para tomar esta decisión incide directamente en la eficiencia y capacidad de especialización del esquema NPE.

Un lenguaje de dominio específico (DSL[2]: *Domain Specific Language*) es un lenguaje de programación diseñado para un dominio particular de aplicación. Al diseñar un DSL, el objetivo principal es abstraer el dominio del problema para facilitar la generación rápida de aplicaciones, por lo que los aspectos relacionados con la eficiencia del DSL son secundarios. Hudak [Hud98] propuso el uso de la evaluación parcial para mejorar la eficiencia de las aplicaciones escritas en un DSL. En la propuesta de Hudak se plantea la aplicación de la compilación por evaluación parcial [Fut71], i.e., especializar un interprete con respecto a un programa y, de ese modo, reducir la sobrecarga de interpretación.

A continuación introducimos el contexto de los lenguajes de dominio específico, así como el concepto de evaluación parcial de programas y la relación entre ambos. Posteriormente, nos centramos en la evaluación parcial de programas lógico funcionales y, en particular, en los aspectos de terminación. Finalmente, planteamos los objetivos de la tesis.

1.1. Lenguajes de dominio específico

Un lenguaje de dominio específico es un lenguaje de programación diseñado para un dominio de aplicación particular [Hud98]. A partir de un DSL se pueden desarrollar programas para el dominio de aplicación de manera rápida y eficiente. Un DSL no es necesariamente de "propósito general", al contrario, debe capturar de manera precisa la semántica del dominio de aplicación. Bentley [Ben86] estableció otro requisito al definirlos como lenguajes pequeños (*little languages*).

Algunos ejemplos de DSLs son: Lexx y Yacc, para generar analizadores léxicos y sintácticos de programas; PERL, para la manipulación de texto; VHDL, para describir componentes hardware, etc.

Existen muchas ventajas en el uso de DSLs. De entre ellas, destacamos que los programas son generalmente más fáciles de escribir, entender y mantener compara-

[2] En concordancia con la literatura, en este documento usaremos el acrónimo DSL—por sus siglas en inglés—para referirnos a los lenguajes de dominio específico.

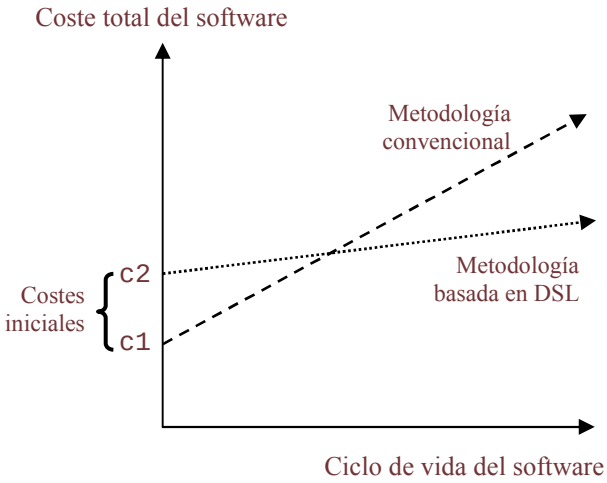


Figura 1.1: Tendencia de coste de un DSL en el tiempo [Hud98].

dos con los programas desarrollados en lenguajes de propósito general. Aunque estas ventajas están presentes en el uso de un lenguaje de alto nivel, un DSL es aún de más alto nivel y, consecuentemente, puede utilizarse incluso por aquellos que no son programadores expertos.

En la Figura 1.1 se compara, por un lado, la evolución del coste (en el tiempo) de desarrollar un DSL (`c2`) con respecto al coste implicado en producir una solución convencional a la medida de las necesidades del dominio (`c1`). Inicialmente, la solución convencional es más económica (no hay que diseñar un nuevo lenguaje de programación) pero, al transcurrir el tiempo, las líneas que comparan las tendencias se intersectan debido a que los cambios requeridos en el sistema diseñado para el dominio son materializados más fácilmente en el contexto del DSL (a menor coste). Así, la tecnología DSL se adapta mejor a la evolución del sistema.

Sin embargo, diseñar e implementar un lenguaje de programación desde cero (sistema de tipos, sintaxis, etc) y las herramientas de compilación asociadas (analizador léxico, analizador sintáctico, optimizadores, generadores de código, etc) implica un proceso tan costoso que podría dar lugar, inclusive, a que las tendencias de costes mencionadas nunca se intersectasen. Una aproximación alternativa sugiere empotrar el nuevo DSL en un lenguaje de propósito general, dando lugar a los llamados lenguajes empotrados de dominio específico.

1.1.1. Lenguajes empotrados

Para hacer asequible la escritura de un nuevo lenguaje de programación, Hudak [Hud98] sugiere no desarrollar el nuevo lenguaje desde cero. En su lugar, es más conveniente aprovechar la infraestructura de algún otro lenguaje de propósito general (al cual se le llama *lenguaje anfitrión*), buscando que sus características aporten ventajas significativas para modelar el dominio deseado. Al lenguaje obtenido a partir de esta nueva aproximación se le llama *lenguaje empotrado de dominio específico* (DSEL[3]: *Domain Specific Embedded Language*). Sobre esta base, el diseño del lenguaje se centra sólo en aspectos semánticos, de manera que la sintaxis y las herramientas asociadas (depuradores, optimizadores, etc) del lenguaje anfitrión se pueden reutilizar.

Landin [Lan66] propuso previamente usar un lenguaje de programación "anfitrión" (ya existente) que proporcionase una infraestructura genérica apropiada (tipos de datos, abstracción de datos y funciones, etc) al cual se le agregaría un vocabulario adaptado al dominio. El vocabulario consistiría en uno o más tipos de datos y funciones definidas sobre los tipos de datos básicos.

La idea de empotrar un lenguaje pequeño en otro más grande con la finalidad de resolver alguna tarea específica ha sido empleada, e.g., en el uso de macros. De hecho, los lenguajes Lisp [LR64] y C++ soportan empotramiento por medio de macros. Por otro lado, existen algunas aproximaciones modernas, orientadas a objetos, tales como Jakarta [BLS98], que atienden el problema desde un punto de vista de generación de software, i.e., un DSL es un lenguaje de especificación que algún generador de software pone a disposición del programador para, a continuación, crear el programa de interés. No obstante, una aproximación correcta de empotramiento debe reunir las siguientes características: primero, el empotramiento debe ser puro, i.e., dado un programa para el dominio, se debe ejecutar sin la presencia de ningún tipo de preproceso, ni expansión de código al estilo de las macros, ni tampoco debe asociar un proceso de generación de código a partir de una especificación. En segunda instancia, un DSEL debe enfatizar la importancia de la semántica del dominio, esto es, que la combinación de objetos del DSEL sea un modelo que refleje la semántica de los objetos en el mundo real.

La aproximación estándar para la construcción de DSELs consiste en implementarlos como librerías [SCK04] en un lenguaje que proporcione las ventajas técnicas citadas a continuación:

- Manejo de funciones de orden superior para modelar conceptos que puedan tratarse como elementos básicos del lenguaje.
- Un sistema de tipos flexible y extensible que permita abstraer y modelar los

[3] En concordancia con la literatura usaremos el acrónimo formado a partir de las siglas en inglés.

conceptos del dominio.

- Mecanismos de control sintáctico (por ejemplo, que permita agregar nuevos operadores al lenguaje).
- Un mecanismo de computación perezoso para operar con estructuras de datos infinitas, usualmente útiles para crear modelos de conceptos del dominio.

Los lenguajes declarativos poseen tales características, por lo que son buenos anfitriones para el desarrollo de DSELs. Por otro lado, el paradigma declarativo lógico funcional agrega algunas características adicionales, e.g., indeterminismo, variables lógicas, etc., interesantes en el campo de los DSELs. No obstante, los lenguajes construidos a partir de la aproximación DSEL producen generalmente código ineficiente, como trataremos a continuación.

1.1.2. Eficiencia de los lenguajes empotrados

Cuando se diseña un lenguaje empotrado de dominio específico, el objetivo principal es abstraer el dominio del problema para facilitar la generación rápida de aplicaciones. Por otro lado, los aspectos relacionados con la eficiencia, tanto del código del DSEL como su traducción al lenguaje anfitrión, no se tienen en cuenta. Además, el análisis de eficiencia debe revisarse a dos niveles: primero, teniendo en cuenta las construcciones del DSEL y, segundo, considerando su traducción a código anfitrión.

En [Hud98] se indica que un programa escrito a partir de un DSEL puede verse como una aplicación ejecutable en sí misma y que, sin embargo, es conveniente considerar la existencia de un intérprete que capture su semántica y que le de significado. Este enfoque permite el diseño de intérpretes modulares en donde cada módulo es un bloque que implementa algún aspecto del dominio. Las características nuevas que le sean requeridas al DSEL se pueden ir agregando inclusive en etapas posteriores al diseño del DSEL. El problema de esta aproximación es que cada bloque impone una capa particular de interpretación. Hudak plantea el uso de evaluación parcial para resolver este problema.

Seefried, Chakravarty y Keller [SCK04] afirman que los DSELs se compilan frecuentemente a código ineficiente debido a que los tipos de datos del dominio se representan con tipos de datos algebraicos y se interpretan por funciones recursivas. Cuando el código es traducido al lenguaje anfitrión, esta sobrecarga de interpretación está presente en el código generado ya que el compilador del lenguaje anfitrión "no sabe" cómo optimizar el uso de los tipos de datos del dominio (puesto que los compiladores optimizan sólo a nivel de las construcciones del lenguaje anfitrión). Seefried et al. proponen el uso de un lenguaje anfitrión cuyo compilador soporte meta-programación

en tiempo de compilación que les permita manipular el código en dos sentidos: del programa DSEL al lenguaje anfitrión (reificación) y del lenguaje anfitrión al programa DSEL (reflexión), utilizando para ello estructuras de datos intermedias.

Elliot et al. [EFM00] optaron por incluir un compilador incrustado en el DSEL, de tal manera que al introducirse la aplicación DSEL en el compilador del lenguaje anfitrión obtienen un programa optimizado en un lenguaje destino "impaciente" (en el que las llamadas a función exigen primero la evaluación de los argumentos) de primer orden.

Backhouse [Bac02] aporta una metodología basada en *interpretación abstracta* (Cousot y Cousot [CC77]) para analizar programas de dominio específico, con el objeto de llevar a cabo una búsqueda de errores y la revisión de las condiciones asociadas para poder ejecutar optimizaciones de domino específico. Christensen [Chr03] sugiere el uso de evaluación parcial de programas para mejorar el rendimiento de los programas de lenguajes de dominio específico empotrados. Herman y Meunier [HM04] realizan un análisis estático de programas empotrados mediante una aproximación "ligera" a la evaluación parcial (usando macros del lenguaje anfitrión) con la finalidad de encontrar errores a nivel del DSEL imposibles de detectar en el nivel de código anfitrión. Para ello, proveen los datos estáticos necesarios para el análisis. En este caso, el objetivo es la detección temprana de errores en programas escritos a partir de un DSEL.

1.2. Lenguajes declarativos lógico funcionales

Los lenguajes declarativos son anfitriones adecuados para desarrollar lenguajes de dominio específico empotrados. Es por ello, que podemos encontrar en la literatura ejemplos variados de DSELs para distintos dominios, e.g., en el campo de la programación de robots [PHH99], para el desplazamiento de robots humanoides [HH03], en el campo de la música [HMGW96], en la programación de gráficos [EH97], en la programación de animaciones [Hud00], etc. En el paradigma lógico podemos citar las gramáticas de cláusulas definidas (las DCGs—por sus siglas en inglés—de Prolog [CM87]). Las DCGs constituyen un medio propicio para representar relaciones gramaticales en aplicaciones que requieran análisis sintáctico—*parsing*—de expresiones, de hecho, pueden usarse para la creación de lenguajes de programación y en aplicaciones de lenguaje natural, entre otras. También, existen notables esfuerzos para incorporar las propiedades de la lógica difusa a un entorno de programación lógica, i.e., un DSEL que modele operaciones de la lógica difusa incrustado en Prolog (algunos avances sobre los principios operacionales en [JMP06]).

No obstante, existen pocas evidencias de la creación de tales lenguajes en el campo lógico funcional (algunas excepciones son, e.g., para la creación de interfaces gráficas

de usuario [Han00a] y para *web-scripting* [Han00b]). Aún así, el paradigma lógico funcional (el cual se puede considerar "maduro" hoy en día) ofrece características adicionales a los paradigmas declarativos puros que los hacen interesantes para la experimentación del modelado de DSLs.

A continuación revisamos brevemente los principales conceptos de la programación lógico funcional, introducimos el lenguaje lógico funcional Curry y, finalmente, presentamos una síntesis de Rose, un DSEL empotrado en Curry.

1.2.1. Antecedentes

Los lenguajes de programación lógico funcionales combinan las características más importantes de la programación funcional (funciones anidadas, computaciones funcionales eficientes conducidas por la demanda, orden superior), de la programación lógica (variables lógicas, estructuras de datos parciales, restricciones, búsqueda encapsulada), y de la programación concurrente (computaciones concurrentes con sincronización sobre las variables lógicas). Así, el modelo de computación más popular de los lenguajes integrados modernos se basa en la combinación de dos principios operacionales diferentes: *narrowing* y residuación. El mecanismo de *narrowing* permite la instanciación de variables en las expresiones de forma que luego se puedan aplicar pasos de reducción sobre las expresiones instanciadas. Dicha instanciación se computa normalmente unificando un subtérmino de la expresión con la parte izquierda de alguna regla del programa; en otros casos, viene impuesta por el uso de ciertas estructuras de datos que se usan para guiar el proceso, como son los árboles definicionales de [Ant92].

El mecanismo de *narrowing* es completo en el sentido de la programación lógica —computación de todas las soluciones— así como en el de la programación funcional —computación de valores—. Hanus presentó en [Han94] una revisión muy completa de los lenguajes de programación lógico funcionales. Se han formulado diversas estrategias de *narrowing* perezoso con la finalidad de encontrar una estrategia óptima que permita operar con estructuras de datos infinitas. Debido a sus propiedades de optimalidad respecto a la longitud de las derivaciones y al número de soluciones computadas, el *narrowing* necesario es actualmente la mejor estrategia de *narrowing* perezoso para programas lógico funcionales (ver Antoy et al. [AEH00]).

Por otra parte, el segundo principio operacional de los lenguajes lógico funcionales, la residuación, se basa en la idea de retrasar, o *suspender*, la evaluación de las llamadas a función hasta que estén listas para una evaluación determinista (i.e., puramente funcional). La residuación preserva la naturaleza determinista de las funciones y soporta computaciones concurrentes mediante sincronización sobre variables lógicas.

1.2.2. El lenguaje lógico funcional Curry

Curry [Han03] es un lenguaje declarativo multi-paradigma, el cual combina de manera elegante características de la programación funcional, de la programación lógica y de la programación concurrente.

Un programa Curry [Han03] especifica la semántica de las expresiones que lo componen donde los objetivos (habituales en la programación lógica) son aquí expresiones de carácter particular, de un tipo denominado `Success`. La ejecución de un programa Curry implica simplificar una expresión hasta que un valor (o solución) sea computado. Para diferenciar entre valores y expresiones reducibles Curry hace un distinción estricta entre *constructores* (de datos) y *operaciones* o *funciones definidas* sobre esos datos. De esa forma, un programa se compone de un conjunto de tipos y de un conjunto de declaraciones de función. Las declaraciones de tipo definen los dominios computacionales (constructores) y las declaraciones de función las operaciones sobre esos dominios. Los predicados en el sentido de la programación lógica se pueden considerar como restricciones, i.e., funciones cuyo resultado es de tipo `Success`.

Curry incluye un conjunto de constructores de tipo pre-definidos, como `Bool`, `Int`, `->`, o listas y tuplas. Puesto que es un lenguaje de orden superior, los tipos de las operaciones y de los constructores se escriben en su forma currificada:

τ_1 `->` τ_2 `->` ... `->` τ_n`->` τ

donde τ no es un tipo funcional. En este caso, n se denomina la *aridad* de la operación o del constructor.

También, el programador puede escribir sus propias declaraciones de tipo mediante:

`data` T α_1 ... α_n `=` C_1 τ_{11} ... τ_{1n_1} | ... | C_k τ_{k1} ... τ_{kn_k}

donde se introduce T, un nuevo constructor de tipo n-ario, y también k nuevos *constructores* de datos: $C_1, \ldots, C_k$. Cada τ_{ij} es una *expresión de tipo* formada a partir de las *variables de tipo* $\alpha_1 \ldots \alpha_n$ y, posiblemente, de constructores de tipo. Por ejemplo

```
data Boolean = True | False
```

define un tipo de datos booleano con constructores de aridad 0 (*constantes*); a continuación

```
data TreeInt = Leaf Int | Node (TreeInt) Int (TreeInt)
```

declara un tipo para el manejo de árboles binarios que contienen enteros; finalmente

```
data TreeP a = Leaf a    | Node (TreeP a )  a  (TreeP a)
```

define árboles binarios que contendrán cualquier tipo de dato (*tipos polimórficos*).

Las listas son un tipo preestablecido en Curry que se definen mediante la siguiente declaración:

```
data List a = [] | a :  List a
```

Aquí, ":" es un operador infijo, i.e., "`a: List a`" es una notación alterna para "`(:) a (List a)`". La notación "`[a]`" se usa para denotar listas de cualquier tipo en lugar de "`List a`". Además, Curry soporta la notación usual para listas, i.e., `[0,1,2]` es una abreviación para `0:(1:(2:[]))`.

Se pueden hacer *declaraciones de sinónimos de tipo*, del modo que sigue:

`type` $T\ \alpha_1\ \ldots\ \alpha_n$ `=` τ

que introduce T, un nuevo constructor de tipo n-ario. Aquí, $\alpha_1 \ldots \alpha_n$ son variables de tipo y τ es una expresión de tipo formada a partir de constructores de tipo y de las variables de tipo (no se admiten definiciones recursivas). Por ejemplo

```
type ListaInt  = [Int]
type Lista a   = [a]
```

declara el sinónimo `ListaInt` para referirse a una lista de enteros y `Lista` para referirse a una lista de elementos de cualquier tipo.

Un *término constructor* es una variable x o una aplicación de un constructor $c\ t_1 \ldots t_n$ donde c es un constructor de aridad n y $t_1 \ldots t_n$ son términos constructores. Una *expresión* es una variable o una aplicación (parcial) $\varphi\ e_1 \ldots e_m$ donde φ es una función o un constructor y $e_1 \ldots e_m$ son expresiones. Un término constructor se denomina *básico* si no contiene ninguna variable. Los términos constructores básicos corresponden a valores en el dominio definido; las expresiones que contienen operaciones se deben evaluar a términos constructores.

Las funciones se definen mediante una declaración de tipo (misma que se puede omitir), seguida por una secuencia de reglas (o ecuaciones). La declaración de una función tiene la forma:

f`::`τ_1 `->` τ_2 `->` `...` `->` τ_n `->` τ

donde $\tau_1 \ldots \tau_n, \tau$ son expresiones de tipo. La forma más simple de la definición de una regla o ecuación para una función f de aridad n es la siguiente:

$f\ t_1 \ldots t_n$ `=` e

Aquí, $t_1, \ldots, t_n$ son términos constructores y la *parte derecha* de la regla e es una *expresión*. Además, la *parte izquierda* de la regla no puede contener variables repetidas. Las funciones pueden estar definidas también, por *ecuaciones condicionales*, las cuales tienen la forma siguiente:

$f\ t_1 \ldots t_n$ `|` c `=` e

donde las condiciones (o *guardas*) c, pueden ser: una función que devuelve un booleano o una restricción. La *restricción ecuacional* $e_1 =:= e_2$ es una restricción elemental (devuelve el tipo `Success`) entre dos expresiones, la cual se satisface si ambas expresiones

se pueden reducir al mismo término constructor básico (i.e., *igualdad estricta* [MR92]).

En Curry es posible definir funciones con un comportamiento indeterminado mediante la introducción de varias reglas que se solapan por la parte izquierda. Por ejemplo la siguiente función *indeterminista*

```
insert x []     = []
insert x (y:ys) = x : y : ys
insert x (y:ys) = y : insert x ys
```

inserta un elemento en una posición arbitraria de una lista. Las funciones indeterministas en Curry no causan sobrecarga al lenguaje, puesto que las técnicas para su implementación están ya contenidas en la parte de programación lógica de Curry.

Las características de orden superior de Curry incluyen la aplicación parcial de funciones y las abstracciones lambda. La aplicación de funciones se denota por la yuxtaposición de la función y sus argumentos. Por ejemplo, la típica función `map` se define en Curry por:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Las funciones anónimas (abstracciones lambda) son expresiones que tienen la forma siguiente:

$\backslash \mathtt{x}_1 \ldots \mathtt{x}_n$ `->` exp

La aplicación de la función anónima sobre sus argumentos de entrada $e_1, \ldots, e_n$ produce el mismo resultado que la llamada a función "`foo` $e_1 \ldots e_n$", donde `foo` se define como:

`foo` $\mathtt{x}_1$ `...` $\mathtt{x}_n$ `=` exp

El lenguaje Curry es un lenguaje *perezoso*, por ejemplo, en el siguiente programa:

```
from x          = x : from (x + 1)
takeOne (y:ys) = y
one             = takeOne (from 0)
```

la función `one` devuelve el elemento `0`, porque se aplica la función `takeOne` que a su vez devuelve el elemento en cabeza de una lista, no se requiere la evaluación completa (infinita) de `from 0` antes de invocar a `takeOne`.

Además, Curry permite el uso de funciones que no están definidas en el programa del usuario, tales como operadores aritméticos, ciertas funciones de orden superior (`map`, `foldr`, etc.), operadores de entrada/salida de datos, etc. En el informe de Curry [Han03] se ofrece al lector información más detallada acerca de las características de este lenguaje declarativo multi-paradigma.

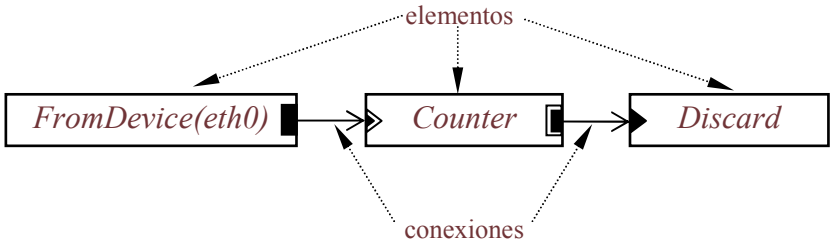


Figura 1.2: Un *router* Click.

1.2.3. El DSEL Rose

En esta sección presentamos el lenguaje empotrado en Curry de dominio específico Rose.

En ambientes de redes heterogéneos se requieren dispositivos para interconectar las diferentes tecnologías de comunicación que los forman. En Internet, dicho dispositivo es el *router*[4]. Básicamente, los *routers* conectan dos o más redes y reenvían paquetes de datos entre ellos. Así, la función primaria de un *router* es determinar el mejor camino para los paquetes en una red compleja. En principio, los *routers* han sido desarrollados como componentes hardware; sin embargo, hay una clara tendencia hacia la extensión del conjunto de funciones que las redes de *routers* deberían soportar. Estas nuevas funciones incluyen, e.g., filtrado de paquetes de datos, traducción de direcciones de Internet, monitorización, etc. La flexibilidad requerida para cubrir todas estas nuevas funciones motivó la definición de los llamados *routers extensibles* [GP02], los cuales permiten personalizar la funcionalidad del *router*.

Dentro de los *routers* extensibles, destacan los *routers* Click [KMC+00] que representan cada gránulo de funcionalidad de un *router* mediante los llamados "elementos" (objetos de clases escritas en C++); de esta manera, un *router* es un conjunto de instancias de elementos interconectados.

La funcionalidad del *router* que se muestra en la Figura 1.2—que se compone de tres elementos—es la siguiente: primero, se recogen paquetes de datos de la fuente de datos por medio del elemento `FromDevice(eth0)`, posteriormente se cuentan `(Counter)` y, finalmente, son eliminados `(Discard)`.

En [RSV03a, RSV03b, RSV04a] presentamos el lenguaje Rose para la especificación de *routers* Click. Rose simula las funciones de un *router* siguiendo el modelo Click. El enfoque que se adoptó fue el de construir Rose como un DSEL cuyo anfitrión es el lenguaje multi-paradigma Curry [Han03]. Así, es posible reutilizar las

[4] En la literatura, aún en castellano, es común referirse al término *router*—como haremos a partir de este punto—en lugar de su traducción literal, encaminador.

características y herramientas de un lenguaje declarativo muy expresivo.

La unidad de datos de un *router* es el paquete de datos, el cual es considerado aquí como una lista finita de *bytes* (codificados como enteros). Los flujos de datos que atiende el *router* son, cada uno, una lista potencialmente infinita de paquetes:

```
type Packet = [Int]
type Stream = [Packet]
```

El formato general de un elemento Rose es:

```
element :: [Conf] -> [Stream] -> [Stream]
```

Así, un elemento Rose recibe, por un lado, una lista con los argumentos de configuración y, por otro, un flujo de paquetes; la salida será un flujo de paquetes ya procesado.

Por ejemplo, un *router* simple formulado en el lenguaje Rose a partir del caso de la Figura 1.2 es el siguiente:

```
simpR = seqOfe [fromDevice [Eth 0], Counter [], Discard []]
```

Este *router* está constituido por una secuencia de elementos (incluyendo los parámetros de configuración) enlazados a partir de la función conectora `seqOfe`, que compone elementos en secuencia:

```
seqOfe :: [ [Stream] -> [Stream] ] -> [Stream] -> [Stream]
seqOfe [] = id
seqOfe (elem : es) = \input -> seqOfe es (elem input)
```

La función `seqOfe` toma una lista de elementos (con sus parámetros de configuración) y un flujo de paquetes (`input`), el cual es aplicado al primer elemento y, la salida de éste, se pasa al resto de elementos recursivamente.

El DSEL Rose, al igual que el código de cualquier DSEL, produce generalmente código ineficiente (ver sección 1.1.2) debido entre otras cosas a la sobrecarga de interpretación [SCK04]. Una aplicación escrita a partir del DSEL Rose, es decir una especificación de un *router*, será interpretada por Rose y el código resultante será interpretado nuevamente, pero ahora por el intérprete del lenguaje anfitrión Curry. La sobrecarga de interpretación se podría reducir considerablemente mediante una técnica bien conocida en el campo de la evaluación parcial: la especialización de intérpretes [Jon04].

1.3. Evaluación parcial de programas

La evaluación parcial de programas es una técnica formal para la transformación de programas útil, por ejemplo, para mejorar la eficiencia de los lenguajes de dominio

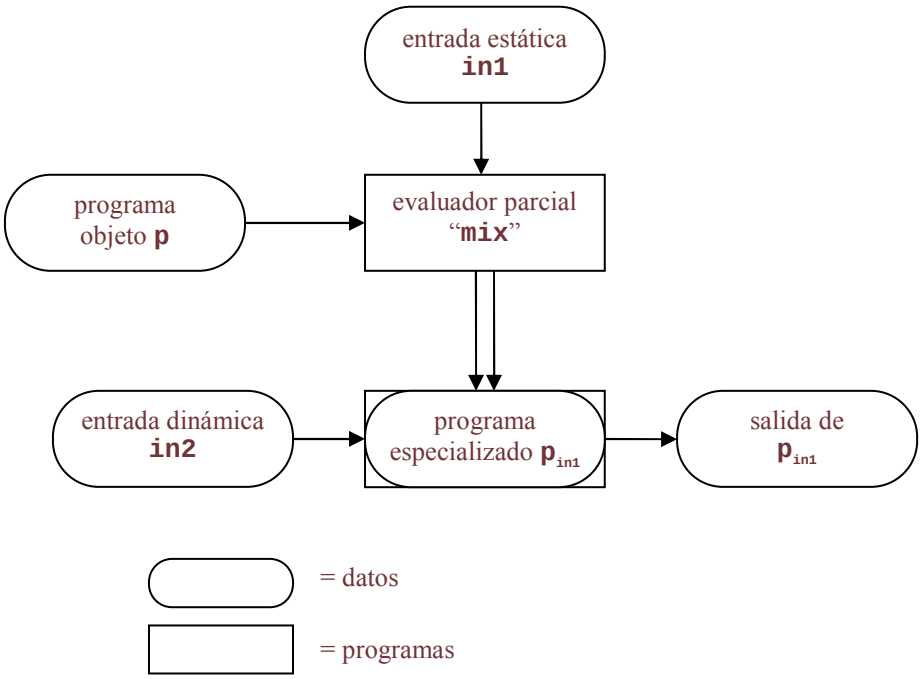


Figura 1.3: El proceso de evaluación parcial [JGS93].

específico.

Si consideramos una función que toma dos argumentos de entrada, dicha función puede *especializarse* para uno de sus argumentos. El nombre que recibe esta operación en análisis matemático es el de "restricción" o "proyección". Sin embargo, la evaluación parcial trata con programas en lugar de funciones [JGS93], como veremos en los ejemplos siguientes.

Dados un programa `p` y sus datos de entrada: `in1` e `in2`, el evaluador parcial recibe el programa `p`, junto con parte de sus datos de entrada, e.g. `in1`, los llamados datos *estáticos*. El evaluador parcial produce como salida un nuevo programa $\texttt{p}_{\texttt{in1}}$, el cual, al ejecutarse junto con su segundo argumento `in2` (inicialmente desconocido —dato *dinámico*), produce el mismo resultado que `p` habría computado con ambos datos. En otras palabras, un evaluador parcial es un especializador de programas. Este proceso se ilustra en la Figura 1.3, donde el evaluador parcial recibe el nombre de `mix`.

A continuación se muestra un ejemplo de un programa `p` que implementa una función para computar x^n, i.e., la potencia `n` de `x`:

```
p   ≡   f n x = if n == 1 then x
```

```
                                    else x * (f (n - 1) x)
```

A continuación, vamos a especializar el programa `p` para `n=3`; la llamada inicial será por tanto `(f 3 x)`. El resultado es el siguiente programa:

```
p_3 ≡  f_3 x = x * x * x
```

el cual es más eficiente que el original. La transformación consiste en precomputar todas las expresiones en las que aparece el dato estático (`n`) y desplegar todas las llamadas recursivas a la función `f`. Esta optimización arroja buenos resultados porque el control del programa está completamente determinado por `n`. Si, por otro lado, el dato estático hubiese sido `x=3` y `n` hubiese sido dinámico, la especialización no daría mejoras significativas en el tiempo de ejecución del nuevo programa.

En general, el proceso de especialización se realiza por medio de la ejecución de aquellas computaciones que dependen sólo de los datos estáticos, generando código residual para las computaciones que dependen de los datos dinámicos. Así, un evaluador parcial realiza una mezcla de acciones de ejecución y generación de código. Ershov [EO87] le llamó al proceso "*mixed computation*", lo que dió origen al término `mix` (con el que habitualmente se denominan a los evaluadores parciales).

La evaluación parcial está compuesta por tres técnicas fundamentales: computación simbólica, que consiste en realizar computaciones cuando se tiene sólo el nombre del dato y no el dato mismo, desplegado de llamadas a función y especialización de puntos de programa; esta última es equivalente a la técnica de memorización de puntos del programa. La memorización registra los puntos del programa que han sido evaluados parcialmente de manera que, al encontrar un término "similar" a otro previamente procesado, entonces ya no se procede con la computación.

1.4. Evaluación parcial de programas lógico funcionales

La semántica operacional estándar de los lenguajes declarativos integrados está basada en el mecanismo de *narrowing* [Sla74], el cual es también la base para conducir la evaluación parcial de programas lógico funcionales en la aproximación de [AV02].

La evaluación parcial dirigida por *narrowing* (NPE: *Narrowing-driven Partial Evaluation*) es una técnica potente para la especialización de sistemas de reescritura [AV02], i.e., para el componente de primer orden de muchos lenguajes declarativos (lógico) funcionales como Haskell [PJ03], Curry [Han03] o Toy [LFSH99]. El entorno de desarrollo del lenguaje Curry, PAKCS (Portland-Aachen-Kiel Curry *System* [HeAE+04]), cuenta actualmente con un evaluador parcial desarrollado a partir del esquema NPE. En [AHV02] se puede encontrar una evaluación experimental del

mismo.

A continuación, presentamos una panorámica del estado del arte en la evaluación parcial dirigida por *narrowing*.

1.4.1. Antecedentes

La evaluación parcial tradicional tiene como fin especializar programas con respecto a datos conocidos; sin embargo, algunas técnicas van más allá de este objetivo y son capaces de realizar optimizaciones de programas aún más potentes, inclusive cuando no existen datos conocidos (e.g., para eliminar estructuras de datos intermedias, de manera similar a la deforestación de Wadler [Wad90]). Tal es el caso de diversos métodos de evaluación parcial como la supercompilación positiva (Srensen et al. [SGJ96]) para programas funcionales; la deducción parcial (Lloyd y Shepherdson [LS91]) para programas lógicos y la evaluación parcial dirigida por *narrowing* (Alpuente et al. [AFV98]) para programas lógico funcionales.

La evaluación parcial ha sido estudiada profusamente tanto en el paradigma lógico como en el funcional. En esencia, las aproximaciones en ambos paradigmas son similares; sin embargo, los métodos generales difieren con frecuencia debido a los distintos modelos que subyacen y también a las diferentes perspectivas desde las que son enfocados. Con la finalidad de evitar la duplicación de esfuerzos, se han desarrollado aproximaciones que generalizan distintas técnicas de evaluación parcial. Por ejemplo, Pettorossi y Proietti [PP96] presentan una metodología general, la cual puede verse como un patrón común para razonar acerca de la especialización de programas lógicos y funcionales; también establecen correspondencias precisas entre las diferentes técnicas. Jones [Jon94] presenta un marco abstracto para describir la especialización de programas. Tanto la evaluación parcial de programas lógicos (Lloyd [Llo94]) como el mecanismo de *driving* de Srensen y Glück [SG95] se presentan como instancias del nuevo marco.

Más recientemente, Alpuente et al. [AFV98] introdujeron un esquema general para la especialización de programas lógico funcionales. El mecanismo de *narrowing* se utiliza tanto para ejecutar programas lógico funcionales como para la evaluación parcial. La dimensión lógica del mecanismo de *narrowing* permite tratar con expresiones que contienen información parcial (por medio de variables lógicas y unificación); por otro lado, la dimensión funcional permite considerar estrategias de evaluación eficientes.

El método NPE está formalizado dentro del marco teórico establecido por Lloyd y Shepherdson [LS91] para la evaluación parcial de programas lógicos. Sin embargo, algunas nociones se han generalizado con la finalidad de tratar con características tales como llamadas a función anidadas, el uso de estrategias de evaluación *perezosa* o *impaciente*, etc. Los aspectos de control se abordan por medio del uso de técnicas

estándar, como las desarrolladas por Martens y Gallagher [MG95], Srensen y Glück [SG95], etc. Además, Alpuente et al. [AFV98] hacen una distinción clara entre los aspectos de *control local* y *control global* (de forma similar a [MG95]). Informalmente, el control local tiene que ver con la construcción de árboles de *narrowing* parciales para términos independientes, mientras que el control global se encarga del *cierre* de los programas especializados. La condición de cierre garantiza que las llamadas a función que podrían ocurrir durante la ejecución de un programa especializado estén cubiertas por alguna regla del mismo [LS91]. Con esto, se asegura la corrección del proceso de especialización.

A grandes rasgos, el algoritmo de especialización instrumentado por NPE comienza con la evaluación parcial del conjunto de las llamadas a función que aparecen en el objetivo inicial y, entonces, especializa de manera recursiva todos los términos generados dinámicamente durante el proceso. El control local incorpora una estrategia de desplegado, en tanto que el control global provee un operador de generalización para asegurar la terminación global del proceso. En suma, el marco define un método de evaluación parcial automático, el cual es independiente de la estrategia de *narrowing*, finito (para instancias apropiadas) y que asegura la condición de cierre para los programas especializados (residuales). Siguiendo la terminología de Glück y Sorensen [GS96], decimos que la estrategia de control global puede producir especializaciones *polivariantes* y *poligenéticas*, i.e., NPE es capaz de generar diferentes versiones especializadas a partir de una definición de función y, por otro lado, puede combinar en una sola función especializada varias definiciones de función originalmente distintas.

En [Vid96, AFV98] se plantea con mayor detalle las bases del marco. En concreto, se presenta la formulación de un método genérico de evaluación parcial de programas lógico funcionales basado en el uso del propio mecanismo operacional de *narrowing*. El método es paramétrico con respecto a la relación de *narrowing*, la estrategia de desplegado y el operador de generalización. En el documento se demuestra la corrección, completitud y el carácter finito de la técnica.

En [AFJV97, Jul00] se formula una instancia del algoritmo genérico de NPE [AFV98] basada en el empleo de *narrowing* perezoso (Moreno-Navarro y Rodríguez-Artalejo [MR92]). El proceso de evaluación parcial lo formalizan en dos fases. En la primera, se aplica una instancia concreta del método de NPE y, a continuación, se introduce una fase de renombramiento que es necesaria para conseguir recuperar la disciplina de constructores (reglas sin funciones anidadas y sin variables repetidas en la parte izquierda). El postproceso de renombramiento también es necesario para lograr la llamada condición de independencia del conjunto de términos evaluados parcialmente, una condición indispensable para garantizar que el programa transformado no produce respuestas adicionales y por lo tanto indeseadas. En [Jul00] se introducen mejoras a los mecanismos de control mediante una regla de desplegado dinámica y se

introducen técnicas de partición similares a las de [GJMS96, LDdW96] de los términos evaluados parcialmente para evitar una acción de generalización excesiva a nivel global.

Alpuente et al. [AHLV99] introducen una instancia del método NPE para programas inductivamente secuenciales basada en la estrategia de *narrowing* necesario (Antoy *et al.* [AEH00]). El método traslada al esquema de evaluación parcial la idea de evaluar código sólo cuando es necesario. Además, esta instancia preserva la estructura del programa original, i.e., el programa residual es también inductivamente secuencial, propiedad que no se cumple en general para otras instancias del marco NPE (ver [AHLV99]).

Albert et al. [AAHV99a, AAHV99b] definen un marco de evaluación parcial para programas lógico funcionales con residuación. Antes, formularon operadores de control para especializar programas que incluyeran símbolos de función primitivos [AAF+98a, AAF+98b].

En [AV02] aparece un compendio de las propiedades y conceptos del esquema NPE: cierre, resultante, renombramiento, control local, control global, etc. así como el algoritmo utilizado en el proceso de la especialización. En [AHLV05] se presenta el marco formal del esquema de evaluación parcial dirigido por *narrowing* necesario. Se introducen y demuestran formalmente las propiedades del esquema para especializar programas inductivamente secuenciales, e.g., corrección, independencia, cierre, etc. En el Capítulo 3 presentamos un resumen de ello.

La evaluación parcial de programas guarda una estrecha relación con otras técnicas de transformación de programas. En particular, en [AFMV00] se estudia la relación entre la evaluación parcial NPE y la técnica de transformación basada en plegado/desplegado (*folding/unfolding*, ampliamente tratada en [Mor00] para programas lógico funcionales).

La transformación por plegado/desplegado se introdujo por primera vez en [BD77] para la optimización de programas funcionales. La aproximación se basa, usualmente, en la construcción de una secuencia de programas equivalentes (por medio de una "estrategia"), cada programa se obtiene a partir de otros precedentes mediante el uso de una regla de transformación "elemental". Las reglas esenciales son *plegado* y *desplegado*, i.e., la contracción (reemplazamiento de cierto fragmento de código por su llamada a función) y la expansión (el reemplazo de una llamada a función por su definición aplicando la correspondiente sustitución) de subexpresiones de un programa, empleando para ello las definiciones del propio programa o de alguno precedente. En [AFMV99] introdujeron una metodología de transformación para programas lógico funcionales perezosos, posteriormente, en [AFMV00] extendieron las reglas de transformación para programas lógicos de [TS84]. Tal extensión permitió procesar programas lógico funcionales perezosos basados en *narrowing* necesario; también, de-

mostraron que las transformaciones por plegado/desplegado son capaces de conseguir los efectos de la evaluación parcial y, finalmente; propusieron un algoritmo composicional automático (basado en plegado/desplegado) el cual efectúa un pre-proceso que se basa en evaluación parcial para generar las *eurekas* (conjuntos de ecuaciones que se añaden a un programa para obtener una transformación eficiente del mismo) lo que garantiza que se alcanzará la optimización buscada. Es interesante notar que la terminación global del algoritmo se asegura por medio de la condición de cierre, definida originalmente para el marco NPE.

El desarrollo de un esquema de evaluación parcial para programas lógico funcionales realistas requiere el tratamiento de características avanzadas, tales como: orden superior, restricciones, llamadas a funciones externas, etc. Para tratar con tales características se requeriría un cálculo operacional muy complejo. En [HP99] se introduce una representación abstracta para programas en la que los árboles definicionales [Ant92] (usados para guiar la estrategia de *narrowing* necesario) son hechos explícitos por medio de construcciones *case*. También, se formula una semántica operacional para esos programas: el cálculo LNT (del inglés *Lazy Narrowing with definitional Trees*). En [AHV00a, AHV00b] se introduce un marco de trabajo en el que los programas de alto nivel son traducidos a la representación abstracta de programas y se incluye una extensión residualizante del cálculo LNT, i.e., el cálculo RLNT (*Residualizing LNT*). A partir del nuevo cálculo es posible implementar un evaluador parcial realista para programas en representación abstracta. En [AHV03] se demuestra la equivalencia entre el cálculo LNT y RLNT. En [AHV01] y [AHV02] se describe el esquema práctico de evaluación parcial dirigido por *narrowing* para programas abstractos (traducidos a partir de programas Curry) y la forma en que se resuelven las características extendidas del lenguaje: guardas, restricciones, funciones externas, orden superior, etc.

Con la idea de evaluar la mejora del proceso de evaluación parcial en los programas transformados en [AAV00, AAV01] se define un marco formal para medir la efectividad del proceso de evaluación parcial de programas lógico funcionales. Se introduce una serie de criterios: número de pasos de reducción, número de aplicaciones de función y el esfuerzo en el emparejamiento de patrones o en la unificación. Más tarde, en [Vid02, Vid04] se agregan criterios relacionados con el orden superior y con el indeterminismo; se modifican las semánticas LNT y RLNT para incluir costes y se desarrolla un nuevo evaluador parcial NPE. La nueva herramienta soporta los principios básicos de los programas lógico funcionales: *narrowing* y residuación e informa de la mejora conseguida en los programas especializados. De esta manera, se pueden relacionar el coste de ejecutar el programa residual con respecto al original.

Finalmente, los trabajos más recientes precedentes a esta tesis relacionan la evaluación parcial de programas lógico funcionales con la técnica de *slicing*. La técnica de *slicing* [Wei79, Wei81] de programas es un método para descomponer programas

en trozos (*slices*) mediante el análisis de sus datos y del control de flujo. Un trozo de un programa se compone de aquellas sentencias del programa que están relacionadas (potencialmente) con los valores computados en un punto del programa o para una variable (lo cual recibe el nombre de: criterio de *slicing*). Sus aplicaciones están relacionadas con la depuración, mantenimiento, prueba y reuso de código. En [Vid03, SV07] se introduce una técnica para obtener trozos de un programa a partir de un programa especializado mediante el esquema NPE. Posteriormente [OSV04a, OSV04c, OSV05] plantean una aproximación ligera de especialización en la que parten de una técnica de *slicing*[5] que les permite extraer trozos de un programa, a los que se les aplica una transformación para hacerlos ejecutables, y así, obtener un programa especializado.

De manera concurrente a los trabajos de evaluación parcial dirigida por *narrowing*, Lafave y Gallagher [Laf98, LG97] presentaron un marco teórico de evaluación parcial para programas lógico funcionales (en particular, se trata de programas Escher [Llo95]). Tales programas son procesados por un modelo computacional basado en reescritura. También formalizaron un algoritmo automático de evaluación parcial a partir del marco, que utiliza restricciones para representar información asociada a las expresiones de los programas. El evaluador parcial utiliza la información aportada por las restricciones para tomar decisiones de especialización de manera *online*.

El mecanismo de computación de los lenguajes lógico funcionales basados en reescritura permite la simplificación de términos parcialmente instanciados, pero requiere cierto "nivel de instanciación" para poder evaluar un término. En el evaluador parcial de [Laf98, LG97] se extiende el mecanismo computacional del lenguaje objeto con un *paso de reinicio* para procesar términos que no están suficientemente instanciados. Cuando hay un término que requiere instanciaciones la computación se *reinicia* mediante el reemplazo del último término por otro con las instanciaciones mínimas para que pueda computarse.

El paso de reinicio contrasta con el método NPE, donde el propio mecanismo de computación de *narrowing* es capaz de hacer las instanciaciones necesarias para proceder con las computaciones. [Laf98, LG97] distinguen también dos niveles de control, el control local que verifica si un término a computar es válido de acuerdo a un orden definido y, por otro lado, el nivel global asociado al paso de reinicio. En este esquema, los términos que se computan parcialmente se colocan en una estructura en forma de árbol denominada "árbol-m" (*m-tree* en el original). Así, el crecimiento del árbol se observa tanto por el control local como por el global, el cual controla que se realice sólamente un numero finito de pasos de reinicio para cada rama.

De las dos aproximaciones de evaluación parcial para programas lógico funcionales,

[5] Las técnicas de *slicing* están montadas sobre métodos para traceado (*tracing*) de programas [OSV04b], las cuales se basan en una estructura de datos que registra los pasos de ejecución de un programa y sus datos: *redex trails* [SR97].

ha sido NPE la que se ha desarrollado en mayor medida hasta la fecha.

1.4.2. Evaluación parcial dirigida por *narrowing*

A continuación, ilustramos a grandes rasgos la extracción de un programa especializado a partir de un conjunto de computaciones de *narrowing*. La entrada para el evaluador parcial es un sistema de reescritura (un típico programa funcional de primer orden) y una llamada a función inicial, la cual suele contener algunos datos conocidos (datos estáticos).

Ejemplo 1.1 *Consideremos el siguiente programa:*

```
inc x       =  add (S Z) x
add Z y     =  y
add (S x) y =  S (add x y)
```

donde se define la operación incremento `inc` *la cual invoca a la función* `add`*, que implementa la suma sobre números naturales formados a partir de los constructores* `Z` *(cero) y* `S` *(sucesor). Podemos evaluar parcialmente la llamada a* `inc x` *con respecto al programa para obtener una definición directa de la función* `inc` *(i.e., especializando la función* `add` *que cuenta con un argumento estático* `S Z`*).*

El evaluador parcial debe construir alguna forma de "árbol de ejecución simbólica". Decimos "simbólica" porque los términos pueden contener variables y decimos "árbol" porque la instanciación indeterminista de las variables libres puede producir más de una computación alternativa.

La construcción de tales árboles de ejecución simbólica es explícita en algunas técnicas de evaluación parcial (e.g., en la evaluación parcial dirigida por *narrowing* [AV02]). En otras técnicas, la construcción de un árbol de ejecución simbólica es solo implícita; por ejemplo, muchos evaluadores parciales para programas funcionales (ver, e.g., [JGS93]) incluyen un algoritmo que aplica iterativamente los siguientes pasos: (1) selecciona una llamada a función, (2) realiza una serie de ejecuciones simbólicas, y (3) a partir de las expresiones evaluadas parcialmente extrae el conjunto de llamadas a función pendientes (los sucesores de la llamada a función inicial) para procesarse en la siguiente iteración del algoritmo. Obsérvese que si se agrega una flecha saliendo de cada término a su conjunto de sucesores obtendríamos algo muy similar al citado árbol de ejecución simbólica.

Para realizar computaciones simbólicas, la elección del mecanismo de *narrowing* surge de manera muy natural, ya que combina reducciones funcionales con la instanciación de variables libres. En la Figura 1.4 se muestra el árbol de ejecución simbólica

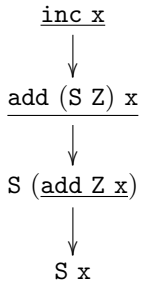


Figura 1.4: Árbol de ejecución simbólica del Ejemplo 1.1.

para la llamada inicial `inc x` del Ejemplo 1.1 (en cada paso, aparece subrayada la función que se selecciona para su evaluación).

Como podemos observar en la Figura 1.4, no fue necesaria la instanciación de variables libres, por lo que obtenemos una evaluación totalmente determinista. A partir de las computaciones del árbol se puede obtener el programa especializado (*residual*). Para ello, por cada *paso de narrowing*[6] que computa un término t a partir de un término s en el árbol mencionado, se extrae una regla residual: $\sigma(s) = t$, donde σ es la sustitución computada en el paso de *narrowing*. En el Ejemplo 1.1 obtenemos las siguientes reglas residuales:

```
inc x       =  add (S Z) x
add (S Z) x =  S (add Z x)
add Z x     =  x
```

Sin embargo, las reglas residuales no son necesariamente reglas de programa válidas, por lo que se requiere un post-proceso de renombramiento y de simplificación de reglas [AFJV97, AFV98]. El resultado del renombramiento es en este caso:

```
inc x  =  add1 x
add1 x =  S (add0 x)
add0 x =  x
```

Finalmente, una simplificación sencilla de las llamadas a función genera el programa residual mostrado a continuación:

```
inc x  =  S x
```

[6] Un paso de *narrowing* se representa en símbolos por $s \leadsto_\sigma t$, los cuales describen la computación de un término t a partir de un término s aplicando una sustitución σ al término s para hacer posible el paso de reducción. Para una explicación formal ver la Sección 2.4.1.

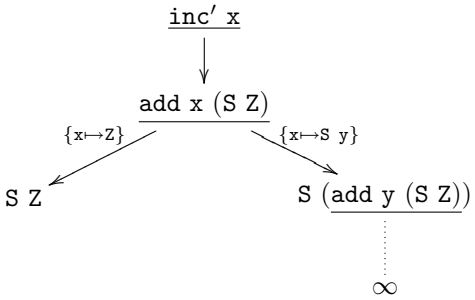


Figura 1.5: Árbol de ejecución simbólica del Ejemplo 1.2.

1.4.3. El problema de la terminación

En el Ejemplo 1.1 observamos que se produce un conjunto finito de computaciones que conduce a un resultado; naturalmente, no siempre es así. Sin embargo, en los casos en que aparece alguna computación para evaluarse, de manera iterativa, que es igual (módulo renombramiento de variables) a otra anterior, se puede evitar la evaluación repetida del término. Para ello, los evaluadores parciales incluyen usualmente una técnica denominada *memoization* que memoriza los términos computados. A continuación mostramos un ejemplo.

Ejemplo 1.2 *Consideremos una nueva definición para la función incremento del Ejemplo 1.1:*

$$\texttt{inc}'\ \texttt{x} \quad = \quad \texttt{add x (S Z)}$$

donde el primer argumento de la llamada a la función `add` *es una variable libre.*

Aunque el árbol de ejecución simbólica para `inc′ x` *(Figura 1.5) es infinito, un evaluador parcial terminaría ya que la llamada a función* `add y (S Z)` *es igual a* `add x (S Z)` *módulo renombramiento de variables.*

El programa residual asociado a la ejecución parcial que aparece en la Figura 1.5, es el siguiente:

$$\begin{aligned} \texttt{inc}'\ \texttt{x} &= \texttt{add x (S Z)} \\ \texttt{add Z (S Z)} &= \texttt{S Z} \\ \texttt{add (S y) (S Z)} &= \texttt{S (add y (S Z))} \end{aligned}$$

En este caso, tenemos una regla residual asociada al primer paso de la derivación y dos reglas que se corresponden, cada una, con un paso indeterminista.

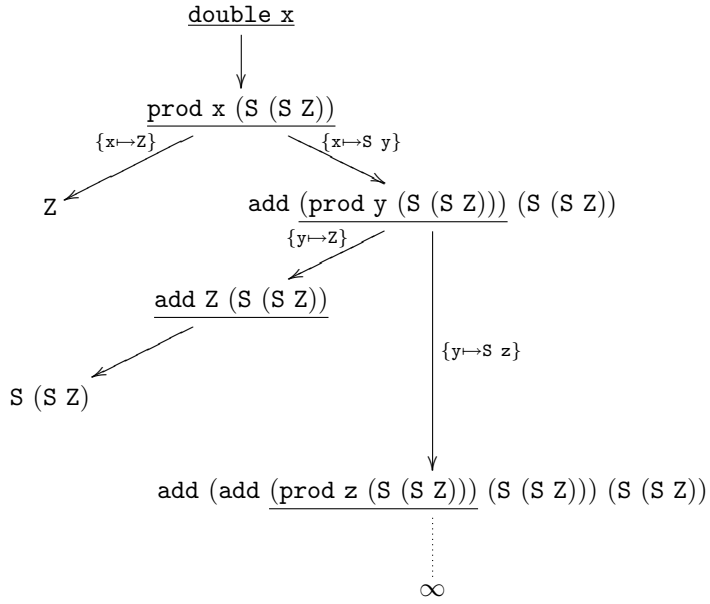


Figura 1.6: Árbol de ejecución simbólica del Ejemplo 1.3.

La terminación del árbol de ejecución simbólica se puede garantizar cuando las computaciones simbólicas son *cuasi-terminantes* [Der87], i.e., cuando se obtiene un conjunto finito de llamadas a función diferentes módulo renombramiento de variables. De hecho, las computaciones del árbol de la Figura 1.5 son cuasi-terminantes. Sin embargo, en general el mecanismo de ejecución simbólica puede dar lugar a computaciones tanto no terminantes como no cuasi-terminantes.

Ejemplo 1.3 *Consideremos el siguiente programa:*

$$\begin{array}{rcl} \texttt{double x} & = & \texttt{prod x (S (S Z))} \\ \texttt{prod Z y} & = & \texttt{Z} \\ \texttt{prod (S x) y} & = & \texttt{add (prod x y) y} \end{array}$$

donde `double` *invoca el doble producto del argumento expresado en números naturales.*

Dada la llamada a función inicial `double x`*, el árbol de ejecución simbólica que se computa es infinito (Figura 1.6).*

Una alternativa para asegurar el carácter finito del árbol de ejecución simbólica es la *generalización* de los términos que producen ramas infinitas. A grandes rasgos,

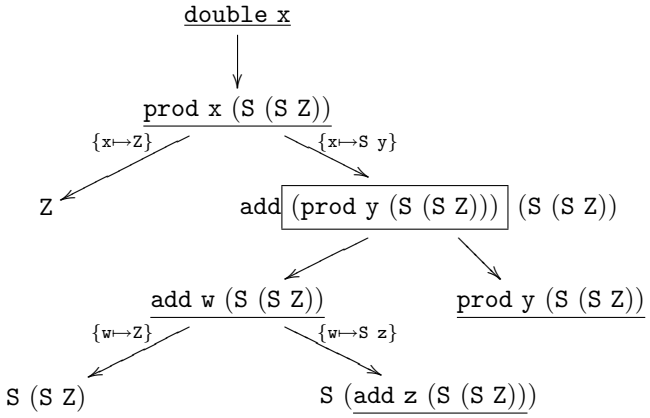


Figura 1.7: Árbol de ejecución simbólica del Ejemplo 1.3 aplicando generalización.

la generalización sustituye aquellos subtérminos que producirían ramas infinitas por variables nuevas, de manera que, al reiniciarse la computación con el término generalizado, ya no hay peligro de que aparezcan cómputos infinitos. La selección de términos a generalizar se puede decidir en una etapa de pre-proceso (evaluación parcial *offline*) o durante la propia evaluación parcial (evaluación parcial *online*).

En el árbol de la Figura 1.6, la terminación se puede garantizar mediante la generalización de la segunda llamada a la función `prod` como se muestra en la Figura 1.7 (el subtérmino generalizado aparece en un recuadro).

En las computaciones de la Figura 1.7, el árbol de ejecución simbólica se mantiene finito debido a que todas las hojas son valores (i.e., no contienen llamadas a función), e.g., "`Z`" y "`S (S Z)`", o contienen una llamada a función que es igual (módulo renombramiento de variables) a una llamada previa en el árbol, el caso de "`prod y (S (S Z))`" y "`add z (S (S Z))`", los cuales son variantes de "`prod x (S (S Z))`" y "`add w (S (S Z))`", respectivamente.

Del árbol de ejecución simbólica de la Figura 1.7 se puede extraer el siguiente programa residual:

```
      double x  =  prod x (S (S Z))
  prod Z (S (S Z))  =  Z
prod (S y) (S (S Z))  =  add (prod y (S (S Z))) (S (S Z))
    add Z (S (S Z))  =  S (S Z)
add (S z) (S (S Z))  =  S (add z (S (S Z)))
```

Hasta aquí, se ha ilustrado como opera el mecanismo de computación simbólica, *narrowing*, para construir el árbol de computaciones y cómo éste se puede usar para extraer el programa especializado. Estas nociones se aprovechan en la tesis para la formulación de un nuevo método de evaluación parcial.

1.5. Objetivos de la tesis

A continuación presentamos el planteamiento metodológico de la memoria y la estructura del mismo.

1.5.1. Planteamiento

Los evaluadores parciales se clasifican en dos grandes categorías: *online* y *offline*, de acuerdo al momento temporal en que se consideran los aspectos de terminación. Los evaluadores parciales *online* son usualmente más precisos ya que tienen más información disponible. En particular, el esquema original NPE, el cual sigue la aproximación *online*, considera una variante del teorema de Kruskal (*Kruskal's Tree Theorem*) llamada *"subsumción homeomórfica"*[7] [Leu02], para asegurar la terminación del proceso: si un término *subsume* algún término previo en la misma computación de *narrowing*, se aplica alguna forma de generalización —usualmente el operador de generalización más específica— y la evaluación parcial se reinicia con los términos generalizados [AFV98]. Sin embargo, esta precisión adicional tiene también un coste asociado: las pruebas para el test de subsumción, junto con las generalizaciones asociadas, hacen que el esquema NPE *online* sea muy costoso (en términos de tiempo y espacio), por lo que no se adapta adecuadamente a problemas realistas como la especialización de intérpretes [Jon04].

Los evaluadores parciales *offline* proceden usualmente en dos etapas; la primera etapa procesa un programa e incluye anotaciones para guiar las computaciones parciales (e.g., para identificar aquellas llamadas a función que pueden desplegarse sin riesgo de no terminación); entonces, una segunda etapa, la de evaluación parcial propiamente dicha, sólo debe obedecer las anotaciones y por tanto el especializador es mucho más rápido que en la aproximación *online*.

1.5.2. Objetivo general

El objetivo general de esta tesis consiste en desarrollar los fundamentos de una aproximación *offline* a la evaluación parcial dirigida por *narrowing*. Ello nos permitirá afrontar la evaluación parcial de las aplicaciones escritas en un DSEL como Rose.

[7] Del inglés *homeomorphic embedding*. Esta relación se introduce en la Sección 3.6.1.

1.5.3. Contribuciones

Las principales contribuciones de esta tesis son las siguientes:

Condiciones para la cuasi-terminación de las computaciones por *narrowing* necesario. En este sentido, hemos presentado una caracterización de programas cuasi-terminantes llamados "no crecientes" en la que las computaciones por *narrowing* necesario presentan una secuencia finita de llamadas a función diferentes (módulo renombramiento de variables). La propiedad de *cuasi-terminación* en las computaciones de los programas es muy útil para el análisis y transformación de los mismos. Los resultados en esta línea han sido publicados en: [RSV05b, RSV05c, RSV05d, RSV07].

Aproximación *offline* a la NPE. Uno de los aspectos cruciales de la evaluación parcial es asegurar que el proceso de especialización termine. En este sentido, la cuasi-terminación es importante toda vez que su presencia es frecuentemente una condición suficiente para la terminación del proceso [Hol91]. En [RSV05b] formulamos un esquema de evaluación parcial dirigido por *narrowing* más eficiente y que asegura la terminación siguiendo el estilo *offline*. Con ese fin, partimos de la caracterización de programas cuasi-terminantes. Sin embargo, la clase de programas es muy restrictiva, por lo que planteamos un algoritmo que acepta programas *inductivamente secuenciales* —una clase mucho más amplia sobre la que está definido el *narrowing* necesario— y devuelve un programa anotado. A continuación, planteamos la necesidad de una relación de *narrowing* extendida: *narrowing* generalizante, que trate de forma apropiada los términos anotados; finalmente, incorporamos la extensión de *narrowing* generalizante al esquema NPE. Los aspectos relacionados con esta línea han sido publicados en: [RSV05b, RSV05c].

Adaptación del principio de terminación por grafos *size-change*. Recientemente, se ha formulado un nuevo principio para el análisis de terminación de los programas basado en el cambio de tamaño de los argumentos de las funciones [LJBA01, GJ05]. Con la finalidad de mejorar la precisión de la fase de anotación propia del esquema *offline* de evaluación parcial, hemos adaptado los grafos de cambio de tamaño (grafos *size-change*), introducidos originalmente para los lenguajes funcionales, a los lenguajes lógico funcionales. En particular los grafos nos son útiles para determinar una forma particular de cuasi-terminación, que finalmente utilizamos para realizar anotaciones al estilo de nuestra propuesta original [RSV05b]. Estos resultados han sido publicados en [ARSV06].

Desarrollo de herramientas. Hemos desarrollado un prototipo para la evaluación parcial *offline* dirigida por *narrowing*. La validación experimental del prototipo arroja resultados que muestran una mejora significativa en el tiempo de especialización *offline* con respecto al *online*. De forma muy resumida, el tiempo medio de especialización *offline* es el 20 % del tiempo medio registrado con la herramienta *online*. Sin embargo, los programas especializados son un 5 % más lentos con respecto a los programas especializados siguiendo el estilo *online*. Los detalles de la implementación de esta herramienta se encuentran en [RSV05a].

Lenguajes de domino específico empotrado. Los DSELs generan código ineficiente debido a la sobrecarga de interpretación, por lo que [Hud98, SCK04, Chr03, HM04] entre otros, plantean el uso de las técnicas de evaluación parcial como un medio para remediar este inconveniente. En este punto, seleccionamos el dominio de los *routers*, en particular el modelo de *routers software Click* [KMC+00]. A continuación desarrollamos un lenguaje de especificación para *routers* denominado *Rose*, empotrado en el lenguaje Curry. Los DSELs aportan una vía de prueba interesante en dos aspectos: primero, permiten ejercitar los conceptos de la programación lógico funcional en la generación de lenguajes de dominio específico; segundo, proporcionan ejemplos para la especialización de programas mediante las técnicas NPE *offline* desarrolladas. Presentamos estos avances en [RSV03b, RSV03a, RSV04a, RSV04b, RAS06].

1.6. Organización de la memoria

Esta tesis ha sido estructurada en nueve capítulos. En el Capítulo 2 presentamos un resumen de los fundamentos técnicos que sirven de base para formalizar los conceptos empleados en el resto del documento. A continuación en el Capítulo 3, presentamos un resumen de los resultados más importantes en el marco de la evaluación parcial dirigida por *narrowing* previos al presente trabajo.

En el Capítulo 4, formulamos una caracterización de los programas gracias a la cual se obtienen computaciones cuasi-terminantes con respecto al mecanismo de computación de *narrowing* necesario. Dado que la caracterización sintáctica da lugar a una clase de programas muy limitada, se introduce un pre-proceso que anota aquellos términos que no cumplen la caracterización. Esta etapa es incorporada de manera *offline* en el contexto de la evaluación parcial dirigida por *narrowing*. En el Capítulo 5, exponemos las ventajas para la meta-programación del lenguaje lógico funcional Curry y ejemplificamos cómo se anotan y procesan los programas durante la especialización. A continuación, damos una descripción de la herramienta de especialización e introducimos los aspectos de implementación inherentes a la nueva herramienta de

evaluación parcial *offline*.

Enseguida discutimos (en el Capítulo 6) la adaptación del método para el análisis de terminación por grafos *size-change* que aproximan el cambio de tamaño de los argumentos de las funciones, al esquema *offline* presentado en el Capítulo 4.

En el Capítulo 7, nos centramos en la exposición del lenguaje Rose (para la especificación de *routers*), el cual es un lenguaje de dominio específico empotrado en el lenguaje Curry. Introducimos la librería de elementos, algunos ejemplos representativos y una evaluación experimental. Después, en el Capítulo 8, describimos experimentos para ilustrar la aplicación de la evaluación parcial *offline* dirigida por *narrowing*.

Finalmente, en el Capítulo 9 mostramos un resumen de las aportaciones de esta investigación y planteamos una serie de líneas de trabajo futuro relacionadas, sobre todo, con la aproximación *offline* a la evaluación parcial.

Capítulo 2

Preliminares

En el presente capítulo introducimos los conceptos fundamentales de la programación lógico funcional que se requieren en el resto de la tesis. Tras plantear los conceptos básicos de signaturas, términos (Sección 2.1) y sustituciones (Sección 2.2), presentamos una serie de nociones y definiciones relacionas con los sistemas de reescritura de términos (Sección 2.3). Posteriormente nos centramos en la introducción del paradigma de programación lógico funcional (Sección 2.4) y su semántica operacional. Para un mayor detalle acerca de los conceptos introducidos en este capítulo, se puede consultar: Baader y Nipkow [BN98], Dershowitz y J.P. Jouannaud [DJ90], Klop [Klo92], Antoy [Ant92], Antoy et al. [AEH00] y Hanus [Han94].

2.1. Signaturas y términos

En este documento consideramos una *signatura* heterogénea Σ, dividida en un conjunto de *constructores* $\mathcal{C}$ y un conjunto de *operaciones* o *funciones* definidas $\mathcal{D}$; algunas veces, utilizamos $\mathcal{F}$ para referirnos al conjunto formado por constructores y funciones: $\mathcal{F} = \mathcal{C} \cup \mathcal{D}$. Escribimos $c/n \in \mathcal{C}$ y $f/n \in \mathcal{D}$ para referirnos a un símbolo constructor o a un símbolo de función, respectivamente, donde n expresa la aridad del símbolo en cuestión. $\mathcal{V}$ es el conjunto de variables (e.g., $x, y, \ldots$) tal que $\mathcal{F} \cap \mathcal{V} = \varnothing$. Asumimos la existencia de, al menos, un tipo primitivo *Bool* que contiene los constructores booleanos constantes (de aridad 0) *true* y *false*.

Para denotar al conjunto de *términos* y *términos constructores* (ambos incluyendo posiblemente variables de $\mathcal{V}$) usamos $\mathcal{T}(\mathcal{F}, \mathcal{V})$ y $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectivamente. Con $\mathcal{V}ar(t)$ denotamos al conjunto de variables que aparecen en un término t. Se denomina variable *fresca* a una variable nueva que no ha sido empleada con anterioridad. Un término

t es *básico* si $\mathcal{V}ar(t) = \varnothing$. Un término t es una variante de t' si ambos son iguales módulo renombramiento de variables. Un término es *lineal* si no contiene ocurrencias repetidas de ninguna variable. Una lista finita de objetos $o_1, \ldots, o_n$ se representa con $\overline{o_n}$.

Un *patrón* es un término que tiene la forma $f(\overline{d_n})$ donde $f/n \in \mathcal{D}$ y $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. $root(t)$ denota el símbolo en la raíz del término t visto como un árbol. Se dice que un término está *encabezado por un símbolo de función* si $root(t) \in \mathcal{D}$.

Usamos la función estándar *depth* para denotar la máxima profundidad de un término.

$$depth(t) = \begin{cases} 1 & \text{si } t \text{ es una constante o una variable} \\ 1 + max(\{\overline{depth(t_n)}\}) & \text{si } t \text{ es de la forma } f(\overline{t_n}), n > 0 \end{cases}$$

Los términos se pueden ver como árboles etiquetados de la forma habitual. Las *posiciones* $(p, q, \ldots)$ de un término t se representan por secuencias (posiblemente vacías) de números naturales que sirven para denotar los subtérminos de t. $\mathcal{P}os(t)$ denota el conjunto de posiciones de un término t, que se define recursivamente como sigue:

$$\mathcal{P}os(t) = \begin{cases} \{\epsilon\} & \text{si } t \in \mathcal{V} \\ \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \ \wedge \ p \in \mathcal{P}os(t_i)\} & \text{si } t = f(t_1, \ldots, t_n) \\ & \text{donde } f \in \mathcal{F} \end{cases}$$

$\mathcal{FP}os(t)$ denota el conjunto de posiciones *no variables* de un término t. Las posiciones están ordenadas por el orden de *prefijo*: $p \leq q$ si existe w tal que $p.w = q$. Denotamos con ϵ la secuencia vacía. Si p y q son posiciones, escribimos $p \leq q$ si p está encima o es un *prefijo* de q, mientras que escribimos $p \perp q$ si p y q son posiciones disjuntas (i.e., no verifican $p \leq q$ ni $q \leq p$). $t|_p$ denota el subtérmino de t en la posición p como sigue:

$$t|_p = \begin{cases} t & \text{si } p = \epsilon \\ t_i|_q & \text{si } p = i.q \text{ y } t = f(t_1, \ldots, t_k), \text{ con } 1 \leq i \leq k \text{ y } f \in \mathcal{F} \end{cases}$$

$t[s]_p$ denota el término t donde el subtérmino en la posición p ha sido reemplazado por el término s.

2.2. Sustituciones

Una *sustitución* σ se denota por $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ donde $\sigma(x_i) = t_i$ para $i = 1, \ldots, n$ (con $x_i \neq x_j$ si $i \neq j$), y $\sigma(x) = x$ para cualquier otra variable x.

El conjunto (finito) $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ recibe el nombre de *dominio* de σ. Denotamos por $\mathcal{R}an(\sigma)$ las variables que ocurren en el rango de σ, i.e., si $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, entonces $\mathcal{R}an(\sigma) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$.

Aunque es común utilizar letras griegas para hacer referencia a sustituciones, al denotar la sustitución vacía o identidad utilizaremos el símbolo *id*. Una sustitución σ es *constructora* (*básica*) si para toda variable $x \in \mathcal{D}om(\sigma)$, $\sigma(x)$ es un término constructor (básico). La *composición* de dos sustituciones σ y τ se define por $(\sigma \circ \tau)(x) = \sigma(\tau(x))$ para toda $x \in \mathcal{V}$. Las sustituciones se extienden a morfismos sobre términos por $\sigma(f(\overline{t_n})) = f(\overline{\sigma(t_n)})$ para cada término $f(\overline{t_n})$.

Consideramos el *preorden* $\leq$ (de *subsumción*) entre dos sustituciones como sigue: $\theta \leq \sigma$ si y sólo si existe una sustitución γ tal que $\sigma = \gamma(\theta)$. De esta manera, decimos que θ es *más general* que σ. El preorden de subsumción induce un preorden parcial sobre términos dado por $t \leq t'$ si y sólo si existe una sustitución γ tal que $t' = \gamma(t)$. En este caso, decimos que t' es una *instancia* de t.

Dada una sustitución θ y un conjunto de variables $V \subseteq \mathcal{V}$, denotamos con $\theta_{\restriction V}$ la sustitución obtenida a partir de θ restringiendo su dominio a V. Escribimos $\theta = \sigma\ [V]$ si $\theta_{\restriction V} = \sigma_{\restriction V}$ y $\theta \leq \sigma\ [V]$ denota la existencia de una sustitución γ tal que $\gamma \circ \theta = \sigma\ [V]$. Se denomina *unificador* de dos términos s y t a una sustitución σ tal que $\sigma(s) = \sigma(t)$. Un unificador σ es el *unificador más general*: mgu (del inglés *most general unifier*), si $\sigma \leq \sigma'\ [\mathcal{V}]$ para cualquier otro unificador σ'. Un mgu es único salvo (composición con sustituciones de) renombramiento.

2.3. Sistemas de reescritura de términos

Un *Sistema de Reescritura de Términos*: SRT (del término original en inglés *Term Rewriting System*) es un conjunto de reglas de reescritura (o ecuaciones orientadas) de la forma $l \to r$ tal que $l \notin \mathcal{V}$ y $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Al término l se le llama la *parte izquierda* de la regla y a r la *parte derecha*. Un SRT $\mathcal{R}$ está *basado en constructores* si para toda regla $l \to r \in \mathcal{R}$, l es un patrón, i.e, tiene la forma $f(s_1, \ldots, s_n)$ donde $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ son términos constructores para todo $i = 1, \ldots, n$. La *definición* de f en $\mathcal{R}$ es el conjunto de reglas en $\mathcal{R}$ cuyo símbolo en cabeza de la parte izquierda es f. Un SRT $\mathcal{R}$ es *lineal por la izquierda* (respectivamente *lineal por la derecha*) si l (respectivamente r) es lineal para todas las reglas $l \to r \in \mathcal{R}$.

Un *paso de reescritura* es la aplicación de una regla de reescritura a un término, i.e., $t \to_{p,R} s$ si existe una posición $p \in \mathcal{P}os(t)$, una regla de reescritura $R = l \to r$ y una sustitución σ tal que $t|_p = \sigma(l)$ y $s = t[\sigma(r)]_p$ (a menudo omitiremos p y R en un paso de computación cuando esté claro por el contexto). A la parte izquierda instanciada $\sigma(l)$ de una regla de reducción se le denomina *redex* (del inglés *red*ucible

*ex*pression, i.e., expresión reducible). Un redex $t|_p$ de un término t es un *redex más externo* si no existe otro redex $t|_q$ de t con $q < p$. El paso de reescritura aplicado sobre un redex más externo se denomina *paso de reescritura más externo*. La reescritura más externa es la base operacional de los lenguajes funcionales perezosos. A la acción de emparejar un subtérmino $t|_p$ para que ajuste con la parte izquierda instanciada $\sigma(l)$ de alguna regla tal que $t|_p = \sigma(l)$ se le llama *emparejamiento de patrones* (del inglés *pattern matching*).

Un término t es *irreducible* o está en *forma normal* si no existe ningún término s tal que $t \to s$. Denotamos con $t\!\downarrow$ la forma normal de t. Un término t está en *forma normal de cabeza* si no se puede reducir a un redex.

A partir de la relación binaria $\to$, $\to^+$ denota el cierre transitivo de $\to$ y $\to^*$ denota el cierre reflexivo y transitivo de la relación indicada. Un SRT $\mathcal{R}$ se dice *noetheriano* (o terminante) si no hay una secuencia infinita de la forma $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} \ldots$; $\mathcal{R}$ es *confluente* si para todo término t tal que $t \to^*_{\mathcal{R}} s_1$ y $t \to^*_{\mathcal{R}} s_2$, entonces existe un término s tal que $s_1 \to^*_{\mathcal{R}} s$ y $s_2 \to^*_{\mathcal{R}} s$ (en símbolos $s_1 \downarrow s_2$). Un SRT que es noetheriano y confluente se llama *canónico* o *completo*.

Decimos que dos reglas $l \to r$ y $l' \to r'$ se *solapan* si existe una posición no variable $p \in \mathcal{FP}os(l)$ y un unificador más general σ tal que $\sigma(l|_p) = \sigma(l')$. Al par formado por $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ se le llama *par crítico* (si $p = \epsilon$ se le denomina *overlay*). El par crítico $\langle t, s \rangle$ es trivial si $t = s$ (módulo renombramiento de variables). Un SRT que es lineal por la izquierda sin pares críticos se denomina *ortogonal*. Si todos los pares críticos de un SRT son *overlays* triviales, éste se denomina *cuasi ortogonal*, y, si sólo tiene pares críticos triviales se le llama *débilmente ortogonal*.

2.4. Programación lógico funcional

El gran interés generado en la década de los 90 en la integración de los paradigmas declarativos más populares, i.e., la programación funcional y la programación lógica, dio como fruto varias propuestas de lenguajes *lógico funcionales* integrados que combinaron las ventajas de ambos paradigmas (una revisión general aparece en [Han94]). Los lenguajes lógico funcionales extienden ambos campos: los lenguajes funcionales son extendidos con características tales como inversión de funciones, estructuras de datos parciales y variables lógicas [Red85], mientras que los lenguajes lógicos se enriquecen con la posibilidad de anidar funciones y un principio operacional más eficiente [Han90] (por lo que dependen en menor medida de características impuras como el "corte" de Prolog).

Narrowing es un modelo computacional que tiene particular importancia en el contexto de los lenguajes lógico funcionales, lo que se debe a que la mayoría de propuestas

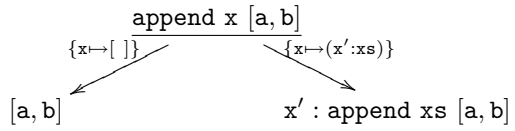


Figura 2.1: Árbol de ejecución simbólica para `append x [a,b]`.

para la integración de la programación funcional y lógica están basadas precisamente en *narrowing* (ver e.g., [BE86, Fri85, GM86, Han90, MR92, Red85]).

Antes de continuar es preciso hacer una aclaración en cuanto a la notación de los ejemplos. Aún cuando consideramos un lenguaje de primer orden (propiamente SRTs), en los ejemplos utilizamos notación currificada, como en los lenguajes funcionales. Además, utilizamos mayúsculas para las iniciales de los constructores, las variables son minúsculas al igual que las iniciales de los nombres de las funciones definidas (inspirados por la sintaxis de Curry). De igual modo, para escribir reglas usamos $\mathtt{l} = \mathtt{r}$ en vez de $l \to r$.

Ejemplo 2.1 *Consideremos la siguiente definición de la función* `append` *para concatenar dos listas*

$$\begin{array}{rcl} \mathtt{append}\ [\,]\ \mathtt{y} & = & \mathtt{y} \\ \mathtt{append}\ (\mathtt{x} : \mathtt{xs})\ \mathtt{y} & = & \mathtt{x}\ :\ \mathtt{append}\ \mathtt{xs}\ \mathtt{y} \end{array}$$

donde : *es el constructor de listas y* [ ] *denota la lista vacía:*

La función `append` introducida en el Ejemplo 2.1 nos permite concatenar dos listas, e.g., `append` [a] [b, c, d] (donde $\mathtt{a}, \ldots, \mathtt{d}$ representan datos) produce [a, b, c, d]. En un lenguaje funcional puro, todos los parámetros de `append` deben ser básicos. Por otro lado, *narrowing* puede llevar a cabo computaciones aún cuando todos o parte de los argumentos son desconocidos. Por ejemplo, en la Figura 2.1, al computar `append x` [a, b] se obtiene:

- el término [a, b] con una sustitución asociada $\{\mathtt{x} \mapsto [\,]\}$
- y el término, $\mathtt{x}'$: `append xs` [a, b] con una sustitución asociada $\{\mathtt{x} \mapsto (\mathtt{x}' : \mathtt{xs})\}$

En la Sección 2.4.1 introducimos formalmente la definición de *narrowing* así como la relación que guarda el término computado y la sustitución asociada.

A partir de este punto usamos el término *programa lógico funcional* para referirnos a un SRT finito basado en constructores lineal por la izquierda. Una subclase de los SRTs basados en constructores son los sistemas de reescritura *inductiva-*

mente secuenciales. Esta subclase se ha definido, estudiado y empleado en la implementación de lenguajes de programación tanto funcionales como lógico funcionales [Ant92, AEH00, Han97, HLM98, LLR93]. La razón de su uso se debe a que aporta optimalidad a las computaciones en ambos paradigmas. Los sistemas de reescritura inductivamente secuenciales son programas funcionales típicos, que pueden verse como SRTs basados en constructores donde la parte izquierda de las reglas presenta patrones discriminantes, i.e., patrones que permiten diferenciar cada regla del resto de ellas.

2.4.1. *Narrowing* y estrategia de *narrowing*

Narrowing, introducido originalmente en el campo de la demostración automática de teoremas [Sla74] es una relación sobre términos inducida por un sistema de reescritura de términos. El prodecimiento de *narrowing* puede verse como una extensión de la reescritura de términos y se puede implementar eficientemente sustituyendo el emparejamiento de patrones por unificación en el procedimiento de reducción. Aunque un programa lógico funcional es propiamente un SRT, los términos para evaluarse (con respecto al SRT) pueden presentar variables libres. Por ello, la semántica operacional de los lenguajes lógico funcionales está basada en *narrowing*.

A grandes rasgos, para evaluar un término que contiene variables *narrowing* instancia de manera indeterminista las variables del término de forma que sea posible realizar un paso de reescritura. La técnica computa el unificador más general entre un subtérmino y la parte izquierda de alguna regla y reemplaza éste por la parte derecha instanciada de la regla [Han94]. A menudo se impone que el unificador computado sea el más general, aunque dicha condición puede relajarse en ocasiones exigiendo que sea únicamente un unificador arbitrario. De hecho [Pad88] distingue entre *narrowing* y *narrowing* más general en función de que sea un unificador arbitrario o el unificador más general respectivamente. En la estrategia de evaluación denominada *narrowing necesario* (Sección 2.4.2) con el fin de llevar a cabo sólo los pasos de reducción "necesarios" no se exige computar el unificador más general [AEH00].

Formalmente, $t \leadsto_{p,R,\sigma} t'$ es un *paso de narrowing* si p es una posición no variable de t y $\sigma(t) \rightarrow_{p,R} t'$, donde σ es (usualmente) el mgu de $t|_p$ y la parte izquierda l de la regla R. La sustitución σ restringe su dominio a $\mathcal{V}ar(t)$, lo cual es correcto ya que la aplicación de la sustitución $\sigma(t)$ hace posible el paso de reescritura implícito en un paso de *narrowing*.

Una secuencia de pasos de *narrowing* $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ se denota por medio de $t_0 \leadsto^*_\sigma t_n$ con $\sigma = \sigma_n \circ \ldots \circ \sigma_1$ ($\sigma = id$ si $n = 0$). En la programación lógico funcional estamos interesados en computar *valores* (términos constructores) así como *respuestas* (sustituciones); por ello, decimos que una *derivación* de *narrowing* $t \leadsto^*_\sigma c$ *computa*

el resultado c con respuesta σ si c es un término constructor[1].

Las derivaciones de *narrowing* se pueden representar, siguiendo el estilo de [LS91], mediante un árbol (posiblemente infinito) con un número finito de ramas.

Formalmente, dado un SRT $\mathcal{R}$ y un término t con símbolo de función en cabeza, un *árbol de narrowing* para t en $\mathcal{R}$ es un árbol que satisface las condiciones siguientes: (a) cada nodo del árbol es un término, (b) el nodo raíz es t, y (c) si s es un nodo del árbol entonces, para cada paso de *narrowing* $s \leadsto_{p,R,\sigma} s'$, el nodo tiene un hijo s' y el arco correspondiente en el árbol se etiqueta con (p, R, σ).

Adoptamos la convención de que cualquier derivación es potencialmente incompleta (de esa manera una rama puede ser de fallo, incompleta, de éxito o infinita). Una *hoja de fallo* contiene una expresión que no es un término constructor y que no es susceptible de reducción por *narrowing*.

Una *ecuación* es un par $t \approx t'$ de términos del mismo tipo. *Narrowing* puede computar valores para las variables contenidas en una ecuación y de esa manera hacer que sea verdadera, i.e., es capaz de resolver ecuaciones. En muchos lenguajes funcionales y lógico funcionales el objetivo de sus semánticas es la reducción de los términos a términos constructores básicos (en lugar de expresiones arbitrarias). Así, es normal considerar que los valores a los cuales se reducen los términos es a términos constructores básicos y que una ecuación se cumple si ambos lados lados de la ecuación tienen el mismo valor. [AEH00] define la validez de una ecuación como la *igualdad estricta* sobre términos en el mismo sentido de los lenguajes lógico funcionales K-LEAF [GLMP91] y BABEL [MR92] donde una ecuación se satisface, propiamente, si ambos lados son equivalentes a un mismo término constructor básico. Formalmente:

Definición 1 (ecuación, solución [AEH00]**)** *Una ecuación es un par de términos $t \approx t'$ del mismo tipo. Una sustitución σ es una* solución *para una ecuación $t \approx t'$ sii $\sigma(t)$ y $\sigma(t')$ son reducibles al mismo término constructor básico.*

Las ecuaciones se pueden interpretar como términos si se define el símbolo $\approx$ como un símbolo de operación binario. Por lo tanto, nociones para términos tales como sustituciones, reescritura, *narrowing*, etc., se podrán usar también para ecuaciones. La semántica de la función $\approx$ se define mediante las siguientes reglas:

$$\begin{aligned} c \approx c &\rightarrow true \\ c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) &\rightarrow (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) \\ true \wedge x &\rightarrow x \end{aligned}$$

donde $\wedge$ denota un símbolo infijo asociativo por la derecha y c un constructor de aridad 0 en la primer regla y de aridad $n > 0$ en la segunda. Las reglas no cam-

[1] A veces se relaja a forma normal de cabeza, i.e., una variable o un término encabezado por un símbolo constructor.

bian la ortogonalidad de un sistema de reescritura (lo mismo ocurre con los sistemas inductivamente secuenciales). A partir de las reglas recién definidas, la solución de una ecuación se computa mediante su reducción a *true* (utilizando *narrowing*). En *K-LEAF* [GLMP91] y BABEL [MR92] se sigue esta misma aproximación.

La definición de una *estrategia* adecuada para la reducción de los términos es uno de los aspectos más importantes en el diseño de lenguajes lógico funcionales. A continuación definimos formalmente el concepto de estrategia de *narrowing*.

Definición 2 (estrategia de *narrowing* [AEH00]) *Una estrategia de narrowing es una función que toma un término y computa un conjunto de triples. Si $\mathcal{S}$ es una estrategia de narrowing, t es un término y $(p, l \to r, \sigma) \in \mathcal{S}(t)$, entonces p es una posición de t, $l \to r$ es una regla de reescritura y σ es una sustitución tal que:*

$$t \leadsto_{p,l\to r,\sigma} \sigma(t[r]_p)$$

es un paso de narrowing.

Una estrategia de *narrowing* es *correcta* si cada sustitución computada mediante *narrowing* es un unificador y *completa* si todas las soluciones o las más representativas del conjunto de todas las posibles soluciones son computadas, i.e., para cada unificador del programa hay una sustitución computada que es cuando menos tan general como el unificador. En general, el procedimiento de *narrowing* es indeterminista debido a la existencia de dos grados de libertad: la elección del subtérmino a reducir y la elección de la regla. Esto conduce a un espacio de búsqueda muy amplio. Por ello, se han llevado a cabo notables esfuerzos con el interés de conseguir una estrategia de *narrowing* que reduzca el espacio de búsqueda de las computaciones, e.g., *narrowing* básico: Hullot [Hul80]; *innermost*: Fribourg [Fri85]; *outermost*: Echahed [Ech90]; *estándar*: Darlington y Guo [DG89], Ida y Nakahara [IN97], Middeldorp y Okui [MO98] , You [You91]; *perezoso*: Giovannetti et al. [GLMP91], Hans et al. [HLW92], Moreno-Navarro y Rodríguez-Artalejo [MR92], Reddy [Red85]; LSE *Narrowing* que aplica tests de redundancia para evitar soluciones repetidas: Bockmayr et al. [BKW95]. Cada estrategia impone ciertas condiciones sobre los sistemas de reescritura con la finalidad de asegurar la completitud en el cómputo de las soluciones.

2.4.2. *Narrowing* necesario

Narrowing necesario es una estrategia para *sistemas inductivamente secuenciales* [Ant92] que preserva la completitud del *narrowing* y sólo ejecuta pasos que son "inevitables" o necesarios durante la computación.

El cómputo de pasos inevitables en dicha estrategia conduce a la optimalidad de la misma. Así pues, la estrategia de *narrowing* necesario posee propiedades de optimalidad con respecto a la longitud de las derivaciones[2] y al número de soluciones computadas para la caracterización de programas indicada. La noción de paso inevitable es bien conocida en la reescritura. Los sistemas *ortogonales* tienen la propiedad de que, para todo término t que no esté en forma normal, existe un redex llamado *necesario*, que eventualmente se debe reducir para calcular la forma normal de t [HL92, KM91, O'D77]. Así, la reescritura reiterada de redexes necesarios de un término es suficiente para computar su forma normal (si existe). En otras palabras, en los sistemas ortogonales basta con considerar redexes necesarios. Este hecho ha sido extendido a *narrowing* y, en concreto, a los sistemas inductivamente secuenciales—una subclase de los sistemas ortogonales—en [AEH00].

Un segundo resultado de optimalidad tiene que ver con las sustituciones computadas por una derivación de *narrowing*: derivaciones distintas de *narrowing* necesario computan conjuntos disjuntos de sustituciones. Esta propiedad complementa el carácter necesario de un paso en el sentido de que las derivaciones computadas por la estrategia son necesarias también. A grandes rasgos, cualquier solución que se computa con una derivación no se computa con ninguna otra; así, cada derivación que conduce a una solución es necesaria también, además de cualquier paso de la derivación.

Definición 3 (*narrowing* necesario [AEH00]**)** *Un paso de narrowing* $t \leadsto_{p,R,\sigma} t'$ *es necesario sii, para cada* $\eta \geq \sigma$, p *es la posición de un redex necesario de* $\eta(t)$. *Una derivación de narrowing es necesaria sii cada paso de la derivación es necesario.*

La definición anterior agrega un nuevo nivel de dificultad al cómputo de pasos de *narrowing* necesarios en comparación con la reescritura necesaria, i.e., para computar formas normales se tiene que tomar en cuenta cualquier instanciación del término en la derivación. No obstante, el problema tiene una solución eficiente en los sistemas inductivamente secuenciales, gracias a que se elimina el requisito de que el unificador de un paso de *narrowing* deba ser el más general. Así pues, en *narrowing* necesario se computan unificadores arbitrarios, los cuales son esenciales para capturar el carácter *necesario* de un paso de *narrowing*. Los unificadores calculados, de alguna forma "anticipan" ciertos enlaces para las variables que, de todos modos, se habrían computado más adelante en la derivación. Esto complica de alguna manera la definición de la estrategia de *narrowing* pero, como contrapartida, permite obtener el refinamiento de *narrowing* más eficiente que se conoce.

Nos centraremos ahora en una clase de sistemas de reescritura para los que existe una estrategia de *narrowing* necesario computable y eficiente, i.e., los *sistemas inductivamente secuenciales*. Los programas de esta clase comparten la propiedad de

[2] Sólo si se considera una semántica basada en la compartición de variables (*sharing*).

que las reglas que definen cualquier operación o función se pueden organizar en una estructura jerárquica llamada *árbol definicional* [Ant92]. Esta estructura se utiliza para implementar la reescritura necesaria; a partir de aquí este hecho se generaliza a *narrowing*.

Árboles definicionales

Un árbol definicional se define como un conjunto parcialmente ordenado de patrones con algunas restricciones adicionales. Para clasificar los nodos del árbol se utilizan las funciones (sin interpretación) *branch* y *leaf*.

Definición 4 (árbol definicional [AEH00]**)** *$\mathcal{T}$ es un árbol definicional parcial o pdt (del inglés "partial definitional tree") con patrón π sii se cumple uno de los siguientes casos:*

$\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$ donde π es un patrón y o es la posición de una variable (la variable inductiva) en el patrón π. El tipo de $\pi|_o$ tiene k constructores $c_1, \ldots, c_k$, $k > 0$, y para todo $i \in \{1, \ldots, k\}$, $\mathcal{T}_i$ es un pdt con patrón $\pi[c_i(x_1, \ldots, x_n)]_o$, siendo n la aridad de c_i y $x_1, \ldots, x_n$ variables frescas diferentes.

$\mathcal{T} = leaf(\pi)$ donde π es un patrón

El conjunto de *pdt's* sobre la signatura Σ se simboliza con $\mathcal{P}(\Sigma)$. Sea $\mathcal{R}$ un sistema de reescritura, $\mathcal{T}$ es un *árbol definicional* de una operación (función) f si y sólo si $\mathcal{T}$ es un *pdt* cuyo patrón tiene la forma $f(x_1, \ldots, x_n)$, donde n es la aridad de f y $x_1, \ldots, x_n$ son variables frescas distintas, y por cada regla $l \to r$ de $\mathcal{R}$ con $l = f(x_1, \ldots, x_n)$ existe una hoja $leaf(\pi)$ de $\mathcal{T}$ tal que l es una variante de π (a menudo diremos que el nodo $leaf(\pi)$ representa la regla $l \to r$). Un árbol definicional $\mathcal{T}$ de una operación f se dice *minimal* si y sólo si bajo cada nodo de tipo *branch* de $\mathcal{T}$ hay una hoja que representa a una regla de f.

A una función f de un sistema de reescritura $\mathcal{R}$ la llamamos *inductivamente secuencial* [AEH00] sii existe un árbol definicional $\mathcal{T}$ para f tal que cada nodo *leaf* de $\mathcal{T}$ representa como mucho una regla de $\mathcal{R}$ y todas las reglas tienen representación. A un sistema de reescritura $\mathcal{R}$ lo llamamos *inductivamente secuencial* si todas sus funciones definidas son inductivamente secuenciales.

Para facilitar la comprensión del concepto de árbol definicional, a menudo es conveniente dar una representación gráfica en la que cada nodo se etiqueta con un patrón, la posición inductiva en las ramas se enmarca dentro de una caja y las hojas contienen las reglas correspondientes.

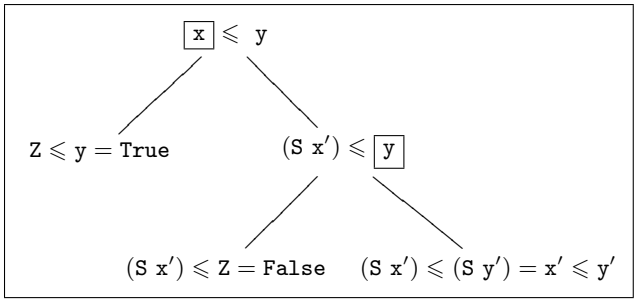


Figura 2.2: Árbol definicional para la función "⩽".

Ejemplo 2.2 *Consideremos la siguiente función "menor o igual" ⩽ sobre números naturales*

$$\begin{array}{rcl} \texttt{Z} \leqslant \texttt{y} & = & \texttt{True} \\ (\texttt{S x}) \leqslant \texttt{Z} & = & \texttt{False} \\ (\texttt{S x}) \leqslant (\texttt{S y}) & = & \texttt{x} \leqslant \texttt{y} \end{array}$$

La Figura 2.2 muestra el árbol definicional para la función ⩽. Cada nodo que representa una rama contiene el patrón del nodo correspondiente del árbol definicional. Cada nodo de tipo hoja representa una regla. Los argumentos inductivos de cada rama se enmarcan con un recuadro.

Para la exposición de la estrategia de *narrowing* necesario es importante distinguir si un nodo tipo hoja de un árbol definicional de una función f representa o no una regla de la definición de f. Para señalar que $leaf(\pi)$ es un *pdt* que no representa a ninguna regla se va a usar $exempt(\pi)$ en lugar de $leaf(\pi)$. De la misma manera, se abrevia con $rule(\pi, \sigma(l) \rightarrow \sigma(r))$ el hecho de que $leaf(\pi)$ es un *pdt* que representa alguna regla $l \rightarrow r$ del sistema de reescritura considerado, donde σ es una sustitución de renombramiento tal que $\sigma(l) = \pi$. Los patrones de un árbol definicional son un conjunto finito parcialmente ordenado por el preorden de subsumción.

Estrategia de *narrowing* necesario

A continuación presentamos una descripción informal de la estrategia de *narrowing* necesario. Por *narrowing* necesario hacemos referencia a la estrategia de *narrowing* necesario más externo (*outermost*) introducida en [AEH00]. Sea $t = f(t_1, \ldots, t_n)$ un término. Primero, se unifica t con algún elemento maximal del conjunto de patrones del árbol definicional de f. Tal patrón se denota con π, al unificador más general de t y π con τ y, finalmente, con $\mathcal{T}$ el *pdt* en el que se encuentra π.

- Si $\mathcal{T}$ es un *pdt* que representa una regla, entonces se aplica *narrowing* a $\tau(t)$ en la posición ϵ con la regla representada por $\mathcal{T}$.

- Si $\mathcal{T}$ es un *pdt* cuya raíz es un *exempt*, entonces $\tau(t)$ no se puede reducir por *narrowing* a un termino encabezado por constructor.

- Si $\mathcal{T}$ es un *pdt* cuya raíz es un *branch*, entonces hay que volver a aplicar la estrategia pero ahora con $\tau(t|_o)$, donde o es la siguiente posición inductiva indicada en $\mathcal{T}$ y τ es la sustitución que asegura que el paso será necesario (i.e., la sustitución anticipada). El resultado de la invocación recursiva es compuesto con τ y o. Los detalles de la composición se muestran en la definición formal introducida más adelante.

Estas computaciones se implementan en la función λ. Dicha función toma un término encabezado por un símbolo de función t y el árbol definicional $\mathcal{T}$ del símbolo en cabeza de t y, de manera indeterminista, devuelve una terna (p, R, σ), donde p es una posición de t, R es o bien una regla $l \to r$ de $\mathcal{R}$ o el símbolo especial "?" y σ es una sustitución.

- Si $R = l \to r$ entonces la estrategia lleva a cabo el paso de *narrowing* $t \leadsto_{p,l\to r,\sigma} \sigma(t[r]_p)$

- Si $R = ?$ entonces la estrategia se rinde dado que no es posible derivar un término encabezado por un constructor mediante *narrowing* a partir de t.

En lo que sigue $pattern(\mathcal{T})$ denota el patrón de $\mathcal{T}$ y $\prec$ denota el orden noetheriano sobre $\mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{P}(\Sigma)$ que se define por: $(t_1, \mathcal{T}_1) \prec (t_2, \mathcal{T}_2)$ si y sólo si:

1. t_1 tiene menos ocurrencias de símbolos de función que t_2

2. o bien, $t_1 = t_2$ y $\mathcal{T}_1$ es un subárbol propio de $\mathcal{T}_2$.

Definición 5 [AEH00] *La función λ toma dos argumentos: un término t encabezado por un símbolo de función y un pdt $\mathcal{T}$ tal que pattern($\mathcal{T}$) y t unifican. La función λ produce un conjunto de triples de la forma (p, R, σ) donde p es una posición de t, R es una regla $l \to r$ de $\mathcal{R}$ o bien el símbolo especial "?" y σ es un unificador de*

pattern$(\mathcal{T})$ *y* t. *La función* λ *se define por inducción sobre* $\prec$ *como sigue:*

$$\lambda(t,\mathcal{T}) \ni \begin{cases} (\epsilon, R, mgu(t,\pi)) & \text{si } \mathcal{T} = rule(\pi, R); \\ (\epsilon, ?, mgu(t,\pi)) & \text{si } \mathcal{T} = exempt(\pi); \\ (p, R, \sigma) & \text{si } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t \text{ y } pattern(\mathcal{T}_i) \text{ unifican para algun } i \\ & \quad \text{y } (p, R, \sigma) \in \lambda(t, \mathcal{T}_i); \\ (o.p, R, \sigma \circ \tau) & \text{si } \mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), \\ & \quad t \text{ y } pattern(\mathcal{T}_i) \text{ no unifican para ningún } i, \\ & \quad \tau = mgu(t, \pi), \\ & \quad \mathcal{T}' \text{es un árbol definicional del símbolo raíz} \\ & \quad \text{de } \tau(t|_o) \text{ y } (p, R, \sigma) \in \lambda(\tau(t|_o), \mathcal{T}'); \end{cases}$$

En el tercer caso, la función λ está bien definida por los motivos que se exponen a continuación. Por la definición de *pdt*, existe un sub*pdt* $\mathcal{T}_i$ de $\mathcal{T}$ tal que *pattern*$(\mathcal{T}_i)$ y t unifican si $t|_o$ está encabezado por un símbolo constructor o es una variable.

λ está bien definida en el cuarto caso, toda vez que puede ocurrir sólo si $t|_o$ está encabezado por un símbolo de función. En este caso $\tau|_{\mathcal{V}ar(t)}$ es una sustitución constructora ya que π es un patrón lineal. Puesto que t es un término encabezado por un símbolo de función y $o \neq \epsilon$, $\tau(t|_o)$ tiene menos ocurrencias de símbolos de función definidos que t. Así mismo, como $t|_o$ es un término encabezado por un símbolo de función, entonces debe aplicarse $\tau(t|_o)$ antes de la llamada recursiva de λ. Por la definición de *pdt*, *pattern*$(\mathcal{T}') \leq \tau(t|_o)$. Y, por tanto, *pattern*$(\mathcal{T}')$ y $\tau(t|_o)$ unifican, lo que implica que λ está bien definida en este caso también.

Como en los procedimientos de prueba para programación lógica, asumimos que los árboles definicionales que se usan en un paso de *narrowing* contienen siempre variables frescas.

La computación de $\lambda(t, \mathcal{T})$ implica una selección no determinista cuando $\mathcal{T}$ es un *pdt* encabezado por un *branch* (el entero i expresado en la función λ cuando $\tau(t|_o)$ es una variable). Cuando $t|_o$ es un término encabezado por un símbolo de función, la sustitución τ es la sustitución *anticipada* que garantiza que la posición computada es necesaria. La sustitución se pasa en la llamada recursiva a λ para asegurar la consistencia de la computación cuando t no es lineal. La sustitución anticipada no existe cuando $t|_o$ no es un término encabezado por símbolo de función ya que el patrón en $\mathcal{T}_i$ es una instancia de π. Por lo tanto, con σ se amplía la sustitución anticipada.

El *narrowing* necesario es correcto y completo

El *narrowing* necesario es un procedimiento *correcto* y *completo* para resolver ecuaciones siempre que incorporemos las reglas de igualdad estricta. En la siguiente proposición se muestra la equivalencia entre la reducibilidad al mismo término constructor básico y la reducibilidad a *true* usando las reglas de la igualdad.

Proposición 6 [AEH00] *Sea $\mathcal{R}$ un sistema de reescritura de términos sin reglas definidas para "$\approx$" y "$\wedge$". Sea $\mathcal{R}'$ un sistema obtenido agregando las reglas de igualdad para $\mathcal{R}$. Las proposiciones siguientes son equivalentes para todos los términos t y t':*

1. *t y t' son reducibles en $\mathcal{R}$ al mismo término constructor básico*
2. *$t \approx t'$ es reducible en $\mathcal{R}'$ a true.*

La corrección del *narrowing* necesario es un caso especial del *narrowing* general [AEH00]. A continuación, el Teorema 7 establece la *corrección* del *narrowing* necesario para sistemas de reescritura inductivamente secuenciales:

Teorema 7 (corrección del *narrowing* necesario [AEH00]**)** *Sea $\mathcal{R}$ un sistema de reescritura de términos inductivamente secuencial extendido con las reglas de igualdad. Si $t \approx t' \leadsto^*_\sigma$ true es una derivación de narrowing necesario, entonces σ es una solución para $t \approx t'$.*

El *narrowing* necesario sólo instancia variables a términos constructores. Por ello, en [AEH00] sólo se muestra que el *narrowing* necesario es completo para soluciones constructoras. Esto no debe entenderse como una limitación práctica porque las soluciones más generales contendrían expresiones no evaluadas o indefinidas. Tampoco es una limitación con respecto a otros trabajos, ya que otras estrategias de *narrowing* más generales son completas sólo para soluciones normalizables [MH94] mientras que el *narrowing* perezoso es completo para soluciones constructoras [GLMP91, MR92]. Además, *narrowing* necesario es completo para la clase de los sistemas de reescritura ortogonales con respecto a las sustituciones irreducibles, aunque tanto los pasos de reescritura necesaria como los pasos de *narrowing* necesario no son computables para esta clase de sistemas de reescritura. El Teorema 8 muestra la *completitud* de la estrategia λ y, consecuentemente, del *narrowing* necesario *outermost* para sistemas de reescritura inductivamente secuenciales.

Teorema 8 (completitud del *narrowing* necesario [AEH00]**)** *Sea $\mathcal{R}$ un sistema de reescritura de términos inductivamente secuencial extendido por las reglas de igualdad. Sea la sustitución constructora σ una solución de la ecuación $t \approx t'$ y V un conjunto finito de variables formado por $\mathcal{V}ar(t) \cup \mathcal{V}ar(t')$. Entonces, existe una derivación $t \approx t' \overset{*}{\leadsto}_{\sigma'}$ true computada por λ tal que $\sigma' \leq \sigma[V]$.*

El Teorema 9 completa la caracterización de la estrategia de *narrowing* necesario mostrando que la función λ no computa soluciones redundantes. Para identificar tales soluciones, se utiliza la noción de *independencia*. Dos sustituciones son independientes cuando no "unifican", i.e., dadas dos sustituciones σ y σ', decimos que son independientes sobre el conjunto de variables V sii existe algún $x \in V$ tal que $\sigma(x)$ y $\sigma'(x)$ no son unificables.

Teorema 9 (independencia del *narrowing* necesario [AEH00]) *Sea $\mathcal{R}$ un sistema de reescritura de términos inductivamente secuencial extendido con las reglas de igualdad, e una ecuación y $V = Var(e)$. Sean $e \leadsto^+_\sigma true$ y $e \leadsto^+_{\sigma'} true$ dos derivaciones distintas computadas por λ. Entonces, σ y σ' son independientes sobre V.*

Así, la estrategia de *narrowing* necesario es completa, correcta, produce soluciones independientes y computa sólo las derivaciones que son necesarias.

Capítulo 3

Evaluación parcial dirigida por *narrowing*

En este Capítulo se presentan los conceptos de la evaluación parcial de programas lógico funcionales dirigida por *narrowing* necesario. Las definiciones de resultante, pre–evaluación parcial, cierre de los programas, renombramiento, evaluación parcial, corrección y completitud aparecen en [AHLV99, AHLV05]. Finalmente, mostraremos el algoritmo de evaluación parcial y los aspectos relacionados de control local y de control global con base en [AFV98, AV02].

3.1. Antecedentes

En [AFV96, Vid96, AFV98] se define el primer marco de evaluación parcial para programas lógico funcionales. En ese marco, se utiliza *narrowing* (la semántica operacional estándar de los lenguajes integrados) para dirigir el proceso de evaluación parcial. La metodología, es paramétrica con respecto a la estrategia de *narrowing* empleada. Posteriormente, se mejoraron las técnicas de control y de renombramiento empleadas en el método y se extendieron para una estrategia de *narrowing* perezoso [AFJV97, AAF+98a]. Las aproximaciones para la evaluación parcial de los lenguajes lógico funcionales que se han desarrollado más recientemente [AHV02, AHV03] usan una forma de *narrowing* necesario [AEH00] para llevar a cabo las computaciones en tiempo de evaluación parcial. En paralelo, se postularon los fundamentos del esquema para la evaluación parcial dirigida por *narrowing* necesario, los cuales fueron introducidos en [AHLV99, AHLV05]. Los conceptos que se presentan a continuación corresponden a este último contexto.

3.2. Resultantes y pre–evaluación parcial

Para generar el programa especializado se hace uso de la noción de resultante. En lo que sigue, se usará $s \leadsto_\sigma^+ t$ para denotar una derivación de *narrowing* compuesta de al menos un paso de *narrowing*.

Definición 10 (resultante [AHLV05]**)** *Sea $\mathcal{R}$ un SRT y s un término. Dada una derivación de narrowing $s \leadsto_\sigma^+ t$, su resultante asociado es la regla de reescritura $\sigma(s) \to t$.*

Hay que hacer notar que, si la llamada s que se ha especializado no es un patrón lineal, entonces las partes izquierdas de los resultantes tampoco serán patrones lineales. Por lo tanto, las reglas resultantes no serían reglas de programa válidas.

Ejemplo 3.1 *Consideremos el siguiente SRT inductivamente secuencial:*

$$\begin{array}{rcl} \texttt{double x} & = & \texttt{add x x} \\ \texttt{add Z n} & = & \texttt{n} \\ \texttt{add (S m) n} & = & \texttt{S (add m n)} \end{array}$$

donde `Z` *y* `S` *se utilizan para denotar los constructores de números enteros. Dado el término* `add (double w) w` *y la siguiente derivación de narrowing necesario (se subraya el redex seleccionado en cada paso de narrowing):*

$$\begin{array}{rl} \texttt{add (}\underline{\texttt{double w}}\texttt{) w} & \leadsto_{\texttt{id}} \texttt{ add (}\underline{\texttt{add w w}}\texttt{) w} \\ & \leadsto_{\{\texttt{w} \mapsto \texttt{(S m)}\}} \texttt{ add (S (add m (S m))) (S m)} \end{array}$$

se computa el resultante asociado a la derivación:

$$\texttt{add (double (S m)) (S m)} \;=\; \texttt{add (S (add m (S m))) (S m)}$$

Este resultante no permite formar una regla de programa válida debido, por un lado, a que la parte izquierda contiene símbolos de función definida anidados: ("`add`*" y "*`double`*") y, segundo, contiene ocurrencias múltiples de la misma variable.*

Más adelante se introduce un post-proceso de renombramiento con la finalidad de producir reglas de programa legales. El post-proceso de renombramiento no sólo elimina estructuras redundantes, además consigue especializaciones *independientes* en el sentido de [LS91]. Esto es necesario para la corrección de la transformación de la evaluación parcial. A grandes rasgos, la independencia asegura que se distingan correctamente las diferentes especializaciones para una misma función, lo cual es crucial para lograr la llamada polivarianza[1] de la especialización.

[1] A partir de una misma función se puede obtener más de una versión especializada.

La *(pre–)evaluación parcial* de un término s se obtiene por la construcción de un árbol de *narrowing* para s (posiblemente incompleto) y luego extrayendo las definiciones especializadas, i.e., los resultantes, formados a partir de las derivaciones (que no son de fallo[2]) del árbol.

Definición 11 (pre–evaluación parcial [AHLV05]**)** *Sea $\mathcal{R}$ un SRT y s un término. Sea $\mathcal{T}$ un árbol de narrowing finito (posiblemente incompleto) para s en $\mathcal{R}$ tal que no se haya reducido ningún término encabezado por un símbolo constructor. Sean $\overline{t_n}$ los términos en las hojas que no son de fallo de $\mathcal{T}$. Entonces, al conjunto de resultantes $\{\sigma_i(s) \to t_i \mid i = 1, \ldots, n\}$ para las secuencias de narrowing $\{s \leadsto^+_{\sigma_i} t_i \mid i = 1, \ldots, n\}$ se le denomina pre–evaluación parcial de s en $\mathcal{R}$.*

La pre–evaluación parcial de un conjunto de términos S en $\mathcal{R}$ se define como la unión de las pre–evaluaciones parciales para los términos de S en $\mathcal{R}$.

Ejemplo 3.2 *Consideremos la función* `append` *para concatenar dos listas (usamos "`[]`" y ":" como constructores de listas):*

$$\begin{array}{rcl} \texttt{append [] ys} & = & \texttt{ys} \\ \texttt{append (x : xs) ys} & = & \texttt{x : append xs ys} \end{array}$$

junto con el conjunto de términos a especializar

$$S = \{\texttt{append (append xs ys) zs},\ \texttt{append xs ys}\ \}$$

Dados los árboles de narrowing necesario de la Figura 3.1, la pre–evaluación parcial asociada de S en $\mathcal{R}$ es como sigue:

$$\begin{array}{l} \texttt{append (append [] ys) zs} = \texttt{append ys zs} \\ \texttt{append (append (x' : xs') ys) zs} = \texttt{x' : append (append xs' ys) zs} \\ \texttt{append [] zs} = \texttt{zs} \\ \texttt{append (y : ys) zs} = \texttt{y : append ys zs} \end{array}$$

La restricción que impide evaluar términos encabezados por un símbolo constructor se debe respetar, el ejemplo que sigue ilustra la consecuencia de no hacerlo.

Ejemplo 3.3 *Consideremos el siguiente programa $\mathcal{R}$:*

$$\begin{array}{rcl} \texttt{f Z} & = & \texttt{Z} \\ \texttt{g x} & = & \texttt{S (f x)} \\ \texttt{h (S x)} & = & \texttt{S Z} \end{array}$$

[2] Una derivación de fallo llega a una expresión que no es un término constructor y que no se puede reducir por *narrowing*.

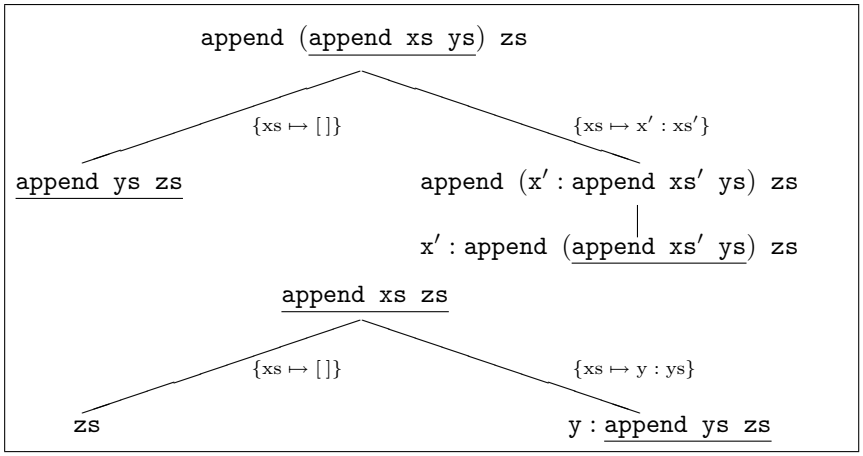


Figura 3.1: Arboles de *narrowing* necesario para `append (append xs ys) zs` y `append xs zs`.

junto con el conjunto de llamadas $S = \{\texttt{g x}, \texttt{h x}\}$. *A partir de las siguientes derivaciones de narrowing:*

$$\begin{array}{l} \underline{\texttt{g x}} \leadsto_{\texttt{id}} \texttt{S (}\underline{\texttt{f x}}\texttt{)} \leadsto_{\{\texttt{x}\mapsto\texttt{Z}\}} \texttt{(S Z)} \\ \underline{\texttt{h x}} \leadsto_{\{\texttt{x}\mapsto\texttt{S y}\}} \texttt{(S Z)} \end{array}$$

La pre–evaluación parcial de S *en* $\mathcal{R}$ *está formada por las siguientes reglas, que constituyen un nuevo programa* $\mathcal{R}'$*:*

$$\begin{array}{rcl} \texttt{g Z} & = & \texttt{S Z} \\ \texttt{h (S y)} & = & \texttt{S Z} \end{array}$$

Ahora bien, la ecuación $\texttt{h (g (S Z))} \approx \texttt{x}$ *tiene la siguiente derivación de éxito (por narrowing necesario) en* $\mathcal{R}$*:*

$$\begin{array}{lll} \texttt{h (}\underline{\texttt{g (S Z)}}\texttt{)} \approx \texttt{x} & \leadsto_{id} & \underline{\texttt{h (S (f (S Z)))}} \approx \texttt{x} \\ & \leadsto_{id} & \texttt{S Z} \approx \texttt{x} \\ & \leadsto^{*}_{\{\texttt{x}\mapsto\texttt{S z}\}} & \texttt{true} \end{array}$$

en tanto que la misma expresión produce un fallo en el programa especializado $\mathcal{R}'$*.*

El problema que se muestra en el Ejemplo 3.3 se debe a que los enlaces de las variables se *propagan hacia atrás*, i.e., a las partes izquierdas de los resultantes. En un contexto

de evaluación perezosa la instanciación de las variables en las partes izquierdas a partir de enlaces que provienen de la evaluación de términos que están encabezados por un símbolo constructor puede restringir, incorrectamente, el dominio de las funciones especializadas (en el ejemplo, la función "g"). En el Capítulo 5 se describe una solución introducida por [AHV00b] para remediar este problema.

3.3. El cierre de los programas especializados

En un programa especializado se debe garantizar que para toda posible llamada a función que puede ocurrir durante la ejecución de tal programa, exista una definición de función apropiada para realizar el cómputo correspondiente. La condición recursiva de *cierre* de un programa asegura que durante la ejecución del mismo cada llamada a función esté cubierta por alguna regla del programa resultante.

El cierre se formaliza verificando inductivamente que las diferentes llamadas a función en las partes derechas de las reglas estén suficientemente cubiertas por las funciones especializadas.

Informalmente, un término t encabezado por un símbolo de función definida se considera cerrado con respecto a un conjunto de llamadas S, si es una instancia de un término de S y los términos que aparecen en el codominio de la sustitución de emparejamiento están también cerrados recursivamente en S.

Ejemplo 3.4 *Considérese el conjunto de llamadas:*

$$\mathcal{S}_1 = \{\texttt{add Z y},\ \texttt{add (S Z) y}\}$$

La llamada a función `add x y` *no se puede considerar* cerrada *respecto a* $\mathcal{S}_1$ *porque no es instancia de ningún elemento de* S_1*. Por otro lado, si el conjunto de llamadas es:*

$$\mathcal{S}_2 = \{\texttt{add x y}\}$$

entonces la llamada a función `add (add Z v) w` *es cerrada respecto a* $\mathcal{S}_2$*. Veamos, para que empareje* `add (add Z v) w` *con* `add x y` *se necesita la sustitución* $\{\texttt{x} \mapsto \texttt{add Z v}\}$*. Además, el codominio de las sustituciones debe estar también cerrado, lo cual se cumple puesto que* `add Z v` *es una instancia de* `add x y`*.*

Definición 12 (cierre [AHLV05]**)** *Sea S un conjunto finito de términos. Decimos que un término t es S-cerrado si se cumple $closed(S, t)$, donde la función closed se*

define inductivamente como sigue:

$$closed(S,t) \Leftrightarrow \begin{cases} true & \text{si } t \in \mathcal{V} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \text{si } t = c(\overline{t_n}),\ c \in \mathcal{C}^*, \\ & n \geq 0 \\ \bigwedge_{x \mapsto t' \in \theta} closed(S,t') & \text{si } \exists s \in S \\ & \text{tal que } \theta(s) = t \\ & \text{para alguna} \\ & \text{sustitución } \theta \end{cases}$$

donde $\mathcal{C}^* = (\mathcal{C} \cup \{\approx, \wedge\})$.

Se dice que un conjunto de términos T *está* S*-cerrado, y se denota mediante* $closed(S,T)$*, si para todo* $t \in T$ *se cumple* $closed(S,t)$*, y decimos que un SRT* $\mathcal{R}$ *es* S*-cerrado si se cumple* $closed(S, \mathcal{R}_{calls})$.

Se usa $\mathcal{R}_{calls}$ *para denotar el conjunto de los términos en las partes derechas de las reglas de* $\mathcal{R}$.

Por ejemplo, la pre–evaluación parcial del Ejemplo 3.2 es cerrada con respecto al conjunto de llamadas evaluadas parcialmente: {`append (append xs ys) zs`, `append xs ys`}.

De acuerdo a la Definición 12 (que es indeterminista), una expresión encabezada por un símbolo de función "primitivo" (e.g., la conjunción $t_1 \wedge t_2$ o el símbolo de igualdad para formar una ecuación $t_1 \approx t_2$) se puede probar que es cerrada con respecto a S de dos formas: verificando que t_1 y t_2 son S-cerrados o probando que la conjunción (o ecuación) es una instancia de una llamada en S (seguida por una prueba inductiva de los subtérminos). Esto es útil cuando no se está interesado en especializar expresiones complejas (como conjunciones o ecuaciones) pero queremos que se puedan ejecutar después de la especialización. Nótese que esta forma de proceder es correcta ya que se considera que las reglas que definen las funciones primitivas "$\approx$" y "$\wedge$" se agregan automáticamente a cada programa, de ahí que las llamadas a esos símbolos estén convenientemente cubiertas en el programa especializado. En [AAF+98a] se puede encontrar una técnica general para tratar con símbolos primitivos la cual divide los términos de manera determinista antes de probar la condición de cierre en ellos.

3.4. Renombramiento

En general, dada una llamada s y un programa $\mathcal{R}$, existe un número infinito de pre–evaluaciones parciales diferentes de s en $\mathcal{R}$. Se asume una regla fija llamada *regla de desplegado* para generar resultantes, la cual determina las expresiones que se

deben derivar por *narrowing* (usando una estrategia de *narrowing* definida) y decide también, cuando parar la construcción de árboles de *narrowing* [AAF+98a, AFV98, AHV02].

En lo que sigue, denotamos con pre–EP–NN (donde EP–NN se obtiene a partir *E*valuación *P*arcial por *N*arrowing *N*ecesario) el conjunto de resultantes computados para S en $\mathcal{R}$ considerando una regla de desplegado que construye árboles de *narrowing* necesario finitos. Se usará el acrónimo EP–NN para las reglas renombradas las cuales serán el resultado del post-proceso de *renombramiento* correspondiente. La idea tras esta transformación es que, para cualquier llamada a función (cerrada respecto a un conjunto de llamadas considerado), las respuestas computadas para esa llamada en el programa original y las respuestas computadas por la llamada renombrada en el programa especializado (también renombrado) coinciden.

Para definir un evaluador parcial basado en *narrowing* necesario y para asegurar que el programa resultante sea inductivamente secuencial siempre que el original lo sea, se tiene que garantizar que el conjunto de términos especializados (después del renombramiento) contenga sólo patrones lineales con distintos símbolos en cabeza. Esta propiedad se logra mediante la introducción de un nuevo símbolo de función para cada término especializado y, entonces, reemplazando cada llamada en el programa especializado por una llamada a la función renombrada correspondiente.

Las partes izquierdas del programa especializado (las cuales son instancias constructoras de los término especializados) se reemplazan por instancias de los nuevos patrones lineales correspondientes por medio del renombramiento.

Definición 13 (renombramiento independiente [AHLV05]**)** *Un renombramiento independiente ρ para un conjunto de términos S es una función de términos a términos que se define como sigue: para $s \in S$, $\rho(s) = f_s(\overline{x_n})$, donde $\overline{x_n}$ son las distintas variables en s en el orden de aparición y f_s es un símbolo de función nuevo, que no aparece en $\mathcal{R}$ o S y es diferente al símbolo en cabeza de cualquier otro $\rho(s')$, con $s' \in S$ y $s' \neq s$. Además, se denota con $\rho(S)$ al conjunto $S' = \{\rho(s) \mid s \in S\}$.*

Veamos a continuación, un ejemplo:

Ejemplo 3.5 *Si se considera el conjunto*

$$S = \{\texttt{append (append xs ys) zs},\ \texttt{append xs ys}\}$$

El renombramiento independiente con respecto a S es:

$$\begin{aligned}\rho = \{\ &\texttt{append xs ys} \mapsto \texttt{app xs ys},\\ &\texttt{append (append xs ys) zs} \mapsto \texttt{dapp xs ys zs}\ \}\end{aligned}$$

Mientras el renombramiento independiente es suficiente para renombrar las partes izquierdas de los resultantes (ya que son instancias constructoras de las llamadas especializadas) las partes derechas se renombran por medio de la función auxiliar ren_ρ, la cual reemplaza *recursivamente* cada llamada en la expresión dada por una llamada a la función renombrada correspondiente (de acuerdo a ρ).

Definición 14 (función de renombramiento [AHLV05]**)** *Sea S un conjunto finito de términos y ρ una función de renombramiento independiente de S. Dado un término t, la función indeterminista ren_ρ se define inductivamente como sigue:*

$$ren_\rho(t) = \begin{cases} t & \text{si } t \in \mathcal{V} \\ c(\overline{ren_\rho(t_n)}) & \text{si } t = c(\overline{t_n}),\ c \in \mathcal{C}^*, \text{ y } \ n \geq 0 \\ \theta'(\rho(s)) & \text{si } \exists \theta, \exists s \in S \text{ tal que } t = \theta(s) \text{ y} \\ & \quad \theta' = \{x \mapsto ren_\rho(\theta(x)) \mid x \in \mathcal{D}om(\theta)\} \\ t & \text{en caso contrario} \end{cases}$$

donde $\mathcal{C}^* = (\mathcal{C} \cup \{\approx, \wedge\})$.

Al igual que la prueba del cierre, una ecuación $s \approx t$ se puede renombrar (de manera indeterminista) por el renombramiento independiente de s y t o bien mediante el reemplazo de la ecuación que se considera por una llamada a la nueva función renombrada correspondiente (si la ecuación es una instancia de alguna llamada especializada en S). Nótese, además, que la función de renombramiento es una función *total*: si un término t encabezado por un símbolo de función no es una instancia de ningún término en S (lo cual puede ocurrir si t no es S-cerrado), la función $ren_\rho(t)$ devuelve el mismo término t (i.e., t no se renombra).

3.5. Evaluación parcial

A continuación se define formalmente la noción de evaluación parcial:

Definición 15 (evaluación parcial [AHLV05]**)** *Sea $\mathcal{R}$ un SRT, S un conjunto finito de términos y $\mathcal{R}'$ una pre–evaluación parcial de $\mathcal{R}$ con respecto a S. Sea ρ un renombramiento independiente de S. La evaluación parcial $\mathcal{R}''$ de $\mathcal{R}$ con respecto a S (bajo ρ) se define como sigue:*

$$\mathcal{R}'' = \bigcup_{s \in S} \{\theta(\rho(s)) \to ren_\rho(r) \mid \theta(s) \to r \in \mathcal{R}' \text{ es un resultante para } s \text{ en } \mathcal{R}\}$$

Ejemplo 3.6 *Consideremos el programa* `append` *y el conjunto de términos* S *del Ejemplo 3.2, junto con el renombramiento independiente* ρ *del Ejemplo 3.5. Una evaluación parcial* $\mathcal{R}'$ *de* $\mathcal{R}$ *con respecto a* S *(bajo* ρ*) es:*

```
dapp [ ] ys zs     =  app ys zs
dapp (x : xs) ys zs =  x : (dapp xs ys zs)
app [ ] ys         =  ys
app (x : xs) ys    =  x : (app xs ys)
```

Para un renombramiento ρ, la forma renombrada de un programa $\mathcal{R}$ puede depender de la estrategia que se emplea para seleccionar el término de $\rho(S)$ el cual se usa para renombrar una llamada t dada en $\mathcal{R}$ (e.g., `append (append xs ys) zs`), puesto que, en general, puede existir más de un término en S que cubre la llamada t. Una elección inconveniente podría hacer que se perdiera algo de la potencia de especialización. Para obtener el mejor potencial de la especialización se hace uso de heurísticas apropiadas; en [AHV02] se introducen algunas de tales técnicas que son empleadas en la implementación de un evaluador parcial para programas lógico funcionales.

El teorema que sigue establece una propiedad muy importante de la evaluación parcial basada en *narrowing* necesario: si el programa de entrada es inductivamente secuencial, entonces el programa evaluado parcialmente es también inductivamente secuencial; con ello, para evaluar las llamadas en el programa especializado se puede aplicar también la estrategia de *narrowing* necesario.

Teorema 16 [AHLV05] *Sea* $\mathcal{R}$ *un programa inductivamente secuencial y* S *un conjunto finito de términos con símbolo de función en cabeza. Entonces, cada EP–NN de* $\mathcal{R}$ *con respecto a* S *es inductivamente secuencial.*

Otra propiedad interesante de la evaluación parcial dirigida por *narrowing* necesario se formaliza en el teorema que sigue. El teorema garantiza que un término que se puede normalizar de manera determinista con respecto al programa original, no puede causar una evaluación indeterminista con respecto al programa especializado obtenido por EP–NN.

Teorema 17 [AHLV05] *Sea* $\mathcal{R}$ *un programa inductivamente secuencial,* S *un conjunto finito de términos con símbolo de función en cabeza,* ρ *un renombramiento independiente de* S*, y* e *una ecuación. Sea* $\mathcal{R}'$ *un EP–NN de* $\mathcal{R}$ *con respecto a* S *(bajo* ρ*) tal que* $\mathcal{R}' \cup \{e'\}$ *es* S'*-cerrado, donde* $e' = ren_\rho(e)$ *y* $S' = \rho(S)$*. Si* e *se reduce de manera determinista a true con respecto a* $\mathcal{R}$*, entonces* e' *se reduce de manera determinista a true con respecto a* $\mathcal{R}'$*.*

Desde un punto de vista de implementación, la propiedad definida en el Teorema 17 es deseable e importante en los programas especializados, ya que la implementación

de pasos indeterministas es una operación costosa en los lenguajes lógicos. Inclusive, el indeterminismo en los programas especializados puede repercutir en derivaciones infinitas, lo cual podría tener el efecto de que algunas soluciones ya no sean computables en una implementación secuencial basada en *backtracking*. Esencialmente, las computaciones deterministas se preservan gracias al uso de *narrowing* necesario para realizar computaciones parciales sobre programas inductivamente secuenciales.

Ejemplo 3.7 *Considérese la función "$\leqslant$" junto con la función simple* `foo`*:*

$$\begin{array}{rcl} \texttt{foo Z} & = & \texttt{Z} \\ \texttt{Z} \leqslant \texttt{y} & = & \texttt{true} \\ \texttt{(S x)} \leqslant \texttt{Z} & = & \texttt{false} \\ \texttt{(S x)} \leqslant \texttt{(S y)} & = & \texttt{x} \leqslant \texttt{y} \end{array}$$

Dada una llamada a función de la forma $\texttt{x} \leqslant \texttt{foo y}$*, algunas estrategias de narrowing (e.g., narrowing perezoso) tienen dos caminos para proceder: primero, reduciendo la llamada a la función "$\leqslant$" usando la primera regla*

$$\texttt{x} \leqslant \texttt{foo y} \quad \leadsto_{\{\texttt{x} \mapsto \texttt{Z}\}} \quad \texttt{true}$$

y, la segunda, reduciendo la llamada a la función `foo` *(la cual se requiere por la segunda y tercera reglas de "$\leqslant$"):*

$$\texttt{x} \leqslant \texttt{foo y} \quad \leadsto_{\{\texttt{y} \mapsto \texttt{Z}\}} \quad \texttt{x} \leqslant \texttt{Z}$$

De esta manera, sus resultantes asociados son como sigue:

$$\begin{array}{rcl} \texttt{Z} \leqslant \texttt{foo y} & = & \texttt{true} \\ \texttt{x} \leqslant \texttt{foo Z} & = & \texttt{x} \leqslant \texttt{Z} \end{array}$$

Ahora, dada una llamada de la forma $\texttt{Z} \leqslant \texttt{foo w}$*, los dos resultantes se pueden aplicar; sin embargo, el segundo es cláramente redundante. De hecho, el segundo resultante sólo es significativo para evaluar aquellas llamadas cuyo primer argumento es de la forma* $(\texttt{S} \cdots)$*, ya que sólo la segunda y tercera reglas de "$\leqslant$" demandaron la evaluación de la llamada a* `foo Z` *que dio lugar a este resultante. La ventaja de usar narrowing necesario es que éste aplica algunos enlaces adicionales para que esta información sea explícita en los resultantes computados; e.g., los resultantes obtenidos por narrowing necesario son:*

$$\begin{array}{rcl} \texttt{Z} \leqslant \texttt{foo y} & = & \texttt{true} \\ \underline{\texttt{S w}} \leqslant \texttt{foo Z} & = & \texttt{S w} \leqslant \texttt{Z} \end{array}$$

De esta manera, se evita la introducción adicional de indeterminismo.

Para finalizar, en [AHLV05] se presentan de manera formal las propiedades fundamentales del esquema de evaluación parcial.

Primero, prueban la corrección (respecto a la completitud) de la transformación, i.e., se prueba que para cada respuesta computada por *narrowing* necesario en el programa original (respecto al programa especializado) existe una respuesta más general en el programa especializado (respecto al programa original) para los términos considerados. Como resultado final concluyen la corrección fuerte de la evaluación parcial dirigida por *narrowing*, i.e., las respuestas computadas en el programa original y el programa especializado coinciden (módulo renombramiento de variables).

La proposición que sigue es clave en la prueba de corrección [AHLV05] de la transformación EP–NN.

Proposición 18 [AHLV05] *Sea $\mathcal{R}$ un programa inductivamente secuencial. Sea e una ecuación, S un conjunto finito de términos con símbolo de función en cabeza, ρ un renombramiento independiente de S, y $\mathcal{R}'$ una EP–NN de $\mathcal{R}$ con respecto a S (bajo ρ) tal que $\mathcal{R}' \cup \{e'\}$ es S'-cerrado, donde $e' = ren_\rho(e)$ y $S' = \rho(S)$. Si $e' \rightarrow^* true$ en $\mathcal{R}'$ entonces $e \rightarrow^* true$ en $\mathcal{R}$.*

Enseguida se formula la corrección:

Teorema 19 (corrección [AHLV05]**)** *Sea $\mathcal{R}$ un programa inductivamente secuencial. Sea e una ecuación, $V \supseteq \mathcal{V}ar(e)$ un conjunto finito de variables, S un conjunto finito de términos con símbolo de función en cabeza, y ρ un renombramiento independiente de S. Sea $\mathcal{R}'$ una EP–NN de $\mathcal{R}$ con respecto a S (bajo ρ) tal que $\mathcal{R}' \cup \{e'\}$ es S'-cerrado, donde $e' = ren_\rho(e)$ y $S' = \rho(S)$. Si $e' \leadsto^*_{\sigma'} true$ es una derivación por narrowing necesario para e' en $\mathcal{R}'$, entonces existe una derivación por narrowing necesario $e \leadsto^*_\sigma true$ en $\mathcal{R}$ tal que $(\sigma \leq \sigma')$ $[V]$.*

De manera similar, la completitud se apoya en la proposición que sigue:

Proposición 20 [AHLV05] *Sea $\mathcal{R}$ un programa inductivamente secuencial. Sea e una ecuación, S un conjunto finito de términos con símbolo de función en cabeza, ρ un renombramiento independiente de S, y $\mathcal{R}'$ una EP–NN de $\mathcal{R}$ con respecto a S (bajo ρ) tal que $\mathcal{R}' \cup \{e'\}$ es S'-cerrado, donde $e' = ren_\rho(e)$ y $S' = \rho(S)$. Si $e \rightarrow^* true$ en $\mathcal{R}$ entonces $e' \rightarrow^* true$ en $\mathcal{R}'$.*

La completitud de la transformación EP–NN es una consecuencia directa de la proposición previa y de la corrección y completitud del *narrowing* necesario.

Teorema 21 (completitud [AHLV05]**)** *Sea $\mathcal{R}$ un programa inductivamente secuencial. Sea e una ecuación, $V \supseteq \mathcal{V}ar(e)$ un conjunto finito de variables, S un conjunto*

finito de términos con símbolo de función en cabeza, y ρ un renombramiento independiente de S. Sea $\mathcal{R}'$ una EP–NN de $\mathcal{R}$ con respecto a S (bajo ρ) tal que $\mathcal{R}' \cup \{e'\}$ es S'-cerrado, donde $e' = ren_\rho(e)$ y $S' = \rho(S)$. Si $e \leadsto^*_\sigma$ true es una derivación por narrowing necesario para e en $\mathcal{R}$, entonces existe una derivación de narrowing necesario $e' \leadsto^*_{\sigma'}$ true en $\mathcal{R}'$ tal que $(\sigma' \leq \sigma)$ $[V]$.

Finalmente, se postula la corrección fuerte de la transformación EP–NN. Se prueba como una consecuencia directa del teorema de la corrección y completitud, junto con la independencia de soluciones que se computan por *narrowing* necesario.

Teorema 22 (corrección fuerte [AHLV05]**)** *Sea $\mathcal{R}$ un programa inductivamente secuencial. Sea e una ecuación, $V \supseteq \mathcal{V}ar(e)$ un conjunto finito de variables, S un conjunto finito de términos con símbolo de función en cabeza, y ρ un renombramiento independiente de S. Sea $\mathcal{R}'$ una EP–NN de $\mathcal{R}$ con respecto a S (bajo ρ) tal que $\mathcal{R}' \cup \{e'\}$ es S'-cerrado, donde $e' = ren_\rho(e)$ y $S' = \rho(S)$. Entonces, $e \leadsto^*_\sigma$ true es una derivación de narrowing necesario para e en $\mathcal{R}$ sii existe una derivación de narrowing necesario $e' \leadsto^*_{\sigma'}$ true en $\mathcal{R}'$ tal que $(\sigma' = \sigma)$ $[V]$ (módulo renombramiento de variables).*

3.6. Aspectos de control

El método para la evaluación parcial de programas lógico funcionales dirigida por *narrowing* [AV02] consiste básicamente de la construcción iterativa de árboles parciales de *narrowing* hasta que todas sus hojas estén cubiertas (cerradas) por los nodos raíz. La formulación original del esquema NPE [AFV96, AFV98, Vid96] es paramétrica con respecto a la *relación de narrowing* que se usa para la construcción de los árboles parciales, la *regla de desplegado* que determina cuándo y cómo terminar la construcción de esos árboles y el *operador de generalización* que se usa para garantizar que el número de árboles se mantiene finito.

Las técnicas que se introducen en [AFV98] para el control de la terminación del método de evaluación parcial NPE, pueden verse como una adaptación de ciertos formalismos empleados para el control de la terminación en campos como la deducción parcial y la supercompilación positiva.

A continuación describimos uno de los formalismos empleados: la relación de subsumción homeomórfica, detallamos el algoritmo de evaluación parcial NPE, y finalmente nos centramos en los aspectos de control local y control global.

3.6.1. La relación de subsumción homeomórfica

Una herramienta común para probar propiedades de terminación se basa en la noción intuitiva de órdenes, en los que un término que es "sintácticamente más simple" que otro, entonces es también, menor que el otro en ese orden. Un buen preorden interesante es la relación de subsumción homeomórfica [Leu02]. La definición siguiente extiende la relación de subsumción homeomórfica de [DJ90] a términos no básicos y formaliza la noción de "sintácticamente más simple".

Definición 23 (relación de subsumción homeomórfica [SG95]**)** *La relación de subsumción homeomórfica $\trianglelefteq$ sobre términos de $\mathcal{T}(\mathcal{F},\mathcal{V})$ se define como la relación más pequeña que satisface que $x \trianglelefteq y$ para toda $x, y \in \mathcal{V}$ y $s = f(s_1,\ldots,s_m) \trianglelefteq g(t_1,\ldots,t_n) = t$, si y sólo si:*

1. *$f = g$ (con $m = n$) y $s_i \trianglelefteq t_i$ para todo $i = 1,\ldots,n$ o*
2. *$s \trianglelefteq t_j$, para algún j, $1 \leq j \leq n$.*

Informalmente, un término t_1 *subsume* a un término t_2 si t_2 puede obtenerse a partir de t_1 mediante la eliminación de algunos operadores, e.g.,

$$\mathtt{S}\ (\underline{\mathtt{S}}\ ((\underline{\mathtt{u}} + \mathtt{w}) \underline{\times} (\underline{\mathtt{u}} \underline{+} (\mathtt{S}\ \underline{\mathtt{v}}))))$$

subsume a $\mathtt{S}\ (\mathtt{u} \times (\mathtt{u} + \mathtt{v}))$. El siguiente resultado es una reformulación del teorema de Kruskal (*Kruskal's Tree Theorem*).

Teorema 24 [AFV98] *La relación de subsumción homeomórfica $\trianglelefteq$ es un buen preorden sobre el conjunto $\mathcal{T}(\mathcal{F},\mathcal{V})$ para signaturas $\mathcal{F}$ finitas, esto es, $\trianglelefteq$ es un preorden (i.e., una relación binaria reflexiva y transitiva) y, para cualquier secuencia infinita de términos $t_1, t_2, \ldots$ con un número finito de operadores, existen j, k con $j < k$ y $t_j \trianglelefteq t_k$ (i.e., la secuencia se auto-subsume).*

3.6.2. El algoritmo básico

El procedimiento, que opera en el evaluador parcial NPE *online*, se formaliza en la Figura 3.2. A grandes rasgos, el algoritmo procede como sigue. Dado un programa de entrada $\mathcal{R}$ y un conjunto de términos T, la primera etapa consiste en aplicar una regla de desplegado para computar árboles de *narrowing* finitos (posiblemente incompletos) para esos términos; el algoritmo devuelve el conjunto de resultantes—i.e., el programa—asociado a las derivaciones que van de la raíz a las hojas de tales árboles. Entonces, se aplica un operador de generalización para agregar los términos en las partes derechas de los resultantes al conjunto de términos que ya han sido evaluados

Entrada: un programa $\mathcal{R}$ y un conjunto de términos T
Salida: un conjunto de términos S
Inicialización: $i := 0;\ T_0 := T$
Repetir
$\mathcal{R}' := \mathit{desplegar}(T_i, \mathcal{R})$;
$T_{i+1} := \mathit{generalizar}(T_i, \mathcal{R}'_{\mathit{calls}})$;
$i := i + 1$;
Hasta $T_i = T_{i-1}$ (módulo renombramiento de variables)
Devolver: $S := T_i$

Figura 3.2: Algoritmo básico de evaluación parcial dirigida por *narrowing*.

parcialmente; la fase de generalización produce un nuevo conjunto de términos, los cuales pueden requerir de evaluación y, de esa manera, el proceso se repite mientras se introduzcan nuevos términos. Hay que notar que el algoritmo no devuelve un programa evaluado parcialmente sino un conjunto de términos, los cuales determinan sin ambigüedad la evaluación parcial asociada. En particular, aplicando una vez más la misma regla de desplegado, se generan los resultantes correspondientes que forman el programa residual. Enseguida se aplica el post-proceso de renombramiento sobre el programa residual.

El procedimiento sigue el estilo del método de deducción parcial de Martens y Gallagher [MG95] donde se distinguen con claridad dos niveles de control: el *nivel local*—el cual es operado por la regla de desplegado—y el *nivel global*—que se controla con el operador de generalización. Esto es, para asegurar la terminación del algoritmo, se tiene que garantizar por un lado, la terminación *local* y por otro, la terminación *global*, i.e., los árboles de *narrowing* parciales tienen que ser finitos y la construcción iterativa de los árboles parciales tiene también que concluir.

3.6.3. El control local y el control global

Con el fin de asegurar la terminación local del algoritmo, la regla de desplegado tiene que incorporar algún mecanismo para detener la construcción de los árboles de *narrowing*. Con este propósito existen varias técnicas, e.g., profundidad-acotada, verificación de ciclos [Bol93], órdenes bien fundados [BSM92], buenos preórdenes [SG95], etc. En el contexto de la aproximación dirigida por *narrowing*, las reglas de desplegado se han definido mediante el uso del tipo particular de buen preorden de subsumción homeomórfica (véase Sección 3.6.1).

Las reglas de desplegado basadas en un preorden de subsumción homeomórfica permiten la expansión de las derivaciones de *narrowing* hasta alcanzar un término

que subsume a alguno de los términos previos en la misma derivación [AAF+98a, AFV96, AFV98].

El control global no puede manejarse con la misma flexibilidad que el control local ya que repercute en la corrección del método. En el nivel local, esta flexibilidad permite detener la construcción de un árbol en cualquier punto preservando la corrección. En contraste, no se puede detener la construcción iterativa de árboles parciales hasta que todas sus hojas estén cerradas con respecto al conjunto correspondiente de sus nodos raíz. Esta condición es necesaria para asegurar que la evaluación parcial resultante esté cerrada y, de ahí, se garantice la corrección. Por otro lado, puede pasar que esta condición nunca se alcance y, en este caso, el proceso iterativo se ejecutaría infinitamente. Por lo tanto, el control global, incluye usualmente algún tipo de generalización para forzar la terminación del proceso. El operador de generalización más popular es el de la *generalización más específica*: *msg* (del inglés *most specific generalization*) entre términos. Se dice que un término t es una *generalización* de los términos t_1 y t_2 si ambos t_1 y t_2 son instancias de t. Además, el término t es el *msg* de t_1 y t_2 si t es una generalización de t_1 y t_2 y, para cualquier otra generalización t' de t_1 y t_2, t' es una instancia de t.

En el contexto de la aproximación de evaluación parcial dirigida por *narrowing*, los operadores de abstracción se apoyan en el orden de subsumción homeomórfica para poder decidir cuándo generalizar y cuándo continuar con la construcción iterativa de árboles parciales de *narrowing*. El operador de generalización *generalizar* toma dos conjuntos de términos (los términos que ya han sido evaluados parcialmente T_i y los términos que se van a agregar a este conjunto, $\mathcal{R}'_{calls}$, como se muestra en la Figura 3.2) y devuelve una aproximación *correcta* de $T_i \cup \mathcal{R}'_{calls}$. Por corrección se entiende que cada término en $T_i \cup \mathcal{R}'_{calls}$ es cerrado con respecto al conjunto de términos resultantes a partir de $generalizar(T_i, \mathcal{R}'_{calls})$. Con la finalidad de agregar un nuevo término, t, al conjunto actual de términos evaluados parcialmente, S, los operadores de generalización proceden del modo que sigue [AFV96, AFV98]:

- si t no subsume ningún término en S, entonces t se agrega a S;
- si t subsume algún término en S, e.g., t', se distinguen dos casos:
  - si t ya es S-cerrado, entonces se descarta;
  - de otro modo, ambos términos se generalizan mediante el cómputo del *msg* de t y t', llámese t'' y, entonces, el operador de generalización se aplica recursivamente para agregar t'' así como los términos en la sustitución de emparejamiento (i.e., los términos en σ y en σ', con $\sigma(t'') = t$ y $\sigma'(t'') = t'$).

La aplicción del *msg* implica una pérdida de información debido a la generalización de los términos, por ello su uso se retarda hasta que es inevitable.

3.7. Conclusiones

En este capítulo se han introducido los conceptos y resultados del esquema de evaluación parcial dirigido por *narrowing*. En particular, tales resultados tienen que ver con la corrección y la completitud del método con respecto a la estrategia de *narrowing* necesario. También se ha introducido el algoritmo general para el proceso de la evaluación parcial y los controles asociados, i.e., el control local y el control global.

Es importante la presentación de este marco para contextualizar, en este documento, la extensión al método que se propone en el Capítulo 4: la que llamamos evaluación parcial *offline* dirigida por *narrowing*.

Capítulo 4

Evaluación parcial *offline* dirigida por *narrowing*

En el marco de la evaluación parcial dirigida por *narrowing* (NPE [AV02]) el mecanismo de computación simbólica que se utiliza para realizar las computaciones parciales durante el proceso de especialización es el propio mecanismo de *narrowing*. En este capítulo introducimos una nueva aproximación *offline* al esquema NPE. Parte de los desarrollos presentados en este capítulo están publicados en [RSV05d, RSV05b, RSV07].

4.1. Antecedentes

Los evaluadores parciales se clasifican en dos grandes categorías: *online* y *offline*, de acuerdo al momento temporal en que se consideran los aspectos de terminación. Los evaluadores parciales *online* son usualmente más precisos ya que disponen de más información. En particular, el esquema original NPE, el cual sigue la aproximación *online*, considera una variante del teorema de Kruskal (*Kruskal's Tree Theorem*) llamada *subsumción homeomórfica* [Leu02] para asegurar la terminación del proceso: si un término subsume algún término previo en la misma computación de *narrowing*, se aplica alguna forma de generalización—usualmente el operador de generalización más específica—y la evaluación parcial se reinicia con los términos generalizados [AFV98]. Sin embargo, esta precisión adicional tiene un coste asociado: las pruebas para el test de subsumción homeomórfica, junto con las generalizaciones asociadas, hacen que el esquema NPE *online* sea muy costoso (en términos de tiempo y espacio); por ello, no se adapta adecuadamente a problemas realistas como la especialización de intérpretes

[Jon04].

Los evaluadores parciales *offline* proceden usualmente en dos etapas; la primera etapa procesa un programa e incluye anotaciones para guiar las computaciones parciales (e.g., para identificar aquellas llamadas a función que pueden desplegarse sin riesgo de no terminación); entonces, una segunda etapa, la de evaluación parcial propiamente dicha, sólo debe obedecer las anotaciones y, por tanto, el especializador es mucho más rápido que en la aproximación *online*.

En las siguientes secciones planteamos los principios necesarios para el desarrollo de una aproximación *offline* a la evaluación parcial dirigida por *narrowing*. Mientras que el esquema original NPE garantiza la cuasi-terminación de las computaciones a partir de la aplicación de varios tests de terminación y operadores de generalización de manera *online* (y de ese modo conduce a la terminación del proceso de especialización). Nosotros, desde una perspectiva distinta, vamos a introducir una caracterización de programas que asegura que las computaciones de *narrowing* son siempre cuasi-terminantes. Esta caracterización define una clase muy limitada de programas y, por ello, planteamos un esquema de anotación válido para la clase de programas inductivamente secuenciales. Para procesar las anotaciones incluimos una extensión del *narrowing* que llamamos *narrowing generalizante*, presentamos varios ejemplos de la técnica planteada y, finalmente, discutimos su eficiencia.

4.2. Cuasi-terminación de las computaciones de *narrowing* necesario

En el núcleo del esquema NPE encontramos un método para construir una representación finita de un espacio de computación (normalmente) infinito. Para ser precisos, dado un sistema de reescritura $\mathcal{R}$ y un término t, NPE construye una representación finita de todas las derivaciones posibles de t—y de cualquiera de sus instancias si t contiene variables—en $\mathcal{R}$, y entonces, extrae un sistema de reescritura nuevo, frecuentemente más simple y eficiente que el original. Dado que t puede contener variables, se requiere alguna forma de *computación simbólica*. En el marco NPE se usa un refinamiento de *narrowing* para procesar las computaciones simbólicas; en particular, *narrowing* necesario [AEH00] es la estrategia que presenta mejores propiedades (como se demuestra en [AHLV05]). En general, el espacio de *narrowing* de un término puede ser infinito. Sin embargo, aún en ese caso, si el programa es cuasi-terminante con respecto a la estrategia de *narrowing* considerada, NPE puede todavía terminar. La razón se debe a que la evaluación parcial de varias ocurrencias del mismo término (módulo renombramiento de variables) en una computación se puede evitar mediante la inserción de una llamada residual a alguna de las variantes que se han encontrado

previamente. Esta técnica se conoce como *inserción de puntos de especialización* en la literatura de evaluación parcial [GJ05].

Para el planteamiento de la cuasi-terminación requerimos las siguientes definiciones preparatorias:

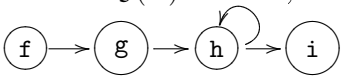
Definición 25 (grafo de dependencias funcionales) *Dado un SRT $\mathcal{R}$, su grafo de dependencias funcionales, en símbolos $\mathcal{G}(\mathcal{R})$, contiene nodos etiquetados con los símbolos de función en $\mathcal{D}$ y existe un arco del nodo f al nodo g sii hay una llamada a la función g en la parte derecha de alguna de las reglas en la definición de la función f.*

Definición 26 (función cíclica, función no cíclica) *Sea $\mathcal{R}$ un SRT. Una función $f \in \mathcal{D}$ es cíclica si el nodo f forma parte de un ciclo en $\mathcal{G}(\mathcal{R})$ y en caso contrario es no cíclica.*

Ejemplo 4.1 *Consideremos el siguiente SRT $\mathcal{R}$:*

$$\begin{array}{rcl} \texttt{f (S x) y} & = & \texttt{g x y} \\ \texttt{g x (S y)} & = & \texttt{h x y} \\ \texttt{h Z y} & = & \texttt{y} \\ \texttt{h (S x) y} & = & \texttt{C (i x) (h x y)} \\ \texttt{i x} & = & \texttt{x} \end{array}$$

donde $\texttt{f}, \texttt{g}, \texttt{h}, \texttt{i} \in \mathcal{D}$ son funciones definidas y $\texttt{Z}, \texttt{S}, \texttt{C} \in \mathcal{C}$ son símbolos constructores. El grafo de dependencias funcionales $\mathcal{G}(\mathcal{R})$ asociado, es como sigue:



En el grafo observamos que la función h *es cíclica mientras que* f, g, i *son no cíclicas.*

Las funciones que no están en un ciclo no pueden introducir computaciones no terminantes (ni cuasi-terminantes) por si mismas siempre que las funciones cíclicas (a las que invocan) no las introduzcan. Por lo tanto, centraremos nuestra atención en las funciones cíclicas. A continuación introducimos la definición de *profundidad de una variable*:

Definición 27 (profundidad de una variable [CK96]**)** *La profundidad de una variable x en un término constructor t, en símbolos $dv(t, x)$, se define como sigue:*

$$\begin{array}{rcll} dv(c(\overline{t_n}), x) & = & 1 + max(\overline{dv(t_n, x)}) & \text{si } x \in \mathcal{V}ar(c(\overline{t_n})) \\ dv(c(\overline{t_n}), x) & = & -1 & \text{si } x \notin \mathcal{V}ar(c(\overline{t_n})) \\ dv(y, x) & = & 0 & \text{si } x = y \ e \ y \in \mathcal{V} \\ dv(y, x) & = & -1 & \text{si } x \neq y \ e \ y \in \mathcal{V} \end{array}$$

donde $c \in \mathcal{C}$ es un símbolo constructor con aridad $n \geqslant 0$.

En la siguiente definición introducimos la noción de función *no creciente*, la cual, informalmente hablando, es una función que siempre *consume* sus parámetros o los deja inalterados:

Definición 28 (función no creciente) *Sea $\mathcal{R}$ un SRT constructor lineal por la izquierda. Una función $f \in \mathcal{D}$ es no creciente sii cada regla $f(\overline{s_n}) \to r$ en la definición de f satisface las siguientes condiciones:*

1. *la parte derecha de la regla no contiene símbolos de función definida anidados (i.e., símbolos de función definida que aparezcan en los argumentos de otro símbolo de función definida) y*
2. *$dv(s_i, x) \geqslant dv(t_j, x)$ para todos los subtérminos que están encabezados por un símbolo de función definida $g(\overline{t_m})$ en r, donde $i \in \{1, \ldots, n\}$, $x \in \mathcal{V}ar(s_i)$ y $j \in \{1, \ldots, m\}$.*

Ejemplo 4.2 *Una función definida por la única regla*

$$\mathtt{f\ x\ y\ (S\ z)} = \mathtt{C\ (g\ x)\ (h\ z)}$$

con $\mathtt{S}, \mathtt{C} \in \mathcal{C}$ y $\mathtt{f}, \mathtt{g}, \mathtt{h} \in \mathcal{D}$, es no creciente ya que se cumplen las siguientes relaciones:

$$\begin{array}{rcl} dv(\mathtt{x},\mathtt{x}) = \mathtt{0} & \geqslant & 0 = dv(\mathtt{x},\mathtt{x}) \\ dv(\mathtt{x},\mathtt{x}) = \mathtt{0} & \geqslant & -1 = dv(\mathtt{z},\mathtt{x}) \end{array}$$

$$\begin{array}{rcl} dv(\mathtt{y},\mathtt{y}) = \mathtt{0} & \geqslant & -1 = dv(\mathtt{x},\mathtt{y}) \\ dv(\mathtt{y},\mathtt{y}) = \mathtt{0} & \geqslant & -1 = dv(\mathtt{z},\mathtt{y}) \end{array}$$

$$\begin{array}{rcl} dv((\mathtt{S\ z}),\mathtt{z}) = \mathtt{1} & \geqslant & -1 = dv(\mathtt{x},\mathtt{z}) \\ dv((\mathtt{S\ z}),\mathtt{z}) = \mathtt{1} & \geqslant & 0 = dv(\mathtt{z},\mathtt{z}) \end{array}$$

Intuitivamente, la variable x *sólo se copia, la variable* y *no aparece en el lado derecho de la regla y (la profundidad de) la variable* z *decrece.*

Definición 29 (derivación cuasi-terminante) *Una derivación de un término t con respecto a una estrategia de narrowing es cuasi-terminante si contiene sólo un número finito de términos diferentes (módulo renombramiento de variables).*

Ejemplo 4.3 *Consideremos el SRT $\mathcal{R}$ de la típica función* `length` *para computar la longitud de una lista:*

$$\begin{array}{rcl} \mathtt{length\ [\,]} & = & \mathtt{Z} \\ \mathtt{length\ (x : xs)} & = & \mathtt{S\ (length\ xs)} \end{array}$$

donde `S` *y* `Z` $\in \mathcal{C}$. *Dado el término inicial* `length x`, *al computar por narrowing necesario tenemos:*

$$\begin{array}{lll} \texttt{length x} & \leadsto_{\{\texttt{x} \mapsto \texttt{x}':\texttt{xs}'\}} & \texttt{S (}\underline{\texttt{length xs}'}\texttt{)} \\ & \leadsto_{\{\texttt{xs}' \mapsto \texttt{x}'':\texttt{xs}''\}} & \texttt{S (S (}\underline{\texttt{length xs}''}\texttt{))} \\ & \leadsto & \ldots \end{array}$$

i.e., la derivación de `length x` *en* $\mathcal{R}$ *con respecto a narrowing necesario es cuasi-terminante.*

Análogamente a Dershowitz [Der87], decimos que un SRT es *cuasi-terminante para un conjunto de términos* T *con respecto a narrowing necesario* sii todas las derivaciones por *narrowing* necesario que se producen a partir de los términos en T son cuasi-terminantes. A continuación enunciamos una condición suficiente para la cuasi-terminación:

Definición 30 (SRT no creciente) *Sea* $\mathcal{R}$ *un SRT inductivamente secuencial.* $\mathcal{R}$ *es no creciente sii todas las funciones* $f \in \mathcal{D}$ *son lineales por la derecha y no cíclicas o no crecientes.*

Aunque en la Definición 30 solo consideramos programas inductivamente secuenciales, la restricción a SRTs inductivamente secuenciales no es realmente necesaria (i.e., bastaría con SRTs constructores lineales por la izquierda). Sin embargo, imponemos esta condición porque el *narrowing* necesario solamente está definido para esta clase de SRTs. Por otro lado, la linealidad por la derecha es necesaria, no sólo para garantizar la cuasi-terminación (como veremos más adelante), sino también para asegurar que el desplegado de funciones no introduce computaciones repetidas.

A continuación formulamos nuestro resultado de cuasi-terminación:

Teorema 31 *Si* $\mathcal{R}$ *es un SRT no creciente, entonces* $\mathcal{R}$ *es cuasi-terminante para cualquier término lineal con respecto a narrowing necesario.*

Para demostrar el Teorema 31, necesitamos primero algunas nociones preliminares:

Dado un SRT no creciente $\mathcal{R}$ y una función no cíclica $f \in \mathcal{D}$, denotamos mediante $path(f)$ la longitud del camino más largo desde el nodo f en $\mathcal{G}(\mathcal{R})$ hasta un nodo terminal (i.e., un nodo sin arcos salientes) o bien a un ciclo; para una función cíclica f, establecemos $path(f) = 0$. Definimos ahora la *complejidad*: $comp(t)$, de un término t como el *multiconjunto* finito $\langle path(f_1), \ldots, path(f_n) \rangle$, donde $f_1, \ldots, f_n$ son los símbolos de función definida de t. La complejidad de un término proporciona información acerca de la *distancia* de este término hasta el punto en el que la computación se termina o bien entra en un ciclo.

Consideremos ahora el orden total bien fundado $<_{mul}$ sobre multiconjuntos de complejidades mediante la extensión del orden bien fundado $<$ sobre los naturales $\mathbb{N}$ al conjunto $M(\mathbb{N})$ de multiconjuntos finitos sobre $\mathbb{N}$. El conjunto $M(\mathbb{N})$ está bien fundado bajo el orden $<_{mul}$ ya que $\mathbb{N}$ está bien fundado bajo $<$. Sean C, C' dos multiconjuntos de complejidades, entonces $C <_{mul} C' \Leftrightarrow \exists X \subseteq C, X' \subseteq C'$ tal que $C = (C' - X') \cup X$ y $\forall n \in X$, $\exists n' \in X'.\ n < n'$, donde $<$ es el orden estricto habitual sobre $\mathbb{N}$. Diremos $C \leqslant_{mul} C'$ si $C = C'$ o $C <_{mul} C'$.

Dado un término t, su *profundidad* se expresa con $depth(t)$:

$$\begin{array}{rcll} depth(x) & = & 0 & \text{si } x \in \mathcal{V} \\ depth(h(s_1, \ldots, s_m)) & = & 1 + max(depth(s_1), \ldots, depth(s_m)) & \text{si } h \in \mathcal{F}, \\ & & & m \geqslant 0 \end{array}$$

Usaremos esta noción en la demostración para establecer un *límite* en la profundidad de los términos computados.

Demostración. Para demostrar el Teorema 31, probaremos que hay un límite en la profundidad de los términos computados, porque esto implica que sólo se puede obtener un número finito de términos diferentes (módulo renombramiento de variables) por *narrowing* necesario.

Consideremos una derivación arbitraria (posiblemente infinita) de *narrowing* necesario para un término lineal t en $\mathcal{R}$. Asociamos un par (C_s, D_s) a cada término s en esta derivación, donde $C_s = comp(s)$ es la complejidad de s y $D_s = depth(s)$ es su profundidad. Trivialmente, para cada paso de *narrowing* necesario, $s \leadsto s'$, en esta derivación, sus pares asociados, (C_s, D_s) y $(C_{s'}, D_{s'})$, cumplen la condición $C_{s'} \leqslant_{mul} C_s$ (ya que $C_s = C_{s'}$ cuando una función cíclica es desplegada y $C_{s'} <_{mul} C_s$ cuando una función no cíclica es desplegada). Por lo tanto, para probar que la derivación considerada por *narrowing* necesario es cuasi-terminante, basta probar que cada subderivación (posiblemente infinita) en la cual el primer componente permanece sin cambio sólo puede contener un número finito de términos distintos módulo renombramiento de variables.

Consideremos una de las subderivaciones: $t_0 \leadsto t_1 \leadsto t_2 \leadsto \ldots$ Ahora, vamos a considerar un reordenamiento de la derivación en el cual los redexes se van a explotar en el orden más a la izquierda y más interno. Además, consideramos que en la derivación reordenada sólo se computan unificadores más generales, restringidos a las variables de $\mathcal{V}ar(s_{i-1})$ (en lugar de unificadores simples, i.e., no se anticipan algunos enlaces como en λ [AEH00, Def. 13]). Esta derivación más interna se denota por $s_0 \leadsto_{p_1,R_1,\sigma_1} s_1 \leadsto_{p_2,R_2,\sigma_2} s_2 \leadsto_{p_3,R_3,\sigma_3} \ldots$ Es claro que, la subderivación original es cuasi-terminante sii la subderivación reordenada es cuasi-terminante. Estas condiciones sobre la subderivación considerada simplifican el resto de la demostración.

Consideremos el primer paso de *narrowing*, $s_0 \leadsto_{p_1,R_1,\sigma_1} s_1$, en la subderivación. Las reglas de $\mathcal{R}$ son lineales por la derecha, lo que implica $s_1 = s_0[\theta_1(r_1)]_{p_1}$, donde $\delta_1 = \theta_1 \cup \sigma_1$ es el unificador más general de $s_0|_{p_1}$ y l_1, con $R_1 = (l_1 \to r_1)$ y $\sigma_1(s_0|_{p_1}) = \theta_1(l_1)$, i.e., los enlaces en σ_1 no necesitan aplicarse a $s_0[\theta_1(r_1)]_{p_1}$. Por la Definición 28, como la función desplegada es no creciente tenemos que $dv(s'_i, x) \geqslant dv(t'_j, x)$ para todos los subtérminos encabezados por un símbolo de función $g(\overline{t'_m})$ en r_1, donde $l_1 = f(\overline{s'_n})$, $i \in \{1, \ldots, n\}$, $x \in \mathcal{V}ar(s_i)$ y $j \in \{1, \ldots, m\}$. Como θ_1 sólo contiene enlaces para las variables en $\mathcal{V}ar(l_1)$, la profundidad de $\theta_1(r_1)$ está limitada por $max(depth(s_0|_{p_1}), depth(r_1|_{p_{11}}), \ldots, depth(r_1|_{p_{1n_1}})) + k$, donde $p_{11}, \ldots, p_{1n_1}$ son las posiciones de los subtérminos encabezados por símbolo de función en r_1 y k es un número finito para tomar en cuenta la profundidad de los constructores demandados por las funciones más externas en s_0 (e.g., $k = 0$ si $p_1 = \epsilon$). Si $\theta_1(r_1)$ aún contiene redexes de *narrowing* necesario, entonces tenemos otro paso de *narrowing*, $s_1 \leadsto_{p_2,R_2,\sigma_2} s_2$, donde $s_2 = s_1[\theta_2(r_2)]_{p_2}$ y $\delta_2 = \theta_2 \cup \sigma_2$ es el unificador más general de $s_1|_{p_2}$ y l_2, con $R_2 = (l_2 \to r_2)$ y $\sigma_2(s_1|_{p_2}) = \theta_2(l_2)$. Nuevamente, dado que la función desplegada es no creciente, la profundidad de $\theta_2(r_2)$ está acotada por $max(depth(s_1|_{p_2}), depth(r_2|_{p_{21}}), \ldots, depth(r_2|_{p_{2n_2}})) + k$, donde $p_{21}, \ldots, p_{2n_2}$ son las posiciones de los subtérminos encabezados por símbolo de función en r_2. Además, puesto que $s_1|_{p_2}$ es un subtérmino de $\theta_1(r_1)$, i.e., $s_2 = s_0[\theta_1(r_1)[\theta_2(r_2)]_{p_2}]_{p_1}$, tenemos que $\theta_2(r_2)$ está acotado por:

$$
\begin{array}{l}
max(depth(s_0|_{p_1}), \\
\quad depth(r_1|_{p_{11}}), \ldots, depth(r_1|_{p_{1n_1}}), \\
\quad depth(r_2|_{p_{21}}), \ldots, depth(r_2|_{p_{2n_2}})) + k
\end{array}
$$

Extendiendo este resultado a la subderivación (posiblemente infinita) donde $s_0|_{p_1}$ está completamente computada por *narrowing*, tenemos que $s_i|_{p_1}$ está acotado en la derivación por:

$$
\begin{array}{l}
max(depth(s_0|_{p_1}), \\
\quad depth(r_1|_{p_{11}}), \ldots, depth(r_1|_{p_{1n_1}}), \\
\quad \ldots, \\
\quad depth(r_m|_{p_{m1}}), \ldots, depth(r_m|_{p_{mn_m}})) \\
\quad + k
\end{array}
$$

donde $r_1, \ldots, r_m$ son las partes derechas de las reglas en las definiciones de todas las funciones no crecientes y $p_{j1}, \ldots, p_{jn_j}$, $1 \leqslant j \leqslant m$, son las posiciones de los subtérminos encabezados por símbolo de función en tales partes derechas. Dado que hay un límite para la profundidad de los términos encontrados durante la reducción del redex más interno $s_0|_{p_1}$, sólo se puede obtener un número finito de términos diferentes módulo renombramiento de variables.

Una vez que $s_0|_{p_1}$ está completamente computado por *narrowing*, distinguimos dos casos: si una función no cíclica es desplegada, la demostración concluye toda vez que esta reducirá estrictamente el primer componente del par asociado; si una función cíclica es desplegada, entonces se puede realizar un razonamiento similar sobre el nuevo redex. Por lo tanto, la profundidad de los términos computados en cada subderivación donde el primer componente permanece sin cambios está acotado por:

$$\begin{aligned} l \times max(&depth(s_0|_{p_1}),\\ &depth(r_1|_{p_{11}}), \ldots, depth(r_1|_{p_{1n_1}}),\\ &\ldots,\\ &depth(r_m|_{p_{m1}}), \ldots, depth(r_m|_{p_{mn_m}}))\\ &+ k' \end{aligned}$$

para una k' suficientemente grande (pero finita), donde l es el número de funciones no cíclicas en s_0, lo cual concluye la demostración. □

El análisis de la cuasi-terminación del mecanismo de *narrowing* implica sus propias particularidades, i.e., no hay una similitud clara entre cuasi-terminación con respecto a *narrowing* necesario y las condiciones relacionadas en la reescritura de términos. Por ejemplo, consideremos el siguiente SRT:

$$\begin{aligned} \texttt{f Z y} &= \texttt{y}\\ \texttt{f (S x) y} &= \texttt{f x (S y)} \end{aligned}$$

el cual viola las condiciones necesarias para los SRTs no crecientes, con $\texttt{Z}, \texttt{S} \in \mathcal{C}$ y $\texttt{f} \in \mathcal{D}$. Este SRT es trivialmente terminante con respecto a la reescritura (a pesar de que el segundo argumento es creciente), ya que el primer parámetro de la función `f` decrece estrictamente con cada llamada recursiva. Sin embargo no es cuasi-terminante con respecto a *narrowing* necesario, como se muestra en la siguiente computación (infinita):

$$\begin{aligned} \texttt{f x y} &\leadsto_{\{\texttt{x} \mapsto \texttt{S x}'\}} \texttt{f x}' \texttt{ (S y)}\\ &\leadsto_{\{\texttt{x}' \mapsto \texttt{S x}''\}} \texttt{f x}'' \texttt{ (S (S y))}\\ &\leadsto \quad \ldots \end{aligned}$$

Existen otras aproximaciones al análisis de terminación relacionadas, como la terminación por "cambio de tamaño" de los argumentos (*size-change analysis* [LJBA01]) adaptada a SRTs en [TG05]. Inicialmente, esta aproximación no es útil para garantizar la cuasi-terminación por *narrowing* necesario (si no se hace una adaptación del marco al contexto de los lenguajes lógico funcionales, como hacemos en el Capítulo 6), ya que sólo asegura que *algunos* parámetros decrecen (pero no todos ellos) lo cual no es suficiente en nuestro contexto, donde todos los parámetros podrían ser desconocidos (i.e., variables libres).

La linealidad por la derecha es un requisito esencial inclusive para las funciones más simples. Consideremos, por ejemplo, las siguientes funciones no crecientes:

$$\begin{array}{rcl} \mathtt{f\ Z\ y} & = & \mathtt{y} \\ \mathtt{f\ (S\ x)\ y} & = & \mathtt{f\ x\ y} \\ \mathtt{g\ x} & = & \mathtt{f\ x\ x} \end{array}$$

donde $\mathtt{Z}, \mathtt{S} \in \mathcal{C}$ y $\mathtt{f}, \mathtt{g} \in \mathcal{D}$. Se trata de un SRT que viola las condiciones de programas no crecientes. En concreto, la función `g` adolece de linealidad por la derecha, lo que provoca que la propagación de los enlaces (debidos a un paso de *narrowing*) de la variable repetida de lugar un argumento creciente:

$$\begin{array}{lllll} \mathtt{g\ x} & \leadsto_{\mathtt{id}} & \mathtt{f\ x\ x} & \leadsto_{\{\mathtt{x} \mapsto \mathtt{S\ x'}\}} & \mathtt{f\ x'\ (S\ x')} \\ & & & \leadsto_{\{\mathtt{x'} \mapsto \mathtt{S\ x''}\}} & \mathtt{f\ x''\ (S\ (S\ x''))} \\ & & & \leadsto & \ldots \end{array}$$

Así pues, cuando se viola la linealidad por la derecha el programa se convierte en no cuasi-terminante.

Veamos ahora cómo el uso de la estrategia de *narrowing* necesario es crucial, i.e., no se asegura la cuasi-terminación para otras estrategias de *narrowing* (e.g., *narrowing innermost*). Por ejemplo, en el siguiente SRT todas sus reglas son lineales por la derecha y las funciones cíclicas no contienen anidado ningún símbolo de función definida por el usuario:

$$\begin{array}{rcl} \mathtt{f\ x} & = & \mathtt{g\ (h\ x)} \\ \mathtt{g\ x} & = & \mathtt{x} \\ \mathtt{h\ Z} & = & \mathtt{Z} \\ \mathtt{h\ (S\ x)} & = & \mathtt{S\ (h\ x)} \end{array}$$

por tanto, con *narrowing* necesario se consiguen computaciones cuasi-terminantes mientras que la estrategia de *narrowing innermost* produce la siguiente derivación no cuasi-terminante:

$$\begin{array}{lllll} \mathtt{f\ x} & \leadsto_{\mathtt{id}} & \mathtt{g\ (h\ x)} & \leadsto_{\{\mathtt{x} \mapsto \mathtt{S\ x'}\}} & \mathtt{g\ (S\ (h\ x'))} \\ & & & \leadsto_{\{\mathtt{x'} \mapsto \mathtt{S\ x''}\}} & \mathtt{g\ (S\ (S\ (h\ x'')))} \\ & & & \leadsto & \ldots \end{array}$$

Por otro lado, las caracterizaciones más cercanas a nuestra aproximación han sido presentadas por Wadler [Wad90] y Chin y Khoo [CK96]. Wadler introdujo la noción de funciones *treeless* para asegurar la terminación del proceso de *deforestación* [Wad90]. Las funciones *treeless* son una subclase de nuestras funciones no crecientes donde todas las llamadas a función en las partes derechas de las reglas sólo tienen variables en sus argumentos.

Chin y Khoo [CK96] introdujeron la clase de los *consumidores no crecientes* y probaron que cualquier conjunto de funciones mutuamente recursivas que sean consumidores no crecientes se pueden transformar en un conjunto equivalente de funciones *treeless* de forma que la técnica de deforestación se puede aplicar. Esta caracterización difiere de la nuestra principalmente en dos puntos. Primero, Chin y Khoo solo exigen llamadas a función lineales en las partes derechas de las reglas (en lugar de exigir la linealidad en toda la parte derecha de las reglas, como nosotros lo exigimos). Esta definición (menos restrictiva) no es correcta en nuestro contexto. Consideremos, por ejemplo, los siguientes consumidores no crecientes de acuerdo a Chin y Khoo [CK96]:

$$\begin{array}{rcl} \texttt{f x} & = & \texttt{C (g x) x} \\ \texttt{g (S x)} & = & \texttt{g x} \\ \texttt{h (C (S x) y)} & = & \texttt{x} \end{array}$$

donde $\texttt{C}, \texttt{S} \in \mathcal{C}$ y $\texttt{f}, \texttt{g}, \texttt{h} \in \mathcal{D}$. Aquí, dado el término inicial, `h (f x)`, el *narrowing* necesario genera una derivación infinita que no es cuasi-terminante:

$$\begin{array}{lll} \texttt{h (f x)} & \leadsto_{id} & \texttt{h (C (g x) x)} \\ & \leadsto_{\{\texttt{x} \mapsto \texttt{S x}'\}} & \texttt{h (C (g x}') \texttt{ (S x}'\texttt{)))} \\ & \leadsto_{\{\texttt{x}' \mapsto \texttt{S x}''\}} & \texttt{h (C (g x}''\texttt{) (S (S x}''\texttt{)))} \\ & \leadsto & \ldots \end{array}$$

En segundo lugar, Chin y Khoo no aceptan llamadas a función anidadas en la parte derecha de ninguna regla del programa (nosotros imponemos la misma restricción pero sólo en las funciones cíclicas). Por contra, nosotros aceptamos términos arbitrarios (lineales) en las partes derechas de las funciones no cíclicas, lo que nos permite abarcar un rango más amplio de funciones.

La cuasi-terminación es una propiedad muy importante para el análisis y transformación de programas, e.g., para la especialización de programas, ya que generalmente la cuasi-terminación implica la terminación del proceso, como mostramos en la sección que sigue.

4.3. Del esquema NPE *online* al esquema NPE *offline*

La formulación original del esquema NPE asegura la terminación en el estilo *online* (ver, e.g., [AHV02, AV02, AFV98]), i.e., durante la evaluación parcial se utilizan diferentes tests de terminación y operadores de generalización para garantizar que sólo sea computado un número finito de términos diferentes (módulo renombramiento de

variables). Como se ha dicho, este esquema lleva a cabo optimizaciones muy potentes, aunque resulta muy costoso (en términos de consumo tanto de tiempo, como de recursos de cálculo) y, de esa manera, no se adapta de manera adecuada a la optimización de aplicaciones realistas. Con el fin de remediar esta situación, en esta sección introducimos un método NPE más rápido que asegura la terminación de manera *offline* mediante la inclusión de una etapa de pre-proceso basada en la noción de SRT no-creciente.

En principio, un método NPE simplista podría restringir los programas para especialización a SRTs no crecientes ya que no se requeriría la aplicación de tests de terminación ni generalizaciones durante el proceso de evaluación parcial, toda vez que las computaciones por *narrowing* necesario a partir de SRTs no crecientes ya son cuasi-terminantes (cf. Teorema 31). Este enfoque daría lugar a una herramienta NPE muy rápida en la que sólamente se aplicarían tests de igualdad módulo renombramiento de variables. Desgraciadamente la clase de programas susceptibles de transformación por ese método sería muy restrictiva.

Por todo lo anterior, consideraremos ahora una clase más amplia de SRTs, i.e., la clase de programas para los cuales NPE fue originalmente definido: los programas inductivamente secuenciales y definiremos un algoritmo que *anote* las expresiones que pueden causar la no-cuasi-terminación de las computaciones por *narrowing* necesario. Tales anotaciones serán más tarde utilizadas por una relación de *narrowing* necesario extendida para que sea capaz de *generalizar* los subtérminos problemáticos.

Definimos $\mathcal{F}_\bullet = \mathcal{F} \cup \{\bullet\}$, donde $\bullet \notin \mathcal{F}$ es un símbolo nuevo. Dado el SRT $\mathcal{R}$, la anotación del término t se consigue reemplazando t por $\bullet(t)$. Las funciones auxiliares que se muestran a continuación serán usadas para manipular términos anotados:

$$\begin{array}{rcll} gen(x) & = & x & \text{si } x \in \mathcal{V} \\ gen(h(\overline{t_n})) & = & h(\overline{gen(t_n)}) & \text{si } h \in \mathcal{F},\ n \geqslant 0 \\ gen(\bullet(t)) & = & y & \text{donde } y \in \mathcal{V} \text{ es una variable nueva} \end{array}$$

i.e., dado un término anotado $t \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$, la expresión $gen(t) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ devuelve una generalización de t mediante el reemplazo de los subtérminos anotados por variables nuevas.

$$\begin{array}{rcll} aterms(x) & = & \varnothing & \text{si } x \in \mathcal{V} \\ aterms(h(\overline{t_n})) & = & \bigcup_{i=1}^{n} aterms(t_i) & \text{si } h \in \mathcal{F},\ n \geqslant 0 \\ aterms(\bullet(t)) & = & \{t\} \cup aterms(t) & \end{array}$$

Aquí, la expresión $aterms(t) \subseteq \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$ devuelve el conjunto de subtérminos anotados (los cuales pueden incluir anotaciones) en $t \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$.

En los ejemplos que siguen ilustramos el uso de las funciones *gen* y *aterms*:

$$
\begin{array}{lcl}
gen(\texttt{f x (g (h y))}) & = & \texttt{f x (g (h y))} \\
gen(\texttt{f x} \bullet \texttt{(g (h y))}) & = & \texttt{f x w} \\
gen(\texttt{f x} \bullet \texttt{(g} \bullet \texttt{(h y))}) & = & \texttt{f x w} \\
aterms(\texttt{f x (g (h y))}) & = & \{\,\} \\
aterms(\texttt{f x} \bullet \texttt{(g (h y))}) & = & \{\texttt{g (h y)}\} \\
aterms(\texttt{f x} \bullet \texttt{(g} \bullet \texttt{(h y))}) & = & \{\texttt{g} \bullet \texttt{(h y), h y}\}
\end{array}
$$

Ahora, nos centramos en definir una transformación para la anotación de programas inductivamente secuenciales. Intuitivamente, revisaremos las partes derechas de las reglas y anotaremos aquellos argumentos de llamadas a función que contienen símbolos de función definida (evitando funciones anidadas) o bien, sus argumentos crecientes. Las anotaciones van de fuera hacia adentro, i.e., cada subtérmino anotado será tratado de manera similar a la parte derecha original de la regla (de esta manera se pueden presentar anotaciones anidadas); finalmente, todas las ocurrencias repetidas, excepto una, de la misma variable son anotadas para garantizar la linealidad. Más formalmente,

Definición 32 ($ann(\mathcal{R})$) *Sea $\mathcal{R} = \{l_i \to r_i \mid i = 1, \ldots, k\}$ un SRT inductivamente secuencial sobre $\mathcal{F}$. El SRT anotado, $ann(\mathcal{R})$, sobre $\mathcal{F}_\bullet$ se da por el conjunto de reglas $\{l_i \to r'_i \mid i = 1, \ldots, k\}$ donde r'_i, $i = 1, \ldots, k$, se computa como sigue:*

1. *Si $root(l_i)$ es una función no cíclica, entonces r'_i se obtiene a partir de r_i anotando todas las ocurrencias de la misma variable, excepto una (e.g., la de la posición más a la izquierda), para que $gen(r'_i)$ sea un término lineal.*

2. *Si $root(l_i)$ es cíclica, entonces r'_i se obtiene a partir de $qs(l_i, r_i)$ anotando el menor número de variables tales que $gen(t)$ sea lineal para todo $t \in \{qs(l_i, r_i)\} \cup aterms(qs(l_i, r_i))$.*

En la Definición 32 se hace uso de la función qs la cual mostramos en la Figura 4.1. Básicamente, la función auxiliar qs ignora los símbolos constructores hasta que encuentra un subtérmino encabezado por un símbolo de función definida $f(t_1, \ldots, t_n)$. Entonces, para cada argumento t_i, se procede (llamando a qs') como sigue:

- si t_i es un término constructor y todas las variables cumplen la propiedad de función no creciente, entonces t_i permanece sin cambios;

- en caso contrario, el subtérmino considerado, t_i, se anota y el proceso se reinicia para t_i.

$$qs(l,t) = \begin{cases} t & \text{si } t \in \mathcal{V} \text{ es una variable} \\ c(\overline{qs(l,t_n)}) & \text{si } t = c(\overline{t_n}),\ c \in \mathcal{C},\ \text{y } n \geqslant 0 \\ f(\overline{t'_n}) & \text{si } t = f(\overline{t_n}),\ f \in \mathcal{D},\ \text{y } t'_i = qs'(l,t_i) \\ & \quad \text{para todo } i = 1,\ldots,n,\ n \geqslant 0 \end{cases}$$

$$qs'(f(\overline{p_n}),t) = \begin{cases} t & \text{si } t \in \mathcal{T}(\mathcal{C},\mathcal{V}) \text{ es un término constructor} \\ & \quad \text{y } dv(p_i,x) \geqslant dv(t,x) \\ & \quad \text{para toda } x \in \mathcal{V}ar(p_i),\ i = 1,\ldots,n \\ \bullet(qs(f(\overline{p_n}),t)) & \text{en caso contrario} \end{cases}$$

Figura 4.1: Funciones auxiliares qs y qs'.

De manera trivial, para cualquier SRT no creciente $\mathcal{R}$, tenemos que $ann(\mathcal{R}) = \mathcal{R}$. Además, si $\mathcal{R}$ es un SRT inductivamente secuencial entonces $ann(\mathcal{R})$ también lo es, ya que las partes izquierdas de las reglas no se modifican.

Notemos que la Definición 32 es indeterminista ya que no establece que ocurrencia de una variable debería dejarse sin anotar cuando hay ocurrencias repetidas de la misma variable. En algunos casos, esta decisión puede afectar el resultado de la evaluación parcial. El problema se podría resolver, en algunos casos, permitiendo al programador seleccionar la variable estática que debería permanecer sin anotación.

Ejemplo 4.4 *Consideremos el programa inductivamente secuencial $\mathcal{R}$:*

$$\begin{array}{rcl} \mathtt{f\ Z\ y} & = & \mathtt{y} \\ \mathtt{f\ (S\ x)\ y} & = & \mathtt{g\ x\ (f\ x\ (S\ y))} \\ \mathtt{g\ x\ y} & = & \mathtt{g\ y\ x} \end{array}$$

donde $\mathtt{f},\mathtt{g} \in \mathcal{D}$ y $\mathtt{Z},\mathtt{S} \in \mathcal{C}$. El SRT anotado, $ann(\mathcal{R})$, es como sigue:

$$\begin{array}{rcl} \mathtt{f\ Z\ y} & = & \mathtt{y} \\ \mathtt{f\ (S\ x)\ y} & = & \mathtt{g\ x} \bullet \mathtt{(f\ x} \bullet \mathtt{(S\ y))} \\ \mathtt{g\ x\ y} & = & \mathtt{g\ y\ x} \end{array}$$

Observemos que las ocurrencias repetidas de x en la segunda regla no se deben anotar toda vez que:

$$aterms(\mathtt{g\ x} \bullet \mathtt{(f\ x} \bullet \mathtt{(S\ y))}) = \{\mathtt{f\ x} \bullet \mathtt{(S\ y)},\ \mathtt{(S\ y)}\}$$

y, en este caso, $gen(t)$ es lineal para todo

$$t \in \{\mathtt{g\ x} \bullet \mathtt{(f\ x} \bullet \mathtt{(S\ y))},\ \mathtt{f\ x} \bullet \mathtt{(S\ y)},\ \mathtt{(S\ y)}\}$$

i.e., $\mathtt{g(x,w_1)}$, $\mathtt{f(x,w_2)}$, *y* $\mathtt{(S\ y)}$ *son términos lineales, donde* $\mathtt{w_1}$ *y* $\mathtt{w_2}$ *son variables nuevas.*

En el esquema original NPE las computaciones parciales se realizan por medio de *narrowing* necesario; a continuación, vamos a extender esta relación con la finalidad de que el mecanismo pueda procesar (generalizar) los subtérminos anotados para asegurar así la terminación del proceso de evaluación parcial.

Definición 33 (*narrowing* necesario generalizante) *Sea $\mathcal{R}$ un SRT inductivamente secuencial anotado sobre $\mathcal{F}_\bullet$. La relación de narrowing necesario generalizante, en símbolos $\rightsquigarrow$, se define como la menor relación que satisface:*

(*narrowing* necesario)

$$\frac{s \rightsquigarrow_{p,R,\sigma} t}{s \rightsquigarrow_\sigma t} \qquad \textit{si } root(s) \in \mathcal{D} \textit{ y } s \in \mathcal{T}(\mathcal{F},\mathcal{V})$$

(generalización)

$$\frac{t \in \{s\} \cup aterms(s)}{s \rightsquigarrow_\bullet gen(t)} \qquad \textit{si } root(s) \in \mathcal{D} \textit{ y } s \notin \mathcal{T}(\mathcal{F},\mathcal{V})$$

(descomposición)

$$\frac{s = c(t_1,\ldots,t_n) \ \wedge\ i \in \{1,\ldots,n\}}{s \rightsquigarrow_\mathcal{C} t_i} \qquad \textit{si } root(s) \in \mathcal{C}$$

Una derivación de *narrowing* necesario generalizante $s \rightsquigarrow^*_\sigma t$ se compone de pasos de *narrowing* necesario (para términos encabezados por un símbolo de función sin anotaciones), de generalizaciones (para términos anotados), y de descomposición de constructores (para términos encabezados por un constructor sin anotaciones), donde σ es la composición de las sustituciones que etiquetan los pasos de *narrowing* necesario propiamente dicho. Notemos que toda vez que *narrowing* necesario computa solo a *forma normal en cabeza* (i.e., una variable o un término encabezado por constructor), la regla de descomposición se requiere para asegurar que todas las posibles funciones internas sean eventualmente evaluadas parcialmente.

Una observación interesante es que el paso de generalización es en cierta forma equivalente a la operación de *splitting* de la *deducción parcial conjuntiva* (*conjunctive partial deduction*, CPD) de programas lógicos [DGJ+99]. Mientras que la CPD considera conjunciones de átomos, nosotros tratamos con términos que pueden contener símbolos de función anidados. Por lo tanto, el aplanamiento mediante la generalización de una llamada a función anidada es básicamente equivalente a aplicar *splitting* a una conjunción (en ambos casos se pierde cierta información).

El siguiente resultado muestra la corrección del algoritmo de anotación.

Teorema 34 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y t un término lineal. Cada derivación de narrowing necesario generalizante para t en $ann(\mathcal{R})$ es cuasi-terminante.*

Esquema. A continuación presentamos el esquema de la demostración. Esencialmente, $ann(\mathcal{R})$ devuelve un SRT nuevo, el cual no es no creciente pero que se puede descomponer en un número de SRTs que se comporten como SRTs no crecientes con respecto a *narrowing* necesario. Para ser precisos, consideremos un SRT inductivamente secuencial $\mathcal{R}$. Entonces, cada SRT $\mathcal{R}_i$ que se obtiene a partir de $ann(\mathcal{R})$ mediante el reemplazo de cada regla $(l \to r) \in ann(\mathcal{R})$ por una regla (sin anotación) $l \to gen(t)$, donde $t \in \{r\} \cup aterms(r)$, es un SRT no creciente excepto por el hecho de que puede contener algunas variables extra (i.e., variables que aparecen en el lado derecho de alguna regla pero que no aparecen en la parte izquierda correspondiente). Por ejemplo, el SRT anotado del Ejemplo 4.4 se descompone en los siguientes conjuntos de reglas:

$$\begin{array}{lcll}
\mathcal{R}_1 & = & \{ & \mathtt{f\ Z\ y} = \mathtt{y}, \quad \mathtt{f\ (S\ x)\ y} = \mathtt{g\ x\ w_1}, \\
 & & & \mathtt{g\ x\ y} = \mathtt{g\ y\ x} \quad \} \\
\mathcal{R}_2 & = & \{ & \mathtt{f\ Z\ y} = \mathtt{y}, \quad \mathtt{f\ (S\ x)\ y} = \mathtt{f\ x\ w_2}, \\
 & & & \mathtt{g\ x\ y} = \mathtt{g\ y\ x} \quad \} \\
\mathcal{R}_3 & = & \{ & \mathtt{f\ Z\ y} = \mathtt{y}, \quad \mathtt{f\ (S\ x)\ y} = \mathtt{S\ y}, \\
 & & & \mathtt{g\ x\ y} = \mathtt{g\ y\ x} \quad \}
\end{array}$$

Dado que las variables extra no afectan la demostración del Teorema 31, la afirmación se cumple a partir de la equivalencia entre las derivaciones de *narrowing* necesario generalizante en $ann(\mathcal{R})$ y las derivaciones de *narrowing* necesario en cada $\mathcal{R}_i$. □

4.4. El método de evaluación parcial *offline* dirigido por *narrowing*

En la presente sección, primero describimos el método *offline* NPE, el cual se basa en el uso de programas anotados que son procesados con *narrowing* necesario generalizante. A continuación ilustramos el nuevo esquema por medio de algunos ejemplos seleccionados y, finalmente, presentamos un resumen de experimentos llevados a cabo con un prototipo implementado (con la aproximación de NPE *offline*) donde se muestran las ventajas con respecto al método NPE anterior (*online*).

4.4.1. Descripción del método NPE *offline*

En nuestra aproximación NPE *offline*, a partir de un SRT inductivamente secuencial $\mathcal{R}$, aplicamos una *primera etapa* que consiste en computar la anotación del SRT:

$ann(\mathcal{R})$. Enseguida, la *segunda etapa*—la de la propia evaluación parcial—toma el SRT anotado $ann(\mathcal{R})$ junto con un término inicial (lineal); t, construye el árbol (finito) de *narrowing* necesario generalizante para t en $ann(\mathcal{R})$ y extrae del árbol el programa evaluado parcialmente (residual).

Esencialmente, los programas residuales se extraen mediante la formación de los llamados *resultantes*, $\sigma(s) \to rann(t)$, para cada paso de *narrowing* necesario $s \leadsto_\sigma t$ en el árbol de *narrowing* necesario generalizante considerado. La función $rann$ elimina las ocurrencias de "$\bullet$" en un término t.

En general, las partes izquierdas de las reglas no tienen la forma $f(s_1, \ldots, s_n)$, con $\overline{s_n}$ términos constructores, es decir, no son legales ya que algún s_i puede contener símbolos de función definida. Por tanto, es necesario restaurar la disciplina de constructores del programa original. Para ello se aplica el renombramiento independiente ρ de términos introducido en la Sección 3.4.

Básicamente, el paso de especialización del esquema NPE *offline* procede como sigue: dado el programa anotado $ann(\mathcal{R})$ y un término lineal t, se lleva a cabo la construcción del árbol de *narrowing* necesario generalizante para t en $ann(\mathcal{R})$, donde además, se realiza un comprobación para verificar si se ha computado alguna variante del término actual, en cuyo caso, se detiene la computación. La cuasi-terminación de las computaciones por *narrowing* necesario generalizante (Teorema 34) garantiza que el árbol construido es finito. Una vez que ya se ha construido el árbol, computamos el renombramiento independiente ρ para el conjunto de términos $\{s \mid s \leadsto_\sigma t\}$. Mientras la función ρ es suficiente para renombrar las partes izquierdas de los resultantes, las partes derechas requieren una función mucho más elaborada, ren_ρ (definida en la Sección 3.4), la cual reemplaza recursivamente cada llamada a función en un término por la llamada a la función renombrada por ρ.

A continuación introducimos formalmente el método NPE *offline*:

Definición 35 (NPE *offline*) *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $f(\overline{x_n})$ un término lineal con $f \in \mathcal{D}$. Esto no es una restricción ya que uno puede, de manera alternativa, considerar un término arbitrario t agregando simplemente una nueva definición de función $f(\overline{x_n}) \to t$ a $\mathcal{R}$, donde $\overline{x_n}$ son las diferentes variables de t.*

La NPE offline de $\mathcal{R}$ con respecto a $f(\overline{x_n})$ se obtiene como sigue:

1. *Primero, computamos el SRT anotado $ann(\mathcal{R})$.*

2. *Entonces, construimos un árbol (finito) de narrowing necesario generalizante, τ, para $f(\overline{x_n})$ en $ann(\mathcal{R})$, donde cada derivación termina cuando llega a un término constructor o a un término encabezado por un símbolo de función que es un renombramiento de las variables de algún término previo en la misma derivación (o en alguna previa).*

3. *Finalmente, el SRT residual contiene una regla (renombrada)*

$$\sigma(\rho(s)) \to ren_\rho(rann(s'))$$

por cada paso de narrowing necesario propiamente dicho $s \leadsto_\sigma s'$ *en* τ. *Aquí,* ρ *es un renombramiento independente de* $\{s \mid s \leadsto_\sigma s' \in \tau\}$.

Por simplicidad, en la Definición 35, extraemos un resultante por cada paso individual de *narrowing* necesario. Es claro que son posibles algoritmos más refinados para la extracción de resultantes a partir del árbol de *narrowing* necesario generalizante; e.g., en algunos casos, uno puede extraer un sólo resultante asociado a una *secuencia* de pasos de *narrowing* en lugar de solamente a un paso de *narrowing*.

Como resultado de la evaluación parcial pueden aparecer algunas funciones intermedias (innecesarias) en el programa residual. No obstante, este tipo de reglas *redundantes* pueden evitarse mediante un *post-proceso de desplegado* [JGS93]. Una regla que define a una función f se considera redundante [AHV02] si:

1. f no pertenece al conjunto de llamadas a función originales a evaluarse parcialmente.
2. f se llama sólo una vez en el programa residual.
3. f no es recursiva.

cuando la función f es redundante, el proceso de post-proceso de desplegado actúa del siguiente modo: la llamada a la función f se despliega y su definición se retira del programa final residual.

A continuación establecemos la corrección y terminación de este método de evaluación parcial.

Teorema 36 *Sea* $\mathcal{R}$ *un SRT inductivamente secuencial y* $f(\overline{x_n})$ *un término lineal con* $f \in \mathcal{D}$. *El algoritmo de la Definición 35 siempre termina y lleva a cabo la computación de un SRT inductivamente secuencial* $\mathcal{R}'$ *tal que narrowing necesario computa los mismos resultados para* $f(\overline{x_n})$ *en* $\mathcal{R}$ *y en* $\mathcal{R}'$.

Demostración. A continuación bosquejamos la demostración de este resultado. La demostración completa se puede seguir a partir de los resultados y nociones técnicas presentadas en los trabajos de [AFV98] y [AHLV05]. Básicamente, establecemos una correspondencia entre nuestro método de NPE *offline* y el esquema original (*online*).

Como ya se ha visto en la Sección 3.6.2, el algoritmo original [AFV98] es paramétrico con respecto a la *relación de narrowing* que se usa en la construcción de árboles

parciales, la *regla de desplegado* que determina cuándo y cómo terminar la construcción de tales árboles, y el *operador de generalización* que se usa para garantizar que el número de árboles se mantenga finito. Haciendo uso de la parametricidad y dado un árbol de *narrowing* necesario generalizante, τ, para $f(\overline{x_n})$ en $ann(\mathcal{R})$ mostramos que el conjunto de términos $\{s \mid s \rightsquigarrow_\sigma t \in \tau\}$ computados por la Def. 35, se puede computar además por una instancia apropiada del citado algoritmo.

Para este propósito, consideramos una regla para desplegar que sólo ejecuta un paso de *narrowing* necesario, i.e., $desplegar(T, \mathcal{R}) = \{\sigma(t) \to s \mid t \rightsquigarrow_\sigma s\}$, el cual es cláramente terminante. Por otro lado, el operador de control global está definido por:

$$generalizar(T, \{r_1, \ldots, r_n\}) = T \cup \{s \mid r_i \approx_\tau r'_i \text{ y } r'_i \downarrow_{\bullet/\mathcal{C}} s,\ i = 1, \ldots, n\}$$

donde $r_i \approx_\tau r'_i$ denota que $r'_i \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$ es el término asociado (posiblemente anotado) a r_i en el árbol de *narrowing* necesario generalizante τ, y $r'_i \downarrow_{\bullet/\mathcal{C}} s$ denota que existe una secuencia (posiblemente vacía) de pasos de *narrowing* necesario generalizante a partir de r'_i a s la cual aplica sólo pasos de generalización ($\rightsquigarrow_\bullet$) y descomposición ($\rightsquigarrow_\mathcal{C}$), y que ya no se puede reducir más con este tipo de pasos de *narrowing* generalizante. Básicamente, nuestro operador de control global, ignora la información *online* (i.e., el conjunto de términos en T que ya han sido evaluados parcialmente), esto se debe a que emplea las anotaciones en τ para efectuar su función. Esta definición satisface trivialmente las condiciones de operadores de control global en [AFV98] (i.e., todos los términos en $T \cup \{r_1, \ldots, r_n\}$ están cerrados con respecto al conjunto $generalizar(T, \{r_1, \ldots, r_n\})$.

En función de que tanto la regla de desplegado y el operador de control global son correctos de acuerdo a [AFV98], tenemos que el conjunto de resultantes $\{\sigma(s) \to t \mid s \rightsquigarrow_\sigma t \in \tau\}$ está cerrado con respecto al conjunto de términos $\{s \mid s \rightsquigarrow_\sigma t \in \tau\}$. Por lo tanto, por el Teorema 4.16 en [AHLV05], *narrowing* necesario computa los mismos resultados en $\mathcal{R}$ y en el SRT obtenido mediante el renombramiento de las reglas en $\{\sigma(s) \to t \mid \rightsquigarrow_\sigma t \in \tau\}$. Adicionalmente, el programa residual es inductivamente secuencial [AHLV05, Teorema 4.9]. Finalmente, la terminación del proceso es una consecuencia inmediata del Teorema 34. □

4.4.2. Aspectos de control

En este capítulo hemos presentado una primera aproximación al método de evaluación parcial *offline* dirigido por *narrowing* el cual se basa en la caracterización de SRTs no crecientes, en el Capítulo 6 introducimos una aproximación alternativa que utiliza un formalismo llamado grafos *size-change* [LJBA01]. Cada método presenta requerimientos ligeramente diferentes en cuanto a los aspectos de control, a continuación nos centramos en la presente aproximación.

Para la descripción de los dos niveles de control hacemos uso de la introducción formal desarrollada en la demostración del Teorema 36. En cuanto al control local consideramos una regla de desplegado que ejecuta sólo un paso de *narrowing* necesario, i.e., $t \leadsto_\sigma s$ y de esa forma asegura la terminación.

Por otro lado, en el nivel global, a partir de términos en el árbol de *narrowing* necesario generalizante τ, se lleva a cabo una secuencia de pasos de *narrowing*, que sólo aplica pasos de descomposición ($\leadsto_{\mathcal{C}}$) o de generalización ($\leadsto_\bullet$) hasta que ya no se puede reducir más con ese tipo de pasos. La secuencia puede ser vacía ya que el término sujeto al análisis global podría no estar anotado o no estar encabezado por un símbolo constructor. De manera distinta al control global de [AFV98], aquí se ignora la información *online* (i.e., el conjunto de términos T que ya han sido evaluados parcialmente), i.e., para descubrir los términos que se han de especializar sólo se tiene en cuenta las anotaciones del término analizado de τ.

De este modo, el control local avanza un paso de *narrowing* y el control global generaliza, descompone, y determina qué expresiones son susceptibles de computar, asegurando que el crecimiento del árbol sea finito mientras se garantiza la corrección (cierre) del conjunto de resultantes computado (véase Teorema 36).

El control local es trivial [PHG99], i.e., efectúa sólo un paso de desplegado, sin embargo, es adecuado ya que asegura que el conjunto de términos que han sido especializados crece en cada ciclo (del algoritmo de especialización de la Figura 3.2) lo que permite el acercamiento al cierre de la especialización [AFV98] y cumple aún con las condiciones típicas (evitar la explosión de espacio de búsqueda en los árboles de *narrowing* y la duplicación de cómputo). Por otro lado es un mecanismo fácil de computar, lo que contribuye a uno de los objetivos iniciales del estudio, i.e., permitir la adaptación del método de evaluación parcial a programas realistas. Ciertamente se pierde precisión pero se gana adaptabilidad del método.

4.4.3. Ejemplos

En esta sección, ilustramos el método NPE *offline* introducido en el presente capítulo por medio de algunos ejemplos seleccionados.

Especialización de programas. Nuestro primer ejemplo ilustra el uso del método NPE *offline* para la especialización de programas. Consideremos el siguiente SRT que

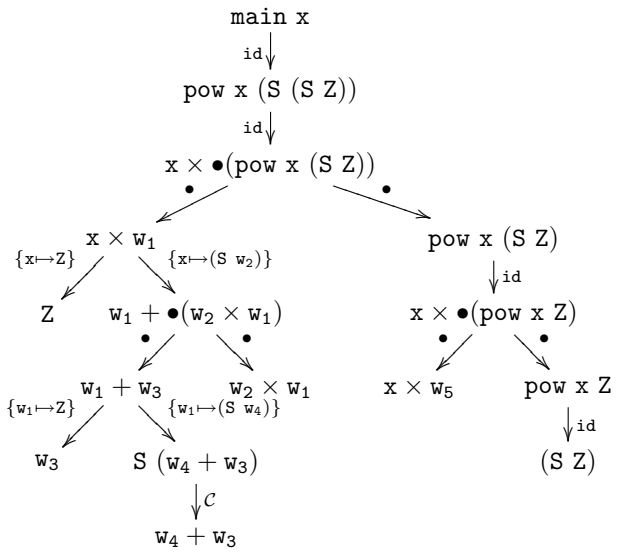


Figura 4.2: Árbol de *narrowing* necesario generalizante para `main x`.

ha sido anotado de acuerdo a la Definición 32:

$$\begin{array}{rcl} \texttt{main x} & = & \texttt{pow x (S (S Z))} \\ \texttt{pow x Z} & = & \texttt{(S Z)} \\ \texttt{pow x (S n)} & = & \texttt{x} \times \bullet\texttt{(pow x n)} \\ \texttt{Z} \times \texttt{m} & = & \texttt{Z} \\ \texttt{(S n)} \times \texttt{m} & = & \texttt{m} + \bullet\texttt{(n} \times \texttt{m)} \\ \texttt{Z} + \texttt{m} & = & \texttt{m} \\ \texttt{(S n)} + \texttt{m} & = & \texttt{S (n} + \texttt{m)} \end{array}$$

Los números naturales se forman a partir de `S` y `Z`. Dado el término inicial `main x`, construimos el árbol de *narrowing* necesario generalizante presentado en la Figura 4.2. Entonces, extraemos el SRT residual asociado, el cual contiene las siguientes reglas:

$$\begin{array}{rcl} \texttt{main x} & = & \texttt{pow}_2\ \texttt{x} \\ \texttt{pow}_2\ \texttt{x} & = & \texttt{x} \times \texttt{pow}_1\ \texttt{x} \\ \texttt{pow}_1\ \texttt{x} & = & \texttt{x} \times \texttt{pow}_0\ \texttt{x} \\ \texttt{pow}_0\ \texttt{x} & = & \texttt{S Z} \end{array}$$

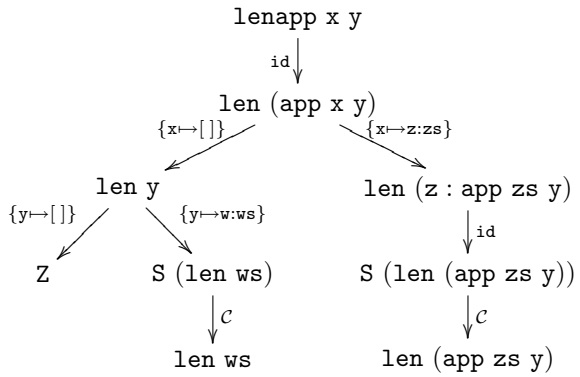


Figura 4.3: Árbol de *narrowing* necesario generalizante para `lenapp x y`.

junto con las definiciones originales de "×" y "+". El renombramiento independente considerado es como sigue:

$$\begin{aligned}\rho = \{ \quad \texttt{main x} &\mapsto \texttt{main x},\\ \texttt{pow x (S (S Z))} &\mapsto \texttt{pow}_2\ \texttt{x},\\ \texttt{pow x (S Z)} &\mapsto \texttt{pow}_1\ \texttt{x},\\ \texttt{pow x Z} &\mapsto \texttt{pow}_0\ \texttt{x},\\ \texttt{x} \times \texttt{y} &\mapsto \texttt{x} \times \texttt{y},\\ \texttt{x} + \texttt{y} &\mapsto \texttt{x} + \texttt{y} \quad \}\end{aligned}$$

Adicionalmente, las cuatro reglas se pueden simplificar fácilmente por medio del uso del post-proceso de desplegado del siguiente modo:

$$\texttt{main x} = \texttt{x} \times (\texttt{x} \times (\texttt{S Z}))$$

ya que las funciones $\texttt{pow}_2$, $\texttt{pow}_1$, y $\texttt{pow}_0$ son solo funciones intermedias (i.e., sólo hay una llamada a cualquiera de ellas y no son recursivas). Este ejemplo muestra que, a pesar de la anotación de algunos subtérminos, el poder de especialización del esquema original NPE (*online*) no se pierde con nuestra aproximación *offline*.

Deforestación. Nuestro segundo ejemplo está relacionado con la deforestación de Wadler, una técnica de transformación de programas para eliminar estructuras de

datos intermedias [Wad90]. El ejemplo considera el siguiente SRT $\mathcal{R}$:

$$\begin{array}{rcl} \texttt{lenapp x y} & = & \texttt{len (app x y)} \\ \texttt{len [ ]} & = & \texttt{Z} \\ \texttt{len (x : xs)} & = & \texttt{S (len xs)} \\ \texttt{app [ ] y} & = & \texttt{y} \\ \texttt{app (x : xs) y} & = & \texttt{x : app xs y} \end{array}$$

donde `lenapp x y` computa la longitud de la concatenación de las listas `x` e `y`. Esta función no es eficiente ya que la concatenación de `x` e `y` genera una estructura de datos intermedia. El SRT $\mathcal{R}$ de entrada ya es no creciente, por lo tanto, tenemos que $ann(\mathcal{R}) = \mathcal{R}$. Dado el término inicial `lenapp x y`, construimos el árbol de *narrowing* necesario generalizante que mostramos en la Figura 4.3. Extraemos las reglas residuales y utilizando el siguiente renombramiento independiente:

$$\begin{array}{rcll} \rho = \{ \quad \texttt{lenapp x y} & \mapsto & \texttt{lenapp x y}, & \\ \texttt{len (app x y)} & \mapsto & \texttt{la x y}, & \\ \texttt{len y} & \mapsto & \texttt{len y}, & \\ \texttt{len (z : app zs y)} & \mapsto & \texttt{la2 z zs y} & \} \end{array}$$

el SRT residual asociado es como sigue:

$$\begin{array}{rcl} \texttt{lenapp x y} & = & \texttt{la x y} \\ \texttt{la [ ] y} & = & \texttt{len y} \\ \texttt{la (z : zs) y} & = & \texttt{la2 z zs y} \\ \texttt{la2 z zs y} & = & \texttt{S (la zs y)} \end{array}$$

junto con la definición original de la función *len*. Como en los ejemplos previos, se aplica un post-proceso de desplegado, que elimina la función intermedia `la2`. Podemos observar que el SRT residual está completamente "deforestado" (i.e., ya no se construye un lista intermedia).

Eliminación del orden superior. Nuestro último ejemplo tiene que ver con la eliminación de funciones de orden superior. En algunos lenguajes de programación, las características de orden superior son *desfuncionalizadas* [Rey98, War82], i.e., son expresadas por medio de un programa de primer orden con un operador de aplicación explícito. Por ejemplo, el SRT que sigue ha sido ya anotado de acuerdo a la Definición 32, e incluye la formulación de la típica función de orden superior `map`:

$$\begin{array}{rcl} \texttt{minc x} & = & \texttt{map inc}_0\ \texttt{x} \\ \texttt{map f [ ]} & = & \texttt{[ ]} \\ \texttt{map f (x : xs)} & = & \texttt{(apply } \bullet \texttt{ (f) x) : (map f xs)} \\ \texttt{inc x} & = & \texttt{S x} \\ \texttt{apply inc}_0\ \texttt{x} & = & \texttt{inc x} \end{array}$$

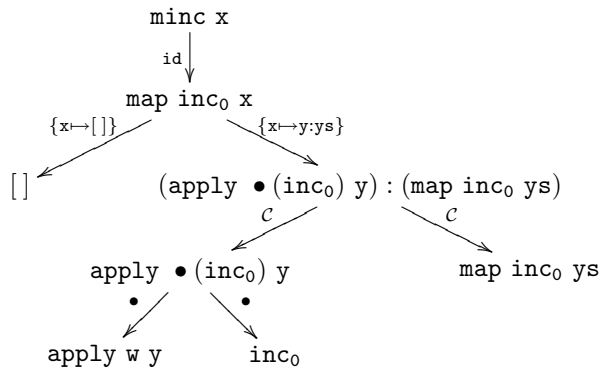


Figura 4.4: Árbol de *narrowing* necesario generalizante para `minc x`.

En el ejemplo, utilizamos el operador de aplicación explícita `apply` junto con la función $\texttt{inc}_0$ (un símbolo constructor) para la aplicación parcial.

En este ejemplo hemos anotado la ocurrencia de la variable `f` de más a la izquierda en la tercera regla del programa. Esto es esencial para obtener una definición de primer orden para $\texttt{map inc}_0\ \texttt{x}$. De hecho, si se anota la segunda ocurrencia de la variable `f`, el evaluador parcial devuelve básicamente, el programa original.

Dado el término inicial `minc x`, se construye el árbol de *narrowing* necesario generalizante de la Figura 4.4. Notemos que `apply w y` no se reduce más porque, como se mencionó antes, esta llamada de orden superior contiene una variable funcional libre, y así, su evaluación suspende (lo que significa que la definición original de `apply` debería incluirse en el programa residual).

El renombrado independiente produce:

$$\begin{array}{rcl} \rho = \{ \quad \texttt{minc x} & \mapsto & \texttt{minc x}, \\ \texttt{map inc}_0\ \texttt{ys} & \mapsto & \texttt{mapinc ys}, \\ \texttt{inc y} & \mapsto & \texttt{inc y}, \\ \texttt{apply w y} & \mapsto & \texttt{apply w y} \quad \} \end{array}$$

y el SRT residual computado por el método *offline* NPE es el que sigue:

$$\begin{array}{rcl} \texttt{minc x} & = & \texttt{mapinc x} \\ \texttt{mapinc [ ]} & = & \texttt{[ ]} \\ \texttt{mapinc (y : ys)} & = & \texttt{(apply inc}_0\ \texttt{y) : mapinc ys} \\ \texttt{inc y} & = & \texttt{(S y)} \\ \texttt{apply inc}_0\ \texttt{y} & = & \texttt{inc y} \end{array}$$

Finalmente, usando el post-proceso de desplegado obtenemos el siguiente SRT:

$$\begin{array}{rcl} \texttt{minc x} & = & \texttt{mapinc x} \\ \texttt{mapinc [ ]} & = & [\,] \\ \texttt{mapinc (y : ys)} & = & (\texttt{S y}) : \texttt{mapinc ys} \end{array}$$

donde el operador de aplicación explícita `apply` no se requiere más. Hay que hacer notar que esta transformación produce con frecuencia mejoras muy significativas en la velocidad de ejecución de los programas (vease, e.g., [AHV02]).

4.4.4. Evaluación experimental

El método NPE *offline*, introducido en la Sección 4.4.1 ha sido implementado en el lenguaje declarativo multi-paradigma Curry [Han03]. Los ficheros de código fuente del evaluador parcial son públicos y están en:

`http://www.dsic.upv.es/users/elp/german/offpeval/`

La herramienta de evaluación parcial es puramente declarativa y acepta programas Curry. Los programas de prueba son:

`ackermann:` Esta es la conocida función de Ackermann, en el ejercicio se especializa la función para argumentos de entrada mayores o iguales a 10.

`allones:` El objetivo de este programa es producir automáticamente una nueva función que transforme todos los elementos de una lista en "1"; para ello, computa primero la longitud de la lista original y, entonces, construye una nueva lista de la misma longitud cuyos elementos son 1. Este es un ejemplo típico de deforestación [Wad90].

`fliptree:` Este es otro ejemplo de deforestación. Aquí, el objetivo es dar vuelta a una estructura de árbol dos veces para obtener así, el árbol original. No se proporcionan valores estáticos de entrada.

`foldr.allones:` El objetivo de este programa de prueba es la especialización de una función que concatena un número de listas y, entonces, transforma todos los elementos en 1. La función original se define por medio del combinador de orden superior *foldr*. La especialización considera que una de las listas es conocida.

`foldr.sum:` En este caso, producimos una función especializada para sumar los elementos de una lista (con un prefijo dado) mediante el uso de la función de orden superior *foldr*.

`fun_inter`: Consiste en la especialización de un intérprete funcional simple para un programa dado.

`gauss`: Aquí, el objetivo es especializar la función clásica de Gauss para números naturales mayores o iguales a 5.

`kmp_matcher`: Es un comparador de patrones especializado para un patrón dado. Este programa de prueba se conoce como el test "KMP" [CD89].

`power`: Se trata del ejercicio de especialización mostrado en la Sección 4.4.3 para un exponente definido (3).

La Tabla 4.1 muestra los resultados de los experimentos: Para cada experimento mostramos el tiempo que emplea la herramienta de evaluación parcial previa (NPE *online*) en especializar el programa y el tiempo que le toma a la nueva herramienta realizar la especialización del mismo ejemplo (NPE *offline*). En relación a la nueva herramienta particularizamos en:

anot: el tiempo para analizar y anotar el programa original.

mix: el tiempo para llevar a cabo las computaciones parciales y extraer el programa residual.

La columna "mejora1" indica el grado de mejora del programa especializado con respecto al original con la técnica NPE *online*, y "mejora2" para la técnica NPE *offline*; la "mejora" se computa con el cociente *orig*/*espec*, donde *orig* y *espec* es el tiempo de ejecución absoluto del programa original y especializado respectivamente. Los tiempos están expresados en milisegundos y son el promedio de diez ejecuciones en un PC 2.6 Ghz (Linux-PC, Intel Pentium IV con 1MB de RAM). Los objetivos de entrada se diseñaron para dar tiempos de ejecución razonablemente grandes. Los programas se ejecutaron con el compilador `curry2prolog` de PAKCS [HeAE+04].

Como puede verse en la Tabla 4.1, hemos reducido aproximadamente el tiempo de evaluación parcial a un 20 % de la herramienta original NPE *online*, lo que significa que nuestro objetivo principal se ha cumplido.

La mayoría de los programas de prueba ejemplifican problemas de *especialización* (en lugar de problemas de optimización), lo que explica los resultados tan positivos obtenidos por nuestra herramienta NPE *offline*. Remarcamos, sin embargo, que el nuevo método no es capaz de pasar el llamado test "KMP" [CD89] (véase el programa de prueba `kmp_matcher`). Para pasar el test KMP hay dos requisitos fundamentales: una buena propagación de información y un análisis de terminación más poderoso que evite un alto grado de generalización. Por un lado, nuestro esquema *offline* propaga información tanto como la anterior aproximación *online* (que sí pasa el test KMP),

Tabla 4.1: Tiempos de ejecución de los programas de prueba (milisegundos).

Programa	NPE *online*	mejora1 (*online*)	NPE *offline*		mejora2 (*offline*)
			anot	mix	
ackermann	20214	1.133	51	1200	5.217
allones	4135	1.076	61	340	1.042
fliptree	275	1.015	65	245	1.010
foldr.allones	3602	1.068	143	662	1.901
foldr.sum	12265	1.149	88	630	1.119
fun_inter	14609	—	238	375	—
gauss	1738	1.009	49	916	0.991
kmp_matcher	7065	7.400	110	7550	1.684
power	493	0.638	63	258	1.004
Media	**7155**	**1.81**	**96.44**	**1352.89**	**1.75**

pero nuestro análisis de terminación (implícito) es más simple. Resultará interesante investigar si una aproximación combinada *online*/*offline* es conveniente.

Nuestro evaluador parcial trabaja bien con funciones aritméticas (programa de prueba `ackermann`), con programas para la simplificación de llamadas de orden superior (ejemplos `foldr.allones` y `foldr.sum`), y con un intérprete funcional simple (programa de prueba `fun_inter`), donde las mejoras no se muestran ya que el tiempo de ejecución del programa especializado es cero (i.e., el programa de entrada para el intérprete se ha evaluado completamente).

4.5. Conclusiones

En resumen, hemos introducido, por primera vez, una caracterización para SRTs que asegura la cuasi-terminación de las computaciones por *narrowing* necesario. Este es por sí solo un problema difícil, de interés particular, que no había sido abordado antes. Hemos considerado programas inductivamente secuenciales en lugar de la clase de SRTs caracterizada (que es muy restrictiva) y hemos introducido un algoritmo que anota aquellos subtérminos que pueden causar la no cuasi-terminación por *narrowing* necesario. Además, formulamos una extensión de *narrowing* necesario generalizante guiada por las anotaciones. Finalmente, describimos el modo en que nuestros desarrollos se pueden usar para definir un nuevo esquema NPE correcto y terminante, que asegura la terminación en el estilo *offline*. Los resultados obtenidos a partir de los programas de prueba son muy positivos y demuestran la utilidad de la nueva aproximación.

Por otro lado, debemos mencionar una de las aproximaciones más recientes para asegurar la (cuasi) terminación de programas funcionales, i.e., la aproximación basada en los grafos de cambio de tamaño (grafos *size-change* [LJBA01]), que se han usado ya en el contexto de evaluación parcial en [GJ05]. Nuestro análisis de anotación es muy simple y de ese modo, muy rápido, por lo que es posible suponer que el uso de una análisis más fino (como los grafos *size-change*) permita alcanzar una mayor precisión, sin embargo, es de esperar también una repercusión negativa en el tiempo que se emplee para el análisis.

Adicionalmente, nuestro algoritmo para la anotación de SRTs es independiente del término inicial seleccionado para especializar y su contexto. Esto es, un SRT se anota sólo una vez y entonces se le puede especializar con respecto a diferentes términos sin computar nuevas anotaciones. No consideramos el hecho de que algunos términos iniciales podrían propagar cierta "cantidad" de información estática. Está claro que no estamos explotando la estructura del término inicial.

Por todo ello, en el Capítulo 6 estudiamos la aplicación de los grafos *size-change* junto con un análisis típico de tiempo de enlace: *binding-time analysis*. Esto permitirá por un lado, emplear un análisis más fino de los programas y por otro, aprovechar la propagación de los datos estáticos en el contexto de los lenguajes lógico funcionales.

En el capítulo que sigue mostramos la aplicación de los conceptos aquí introducidos, en el desarrollo de una herramienta de evaluación parcial *offline*.

Capítulo 5

Implementación del evaluador parcial *offline*

En este capítulo nos centramos en la descripción del desarrollo de la herramienta de evaluación parcial *offline* dirigida por *narrowing* para programas Curry. Aunque el nuevo evaluador parcial *offline* es menos preciso que el anterior que sigue la aproximación *online* [AHV02], es más rápido y de este modo permite abordar la especialización de programas más grandes.

Para ello, describimos primero la representación intermedia para programas Curry, enseguida detallamos las facilidades para la meta-programación de programas en la representación intermedia, posteriormente, nos centramos en la exposición de la herramienta, y, finalmente, presentamos un ejemplo de su puesta en marcha.

Parte del material expuesto en este Capítulo se encuentra publicado en [RSV05a].

5.1. La representación abstracta de programas

Los evaluadores parciales incluyen, regularmente, un intérprete (no estándar) [CD93]. Esto repercute en la necesidad de semánticas operacionales muy elaboradas y, a su vez en métodos de evaluación parcial cada vez más complejos. En otros contextos (e.g., [Bon89, GK93, NPT96]) se ha aplicado con éxito una aproximación que consiste en considerar programas escritos en un lenguaje de programación intermedio (con una semántica operacional más simple) y traducirlos automáticamente del nivel fuente a la representación intermedia.

Hanus y Prehofer [HP99] introdujeron una representación abstracta de programas lógico funcionales en la cual los árboles definicionales (usados para guiar la estrate-

$$
\begin{array}{lcl}
\mathcal{R} & ::= & \mathcal{D}_1 \ldots \mathcal{D}_m \\
D & ::= & f(x_1, \ldots, x_n) = e \\
e & ::= & x \\
 & | & c(e_1, \ldots, e_n) \\
 & | & f(e_1, \ldots, e_n) \\
 & | & case \quad e_0 \quad of \quad \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\} \\
 & | & fcase \quad e_0 \quad of \quad \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\} \\
p & ::= & c(x_1, \ldots, x_n)
\end{array}
$$

Figura 5.1: Representación abstracta de programas.

gia de *narrowing* necesario) se hacen explícitos por medio de construcciones "*case*". Este esquema ofrece un control más explícito y conduce a un cálculo más simple con respecto al *narrowing* necesario estándar. En [AHV00b] se incorporó a la representación de [HP99] información acerca del tipo de evaluación de las funciones (flexibles o rígidas) y, posteriormente, en [AHV02] se extendió la representación para cubrir las características extendidas del lenguaje Curry (e.g., restricciones, guardas, etc). En la Figura 5.1 presentamos la sintaxis básica del lenguaje sin considerar las características extendidas del lenguaje Curry. La sintaxis representa un subconjunto de la introducida por [AHV02]. Siguiendo la terminología de [HP99] se le denomina "la representación *plana* de programas".

Dentro de la representación plana de primer orden, un programa $\mathcal{R}$ consiste en una secuencia de definiciones de función $\mathcal{D}$ tales que cada función está definida por una regla cuyo lado izquierdo contiene sólamente argumentos variables. La parte derecha es una expresión e que se compone de variables (e.g., $x, y, z, \ldots$), constructores (e.g., $a, b, c, \ldots$), llamadas a función (e.g., $f, g, \ldots$), y expresiones case para articular la representación del emparejamiento de patrones.

Los programas fuente escritos en Curry se pueden traducir automáticamente a la representación plana; de hecho, coincide con la representación FlatCurry, utilizada durante la compilación de programas Curry [AH00, HeAE$^+$04, LK99].

En este documento conservamos los dos tipos de expresiones *case* introducidas en [AHV02] donde se emplean para representar funciones *flexibles* o *rígidas*. La forma de una expresión case es la siguiente:

$$(f)case \quad e \quad of \quad \{c_1(\overline{x_{n_1}}) \rightarrow e_1; \ldots; c_k(\overline{x_{n_k}}) \rightarrow e_k\}$$

donde e es una expresión, $c_1, \ldots, c_k$ son diferentes constructores del tipo de e, y $e_1, \ldots, e_k$ son expresiones, que a su vez pueden contener expresiones *(f)case*. Las variables $\overline{x_{n_i}}$ son variables *locales* que aparecerán sólo en la subexpresión correspon-

diente e_i. La diferencia entre *case* y *fcase* solo se hace patente cuando el argumento e es una variable libre en tiempo de ejecución: *case* suspende (acción que corresponde al principio de la residuación de la programación lógico funcional) mientras que *fcase* enlaza de manera indeterminista la variable con algún patrón en una rama de la expresión case (principio correspondiente al *narrowing*). Se denominan funciones flexibles si están definidas sólo por *fcase* y rígidas si están definidas sólo por *case*.

Ejemplo 5.1 *Consideremos las reglas que definen la función* `append` *para concatenar dos listas:*

```
append []     ys = ys
append (x:xs) ys = x : append xs ys
```

La representación plana de esta función se muestra a continuación:

```
append(xs,ys) = fcase xs of {   []   → ys;
                              z:zs → z : append(zs,ys) }
```

La parte izquierda de la función tiene sólamente variables, mientras que los posibles términos constructores están representados en los patrones de cada una de las ramas de la estructura case.

5.2. Facilidades de meta-programación en Curry

El lenguaje intermedio FlatCurry para representar programas lógico funcionales surge con la finalidad de ofrecer una interface común para conectar diferentes herramientas que manipulan programas Curry o programas escritos en algún otro lenguaje declarativo lógico funcional (e.g., Toy [LFSH99]).

La implementación de nuestro evaluador parcial hace uso de las facilidades para la meta-programación del lenguaje Curry. Por ejemplo, las librerías `FlatCurry` y `FlatCurryTools` contienen diversas utilidades para manipular programas en representación intermedia.

En FlatCurry todas las funciones se definen en el nivel más alto (i.e., las declaraciones de funciones locales en los programas fuente se hacen globales por medio de *lambda lifting*) y la estrategia de emparejamiento de patrones se define explícitamente a través del uso de las expresiones *case*.

En este contexto, el siguiente tipo de datos representa un programa en FlatCurry:

```
data Prog = Prog String     --nombre del módulo del programa
                 [String]   --módulos importados
                 [TypeDecl] --declaraciones de tipo
                 [FuncDecl] --declaraciones de función
```

```
                [OpDecl]   --declaraciones de operadores
```

Por simplicidad, sólo mostramos los tipos de datos para representar la declaración de funciones:

```
data FuncDecl = Func QName      --nombre cualificado de función
                     Int        --aridad
                     Visibility --pública/privada
                     TypeExpr   --tipo de la función
                     Rule       --regla de la función
```

La estructura de datos para una función contiene el nombre cualificado (i.e., una tupla con el nombre del módulo y el nombre de la función) la aridad, etc. Finalmente, `Rule` representa la regla única que constituye una función:

```
data Rule = Rule [VarIndex] Expr
```

De esta manera, cada función se representa por un sola regla cuya parte izquierda contiene las variables diferentes (`[VarIndex]`) que recibe la función y la parte derecha el propio cuerpo de la función encapsulado en `Expr`.

```
data Expr = Var VarIndex
          | Lit Literal
          | Comb CombType QName [Expr]
          | Or Expr Expr
          | Case CaseType Expr [BranchExpr]

data CombType = FuncCall | ConsCall
data CaseType = Rigid | Flex

data BranchExpr = Branch Pattern Expr
data Pattern = Pattern QName [VarIndex] | LPattern Literal
```

`Expr` es una expresión que puede contener variables, literales (`Lit Literal`, un tipo de dato para representar una constante entera, flotante o caracter), llamadas a función (`FuncCall`), llamadas a constructor (`ConsCall`), disyunciones y expresiones *case*, tanto del tipo flexible como rígido. Las ramas del *case* se representan con el tipo `BranchExpr`, que a su vez hace uso del tipo `Pattern` y recursivamente del tipo `Expr`, i.e., puede haber expresiones anidadas.

Ejemplo 5.2 *Consideremos el programa escrito en el lenguaje Curry contenido en el módulo:* `reverse.curry`*, que incluye la función* `rev` *para invertir una lista haciendo uso de un parámetro acumulador:*

```
rev xs = rr xs []

rr []     ys = ys
rr (x:xs) ys = rr xs (x:ys)
```

La función `rr` *se representa en FlatCurry por medio de la siguiente estructura de datos:*

```
Func ("reverse","rr") 2 Public (FuncType ...)
(Rule [0,1]
 (Case Flex (Var 0)
   [Branch (Pattern ("prelude","[]") [])
           (Var 1),
    Branch (Pattern ("prelude",":") [2,3])
           (Comb FuncCall ("reverse","rr")
              [Var 3,
               Comb ConsCall ("prelude",":")
                    [Var 2, Var 1]
              ])]))
```

Existen funciones (e.g., `readFlatCurry`) mediante las cuales es posible leer un fichero de un programa escrito en Curry y traducirlo a código intermedio FlatCurry. Una vez que tenemos el programa en código intermedio, podemos manipular las expresiones que representan al programa y, finalmente, almacenarlo en un fichero también en FlatCurry utilizando la función `writeFCY`.

5.3. La herramienta de evaluación parcial

En esta sección presentamos, primero, la semántica que sigue el evaluador parcial NPE y después nos enfocamos en la descripción de la propia herramienta de evaluación parcial *offline*, las fases de las que consta y los principales procesos que realiza.

5.3.1. La semántica de residualización

Hanus y Prehofer [HP99] introdujeron una transformación automática para traducir programas inductivamente secuenciales a programas planos. También formularon una semántica operacional apropiada para esos programas: el cálculo de *narrowing* perezoso con árboles definicionales LNT (*Lazy Narrowing with definitional Trees*), el cual es equivalente a *narrowing* necesario sobre programas inductivamente secuenciales. El método de transformación de [HP99] podría extenderse para cubrir programas que

contengan anotaciones de evaluación; a saber, las funciones flexibles serían transformadas mediante *fcase* y las rígidas mediante expresiones *case*. Aún más, el cálculo LNT de [HP99] se puede extender para evaluar correctamente expresiones *case/fcase*.

No obstante, el uso de la semántica estándar durante la evaluación parcial presentaría ciertos inconvenientes. En particular uno de lo principales problemas tiene que ver con la propagación de los enlaces de las variables hacia las partes izquierdas de las reglas residuales [AHV00b]. Recordemos que un resultante $\sigma(s) = t$ se extrae a partir de una derivación de *narrowing* $s \leadsto^+_\sigma t$. En el contexto de los lenguajes lógico funcionales perezosos, la propagación de los enlaces hacia atrás puede provocar una restricción incorrecta sobre el dominio de las funciones (considerando la capacidad para computar expresiones con forma normal en cabeza) y, de esa manera, la pérdida de la corrección de la transformación siempre que algún término con forma normal en cabeza se evalúe durante la evaluación parcial. El siguiente ejemplo de [AHV00b] ilustra este punto.

Ejemplo 5.3 *Consideremos el siguiente programa*

```
isZero  Z           = True
nonEmptyList (x:xs) = True
foo x               = isZero x : []
```

Enseguida, la computación para `foo y` *es:*

$$\texttt{foo y} \leadsto_{\texttt{id}} \underline{\texttt{(isZero y)}}\texttt{:[]} \leadsto_{\{\texttt{y} \mapsto \texttt{Z}\}} \texttt{True:[]}$$

donde la expresión (`isZero y`) : [ ] *está en forma normal de cabeza. El resultante asociado es:*

```
foo Z = True:[]
```

Si observamos ahora la expresión `nonEmptyList (foo (S Z))`*, ésta se puede evaluar a* `True` *en el programa original:*

$$\begin{aligned}\texttt{nonEmptyList (}&\underline{\texttt{foo (S Z)}}\texttt{)}\\ &\leadsto_{\texttt{id}} \underline{\texttt{nonEmptyList (isZero (S Z))}}\texttt{:[]}\\ &\leadsto_{\texttt{id}} \texttt{True}\end{aligned}$$

Sin embargo, si usamos la regla residual de `foo` *(junto con las definiciones originales para* `isZero` *y* `nonEmptyList`*) no podemos llegar al mismo resultado.*

Si se restringe la evaluación de las expresiones en forma normal de cabeza se puede reducir drásticamente (en algunos casos) la capacidad de optimización de la transformación. Por ello, en [AHV00b] se define una versión *residualizante* del cálculo LNT que permite evitar esta restricción. En el nuevo cálculo, los enlaces de las variables se codifican por medio de expresiones *case* (y se consideran código "residual"). En

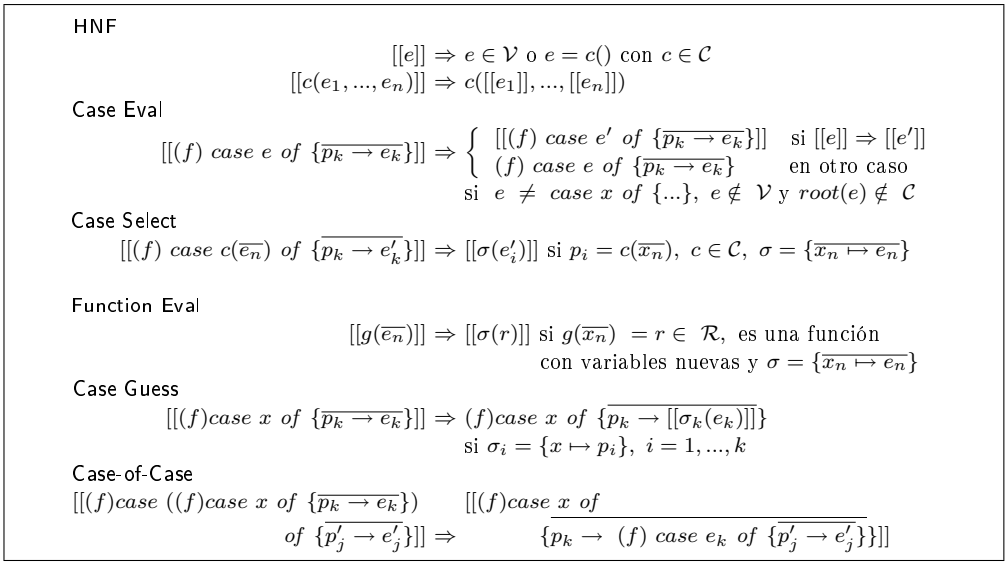


Figura 5.2: El cálculo RLNT [AHV02].

la Figura 5.2 se muestran las reglas de inferencia del cálculo RLNT (*Residualizing* LNT). A continuación explicaremos las reglas de inferencia que definen la relación $\Rightarrow$ de un paso. Los símbolos "$[\![$" y "$]\!]$" en una expresión como $[\![t]\!]$ son puramente sintácticos (i.e., no denotan "el valor de t"). De hecho, se emplean solo para *guiar* las reglas de inferencia y, más importante, para indicar qué parte de una expresión se puede evaluar (dentro de los corchetes) y qué parte se tiene que residualizar definitivamente (fuera de los corchetes). Las reglas del cálculo RLNT son:

HNF: Las reglas HNF son para evaluar expresiones en forma normal en cabeza. Si la expresión es una variable o un constructor constante, los corchetes se eliminan y el proceso de evaluación termina. En caso contrario, la evaluación procede con los argumentos. Aunque a la hora de evaluar los argumentos se presenta cierto indeterminismo, éste puede evitarse fácilmente si se considera una regla de selección fija, e.g., escogiendo el argumento de más a la izquierda que no sea un término constructor.

Case of Case: Esta regla mueve el *case* externo dentro de las ramas del case interno.

Case Select: Esta regla aplica emparejamiento entre su argumento constructor y uno de los patrones de las ramas y continúa la evaluación de la rama seleccionada.

Case Eval: Esta regla inicia la evaluación del argumento del *case* creando una llamada a su subtérmino. Si hay algún progreso en la evaluación del argumento del *case*, i.e., la expresión no se *suspende* (lo cual se denota por el hecho de que los corchetes son eliminados), entonces, se procede a evaluar la expresión *case* resultante; en caso contrario, la expresión completa suspende y se devuelve la expresión *case* original (sin corchetes).

Case Guess: Representa la principal diferencia con respecto al cálculo LNT estándar. Para imitar la instanciación de variables en pasos de *narrowing* necesarios LNT, define esta regla como:

$$[\![fcase\ x\ of\ \{\overline{p_k \rightarrow t_k}\}]\!] \Rightarrow^{\sigma} \quad [\![\sigma(t_i)]\!] \quad \text{si } \sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k$$

Sin embargo, en este caso se propagan los enlaces de las variables hacia atrás. RLNT "residualiza" la estructura *case* y continúa con la evaluación de las diferentes ramas (aplicando la sustitución correspondiente con la finalidad de propagar los enlaces hacia adelante en la computación). Debido a esta modificación, en el cálculo RLNT no hace falta ninguna distinción entre las expresiones *case* flexibles y *case* rígidas. Además, el cálculo resultante no computa *respuestas*; en su lugar, se representan en las expresiones derivadas por medio de expresiones *case* con variables como argumentos.

Function Eval: Esta regla lleva a cabo el desplegado de una llamada a función. Es un desplegado puramente funcional ya que los argumentos de la parte izquierda de la función son todos variables.

La corrección del nuevo cálculo RLNT con respecto al cálculo LNT aparece en [AHV03], primero, introducen la relación auxiliar $\hookrightarrow$, que se define como:

$$fcase\ x\ of\ \{\overline{p_k \rightarrow e_k}\} \hookrightarrow_{\sigma} e_i$$

donde $\sigma = \{x \mapsto p_i\},\ i = 1, \ldots, k$.. Esta relación indeterminista se va a usar para extraer los enlaces codificados por las expresiones *case* residualizadas. Finalmente la equivalencia entre LNT y RLNT se formula en el siguiente teorema:

Teorema 37 *Sea e una expresión, e' una expresión en forma normal en cabeza y $\mathcal{R}$ un programa plano. Por cada derivación LNT $[\![e]\!] \Rightarrow^{*}_{\sigma} e'$ en $\mathcal{R}$, existe una derivación RLNT $[\![e]\!] \Rightarrow^{*} e''$ en $\mathcal{R}$ tal que $e'' \hookrightarrow^{*}_{\sigma} e' \not\hookrightarrow$, y viceversa.*

Informalmente, por cada derivación LNT partiendo de $[\![e]\!]$ a la expresión en forma normal en cabeza e', computando σ, existe una derivación RLNT de $[\![e]\!]$ a alguna expresión e'' en la cual la sustitución computada σ está codificada en e'' por medio de expresiones *case* y se puede obtener por una secuencia (finita) de pasos con $\hookrightarrow$ (derivando la misma expresión e').

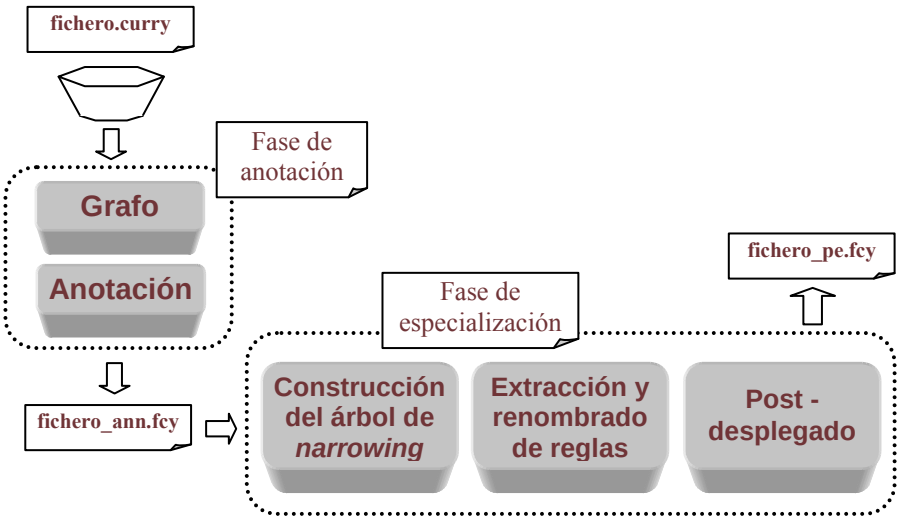


Figura 5.3: Los procesos de la fase de anotación y especialización.

5.3.2. Fases del evaluador parcial

La herramienta de evaluación parcial está basada en la aproximación *offline* de la evaluación parcial dirigida por *narrowing* planteada en el Capítulo 4, en la que se distinguen cláramente la fase de pre-proceso, i.e., de anotación, y la de la propia evaluación parcial.

El esquema completo de la herramienta de evaluación parcial se muestra en la Figura 5.3. Primero, a partir del fichero de entrada `fichero.curry` se construye un grafo (Definición 25) para clasificar las funciones del programa como cíclicas o no cíclicas; después, a partir de la información recogida se aplica el algoritmo de anotación $ann(\mathcal{R})$ introducido en la Definición 32. La salida de la fase de anotación es `fichero_ann.fcy`, el cual constituye la entrada para el primero de los tres subprocesos de la fase de especialización. Finalmente, obtendremos el programa especializado en `fichero_pe.fcy`.

El código de la herramienta de evaluación parcial está estructurado en varios módulos, e.g., los de la fase de anotación son: `Graph.curry` y `Ann.curry`, mientras que los principales módulos de la fase de especialización son:

GNNTrees.curry: Contiene el tipo y funciones para construir árboles de búsqueda de *narrowing*.

Generalization.curry: Agrupa las funciones para la generalización de expresiones.

RLNT.curry: Contiene la implementación del cálculo RLNT para realizar computaciones parciales.

OffPeval.curry: Es el módulo principal que contiene las funciones para la anotación y la especialización de expresiones.

A continuación abundamos en los procesos de las dos fases y también en las funciones de los módulos más importantes.

5.3.3. Fase de anotación

En la Sección 4.2 formalizamos una caracterización sintáctica para programas, a los que llamamos *no crecientes*, los cuales garantizan la cuasi-terminación de las computaciones por *narrowing* necesario. De manera intuitiva, los programas no crecientes cumplen las condiciones siguientes:

- la parte derecha de cada regla en una definición de función es lineal, y,
- si una función pertenece a un ciclo, entonces no contiene llamadas a función anidadas y, además, debe consumir sus parámetros o dejarlos sin cambio (i.e., la profundidad de las variables en las partes derechas de las reglas no debe crecer con respecto a su profundidad en los parámetros).

Gracias a la cuasi-terminación de los programas es posible garantizar la terminación del proceso de evaluación parcial. No obstante, la caracterización de programas no crecientes es restrictiva. Para especializar una clase más amplia de programas, la herramienta de evaluación parcial tomará en cuenta la clase de los programas inductivamente secuenciales (i.e., básicamente, programas Curry).

Una vez computado el grafo de las funciones del programa, aplicamos el algoritmo $ann(\mathcal{R})$ (adaptado a expresiones planas) el cual recorre la expresión, escrita en FlatCurry, que representa el programa objeto. Durante el recorrido se anotan aquellas expresiones que violan las condiciones de la caracterización de programas no crecientes.

Ejemplo 5.4 *Consideremos el programa* `gauss.curry` *en código fuente que computa la función de Gauss para números naturales:*

```
g Z     = Z
g (S n) = + (S n) (g n)

+  Z    y = y
+ (S x) y = S (+ x y)
```

donde + representa la función (no predefinida) de la suma de números naturales. $ann(\mathcal{R})$ lleva a cabo la siguiente anotación:

```
g  Z    = Z
g (S n) = + (S n) (GEN (g n))

+  Z    y = y
+ (S x) y = S (+ x y)
```

donde (la función identidad) `GEN` *es la función auxiliar que usamos para anotar.*

La adaptación del algoritmo para anotación de sistemas inductivamente secuenciales $ann(\mathcal{R})$ al lenguaje FlatCurry no es difícil. Básicamente, en lugar de inspeccionar las partes derechas de las reglas se inspeccionan las *ramas* de cada definición de función. Por ejemplo para la función "g" introducida arriba, el módulo de anotación devuelve la siguiente expresión en FlatCurry:

```
Func ("gauss","g") 1 Public (FuncType ...)
(Rule [0]
(Case Flex (Var 0)
 [Branch (Pattern ("gauss","Z") [])
    (Comb ConsCall ("gauss","Z") []),
  Branch (Pattern ("gauss","S") [1])
    (Comb FuncCall  ("gauss","+")
      [(Comb ConsCall ("gauss","S") [(Var 1)]),
       (Comb FuncCall ("gauss","GEN")
         [(Comb FuncCall  ("gauss","g") [(Var 1)]
                                        )])])]))
```

donde "`GEN`" está definida en el propio fichero a especializar[1].

5.3.4. Fase de especialización

El evaluador parcial *offline* dirigido por *narrowing* aplica tres procesos secuenciales (Figura 5.3) a los programas FlatCurry: construcción del árbol de *narrowing*, extracción de nuevas reglas y renombramiento y, finalmente, el proceso de post-desplegado.

[1] Está claro que "`GEN`" podría ser parte de la librería estándar del preludio de Curry como ocurre con la función "`PEVAL`" del evaluador parcial *online* original.

Construcción del árbol de búsqueda de *narrowing*

Este proceso se basa en la extensión de *narrowing necesario generalizante* (introducido formalmente en la Definición 33) para manipular programas anotados. Básicamente, las operaciones efectuadas son

Generalización: si una expresión FlatCurry contiene anotaciones (i.e., ocurrencias de la función "`GEN`"), la expresión es transformada en varias expresiones mediante el reemplazo de las subexpresiones anotadas por variables nuevas.

Narrowing: si la expresión en FlatCurry es una llamada a función y no contiene anotaciones, se ejecuta un paso de *narrowing* necesario (empleando la semántica RLNT).

Descomposición: en caso de que la expresión en FlatCurry sea una llamada a constructor, entonces ésta se reduce a sus argumentos (si los hay).

Estas operaciones se aplican a una expresión inicial y subsecuentes (obtenidas por las propias operaciones) hasta que no aparezcan nuevas expresiones diferentes (módulo renombramiento de variables). El algoritmo concreto lo detallaremos en la Sección 5.4.3. Las expresiones computadas y sus derivaciones conforman el *árbol de búsqueda* de *narrowing* necesario generalizante (la estructura de datos utilizada la mostramos en la Sección 5.4.1). A partir del árbol de *narrowing* extraeremos las reglas residuales.

Ejemplo 5.5 *En la Figura 5.4 mostramos (en código fuente) el árbol de búsqueda computado por narrowing necesario generalizante para la llamada a función inicial*

```
g (S (S x))
```

con respecto al programa anotado del Ejemplo 5.4. En la Figura 5.4, $\mathtt{w}_1 \ldots \mathtt{w}_5$ *son variables nuevas, los pasos de generalización se denotan por* $\rightarrow_\bullet$*, los pasos de narrowing necesario por* $\rightarrow_\sigma$*, donde* σ *es la sustitución computada, y los pasos de descomposición con* $\rightarrow_\mathcal{C}$*.*

Extracción de nuevas reglas y renombramiento

El segundo proceso extrae los resultantes a partir del árbol de búsqueda (con expresiones FlatCurry). Con este propósito, implementamos una función que atraviesa el espacio de búsqueda del *narrowing* necesario generalizante y extrae una regla residual (sin anotaciones "`GEN`") por cada paso de *narrowing* propiamente dicho. Esto es, no se tienen en cuenta los pasos de generalización o de descomposición a la hora de generar el programa residual. Por ejemplo, el programa residual asociado al espacio de búsqueda de la Figura 5.4 es como sigue:

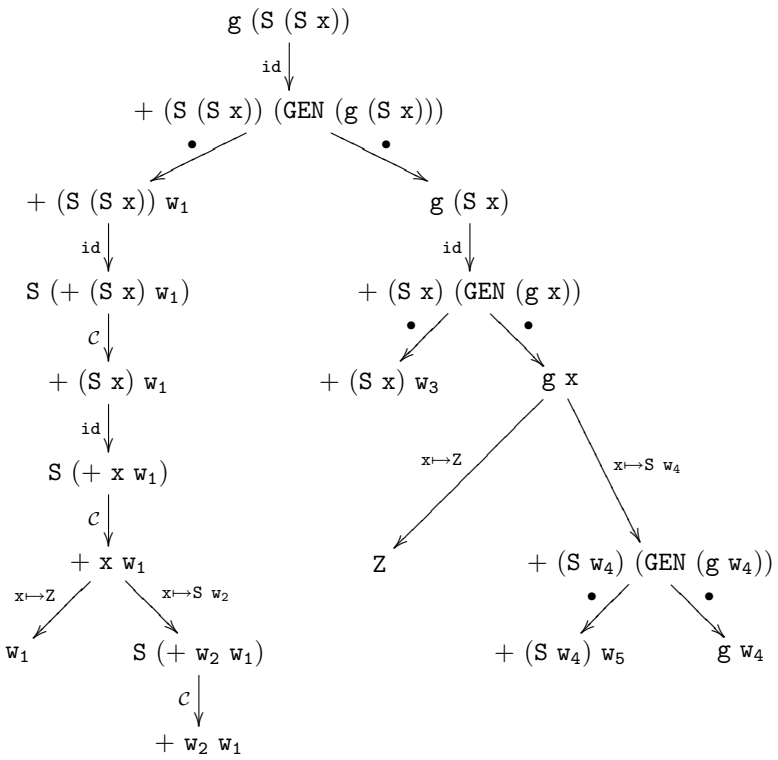


Figura 5.4: Árbol de *narrowing* necesario generalizante para el Ejemplo 5.4.

```
g (S (S x)) = + (S (S x)) (g (S x))
g (S x)     = + (S x) (g x)
g x         =  fcase x of
                      { Z    → Z;
                        S w4 → + (S w4) (g w4) }
+ (S (S x)) w1 = S (+ (S x) w1)
+ (S x)     w1 = S (+ x w1)
+  x        w1 = fcase x of
                       { Z    → w1;
                         S w2 → S (+ w2 w1) }
```

Como podemos ver, los resultantes no tienen una notación plana legal, i.e., no son reglas de programa válidas. Para ello, aplicamos el proceso de renombramiento (Sección 3.4). Para el ejemplo obtenemos:

```
g2(x)     = +2(x,g1(x))
g1(x)     = +1(x,g0(x))
g0(x)     = fcase x of
                  {  Z    → Z;
                     S w4 → +1(w4,g0(w4))  }
+2(x,w1) = S (+1(x,w1))
+1(x,w1) = S (+0(x,w1))
+0(x,w1) = fcase  x of
                  {  Z    → w1;
                     S w2 → S (+0(w2,w1))}
```

Post-desplegado

Finalmente, el tercer proceso de la fase de especialización lleva a cabo un post-proceso de desplegado [JGS93] para eliminar llamadas a función intermedias. Esta fase ya estaba incluida en el evaluador parcial *online* y no requirió ninguna extensión. En nuestro ejemplo, la fase de post-desplegado devuelve las siguientes funciones especializadas.

```
g2 x   = S (S (+ x (g1 x)))
g1 x   = S (+ x (g0 x))
g0 x   = fcase x of
                 { Z    → Z;
                   S w4 → S (+ w4 (g0 w4)) }
+ x w1 = fcase x of
                 { Z    → w1;
                   S w2 → S (+ w2 w1) }
```

5.4. Construcción del árbol de *narrowing*

La construcción del árbol de búsqueda de *narrowing* es el proceso central en la especialización de programas lógico funcionales. Constituye, asimismo, la parte del esquema de evaluación parcial dirigido por *narrowing* que presenta mayores contrastes en su aproximación *offline* con respecto a la versión *online*. En esta sección detallamos los tres aspectos principales que tienen que ver con la construcción del árbol de búsqueda de *narrowing*:

- La estructura de datos donde se almacena el árbol de búsqueda de *narrowing*.
- La semántica para realizar computaciones parciales.
- El algoritmo de control para la construcción del árbol.

5.4.1. Estructura de datos del árbol de búsqueda de *narrowing*

El nuevo tipo de dato que define la estructura para almacenar el árbol de búsqueda generado por el *narrowing* necesario generalizante es el siguiente:

```
data GNNTree =  Leaf Expr
              | LeafM  Expr Expr
              | TreeN  Expr [GNNTree]
              | TreeNM Expr [GNNTree]
              | TreeD  Expr [GNNTree]
              | TreeG  Expr [GNNTree]
```

`GNNTree` contiene una definición de constructores para representar cada posible tipo de nodo:

`Leaf:` para una expresión en una hoja del árbol (e.g., una llamada a constructor con argumentos constructores o una variable).

`LeafM:` para una expresión cuya computación ha sido detenida porque es igual módulo renombramiento de variables a una anterior que ya ha sido procesada.

`TreeN:` para una expresión sobre la que se aplica un paso RLNT (e.g., una llamada a función, un *case*, etc).

`TreeNM:` para una llamada a función que se computa por primera vez (nodos memorizados, llamados también puntos de control). Este nodo inicialmente se etiqueta con `TreeN`, pero en un segundo recorrido al árbol, posterior a la especialización, se cambia a `TreeNM` (tomando como base los nodos `LeafM`) para efectos de extracción del programa residual.

`TreeD:` para una llamada a constructor que será descompuesta en sus argumentos.

`TreeG:` para una expresión que se va a generalizar y a producir varias subexpresiones.

En todos los casos `[GNNTree]` contendrá los subárboles hijos de la expresión `Expr`. Por ejemplo, el árbol de *narrowing* de la Figura 5.4, contiene los siguientes tres nodos `TreeNM` (memorizados):

```
(TreeNM (+ (S x) w1) [...] )
(TreeNM (+  x    w1) [...] )
(TreeNM (g x)        [...] )
```

Encontramos, además, expresiones en el árbol que no han sido procesadas (i.e., `LeafM`) porque son iguales (módulo renombramiento de variables) a nodos previos

```
(LeafM (+ w2 w1) (+  x    w1) )
```

```
(LeafM (+ (S x ) w3) (+ (S x) w1) )
(LeafM (+ (S w4) w5) (+ (S x) w1) )
(LeafM (g w4)        (g x) )
```

donde la primera es la expresión suspendida, y la segunda es la variante previamente computada. Los nodos `TreeG` que contienen anotaciones para generalización en el árbol en referencia son:

```
(TreeG (+ (S (S x)) (GEN (g (S x)))) [...] )
(TreeG (+ (S x )    (GEN (g x ))) [...] )
(TreeG (+ (S w4)    (GEN (g w4))) [...] )
```

A continuación presentamos dos expresiones planas que se van reducir aplicando alguna de las reglas de la semántica de residualización RLNT y que corresponden al inicio del árbol de la Figura 5.4:

```
(TreeN (gauss (S (S v0))) [...] )
(TreeN (fcase (S (S v0)) of
          { Z       → Z;
           (S v102) → add (S v102) (GEN (gauss v102))
          })        [...])
 . . .
```

La primera expresión: `(TreeN (gauss (S (S v0))) [...] )` es una llamada a función, al desplegarse se obtiene una segunda expresión, que estará sujeta a un segundo paso de computación aplicando otra regla de la semántica RLNT.

El módulo `GNNTrees.curry` de la herramienta de evaluación parcial es el que contiene la definición del tipo `GNNTree` y las funciones correspondientes (e.g., generalización de expresiones) para manipular los árboles de búsqueda de *narrowing*.

5.4.2. RLNT

El módulo `RLNT.curry` de la herramienta de evaluación parcial contiene una implementación del cálculo RLNT [AHV03] (introducido en la Sección 5.3.1) para realizar computaciones parciales. El núcleo de este módulo es la función `rlnt`, que toma una expresión de entrada (sin anotaciones de generalización) en FlatCurry y, al evaluarse, produce una nueva expresión mediante la aplicación de *sólo* una de las reglas siguientes, de acuerdo a la forma de la expresión.

`f (`e_1`,...,`e_s`)`**:** Se trata de una llamada a función. `rlnt` devuelve el cuerpo de `f` desplegado.

`(f)case x of {`$p_1 \rightarrow e_1$`; ...; `$p_k \rightarrow e_k$`}`**:** Aquí, `rlnt` devuelve la expresión *case* con los enlaces de la variable propagados a todas las ramas del *case*. El *case*

se devuelve al algoritmo de control del evaluador parcial para que continúe con la evaluación de cada una de las ramas.

`(f)case (Lit) of {`$p_1 \rightarrow e_1$`; ...; `$p_k \rightarrow e_k$`}`**:** En FlatCurry es posible que el argumento de un *case* sea un literal, `Lit`, i.e., una constante entera, flotante o de tipo carácter. En este caso `rlnt` aplica el emparejamiento a uno de los patrones y devuelve la expresión asociada. Se corresponde a la regla Case Select.

`(f)case C(`$e_1,\ldots,e_s$`) of {`$p_1 \rightarrow e_1$`; ...; `$p_k \rightarrow e_k$`}`**:** Se devuelve la expresión correspondiente a la rama cuyo patrón empareja con el argumento del *case*. Corresponde también a la regla Case Select del cálculo RLNT.

`(f)case f(`$e_1,\ldots,e_s$`) of {`$p_1 \rightarrow e_1$`; ...; `$p_k \rightarrow e_k$`}`**:** Ahora el argumento del *case* es una llamada a función (regla Case Eval del cálculo RLNT), la cual es desplegada y el *case* es devuelto con su argumento desplegado.

`(f)case ((f)case x of {`$p_1 \rightarrow e_1$`; ...; `$p_k \rightarrow e_k$`}) of {`$\overline{p'_j \rightarrow e'_j}$`}`**:** Aquí, el argumento del *case* es otro *case*. En esta expresión el *case* externo será pasado a las ramas del *case* más interno mediante la aplicación de la regla Case-of-Case del cálculo RLNT.

`(f)case ((f)case (exp) of {`$p_1 \rightarrow e_1$`; ...; `$p_k \rightarrow e_k$`}) of {`$\overline{p'_j \rightarrow e'_j}$`}`**:** Si la expresión *case* anidada contiene como argumento una expresión que no es una variable `rlnt` procede con la expresión interna. La aplicación recursiva de `rlnt` es: `((f)case (rlnt ((f)case (exp) of {`$p_1 \rightarrow e_1$`; ...; `$p_k \rightarrow e_k$`})) of {`$\overline{p'_1 \rightarrow e'_1}$`})` que se corresponde con la regla Case Eval del cálculo RLNT.

En las reglas podemos observar que la función `rlnt` realiza la computación específica a cada expresión en FlatCurry sin considerar ningún aspecto de control de terminación.

5.4.3. Aspectos de control

En esta sección discutimos primero los aspectos de control del nuevo esquema *offline* de evaluación parcial dirigido por *narrowing*. Luego describiremos el módulo de especialización `OffPeval.curry` que implementa el algoritmo principal de especialización de la herramienta.

Control local y control global

En el algoritmo paramétrico de evaluación parcial dirigido por *narrowing* de la Sección 3.6.2, se distinguen dos niveles de control que permiten garantizar la terminación del proceso: el *control local*, el cual se computa por medio de una regla de desplegado

y *el control global* que se lleva a cabo por medio del operador de generalización. De manera trivial, para asegurar la terminación del algoritmo, se debe garantizar la terminación local y la terminación global, i.e., los árboles parciales computados por la regla de desplegado deben ser finitos y la construcción iterativa de árboles parciales debe concluir también.

En lo que respecta a la aproximación *offline* los aspectos de control se realizan del modo que sigue:

control local: Básicamente el control local está determinado por el número de operaciones del cálculo RLNT que se aplican, en concreto, aplicamos un paso de reducción. En la Sección 5.4.2 se describe el módulo que computa los pasos de reducción por las reglas RLNT.

control global: El control global analiza la expresión a reducir si contiene algún subtérmino anotado o está encabezado por constructor entonces se procede a su generalización o a su descomposición, respectivamente. El control global analiza si hay nuevas expresiones (módulo renombramiento de variables) en cuyo caso se deben computar; en caso contrario se concluye.

A grandes rasgos se construye un árbol de *narrowing*. En cada oportunidad se ejecuta un paso RLNT (crece la rama del árbol) y se analiza la expresión resultante para decidir si se debe continuar, descomponer, generalizar o concluir. De este modo el crecimiento del árbol se controla de manera local y el carácter finito del árbol se asegura gracias al control global.

Módulo de especialización

En la Figura 5.5 mostramos el algoritmo principal del módulo de especialización `OffPeval.curry` que construye el árbol finito de búsqueda de *narrowing* generalizante. El algoritmo principal se llama (`gnn`) y recibe tres argumentos:

- el término inicial `t` que va a evaluarse parcialmente.
- el programa `p` respecto al cual se va a realizar la especialización.
- un estructura de datos `m` para llevar el registro de las expresiones computadas (memoria).

`gnn` procede con la inspección de la expresión t como se indica a continuación:

`isCons t`: Si `t` está formada únicamente por llamadas a constructor, es una variable o un literal, entonces, `gnn` devuelve un nodo terminal `Leaf`, concluyendo la derivación de esa rama del árbol de búsqueda.

Entrada: un programa `p`, un término inicial `t` y la memoria `m`
Salida: un árbol de *narrowing* necesario generalizante τ
Inicialización: `m`=[ ]
Llamada inicial: `gnn p t m`

```
gnn p t m
  case t of
    isCons t = (Leaf t, m)
    isConsRooted t = (TreeD t (gnnD p t m))
    hasGEN t y isOpRooted t = (TreeG t (gnnG p t m))
    existsRen t m = (LeafM t, m)
    otherwise = (TreeN t (gnnRLNT p t m))
```

Resultado: τ

Figura 5.5: Algoritmo principal del módulo de especialización *offline*.

`isConsRooted t`**:** Si `t` es una expresión encabezada por un constructor, entonces corresponde la creación de un subárbol de descomposición `TreeD` y el cómputo recursivo de sus subexpresiones por medio de la función `gnnD`.

`hasGEN t y isOpRooted t`**:** Si la expresión `t` contiene anotaciones para ser generalizadas y está encabezada por una llamada a función, entonces se crea un subárbol de generalización `TreeG` y se computan recursivamente las subexpresiones generalizadas por medio de `gnnG`.

`existsRen t m`**:** Si la expresión es igual a una anterior en `m` por módulo renombramiento de variables entonces se genera un nodo terminal `LeafM`.

`otherwise`**:** Si ninguno de los casos anteriores se cumple se aplica un paso de *narrowing* necesario creando el nodo `TreeN` para el subárbol correspondiente e invocando la función `gnnRLNT`.

La función `gnn` se invoca recursivamente desde las funciones `gnnD`, `gnnG` y `gnnRLNT`. El proceso continúa iterativamente hasta que no se producen nuevas expresiones para computarse. De esta manera, se construye una estructura de datos finita `GNNTree` a partir de las derivaciones realizadas sobre el término inicial `t`.

5.5. El evaluador parcial *offline* en la práctica

En esta sección describimos el uso de la herramienta de evaluación parcial *offline*.

El evaluador parcial *offline* se ejecuta en el compilador `curry2prolog` del entorno PAKCS [HeAE+04] de Curry. Así, primero cargamos el módulo `OffPEval` en el entorno de PAKCS:

```
prelude> :l OffPeval
```

Entonces, iniciamos el proceso de especialización con el comando `annotate` aplicado al programa que vamos a anotar. En la Figura 5.6, mostramos una sesión de anotación para el programa `simpleInt2.curry` que se encuentra en el directorio `ictccd`, con el siguiente comando:

```
OffPeval> annotate "ictccd/simpleInt2"
```

A continuación, cargamos y mostramos en pantalla el programa recién anotado: `simpleInt2_ann.fcy` (Figura 5.6).

Ahora, procedemos a la evaluación parcial propiamente dicha del programa. En la Figura 5.7 podemos observar una sesión de especialización. Para ello cargamos nuevamente `OffPeval` y mediante la orden

```
OffPeval> mix "ictccd/simpleInt2"
```

iniciamos la transformación del programa `simpleInt2_ann.fcy`; enseguida, cargamos y mostramos en el entorno de ejecución el programa especializado `simpleInt2_pe.fcy` y, finalmente llevamos a cabo una prueba de ejecución. La herramienta asume que la primera función del programa es la que inicia la especialización.

El usuario puede incluir en la presentación de salida información de las diferentes etapas intermedias de la especialización activando un conjunto de banderas de tipo `Boolean` en el módulo `OffPeval`. Las banderas son:

`gnnFlag`**:** Mostrar el árbol de *narrowing* necesario generalizante construido.

`gnnMemoFlag`**:** Visualizar el árbol de *narrowing* necesario generalizante incluyendo nodos memorizados.

`resulFlag`**:** Mostrar resultantes sin renombramiento.

`renFlag`**:** Para mostrar el cómputo del renombramiento independiente.

`renresFlag`**:** Para desplegar los resultantes renombrados.

`postFlag`**:** El programa final después del post-desplegado.

Por omisión, todos sus valores son `False` (i.e., no se muestra ninguna información intermedia).

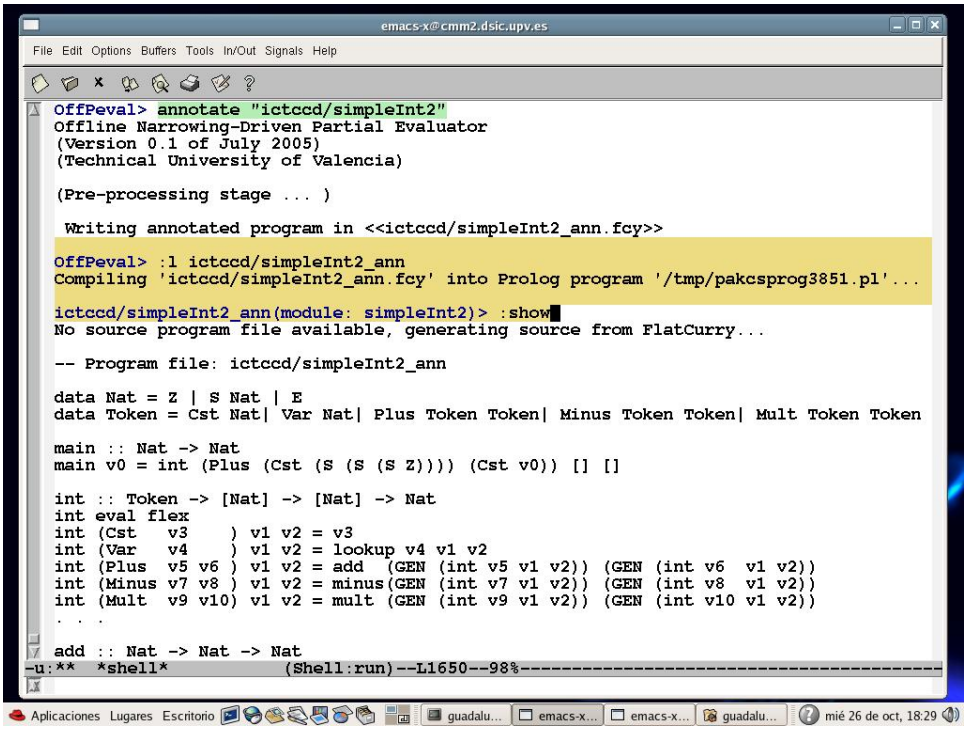


Figura 5.6: Sesión de anotación.

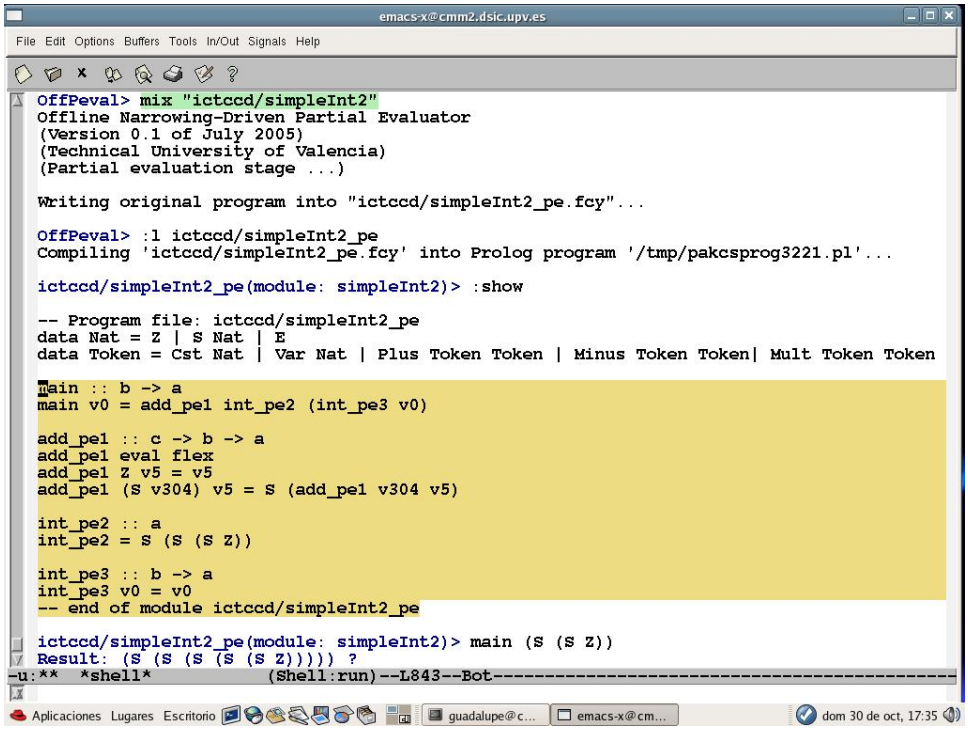


Figura 5.7: Sesión de especialización.

5.6. Conclusiones

Para concluir, compararemos el esquema *offline* respecto al *online* desde dos puntos de vista, primero desde el punto de vista algorítmico y luego desde la perspectiva de la eficiencia de las dos aproximaciones.

El nuevo esquema de especialización aporta un algoritmo de especialización más simple con respecto a la aproximación de evaluación parcial *online* dirigida por *narrowing*.

Además, el algoritmo de especialización *offline* aplica un test muy simple para asegurar terminación, evitando que se computen dos expresiones iguales módulo renombramiento de variables. Gracias a la propiedad de la cuasi-terminación sobre la que se basa el esquema *offline*, la terminación está garantizada. El test se ejecuta de manera adecuada con expresiones grandes. Por otro lado, en el esquema *online* se hace uso de un test de subsumción homeomórfica el cual implica un coste de computación (tiempo de especialización) mayor. En la medida que las expresiones crecen es más costoso evaluar el test de subsumción y, por ello, garantizar la terminación es más complicado.

Desde un punto de vista de eficiencia diremos que los experimentos reportados en el Capítulo 4 indican que el esquema *offline* redujo significativamente los tiempos de evaluación parcial, es decir, el promedio de anotación junto con especialización es el 20 % del promedio de tiempo requerido por la herramienta original NPE mientras obtenemos aún mejoras en los programas especializados.

Capítulo 6

Análisis por grafos *size-change*

En este capítulo presentamos una mejora al esquema de anotación de programas para la evaluación parcial *offline* dirigida por *narrowing* definido en [RSV05b] y detallado en el Capítulo 4. El nuevo método hace uso de la información proporcionada por los llamados grafos *size-change*[1] y la salida de un análisis de tiempo de enlace (BTA, del inglés *Binding-Time Analysis*) estándar para llevar a cabo las anotaciones. Parte del material incluido en este capítulo se ha publicado en [ARSV06].

6.1. Antecedentes

En el Capítulo 4 identificamos una clase de sistemas de reescritura cuasi-terminantes llamados *no crecientes*. Esta caracterización es sintáctica y, por lo mismo, es razonablemente fácil de comprobar. Sin embargo, es demasiado restrictiva para poder aplicarse en la práctica. Por ello, hemos introducido, en el mismo capítulo, un esquema *offline* para la evaluación parcial dirigida por *narrowing* basado en los siguiente puntos:

1. La anotación de las expresiones del programa que *violan la propiedad de programa no-creciente* y

2. La definición de una extensión adecuada de *narrowing* necesario para realizar computaciones parciales en la que los subtérminos anotados son *generalizados* (y, de esta manera, se asegura la terminación del proceso de especialización).

[1] Conservamos el nombre de los grafos en el idioma inglés para estar acordes con la reciente literatura al respecto.

Ejemplo 6.1 *Consideremos el siguiente SRT:*

$$\begin{array}{rcl} \texttt{f Z y} & = & \texttt{y} \\ \texttt{f (S x) y} & = & \texttt{f x (S y)} \end{array}$$

El método del Capítulo 4 (Definición 32), anota el segundo argumento de la llamada a función `f` *que se encuentra en la segunda regla ya que no es no creciente.*

$$\begin{array}{rcl} \texttt{f Z y} & = & \texttt{y} \\ \texttt{f (S x) y} & = & \texttt{f x} \bullet \texttt{(S y)} \end{array}$$

Sin embargo, si llamamos a la función `f` *con un primer argumento con valor conocido (estático), entonces el programa termina ya que al decrecer dicho argumento, en algún momento se aplicaría el caso base (i.e., la primera regla) y la computación terminaría. La anotación del segundo argumento implica una pérdida innecesaria de información durante la especialización.*

El Ejemplo 6.1 muestra que la caracterización del Capítulo 4 para anotar programas asegura terminación pero pierde precisión. De ahí que resulte necesario un método de anotación más preciso.

En este capítulo reemplazamos la caracterización de sistemas de reescritura no crecientes por el uso de un análisis basado en los *grafos size-change* [LJBA01]. Los grafos aproximan los cambios en el tamaño de los parámetros en las sucesivas llamadas a función. En particular, usamos la información de los grafos *size-change* para identificar una forma específica de cuasi-terminación, en la que se producen sólamente un número finito de diferentes llamadas a función (módulo renombramiento de variables) para su computación. Con este fin, vamos a hacer uso de la información calculada por un proceso de BTA. Dicha información nos permite saber si un argumento específico de una función es estático o dinámico (desconocido). Cuando la información reunida a partir del uso combinado de grafos *size-change* y el BTA no nos permite inferir que el sistema de reescritura cuasi-termina, procedemos como en el Capítulo 4 y anotamos los términos problemáticos que se deben generalizar en tiempo de evaluación parcial. Finalmente, presentamos los resultados de algunos programas que utilizamos para aplicar tests al desarrollo del nuevo análisis y concluimos.

6.2. Grafos *size-change*

En esta sección presentamos las nociones básicas relacionadas con los grafos *size-change* a partir de [KNT99, LJBA01, TG05].

Primero necesitamos algunas definiciones: un *orden* $\succ$ es una relación binaria, antisimétrica y transitiva y un *preorden* $\succsim$ es una relación binaria reflexiva y transitiva.

Una relación binaria $\succ$ está *bien fundada* sii no existe una secuencia decreciente infinita $t_0 \succ t_1 \succ t_2 \succ \ldots$ En lo sucesivo, diremos que un orden $\succ$ es *cerrado bajo sustituciones* (o *estable*) si $s \succ t$ implica que $\sigma(s) \succ \sigma(t)$ para todo $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ y sustitución σ.

Definición 38 (par de reducción) *Decimos que $(\succsim, \succ)$ es un par de reducción sobre $\mathcal{T}(\Sigma, \mathcal{V})$ si $\succsim$ es un preorden y $\succ$ es un orden bien fundado sobre términos donde, tanto $\succsim$ como $\succ$, son cerrados bajo sustitución y compatibles (i.e., $\succsim \circ \succ \subseteq \succ$ y $\succ \circ \succsim \subseteq \succ$ pero $\succsim \subseteq \succ$ no es necesario). Además, $s\ R\ t$ implica que $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ para todo $R \in \{\succsim, \succ\}$ y términos s y t.*

Definición 39 (grafo *size-change*) *Sea $(\succsim, \succ)$ un par de reducción. Para cada regla $f(\overline{s_n}) \to r$ de un SRT $\mathcal{R}$ y cada subtérmino $g(\overline{t_m})$ de r donde $g \in \mathcal{D}$, definimos un grafo* size-change *como sigue:*

- *El grafo tiene n nodos de salida marcados con $\{1_f, \ldots, n_f\}$ y m nodos de entrada marcados con $\{1_g, \ldots, m_g\}$.*
- *Si $s_i \succ t_j$, entonces hay un arco dirigido marcado con "$\succ$" del nodo de salida i_f al nodo de entrada j_g. En otro caso, si $s_i \succsim t_j$, entonces hay un arco marcado con "$\succsim$" de i_f a j_g. En caso contrario, no hay un arco entre i_f y j_g. Omitimos frecuente los subíndices f y g cuando está claro por el contexto.*

Un grafo size-change es por tanto un grafo bipartito $G = (V, W, E)$ donde $V = \{1_f, \ldots, n_f\}$ y $W = \{1_g, \ldots, m_g\}$ son las etiquetas de los nodos de salida y de los nodos de entrada, respectivamente, y $E \subseteq V \times W \times \{\succsim, \succ\}$ denota los arcos.

A continuación centramos nuestra atención en los ciclos del programa, i.e., en las funciones (potencialmente) cíclicas; para ello introducimos la noción de *multigrafo* y de concatenación de grafos.

Definición 40 (multigrafo, concatenación, multigrafo maximal) *Cada grafo size-change de $\mathcal{R}$ es un multigrafo de $\mathcal{R}$ y, si $G = (\{1_f, \ldots, n_f\}, \{1_g, \ldots, m_g\}, E_1)$ y $H = (\{1_g, \ldots, m_g\}, \{1_h, \ldots, p_h\}, E_2)$ son multigrafos con respecto al mismo par de reducción $(\succsim, \succ)$, entonces la concatenación $G \cdot H = (\{1_f, \ldots, n_f\}, \{1_h, \ldots, p_h\}, E)$ es también un multigrafo de $\mathcal{R}$. Para $1 \leq i \leq n$ y $1 \leq k \leq p$, E contiene un arco a partir i_f a k_h sii E_1 contiene un arco de i_f a algún j_g y E_2 contiene un arco de j_g a k_h. Si hay un j_g donde el arco de E_1 o E_2 está etiquetado con "$\succ$", entonces el arco en E se etiqueta con "$\succ$" también. En caso contrario, se etiqueta con "$\succsim$". Un multigrafo G se denomina maximal si sus nodos de entrada y sus nodos de salida están etiquetados con $\{1_f, \ldots, n_f\}$ para alguna función f y si éste es idempotente, i.e., $G = G \cdot G$.*

Enseguida concretamos el par de reducción $(\succsim, \succ)$ que utilizaremos en ejemplos subsecuentes del modo que sigue:

- $s \succsim t$ sii $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ y para todo $x \in \mathcal{V}ar(t)$, $dv(t,x) \leqslant dv(s,x)$;
- $s \succ t$ sii $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ y para todo $x \in \mathcal{V}ar(t)$, $dv(t,x) < dv(s,x)$.

donde $dv(t,x)$ [CK96] representa la profundidad de una variable x en un término constructor t (véase la Definición 27). En el caso de que un término t no sea constructor, asumimos que $dv(t,x)$ devuelve un valor muy elevado (infinito).

Ejemplo 6.2 *Consideremos el siguiente ejemplo que computa la inversa de un lista:*

```
rev []          =  []
rev (x : xs)    =  app (rev xs) (x : [])

app [] y        =  y
app (x : xs) y  =  x : app xs y
```

los grafos size-change para este programa, en la notación estándar son:

$$G_1: \quad 1_{rev} \xrightarrow{\succ} 1_{rev} \qquad G_2: \quad 1_{rev} \overset{\succsim}{\searrow} \begin{matrix} 1_{app} \\ 2_{app} \end{matrix} \qquad G_3: \quad \begin{matrix} 1_{app} \xrightarrow{\succ} 1_{app} \\ 2_{app} \xrightarrow{\succsim} 2_{app} \end{matrix}$$

Para ilustrar cómo se forman los grafos size-change, vamos a mostrar el término t asociado a cada nodo de la forma t_{nodo}.

El grafo G_1 resulta de la relación $(x : xs)_{1_{rev}} \succ (xs)_{1_{rev}}$ entre los términos (subrayados) de la segunda regla:

```
rev (x : xs)    =  app (rev xs) (x : [])
```

i.e., la profundidad de la variable `xs` *decrece. El grafo G_2 modela la relación $(x : xs)_{1_{rev}} \succsim (x : [])_{2_{app}}$ entre los argumentos de entrada de* `rev` *y los argumentos en la llamada a la función* `app`*, también de la segunda regla:*

```
rev (x : xs)    =  app (rev xs) (x : [])
```

la relación $\succsim$ se ha establecido porque la profundidad de la variable `x` *se mantiene. El nodo 1_{rev} no se conecta con 1_{app} porque el término asociado al nodo 1_{app} no es un término constructor. Finalmente el grafo G_3 modela las relaciones $(x : xs)_{1_{app}} \succ (xs)_{1_{app}}$ e $(y)_{2_{app}} \succsim (y)_{2_{app}}$ de la cuarta regla del programa*

```
app (x : xs) y  =  x : app xs y
```

Los grafos maximales se obtienen a partir del cierre transitivo de los grafos:

Definición 41 (cierre transitivo de los grafos [LJBA01]) *Decimos que $\mathcal{S}$ es el cierre transitivo de los grafos size-change para un programa $\mathcal{R}$ si $\mathcal{S}$ es el conjunto de multigrafos más pequeño que contiene los grafos size-change tales que la concatenación $G_1 \cdot G_2$ está en $\mathcal{S}$, siempre que $G_1 \in \mathcal{S}$ y $G_2 \in \mathcal{S}$.*

El algoritmo para construir el cierre de $\mathcal{S}$ consiste básicamente en:

1. Incluir en $\mathcal{S}$ cada grafo *size-change* obtenido a partir del programa.
2. Para cada grafo *size-change* $G = (\{1_f, \ldots, n_f\}, \{1_g, \ldots, m_g\}, E_1)$ donde $G \in \mathcal{S}$ y otro grafo $H = (\{1_g, \ldots, m_g\}, \{1_h, \ldots, p_h\}, E_2)$ también con $H \in \mathcal{S}$ agregar $G \cdot H$ en $\mathcal{S}$ siempre que $G \cdot H \notin \mathcal{S}$. Este segundo paso se repite hasta que no se obtienen nuevos multigrafos.

Ejemplo 6.3 *Consideremos nuevamente el Ejemplo 6.2, su cierre $\mathcal{S}$ está formado por los multigrafos:*

$$G_1: \quad 1_{rev} \xrightarrow{\succ} 1_{rev} \qquad G_2: \quad 1_{rev} \overset{\succ}{\underset{\sim}{\searrow}} \begin{matrix} 1_{app} \\ 2_{app} \end{matrix}$$

$$G_{12}: \quad 1_{rev} \overset{\succ}{\searrow} \begin{matrix} 1_{app} \\ 2_{app} \end{matrix} \qquad G_3: \quad \begin{matrix} 1_{app} \xrightarrow{\succ} 1_{app} \\ 2_{app} \xrightarrow{\succsim} 2_{app} \end{matrix}$$

A partir del cierre de $\mathcal{S}$ los multigrafos maximales son:

$$G_1: \quad 1_{rev} \xrightarrow{\succ} 1_{rev} \qquad G_3: \quad \begin{matrix} 1_{app} \xrightarrow{\succ} 1_{app} \\ 2_{app} \xrightarrow{\succsim} 2_{app} \end{matrix}$$

El teorema de terminación a partir de grafos *size-change* adaptado a SRTs [TG05] es como sigue:

Teorema 42 (terminación *size-change*) *Un SRT $\mathcal{R}$ sobre la signatura $\mathcal{F}$ es terminante size-change con respecto al par de reducción $(\succsim, \succ)$ sobre $\mathcal{T}(\mathcal{F}, \mathcal{V})$ sii cada multigrafo maximal contiene al menos un arco de la forma $i \xrightarrow{\succ} i$.*

Los multigrafos maximales en el Ejemplo 6.3 sugieren que el programa del Ejemplo 6.2 es terminante *size-change*; sin embargo, en el análisis de la siguiente sección veremos que no es terminante.

6.3. Cuasi-terminación basada en grafos *size-change*

En la presente sección introducimos una nueva caracterización de SRTs que son cuasi-terminantes basada en grafos *size-change*, junto con un algoritmo de anotación de programas guiado por dicha caracterización.

En el Capítulo 4 afirmamos que, al presentarse sólo un conjunto finito de diferentes términos (módulo renombramiento de variables) en las computaciones, entonces el proceso de evaluación parcial termina. La terminación se consigue gracias a un tipo de memorización la cual está integrada, generalmente, con el proceso de evaluación parcial y que permite evitar la evaluación repetida de los mismos términos. Sin embargo, en el nuevo contexto, esta formulación no es adecuada por lo que presentamos una aproximación más general con respecto a la introducida en el Capítulo 4.

La nueva formulación se basa en la descripción particular de un tipo de derivaciones por *narrowing* necesario en las que se presenta para su cómputo sólo un número finito de diferentes llamadas a función (módulo renombramiento de variables), i.e., redexes. Propiedad a la que llamamos EP-terminación.

El nuevo esquema presenta algunos requerimientos en cuanto a los aspectos de control, mismos que se enuncian más adelante, al igual que la definición formal del nuevo esquema de cuasi-terminación basado en la propiedad de EP-terminación y en los grafos *size-change*.

6.3.1. EP-terminación

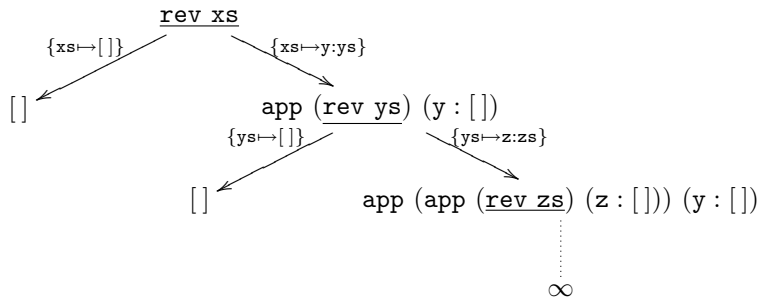
Utilizamos el acrónimo EP para referirnos a la evaluación parcial, y nos referimos a *EP-terminación* para denotar un tipo particular de terminación relacionado con la evaluación parcial:

Definición 43 (EP-terminación, SRT EP-terminante) *Una computación por narrowing necesario posee la propiedad de EP-terminación si el desplegado de funciones genera sólo un número finito de llamadas a función (i.e., redexes) diferentes módulo renombramiento de variables. Un SRT es EP-terminante si cada posible computación por narrowing necesario es EP-terminante.*

Observemos que de este modo un SRT EP-terminante no asegura la cuasi-terminación de sus computaciones pero sí el que sólo sea evaluado un número finito de llamadas a función (redexes) diferentes.

Ejemplo 6.4 *Dado el SRT del Ejemplo 6.2 y la llamada inicial* `rev xs`*, se producen*

las siguientes derivaciones por narrowing necesario:



las cuales contienen un número infinito de términos diferentes pero sólo un número finito de diferentes llamadas a función (módulo renombramiento de variables) a computar.

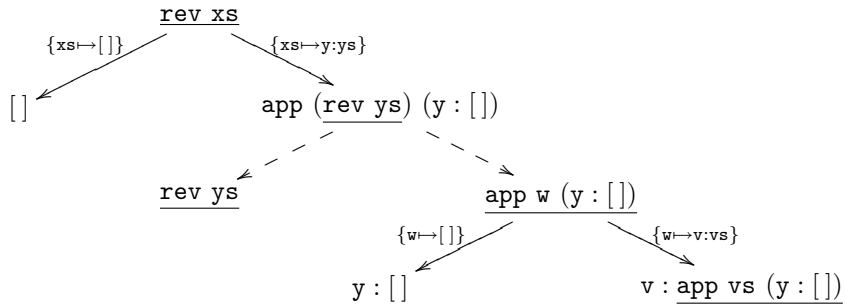
Un aspecto muy positivo es que esta condición es suficiente para asegurar la terminación de nuestro esquema de evaluación parcial si consideremos un procedimiento de especialización con una ligera modificación al introducido en el Capítulo 4. En concreto, requerimos una nueva definición del control local y el control global como se verá a continuación.

6.3.2. Aspectos de control

Para que la propiedad de EP-terminación sea suficientemente explotada y conducir a la terminación del proceso de evaluación parcial se requiere una adecuación de los aspectos de control un tanto diferentes a los introducidos en la Sección 4.4.2.

- *control local*: una derivación de *narrowing* necesario generalizante se detiene cuando la llamada a función seleccionada es una variante de una llamada a función reducida previamente en la misma derivación. Para ello se hace uso de alguna técnica de memorización;
- *control global*: una vez que el despliegue de una función se detiene, los términos no constructores en las hojas del árbol de *narrowing* necesario generalizante se "aplanan" completamente antes de agregarlos al conjunto de llamadas (a ser) evaluadas parcialmente. Por ejemplo, el término `f (g x) (h y)` implica que se agreguen las llamadas a función (`f` $\mathtt{w}_1$ $\mathtt{w}_2$), (`g x`) y (`h y`) al conjunto actual de llamadas (a ser) evaluadas parcialmente, donde $\mathtt{w}_1, \mathtt{w}_2$ son variables frescas. Este paso de "aplanado" es necesario para conseguir la terminación del proceso de evaluación parcial mediante la propiedad EP-terminación.

Ejemplo 6.5 *Considerando nuevamente el Ejemplo 6.2 y la llamada inicial* `rev xs`*, pero teniendo en cuenta los nuevos criterios para terminación del control local y global, las derivaciones por narrowing necesario son las siguientes:*



donde denotamos con $\dashrightarrow$ *la acción del control global. De este modo, las derivaciones del ejemplo son finitas.*

6.3.3. Cuasi-terminación por EP-terminación

En esta sección vamos a definir un tipo de cuasi-terminación con base en la propiedad de la EP-terminación y aprovechando la descripción del comportamiento de los argumentos en las llamadas a función en un programa que proveen los grafos *size-change*.

Antes de describir el esquema, vamos a revisar nuevamente los aspectos de la linealidad. La carencia de linealidad por la derecha, al igual que en el Capítulo 4 trae consigo problemas de terminación.

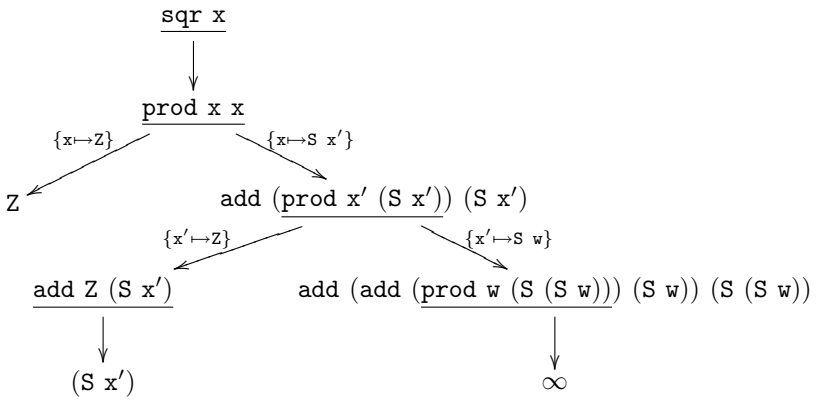
Ejemplo 6.6 *Consideremos el programa* `sqr` *para calcular el cuadrado de un número* `x` *representado con números naturales:*

```
sqr x     =  prod x x

add Z y   =  y
add (S x) y  =  S (add x y)

prod Z y  =  Z
prod (S x) y  =  add (prod x y) y
```

las derivaciones por narrowing necesario para la llamada a función inicial `sqr x` *son:*



Podemos observar que cuando no hay linealidad por la derecha, aparece un número infinito de términos diferentes y un número infinito de llamadas a función diferentes, i.e., la computación no es cuasi-terminante ni tampoco EP-terminante.

Es común que los evaluadores parciales *offline* incluyan un análisis de tiempo de enlace, BTA. Dado un programa $\mathcal{R}$ para especializar y la información acerca de qué datos de la llamada a función inicial son conocidos y cuáles desconocidos, un BTA simple aproxima todos los valores dentro del programa. El BTA devuelve una lista de valores abstractos: *estático* y *dinámico*, asociados a los argumentos de cada función del programa (en la Sección 6.4 documentamos un BTA para programas Curry). Un argumento estático significa que en tiempo de especialización será definitivamente conocido (lo cual implica que es un argumento sin variables) mientras que uno dinámico será posiblemente desconocido.

A continuación requerimos que el componente $\succsim$ del par de reducción $(\succsim, \succ)$ sea de *particionamiento finito* como en [DdSL$^+$97].

Definición 44 (preorden de particionamiento finito) *Sea $\succsim$ un preorden sobre $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Decimos que $\succsim$ es de particionamiento finito si el conjunto $\{s \mid t \succsim s\}$ contiene un número finito de términos no variantes para cualquier término t.*

Intuitivamente, la idea es que $\succsim$ es de particionamiento finito sii no hay infinitos términos "iguales" respecto al orden $\succsim$, lo cual es necesario para garantizar la propiedad de EP-terminación.

El teorema que sigue establece las condiciones suficientes para asegurar la EP-terminación, aprovechando la información reunida por los grafos *size-change* y el BTA.

Teorema 45 *Sea $\mathcal{R}$ un SRT y $(\succsim, \succ)$ un par de reducción. $\mathcal{R}$ es EP-terminante si cada multigrafo maximal asociado a una función $f \in \mathcal{R}$ de aridad n contiene,*

(i) al menos un arco $i_f \xrightarrow{\succ} i_f$ para algún $i \in \{1, \ldots, n\}$ tal que i_f es estático, o bien

(ii) un arco $i_f \xrightarrow{Rel} i_f$ para todo $i = \{1, \ldots, n\}$ donde $Rel \in \{\succ, \succsim\}$ tal que $\succsim$ es de particionamiento finito.

Además, requerimos que $\mathcal{R}$ sea lineal por la derecha para todas las variables dinámicas, i.e., no puede haber repeticiones de la misma variable dinámica en las partes derechas de las reglas de $\mathcal{R}$.

Demostración. El teorema lo vamos a demostrar por contradicción. Supongamos que la computación de un término inicial lineal t_0 con respecto a un programa EP-terminante $\mathcal{R}$ produce una secuencia infinita de términos: $t_0 \leadsto t_1 \leadsto t_2 \leadsto \ldots$ Enseguida, vamos a considerar un reordenamiento de la derivación en el cual los redexes se van a explotar en el orden más a la izquierda y más interno. Además, consideramos que en la derivación reordenada sólo se computan unificadores más generales, restringidos a las variables de $\mathcal{V}ar(s_{i-1})$ (en lugar de unificadores simples, i.e., no se anticipan algunos enlaces como en λ [AEH00, Def. 13]). Esta derivación más interna se denota por $s_0 \leadsto_{p_1,R_1,\sigma_1} s_1 \leadsto_{p_2,R_2,\sigma_2} s_2 \leadsto_{p_3,R_3,\sigma_3} \cdots$

Del mismo modo que [TG05], para dos grafos *size-change* o multigrafos G y H donde los nodos de entrada de G tienen las mismas etiquetas de los nodos de salida de H, con $G \circ H$ expresamos el grafo que resulta de yuxtaponer G y H, i.e., un tipo de concatenación en la que no se borran los nodos de entrada de G y los de salida de H, a diferencia de la concatenación $G \cdot H$ donde sí se borran los nodos intermedios.

Por cada paso $s_j \leadsto s_{j+1}$ de la secuencia infinita de computaciones con $j \in \mathbb{N}$ existe un grafo *size-change* G_j. Además, por cada j tenemos $s_{j+1} = s_j[\theta_j(r_j)]_{p_j}$ donde $\delta_j = \sigma_j \cup \theta_j$ es el unificador más general de $s_j|_{p_j}$ y l_j, con $R_j = (l_j \to r_j)$ y $\sigma_j(s_j|_{p_j}) = \theta_j(l_j)$, i.e., los enlaces en σ_j no necesitan aplicarse en $s_j[\theta_j(r_j)]_{p_j}$.

Toda vez que $\mathcal{R}$ es EP-terminante cada multigrafo maximal de una función $f \in \mathcal{R}$ con aridad n tiene:

(i) un arco $i_f \xrightarrow{\succ} i_f$ para algún $i \in \{1, \ldots, n\}$ o,

(ii) un arco $i_f \xrightarrow{Rel} i_f$ para todo $i = \{1, \ldots, n\}$ donde $Rel \in \{\succ, \succsim\}$.

A partir de dos arcos $i_f \xrightarrow{Rel} j_g$ en G_1 y $j_g \xrightarrow{Rel} k_h$ en G_2 donde $Rel \in \{\succ, \succsim\}$, denotamos con *camino* a la trayectoria $i_f \xrightarrow{Rel} j_g \xrightarrow{Rel} k_h$ formada por los arcos de los grafos *size-change* $G_1 \circ G_2$.

Para el caso (i), de manera equivalente a [TG05, Lema 6], el grafo formado por $G_1 \circ G_2 \ldots$ presenta un camino infinito donde aparecen arcos infinitamente y etiquetados con "$\succ$". Sin pérdida de generalidad asumimos que este camino comienza en G_1. Para cada j, sea a_j el nodo de salida en G_j sobre el que se forma el camino. En un término $f(\overline{t_n})|_{a_j}$ expresa el argumento que forma parte del camino.

Sea $f(\overline{t_m}) \to g(\overline{t'_p})$ la forma de la regla $R_j = l_j \to r_j$ aplicada en una computación $s_j \rightsquigarrow s_{j+1}$, donde $g(\overline{t'_p})$ es una llamada a función en r_j, entonces, existe un grafo *size-change* G_j para R_j con un argumento relacionado del modo que sigue $f_j(\overline{t_m})|_{a_j} \succ g_j(\overline{t'_p})|_{a_{j+1}}$, para toda j en el conjunto infinito $J \subseteq I\!N$ y $f_j(\overline{t_m})|_{a_j} \succsim g_j(\overline{t'_p})|_{a_{j+1}}$ para toda $j \in I\!N \setminus J$. m y p son las aridades de f y g respectivamente.

En $j = 0$, ya que $(s_0|_{p_0})|_{a_0}$ es básico (estático), entonces $(s_0|_{p_0})|_{a_0} = \theta_0(f_0(\overline{t_m})|_{a_0})$, en función de que los órdenes "$\succ$" y "$\succsim$" son cerrados bajo sustitución, tenemos $\theta_0(f_0(\overline{t_m})|_{a_0}) \succ \theta_0(g_0(\overline{t'_p})|_{a_1})$ o bien, $\theta_0(f_0(\overline{t_m})|_{a_0}) \succsim \theta_0(g_0(\overline{t'_p})|_{a_1})$. Ahora bien, en función de que el par de reducción es seguro, se cumple $\mathcal{V}ar(g_0(\overline{t'_p})|_{a_1}) \subseteq \mathcal{V}ar(f_0(\overline{t_m})|_{a_0})$, por lo que el carácter básico de $\theta_0(f_0(\overline{t_m})|_{a_0})$ se cumple también para $\theta_0(g_0(\overline{t'_p})|_{a_1})$. Si aplicamos este razonamiento de manera recursiva tenemos que $s_j|_{a_j} \succ s_{j+1}|_{a_{j+1}}$ para todo $j \in J$ y $s_j|_{a_j} \succsim s_{j+1}|_{a_{j+1}}$ para todo $j \in I\!N \setminus J$, lo que contradice el carácter bien fundado del orden "$\succ$".

Para el caso (ii) asumimos (por [TG05, Lema 6]) que el grafo $G_1 \circ G_2 \ldots$ contiene al menos n caminos infinitos cuyos arcos están etiquetados con "$\succ$" o con "$\succsim$". Sin pérdida de generalidad asumimos que tales caminos comienzan en G_1. Para nuestro análisis consideramos uno de los caminos infinitos. Para cada j-ésima computación, sea G_j su grafo *size-change* y a_j el nodo de salida de G_j que participa en el camino infinito. En un término $f(\overline{t_n})|_{a_j}$ expresa el argumento que forma parte del camino.

Sea $f(\overline{t_m}) \to g(\overline{t'_p})$ la forma de la regla $R_j = l_j \to r_j$ aplicada en la computación, donde $g(\overline{t'_p})$ es una llamada a función en r_j, entonces, existe un grafo *size-change* G_j para R_j con un argumento relacionado del modo que sigue $f_j(\overline{t_m})|_{a_j} Rel\ g_j(\overline{t'_p})|_{a_{j+1}}$ con $Rel \in \{\succ, \succsim\}$, m y p son las aridades de f y g respectivamente.

Para $j = 0$, dado que "$\succ$" y "$\succsim$" son cerrados bajo sustitución tenemos que $\theta_0(f_0(\overline{t_m})|_{a_0})\ Rel\ \theta_0(g_0(\overline{t'_p})|_{a_1})$ con $Rel \in \{\succ, \succsim\}$. El par de reducción utilizado por los grafos *size-change* asegura que las variables en los argumentos $i_f \xrightarrow{Rel} i_g$ no crecen:

- $s \succsim t$ sii $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ y para todo $x \in \mathcal{V}ar(t)$, $dv(t,x) \leqslant dv(s,x)$;
- $s \succ t$ sii $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ y para todo $x \in \mathcal{V}ar(t)$, $dv(t,x) < dv(s,x)$.

Por lo tanto, tenemos que:

$$depth(\theta_0(g_0(\overline{t'_p})|_{a_1})) \leqslant max(depth((s_0|_{p_0})|_{a_0}), depth(g_0(\overline{t'_p})|_{a_1})) + k$$

donde k es un número finito para tomar en cuenta los posibles enlaces que podría traer θ_0. Para $j = 1$ tenemos:

$$depth(\theta_1(g_1(\overline{t'_p})|_{a_2})) \leqslant max(depth((s_1|_{p_1})|_{a_1}), depth(g_1(\overline{t'_p})|_{a_2})) + k$$

Para cualquier j:

$$\begin{aligned} depth(\theta_j(g_j(\overline{t'_p})|_{a_{j+1}})) \leqslant max(depth((s_0|_{p_0})|_{a_0}),\ & depth(g_0(\overline{t'_p})|_{a_1}), \\ & depth(g_1(\overline{t'_p})|_{a_2}), \\ & \ldots, \\ & depth(g_j(\overline{t'_p})|_{a_{j+1}})) + k \end{aligned}$$

Los términos en este camino están acotados por una profundidad finita, por lo que podemos obtener sólo un número finito de términos diferentes por módulo renombramiento de variables, lo que contradice la afirmación inicial. □

6.4. Un análisis BTA para programas Curry

En esta sección presentamos un BTA para el lenguaje lógico funcional Curry (considerando la representación intermedia FlatCurry), el cual se utiliza para proveer información al análisis de programas por grafos *size-change* del presente capítulo.

El BTA para programas FlatCurry computa una *división*, i.e., una clasificación para cada i-ésimo parámetro de cada j-ésima función: x_{i_j}, como estático (S) o dinámico (D), sobre el dominio de valores abstractos: $BindingTime$={S,D} siguiendo [JGS93].

Más adelante mostramos los detalles de nuestro análisis BTA, el cual será monovariante. Un BTA monovariante clasifica a cada argumento de función con un sólo valor abstracto que será vigente en todo el programa, i.e., el argumento será siempre estático o siempre dinámico.

Dada una variable x_i y un *entorno de tiempo de enlace* $\tau = (t_1, \ldots, t_a)$, con $1 \leq i \leq a$, mediante $\tau(x_i)$ obtenemos su valor de tiempo de enlace t_i. El tipo del entorno de tiempo de enlace es $BTEnv = BindingTime^*$.

Definición 46 (división) *Sea $\mathcal{P}$ un programa, una división es una función del tipo $div : FuncName \rightarrow BTEnv$, que a partir del nombre de una función proyecta su entorno de tiempo de enlace.*

$$div = [f_1 \mapsto \tau_1, f_2 \mapsto \tau_2, \ldots, f_n \mapsto \tau_n]$$

Denotamos $div(f_i)$ para obtener τ_i, con $1 \leq i \leq n$ y n es el número de funciones de $\mathcal{P}$.

Tabla 6.1: Operación $\sqcup$ sobre valores *BindingTime*.

Operación	Descripción menos dinámica
$S \sqcup S$	S
$S \sqcup D$	D
$D \sqcup S$	D
$D \sqcup D$	D

El BTA computa una división congruente, i.e., un parámetro se puede clasificar como estático sólo si no existe ninguna llamada a función en la que dicho parámetro es dinámico.

Se impone el orden $S \sqsubseteq D$ sobre el conjunto *BindingTime* y $t \sqsubseteq t'$ sii $t = S$ o $t = t'$. Este orden se puede extender a entornos de tiempo de enlace y divisiones como sigue, dado $\tau = (t_1, ..., t_a)$ y $\tau' = (t'_1, ..., t'_a)$,

$$\tau \sqsubseteq \tau' \text{ sii } t_j \sqsubseteq t'_j \text{ para } j \in \{1, ..., a\}$$

y para divisiones:

$$div_1 \sqsubseteq div_2 \text{ sii } div_1(f_k) \sqsubseteq div_2(f_k)$$

para todas las funciones f_k del programa.

Se introduce ahora el operador de *menor cota superior* $\sqcup$ que devuelve la descripción menos dinámica de dos o más valores *BindingTime*. En la Tabla 6.1 se ilustra su cómputo.

El operador $\sqcup$ se puede aplicar a entornos BTEnv y a divisiones (de forma análoga al orden $\sqsubseteq$). $\tau \sqcup \tau'$ computa el entorno BTEnv más pequeño (menos dinámico) el cual es al menos tan dinámico como τ y como τ'. Por ejemplo, sea $\tau = (S, D, S)$ y $\tau' = (D, D, S)$, entonces $\tau'' = \tau \sqcup \tau' = (D, D, S)$. De aquí τ'' es más dinámico que τ' y que τ, sin embargo es el BTEnv más pequeño (menos dinámico) posible ya que pueden existir otros BTEnv que sean también más dinámicos que τ' y τ, e.g., (D, D, D).

Siguiendo [JGS93], usamos las funciones $\mathcal{B}_e$ y $\mathcal{B}_v$ para calcular la división de un programa.

Definición 47 *Denotamos por $\mathcal{B}_e[[e]]\tau$ una función para calcular el valor de tiempo de enlace de la expresión e en el entorno τ de la función en que se encuentra e. La*

función $\mathcal{B}_e[[e]]\tau$ *se define como sigue:*

$$
\begin{array}{lcl}
\mathcal{B}_e[[e]] : BTEnv & \rightarrow & BindingTime \\
\mathcal{B}_e[[x]]\tau & = & \tau(x) \\
\mathcal{B}_e[[c(e_1,...,e_a)]]\tau & = & \bigsqcup_{i=1}^{a} B_e[[e_i]]\tau \\
\mathcal{B}_e[[f(e_1,...,e_a)]]\tau & = & \bigsqcup_{i=1}^{a} B_e[[e_i]]\tau \\
\mathcal{B}_e[[(f)case\ x\ of\ \{\overline{p_n \rightarrow e_n}\}]]\tau & = & \tau(x) \bigsqcup (\bigsqcup_{i=1}^{n} B_e[[e_i]]\tau[\overline{y_{ik_i} \mapsto \tau(x)}]) \\
\mathcal{B}_e[[e_1 \quad or \quad e_2]]\tau & = & B_e[[e_1]]\tau \bigsqcup \mathcal{B}_e[[e_2]]\tau
\end{array}
$$

En el lenguaje intermedio aparece la estructura $(f)case\ x\ of\ \{\overline{p_n \rightarrow e_n}\}$, en la que cada patrón p_i representa una instancia válida de la variable x. Es por eso, que en cada patrón pueden aparecer k variables nuevas, las cuales heredan el carácter estático o dinámico de x. Por tanto, las variables nuevas se agregan al entorno τ de la función, lo cual expresamos mediante la notación $\tau[y_{ik_i} \mapsto \tau(x)]$ para cada patrón i.

A grandes rasgos, la función $\mathcal{B}_e[[e]]\tau$ sirve para calcular que tan dinámica es una expresión e. $\mathcal{B}_e[[e]]\tau$ devuelve S si la expresión analizada con respecto al entorno de tiempo de enlace τ (de la función en la que está escrito e) es estática y devuelve D si es dinámica. $\mathcal{B}_e[[e]]\tau$ analiza sólo una expresión (e) y es invocada por la función $B_v[[e]]\tau g$.

Definición 48 *Sea* $\mathcal{B}_v[[e]]\tau g$ *una función para calcular el entorno de tiempo de enlace de las llamadas a la función* g *en* e *a partir del entorno de tiempo de enlace* τ *de la función en la que se encuentra* e.

$$
\begin{array}{lcll}
B_v[[e]] : BTEnv \rightarrow FuncName & \rightarrow & BTEnv & \\
B_v[[x]]\tau g & = & (S,...,S) & \\
B_v[[c(e_1,...,e_a)]]\tau g & = & \bigsqcup_{i=1}^{a} B_v[[e_i]]\tau g & \\
B_v[[f(e_1,...,e_a)]]\tau g & = & t \bigsqcup (B_e[[e_1]]\tau,...,B_e[[e_a]]\tau) & \text{si } f = g \\
 & = & t & \text{si } f \neq g \\
 & & \text{donde} \quad t = \bigsqcup_{i=1}^{a} B_v[[e_i]]\tau g & \\
B_v[[(f)\ case\ x\ of \{\overline{p_n \rightarrow e_n}\}]]\tau g & = & \bigsqcup_{i=1}^{n} B_v[[e_i]]\tau_i g & \\
 & & \text{donde} \quad \tau_i = \tau[\overline{y_{ik_i} \mapsto \tau(x)}] & \\
B_v[[e_1 \quad or \quad e_2]]\tau g & = & B_v[[e_1]]\tau g \bigsqcup B_v[[e_2]]\tau g &
\end{array}
$$

Intuitivamente, dada una expresión e, un BTEnv y un nombre de función g, $B_v[[e]]\tau g$ nos dice cuál debería ser el BTEnv de la función g de acuerdo a las llamadas a función que aparecen en e. Para ello, e se ve afectada por el contexto en que se encuentra, es decir por el BTEnv τ de la función en la que está. De hecho para determinar el BTEnv de una función g en un programa, se visitan todos los cuerpos de función, pasando en cada ocasión el τ de la función visitada y el nombre de la función g para la que se computa su BTEnv.

A continuación nos centramos en los aspectos de corrección del BTA, para ello introducimos las siguientes definiciones preliminares.

Definición 49 *Sea $\mathcal{S}$ un conjunto parcialmente ordenado $\sqsubseteq$ formado por las m posibles divisiones diferentes para un programa $\mathcal{R}$, tal que $|\mathcal{S}| = m$. En la división $div_1 \in \mathcal{S}$ todos los argumentos de las funciones del programa son estáticos y en la división $div_m \in \mathcal{S}$ todos son dinámicos.*

Ejemplo 6.7 *Dado un programa formado por dos funciones* `f` *y* `g` *con aridad 2 y 1 respectivamente, tenemos que:*

$$\begin{array}{lll} \mathcal{S} = \{ & [f_1 \mapsto (S,S), & g_1 \mapsto (S)], \\ & [f_2 \mapsto (S,S), & g_2 \mapsto (D)], \\ & [f_3 \mapsto (S,D), & g_3 \mapsto (S)], \\ & [f_4 \mapsto (S,D), & g_4 \mapsto (D)], \\ & [f_5 \mapsto (D,S), & g_5 \mapsto (S)], \\ & [f_6 \mapsto (D,S), & g_6 \mapsto (D)], \\ & [f_7 \mapsto (D,D), & g_7 \mapsto (S)], \\ & [f_8 \mapsto (D,D), & g_8 \mapsto (D)] \ \} \end{array}$$

De este modo, las m posibles divisiones del programa son 8.

Definición 50 *Sea $\mathcal{R}$ un programa con n funciones, τ_1 el entorno inicial para $f_1 \in \mathcal{R}$ y $\delta = \{div_0, \ldots, div_c, div_{c+1}\}$ una secuencia de divisiones, donde $\delta \subseteq \mathcal{S}$ se obtiene por el cálculo iterativo:*

$$div_{c+1}(f_k) = \begin{cases} \tau_1 \sqcup (\bigsqcup_{i=1}^{n} B_v[[e_i]]\ div_c(f_i)\ f_k) & \text{si } k = 1 \\ \bigsqcup_{i=1}^{n} B_v[[e_i]]\ div_c(f_i)\ f_k & \text{en otro caso, para } k = 2, \ldots, n \end{cases}$$

que se computa mientras div_{c+1} sea diferente de div_c.

Aquí, $div_0 = [f_1 \mapsto \tau_1, f_2 \mapsto (S, \ldots, S), f_n \mapsto (S, \ldots, S)]$.

A grandes rasgos, para calcular la división de una función g, se visitan todas las funciones del programa en busca de g, pasando cada vez el BTEnv (el último calculado) de la función que se visita. Para calcular la división de cada una de las k funciones del programa, esto se hace k veces. Finalmente, el proceso se repite hasta que ya no hay cambios en los cómputos de todos los BTEnv de las funciones del programa.

A continuación reproducimos una serie de definiciones comunes en la literatura matemática y que son necesarias para demostrar las propiedades requeridas en el cálculo del BTA.

Definición 51 (cota superior, cota inferior) *$a \in \mathcal{S}$ es una cota superior de un subconjunto δ de $\mathcal{S}$ si $x \sqsubseteq a$, para todo $x \in \delta$. Del mismo modo, $b \in \mathcal{S}$ es una cota inferior de δ si $b \sqsubseteq x$, para todo $x \in \delta$.*

Definición 52 (supremo, ínfimo) *$a \in \mathcal{S}$ es el supremo de un subconjunto δ de $\mathcal{S}$ si a es una cota superior de δ y, para todos las cotas superiores a' de δ, tenemos que $a \sqsubseteq a'$. Así mismo, $b \in \mathcal{S}$ es el ínfimo de un subconjunto δ de $\mathcal{S}$ si b es una cota inferior de δ y, para todos las cotas inferiores b' de δ, tenemos $b' \sqsubseteq b$*

El supremo de δ es único, si existe, y se denota por lub(δ)—del inglés *least upper bound*—. De manera similar, el ínfimo de δ es único, si existe, y se denota por glb(δ)—del inglés *greatest lower bound*—.

Definición 53 (retículo completo) *Un conjunto parcialmente ordenado L es un retículo completo si lub(δ) y glb(δ) existen para cada subconjunto δ de L.*

El conjunto parcialmente ordenado $\mathcal{S}$ de las posibles divisiones de un programa es un retículo completo.

Definición 54 (función monótona) *Sea L un retículo completo y $T : L \to L$ una función. Se dice que T es monótona si $T(x) \sqsubseteq T(y)$ siempre que $x \sqsubseteq y$.*

El cálculo de la función $div_{c+1}(f_k)$ es monótono porque para un $div_c(f_k)$ dado, siempre se cumple $div_c(f_k) \sqsubseteq div_{c+1}(f_k)$ por extensión de $\sqcup$ que cambia sólo en un sentido, i.e., de S a D en los entornos de tiempo de enlace.

Definición 55 (mayor punto fijo) *Sea L un retículo completo y $T : L \to L$ una función. Decimos que $a \in L$ es el mayor punto fijo de T si a es un punto fijo (esto es, $T(a) = a$) y para todos los puntos fijos b de T, tenemos que $b \sqsubseteq a$.*

En [Llo87], se presenta una variante del conocido teorema del punto fijo (debido a Tarski): si L es un retículo completo y $T : L \to L$ es monótona, entonces, T tiene un mayor punto fijo (y un menor punto fijo).

A continuación establecemos un requisito para la congruencia, i.e., un argumento estático sólo puede depender de argumentos estáticos. Para ello, el cálculo de la división monovariante tiene que asegurar que si un argumento de una función f en la división de un programa es estático, entonces ese argumento será estático en todas las llamadas a f que aparecen en cualquier parte del programa [JGS93]. Denotamos con $(\tau)\downarrow_j$ la proyección del j-ésimo valor de tiempo de enlace de τ, e.g., $(S, D, S)\downarrow_2 = D$.

Teorema 56 *Sea $div_{\mathcal{R}}$ una división del programa $\mathcal{R}$. Una división $div_{\mathcal{R}}$ es congruente si se cumple*

$$(Bv[[e_1]]\tau_1 f_i)\downarrow_j \sqcup \ldots \sqcup (Bv[[e_n]]\tau_n f_i)\downarrow_j = S$$

para todo j-ésimo argumento x con $\tau_i(x) = S$. Donde n es el número de funciones de $\mathcal{R}$, $f_i \in \mathcal{R}$ y $div_{\mathcal{R}} = [f_1 \mapsto \tau_1, \ldots, f_i \mapsto \tau_i, \ldots, f_n \mapsto \tau_n]$.

Demostración. Sea $div_{\mathcal{R}}$ la división que se obtiene a partir del cálculo $div_{c+1}(f_k)$. A partir de la Definición 55 tenemos que la función $div_{c+1}(f_k)$ tiene un mayor punto fijo para $\delta \subseteq \mathcal{S}$ sobre el orden $\sqsubseteq$. El punto fijo se alcanza por $div_{c+1}(f_k)$ para todas las k funciones del programa.

Dado un argumento $\tau_i(x) = S$ de una función f_i obtenido por $div_{c+1}(f_k)$ asumamos que $div_{c+2}(f_k)$ cambia el valor de tiempo de enlace de x, i.e., $\tau_i(x) = D$. Esta suposición es opuesta a la definición de punto fijo, por lo tanto la nueva iteración computa $\tau_i(x) = S$. De aquí, la aplicación de Bv en $(Bv[[e_k]]\tau_k f_i)$, con $1 \leq k \leq n$, no cambia el valor de tiempo de enlace S de x, por consecuencia:

$$(Bv[[e_1]]\tau_1 f_i) \downarrow_j \sqcup \ldots \sqcup (Bv[[e_n]]\tau_n f_i) \downarrow_j = S$$

□

6.5. Nuevo esquema de anotación

A continuación introducimos un nuevo esquema de anotación de programas para el evaluador parcial NPE *offline*. Las condiciones del Teorema 56 definen un tipo particular de programas; sin embargo, el nuevo método de anotación permite ampliar el tipo de programas que manipulamos. Para ello, los resultados del teorema referido nos permiten identificar los términos concretos que debemos anotar para, posteriormente, generalizarlos en tiempo de especialización y garantizar así la terminación de la evaluación parcial (propiedad de EP-terminación).

El algoritmo toma cada símbolo de función f de aridad n, donde f tiene un multigrafo maximal. A continuación, se observan las partes derechas de las reglas del programa y se tienen en cuenta las siguientes consideraciones:

1. Si el multigrafo maximal de f contiene algún argumento estático estrictamente decreciente, entonces no se realizan anotaciones.

2. Si el multigrafo maximal de f presenta, para todos los argumentos, un arco etiquetado con $\succ$ o $\succsim$, entonces no se anota ningún argumento.

3. En cualquier otro caso, se anota el j-ésimo argumento de cada llamada a la función f en el programa que carezca de un arco $j_f \xrightarrow{Rel} j_f$, con $Rel \in \{\succ, \succsim\}$.

Finalmente, en la parte derecha de cada regla del programa se anotan las variables dinámicas necesarias para lograr la linealidad por la derecha. Del mismo modo que en el Capítulo 4, extendemos la signatura $\mathcal{F}$ con el símbolo especial $\bullet$ para llevar a cabo las anotaciones.

En la Definición 57 utilizamos $div(f)$ para indicar la lista de valores estáticos y dinámicos de la función f que computa el BTA monovariante y la proyección $\lfloor_i$ para hacer referencia al i–ésimo elemento de la lista. Los detalles de la función div y el BTA los presentamos en la Sección 6.4.

Definición 57 ($annG(\mathcal{R})$) *Sea $\mathcal{R}$ un SRT inductivamente secuencial sobre la signatura $\mathcal{F}$ y $\mathcal{M}$ el conjunto de los grafos size-change maximales de $\mathcal{R}$. El SRT anotado, $annG(\mathcal{R}, \mathcal{M})$, sobre $\mathcal{F}_\bullet$ se computa como sigue:*

$$annG(\mathcal{R}, \mathcal{M}) = lin(\{l \to ann(r, \mathcal{M}) \mid l \to r \in \mathcal{R}\})$$

donde la función auxiliar ann se define inductivamente así:

$$ann(e, \mathcal{M}) = \begin{cases} e & \text{si } e \in \mathcal{V} \\ c(\overline{ann(t_n, \mathcal{M})}) & \text{si } e = c(\overline{t_n}) \text{ con } c \in \mathcal{C} \\ f(\overline{ann(t_n, \mathcal{M})}) & \text{si } e = f(\overline{t_n}) \text{ y } \forall G \in \mathcal{M}, \\ & \exists i \in \{1, \ldots, n\} \\ & \text{tal que } i_f \xrightarrow{\succ} i_f \in \mathcal{M} \\ & \text{y } div(f)\lfloor_i \text{ es estático} \\ f(\overline{ann(t_n, \mathcal{M})}) & \text{si } e = f(\overline{t_n}) \text{ y } \forall G \in \mathcal{M}, \\ & \forall i = 1, \ldots, n \text{ hay un arco} \\ & i_f \xrightarrow{Rel} i_f \in \mathcal{M} \text{ con } Rel \in \{\succ, \succsim\} \\ f(t'_1, \ldots, t'_n) & \text{en otro caso, donde } e = f(\overline{t_n}) \text{ y} \\ & t'_i = \bullet(ann(t_i, \mathcal{M})) \text{ si } \exists G \in \mathcal{M} \text{ tal} \\ & \text{que no hay un arco } i_f \xrightarrow{Rel} i_f \\ & \text{con } Rel \in \{\succ, \succsim\} \\ & \text{y } t'_i = ann(t_i, \mathcal{M}) \text{ en otro caso} \end{cases}$$

Finalmente, la función auxiliar lin anota las variables dinámicas no anotadas de las partes derechas de las reglas excepto una (por ejemplo la que esté en la posición más a la izquierda).

Ejemplo 6.8 *Consideremos a continuación la función* `front`*, la cual se usa para completar un byte agregando ceros al frente de un conjunto de bits dado en el segundo argumento.*

```
front Z x     =  x
front (S y) x =  front y (Z : x)
```

Tabla 6.2: Comparación de métodos de anotación.

		Peor caso (Args. dinámicos)		Mejor caso (Args. estáticos)	
Prueba nombre	T.Código (bytes)	OffPeval anotaciones	SCG anotaciones	OffPeval anotaciones	SCG anotaciones
`ackermann`	273	1	2	1	0
`fibonacci`	177	2	1	2	0
`fun_inter`	1059	9	8	9	0
`reverse`	95	1	0	1	0
`pascal`	395	4	4	4	0

El multigrafo maximal que resulta del programa es el siguiente:

$$\begin{array}{ccc} 1_{front} & \xrightarrow{\succ} & 1_{front} \\ 2_{front} & & 2_{front} \end{array}$$

A continuación se muestra la versión anotada de la función `front`*, teniendo en cuenta que todos sus argumentos son dinámicos.*

$$\begin{array}{rcl} \texttt{front Z x} & = & \texttt{x} \\ \texttt{front (S y) x} & = & \texttt{front y} \bullet (\texttt{Z} : \texttt{x}) \end{array}$$

A partir del esquema de anotaciones de programas mejorado hemos desarrollado un prototipo el cual se encuentra disponible en:

`http://www.dsic.upv.es/~guadalupe/schg`

En la Tabla 6.2 mostramos un resumen de los experimentos. En la columna SCG se muestra el número de anotaciones computadas por el nuevo método basado en grafos *size-change* y la información de valores estático/dinámico de los argumentos reunida por el BTA monovariante (introducido en la Sección 6.4). En general, el nuevo método reduce de manera notable el número de términos anotados en comparación con la técnica original definida en el Capítulo 4, cuyo número de anotaciones aparece en la columna OffPeval.

En la Tabla 6.2 se muestra, por un lado, el número de anotaciones derivadas de aplicar los tests en el *peor caso*, i.e., cuando todos los argumentos son dinámicos. Y segundo, el *mejor caso*, cuando todos los argumentos son estáticos.

En el peor caso, sólo hay un test (función `ackermann`) en el cual el número de anotaciones se incrementa. En el resto de ejemplos de prueba, y en particular cuando los argumentos se hacen estáticos, el número de anotaciones se reduce. El nuevo método es una mejora muy clara con respecto al anterior esquema (Capítulo 4). Esto

se debe a que el método previo no establece ninguna diferencia entre los argumentos cuando son estáticos o dinámicos, i.e., siempre los considera dinámicos (no se hace uso de un BTA).

6.6. Trabajo relacionado

El trabajo que se ha introducido en este capítulo guarda algunas similitudes con el de Jones y Glenstrup [GJ05]. Básicamente [GJ05] formula el criterio conocido como *bounded anchoring* el cual se basa en grafos *size-change*. El principio es útil para detectar parámetros que pueden actuar como un ancla (porque decrecen) para otros parámetros (que pueden crecer) durante la evaluación parcial. De esa manera, los argumentos ancla son suficientes para asegurar la terminación del proceso de evaluación parcial.

Los resultados del Teorema 45 son cercanos a los de [LJBA01] y [GJ05]. La primera afirmación es básicamente la misma que la introducida por [LJBA01] para asegurar la terminación de los programas por grafos *size-change*; para ello, se busca identificar un argumento estático decreciente, i.e., el parámetro ancla de [GJ05].

No obstante, nuestros resultados para asegurar la terminación de la evaluación parcial presentan ciertas diferencias con respecto a los del esquema de [GJ05]. Por ejemplo, requerimos que la parte derecha de la reglas sea lineal en cuanto a las variables dinámicas, mientras que [GJ05] no. La linealidad por la derecha se requiere en nuestro contexto para evitar situaciones de computaciones infinitas como la mostrada en el Ejemplo 6.6, esto se debe a que *narrowing* propaga *instanciaciones*.

En nuestro trabajo utilizamos un *binding time analysis* estándar, separado de la construcción de los grafos a diferencia de [GJ05] que lo hace en un sólo proceso. Además, [GJ05] extiende los grafos *size-change* con la noción de grafos de dos capas, con la finalidad de aproximar la relación "podría crecer" entre los argumentos. Estas diferencias hacen que nuestro análisis sea de menor coste que el de [GJ05], a la vez que logramos la mejoría deseada en la precisión de anotaciones respecto al esquema del Capítulo 4.

6.7. Conclusiones

El nuevo método de anotación definido en el presente capítulo mejora significativamente el esquema introducido en el Capítulo 4. Aún así, la precisión del método de evaluación parcial basada en grafos *size-change* es susceptible de mejorar. Esto es posible mediante una línea de trabajo para afinar la etapa de anotación, lo cual daría lugar a la definición de un esquema de anotación polivariante en comparación con el

actual que es monovariante. Tal extensión es posible a partir del hecho de que si una función f presenta varios multigrafos maximales se considera siempre el peor caso; i.e., si existe un argumento $i_f \in G1$ con arco y otro $i_f \in G2$ sin arco, para efectos de anotación se considera que ningún grafo tiene arco y se aplica la misma anotación a todas las llamadas a f que aparecen en el programa.

Una aproximación para conseguir anotaciones polivariantes es considerar el contexto en el cual una función ocurre, i.e., conservando la secuencia de llamadas que produce el multigrafo maximal. De esta manera el algoritmo de anotación sería más costoso, pero al mismo tiempo más preciso.

Capítulo 7

El lenguaje empotrado de dominio específico Rose

En este capítulo, introducimos el lenguaje empotrado de dominio específico Rose para la especificación de *routers* software. Rose está empotrado en Curry [Han03], un moderno lenguaje declarativo multi-paradigma. Una ventaja de esta aproximación es que se cuenta con un conjunto de técnicas y herramientas ya desarrolladas para programas Curry que eventualmente podrían aplicarse a los programas Rose. Adicionalmente, en este capítulo, mostramos cómo las características de Curry son particularmente útiles para especificar las configuraciones de los *routers* con un alto nivel de abstracción. Los programas escritos en Rose servirán para ilustrar (en el siguiente capítulo) el uso del evaluador parcial *offline* del Capítulo 5.

En [RSV03b, RSV03a, RSV04a, RSV04b] se encuentra publicado parte del material presentado en este capítulo.

7.1. Antecedentes

Para conectar diferentes tecnologías de comunicación de datos en ambientes heterogéneos se requieren ciertos dispositivos especiales. Dentro de las redes de ordenadores e Internet, el *router* es ese dispositivo. Básicamente los *routers* conectan dos o más redes y retransmiten paquetes de datos entre ellos. Así pues, la función primaria de un *router* es determinar la mejor ruta dentro de una red compleja. Originalmente, los *routers* han sido desarrollados por completo como componentes hardware. A principios de la presente década surgió una tendencia encaminada a extender el conjunto de funciones que las redes de *routers* deberían soportar. Tales nuevas funciones inclu-

yen, e.g., filtrado de paquetes, traducción de direcciones de red, ejecución de proxies de red, monitorización del desempeño, etc. La flexibilidad requerida para cubrir con todas las nuevas funciones motivó la definición de los denominados *routers extensibles* [GP02] basados completamente en software. Los *routers* extensibles se caracterizan porque la funcionalidad del *router* se puede personalizar en tiempo de ejecución.

Por un lado los *routers* basados en hardware son rápidos pero, a la vez, difíciles de configurar y administrar. Por otro, los *routers* software, son más ineficientes, pero mucho más flexibles, fáciles de configurar y administrar, más baratos, etc. En este contexto, el desarrollo de *routers* modernos requiere un gran esfuerzo de diseño, construcción y verificación.

Las aproximaciones más importantes acerca de *routers* extensibles son: Scout, *Router Plugins* y Click. En Scout [PKL99], la unidad de abstracción básica es la ruta: un flujo lineal de datos que comienza en un dispositivo de origen y concluye en un dispositivo final de destino. Cada ruta se compone de etapas que son instancias de un módulo específico, las cuales implementan un protocolo bien definido (IP, TCP, etc). Scout provee herramientas para crear, modificar, planificar y controlar rutas. Los *Router Plugins* han sido implementados en el sistema operativo NetBSD y permiten a los usuarios extensiones limitadas a un *router* IP. Tales extensiones se pueden ubicar en ciertos puntos bien conocidos (llamados compuertas) de la ejecución de un *router* IP. Las compuertas han sido escogidas para adecuarse a una amplia variedad de aplicaciones: *routing*, planificación de paquetes y procesamiento de seguridad.

Finalmente, Click [Koh01, KMC+00] se basa en la composición de elementos simples (objetos de clases escritas en C++) para producir un sistema que implemente el comportamiento deseado. Un elemento controla específicamente un aspecto del comportamiento de un *router*, e.g., modificación de los paquetes, administración de colas, políticas de eliminación de paquetes y planificación de paquetes. Además, pueden construirse nuevas configuraciones a partir de la conexión (cada elemento puede tener múltiples puertos para la conexión) de diferentes elementos por medio del uso de un lenguaje simple (llamado además Click).

De acuerdo a [GP02], de las tres arquitecturas mencionadas, Click es la más flexible. Su facilidad para formar virtualmente cualquier configuración a partir del conjunto de elementos da al programador un alto grado de libertad. Así pues, es posible modificar los *routers* de manera incremental y agregar nuevos servicios. Sin embargo, esta gran flexibilidad requiere un gran soporte para asegurar lo que constituye una configuración bien formada. Para este propósito Click ya incluye un conjunto de herramientas que auxilian al usuario a tratar con configuraciones complejas. No obstante, la inclusión de aspectos semánticos en Click sería útil para el programador, e.g., una descripción abstracta de cada elemento, el número de puertos de entrada y puertos de salida, información acerca de cómo modificar los paquetes en tránsito, etc.

En este capítulo introducimos Rose, un lenguaje de dominio específico basado en Click para la especificación de *routers*. Rose está empotrado en Curry [Han03]. Por lo tanto, las configuraciones de los *routers* son objetos de primera clase, lo que permite usar las características de alto nivel de Curry, tales como: combinadores de orden superior, restricciones, un mecanismo de computación perezoso, variables lógicas, etc. Aún más, existe ya un gran número de herramientas para transformar, optimizar y verificar programas Curry con una base teórica sólida que está al alcance del programador.

Sin embargo, debemos clarificar que no pretendemos competir con Click; en lugar de ello, nuestra aspiración es desarrollar una aproximación complementaria. De hecho, no se pretende desarrollar *routers* software en Curry (como Click lo hace en su entorno), sino sólamente su especificación semántica para el modelado. Consecuentemente, los elementos de Rose sólo contienen información de alto nivel que podría ser útil para el análisis, la simulación, verificación, etc. La ventaja principal de nuestra propuesta es que provee una base apropiada para desarrollar análisis específicos y herramientas de optimización (para Click existen muy pocas herramientas relacionadas, inclusive, no han sido verificadas formalmente).

7.2. El Lenguaje Click

El lenguaje de programación Click describe de manera textual la configuración de un *router*. Click tiene sólo dos constructores básicos que son suficientes para describir cualquier configuración (representada de manera visual por un grafo [Koh01]), ellos son:

declaraciones: sirven para crear (instancias de) elementos, y

conexiones: permiten la conexión entre elementos

Para construir la configuración de un *router*, el usuario escoge ciertos elementos de acuerdo a la funcionalidad deseada y los conecta a través de sus puertos, creando así un grafo dirigido que representa el flujo de los paquetes y las operaciones efectuadas en los nodos sobre ellos. Por ejemplo, el grafo de la Figura 7.1 muestra varios elementos conectados formando un *router* simple. El *router* cuenta los paquetes de entrada y entonces los elimina.

El *router* de la Fig. 7.1 se escribe en el lenguaje Click de la siguiente manera:

```
// Declaraciones                    // Conexiones
src  :: FromDevice(eth0);           src -> ctr;
ctr  :: Counter;                    ctr -> sink;
```

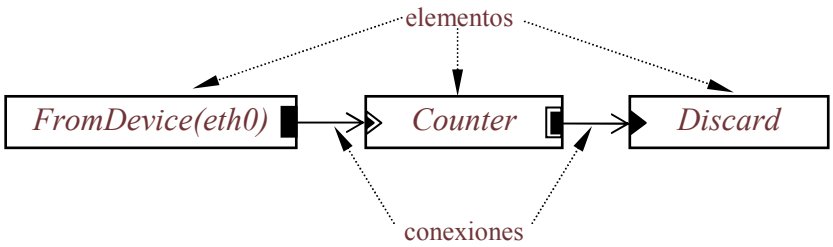


Figura 7.1: Un *router* Click

```
sink :: Discard;
```

En cuanto a los elementos Click [Koh01], éstos se pueden clasificar en las siguientes categorías básicas:

Proveedores de paquetes. Estos elementos generan paquetes espontáneamente, ya sea por la lectura que realizan sobre algún segmento de red, leyéndolos a partir de un fichero, creándolos a partir de ciertos datos especificados o generándolos de manera aleatoria. Estos elementos tienen una salida y ninguna entrada.

Elementos extractores. Eliminan paquetes del sistema (*router*), ya sea por eliminación simple, depositándolos en la red, escribiendo sus contenidos en un fichero o enviándolos a la pila de red de Linux. Estos elementos tienen una entrada y carecen de salidas.

Modificadores de paquetes. Se usan para modificar paquetes de datos. Tienen una entrada y una o más salidas. Los paquetes que llegan por la entrada son modificados y entonces son emitidos por la salida que corresponda; usualmente hay una salida especificada para paquetes indicando errores.

Elementos de enrutamiento. Seleccionan el puerto por el que deberían salir los paquetes basándose en algún algoritmo de conmutación de paquetes, en las características generales del flujo de paquetes o por examinación del contenido de los paquetes. Por ejemplo, un algoritmo de conmutación típico como el "*round robin*" al recibir un paquete que llega por un puerto de entrada lo deposita en un puerto de salida y el siguiente paquete lo deposita en el siguiente puerto de salida, etc. Estos elementos tienen una entrada y dos o más salidas.

Elementos de planificación. Estos elementos seleccionan paquetes a partir de una o varias fuentes de paquetes. Un elemento planificador de paquetes tiene naturalmente dos o más puertos de entrada y uno de salida. Por ejemplo, el planificador podría reaccionar a peticiones de paquetes seleccionando una de sus entradas por turno hasta que alguna produjera algún paquete (extrayendo un paquete de alguna de sus fuentes). Cuando se produce la siguiente petición, intentaría extraer un paquete del siguiente puerto de entrada (el posterior al de la última fuente). Esta sería una operación *round robin*.

7.3. El lenguaje de especificación Rose

El diseño de un nuevo lenguaje de programación implica un esfuerzo considerable en diversos frentes, e.g., en la definición de su sintaxis y semántica, en la implementación de intérpretes y compiladores, en el desarrollo de herramientas para el análisis, depuración, optimización, manipulación de programas, etc. Una aproximación alternativa consiste en construir un lenguaje *empotrado*, i.e., desarrollar librerías en algún lenguaje existente (el lenguaje anfitrión), lo cual permite definir programas en el nuevo lenguaje como objetos del lenguaje anfitrión. En este contexto, todas las herramientas y características del lenguaje anfitrión se podrán usar en el lenguaje empotrado.

Siguiendo esta aproximación hemos desarrollado Rose, un lenguaje para la especificación de *routers* Click empotrado en Curry. Curry tiene propiedades que lo hacen adecuado para el desarrollo de lenguajes de dominio específico[1]. En particular, posee virtudes convenientes para describir *routers* (e.g., un mecanismo de computación perezoso, reglas condicionales con "guardas" y combinadores de orden superior, entre otras); además, permite el uso de ciertas características de tipo lógico (variables lógicas, indeterminismo y búsqueda), que pueden ser útiles para las tareas de simulación y prueba. Las principales ventajas con que contamos en nuestra aproximación y, en particular, por el hecho de empotrar Rose en Curry, son:

Estructuras de datos y recursión. Hacemos uso de las listas de Curry para describir elementos que tienen múltiples entradas. Frecuentemente, tales elementos se definen para cualquier tamaño, una propiedad que se denomina genericidad en los lenguajes de descripción de hardware como VHDL. La recursión es una forma común para definir y manipular convenientemente tales elementos.

Polimorfismo. Algunos elementos deben aceptar como parámetros datos de diferentes tipos, de tal modo que esos datos se pueden reutilizar en varias partes del

[1] De hecho, Curry ya ha sido usado para empotrar otros lenguajes, e.g., un lenguaje para programación distribuida [Han99].

router con diferente información de entrada.

Funciones de orden superior. Las funciones de orden superior son esenciales para construir las configuraciones de un *router*; en particular, para conectar y componer elementos existentes. Los operadores de composición que serán descritos en la Sección 7.3.1 hacen un uso extenso de las funciones de orden superior.

Evaluación perezosa. Curry obedece a un modelo de evaluación perezoso, i.e., las funciones son evaluadas cuando su resultado es demandado. Esta propiedad es particularmente útil cuando se trata con estructuras de datos infinitas (como lo es un flujo de paquetes). Por ejemplo, podemos definir un *router* como una función que genera un flujo infinito de paquetes; sin embargo, si el usuario demanda únicamente la visualización de los primeros 10 paquetes, el resto de paquetes no se generará.

Sistema de tipos. Curry cuenta con un algoritmo de inferencia de tipos estándar que opera durante la compilación. Los errores de tipo suelen ser útiles para detectar errores de compilación en una configuración de un *router*, e.g., para detectar si dos elementos incompatibles han sido conectados.

Características lógicas. Algunas propuestas recientes para la especificación de hardware, particularmente Lava [Cla01] y Hawk [Mat00], están basadas en lenguajes funcionales puros (Haskell). Por otro lado, Curry ofrece características lógicas como funciones indeterministas, variables lógicas, búsqueda encapsulada, etc. Tales características son útiles sobre todo para llevar a cabo simulaciones; por ejemplo, conectando diferentes elementos se tienen flujos indeterministas para los paquetes, los cuales se pueden explorar automáticamente por las computaciones de Curry.

Los elementos de Click son básicamente procesadores de paquetes. Por ello, cada elemento en una configuración de *router* puede abstraerse como una función que recibe un determinado número de flujos de paquetes y devuelve también un cierto número de flujos de paquetes. En concreto, cada puerto de entrada recibe un flujo de paquetes y cada puerto de salida emite un flujo de paquetes. Los paquetes y flujos de paquetes se modelan por medio de la especificación de tipos que sigue:

```
type Packet = [Int]
type Stream = [Packet]
```

Así, un paquete se modela con una lista (finita) de *bytes*, codificados en Rose como enteros, mientras que los flujos de datos (*streams*) se modelan con listas (potencialmente infinitas) de paquetes. Típicamente, los paquetes de datos que manipula un

router como Click, siguen el protocolo de internet IP (*Internet Protocol* [Pos81]); en nuestro caso, representamos la cabecera IP de un paquete de datos con la mencionada lista finita de bytes. En Rose cada elemento Click lo modelamos de la siguiente manera:

```
element :: [Conf] -> [Stream] -> [Stream]
```

i.e., un elemento toma algunos parámetros de configuración (si los hay), una lista de flujos de entrada (que puede ser vacía si se trata de un elemento de la clase de los proveedores de paquetes), y devuelve una lista de flujos de salida (que puede ser vacía si es un elemento de la categoría de elementos de extracción de paquetes). Además, los parámetros de configuración agregan información adicional que define el comportamiento particular de un elemento.

Enseguida vamos a mostrar ejemplos de las cinco categorías de paquetes introducidas en la sección precedente.

Proveedores de paquetes. Estos elementos se codifican como funciones sin flujos de entrada y un sólo flujo de salida. Por ejemplo, el elemento Click `FromDevice(eth)` recoge paquetes de un segmento de red, donde `eth` es un dispositivo de red (una tarjeta de interfaz de red). En Rose lo especificamos del modo siguiente:

```
fromDevice [eth] [] = infiniteSource [eth] []
```

Aquí, la función auxiliar `infiniteSource` es similar a `fromDevice` pero simula el flujo de paquetes fluyendo en un segmento de red. Obviamente no hay diferencia en Rose entre el modelado de los elementos Click `fromDevice` e `infiniteSource`, debido a que no implementamos conexiones a red. Otro elemento Click proveedor de paquetes y modelado en Rose es: `fromDevice_u [parámetros] []`, el cual permite la creación de paquetes con las características indicadas en los `parámetros`.

Elementos extractores. Estos elementos se codifican en Rose por medio de funciones que devuelven sus flujos de entrada sin modificación (contrariamente a Click donde estos elementos no tienen flujos de salida). La razón de este comportamiento es permitir al usuario observar la salida de un *router*, cuestión que puede ser útil durante la fase de especificación para comprobar el comportamiento del *router*. Por ejemplo, el elemento de Click `ToDevice(eth)` envía un flujo de paquetes a un determinado dispositivo físico de red. En Rose, lo codificamos con[2]:

```
toDevice [eth] [ps] = [aux ps]
                      where aux []     = []
```

[2] A continuación no mostraremos el tipo del elemento toda vez que siempre siguen la forma: `[Conf] -> [Stream] -> [Stream]`

```
                                    aux (p:ps) = p : aux ps
```

Aquí, descartamos el dispositivo de red (en la función local `aux`) ya que no estamos interesados en los dispositivos de red a los que los paquetes son enviados. Si se requiriese más información a partir del elemento Click `ToDevice(eth)` podríamos modificar la función para devolver pares de la forma (dispositivo de red, paquete).

`discard [] [ps]` es otro elemento extractor de paquetes de Rose que recibe un flujo `[ps]` de paquetes y los descarta, i.e., los elimina.

Modificadores de paquetes. En Rose estos elementos se modelan por medio de una función que recibe un flujo de paquetes y devuelve uno o dos flujos de salida. Como ejemplo de esta clase de elementos mostramos la especificación del elemento de Click `Strip(n)`, cuya función es borrar los primeros `n` *bytes* de cada paquete (e.g., para retirar la cabecera Ethernet de un paquete). En Rose, `Strip(n)` se codifica como sigue:

```
strip [n] [ps] = [st_ n ps]
                 where st_ m []     = []
                       st_ m (q:qs) = drop m q : st_ m qs
```

donde `drop` es una función de predefinida de Curry (forma parte del Preludio) que devuelve el sufijo de la lista dada sin los primeros n elementos de la lista. Otros elementos de Click que son modelados en Rose son:

`decIPTTL [] [ps]`: Decrementa en uno el contador TTL (*Time To Live*) de tiempo de vida de un paquete de datos. Si el contador llega a 0 elimina el paquete.

`fixIPSrc [Ip dir] [ps]`: Cuando se genera un paquete de error se encarga de escribir la dirección `dir` IP local en él.

`getIPAddress [] [ps]`: Copia la dirección IP destino del paquete en otro campo (de anotación) del paquete para preservar su valor.

`icmpError [Err dir_origen tipo código] [ps]`: Genera nuevos paquetes indicando un error del `tipo` y `código` indicado ocurrido en el ordenador con dirección IP origen `dir_origen`.

`ipFragmenter [I tamaño] [ps]`: Cuando un paquete tiene un tamaño menor al indicado en `tamaño`, se coloca sin cambios; en caso contrario se genera un paquete de error.

`paint [Col c] [ps]`: Coloca la anotación `c` en el campo de anotación correspondiente en el paquete.

`paintTee [Col c] [ps]`: Si el paquete que recibe trae la anotación `c` en el campo correspondiente genera un mensaje de error, en caso contrario devuelve el paquete sin cambios.

Elementos de enrutamiento. Estos elementos tienen un sólo flujo de entrada y dos o más flujos de salida. Por ejemplo, consideremos el elemento de Click: `Classifier(pat`$_1$`,...,pat`$_n$`)` el cual clasifica paquetes de acuerdo a la información contenida en el propio paquete. Para ser más precisos, este elemento toma una secuencia de patrones `pat`$_1$`,...` `,pat`$_n$ (cada patrón contiene uno o más pares que indican un desplazamiento en el paquete y el valor en esa posición), compara cada paquete de entrada contra el conjunto de patrones y coloca el paquete en un puerto de salida dependiendo del primer patrón con el que coincida. Por ejemplo la declaración en Click:

```
Classifier(12/0806 20/0001,
           12/0800,
           -);
```

crea un elemento con tres puertos de salida a utilizar de acuerdo al tipo de paquetes Ethernet que se trate:

1. Solicitudes ARP (protocolo de resolución de direcciones: *Address Resolution Protocol*), estos paquetes tienen en el desplazamiento (*byte*) 12 un valor hexadecimal: 0806, mientras que en el desplazamiento 20 tienen el valor hexadecimal 0001 y serán enviados al primer puerto de salida.

2. Los paquetes IP tienen en el desplazamiento 12 el valor hexadecimal 0800 y son enviados al segundo puerto de salida.

3. En cualquier otro caso de paquetes (denotados por el caso especial "-") se depositan en el tercer puerto.

En Rose, denotamos a los patrones con una lista de expresiones "Pat *cons*", donde *cons* es una lista con un número impar de enteros tales que cada par de enteros *n*,*m* especifica la siguiente condición: el byte *n* del paquete tiene el valor *m*. El último entero de la lista *cons* indica el flujo de salida (el primero es el cero) para los paquetes que cumplen con el patrón. El caso especial "-" (el cual empareja a cualquier paquete) se representa con un patrón con un sólo elemento, i.e., el flujo de salida. De este modo, la cadena de configuración indicada arriba, "12/0806 20/0001, 12/0800, -", se especifica en Rose como sigue:

```
[Pat [12,8,13,6,20,0,21,1,0], Pat [12,8,13,0,1], Pat [2]]
```

Aquí, a cada par que en Click indica desplazamiento/valor lo denotamos con varios pares *byte*/valor; por ejemplo el patrón de Click 12/0800 se representa con la secuencia [12, 8, 13, 0], i.e., en el *byte* 12 debe haber un valor 8, y en el 13 un 0.

La especificación del elemento Click: `Classifier(pat`$_1$`,...,pat`$_n$`)` en Rose es la siguiente:

```
classifier pats [p:ps] = add n p qs
                         where n = class pats p
                               qs = classifier pats [ps]

class (Pat pat:pats) p = let n = class_ pat p
                         in if n == (-1) then class pats p
                                         else n

class_ [n] p = n
class_ (pos:val:es) p = if p!!pos == val then class_ es p
                                         else -1

add n p qs = if n==0 then (p : qs!!0) : tail qs
                     else (qs!!0) : add (n-1) p (tail qs)
```

En general, una llamada de la forma `classifier pats (p:ps)` devuelve una lista de flujos [`qs`$_1$,...,`p:qs`$_n$,...,`qs`$_k$], donde [`qs`$_1$,...,`qs`$_n$,...,`qs`$_k$] es la lista devuelta por `classifier pats ps`. La función `classifier` determina primero el flujo de salida `n` del primer paquete de entrada `p` (usando la función `class`) y entonces agrega (usando la función `add`) el paquete `p` en cabeza del flujo `n` el cual es el resultado de aplicar `classifier` recursivamente al resto de paquetes de entrada `ps`.

Dada una llamada de la forma "`ps!!n`", la función predefinida "!!" devuelve el enésimo elemento de la lista `ps`; por otro lado, una llamada de la forma "`tail ps`" devuelve la cola de `ps`. Otros elementos de enrutamiento de Click modelados en Rose son:

`arpQuerier [Ip dirIPlocal, Eth dirRedLoc, Mapt tabla] [ps]`: Este paquete convierte direcciones de Internet IP en direcciones de red local a partir de los datos del ordenador local y de la `tabla`; si no puede hacerlo, genera un paquete de solicitud a la red en busca de la resolución.

`arpResponder [Mapt tabla] [ps]`: Contesta a una pregunta formulada por medio de un paquete `arpQuerier` mediante la información que tiene en la `tabla`.

`lookupIPRoute [] [ps]`: Es la función más importante del *router*, basándose en la IP de destino determina el puerto adecuado de salida del paquete; propiamente es la acción de enrutamiento.

Elementos de planificación. Estos elementos tienen un flujo de salida y dos o más flujos de entrada. Seleccionan un paquete de entrada a partir de un flujo de entrada (siguiendo alguna política predeterminada) y lo envían a un flujo de salida. Aquí, asumimos que siempre hay paquetes disponibles en cada flujo de entrada. Por ejemplo, la siguiente función en Rose implementa una estrategia trivial "*round-robin*":

```
roundRobin [] ss = [rr ss]
   where rr []     = []
         rr (s:ss) = if s==[] then rr ss
                     else (s!!0) : rr (ss++[tail s])
```

La función `roundRobin` recibe una lista de flujos de entrada y ningún parámetro de configuración. Devuelve el paquete en cabeza del primer flujo de entrada y pasa el resto de paquetes de este flujo a la posición final de la lista de flujos y llama recursivamente al elemento `roundRobin`. En la siguiente llamada tomará un paquete del segundo flujo de entrada y así sucesivamente.

7.3.1. Operadores de composición

Rose ofrece al usuario una librería escrita en Curry que contiene la especificación de los principales elementos de Click. Sin embargo, esto no es suficiente para especificar la configuración de un *router*. El programador requiere métodos sencillos para conectar elementos individuales y así poder construir componentes más grandes. Con ese fin, se incluyen operadores de composición típicos, los cuales son definidos haciendo uso de las facilidades de orden superior del lenguaje Curry.

A continuación mostramos un operador de composición que se usa para combinar cierto número de elementos, de tal forma que la salida de un elemento es la entrada del siguiente. La definición del operador es como sigue:

```
seqOfe :: [[Stream] -> [Stream]] -> [Stream] -> [Stream]
seqOfe []          = id
seqOfe (elem : es) = \input -> seqOfe es (elem input)
```

donde `id` es la función identidad. La función `seqOfe` toma una lista de elementos y devuelve un nuevo elemento cuya entrada se corresponde con la entrada para el primer elemento de la lista y cuya salida es la salida del último elementos de la lista. Por ejemplo, la siguiente función representa la especificación en Rose del *router* de la Figura 7.1

```
simpR = seqOfe [fromDevice [Eth 0], Counter [], Discard []]
```

Otro operador de composición usual es `mult`, que se usa para conectar elementos de enrutamiento (que tienen múltiples flujos de salida) con elementos modificadores de paquetes. La especificación de `mult` es como sigue:

```
mult :: ([Stream] -> [Stream]) -> [[Stream] -> [Stream]]
        -> [Stream] -> [Stream]

mult elem es = \input -> mult_ es (elem input)

mult_ :: [[Stream] -> [Stream]] -> [Stream] -> [Stream]

mult_ es ss = concat (m es ss)
              where m [] []         = []
                    m (e:es) (s:ss) = (e [s]) : m es ss
```

Aquí, `concat` es una función predefinida de Curry para concatenar una lista de listas. La función `mult` toma un elemento con `n` flujos de salida y una lista de `n` elementos, cada uno con un flujo de entrada de tal modo que cada flujo de salida del primer elemento se conecta con un sólo elemento de la lista en el orden de aparición. El conector produce un nuevo elemento cuyo flujo de entrada es la entrada para el primer elemento y cuyos flujos de salida son los flujos de salida de la lista de elementos. Por ejemplo, la función

```
classST = mult (classifier [Pat [12,8,13,6,20,0,21,1,0],
                            Pat [12,8,13,0,1],
                            Pat [2]])
               [strip [I 14],
                checkIPHeader [],
                toDevice [Eth 0]]
```

crea un nuevo componente que toma un flujo de entrada y envía paquetes de solicitud ARP al elemento `strip` (el cual elimina los primeros 14 bytes de un paquete, i.e., la cabecera Ethernet), paquetes IP al elemento `checkIPHeader` (quien verifica que la longitud del paquete es razonable y que la dirección IP de origen sea una dirección legal *unicast*), y el resto de paquetes al elemento de salida de paquetes `toDevice [Eth 0]`.

7.3.2. Operadores de flujo

Los operadores de flujo de Rose permiten recoger el flujo de paquetes a partir de un puerto de salida de un elemento y depositarlo en un puerto de entrada de otro elemento. La lista de flujos de paquetes presenta en la posición 0 la lista de paquetes para el puerto 0, en la posición 1 la lista de paquetes del puerto 1, y así sucesivamente. El operador más simple, `->-`, conecta un elemento con un puerto de salida a un segundo elemento con un puerto de entrada:

```
(->-)::([Stream]->[Stream]) ->
```

```
       ([Stream]->[Stream]) -> [Stream] -> [Stream]
elem1 ->- elem2 =\inp -> let med = elem1 inp in elem2 med
```

A continuación, en el ejemplo siguiente:

```
import rose
EthHdr = [1,1,1,1,1,1,9,9,9,9,9,9]
TTLfld = 200
IPsrc  = [154,250,159,2]
IPtg   = [148,208,179,3]
fd     = fromDevice_u [I 2, Eh EthHdr, Ip IPsrc,
                       Ip IPtg, Ttl TTLfld]
st     = strip [I 14]
d      = discard []
router = fd ->- st ->- d
```

primero se crean dos paquetes mediante el elemento `fd` del tipo `fromDevice_u` con la dirección de red `EthHdr`, la dirección IP origen `IPsrc`, la dirección IP destino `IPtg` y el campo de tiempo de vida TTL con el valor `TTLfld`. Enseguida se crea un elemento (`st`) a partir de `strip` para quitar 14 *bytes* al paquete (en la práctica esto convierte un paquete de red local en un paquete de Internet) y otro del tipo `discard` (`d`) que los elimina. Finalmente, se conectan por medio del operador de flujo `->-`.

El operador `n` a 1: `=>-` conecta el enésimo puerto de salida de un elemento con un segundo elemento con un puerto de entrada.

```
(=>-)::(([Stream]->[Stream]), Int )->
       ( [Stream]->[Stream]) -> [Stream]->[Stream]
(elem1, sp) =>- elem2 = \inp ->let mid = elem1 inp
                                in elem2 [mid !! sp]
```

A continuación, podemos ver un ejemplo del uso del nuevo operador:

```
import rose
EthHdr = [1,1,1,1,1,1,9,9,9,9,9,9]
TTLfld = 0
IPsrc  = [154,250,159,2]
IPtg   = [148,208,179,3]
IPerrs = [148,208,179,1]
fd     = fromDevice_u [I 3, Eh EthHdr, Ip IPsrc,
                       Ip IPtg, Ttl TTLfld]
st     = strip [I 14]
dec    = decIPTTL []
d      = discard []
```

```
router = fd ->- st ->- end
end    = (dec,0)=>- d
end    = (dec,1)=>- icmpError  [Err IPerrs 5 0 ] ->- d
```

En esta configuración de *router* se declaran cuatro elementos, uno creado a partir de `fromDevice_u` (`fd`) que genera tres paquetes con los datos de configuración indicados (en particular el tiempo de vida del paquete será 0). Un segundo elemento del tipo `strip` (`st`) elimina 14 *bytes*; otro de tipo `decIPTTL` (`dec`) que verifica el tiempo de vida del paquete y un último a partir de `discard` (`d`). En el ejemplo, se conecta `fd` con `st` y la salida se entrega a la función indeterminista `end` donde, en el primer caso, se conecta el flujo del puerto 0 con `d`, mientras que en el segundo caso se generará un paquete de error con `icmpError` que finalmente se descarta. Este ejemplo produce paquetes con el tiempo de vida consumido y, al pasar por la revisión de `dec`, genera paquetes de error.

Por último, el operador `m` a `n`: `=>=` enlaza dos elementos a partir del puerto de salida m del primer elemento al puerto n de entrada del segundo elemento.

```
(=>=)::(([Stream]->[Stream]), Int) ->
       (Int,([Stream]->[Stream]) ) -> [Stream] -> [Stream]
(elem1, sp) =>= (tp, elem2) =
    \inp ->let mid = elem1 inp
           in elem2 (inst tp (mid !! sp) [[[ ]]])
```

el cual es útil en la representación del *router* Click IPv4 [Koh01].

7.3.3. Implementación

La librería implementada (Rose) en Curry para la especificación de *routers* incluye los siguientes contenidos:

- Un subconjunto de elementos Click, en particular aquellos necesarios para implementar el *router* IPv4 [Koh01] de Click.
- Operadores de composición y flujo para conectar elementos.
- Funciones para iniciar la prueba (simulación) de las configuraciones de un *router*.

Existen dos funciones de prueba, la primera para probar un *router* que genera un flujo finito de paquetes:

```
simulfr router = router []
```

Y la segunda que opera con un *router* con un flujo infinito de paquetes.

```
simuln router n port =  take n ( router [] !! port)
```

De esta manera, el usuario puede obtener los primeros `n` paquetes enviados al puerto de salida por la especificación del *router*, evitando así un ciclo infinito si el flujo de entrada es infinito. La función predefinida `take` devuelve los primeros `n` elementos de la lista dada.

Adicionalmente, la librería contiene algunas funciones auxiliares, e.g.,

```
isBroadCastIP []    = False
isBroadCastIP (h:t) = if h == 255 then True
                                  else isBroadCastIP t
isUnivIP []    = True
isUnivIP (h:t) = if h /= 0 then False
                           else isUnivIP t
```

para verificar si una dirección IP es una dirección de difusión (contiene algún 255), o si es la dirección IP universal (0.0.0.0). Además también incluye funciones para manipular paquetes de datos, tales como la aplicación de una función a una posición dada de un paquete. Con la finalidad de aplicar los conceptos mencionados se ha escrito la especificación del *router* Click completo [Koh01] para paquetes IPv4 en Rose disponible en `www.dsic.upv.es/~guadalupe`.

7.4. Trabajo relacionado

Hasta donde sabemos, no hay aproximaciones previas para la especificación de configuraciones de *router* basadas en un lenguaje declarativo bien establecido. Encontramos, sin embargo, aproximaciones similares para el diseño y verificación de componentes hardware (Claessen [Cla01] y Matthews [Mat00]) desarrolladas como lenguajes empotrados en el lenguaje funcional perezoso Haskell. En ese sentido, nuestra aproximación se inspira en los trabajos de Claessen [Cla01] y Matthews [Mat00] quienes definieron un lenguaje empotrado en lugar de un lenguaje nuevo. En general, esta aproximación simplifica enormemente el desarrollo de nuevos lenguajes de dominio específico. No obstante, nuestro trabajo presenta algunos aspectos distintivos:

- El lenguaje anfitrión es diferente. Mientras [Cla01, Mat00] usaron Haskell, nosotros empotramos Rose en el lenguaje multi-paradigma Curry, el cual extiende las características funcionales de Haskell con características logicas y restricciones concurrentes. Estas características pueden ser útiles en futuros desarrollos para realizar tareas como: simulación, verificación, optimización, etc.

- Los sistemas especificados son diferentes. Mientras [Cla01, Mat00] especifican circuitos hardware, nuestra aproximación tiene que ver con configuraciones de

router. Aunque comparten algunas similitudes, la especificación de *routers* tienen algunas particularidades que no están presentes en la especificación de circuitos hardware.

7.5. Conclusiones

Hemos introducido el lenguaje de dominio específico Rose para la descripción de configuraciones de *routers*. Rose está empotrado en el lenguaje declarativo multiparadigma Curry, lo que significa que podemos hacer uso de todas las herramientas para análisis, optimización, manipulación de programas, etc, disponibles para Curry. Por ejemplo, podemos usar el evaluador parcial dirigido por *narrowing* para Curry [RSV05a] para especializar configuraciones genéricas de un *router*, un sistema de transformación de plegado/desplegado [AFMV04] para manipular configuraciones de *router* de modo que se preserve la semántica de la transformación o, inclusive, usar las facilidades para hacer trazas de programas en el ambiente de desarrollo PAKCS [HeAE+04] con la finalidad de depurar una especificación de un *router*. Hemos mostrado como los elementos básicos de Click se especifican en Rose y como los operadores de composición se usan para construir configuraciones mayores. De esta forma, Rose permite al usuario especificar configuraciones de *router* de manera concisa, modular y reusable, mientras que conserva su naturaleza declarativa.

Este trabajo ha dado lugar a toda una serie de temas interesantes para realizar extensiones futuras. Por un lado, podría extenderse Rose para cubrir todos los elementos Click existentes. Además, sería interesante implementar traductores de Click hacia Rose y viceversa. Esto facilitaría la interacción con Click y así Rose podría considerarse como una herramienta práctica para el diseño y verificación de especificaciones de *routers* Click, i.e., de *routers* reales.

Sin embargo, el interés particular de Rose en el contexto de esta tesis tiene que ver con el hecho de que los lenguajes de dominio específico empotrados son excelentes instrumentos de prueba para las herramientas de evaluación parcial [Hud98]. Así, se espera que la evaluación parcial de los programas escritos en un lenguaje de dominio específico empotrado puedan optimizarse con respecto a sus librerías. De este modo, probaremos en el siguiente capítulo el desarrollo del evaluador parcial *offline* [RSV05a] especializando ejemplos escritos en Rose.

Capítulo 8

Aplicación del evaluador parcial

En este capítulo presentamos una aplicación de la evaluación parcial, en la que especializamos una librería de un lenguaje empotrado de dominio específico con respecto a un programa de aplicación. En particular escribiremos programas de prueba escritos en la librería Rose. Este proceso es propiamente la especialización de interpretes y es conocido en la literatura de evaluación parcial como compilación por especialización. En [RAS06] se ha publicado parte del material que aquí se expone.

8.1. Antecedentes

El entorno de desarrollo del lenguaje Curry, PAKCS cuenta actualmente con un evaluador parcial desarrollado a partir del esquema NPE *online* de evaluación parcial dirigido por *narrowing*. Con la finalidad de especializar programas más grandes, se ha formulado el nuevo esquema NPE el cual opera de manera *offline* (Capítulos 4, 5 y 6).

Al diseñar un lenguaje de dominio específico (DSL, *Domain Specific Language*) el objetivo principal es abstraer el dominio del problema para facilitar la generación rápida de aplicaciones, por lo que los aspectos relacionados con la eficiencia del DSL son secundarios. Hudak [Hud98] propuso el uso de la evaluación parcial para mejorar la eficiencia de las aplicaciones escritas en un DSL. La propuesta plantea la compilación por evaluación parcial [Fut71], i.e., especializar un interprete con respecto a un programa y, de ese modo, reducir la sobrecarga de interpretación.

El lenguaje de dominio específico que utilizamos en este capítulo para llevar a cabo los experimentos es Rose (que se introduce en el Capítulo 7). Rose ha sido escrito siguiendo la aproximación de lenguaje de dominio específico empotrado (DSEL,

Domain Specific Embedded Language). El anfitrión de Rose es el lenguaje lógico funcional Curry. Curry presenta características bondadosas para el modelado del dominio gracias, entre otras cosas, a las facilidades que ofrece en cuanto a estructuras de datos infinitas y el manejo de orden superior, lo que permite modelar de una manera muy flexible los conceptos del dominio como objetos de primer orden, i.e., funciones como datos, flujos de paquetes de datos de Internet infinitos, etc.

A continuación introducimos el contexto de compilación por evaluación parcial, explicamos el tratamiento que se da en nuestro esquema al orden superior, después mostramos los resultados de los experimentos efectuados y, finalmente discutimos el trabajo relacionado y concluimos.

8.2. Compilación por evaluación parcial

A continuación introducimos la técnica de compilación por evaluación parcial postulada en [Fut71].

Dado un programa `p` escrito en el lenguaje `L`, el resultado de la ejecución de `p` con algún dato de entrada `d` se denota como: $[[\texttt{p}]]_{\texttt{L}}\ \texttt{d} = \texttt{resultado}$.

En la ecuación 8.1 la ejecución del programa `p` (escrito en el lenguaje `S`) con sus datos de entrada es equivalente a la ejecución del *intérprete* `int` (escrito en un lenguaje `L`) con un programa y sus datos como entrada.

$$[[\texttt{p}]]_{\texttt{S}}\ [\texttt{d}] \ = \ [[\texttt{int}]]_{\texttt{L}}\ [\texttt{p},\texttt{d}] \quad \text{def. ecuacional de intérprete} \tag{8.1}$$

La ecuación 8.2 define el evaluador parcial (`mix` escrito en un lenguaje `L`). La especialización del programa `p` con un dato de entrada `d1`, y la ejecución del resultado de la especialización con `d2`, es equivalente a la ejecución de `p` (escrito en el lenguaje `L`) con todos sus datos de entrada. Hay que notar que la parte derecha de la ecuación 8.2 involucra dos ejecuciones de programa. Primero, se genera un programa residual $\texttt{r} = [[\texttt{mix}]]_{\texttt{L}}\ [\texttt{p},\texttt{d1}]$, y segundo, el programa residual `r` se ejecuta con el dato `d2`.

$$[[\texttt{p}]]_{\texttt{L}}\ [\texttt{d1},\texttt{d2}] = [[\ \ [[\texttt{mix}]]_{\texttt{L}}\ [\texttt{p},\texttt{d1}]\ \ ]]_{\texttt{L}}\ [\texttt{d2}] \quad \text{def. ecuacional de } \texttt{mix} \tag{8.2}$$

A continuación se muestra como compilar programas usando un intérprete y el evaluador parcial de programas. Este resultado se conoce como la *primera proyección de Futamura* [Fut71].

Sea `int` un `S`-intérprete escrito en un lenguaje `L`, `s` un programa escrito en un

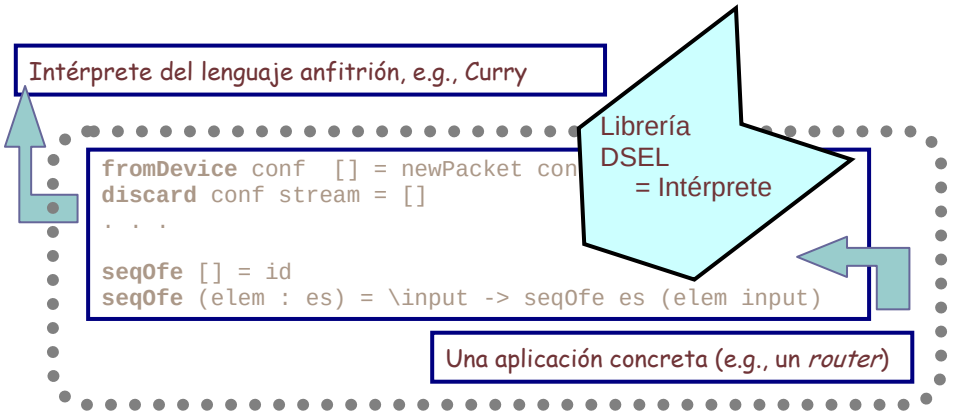


Figura 8.1: Las capas de interpretación de un DSEL.

lenguaje `S`, y `d` sus datos de entrada. La ecuación se demuestra por:

$$\begin{aligned}
[[\texttt{s}]]_{\texttt{S}}\ \texttt{d} &= [[\texttt{int}]]_{\texttt{L}}\ [\texttt{s},\texttt{d}] && \text{por ec. 8.1}\\
&= [[\ (\ [[\texttt{mix}]]_{\texttt{L}}\ [\texttt{int},\texttt{s}]\ )\ ]]_{\texttt{L}}\ \texttt{d} && \text{por ec. 8.2}\\
&= [[\texttt{objeto}]]_{\texttt{L}}\ \texttt{d} && \text{renombrando el programa}\\
& && \text{residual como: } \texttt{objeto}
\end{aligned}$$

La acción de un compilador es propiamente la reducción de un programa a un código intermedio para su ejecución con los datos de entrada. En las ecuaciones mostradas a partir de un intérprete y un programa, se obtiene código intermedio, i.e., $[[\texttt{mix}]]_{\texttt{L}}\ [\texttt{p},\texttt{d1}] = \texttt{objeto}$, producto de la especialización del intérprete para un programa.

A continuación, ilustramos el esquema de compilación aplicado a nuestro caso. En la Figura 8.1 se muestra la secuencia de las capas de interpretación de un DSEL. Primero, la librería DSEL interpreta el programa fuente, e.g., un *router* escrito en Rose. El resultado de la primer capa de interpretación es tomado por el intérprete del lenguaje anfitrión quien lleva a cabo la ejecución final del código. Estas capas se pueden eliminar si aplicamos la técnica de compilación por evaluación parcial, expresada por las siguientes ecuaciones:

$$\begin{aligned}
[[\texttt{router}]]_{\texttt{dsel}}\ [\texttt{d}] &= [[\texttt{dsel_int}]]_{\texttt{Curry}}\ [\texttt{router},\texttt{d}]\\
&= [[\ \ [[\texttt{mix}]]_{\texttt{Curry}}\ [\texttt{dsel_int},\texttt{router}]\ \ ]]_{\texttt{Curry}}\ [\texttt{d}]\\
&= [[\texttt{router_compilado}]]_{\texttt{Curry}}\ \texttt{d}
\end{aligned}$$

De este modo, el *router* compilado queda preparado para su ejecución en Curry, que es más eficiente que compilar el DSEL y el programa sin evaluar parcialmente.

$$
\begin{array}{lcl}
\mathcal{R} & ::= & \mathcal{D}_1 \ldots \mathcal{D}_m \\
D & ::= & f(x_1, \ldots, x_n) = e \\
e & ::= & x \\
 & | & c(e_1, \ldots, e_n) \\
 & | & f(e_1, \ldots, e_n) \\
 & | & case \quad e_0 \quad of \quad \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\} \\
 & | & fcase \quad e_0 \quad of \quad \{p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n\} \\
 & | & partcall(f, e_1, \ldots, e_k) \\
 & | & apply(e_1, e_2) \\
p & ::= & c(x_1, \ldots, x_n)
\end{array}
$$

Figura 8.2: Representación abstracta del lenguaje intermedio extendida para orden superior.

8.3. El orden superior y NPE

El orden superior es una de las características más importantes de un lenguaje declarativo para permitir el desarrollo de DSELs.

En esta sección describimos la representación del orden superior en el lenguaje intermedio de Curry (FlatCurry) y su tratamiento en el esquema NPE. Con ese fin, en la Figura 8.2 mostramos la representación abstracta del lenguaje intermedio (la representación plana) extendida para manipular funciones de orden superior.

El orden superior se implementa mediante la introducción de la función de primer orden: *apply*, para ello se agrega el siguiente axioma [Han03]:

$$[[apply(f(e_1, \ldots, e_m), e)]] \Rightarrow f(e_1, \ldots, e_m, e)$$

si f tiene una aridad n tal que $n > m$. De este modo, una aplicación de función representada por *apply* es evaluada agregando su segundo argumento a la llamada a la función parcial f. En FlatCurry se distinguen las funciones totales de las funciones parciales, que se representan por: *partcall*.

Ejemplo 8.1 *Consideremos el siguiente programa que hace uso de la función de or-*

den superior `map`*:*

```
f x           =  (S x)

main lst      =   map f lst

map h []      =  []
map h (x : xs)  =  (h x) : (map h xs)
```

La representación abstracta de este programa es:

```
f(x) = S(x)

main(lst) = map(partcall(f), lst)

map(x, y)  = fcase x of
      { []      → [];
        (w : ws) → apply(x, w) : map(x, ws)  }
```

La función `main` *invoca a la función* `map`*, que tiene como primer argumento una llamada a función parcial (*`partcall(f)`*), por tanto, en la implementación de* `map` *aparece la estructura* `apply` *que lleva a cabo la aplicación del segundo argumento* `w` *a la función parcial contenida en* `x`*.*

En el esquema NPE *online* [AHV01] las computaciones de evaluación parcial aplican las siguientes reglas:

$$[[apply(e_1, e_2)]] \Rightarrow \begin{cases} [[f(\overline{c_k}, e_2)]] & \text{si } e_1 = partcall(f, \overline{c_k}), \\ & \quad k+1 = aridad(f) \\ partcall(f, \overline{c_k}, e_2) & \text{si } e_1 = partcall(f, \overline{c_k}), \\ & \quad k+1 < aridad(f) \\ try\_eval(apply(e_1, e_2), \{1\}) & \text{otherwise} \end{cases}$$

A grandes rasgos, una función parcial se convierte en una función total agregando el argumento que falta (cuando es posible hacerlo). Si después de agregar un argumento la función no tiene un número de argumentos igual a su aridad, entonces se mantiene como función parcial. En el resto de casos se evalúan los argumentos del *apply* mediante la función *try_eval* en espera de poder llevar a cabo un aplicación a una función parcial después de la evaluación. El segundo argumento de la función *try_eval* es 1 para indicar que el primer argumento de *apply* es estricto, i.e., requiere una forma particular para poder computar el *apply*, en este caso la forma *partcall*.

En el esquema NPE *offline* presentado en el Capítulo 4 se introduce una función explícita *apply* dentro del programa fuente para representar el orden superior, convirtiéndolo así en un programa de primer orden.

Ejemplo 8.2 *Consideremos el siguiente ejemplo introducido en el Capítulo 4 para manipular un programa que hace uso de orden superior:*

$$\begin{aligned}
\texttt{minc x} &= \texttt{map inc}_0\ \texttt{x} \\
\texttt{map f []} &= \texttt{[]} \\
\texttt{map f (x : xs)} &= \texttt{(apply f x) : (map f xs)} \\
\texttt{inc x} &= \texttt{S x} \\
\texttt{apply inc}_0\ \texttt{x} &= \texttt{inc x}
\end{aligned}$$

la función principal `minc` *invoca a la función* `map` *con el argumento constructor* $\texttt{inc}_0$*, que se usa después en la invocación de la función* `apply`*.*

Sin embargo, esta no es una solución conveniente porque implica que cada vez que se use `map` con una función diferente requeriremos crear una nueva regla para `apply` que incluya el constructor relacionado a la nueva función a aplicar.

8.4. Compilación

En esta sección presentamos ejemplos de la compilación por evaluación parcial. Especializamos por un lado el DSEL Rose con respecto a un *router* y por otro, un intérprete simple con respecto a un programa para ese intérprete.

8.4.1. Compilando Rose

Uno de los aspectos fundamentales en la evaluación parcial de programas realistas tiene que ver con el tratamiento del orden superior. En [ST96] optan por hacer una transformación a partir de los programas escritos en orden superior a una representación de primer orden. De esta manera, el evaluador parcial se convierte en un núcleo básico, muy eficiente, que opera sobre programas de primer orden. En nuestro caso, seguiremos esta aproximación; para ello, planteamos primero una transformación de los programas (*routers*) de orden superior a primer orden y después calcularemos la correspondiente anotación (de acuerdo al esquema *offline*). Finalmente, los programas serán especializados con el núcleo básico NPE *offline* de primer orden introducido en el Capítulo 5.

A continuación se introduce un método para llevar a cabo la transformación, sus pasos están inspirados en las reglas de inferencia de [AHV02] para resolver orden

superior en el esquema NPE *online*. Los pasos son:

$$[[apply(e_1, e_2)]] \Rightarrow \begin{cases} [[apply(e_1, e_2'')]] & \text{si } e_2 = apply(e_1', e_2') \text{ y } e_2'' = [[e_2]] \\ [[f(\overline{c_k}, e_2)]] & \text{si } e_1 = partcall(f, \overline{c_k}), \\ & \quad \text{y } k+1 = aridad(f) \\ [[partcall(f, \overline{c_k}, e_2)]] & \text{si } e_1 = partcall(f, \overline{c_k}), \\ & \quad \text{y } k+1 < aridad(f) \\ [[apply(\sigma(r), e_2)]] & \text{si } e_1 = f(\overline{t_n}),\ f(\overline{x_n}) = r \in \mathcal{R} \\ & \quad \text{y } \sigma = \{\overline{x_n \mapsto t_n}\} \\ [[apply(\sigma(t_i'), e_2)]] & \text{si } e_1 = (f) case\ c(\overline{t_n})\ of \\ & \qquad \{\overline{p_k \to t_k'}\} \\ & \quad p_i = c(\overline{x_n}) \text{ y } \sigma = \{\overline{x_n \mapsto t_n}\} \end{cases}$$

$$[[f(\overline{e_n})]] \Rightarrow \begin{cases} [[\sigma(r)]] & \text{si } \exists\ e_i \text{ tal que } e_i = partcall(f, \overline{c_k}), \\ & \quad f(\overline{x_n}) = r \in \mathcal{R} \text{ y } \sigma = \{\overline{x_n \mapsto e_n}\} \\ f(\overline{e_n}) & \text{en cualquier otro caso} \end{cases}$$

El objetivo de estos pasos no es realizar una propagación de datos estáticos por todo el programa (incluyendo funciones de la librería del DSEL), ya que ésto implicaría una pasada de evaluación parcial con un especializador que soporte orden superior. Por tanto, la meta es transformar un *router* i.e., una expresión de orden superior a primer orden. La transformación es posible porque los datos necesarios para llevar a cabo la transformación son conocidos, i.e., un *router* se compone siempre de elementos (funciones) conocidos, puesto que son necesarios para formarlo.

Ejemplo 8.3 *Consideremos el router que se muestra a continuación:*

$$\texttt{router} = ((\texttt{fromDevice Eth1}) -> - (\texttt{discard Void})) \quad [\,]$$

que se forma a partir de dos elementos enlazados por el conector de orden superior $- > -$*, el cual se implementa del modo que sigue:*

$$\texttt{elem1} -> - \texttt{elem2} = \backslash\texttt{inp} \ -> \ \texttt{elem2}\ (\texttt{elem1 inp})$$

El código en lenguaje intermedio de este programa es:

$$\begin{aligned} \texttt{router} = \texttt{apply}(- > - \ (\ & \texttt{partcall}(\texttt{fromDevice}, \texttt{Eth1}), \\ & \texttt{partcall}(\texttt{discard}, \texttt{Void})\ ) \\ , \ [\,]) & \end{aligned}$$

$$- > -(\texttt{x}, \texttt{y}) = \texttt{partcall}(\texttt{lambda}, \texttt{y}, \texttt{x})$$

$$\texttt{lambda}(\texttt{u}, \texttt{v}, \texttt{w}) = \texttt{apply}(\texttt{u}, \texttt{apply}(\texttt{v}, \texttt{w}))$$

donde las funciones locales son globalizadas por "lambda-lifting". Los pasos de traducción del router *se muestran a continuación, el* router *es:*

```
router = apply(− > − ( partcall(fromDevice, Eth1),
                       partcall(discard, Void) )
                , [])
```

El primer argumento de `apply` *es una llamada a función, por lo tanto lo desplegamos:*

```
router = apply( partcall( lambda, partcall(discard, Void),
                          partcall(fromDevice, Eth1) )
                , [])
```

Hay una llamada a función parcial y un argumento, por lo tanto se reduce el `apply`*:*

```
router = lambda( partcall(discard, Void),
                 partcall(fromDevice, Eth1), [])
```

Aún hay llamadas parciales por lo que desplegamos la función `lambda`

```
router = apply( partcall(discard, Void),
                apply( partcall(fromDevice, Eth1), [] ) )
```

Aparece un `apply` *interno que puede reducirse:*

```
router = apply( partcall(discard, Void), fromDevice(Eth1, []) )
```

Finalmente, se reduce el último `apply`*:*

```
router = discard( Void, fromDevice( Eth1, []) )
```

El *router* se puede evaluar parcialmente habida cuenta de que se ha transformado a una expresión de primer orden.

Ejemplo 8.4 *Consideremos el router:*

```
rHO x = seqOfe [ (fromDevice x), (paint (Col 99)), (strip Void),
                 (toDevice x) ]  []
```

El router hace uso de elementos de la librería Rose a los que les hace falta un argumento. La transformación a una expresión de primer orden (ya anotada) se muestra a continuación:

```
rFO x = toDevice (GEN x) (strip Void (paint (Col 99) (fromDevice x [])))
```

Tabla 8.1: Tiempos de ejecución del Ejemplo 8.4 (milisegundos).

Programa	NPE *online*	mejora1 (*online*)	NPE *offline*		mejora2 (*offline*)
			anot	mix	
Router	3163	1.093	414	1168	1.560

La anotación `GEN` *se introduce para obtener un programa no creciente (Capítulo 5).*

El router consta de cuatro elementos. Básicamente, genera paquetes de red local emulando el elemento `fromDevice`*; después, a cada paquete se le coloca una etiquete de identificación:* `99`*. A continuación, el paquete de red local se convierte en paquete de Internet (acción del elemento* `strip`*) y finalmente se emula su colocación en el segmento de red local con el elemento* `toDevice`*.*

El código intermedio especializado del Ejemplo 8.4 es el siguiente:

```
rF0 v0 = toDevice_pe1 v0 v0
toDevice_pe1 v2 v0=FCase v0 of
{  Eth0  -> (: (FCase v2 of
   { Eth0 ->(: (: 4 (: 5... (: 99 ([] )))) toDevice2_pe3)
     Eth1 ->(: (: 4 (: 5... (: 99 ([] )))) toDevice2_pe4)
    })  [] )
   Eth1  -> (: (FCase v2 of
   { Eth0 ->(: (: 4 (: 5... (: 99 ([] )))) toDevice2_pe6)
     Eth1 ->(: (: 4 (: 5... (: 99 ([] )))) toDevice2_pe7)
   })  [])
}
toDevice2_pe3 =(: (: 4 (: 5... (: 99 ([])))) toDevice2_pe3)
toDevice2_pe4 =(: (: 4 (: 5... (: 99 ([])))) toDevice2_pe4)
toDevice2_pe6 =(: (: 4 (: 5... (: 99 ([])))) toDevice2_pe6)
toDevice2_pe7 =(: (: 4 (: 5... (: 99 ([])))) toDevice2_pe7)
```

En el código mostrado las listas se denotan en orden prefijo. Un paquete es simulado por una lista de enteros que tiene en cabeza los valores `(: 4 (: 5...` Básicamente, el código del *router* se ha mezclado con el código de la librería de elementos, lo que corresponde propiamente al proceso de compilación por evaluación parcial, i.e., el código del intérprete (DSEL Rose) y del programa (*router*) son reducidos a un programa objeto (compilación) en código intermedio. El lenguaje intermedio es ejecutable en el entorno de PAKCS [HeAE+04].

En la Tabla 8.1 mostramos las mejoras en los tiempos de ejecución para el *router* del Ejemplo 8.4. Las mejoras se computan por el cociente del tiempo de ejecución (en

milisegundos) absoluto del *router* de orden superior entre el *router* transformado a primer orden y especializado.

En la tabla observamos por un lado que el tiempo de especialización de la herramienta *offline* (NPE *offline*), aún considerando la anotación, es menor con respecto al tiempo de especialización de la herramienta *online* (NPE *online*). Esto constituye una de las propiedades de la aproximación *offline* indicadas en el Capítulo 4. Por otro lado, es interesante observar que la precisión de la herramienta *offline* fue, para este ejemplo, mayor con respecto a la herramienta *online*, ésto no ocurre siempre, ya que en general, la herramienta de especialización *online* es más precisa que la *offline*.

El test para el *router* del Ejemplo 8.4 consistió en requerir 100.000 paquetes de datos al *router*. Un punto a destacar es que las aplicaciones que están sujetas a carga masivas de datos durante la ejecución, el caso del *router*, son especialmente susceptibles de mejora. De este modo, la mejora se comporta en proporción directa al número de paquetes procesados.

8.4.2. Un intérprete funcional simple

El ejemplo que sigue está basado en una adaptación del intérprete funcional simple introducido en [LV05], para evaluar operaciones aritméticas (e.g., adición y multiplicación) sobre constantes o variables de enteros. En nuestro caso las operaciones aritméticas se realizan sobre números naturales.

```
data Nat    =  Z | S Nat | E
data Token  =  Cst Nat   | Var Nat  |  Plus Token Token
             | Minus Token Token    |  Mult  Token Token
```

El intérprete está definido por la función `int`:

```
int (Cst   x  ) vars vals =  x
int (Var   x  ) vars vals =  lookup  x  vars  vals
int (Plus  x y) vars vals =  add    (GEN (int x vars vals))
                                    (GEN (int y vars vals))
int (Minus x y) vars vals =  minus  (GEN (int x vars vals))
                                    (GEN (int y vars vals))
int (Mult  x y) vars vals =  mult   (GEN (int x vars vals))
                                    (GEN (int y vars vals))
```

La función `lookup`, busca el valor asociado a una variable.

```
lookup  x    []    (v:vs) = E
lookup  x (nv:nvs)    []  = E
lookup  x (nv:nvs) (v:vs) = ifte (GEN (eq x nv))  v
                                 (GEN (lookup x nvs vs))
```

Las funciones para llevar a cabo comparaciones entre números naturales, las condicionales y las operaciones aritméticas son las siguientes:

```
eq  Z     Z    = True
eq  Z    (S y) = False
eq (S x)  Z    = False
eq (S x) (S y) = eq x y

ifte True  x  y = x
ifte False x  y = y

add Z      y = y
add (S x)  y = S(add x y)

minus     x    Z  =  x
minus  (S x) (S y)= minus x y

mult Z     _  = Z
mult (S x) y = add y  (GEN (mult x y))
```

La llamada inicial para especializar este programa es:

```
main x = int (Plus (Cst (S (S (S Z)))) (Cst x)) [] []
```

Especializaremos el intérprete con respecto al programa de aplicación escrito en la función principal `main`. En `main` se pretende interpretar un programa que lleva a cabo la adición de la constante `(S (S (S Z)))` con una constante desconocida (dinámica). La evaluación parcial del intérprete sin anotaciones generaría un proceso de computaciones parciales infinito. Las anotaciones para garantizar el carácter finito del proceso se materializan en las funciones `int`, `ifte` y `mult`.

El código objeto del programa especializado es:

```
main v0 = add_pe1 int_pe2 (int_pe3 v0)

add_pe1 Z v5 = v5
add_pe1 (S v304) v5 = S (add_pe1 v304 v5)

int_pe2 = S (S (S Z))

int_pe3 v0 = v0
```

El código se reduce a una llamada a la función `add_pe1` para sumar el dato estático (valor tres: `(S (S (S Z)))`) con un dato dinámico (`v0`), las instrucciones del programa

de aplicación (`Cst`) han sido interpretadas. Del mismo modo, el código extraído a partir del intérprete es solo el necesario para realizar la operación expresada en el programa de aplicación. Por la drástica reducción de código, es claro que el tiempo de ejecución del programa especializado es menor que el tiempo requerido para ejecutar el intérprete original.

8.5. Trabajo relacionado y conclusiones

Algunas otras aproximaciones para la optimización del cógido de un DSEL son las siguientes: Backhouse [Bac02] aporta una metodología basada en *interpretación abstracta* (Cousot y Cousot [CC77]) para analizar los programas de dominio específico, con el objeto de llevar a cabo el chequeo de errores y la revisión de condiciones asociadas y poder ejecutar optimizaciones de domino específico. Christensen [Chr03] sugiere el uso de evaluación parcial de programas para mejorar el rendimiento de los programas escritos en lenguajes de dominio específico empotrados. Herman y Meunier [HM04] realizan un análisis estático de programas empotrados mediante una aproximación "ligera" a la evaluación parcial (usando macros del lenguaje anfitrión) con la finalidad de encontrar errores a nivel del DSEL imposibles de detectar en el nivel de código anfitrión.

El principal objetivo de nuestra investigación ha sido la evaluación parcial; sin embargo, su aplicación a problemas realistas es una de las mayores motivaciones, la razón es que a partir de ello surgen nuevos retos técnicos. Uno de tales retos tiene que ver con asegurar la terminación del proceso de especialización; por ello hemos introducido la aproximación *offline* de evaluación parcial misma que ha sido puesta a prueba en este capítulo.

Los casos de prueba están inspirados en la compilación por evaluación parcial; así, hemos usado un dominio de aplicación importante, como son los *routers* Click. En particular, notamos que la especialización de DSELs es aceptable con la herramienta *offline* introducida en esta tesis.

En el caso de los *routers* la mejora es en proporción directa a la carga de paquetes procesados; por tanto, resulta trascendente para la operación del software descrito considerar esta técnica de optimización de programas. Además, la aproximación *offline* es conveniente para programas grandes.

Capítulo 9

Conclusiones y líneas de trabajo futuro

A continuación, resumimos las aportaciones de la tesis y hacemos mención de las distintas líneas de trabajo futuro.

9.1. Contribuciones

La instancia más reciente (precedente a esta tesis) del esquema de evaluación parcial dirigido por *narrowing* [AFV98], sigue la aproximación *online* [AHV02], el cual considera una variante del teorema de Kruskal (*Kruskal's Tree Theorem*) llamada *subsumción homeomórfica* [Leu02] para asegurar la terminación del proceso: si un término subsume a algún término previo en la misma computación de *narrowing*, se aplica alguna forma de generalización—usualmente el operador de generalización más específica— y la evaluación parcial se reinicia con los términos generalizados [AFV98]. Sin embargo, esta precisión adicional tiene un coste asociado: las comprobaciones para el test de subsumción homeomórfica, junto con las generalizaciones asociadas, hacen que el esquema NPE *online* sea muy costoso, por lo que no se adapta adecuadamente a problemas realistas como la especialización de intérpretes [Jon04].

Por tal motivo hemos introducido una nueva aproximación *offline* que procede en dos etapas; la primera etapa procesa un programa Curry e incluye anotaciones para guiar las computaciones parciales (i.e., para identificar aquellas llamadas a función que pueden desplegarse sin riesgo de no terminación); entonces, en una segunda etapa, la de evaluación parcial propiamente dicha, sólo se obedece las anotaciones y por tanto el especializador es mucho más rápido que en la aproximación *online*.

A continuación resumimos brevemente las principales aportaciones:

1. Una caracterización de programas cuasi-terminantes. Hemos identificado las propiedades para la cuasi-terminación de las computaciones por *narrowing* necesario. A los programas que cumplen las propiedades les llamamos "no crecientes". Así, las computaciones por *narrowing necesario* presentan una secuencia finita de llamadas a función diferentes (módulo renombramiento de variables) en programas no crecientes. La propiedad de cuasi-terminación en las computaciones de los programas es muy útil para el análisis y transformación de los mismos.

2. Un esquema *offline* NPE. En virtud de que la cuasi-terminación de los programas es con frecuecia una condición suficiente para la terminación del proceso de especialización, formulamos un esquema de evaluación parcial dirigido por *narrowing* más eficiente y que asegura la terminación siguiendo el estilo *offline*. Con ese fin, partimos de la caracterización de programas cuasi-terminantes. Puesto que la clase de programas es muy restrictiva, definimos un algoritmo que acepta programas *inductivamente secuenciales* y devuelve un programa con anotaciones en las expresiones que violan la propiedad de programa no creciente. A continuación, llevamos a cabo una ligera extensión al mecanismo de *narrowing* necesario, al que llamamos *narrowing* generalizante. La extensión generaliza las anotaciones en los programas durante la especialización y asegura la terminación.

3. Adaptación del principio de terminación por grafos *size-change*. Llevamos a cabo una adaptación del principio de terminación de los programas funcionales por grafos *size-change* [LJBA01, GJ05] a programas lógico funcionales. La adaptación nos permitió mejorar la precisión de la fase de anotación del esquema *offline* de NPE. En particular los grafos nos fueron útiles para determinar una forma de cuasi-terminación que finalmente utilizamos para realizar anotaciones monovariantes en los programas.

4. Desarrollo de herramientas. Hemos desarrollado un prototipo para la evaluación parcial *offline* dirigida por *narrowing*. La validación experimental del prototipo arrojó resultados que mostraron una mejora significativa en el tiempo de especialización *offline* con respecto al *online* (aunque se pierde cierta precisión).

5. Lenguajes de domino específico empotrado. Desarrollamos un lenguaje de dominio específico: *Rose*, empotrado en Curry para especificar *routers* siguiendo el modelo de *routers software Click* [KMC+00]. El objetivo de la implementación del DSEL fue poner a prueba el nuevo evaluador parcial *offline* mediante

la aplicación del principio de compilación por especialización [Fut71]. La aplicación de la evaluación parcial a un DSEL y un programa de aplicación escrito en el DSEL permitió eliminar la sobrecarga de interpretación presente en los programas desarrollados en lenguajes de dominio específico.

De este modo hemos presentado los principios para el nuevo esquema de evaluación parcial *offline* dirigido por *narrowing* y los hemos puesto a prueba mediante la compilación por evaluación parcial.

9.2. Trabajo futuro

En cuanto a las líneas de trabajo futuro es conveniente mencionar:

1. Mejora del esquema de anotación por grafos *size-change*. En la literatura, se ha aplicado el principio de los grafos *size-change* para conducir anotaciones de los programas; sin embargo, sólo se ha considerado un esquema de anotaciones monovariante. Esto es, una función que aparece en varias expresiones de un programa será anotada del mismo modo en todas las ocurrencias. Por tanto una aproximación polivariante permitiría llevar a cabo anotaciones dependiendo del contexto donde ocurre la función y, de este modo, es posible mejorar la precisión del algoritmo de anotación. Otro aspecto interesante sería considerar los *pesos* [AK03] en el crecimiento o decrecimiento de los argumentos y de ese modo permitir argumentos que crecen temporalmente para luego decrecer.

   Sin embargo, la precisión de la anotación y la consiguiente evaluación parcial podría mejorarse si se considera que una función en diferentes puntos de un programa puede tener diferentes argumentos estáticos y dinámicos (polivariancia). Una calificación polivariante de los argumentos podría evitar que se generalizaran algunas expresiones durante la evaluación parcial.

   De este modo un esquema de anotación polivariante se puede lograr desde dos aproximaciones, primero a partir de la información de los grafos *size-change* y segundo por la clasificación estándar de los argumentos como estáticos y dinámicos.

2. Extensión del esquema *offline* NPE. El esquema actual de evaluación parcial *offline* dirigido por *narrowing* no soporta las características extendidas del lenguaje Curry, e.g., restricciones, funciones predefinidas, guardas, etc. Una extensión natural, de cara a la aplicación de la evaluación parcial en cualquier DSEL es la incorporación de tales características.

3. Una aproximación NPE híbrida. Hasta ahora, las aproximaciones del esquema de evaluación parcial dirigido por *narrowing* han sido *online* u *offline*; sin embargo, la resolución de algunas características, e.g., las funciones *built-in* `(f (x + 1) y)`, se podrían resolver de manera más conveniente de modo *online* que de modo *offline*. Por ejemplo, conocer el valor de `x` permitiría la reducción de la expresión `(x + 1)` de manera *online*. El uso de una aproximación híbrida puede incrementar la precisión del proceso de especialización, ya que, por un lado el análisis *offline* anota expresiones que potencialmente son no terminantes; entonces un análisis *online* podría convertir una expresión (con base en los datos *online*) generalizable en una expresión computable con seguridad. Sin embargo, es claro que el coste de especialización sería mayor.

Bibliografía

[AAF+98a] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In *Proc. of the Int'l Static Analysis Symposium, SAS'98*, pages 262–277. Springer LNCS 1503, 1998.

[AAF+98b] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Polygenetic Partial Evaluation of Lazy Functional Logic Programs. In *Proc. of Appia-Gulp-ProDe, AGP'98*, pages 151–164, 1998.

[AAHV99a] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. of LPAR'99*, pages 376–395. Springer LNAI 1705, 1999.

[AAHV99b] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. Partial Evaluation of Residuating Functional Logic Programs. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 376–395, 1999.

[AAV00] E. Albert, S. Antoy, and G. Vidal. A Formal Approach to Reasoning about the Effectiveness of Partial Evaluation. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000*, 2000.

[AAV01] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042, 2001.

[AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[AFJV97] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN*

Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.

[AFMV99] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 147–162. Springer-Verlag, 1999.

[AFMV00] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. An Automatic Composition Algorithm for Functional Logic Programs. In V. Hlaváč, K. G. Jeffery, and J. Wiedermann, editors, *Proc. of the 27th Annual Conference on Current Trends in Theory and Practice of Informatics, SOFSEM'2000*, pages 289–297. Springer LNCS 1963, 2000.

[AFMV04] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science, Elsevier.*, 311(1-3):479–525, 2004.

[AFV96] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. Riis Nielson, editor, *Proc. of the 6th European Symp. on Programming (ESOP'96)*, pages 45–61. Springer LNCS 1058, 1996.

[AFV98] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.

[AH00] S. Antoy and M. Hanus. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. of the Int'l Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.

[AHLV99] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs. In *Proc. of the Fourth ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'99)*, volume 34.9 of *ACM Sigplan Notices*, pages 273–283. ACM Press, 1999.

[AHLV05] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *Theory and Practice of Logic Programming*, 5(3):273–303, 2005.

[AHV00a] E. Albert, M. Hanus, and G. Vidal. Realistic Program Specialization in a Multi-Paradigm Language. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000*, pages 104–119, 2000.

[AHV00b] E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of the 7th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'00)*, pages 381–398. Springer LNAI 1955, 2000.

[AHV01] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. In *Proc. of 5th Int'l Symp. on Functional and Logic Programming (FLOPS'01)*, pages 326–342. Springer LNCS 2024, 2001.

[AHV02] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[AHV03] E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters*, 85(1):19–25, 2003.

[AK03] H. Anderson and S.C. Khoo. Affine-Based Size-Change Termination. In Atsushi Ohori, editor, *Proceedings of the First Asian Symposium on Programming Languages and Systems, APLAS 2003*, pages 122–140. Springer LNCS 2895, 2003.

[Ant92] S. Antoy. Definitional Trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.

[ARSV06] G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Proc. of the 16th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 55–61. Università Ca' Foscari di Venezia, 2006.

[AV02] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.

[Bac02] K. Backhouse. *Abstract interpretation of domain-specific embedded languages*. PhD thesis, Computing Laboratory, University of Oxford, 2002.

[BD77] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BE86] D. Bert and R. Echahed. Design and Implementation of a Generic, Logic and Functional Programming Language. In *Proc. of First European Symp. on Programming (ESOP'86)*, pages 119–132. Springer LNCS 213, 1986.

[Ben86] J. Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[BKW95] A. Bockmayr, S. Krischer, and A. Werner. Narrowing strategies for arbitrary canonical systems. *Fundamenta Informaticae*, 24(1,2):125 – 155, 1995.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153. IEEE, 2–5 1998.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Bol93] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[Bon89] A. Bondorf. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In *Proc. of Int'l Conf. on Theory and Practice of Software Development, Barcelona, Spain*, pages 81–95. Springer LNCS 352, 1989.

[BSM92] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[CD89] C. Consel and O. Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30:79–86, 1989.

[CD93] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

[Chr03] N.H. Christensen. *Domain-specific languages in software development and the relation to partial evaluation.* PhD thesis, DIKU, Dept. of Computer Science, University of Copenhagen, July 2003.

[CK96] W.N. Chin and S.C. Khoo. Better Consumers for Program Specializations. *Journal of Functional and Logic Programming*, 1996(4), 1996.

[Cla01] K. Claessen. *Embedded Languages for Describing and Verifying Hardware.* PhD thesis, Chalmers University of Technology and Göteborg University, 2001.

[CM87] W.F. Cloksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1987.

[DdSL+97] S. Decorte, D. de Schreye, M. Leuschel, B. Martens, and K.F. Sagonas. Termination Analysis for Tabled Logic Programming. In *Proc. of the 7th Int'l Workshop on Logic Programming Synthesis and Transformation (LOPSTR'97)*, pages 111–127. Springer LNCS 1463, 1997.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

[DG89] J. Darlington and Y. Guo. Narrowing and Unification in Functional Programming: an Evaluation Mechanism for Absolute Set Abstraction. In *Proc. of the Conf. on Rewrite Techniques and Applications, (RTA'89)*, pages 92–108. Springer LNCS 355, 1989.

[DGJ+99] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.

[DJ90] N. Dershowitz and J.P. Jouannaud. Rewrite Systems. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

[Ech90] R. Echahed. On completeness of narrowing strategies. *Theoretical Computer Science*, 72(2-3):133–146, 1990.

[EFM00] C. Elliott, S. Finne, and O. De Moor. Compiling Embedded Languages. In *Proc. of the International Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG '00)*, pages 9–27. Springer LNCS 1924, 2000.

[EH97] C. Elliott and P. Hudak. Functional Reactive Animation. In *Proc. of the 2nd. ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP '97)*, pages 263–273. ACM Press, 1997.

[EO87] A. P. Ershov and B.N. Ostrovski. Controlled Mixed Computation and its Application to Systematic Development of Language-Oriented Parsers. In *The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation*, pages 31–48. North-Holland Publishing Co., 1987.

[Fri85] L. Fribourg. SLOG: a Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 172–185. IEEE, New York, 1985.

[Fut71] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[GJ05] A. J. Glenstrup and N.D. Jones. Termination Analysis and Specialization-Point Insertion in Offline Partial Evaluation. *ACM TOPLAS*, 27(6):1147–1215, 2005.

[GJMS96] R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In *Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'96*, pages 152–166. Springer LNCS 1140, 1996.

[GK93] R. Glück and A.V. Klimov. Occam's Razor in Metacomputation: the Notion of a Perfect Process Tree. In P. Cousot, M. Falaschi, G. Filè, and A. Rauzy, editors, *Proc. of 3rd Int'l Workshop on Static Analysis, WSA'93*, pages 112–123. Springer LNCS 724, 1993.

[GLMP91] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.

[GM86] J. Goguen and J. Meseguer. Eqlog: Equality, Types and Generic Modules for Logic Programming. In D. de Groot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.

[GP02] Y. Gottlieb and L. Peterson. A Comparative Study of Extensible Routers. In *2002 IEEE Open Architectures and Network Programming Proceedings*, pages 51–62, New York, NY USA, June 2002.

[GS96] R. Glück and M.H. Sørensen. A Roadmap to Metacomputation by Supercompilation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 137–160. Springer LNCS 1110, February 1996.

[Han90] M. Hanus. Compiling Logic Programs with Equality. In *Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han97] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, New York, 1997.

[Han99] M. Hanus. Distributed Programming in a Multi-Paradigm Declarative Language. In *Proc. of the Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[Han00a] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *Proc. of the Int'l Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[Han00b] M. Hanus. Server Side Web Scripting in Curry. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000*, pages 366–381, 2000.

[Han03] M. Hanus. Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~mh/curry/`, 2003.

[HeAE+04] M. Hanus (ed.), S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.6.0: The Portland Aachen Kiel Curry System—User Manual. Technical report, University of Kiel, Germany, 2004.

[HH03] L. Huang and P. Hudak. Dance: A Declarative Language for the Control of Humanoid Robots. Technical Report YALEU/DCS/RR-1253, Yale University, August 2003.

[HL92] G. Huet and J.J. Lévy. Computations in Orthogonal Rewriting Systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.

[HLM98] M. Hanus, S. Lucas, and A. Middeldorp. Strongly Sequential and Inductively Sequential Term Rewriting Systems. *Information Processing Letters*, 67(1):1–8, 1998.

[HLW92] W. Hans, R. Loogen, and S. Winkler. On the Interaction of Lazy Evaluation and Backtracking. In M. Bruynooghe and M. Wirsing, editors, *Proc. of the 4th Int'l Symposium on Programming Language Implementation and Logic Programming (PLILP '92)*, pages 355–369. Springer-Verlag, 1992.

[HM04] D. Herman and P. Meunier. Improving the Static Analysis of Embedded Languages via Partial Evaluation. In *Proc. of the 9th ACM SIGPLAN International Conference on Functional Programming (ICFP '04)*, pages 16–27. ACM Press, 2004.

[HMGW96] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore Music Notation - an Algebra of Music. *Journal of Functional Programming*, 6(3):465–483, 1996.

[Hol91] C.K. Holst. Finiteness Analysis. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 473–495. Springer LNCS 523, 1991.

[HP99] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[Hud98] P. Hudak. Modular Domain Specific Languages and Tools. In *Proc. of the 5th Int'l Conf. on Software Reuse (ICSR '98)*, page 134. IEEE Computer Society, 1998.

[Hud00] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, New York, NY, USA, 2000.

[Hul80] J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[IN97] T. Ida and K. Nakahara. Leftmost Outside-in Narrowing Calculi. *Journal of Functional Programming*, 7(2):129–161, 1997.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[JMP06] P. Julián, G. Moreno, and J. Penabad. Operational/Interpretive Unfolding of Multi-adjoint Logic Programs. *Journal of Universal Computer Science*, 12(11):1679–1699, 2006.

[Jon94] N.D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. In N.D. Jones, M. Hagiya, and M. Sato, editors, *Logic, Language and Computation*, pages 206–224. Springer LNCS 792, 1994.

[Jon04] N.D. Jones. Transformation by Interpreter Specialisation. *Science of Computer Programming*, 52:307–339, 2004.

[Jul00] P. Julián. *Especialización de Programas Lógico-Funcionales Perezosos*. PhD thesis, DSIC, Universidad Politécnica de Valencia, 2000.

[Klo92] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[KM91] J.W. Klop and A. Middeldorp. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195, 1991.

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[KNT99] K. Kusakari, M. Nakamura, and Y. Toyama. Argument Filtering Transformation. In *Proc. of the Int'l Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, pages 47–61, 1999.

[Koh01] E. Kohler. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, February 2001.

[Laf98] L. Lafave. *A Constraint-Based Partial Evaluator for Functional Logic Programs and its Application*. PhD thesis, University of Bristol, 1998.

[Lan66] P. J. Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

[LDdW96] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.

[Leu02] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, pages 379–403. Springer LNCS 2566, 2002.

[LFSH99] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.

[LG97] L. Lafave and J.P. Gallagher. Partial Evaluation of Functional Logic Programs in Rewriting-based Languages. Technical Report CSTR-97-001, Department of Computer Science, University of Bristol, Bristol, England, March 1997.

[LJBA01] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. In *ACM Symposium on Principles of Programming Languages (POPL'01)*, volume 28, pages 81–92. ACM press, 2001.

[LK99] W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[Llo94] J.W. Lloyd. Combining Functional and Logic Programming Languages. In *Proc. of the International Logic Programming Symposium*, pages 43–57, 1994.

[Llo95] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.

[LLR93] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of 5th Int'l Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, pages 184–200. Springer LNCS 714, 1993.

[LR64] L.A. Lombardi and B. Raphael. Lisp as the Language for an Incremental Computer. In E.C. Berkeley and D.G. Bobrow, editors, *The Program-*

ming Language Lisp: Its Operation and Applications, pages 204–219. The MIT Press, Cambridge, MA, 1964.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[LV05] M. Leuschel and G. Vidal. Forward Slicing by Conjunctive Partial Deduction and Argument Filtering. In *Proc. of the European Symposium on Programming (ESOP 2005)*, pages 61–76. Springer LNCS 3444, 2005.

[Mat00] J.R. Matthews. *Algebraic Specification and Verification of Processor Microarchitectures*. PhD thesis, University of Washington, 2000.

[MG95] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. 12th Int'l Conf. on Logic Programming (ICLP'95)*, pages 597–611. MIT Press, 1995.

[MH94] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[MO98] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. *Journal of Symbolic Computation*, 25(6):733–757, 1998.

[Mor00] G. Moreno. *Rules and Strategies for Transforming Functional Logic Programs*. PhD thesis, DSIC, Universidad Politécnica de Valencia, 2000.

[MR92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *J. Logic Programming*, 12(3):191–224, 1992.

[NPT96] A.P. Nemytykh, V.A. Pinchuk, and V.F. Turchin. A Self-Applicable Supercompiler. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 322–337. Springer LNCS 1110, 1996.

[O'D77] M. O'Donnell. *Computing in Systems Described by Equations*. Springer LNCS 58, 1977.

[OSV04a] C. Ochoa, J. Silva, and G. Vidal. A Lightweight Approach to Program Specialization. In *Proc. of IV Jornadas de Programación y Lenguajes (PROLE'04), Málaga (Spain)*, pages 41–54, 2004.

[OSV04b] C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 123–134. ACM Press, 2004.

[OSV04c] C. Ochoa, J. Silva, and G. Vidal. Program Specialization Based on Dynamic Slicing. In *Proc. of Workshop on Software Analysis and Development for Pervasive Systems (SONDA'04)*, pages 20–31, 2004.

[OSV05] C. Ochoa, J. Silva, and G. Vidal. Lightweight Program Specialization via Dynamic Slicing. In *Proc. of the Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 1–7. ACM Press, 2005.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.

[PHG99] G. Puebla, M. Hermenegildo, and J. P. Gallagher. An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework. In *Proc. of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, ed. O. Danvy.*, pages 75–84, 1999.

[PHH99] J. Peterson, G. Hager, and P. Hudak. A Language for Declarative Robotic Programming, 1999.

[PJ03] Simon Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[PKL99] L.L. Peterson, S. Karlin, and K. Li. OS Support for General-Purpose Routers. In *Workshop on Hot Topics in Operating Systems (Hot-OS-VII)*, pages 38–43. IEEE Computer Society Technical Committee on Operating Systems, 1999.

[Pos81] J. B. Postel. Internet protocol. RFC 791, Internet Engineering Task Force, September 1981.

[PP96] A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 355–385. Springer LNCS 1110, 1996.

[RAS06] J.G. Ramos, G. Arroyo, and J. Silva. Compiling by Narrowing-Driven Partial Evaluation. In *Proc. of 13th International Congress on Computer Science Research (CIICC 2006)*, pages 221–232, 2006.

[Red85] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, New York, 1985.

[Rey98] J.C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–297, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[RSV03a] J.G. Ramos, J. Silva, and G. Vidal. ROSE: Router Specification in a Declarative Multi-Paradigm Language. In *Proc. of 10th International Congress on Computer Science Research (CIICC'03)*, pages 108–115, 2003.

[RSV03b] J.G. Ramos, J. Silva, and G. Vidal. Towards Router Specification in Curry: The Language ROSE. In *Proc. of III Jornadas de Programación y Lenguajes (PROLE'03)*, pages 105–118, 2003.

[RSV04a] J.G. Ramos, J. Silva, and G. Vidal. An Embedded Language Approach to Router Specification in Curry. In *Proc. of the 30th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2004)*, pages 277–288. Springer-Verlag LNCS 2932, 2004.

[RSV04b] J.G. Ramos, J. Silva, and G. Vidal. Router Specification in a Declarative Multi-Paradigm Language. Technical report, Technical University of Valencia, 2004. Available at `http://www.dsic.upv.es/~guadalupe`.

[RSV05a] J.G. Ramos, J. Silva, and G. Vidal. An Offline Partial Evaluator for Curry Programs. In *Proc. of the Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 49–53. ACM Press, 2005.

[RSV05b] J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. *ACM SIGPLAN Notices (Proc. of ICFP'05)*, 40(9):228–239, 2005.

[RSV05c] J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. Technical report, Technical University of Valencia, 2005. Available at `http://www.dsic.upv.es/~gvidal`.

[RSV05d] J.G. Ramos, J. Silva, and G. Vidal. Offline Narrowing-Driven Partial Evaluation. In *Proc. of the V Jornadas de Programación y Lenguajes (PROLE'05)*, pages 233–238. Thomson, 2005.

[RSV07] J.G. Ramos, J. Silva, and G. Vidal. Ensuring the Quasi-Termination of Needed Narrowing Computations. *Information Processing Letters*, 101(5):220–226, 2007.

[SCK04] S. Seefried, M.M.T. Chakravarty, and G. Keller. Optimising Embedded DSLs Using Template Haskell. In *Proc. of 3rd Int'l Conf. on Generative Programming and Component Engineering (GPCE'04)*, pages 186–205, 2004.

[SG95] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of the Int'l Logic Programming Symposium (ILPS'95)*, pages 465–479. The MIT Press, Cambridge, MA, 1995.

[SGJ96] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[Sla74] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[SR97] J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *PLILP*, pages 291–308, 1997.

[ST96] M. Sperber and P. Thiemann. Realistic Compilation by Partial Evaluation. In *Proc. of the ACM SIGPLAN 1996 conference on Programming language design and implementation, PLDI '96*, pages 206–214. ACM Press, 1996.

[SV07] J. Silva and G. Vidal. Forward Slicing of Functional Logic Programs by Partial Evaluation. *Theory and Practice of Logic Programming*, 7:215–247, 2007.

[TG05] R. Thiemann and J. Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Appl. Algebra Eng. Commun. Comput.*, 16(4):229–270, 2005.

[TS84] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In S. Tärnlund, editor, *Proc. of the 2nd Int'l Conf. on Logic Programming (ICLP'84)*, pages 127–139, 1984.

[Vid96] G. Vidal. *Semantics-Based Analysis and Transformation of Functional Logic Programs.* PhD thesis, DSIC, Universidad Politécnica de Valencia, 1996.

[Vid02] G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62. ACM Press, 2002.

[Vid03] G. Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR 2002)*, pages 219–237. Springer LNCS 2664, 2003.

[Vid04] G. Vidal. Cost-Augmented Partial Evaluation of Functional Logic Programs. *Higher-Order and Symbolic Computation*, 17(1-2):7–46, 2004.

[Wad90] P.L. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.

[War82] D.H.D. Warren. Higher-Order Extensions to Prolog —Are they needed? In Michie Hayes-Roth and Pao, editors, *Machine Intelligence*, volume 10. Ellis Horwood, 1982.

[Wei79] M.D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, The University of Michigan, 1979.

[Wei81] M. Weiser. Program Slicing. In *Proc. of the 5th Int'l. Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

[You91] J-H. You. Unification Modulo an Equality Theory for Equational Logic Programming. *Journal of Computer and System Sciences*, 42:54–75, 1991.