Especialización de Programas Lógico–Funcionales Perezosos.

Pascual Julián Iranzo.
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia.

Memoria presentada para optar al título de:
Doctor en Informática.

Dirigida por:
María Alpuente Frasnedo.

Tribunal de lectura:

Presidente:	**Isidro Ramos Salavert**	**U.P. Valencia**
Vocales:	**Mario Rodríguez Artalejo**	**U.C. Madrid**
	Moreno Falaschi	**U. Udine**
	Ricardo Peña Marí	**U.C. Madrid**
Secretario :	**Salvador Lucas Alba**	**U.P. Valencia**

Mayo de 2000

Resumen.

La *evaluación parcial*, también denominada *especialización de programas*, puede entenderse como una técnica de transformación que consigue la optimización del programa original para una clase de datos de entrada. Más formalmente, dado un programa P y parte de sus datos de entrada $in1$, el objetivo de la evaluación parcial es construir un nuevo programa P_{in1} que cuando se le introduce el resto de los datos de entrada $in2$, computa el mismo resultado que P con toda su entrada $(in1 + in2)$. Esto último asegura la corrección de la transformación efectuada. Contrariamente a lo que sucede con otras técnicas de transformación, el proceso de evaluación parcial puede automatizarse completamente. La evaluación parcial ha sido aplicada intensivamente tanto a los lenguajes imperativos como declarativos y extensivamente a una gran variedad de problemas concretos [108].

Los lenguajes declarativos presentan una semántica simple y con bases teóricas bien establecidas que permiten razonar acerca de la corrección de los programas. Esta característica de los lenguajes declarativos se ha aprovechado en la definición de métodos de evaluación parcial para lenguajes funcionales [53, 108] y lenguajes lógicos [75, 116, 137, 164] que heredan las buenas propiedades de sus mecanismos de base. Recientemente, estas técnicas se han extendido al caso de los lenguajes lógico-funcionales [12, 14, 123]. El marco genérico para la evaluación parcial de programas lógico-funcionales propuesto inicialmente en [12], se basa en el empleo del *narrowing*, una regla de inferencia que combina el principio de reducción de los lenguajes funcionales y el principio de resolución de los lenguajes lógicos.

Esta tesis tiene como principal objetivo investigar las técnicas de especialización de programas lógico-funcionales perezosos dentro del marco establecido en [14], para lo cual introduce:

- `Mejoras en el mecanismo de base`. El objetivo, en este caso, es alcanzar una mayor eficiencia tanto en el proceso de evaluación parcial como en la mejora que se consigue al ejecutar el programa especializado resultante. Partiendo de una instancia del marco genérico, en la que se emplea *narrowing* perezoso como mecanismo de base y para la que demostramos su corrección, la principal aportación en esta línea es el identificar una clase particular de programas, que definimos a partir de la idea de los programas uniformes de [120, 121], sobre los cuales es posible refinar la estrategia

de *narrowing* perezoso, sin pérdida de completitud, obteniéndose un evaluador parcial para el que se consiguen mejores prestaciones que con la estrategia de *narrowing* original.

- `Técnicas avanzadas de especialización`. En esta línea, mejoramos los procedimientos de control del algoritmo de evaluación parcial mediante la introducción de una regla de desplegado dinámica y técnicas de partición que permiten alcanzar una especialización poligenética (i.e., aquella que es capaz de combinar varias definiciones de función del programa original en una definición especializada) sin el empleo de artificios *ad hoc*. Todo esto sin afectar a la terminación del proceso.

También se ha estudiado la efectividad de las técnicas desarrolladas para el caso concreto de (fragmentos de) lenguajes lógico-funcionales modernos con semántica no estricta, como Curry [96] o $\mathcal{TOY}$ [49]. Las nuevas estrategias de control se han incorporado al sistema INDY [4], obteniendose mejoras en las versiones especializadas de la mayoría de los programas de prueba considerados, que resultan ser más precisas y eficientes que las obtenidas en la implementación previa.

Palabras clave: Integración de la programación lógica y funcional, sistemas de reescritura de términos, *narrowing*, estrategias de evaluación perezosa, *narrowing* perezoso, *narrowing* necesario, evaluación parcial, especialización de programas, técnicas de control local y global, relaciones de orden, términación.

Agradecimientos.

Quisiera dirigir las primeras palabras de agradecimiento a María Alpuente, una persona de la que me es muy difícil hablar con objetividad e imparcialidad, pues me une a ella una lejana amistad, que se remonta a los tiempos de la Facultad; después la vida nos ha llevado por diversos caminos unas veces próximos otras alejados. Hoy quiero hablar de su faceta profesional y del ejemplo constante que para mí ha sido, por su rigor e incansable dedicación investigadora. Quisiera agradecerle el haber confiado en mí, en aquella fecha ya lejana de Enero de 1995, cuando me propuso unirme al grupo de investigación que ella dirige (entonces en sus inicios y ahora una fértil realidad). También quisiera agradecerle que desde el primer momento me diera apoyo y ayuda constante, así como su certera orientación a la hora de corregir y depurar las ideas de los sucesivos trabajos en los que he participado con ella y que conforman la presente memoria de tesis. Pero mi principal agradecimiento radica en que me haya enseñado el oficio de investigador; sólo espero haber sido un buen aprendiz.

En segundo lugar es de justicia mencionar a Germán Vidal, para el que no tengo suficientes palabras de agradecimiento. Su absoluta disponibilidad e interés son conmovedores y siempre han estado más allá de su obligación. Su intuición y habilidad natural en la búsqueda de contraejemplos han supuesto una ayuda indispensable en la mejora de mi trabajo. También quisiera agradecerle su trato humano y los consejos y palabras de aliento recibidos en más de una ocasión.

Agradecer también a Elvira Albert su apreciable ayuda en muchos instantes y su paciencia al explicarme las peculiaridades del sistema INDY. También a los recién llegados Santiago Escobar, Cesar Ferri y Cándido Martínez, que demostraron su habilidad técnica.

A José Miguel Benedí, Matilde Celma y Asunción Casanova, que me orientaron en los primeros momentos. A Javier Oliver, que siempre tuvo palabras de aliento, a María José Ramírez, Javier Piris, Carlos Herrero, José Orallo y, en general, a todos los miembros del Grupo Extensiones de la Programación Lógica, del Departamento de Sistemas Informáticos y Computación de la UPV, que ayudaron a crear un ambiente de trabajo agradable durante las sucesivas estancias de investigación realizadas en la UPV.

Una mención especial para Ginés Moreno, más que un amigo y compañero de fatigas, él ya lo sabe, con el deseo de que nuestras vidas sigan discurriendo por el mismo camino. También agradecer a Manuel Prieto su ayuda, compresión y la confianza que depositó en mí al aceptar ser mi tutor en el Departamento de Informática de la Universidad de Castilla - La Mancha. A la Universidad de Castilla - La Mancha por las diversas ayudas económicas que me fueron concedidas durante la realización de este doctorado.

Finalmente, quisiera agradecer a mi familia y, especialmente a mi mujer Nieves, la paciencia y comprensión que han demostrado durante estos años llenos de dificultad y de necesaria disciplina.

A Nieves, a mis padres,
a mis hermanos, a mi familia
y a la memoria de J.L.T. y M.M.

DEL RIGOR EN LA CIENCIA.

"... En aquel Imperio, el Arte de la Cartografía logró tal perfección que el mapa de una sola Provincia ocupaba toda una Ciudad, y el mapa del Imperio, toda una Provincia. Con el tiempo, esos mapas desmesurados no satisfacieron y los Colegios de Cartógrafos levantaron un Mapa del Imperio, que tenía el tamaño del Imperio y coincidia puntualmente con él. Menos adictas al estudio de la cartografía, las generaciones siguientes entendieron que ese dilatado mapa era inútil y no sin impiedad lo entregaron a las inclemencias del Sol y de los Inviernos. En los desiertos del Oeste perduran despedazadas Ruinas del Mapa, habitadas por animales y por mendigos; en todo el País no hay otra reliquia de las Disciplinas Geográficas."

Suárez Miranda:
VIAJES DE VARONES PRUDENTES,
LIBRO CUARTO, CAP. XLV, Lérida, 1658.

Jorge Luis Borges en EL HACEDOR.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción.

1.1 Programación Declarativa.

La *programación declarativa* (a veces llamada *programación inferencial*) puede entenderse como un estilo de programación en el que el programador especifica *qué* debe computarse más bien que *cómo* deben realizarse los cómputos. En este paradigma de programación, de acuerdo con el famoso aserto de Kowalski [118, 117], un *programa = lógica + control*, y la tarea de programar consiste en centrar la atención en la *lógica* dejando de lado el *control*, que se asume automático, al sistema. Con más precisión, la característica fundamental de la programación declarativa es el uso de la lógica como lenguaje de programación, lo cual puede conceptualizarse como sigue:

- Un *programa* es una teoría formal en una cierta lógica, y
- la *computación* se entiende como una forma de inferencia o deducción en dicha lógica.

Los principales requisitos que debe cumplir la lógica empleada son: i) disponer de un lenguaje que sea suficientemente expresivo para cubrir un campo de aplicación interesante; ii) disponer de una semántica operacional, esto es, un mecanismo de cómputo que permita ejecutar los programas; iii) disponer de una semántica declarativa que permita dar un significado a los programas de forma independiente a su posible ejecución; y iv) resultados de corrección y completitud que aseguren que lo que se computa coincide con aquello que es considerado como verdadero (de acuerdo con la noción de verdad que sirve de base a la semántica declarativa). Desde el punto de vista del soporte a la declaratividad, el tercero de los requisitos es, tal vez, el más importante, ya que es el que permite especificar *qué* estamos computando. Concretamente, la semántica declarativa especifica el significado de los objetos sintácticos del lenguaje por medio de su traducción en elementos y estructuras de un dominio (generalmente matemático) conocido.

A pesar de ser un área de trabajo relativamente nueva, en términos del tiempo necesario para el desarrollo y la consolidación de un área de conocimiento, la programación declarativa ha encontrado una gran variedad de aplicaciones. Sin animo de ser exhaustivos, podemos enumerar algunas de sus aplicaciones:

> Procesamiento del lenguaje natural. Representación del conocimiento. Desarrollo de Sistemas de Producción y Sistemas Expertos. Resolución de Problemas. Metaprogramación. Prototipado de aplicaciones. Bases de Datos Deductivas. Servidores y buceadores de información inteligentes. Quimica y biología molecular. Diseño de sistemas VLSI.

Más generalmente, la programación declarativa se ha aplicado en todos los campos de la computación simbólica y por esto también los lenguajes declarativos se denominan a veces, lenguajes de computación simbólica (en contraposición a los lenguajes más tradicionales orientados a la computación numérica).

La programación declarativa incluye tanto la programación lógica como la funcional, cuyas principales características se resumen en los dos próximos apartados.

1.1.1 Programación Lógica.

La *programación lógica* ([26, 118, 134]) se basa en fragmentos de la lógica de predicados, siendo el más popular la lógica de *cláusulas de Horn* (HCL, del inglés *Horn clause logic*), que pueden emplearse como base para un lenguaje de programación al poseer una semántica operacional susceptible de una implementación eficiente, como es el caso de la *resolución SLD*. Como semántica declarativa se utiliza una semántica por *teoría de modelos* que toma como dominio de interpretación un universo puramente sintáctico: el *universo de Herbrand*. La *resolución SLD* es un método de prueba por refutación que emplea el algoritmo de *unificación* como mecanismo de base y permite la extracción de respuestas (i.e., el enlace de un valor a una *variable lógica*). La resolución SLD es un método de prueba correcto y completo para la lógica HCL.

Algunas de las características de la programación declarativa pueden ponerse de manifiesto mediante un sencillo ejemplo.

Ejemplo 1 *Consideremos el conocido problema de la concatenación de dos listas. En programación lógica el problema se resuelve definiendo una relación app con tres argumentos: generalmente, los dos primeros hacen referencia a las listas que se desea concatenar y el tercero a la lista que resulta de la concatenación de las dos primeras. El programa se compone de dos cláusulas de Horn*[1].

$$
\begin{array}{lcl}
app([\,], X, X) & \leftarrow & \\
app([X|X_s], Y, [X|Z_s]) & \leftarrow & app(X_s, Y, Z_s)
\end{array}
$$

[1] Se hace uso de la notación que emplea el lenguaje PROLOG para la representación de listas, en la que las variables se escriben con mayúsculas, el símbolo "[ ]" representa la lista vacía y el símbolo "|" es el constructor de listas.

Programación Lógica	Programación Funcional
Programa: Conjunto de cláusulas que definen relaciones	Programa: Conjunto de ecuaciones que definen funciones.
Semántica operacional: Resolución SLD (unificación)	Semántica operacional: Reducción (ajuste de patrones)
Semántica declarativa: Teoría de modelos (Mod. mínimo)	Semántica declarativa: Algebraica (Mod. inicial) / Denotacional
Primer orden	Orden superior
Uso múltiple	Uso único
Indeterminismo	Determinismo
E/S adireccional	E/S direccional
Variables lógicas	Sin variables lógicas
Sin tipos	Tipos y polimorfismo
Datos parcialmente especificados	Datos completamente especificados
No estructuras infinitas	Estructuras potencialmente infinitas
No evaluación perezosa	Evaluación perezosa

Tabla 1.1: Características de la programación lógica y de la programación funcional: Diferencias.

Este programa tiene una lectura declarativa clara. La primera cláusula afirma que

> *la concatenación de la lista vacía* $[\,]$ *y otra lista* X *es la propia lista* X.

mientras que la segunda cláusula puede entenderse en los siguientes términos.

> *La concatenación de dos listas* $[X|X_s]$ *e* Y *es la lista que resulta de añadir el primer elemento* X *de la lista* $[X|X_s]$ *a la lista* Z_s*, que se obtiene al concatenar el resto* X_s *de la primera lista a la segunda* Y.

Uno de los primeros hechos que advertimos en el Ejemplo 1 es que no hay ninguna referencia explícita al tipo de representación en memoria de la estructura de datos lista. De hecho, una característica de los lenguajes declarativos es que proporcionan una *gestión automática de la memoria*, evitando una de las mayores fuentes de errores en la programación con otros lenguajes (piénsese en el manejo de punteros en el lenguaje C). Por otra parte, el programa puede responder, haciendo uso del mecanismo de resolución SLD, a diferentes cuestiones (*objetivos*) sin necesidad de efectuar ningún cambio en el programa, gracias al hecho de emplearse un mecanismo de cómputo que permite una *búsqueda indeterminista* (*built-in search*) de soluciones. Esta misma característica permite computar con *datos parcialmente definidos* y hace posible que la *relación de entrada/salida* no esté fijada de antemano.

Estas y otras características de los lenguajes de programación lógica se resumen en la Tabla 1.1.

1.1.2 Programación Funcional.

Los *lenguajes funcionales* están enraizados en el concepto de *función* (matemática) y su definición mediante ecuaciones (generalmente recursivas), que constituyen el programa. Desde el punto de vista computacional, la programación funcional se centra en la evaluación de expresiones (funcionales) para obtener un *resultado*.

Ejemplo 2 *Utilizando un lenguaje de programación funcional, el problema del Ejemplo 1 se resuelve definiendo una función app de dos argumentos, que representan las listas que se desea concatenar, y que devuelve como resultado la concatenación de dichas listas. El programa consiste en una declaración de tipos, una declaración de la función app, en la que se fija su dominio y su rango, y dos ecuaciones que la definen*[2].

$$
\begin{array}{lcl}
data\ [t] & = & [\,]\ |\ [t : [t]] \\
app & :: & [t] \rightarrow [t] \rightarrow [t] \\
 & & \\
app\ [\,]\ x & = & x \\
app\ (x : x_s)\ y & = & x : (app\ x_s\ y)
\end{array}
$$

Un primer rasgo a destacar en el programa del Ejemplo 2 es la necesidad de emplear recursos expresivos para fijar el perfil de la función, i.e., la naturaleza del dominio y el rango de la función. En programación funcional la descripción de dominios conduce a la idea de *tipo de datos*. Los tipos de datos se introducen en programación para evitar o detectar errores y ayudar a definir estructuras de datos. Los lenguajes funcionales modernos son lenguajes *fuertemente basados en tipos* (*strongly typed*) que realizan una comprobación de las expresiones que aparecen en el programa para asegurarse de que no se producirán errores durante la ejecución del mismo por incompatibilidades de tipo. También debemos reparar en que se ha definido la estructura de datos lista, mediante una declaración de datos, donde los símbolos "[ ]" y ":" son los *constructores del tipo* y el símbolo t es una *variable de tipo*, de forma que podemos formar listas de diferentes tipos de datos. Esta facultad se denomina *polimorfismo* y es otro rasgo muy apreciado en los lenguajes funcionales modernos[3].

Lo que caracteriza a una función (matemática), además de su perfil, es que a cada elemento de su dominio le corresponde un único elemento del rango; en otras palabras, el resultado (*salida*) de aplicar una función sobre sus argumentos viene determinado exclusivamente por el valor de éstos (su *entrada*). Esta propiedad de las funciones (matemáticas) se denomina *transparencia referencial*. Debido a que los programas en los lenguajes imperativos mantienen un *estado* interno del cómputo, que es modificado durante la ejecución de la secuencia de

[2]Se hace uso de la sintaxis del lenguaje funcional Haskell, en la que el símbolo ":" representa el constructor de listas.

[3]Haskell, además de poseer un sistema de tipos algebraicos polimórficos, permite la declaración de *tipos abstractos de datos* mediante la declaraciones `export/import`, que controla el ocultamiento de la información y propicia la modularidad y la seguridad.

intrucciones, las construcciones funcionales de estos lenguajes pueden presentar *efectos laterales* (*side effects*), incumpliendo la propiedad de transparencia referencial y no denotando, por consiguiente, verdaderas funciones (matemáticas). La transparencia referencial permite el estilo de la programación funcional basado en el razonamiento ecuacional (la substitución de iguales por iguales) y los cómputos deterministas. Desde el punto de vista computacional, otra propiedad de las funciones (matemáticas) es su capacidad para ser compuestas. La composición de funciones es la técnica por excelencia de la programación funcional, que permite la construcción de programas mediante el empleo de funciones primitivas o previamente definidas por el usuario en sus programas. La composición de funciones refuerza la modularidad de los programas.

Ejemplo 3 *En este ejemplo hacemos uso de la función app definida en el Ejemplo 2 y la composición de funciones para definir una versión ingenua de una función que invierte el orden de los elementos en una lista.*

$$
\begin{array}{lcl}
rev & :: & [t] \to [t] \\
\\
rev\ [\,] & = & [\,] \\
rev\ (x : x_s) & = & app\ (rev\ x_s)\ [x]
\end{array}
$$

Una de las características de los lenguajes funcionales que va más alla de la mera composición de funciones es el empleo de las mismas como "ciudadados de primera clase" dentro del lenguaje, de forma que las funciones puedan almacenarse en estructuras de datos, pasarse como argumento a otras funciones y devolverse como resultados. Agrupamos todas estas características bajo la denominación de *orden superior*. A veces también se dice que una función es de orden superior si algunos de sus argumentos es, a su vez, una función, esto es, si está definida sobre un dominio funcional.

Ejemplo 4 *Un ejemplo típico de función de orden superior es la función map, que toma como argumento una función f y una lista, y forma la lista que resulta de aplicar f a cada uno de los elementos de la lista.*

$$
\begin{array}{lcl}
map & :: & (t_1 \to t_2) \to [t_1] \to [t_2] \\
\\
map\ f\ [\,] & = & [\,] \\
map\ f\ (x : x_s) & = & (f\ x) : (map\ f\ x_s)
\end{array}
$$

Diversos autores [169, 173] consideran que un programa funcional incluye también, además de una lista de ecuaciones de definición de funciones, una expresión básica (i.e., sin variables) que se pretende evaluar como *objetivo*. La inclusión de una *expresión inicial* pone de manifiesto el componente dinámico de los leguajes funcionales, donde la ejecución de un programa consiste en la evaluación de la expresión inicial de acuerdo con las ecuaciones de definición de las funciones y alguna forma de reducción. La *reducción* es un proceso por el cual, mediante una secuencia de pasos, transformamos la expresión inicial, compuesta de símbolos de función y de símbolos constructores de datos, en un *valor* (de su tipo), i.e., una expresión que sólo contiene ocurrencias de símbolos constructores. El valor obtenido se considera el resultado de la evaluación. La secuencia de pasos dada en el proceso de reducción depende de la estrategia de reducción empleada. Podemos distinguir dos *estrategias de reducción* o *modos de evaluación*, la denominada impaciente y la perezosa[4]. Una estrategia *impaciente* evalúa primero los argumentos de una función mientras que una *perezosa* evalúa los argumentos sólo si su valor es necesario para el cómputo de dicha función, intentando evitar cálculos innecesarios. Si bien la estrategia impaciente es más fácil de implementar en los computadores con arquitecturas convencionales[5], puede conducir a secuencias de reducción que no terminan en situaciones en las que una evaluación perezosa es capaz de computar un valor. Este hecho, junto con algunas facilidades de programación (e.g., la evaluación perezosa libera al programador de la preocupación por el orden de evaluación de las expresiones [100]) y ventajas expresivas que proporciona (e.g., la habilidad de computar con estructuras de datos infinitas), han hecho que la posibilidad de utilizar un modo evaluación perezosa sea muy apreciada en los lenguajes funcionales modernos[6].

Como puede apreciarse los lenguajes de programación funcional son muy ricos en cuanto a las facilidades que ofrecen[7]. La Tabla 1.1 resume las características de los lenguajes funcionales en comparación con los lenguajes lógicos.

Hasta aquí se ha evitado hablar de los formalismos que sustentan esta clase de lenguajes, tal vez porque no haya un consenso unánime (como en el caso de la programación lógica) en la utilización de un determinado tipo de lógica. Pueden distinguirse tres diferentes aproximaciones: la ecuacional ([159, 160, 161]), la algebraica ([38, 66, 204]) y la funcional clásica ([32, 39, 71, 100, 167, 173]).

La *orientación clásica* de la programación funcional utiliza el λ-cálculo extendido, junto con nociones de la lógica combinatoria, como lógica de base. En esta orientación, el λ-cálculo representa una especie de lenguaje máquina al cual pueden compilarse los programas funcionales escritos en lenguajes más sofisticados y que proporciona un mecanismo operacional básico que permite ejecutar dichos programas: la β-reducción. Para dar significado a las estructuras sintácticas se emplea una semántica denotacional [182, 191].

[4]En el contexto del λ-cálculo, la estrategia de reducción impaciente (*eager*) se identifica con el *orden de reducción aplicativo*, mientras que la perezosa (*lazy*) con el *orden de reducción normal*.

[5]La estategia de evaluación impaciente puede implementarse utilizando las técnicas de paso de parámetros por valor desarrolladas para la compilación de los lenguajes imperativos.

[6]Muchos de los primeros lenguajes funcionales, como Lisp (puro), FP, ML, o Hope, fueron implementados empleando un modo de evaluación impaciente por las razones ya comentadas. Sin embargo, desde que existen técnicas eficientes de implementación de la *reducción en grafos* [167], los lenguajes funcionales modernos como Haskell y Miranda utilizan modos de evaluación perezosa.

[7]Algunas de las ventajas que hemos discutido en este apartado asociadas a los lenguajes funcionales han sido transferidas a lenguajes procedimentales de corte más convencional, como los lenguajes orientados a objetos. Se ha acuñado el acrónimo HOT (del inglés *High Order and strongly Typed*) para hacer referencia a los lenguajes de programación que reúnen estas características. Puede decirse que lenguajes orientados a objetos como Java son HOT, si bien el polimorfismo y el orden superior se obtienen en este lenguaje de manera oscura y un tanto imperfecta a través del mecanismo de la herencia.

Las otras dos aproximaciones pueden verse como caras de la misma moneda. La *lógica ecuacional* proporciona un soporte sintáctico para la definición de funciones (mediante el uso de ecuaciones[8]) y el razonamiento ecuacional (mediante un sistema de deducción ecuacional que plasma como reglas de inferencia las conocidas propiedades reflexiva, simétrica y transitiva de la identidad, así como las propiedades de cierre por substitución de idénticos por idénticos y de cierre por instanciación). Una *interpretación* del lenguaje ecuacional se define adoptando un álgebra como dominio de interpretación y una función de evaluación que asocia: a los símbolos particulares del lenguaje, símbolos de constante y de función, elementos y funciones del dominio, respectivamente; y a las variables, elementos del dominio. Así pues, podemos dar significado a un sistema ecuacional mediante una semántica declarativa algebraica. El teorema de Birkhoff establece la equivalencia entre la posibilidad de deducir la igualdad de dos términos (sin variables) a partir de un sistema ecuacional y la verdad de la correspondiente ecuación en el algebra inicial cociente (modelo del sistema ecuacional)[9]. En esta aproximación la semántica operacional se basa en el empleo de los sistemas de reescritura de términos. El razonamiento ecuacional se automatiza asociando a una teoría ecuacional el sistema de reescritura resultante de imponer direccionalidad a las ecuaciones que lo forman (e.g. orientándolas de izquierda a derecha) y utilizando como mecanismo de ejecución la reducción de expresiones por reescritura. De esta forma, el programa pasa a ser considerado un sistema de reglas de reescritura. En un paso de *reducción* se realiza un ajuste de patrones (*pattern matching*) que empareja un subtérmino del objetivo ecuacional con la parte izquierda de una regla y se reemplaza éste por la correspondiente instancia de la parte derecha de dicha regla. Si el sistema de reescritura considerado es canónico[10], deducir la igualdad de dos términos (sin variables) a partir de un sistema ecuacional será equivalente a comprobar la igualdad (sintáctica) de las formas normales o irreducibles que se obtienen al culminar el proceso de reducción por reescritura de dichos términos.

En una orientación puramente algebraica, el interés está centrado directamente en el uso de los sistemas de reescritura como programas y el cómputo de valores a partir de expresiones funcionales, más bien que en comprobar la validez de objetivos ecuacionales. Ya hemos mencionado que el interés primordial de la programación funcional es el cómputo de valores.

Si bien los sistemas de reescritura y su mecanismo de reducción están muchas veces más cerca, desde el punto de vista sintáctico y computacional, de los programas funcionales que el λ-cálculo y la β-reducción, la utilización de los sistemas de reescritura como formalización de los lenguajes de programación funcional, presenta ciertas dificultades:

[8]En el caso más general, cláusulas de Horn ecuacionales, en las que el único símbolo de predicado permitido es la igualdad.

[9]Equivalentemente, el teorema de Birkhoff puede enunciarse en términos de *validez* de la ecuación (verdad con respecto a todos los modelos del sistema ecuacional) gracias a la propiedad de inicialidad del algebra cociente.

[10]Se dice que un sistema de reescritura es canónico si es confluente y terminante (ver más adelante).

1. Los lenguajes funcionales emplean tipos.

2. Los programas funcionales siguen la *disciplina de constructores*, esto es, en los argumentos de las partes izquierdas de las ecuaciones que definen las funciones solamente pueden aparecer términos formados por constructores y/o variables.

3. En un programa funcional existe un orden entre las ecuaciones que forman el programa (en general, el orden en el que están escritas), lo que permite que exista un solapamiento entre las partes izquierdas de dichas ecuaciones.

4. Es habitual emplear *guardas* en las ecuaciones de definición de las funciones.

Al emplear sistemas de reescritura como programas, cada una de estas dificultades puede salvarse, respectivamente: utilizando signaturas *heterogéneas* (*many sorted*); restringiéndonos a sistemas de reescritura que también sigan la disciplina de constructores; considerando el sistema de reescritura como un conjunto ordenado de reglas de reescritura; y utilizando sistemas de reescritura condicionales o bien ampliando el sistema de reescritura con reglas que definan expresiones condicionales (del tipo *if-then-else*) cuya evaluación puede efectuarse mediante reescritura. Existen sin embargo otras dificultades (relacionadas con el empleo del orden superior y el uso de funciones currificadas) más difícilmente soslayables, que nos obligan a admitir que los programas funcionales son mucho más que sistemas de reescritura de términos. A pesar de la anterior afirmación, es de destacar que la orientación que fundamenta la programación funcional en la lógica ecuacional y en los sistemas de reescritura tiene la ventaja, respecto a la orientación clásica, de que puede establecerse una correspondencia más sencilla con el formalismo que sustenta la programación lógica. Esta ventaja es apreciable a la hora de integrar ambos paradigmas de programación y explica la atención que prestaremos a los sistemas de reescritura de términos en los próximos capítulos.

1.2 Programación Lógico–Funcional.

La división de la programación declarativa presentada en el Apartado 1.1 tiene una raíz histórica ya que en ambas aproximaciones se intenta resolver los mismos problemas con técnicas parecidas. Esto explica el interés existente por lograr la integración de los lenguajes lógicos y funcionales para obtener lo mejor de ambos.

La integración de la programación lógica y funcional es un área de investigación activa desde principios de los años ochenta. Los lenguajes de programación lógico–funcionales permiten integrar algunas de las mejores características de los paradigmas de programación lógica y funcional. De la programación lógica, los lenguajes lógico–funcionales obtienen el uso de la unificación, la potencia de las variables lógicas y la inversión de definiciones, un mecanismo de búsqueda automático indeterminista y la posibilidad de trabajar con estructuras de datos parciales. De la programación funcional obtienen, entre otros beneficios, la expresividad de las funciones, el empleo de tipos, el orden superior y un mecanismo de evaluación más eficiente (determinismo y evaluación perezosa). El empleo de la técnica de reducción permite evitar características extralógicas introducidas en la programación con PROLOG para reducir el espacio de búsqueda (*el operador de corte*). En la última decada se han realizado importantes avances en el campo de la integración de lenguajes declarativos, una panorámica de los cuales puede encontrarse en [94, 99] y [155].

Se han formulado muchas propuestas para la combinación de los estilos de programación lógica y funcional. La gran variedad de escuelas que se aprecia en este campo refleja una diversidad importante también de motivaciones. La forma más sencilla de integración consiste en proporcionar una interfaz para un lenguaje lógico y uno funcional ya existentes [176, 193]. Otro método consiste en traducir los programas lógico–funcionales a programas lógicos puros, aplanando las llamadas anidadas a función [31, 78]. Recientemente, en [135] se ha desarrollado una propuesta de integración de la programación lógica y funcional en la que se considera un lenguaje lógico–funcional con tipos y orden superior (basado en la teoría simple de tipos de Church) y que utiliza un mecanismo operacional que combina la técnica de resolución SLD con el denominado principio de *residuación*. Informalmente, la residuación retrasa aquellas llamadas a función que no están lo suficientemente instanciadas para efectuar un paso (determinista) de reducción. La combinación de ambos mecanismos consiste en aplicar la regla de resolución solamente sobre los objetivos relacionales y reducir de forma determinista los objetivos ecuacionales cuando están listos para ser evaluados.

El principal inconveniente de utilizar la residuación en la definición de la semántica operacional es que se trata de un método incompleto. Por consiguiente, la mayoría de las propuestas para la integración usan los sistemas de reescritura de términos como programas y (alguna variante de) el *narrowing* como mecanismo operacional, soportando así unificación y variables lógicas en un contexto funcional. El procedimiento de *narrowing* puede verse como una extensión de la reescritura que puede ser implementado eficientemente sustituyendo el *ajuste de patrones* por la unificación en el procedimiento de reducción. En un paso de *narrowing* se unifica un subtérmino del objetivo (ecuacional) con la parte izquierda de una regla y se reemplaza éste por la parte derecha instanciada de la regla [11]. Pese a que el algoritmo de *narrowing* es un método correcto y completo, su terminación no está asegurada. La técnica de *narrowing*, como medio para la obtención de un conjunto de soluciones en un problema de $\mathcal{E}$-unificación, fue descrita en los trabajos pioneros de [184] y [68]. La relación definida por Fay en [68], y que denominó *narrowing*, es una variante que posteriormente ha recibido el nombre de *narrowing normalizante* [175, 94]. *Narrowing normalizante* se caracteriza por la normalización del término a evaluar previa a la aplicación

[11] Añadiendo, también, al nuevo objetivo las correspondientes instancias de las condiciones de la regla utilizada, si se trata de un sistema de reescritura de términos condicional

Lenguaje	Principio operacional	Comportamiento
LOGLISP [176]	Combinación lenguaje lógico y Funcional (Resolución + Reducción)	Impaciente
K-LEAF [78]	Aplanamiento + SLD-Resolución	Perezoso
ESCHER [135]	Residuación + SLD-Resolución	Perezoso
SLOG [72]	Narrowing	Impaciente
ALF [92]	Narrowing	Impaciente
LPG [37]	Narrowing	Impaciente
BABEL [157]	Narrowing	Perezoso
$\mathcal{TOY}$ [49]	Narrowing	Perezoso
Curry [98]	Narrowing + Residuación	Perezoso

Tabla 1.2: Algunos lenguajes lógico–funcionales y sus características.

de cada paso de *narrowing*.

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. Se han diseñado muchas estrategias para reducir el tamaño del espacio de búsqueda, eliminando algunas derivaciones inútiles, entre las que podemos citar: *innermost narrowing* [72], *narrowing básico* [103, 149], *narrowing selección* [44], *narrowing perezoso* [23, 93, 157, 174], etc. Cada una de estas estrategias sigue manteniendo, bajo determinadas condiciones, la completitud del cálculo. En [1, 33, 62, 154] y [174] se puede encontrar una revisión, análisis y clasificación de éstas y otras propuestas para la integración. La colección [57] constituye una referencia estándar en el campo. La panorámica más actualizada se presenta en [94] y en [155]. Estudios detallados sobre las condiciones que debe cumplir un programa, para que una determinada estrategia sea correcta y completa, pueden encontrarse en [94] y en [149].

De entre las estrategias perezosas, la estrategia de *narrowing necesario* [23] es la que presenta mejores propiedades de optimalidad con respecto al número de soluciones y a la longitud de las derivaciones (en implementaciones basadas en grafos). Sin embargo, ésta es completa para programas *inductivamente secuenciales*, una clase menos amplia que la clase de los programas para los que es completa la estrategia de *narrowing* perezoso definida en [157]. Recientemente, el principio operacional del *narrowing* necesario se ha combinado con el de residuación para definir la semántica del lenguaje Curry.

Pese a todos los avances producidos y a las ventajas semánticas mencionadas, la realidad es que los lenguajes integrados no están suficientemente difundidos ni son, tampoco, ampliamente usados. Una posible explicación a esta situación es que se ha producido una excesiva proliferación de propuestas de lenguajes lógico–funcionales (algunos de los cuales se muestran en la Tabla 1.2), con el agravante de que muchas de las implementaciones pueden considerarse prototipos experimentales aún inmaduros para ser utilizados en aplicaciones reales. La ausencia de un lenguaje lógico–funcional estándar ha conducido a la programación lógico–funcional a la misma encrucijada en la que se encontraba el campo de la programación funcional, y que denunciaba Paul Hudak [100], hace ahora exactamente una decada. El lenguaje Curry [96, 98] aspira a jugar el mismo papel homogenizador que el lenguaje Haskell desempeña en el área de la programación funcional. Curry es un lenguaje multiparadigma que se está diseñando para incorporar las características más importantes de los lenguajes lógicos y funcionales modernos, como son Gödel y λ-Prolog (lenguajes lógicos), Haskell y SML (lenguajes funcionales) y Alf, BABEL y $\mathcal{TOY}$ (lenguajes lógico–funcionales). Más específicamente, el lenguaje incluye facilidades como los sistemas de tipos polimórficos, módulos, orden superior, entrada/salida declarativa monádica y búsqueda encapsulada. Estas componentes permiten reemplazar muchas de las características impuras de PROLOG, mejorando así la naturaleza declarativa de los lenguajes lógicos.

Pero para desarrollar lenguajes integrados prácticos, es necesario conseguir que su eficiencia sea comparable a la de los lenguajes imperativos (tal y como lo han conseguido el lenguaje funcional Haskell o el lenguaje lógico PROLOG). También es necesario desarrollar técnicas y herramientas que ayuden al programador a manipular y optimizar sus programas. En el próximo apartado consideramos este último aspecto.

1.3 Transformación de Programas y Evaluación Parcial.

La transformación de programas es un método para derivar programas correctos y eficientes partiendo de una especificación ingenua y más ineficiente del problema. Esto es, dado un programa P, se trata de generar un programa P' que resuelve el mismo problema y equivale semánticamente a P, pero que goza de mejor comportamiento respecto a cierto criterio de evaluación [45]. En la literatura se ha propuesto una gran variedad de transformaciones para mejorar el código. Una de las mejor estudiadas es la *evaluación parcial* (PE, del inglés *Partial Evaluation*) de programas (también denominada *especialización* de programas), que ofrece un marco unificado para la investigación acerca de los procesadores de lenguajes, en particular, compiladores e intérpretes [105, 108]. La posibilidad de especializar programas está contenida en el teorema s-m-n de Kleene [111] establecido dentro del contexto de la teoría de las funciones recursivas. Sin embargo, la evaluación parcial trabaja con programas (textos) más bien que con funciones matemáticas. Además, la técnica propuesta por Klenee produce funciones especializadas que se comportan, en algunos casos, “peor” computacionalmente. La evaluación parcial es una técnica de transformación de programas que consiste en la especialización de programas respecto a ciertos datos de entrada, conocidos en tiempo de compilación [73]. La idea es llevar a cabo el mayor número de computaciones posibles en tiempo de compilación, haciendo uso de los datos conocidos, de forma que se obtenga la mejor de las especializaciones posibles. Esto se consigue maximizando la *propagación de la información*. Se espera que el programa resultante pueda ser ejecutado de forma más eficiente ya que, usando el conjunto de datos (parcialmente) conocidos, es posible evitar algunas computaciones (en tiempo de ejecución) que se realizarán (una única vez) durante el proceso de compilación. En general, las técnicas de evaluación parcial incluyen algún criterio de parada para garantizar la terminación del proceso [144]. La evaluación parcial es, por tanto, una técnica de transformación de programas que pone mayor énfasis en los métodos puramente automáticos que las técnicas de transformación de programas tradicionales.

En cuanto a la fundamentación teórica de la evaluación parcial, el problema principal es determinar las condiciones bajo las cuales la técnica de evaluación parcial empleada es correcta y completa. Desde el punto de vista de la semántica operacional, *corrección fuerte* significa que las respuestas computadas por el programa evaluado parcialmente, P', son respuestas computadas también por el programa original, P. Hablamos de *corrección débil* cuando para cada respuesta computada por el programa especializado P' existe una respuesta computada más general en P. La *completitud* (*fuerte/débil*) se corresponde con los conceptos inversos complementarios de los anteriores.

Las transformaciones de plegado y desplegado, definidas por Burstall y Darlington en [48] para programas funcionales, son explotadas en mayor o menor medida por las distintas técnicas de evaluación parcial. El desplegado consiste en el reemplazamiento de una llamada a función por su respectiva definición (aplicando la correspondiente substitución). El plegado es la transformación inversa, es decir, el reemplazamiento de cierto fragmento de código por la correspondiente llamada a función. Para programas funcionales, los pasos de plegado y desplegado sólo involucran ajuste de patrones. En el caso de los programas lógicos, el ajuste de patrones se substituye por la unificación, obteniéndose así mayor potencia de propagación de información.

Una de las principales razones para el interés en la evaluación parcial es el deseo de conseguir una compilación automática y eficiente por medio de programas generales y fácilmente parametrizables. Futamura (1971) es el primer investigador en considerar un evaluador parcial como un programa, además de un transformador, sugiriendo la posibilidad de una autoaplicación [73]. En este artículo Futamura propone las ecuaciones para la compilación y la generación de compiladores (una autoaplicación) usando la evaluación parcial, pero no la ecuación para la generación de generadores de compiladores (doble autoaplicación). Andrei Ershov, que descubrió independientemente la primera de las ecuaciones y la tercera ecuación en 1976, denominó estas ecuaciones *Proyecciones de Futamura* [73, 67].

Aparte de los beneficios comentados que supone la evaluación parcial, disponer de un evaluador parcial como herramienta tiene aplicaciones inmediatas a la hora de facilitar el uso de un lenguaje (declarativo). De todos es sabido que cuando se realiza una especificación sencilla de un problema es más fácil establecer su significado declarativo (correcto); por contra, la ejecución de esta especificación como programa puede resultar ineficiente. Un evaluador parcial puede limar esta dificultad, facilitando y haciendo más eficiente el desarrollo de programas: la idea consistiría en confeccionar una biblioteca de plantillas genéricas y simples (cuyo significado declarativo fuese, sin duda, el esperado) que posteriormente se especializarían de forma automática para producir un código optimizado (i.e. más eficiente); la corrección del proceso de evaluación parcial asegura la corrección semántica del programa especializado. Disponer de una biblioteca de plantillas genéricas, para las tareas más comunes, también mejoraría la reusabilidad del código. Por consiguiente, las ventajas prácticas de la técnica de la evaluación parcial están fuera de toda duda.

1.4 Objetivos y Estructura de la Tesis.

1.4.1 Objetivos de la Tesis.

De forma concisa podemos decir que el objetivo principal de esta tesis ha sido investigar las técnicas y requisitos que permiten optimizar la especialización de programas lógico–funcionales, extendiendo y mejorando el marco genérico para la evaluación parcial establecido en [14]. Debido a que el proceso de desplegado de los objetivos se realiza mediante *narrowing*, decimos que este método de evaluación parcial está guiado por *narrowing* y nos referimos a él mediante el acrónimo NPE (del inglés *Narrowing-driven Partial Evaluation*). Para lograr los fines propuestos, se han desarrollado dos líneas de trabajo consistentes en:

- `Mejorar el mecanismo de base`.
  Haciendo uso de estrategias más eficientes como el *narrowing* perezoso y que aumentan la capacidad expresiva de los lenguajes al permitirse el empleo de programas no terminantes y estructuras de datos infinitas. En esta línea hemos comenzado por identificar una clase particular de programas que permiten optimizar la estrategia de *narrowing* perezoso. El objetivo, en este caso, es alcanzar una mayor eficiencia tanto en el proceso de evaluación parcial como en la mejora de la eficiencia que se consigue al ejecutar el programa especializado resultante.

- `Introducir técnicas avanzadas de evaluación parcial`.
  Mejorando los mecanismos de control del algoritmo de evaluación parcial. Esto se consigue mediante la introducción de una regla de desplegado que propicia un desplegado equilibrado de los árboles de *narrowing* local y un operador de abstracción potente que nos permita alcanzar una especialización poligenética[12] sin el empleo de artificios *ad hoc*; todo esto sin afectar a la terminación del proceso de NPE. El objetivo de estas técnicas es conseguir una propagación adecuada de la información positiva (i.e., basada en la unificación) y, por lo tanto, una mejora en la especialización del programa evaluado parcialmente.

[12]Entendemos por *especialización poligenética* la capacidad de especializar una llamada a función que combine varias llamadas a función del programa original; para más información ver el Apartado 3.1.

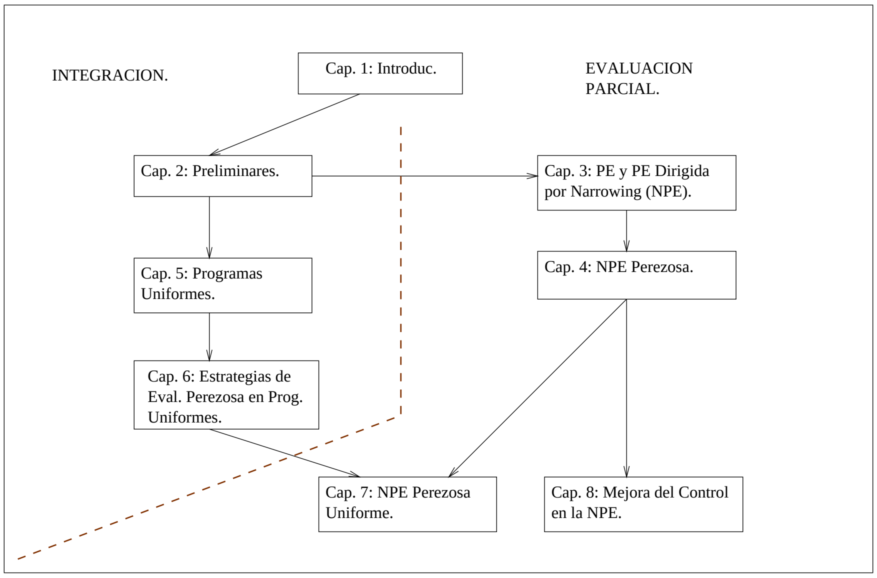

Figura 1.1: Relaciones de dependencia entre los capítulos.

Debido a los claros fundamentos semánticos de los lenguajes lógico–funcionales, éstos ofrecen ventajas significativas cuando se quiere probar formalmente la corrección de las técnicas de especialización. La demostración formal de la corrección de las nuevas técnicas introducidas ha sido un objetivo primordial de esta tesis. Además, se ha querido comprobar la efectividad de las técnicas introducidas para el caso concreto de (fragmentos de) lenguajes lógico–funcionales modernos, con semántica no estricta, como BABEL [157], Curry [96] o $\mathcal{TOY}$ [49]. Con este fin, se han incorporado las nuevas técnicas desarrolladas al prototipo de evaluador parcial para lenguajes lógico–funcionales, desarrollado en la Universidad Politécnica de Valencia, denominado Indy (*Integrated Narrowing-Driven specialization sYstem*) [4] y experimentado con éste, consiguiendo resultados positivos en la mayoría de los programas utilizados en las pruebas.

Finalmente, comentar un objetivo menor: hacer que esta tesis sea un trabajo autocontenido, en la medida de lo posible, y útil a aquellas personas que se enfrentan por vez primera al estudio de la integración de los lenguajes declarativos y al problema de su especialización. En la primera parte de esta tesis se ha realizado un esfuerzo importante para presentar de una manera accesible y rigurosa los conceptos y resultados muchas veces dispersos en la literatura que forman la base de estas materias y que son necesarios para elaborar el resto de la tesis. Naturalmente, la linea expositiva refleja una preferencia personal del autor; sin embargo otras perspectivas también serían posibles.

1.4.2 Estructura de la Tesis.

Esta tesis se compone de 9 capítulos, incluida la presente introducción y el capítulo de conclusiones, así como varios apéndices. La Figura 1.1 muestra el grafo con las relaciones de dependencia existentes entre los capítulos. Los capítulos se han dividido en tres partes:

- **Parte I**: Fundamentos.

  Esta parte presenta el material de base que será empleado en el resto de los capítulos. Si bien, el Apartado 2.9, contrariamente a lo que podría pensarse, contiene también algunas de las primeras aportaciones de esta tesis.

  El Capítulo 2 introduce definiciones, resultados y notaciones sobre conjuntos, relaciones, relaciones de orden, sistemas de reescritura de términos y lógica ecuacional. En él se resumen los principales conceptos sobre la programación lógico–funcional, centrándonos en las estrategias de *narrowing* perezoso y de *narrowing* necesario, así como en las clases de programas para los cuales son completas estas estrategias. Aquí introducimos una formalización precisa de la estrategia de *narrowing* peresozo que nos permitirá una comparación formal con la estrategia de *narrowing* necesario. También demostramos una serie de resultados sobre la naturaleza de las respuestas computadas por la estrategia de *narrowing* perezoso, que son útiles en el desarrollo posterior del trabajo. El Capítulo 3 comienza definiendo el concepto de evaluación parcial y los problemas más relevantes asociados con este tema, para continuar con un resumen de los conceptos y técnicas introducidas en [14] para la especialización de programas lógico–funcionales y que denominamos método NPE.

- **Parte II**: Especialización de programas lógico–funcionales perezosos.

  En esta parte se estudia la adaptación del método NPE para su empleo con una estrategia de *narrowing* perezoso, así como los problemas que esto plantea.

  El Capítulo 4 formula una instancia del algoritmo genérico de NPE basada en el empleo de *narrowing* perezoso. El proceso de evaluación parcial se formaliza en dos fases. En la primera fase se aplica una instancia concreta del método de NPE y a continuación se introduce una fase de renombramiento que es necesario para conseguir recuperar la disciplina de constructores del programa original. El postproceso de renombramiento también es necesario para conseguir la 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. Demostramos la corrección y completitud débil del proceso y discutimos las dificultades para lograr la corrección y completitud fuerte. Desde un punto de vista práctico, ponemos de relieve que, en algunos casos, la introducción de pasos de normalización es beneficiosa para el proceso de especialización. Finalmente, comprobamos que el método de especialización propuesto es capaz de pasar el test conocido como KMP [114, 108]. En el Capítulo 5 caracterizamos una clase de programas que definimos a partir de la idea de los programas uniformes de [120] y [121], estudiamos sus propiedades y demostramos que son una subclase de los programas inductivamente secuenciales, la clase de programas sobre la que resulta completa la estrategia de *narrowing* necesario. En el Capítulo 6 presentamos un estudio formal y exhaustivo de la relación precisa que existe entre la estrategia de *narrowing* necesario y la estrategia de *narrowing* perezoso sobre la clase de los programas uniformes. Demostramos la equivalencia existente entre ambas estrategias en esta clase particular de programas, que entendemos como la clase de programas más amplia para la cual ambas estrategias computan los mismos resultados y respuestas. Definimos un refinamiento de la estrategia de *narrowing* perezoso que llamamos *narrowing* perezoso uniforme y que resulta ser correcta y completa sobre la clase de los programas uniformes. Estos resultados encuentran aplicación inmediata en la optimización de las técnicas de evaluación parcial para programas lógico–funcionales con semántica no estricta. Apoyándonos en ellos, en el Capítulo 7, introducimos una nueva instancia del marco genérico de NPE usando *narrowing* perezoso uniforme y para la cual demostramos su corrección y completitud fuerte. Terminamos este capítulo estudiando la viabilidad práctica del método mediante la realización de una serie de pruebas experimentales.

- **Parte III**: Técnicas avanzadas de especialización.

  En esta parte se desarrollan técnicas avanzadas de especialización, definiendo una regla de desplegado dinámica y un operador de abstracción que permiten obtener resultados análogos a los de la *deducción parcial conjuntiva* [83, 128], la técnica más sofisticada y potente de especialización de programas lógicos.

  En el Capítulo 8 se introduce una regla de desplegado dinámica, que propicia un desplegado equilibrado de los árboles de *narrowing* local, evitando perder muchas oportunidades de especialización que la regla de desplegado formulada en [14], basada en un criterio de terminación muy rígido, no es capaz de aprovechar. Demostramos la terminación del algoritmo de desplegado definido. En la segunda parte del capítulo se estudia la mejora del control global del método de NPE. Para lograr esta mejora, se define un operador de abstracción concreto en el que se da un tratamiento especial a los símbolos primitivos del lenguaje, como la conjunción, y se utilizan técnicas de partición de los términos evaluados parcialmente para impedir la posibilidad de una generalización excesiva. Estas técnicas de partición son similares a las que se proponen en [83] y en [128], y nos permiten alcanzar una especialización poligenética sin el empleo de artificios *ad hoc*. Demostramos que la introducción del nuevo operador de abstracción no afecta a la terminación del proceso de NPE. Finalmente, realizamos una serie de experiencias con Indy con el fin de comprobar que la incorporación de estas técnicas en el algoritmo genérico de NPE son viables y en la práctica producen una mejora en la especialización del programa evaluado parcialmente. De esta forma se obtiene un marco de evaluación parcial más general que los considerados hasta la fecha y que engloba (con los ajustes adecuados) las técnicas de supercompilación positiva y de deducción parcial conjuntiva de manera natural.

Esta memoria finaliza con el Capítulo 9 de conclusiones donde se resumen las principales aportaciones de esta tesis y se esbozan algunas líneas de trabajo futuro.

Se incluyen, además, varios apéndices que contienen: información sobre el sistema INDY, el código de los programas utilizados en los experimentos y las llamadas evaluadas parcialmente.

1.4.3 Fuentes Bibliográficas de los Capítulos.

La amplia lista de referencias bibliográficas suministrada al final de esta memoria sugiere una multiplicidad de fuentes. Si bien esto es así, en este apartado se pretende poner de relieve aquellas fuentes de las que el autor de esta memoria, por diversos motivos, es especialmente deudor.

- Capítulo 2: En lo que se refiere a las nociones matemáticas básicas se ha empleado [179], si bien algunas ideas en cuanto a la organización del material se han extraido de [185]. En lo relativo a los sistemas de reescritura se ha seguido esencialmente el informe técnico de Klop [112], con algunos aportes de [59] y [29]. El material referente a estrategias de reescritura se ha extraido de [141] y de [142]. La idea conductora del apartado sobre razonamiento ecuacional está inspirada en [168].

  En lo referente a la estrategia de *narrowing* perezoso, si bien la formalización precisa de la estrategia es de nuestra cosecha [8, 2], el punto de partida han sido los trabajos del grupo de Mario Rodríguez Artalejo, especialmente [120] y [157]. La formalización de la estrategia de *narrowing* perezoso, en su estado actual, aparecio por vez primera en [8], donde también se presentan algunos resultados originales sobre el comportamiento de dicha estrategia. En lo referente a la estrategia de *narrowing* necesario, la fuente primordial ha sido [22], y para la caracterización declarativa de los árboles definicionales se ha empleado [21] y [19]; algunos resultados sobre la estrategia de *narrowing* necesario se tomaron de [17]. La totalidad del Apartado 2.9 se ha extraido de [9], un extenso informe técnico pendiente de publicación y que se escribió en paralelo con esta memoria de tesis.

- Capítulo 3: Para el concepto de evaluación parcial y los problemas más relevantes asociados con este tema, se ha utilizado principalmente [108], aunque también se tomaron ideas de [87, 126] y de [165]. En lo referente a la especialización de programas lógico–funcionales se ha resumido parte de la información que aparece en [14].

- Capítulo 4: La mayor parte del material de este capítulo se ha extraido de [8]. Para la parte dedicada al indeterminismo de renombramiento se ha empleado material sin publicar. Para ilustrar las dificultades que aparecen en la NPE basada en el uso del *narrowing* perezoso, se ha utilizado material procedente de [17].

- Capítulo 5 y Capítulo 6: La idea de los programas uniformes debe de rastrearse en el campo de la programación funcional (ver [167] para algunas referencias). Una primera aplicación de la noción de programa uniforme a la programación lógico–funcional se encuentra en [64] y en [65], si bien aquí se ha partido de los trabajos [120] y [121]. El material incluido en estos capítulo es original, aunque podría considerarse una continuación natural de los trabajos emprendidos por los autores de [120, 121] y [205] para la obtención de una estrategia de evaluación perezosa eficiente. La casi totalidad del material incluido en estos capítulos se ha extraido de [9].

- Capítulo 7: La totalidad del capítulo se ha extraido de [9], si exceptuamos la parte de resultados experimentales que es nueva.

- Capítulo 8: Aquí se ha partido de las técnicas desarrolladas en [83] y en [128]. Se han adaptado y extendiendo las definiciones que aparecen en [8] y en [14], de forma que sea posible tratar los problemas de control que plantea para la NPE el empleo de funciones primitivas en un entorno de programación lógico–funcional perezosa. Las adaptaciones necesarias son las que impone el tratar con una clase de lenguajes y una técnica de transformación básica (el *narrowing*) diferentes a los utilizados en los trabajos sobre deducción parcial, que hacen que el problema no sea en ningún modo trivial. La totalidad del material de este capítulo se ha extraido de [2].

1.5 Una Nota sobre el Estilo Narrativo.

El lector ya habrá notado que algunos términos técnicos, provenientes de la literatura inglesa dominante, no han sido traducidos. Por ejemplo, se ha decidido mantener la denominación "*narrowing*" ya que la traducción castellana habitual, "*estrechamiento*", nos parece un tanto forzada. En el futuro seguiremos el criterio de mantener la denominación inglesa, cuando la traducción de un término no sea completamente natural y clara. También, por claridad mantendremos los acrónimos con las siglas inglesas. Por ejemplo, escribiremos "hnf" que todo el mundo del área de la programación lógico–funcional identificará fácilmente como el acrónimo de "forma normal en cabeza", en lugar de escribir "fnc", que pensamos que sólo crearía confusión. Por otra parte, cuando el uso de la traducción literal de una voz inglesa se haya generalizado como término técnico en castellano, mantendremos esa denominación. Por ejemplo, somos plenamente conscientes de que la traducción de "instance" por "instancia" no es correcta, ya que las acepciones de esta palabra que pueden encontrarse en los diccionarios de lengua castellana no hacen referencia al significado de la palabra inglesa, que es el de "concreción o ejemplo que ilustra un enunciado general". Sin embargo, en la literatura técnica en castellano, el término "instancia" es bastante estándar. De hecho, la enciclopedia Larousse, en una de sus últimas ediciones, introduce la siguiente acepción de instancia:

– Lóg. Instancia de una fórmula $\forall x \varphi(x)$, resultado, simbolizado $\varphi(t/x)$, de la sustitución por un término t de la variable x.

Como puede apreciarse al leer el Apartado 1.4.3, muchos de los capítulos de esta tesis han sido adaptados a partir de trabajos y articulos escritos en colaboración con otras personas, por este motivo en multiples puntos de esta memoria aparece la palabra "nosotros" o los verbos están conjugados en primera persona del plural. Esto no debe verse como un gusto del autor por el uso del plural mayestático, sino más bien como un indicio de cierta pereza a la hora de cambiar el estilo del discurso narrativo. También denota un hábito que el autor no ha sabido corregir y espera que el lector sabrá disculpar.

Parte I

Fundamentos.

Capítulo 2

Preliminares.

En este capítulo presentamos las nociones matemáticas básicas que emplearemos en el resto de este trabajo. En la primera parte del capítulo introducimos algunas notaciones y resultados muy conocidos sobre conjuntos, relaciones y relaciones de orden. Se espera que estos conceptos sean familiares al lector por lo que no se pretende dar una visión exhaustiva de los mismos. Más bien nuestra intención es resumir brevemente las principales nociones, así como fijar una terminología. Una introducción asequible a estos temas puede encontrarse en [179], algunas ideas se han extraido de [185]. En la segunda parte del capítulo se introduce la terminología y simbología relativas a signaturas y términos de primer orden. Continuamos con un resumen sobre los sistemas de reescritura de términos [29, 59, 112] y la relación de éstos con la lógica ecuacional. La mayoría de los conceptos de esta parte se han extraido de [112]. Finalizamos con una introducción de la programación lógico–funcional, haciendo énfasis en el estudio de las estrategias de *narrowing* perezoso. Aunque por simplicidad las definiciones se presentan para una signatura homogenea, la extensión de las mismas al caso de signaturas heterogeneas ("*many sorted*") es inmediata [163].

2.1 Conjuntos, Relaciones y Funciones.

Un *conjunto* es una colección de objetos. Los objetos que forman un conjunto se denominan elementos o miembros del conjunto. Escribimos $e \in C$ para indicar la pertenencia de un elemento e al conjunto C, y escribimos $e \notin C$ para indicar el hecho contrario. El conjunto que no contiene elementos se denomina *conjunto vacío*, y se denota mediante el símbolo "$\emptyset$". Es habitual describir los conjuntos listando sus elementos entre llaves "{" y "}" (e.g., $\{a, b, c, \ldots, z\}$). Otra forma de describir los conjuntos es asociándoles alguna propiedad mediante el *operador de construcción* de conjuntos "|" (e.g., $\{e \mid e \text{ es una letra minúscula del alfabeto}\}$). Por convención, cualquier objeto puede ocurrir como mucho una vez en un conjunto, de forma que un conjunto no contiene elementos duplicados. El orden en el que aparecen los objetos tampoco cuenta. Dado un conjunto finito C, escribimos $|C|$ para denotar la *cardinalidad* del conjunto, i.e., el número de elementos que contiene el conjunto C.

Denotamos el conjunto de los números naturales $\{0, 1, 2, \ldots\}$ mediante el símbolo $\mathbb{N}$. El segmento inicial $\{1, 2, \ldots k\}$ del conjunto de los números naturales positivos $\mathbb{N}^+$ se denota $\mathbb{N}_k^+$. Cuando $k = 0$, entonces $\mathbb{N}_0^+ = \emptyset$. El conjunto de los números reales se denota por $\mathbb{R}$.

Decimos que S es un *subconjunto* de un conjunto C o que S está *incluido* en C y escribimos $S \subseteq C$ si todo elemento de S es un elemento de C. Empleamos la notación $S \not\subseteq C$ para indicar que S no está incluido en C. Decimos que un conjunto A es *igual* a otro B y escribimos $A = B$, si $A \subseteq B$ y $B \subseteq A$; i.e., A y B contienen los mismos elementos. Escribimos $A \neq B$ si no se cumple que $A = B$. Finalmente, escribimos $S \subset C$ para indicar que $S \subseteq C$, pero $S \neq C$; en este caso decimos que S es un *subconjunto propio* de C.

Emplearemos los símbolos usuales "$\cup$", "$\cap$", "$\setminus$" para denotar la unión, intersección y diferencia de conjuntos, respectivamente. Si $A \cap B = \emptyset$, decimos que A y B son *conjuntos disjuntos*. Escribiremos $A \uplus B$ para representar el conjunto $A \cup B$ cuando deseemos poner de manifiesto que se obtiene por unión de conjuntos disjuntos A y B (esto es, $A \cap B = \emptyset$). Dado un conjunto C, $\wp(C)$ denota el conjunto de todos los subconjuntos de C. El conjunto $\wp(C)$ se denomina el *conjunto potencia* de C o el *conjunto de las partes* de C.

Las operaciones anteriores pueden entenderse como medios para la construcción de conjuntos a partir de otros. Otro medio para la construcción de conjuntos es el producto cartesiano de conjuntos. La idea de producto cartesiano descansa sobre el concepto de n-tupla (ordenada). Una *n-tupla* es una colección ordenada de elementos, que denotamos por $\langle a_1, a_2, \ldots, a_n \rangle$. Decimos que a_1 es el primer elemento, a_2 es el segundo elemento, ..., y a_n es el n-ésimo elemento de la *n-tupla*. En una n-tupla el orden sí importa, esto quiere decir que $\langle a_1, a_2, \ldots, a_n \rangle = \langle b_1, b_2, \ldots, b_n \rangle$ si y sólo si $a_i = b_i$, para todo $i = 1, 2, \ldots, n$. El *producto cartesiano* de los conjuntos $A_1, A_2, \ldots, A_n$, que denotamos por $A_1 \times A_2 \times \ldots \times A_n$, es el conjunto de n-tuplas $\{\langle a_1, a_2, \ldots, a_n \rangle \mid a_i \in A_i, \text{ para todo } i = 1, 2, \ldots, n\}$. Denotamos por A^n el producto cartesiano formado por las n-tuplas de elementos pertenecientes a un mismo conjunto A.

Una *relación* sobre los conjuntos $A_1, A_2, \ldots, A_n$ es un subconjunto del producto cartesiano $A_1 \times A_2 \times \ldots \times A_n$. También se habla de relación *n-aria* . Dada una relación 2-aria, R, decimos que R es una *relación binaria* entre A_1 y A_2. Al conjunto A_1 se le denomina el *dominio* de la relación y al conjunto A_2 su *rango* o *codominio*. A menudo, utilizaremos la notación $a\ R\ b$ en lugar de $\langle a, b \rangle \in R$ para indicar que a está relacionado con b y $a\ \not R\ b$ en lugar de $\langle a, b \rangle \notin R$. La inversa de una relación binaria R es la relación $R^{-1} = \{\langle b, a \rangle \mid a\ R\ b\}$.

Una clase de relaciones de interés especial son las funciones. Dados los conjuntos A y B, una *función* $f : A \to B$ es una relación $f \subseteq A \times B$ que verifica las siguientes condiciones[1]: (1) para todo $a \in A$, existe un $b \in B$ tal que $\langle a, b \rangle \in f$ y (2) si $\langle a, b \rangle, \langle a, b' \rangle \in f$, entonces $b = b'$. Cuando $\langle a, b \rangle \in f$,

[1]El concepto de función que manejamos aquí se denomina a veces *aplicación* o *función total* en la literatura.

escribimos $f(a)$ para referirnos al elemento b y decimos que b es la *imagen* de a. Si no se cumple la condición (1), se dice que f es una función *parcial*. Si no se cumple (2), se dice que f es una función *indeterminista*. Por lo general, hablaremos simplemente de 'funciones', dejando que el contexto del discurso aclare, en su caso, los matices específicos que puedan ser relevantes. Dada una función $f : A \to B$ y un subconjunto $C \subseteq A$, escribimos $f_{\upharpoonright(C)}$ para denotar la *restricción* $f_{\upharpoonright(C)} : C \to B$ de la función f al subconjunto C: $f_{\upharpoonright(C)}(x) = f(x)$ para todo $x \in C$. Dadas dos funciones $g : B \to C$ y $f : A \to B$, la *composición* de ambas funciones se denota como $g \circ f : A \to C$ y se define de manera que $g \circ f(x) = g(f(x))$.

Una función f es *inyectiva* (o *uno a uno*) si para todo $a, a' \in A$, $f(a) = f(a')$ implica que $a = a'$. Una función f es *sobreyectiva* si para todo $b \in B$ existe un $a \in A$ tal que $f(a) = b$. Una función *biyectiva* (o correspondencia *biunívoca*) es una función inyectiva y sobreyectiva. Si $f : A \to B$ es biyectiva entonces la función $f^{-1} : B \to A$ definida por $f^{-1}(b) = a$ si y sólo si $f(a) = b$ se denomina función *inversa* de f.

Una función $f : A \to A$ es *idempotente* si para todo $a \in A$, $f(f(a)) = f(a)$. Un elemento $a \in A$ se denomina un *punto fijo* de f si $f(a) = a$. La función $id_A : A \to A$ definida por $id_A(a) = a$ para todo $a \in A$ se denomina la función *identidad* para el conjunto A. Dada $f : A \to A$, se define la *potencia* n-ésima f^n de f como sigue:

$$f^0 = id_A \text{ y } f^{n+1} = f \circ f^n \text{ para } n \geq 0.$$

A veces emplearemos el mismo símbolo f asociado a una función $f : A \to B$ para denotar su *extensión* a subconjuntos $C \subseteq A$ del dominio de f: $f(C) = \{f(a) \mid a \in C\}$. El conjunto $f(C) \subseteq B$ se denomina conjunto *imagen* del subconjunto C.

Un conjunto C se dice *finito* si existe una biyección entre el conjunto C y un segmento inicial $\mathbb{N}_n^+$ de los números naturales positivos y se dice que es *infinito enumerable* si existe una biyección entre C y $\mathbb{N}^+$. En general, emplearemos la palabra "infinito" para referirnos a conjuntos enumerables, a no ser que se especifique otra cosa.

Una *secuencia finita* s de elementos tomados de un conjunto C es una función $s : \mathbb{N}_n^+ \to C$, y una *secuencia infinita* s es una función cuyo dominio en el conjunto de los naturales positivos; i.e., $s : \mathbb{N}^+ \to C$. Emplearemos el término "secuencia" para referirnos indistintamente tanto a las del primer tipo como a las del segundo. Nos referiremos al n-ésimo elemento de la secuencia escribiendo s_n en lugar de $s(n)$, por consiguiente, también, escribiremos $s_1, \ldots, s_n$ para denotar una secuencia finita y $s_1, s_2 \ldots$ para denotar una secuencia infinita.

Una clase de relaciones muy importantes son las que se establecen entre los elementos de un mismo conjunto, por lo que cobra especial interés estudiar sus propiedades. Sea A un conjunto y $R \subseteq A \times A$ una relación binaria sobre A. Decimos que R es: *reflexiva*, si para todo $a \in A$, $a\ R\ a$; *irreflexiva*, si para todo $a \in A$, $a \not R\ a$; *transitiva*, si para todo $a, b, c \in A$, $a\ R\ b$ y $b\ R\ c$ implican $a\ R\ c$; *antisimétrica*, cuando $a\ R\ b$ y $b\ R\ a$ implican $a = b$; *simétrica*, cuando $a\ R\ b$ implica $b\ R\ a$.

Dado un conjunto A incluido en un conjunto E, y una propiedad P sobre los elementos de E, el *cierre* de A respecto a P es el menor conjunto incluido en E que contiene A y satisface P. Denotamos como $R^=$ el *cierre reflexivo* de la relación R: $R^= = R \cup D$; donde $D = \{\langle a, a \rangle \mid a \in A\}$ es la relación diagonal (o identidad) sobre el conjunto A. El *cierre simetrico* de R es la relación $R \cup R^{-1}$. El *cierre transitivo* de R es la relación R^+ tal que $a\ R^+\ b$ si existe una secuencia $a \equiv s_1, \ldots, s_n \equiv b$, con $n \geq 1$ y $s_i\ R\ s_{i+1}$ para todo $i = 1, \ldots, n-1$. El *cierre reflexivo y transitivo* de R es la relación $R^* = R^= \cup R^+$. El cierre reflexivo simétrico y transitivo de una relación R, denotado por $=_R$, es el cierre reflexivo y transitivo del cierre simétrico, i.e., $=_R = (R \cup R^{-1})^*$.

Si la relación $R \subseteq A \times A$ tiene las propiedades reflexiva, transitiva y simétrica, se dice que es una *relación de equivalencia* sobre A. El subconjunto $[a]_R = \{b \mid b \in A \wedge a\ R\ b\}$ es la *clase de equivalencia* de a definida por R. El conjunto formado por las clases de equivalencia definidas por R, denotado por A/R, se denomina *conjunto cociente* de la relación de equivalencia R sobre A. El conjunto cociente A/R es una partición del conjunto A. Dada una relación arbitraria R sobre A, el cierre reflexivo simétrico y transitivo de R se llama la *relación de equivalencia inducida por* R.

Dada una relación $R \subseteq A \times A$ y un subconjunto $B \subseteq A$, decimos que B es *cerrado bajo* R si, para todo $a \in B$, siempre que $a\ R\ b$, se tiene que $b \in B$. En particular, si $f : A \to A$ es una función, $B \subseteq A$ es cerrado bajo f si $f(B) \subseteq B$.

Dadas dos relaciones R, S sobre A, denotamos como $R \circ S$ la relación sobre A obtenida por *composición de* de ambas: para todo $a, b \in A$, $a\ R \circ S\ b$ si y sólo si existe un $c \in A$ tal que $a\ R\ c$ y $c\ S\ b$. Dados dos conjuntos A, B y relaciones binarias $R \subseteq A \times A$ y $S \subseteq B \times B$, una función $h : A \to B$ es un *homomorfismo* entre las relaciones R y S si para todo $a, b \in A$, $a\ R\ b$ implica que $h(a)\ S\ h(b)$.

2.2 Conjuntos Ordenados.

Una relación binaria $\preceq$ sobre A es una relación de *orden parcial* si es reflexiva, antisimétrica y transitiva. Escribimos $a \prec b$ para indicar que $a \preceq b \wedge a \neq b$. Escribimos $b \succeq a$ para expresar que $a \preceq b$, y $a \parallel b$ para indicar que $a \not\preceq b$ y $b \not\preceq a$. Si $\preceq$ es una relación de orden sobre A, decimos que $\langle A, \preceq \rangle$ (o simplemente A) es un *conjunto ordenado*. La relación $\preceq$ es un orden total si para todo $a, b \in A$, se tiene $a \preceq b$ o bien $b \preceq a$. Una relación $\prec$ irreflexiva y transitiva es un *orden estricto* y $\langle A, \prec \rangle$ un conjunto *estrictamente ordenado*. Un orden $\langle A, \preceq \rangle$ se dice *bien fundado* (*well-founded*) si no existen secuencias infinitas $a_1, \ldots, a_n, \ldots$ tales que $a_{i+1} \prec a_i$ para todo $i \geq 1$.

Un elemento $a \in A$ tal que para todo $b \in A$ siempre que $b \succeq a$ ($b \preceq a$) entonces $b = a$ se denomina elemento *maximal* (*minimal*) de A. Un elemento $a \in A$ es una *cota superior* (*inferior*) de $B \subseteq A$ si para todo $b \in B$, $b \preceq a$ ($a \preceq b$). Decimos que el conjunto B está *acotado superiormente* (*inferiormente*) si B tiene una cota superior (inferior). Una cota superior (inferior) $a \in A$ es un *supremo* (*ínfimo*) de $B \subseteq A$ si, para toda cota superior (inferior) $a' \in A$ de B, se tiene $a \preceq a'$ ($a' \preceq a$). Si $B \subseteq A$ tiene un supremo se denota a éste como $\bigsqcup B$.

Si el supremo (ínfimo) de un subconjunto $B \subseteq A$ pertenece a B, se le denomina el *máximo* (*mínimo*) de B. Si el propio conjunto A tiene un máximo (mínimo) se denota como $\top_A$ ($\perp_A$). Una secuencia (posiblemente infinita) $a_1, a_2, \ldots a_n \ldots$ de elementos de A es una *cadena* si $a_1 \preceq a_2 \preceq \cdots \preceq a_n \preceq \cdots$.

Un tipo de relaciones más débiles que las relaciones de orden, que emplearemos ampliamente en este trabajo, son los preórdenes. Una relación binaria $\trianglelefteq$ sobre A es una relación de *preorden* si es reflexiva y transitiva. Un conjunto A dotado de una relación de preorden $\trianglelefteq$ se dice *preordenado* y al par $\langle A, \trianglelefteq\rangle$ se le denomina *preorden* (*quasi-order*). Hay métodos estándares para obtener relaciones de equivalencia y de orden estricto a partir de un preorden. La relación de equivalencia $\sim$ generada por un preorden $\trianglelefteq$ se define como: $\sim = \trianglelefteq \cap \trianglelefteq^{-1}$; la relación de orden estricto $\triangleleft$ como: $\triangleleft = \trianglelefteq \setminus \trianglelefteq^{-1}$; y la relación de orden estricto $\triangleright$ como: $\triangleright = \trianglelefteq^{-1} \setminus \trianglelefteq$. Un preorden parcial $\langle A, \trianglelefteq\rangle$ se dice que es un conjunto *bien preordenado* (*well quasi-ordered*) si y sólo si para cualquier secuencia infinita de elementos $a_1, a_2, \ldots$ de A, existen números $i < j$ tales que $a_i \trianglelefteq a_j$. También se dice que la relación $\trianglelefteq$ es un *buen* preorden (*well quasi-order*) sobre A. Por consiguiente, si $\langle A, \trianglelefteq\rangle$ está bien preordenado nunca existirán cadenas decrecientes infinitas $a_1 \triangleright a_2 \triangleright \cdots \triangleright a_n \triangleright \cdots$.

2.3 Multiconjuntos.

Los conjuntos no contienen multiples ocurrencias de un mismo elemento. Un *multiconjunto* (*multiset*) es un "conjunto" donde se tiene en cuenta la multiplicidad de ocurrencias de un elemento. Más formalmente, dado un conjunto S, un *multiconjunto* sobre S es una función $f : S \to I\!N$. Los elementos de $D_f = \{s \in S \mid f(s) \neq 0\}$ son los *elementos* de f. Para cada $s \in D_f$, el número $f(s)$ se llama la *multiplicidad* de s en f. Es habitual representar un multiconjunto f, encerrando entre dobles llaves los elementos de D_f, repetidos tantas veces como indica su multiplicidad. Por ejemplo, si para f tenemos que $D_f = \{\{1,2\}\} \subset I\!N$, $f(1) = 3$ y $f(2) = 2$, escribiremos que $f = \{\{1,1,1,2,2\}\}$. Esta notación conjuntista se corresponde con la noción intuitiva de multiconjunto y nos permite utilizar las operaciones de unión, intersección, substración, etc. de conjuntos. Estamos particularmente interesados en los multiconjuntos finitos de números naturales, y denotamos por $M(I\!N)$ el conjunto formado por todos los multiconjuntos finitos sobre $I\!N$. El *orden multiconjunto*, $<_{mul}$, es un orden sobre $M(I\!N)$.

Definición 2.3.1 (Orden Multiconjunto)
Sea $M, M' \in M(I\!N)$, decimos que $M <_{mul} M'$ si y sólo si existe un $X \subseteq M$ y $X' \subseteq M'$ tal que $M = (M' \setminus X') \cup X$ y $\forall n \in X,\ \exists n' \in X'.\ n < n'$, donde $<$ es el orden estricto habitual sobre $I\!N$.

En palabras, $M <_{mul} M'$ si el multiconjunto M puede obtenerse a partir de M' reemplazando repetidamente un elemento $n' \in M'$ por cero o más elementos $n \in M$ con $n < n'$. El orden multiconjunto sobre $M(I\!N)$ es un orden bien fundado y, por lo tanto, se puede aplicar el principio de inducción completa al realizar pruebas en las que interviene $M(\mathbb{N})$. Emplearemos el orden multiconjunto en diversas pruebas sobre terminación del proceso de evaluación parcial.

2.4 Arboles.

Emplearemos árboles para representar términos, espacios de búsqueda, etc. Parece conveniente definir con precisión estos objetos y dotarnos de una nomenclatura.

Definición 2.4.1
Un árbol T es una estructura $\langle N, \succ, \Lambda\rangle$, donde N es un conjunto de nodos, *$\succ$ es una relación irreflexiva y Λ es un elemento distinguido de N llamado la* raíz *de T. Los nodos de T cumplen las siguientes restricciones:*

1. *Λ es el elemento máximo del conjunto estrictamente ordenado $\langle N, \succ^+\rangle$.*
2. *Para todo $k' \in N$ (salvo Λ), existe un único $k \in N$ tal que $k \succ k'$. Se dice que k es el* padre *de k' y que k' es el* hijo *de k. El par ordenado $\langle k, k'\rangle$ se denomina* arco *de T.*

Los nodos minimales de $\langle N, \succ^+\rangle$ reciben el nombre de *hojas*. Una *rama* es una secuencia $k_1, k_2, \ldots$ de nodos tal que $k_1 = \Lambda$ y $k_i \succ k_{i+1}$ para todo índice $i \in \{1, 2, \ldots\}$. La relación $\succ^*$ se denomina relación de descendencia y la $(\succ^{-1})^*$ de ascendencia. Si $k \succ^* k'$ decimos que k es un *ancestro* de k' o bien que k' es un *descendiente* de k.

Dado un nodo k de un árbol T, un subárbol de T con raíz k es el árbol $T' = \langle N', \succ', k\rangle$, donde $N' = \{k' \mid k \succ^* k'\}$ y $\succ' = \succ_{\upharpoonright N'}$. Un *árbol etiquetado* es un árbol dotado de un conjunto L de etiquetas y una función $f_\lambda : N \to L$ que asocia a cada nodo una etiqueta.

Un árbol $T = \langle N, \succ, \Lambda\rangle$ se dice *finito* (*infinito*) si el conjunto N es finito (infinito). Se dice que un árbol está *ramificado finitamente* (*finitely branching*) si cada uno de sus nodos tiene un número finito de hijos. El siguiente resultado es interesante cuando se trabaja con árboles infinitos ramificados finitamente.

Lema 2.4.2 (Lema de König) *Todo árbol infinito ramificado finitamente tiene una rama infinita.*

2.5 Relaciones de Reducción.

Ciertas propiedades básicas de los sistemas de reescritura de términos pueden ilustrarse mediante sistemas abstractos consistentes en un conjunto equipado con una o más relaciones binarias. Estos sistemas reciben en [112] el nombre de *sistemas de reducción abstractos*.

Definición 2.5.1
Un sistema de reducción abstracto *es un par $\langle A, R\rangle$ donde A es un conjunto y R una relación binaria sobre A, que se denomina* relación de reducción.

Una *secuencia de reducción* finita (de longitud n) asociada a una relación R sobre un conjunto A es una secuencia $a_1, a_2, \ldots, a_{n+1}$ de $n+1$ elementos de A tal que, para todo $1 \leq i \leq n$, $a_i \; R \; a_{i+1}$. Se dice que a_{n+1} es un R-reducto[2] de a_1 y que $a_i \; R \; a_{i+1}$ es un *paso de reducción*. También se dice que la anterior es una secuencia de reducción de n pasos.

Una relación binaria R sobre A es *confluente* si, para todo $a, b, c \in A$, siempre que $a \; R^* b$ y $a \; R^* c$, existe un $d \in A$ tal que $b \; R^* d$ y $c \; R^* d$ [101, 178]. De forma análoga, R es *localmente confluente* si, para todo $a, b, c \in A$, cuando $a \; R \; b$ y $a \; R \; c$, existe un $d \in A$ tal que $b \; R^* d$ y $c \; R^* d$. Decimos que R es *fuertemente confluente* si, para todo $a, b, c \in A$, $a \; R \; b$ y $a \; R \; c$ implican que $b \; R^* d$ y $c \; R^= d$ para algún $d \in A$.

Lema 2.5.2 [101] *Sean $R, S \subseteq A \times A$ relaciones tales que $R^+ = S^+$. Si S es fuertemente confluente, entonces R es confluente.*

Una consecuencia inmediata del lema anterior es que toda relación fuertemente confluente es confluente.

Un elemento $\bar{a} \in A$ se denomina una *forma R-normal* si no existe ningún $b \in A$ tal que $\bar{a} \; R \; b$. Decimos que $\bar{a}$ es una forma R-normal de a, si $\bar{a}$ es una forma R-normal y $a \; R^* \bar{a}$. Cuando una relación es confluente, la forma R-normal de un elemento a si existe es *única*.

Decimos que R es *terminante* [101] (*strongly normalizing* [112]) si y sólo si no existe ninguna secuencia infinita $a_0 \; R \; a_1 \; R \; a_2 \cdots$.

Lema 2.5.3 (Lema de Newman [158]**)** *Una relación terminante R sobre un conjunto A es confluente si y sólo si es localmente confluente.*

Cuando una relación es terminante, cada elemento $a \in A$ tiene (al menos) una forma normal. Una propiedad más débil es la de *normalización*, que admite secuencias de reducción infinitas, pero mantiene la existencia de formas normales. Decimos que una relación R sobre A es *normalizante* (*weakly normalizing* [112]) si todo elemento $a \in A$ tiene una forma normal. Evidentemente, una relación terminante es normalizante. Cuando una relación es confluente y normalizante, la forma normal de un elemento existe y es *única*.

2.6 Signaturas y Términos.

En lo que sigue, $\mathcal{X}$ denota un conjunto infinito numerable de *variables* y $\mathcal{F}$ una *signatura*, es decir, un conjunto finito de *símbolos de función*, cada uno de los cuales está dotado de una aridad previamente fijada por una función $ar_{\mathcal{F}} : \mathcal{F} \to \mathbb{N}$ (o simplemente ar, cuando ello no suponga confusión). La función $ar_{\mathcal{F}}$ asocia a cada función el número de argumentos sobre los que actúa. Denotaremos por $\mathcal{F}^n$ el conjunto de las funciones de $\mathcal{F}$ cuya aridad es n; i.e., $\mathcal{F}^n = \{f \mid f \in \mathcal{F} \wedge$

[2]Cuando la relación R involucrada queda clara a partir del contexto discursivo, hablaremos simplemente de reducto. Lo mismo se aplica para el resto de los conceptos definidos en este apartado.

$ar(f) = n\}$. Los símbolos de función con aridad cero se denominan *constantes*. Habitualmente, emplearemos las últimas letras mínusculas del alfabeto latino, w, x, y y z para denotar variables[3], las letras $f, g, h, \ldots$ para denotar funciones de aridad distinta de cero y las primeas letras mínusculas del alfabeto latino, a, b, y c para denotar constantes. Cuando se crea conveniente, se emplearan subíndices para evitar colisiones de nombres. Supondremos que $\mathcal{F} \cap \mathcal{X} = \emptyset$.

Las variables y los símbolos de función forman (una parte de) nuestro vocabulario. De las cadenas de símbolos que pueden formarse con el vocabulario, estamos interesados en aquellas que constituyen *términos* o *expresiones* del lenguaje.

Definición 2.6.1 (Término de Primer Orden)
Un término de primer orden *es una expresión, en la que intervienen variables, constantes y símbolos de función, que se define inductivamente de acuerdo a las siguientes reglas:*

1. *Si $x \in \mathcal{X}$, entonces x es un término.*
2. *Si $c \in \mathcal{F}$ es una constante, entonces c es un término.*
3. *Si $t_1, t_2, \ldots t_n$ son términos y $f \in \mathcal{F}$ es un símbolo de función n-ario, entonces $f(t_1, t_2, \ldots t_n)$ es un término.*

El conjunto de todos los términos se denota como $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

En general, $\mathcal{V}ar(s)$ denota el conjunto de las variables que aparecen en el objeto sintáctico s. Así pues, dado un término t, $\mathcal{V}ar(t)$ es el conjunto de los símbolos de variable presentes en t. Si $\mathcal{V}ar(t) = \emptyset$, decimos que t es un término *básico* (*ground*); $\mathcal{T}(\mathcal{F})$ denota el conjunto de los términos básicos. La identidad de objetos sintácticos se denota mediante el operador relacional "$\equiv$".

2.6.1 Ocurrencias.

Una *ocurrencia* u es una cadena de números enteros positivos $u \in Occ = \{\Lambda\} \cup \{i.v \mid i \in \mathbb{N}^+ \wedge v \in Occ\}$, donde Λ denota la ocurrencia correspondiente a la cadena vacía y '.' denota la concatenación de símbolos. La longitud de una cadena u se escribe $|u|$. Si u es una ocurrencia y W un conjunto de ocurrencias, $u.W$ es el conjunto $\{u.v \mid v \in W\}$. En general, dados dos conjuntos de ocurrencias U y W denotamos por $U.W$ el conjunto $\{u.v \mid u \in U \ \wedge \ v \in W\}$ resultado de concatenar todas las ocurrencias de U con las de W. Establecemos un orden $\leq$ entre ocurrencias empleando el *orden de prefijos* entre cadenas: $u \leq v$ si y sólo si existe una cadena w tal que $v = u.w$. Escribiremos $u \parallel v$ para indicar que u y v no pueden ser comparadas mediante el orden $\leq$; diremos entonces que u y v son ocurrencias disjuntas.

[3] En los programas de ejemplo emplearemos las correspondientes letras mayúsculas para denotar variables.

2.6.2 Substituciones.

Una *substitución* es una función $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$. Normalmente se exige que $\sigma(x) \neq x$ sólo para un número finito de variables. Al conjunto (finito) $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ de dichas variables se le denomina el *dominio* de la substitución. El *codominio* o *rango* de una substitución σ es el conjunto $\mathcal{R}an(\sigma) = \{\sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. Es común representar una substitución σ como un conjunto, haciendo explícitos los *enlaces* elementales x_i/t_i para $1 \leq i \leq n$, con $t_i \equiv \sigma(x_i)$ correspondientes a las variables $\{x_1, \ldots, x_n\}$ del dominio $\mathcal{D}om(\sigma)$, escribiendo $\sigma = \{x_1/t_1, \ldots, x_n/t_n\}$. Los enlaces $(x/t) \in \sigma$ también se denominarán *elementos* de la substitución. Si bien seguimos el criterio habitual de denotar las substituciones mediante letras griegas, hacemos una excepción al denotar la substitución vacía o identidad mediante el símbolo id. Tendremos que $\mathcal{D}om(id) = \emptyset$ o, equivalentemente, $id(x) = x$ para toda $x \in \mathcal{X}$. Una substitución σ se denomina *básica* si para todo $x \in \mathcal{D}om(\sigma)$, $\sigma(x)$ es un término básico. Una substitución ρ es un *renombramiento* si existe ρ^{-1} tal que $\rho \circ \rho^{-1} = \rho^{-1} \circ \rho = id$.

Las substituciones pueden extenderse fácilmente al dominio de los términos.

Definición 2.6.2
Dada una substitución $\sigma : \mathcal{X} \to \mathcal{T}(\mathcal{F}, \mathcal{X})$, su extensión $\vartheta : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{T}(\mathcal{F}, \mathcal{X})$, queda definida inductivamente mediante las siguientes reglas:

$$\vartheta(t) = \begin{cases} \sigma(x) & \texttt{si } (t \in \mathcal{X} \wedge t \equiv x \wedge x \in \mathcal{D}om(\sigma)); \\ z & \texttt{si } (t \in \mathcal{X} \wedge t \equiv z \wedge z \notin \mathcal{D}om(\sigma)); \\ c & \texttt{si } (t \in \mathcal{F} \wedge t \equiv c \wedge ar(c) = 0); \\ f(\vartheta(t_1), \ldots, \vartheta(t_n)) & \texttt{si } (t \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \wedge t \equiv f(t_1, \ldots, t_n) \\ & \quad \wedge\ t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \wedge i \in \{1, \ldots, n\}). \end{cases}$$

Es habitual denotar la extensión de una substitución σ al conjunto de los términos mediante el mismo símbolo σ con el que se denota la substitución original. En lo que sigue se seguirá este convenio. Denotamos el conjunto de las substituciones por $Subst(\mathcal{F}, \mathcal{X})$.

Se define el preorden $\leq$ de *máxima generalidad* sobre el conjunto $Subst(\mathcal{F}, \mathcal{X})$ de substituciones, como sigue: $\theta \leq \sigma$ si y sólo si existe una substitución γ tal que $\sigma = \gamma \circ \theta$. Decimos que θ es *más general* que σ. La *restricción*, $\sigma_{\upharpoonright(V)}$, de una substitución σ a un conjunto de variables $V \subseteq \mathcal{X}$ se define como $\sigma_{\upharpoonright(V)}(x) = \sigma(x)$ si $x \in V$ and $\sigma_{\upharpoonright(V)}(x) = x$ si $x \notin V$. Dado un subconjunto de variables $W \subseteq \mathcal{X}$, escribiremos $\sigma = \sigma'[W]$ si $\sigma_{\upharpoonright(W)} = \sigma'_{\upharpoonright(W)}$ y $\sigma \leq \sigma'[W]$ si existe una substitución σ'' tal que $\sigma'' \circ \sigma = \sigma'[W]$.

Dadas dos substituciones $\sigma, \sigma' \in Subst(\mathcal{F}, \mathcal{X})$ tales que $\mathcal{D}om(\sigma) \cap \mathcal{D}om(\sigma') = \emptyset$, definimos la *unión* $\sigma \cup \sigma'$ de ambas substituciones como $\sigma \cup \sigma'(x) = \sigma(x)$ si $x \in \mathcal{D}om(\sigma)$ y $\sigma \cup \sigma'(x) = \sigma'(x)$ si $x \in \mathcal{D}om(\sigma')$.

El preorden de máxima generalidad sobre el conjunto $Subst(\mathcal{F}, \mathcal{X})$ de las substituciones induce un preorden parcial sobre los términos. Un término t es *más general* que otro s (o también, s es una *instancia* de t), en símbolos $t \leq s$, si $(\exists\sigma).\ s = \sigma(t)$. Dos términos t y t' son *variantes* si existe un renombramiento ρ tal que $t' = \rho(t)$. Un término t *empareja* (o *ajusta*) con un término l si t es una instancia de l. La substitución σ para la cual $t = \sigma(l)$ se denomina *substitución de emparejamiento* (o simplemente *emparejamiento*). Cuando $\mathcal{V}ar(t) \cap \mathcal{V}ar(l) = \emptyset$, el emparejamiento σ puede descomponerse como la unión $\sigma = \sigma_l \cup \sigma_t$ de substituciones σ_l y σ_t tales que $\mathcal{D}om(\sigma_l) = \mathcal{V}ar(l)$ y $\mathcal{D}om(\sigma_t) = \mathcal{V}ar(t)$. Dos términos t, s *unifican* si existe una substitución σ tal que $\sigma(t) = \sigma(s)$. A la substitución σ se le denomina un *unificador* de t y s. Si un unificador θ de los términos t y s verifica que $\theta \leq \sigma$ para cualquier otro unificador σ, decimos que θ es el *unificador más general* (mgu, del inglés *most general unifier*) de t y s. Un mgu es único salvo (composición con substituciones de) renombramiento. Una *generalización* de un conjunto de términos $\{t_1, \ldots, t_n\}$ es un par $\langle t, \{\theta_1, \ldots, \theta_n\}\rangle$ tal que para todo $i = 1, \ldots, n$, $\theta_i(t) = t_i$. Una generalización $\langle t, \Theta\rangle$ es la *generalización más específica* (msg, del inglés *most specific generalization*) de un conjunto S, escrito $msg(S)$, si para cualquier otra generalización $\langle t', \Theta'\rangle$ del conjunto S se cumple que $t' \leq t$, i.e., t' es un término más general que t. El msg de un conjunto de términos es único salvo renombramientos de variables [125].

2.6.3 Términos y Posiciones.

Los términos se representan como árboles etiquetados de la manera usual, siendo las etiquetas de los nodos los símbolos que forman el término. Para identificar un nodo constituyente de un término t empleamos ocurrencias, también denominadas *posiciones*[4].

Definición 2.6.3 (Posiciones)
El conjunto $\mathcal{P}os(t)$ de las posiciones de un término t se define inductivamente como:

$$\mathcal{P}os(t) = \begin{cases} \{\Lambda\} & \texttt{si } t \in \mathcal{X} \\ \{\Lambda\} \cup \{i.u \mid 1 \leq i \leq n \ \wedge\ u \in \mathcal{P}os(t_i)\} & \texttt{si } t \equiv f(t_1, \ldots, t_n) \end{cases}$$

Escribimos $t|_u$ para denotar el *subtérmino* de t en la posición $u \in \mathcal{P}os(t)$: $t|_\Lambda = t$ y $f(t_1, \ldots, t_k)|_{i.u} = t_i|_u$ donde $1 \leq i \leq k$. Escribimos $t[s]_u$ para referirnos al término donde el subtérmino en la posición u, $t|_u$, se ha *reemplazado* por el término s: $t[s]_\Lambda = s$ y $f(t_1, \ldots, t_i, \ldots, t_k)[s]_{i.u} = f(t_1, \ldots, t_i[s]_u, \ldots, t_k)$ donde $1 \leq i \leq k$. Escribimos $\mathcal{H}ead(t)$ para referirnos al símbolo que etiqueta la *raíz* del (árbol asociado al) término t: $\mathcal{H}ead(x) = x$ si $x \in \mathcal{X}$ y $\mathcal{H}ead(f(t_1, \ldots, t_k)) = f$ si $f \in \mathcal{F}$. También se dice que $\mathcal{H}ead(t)$ es el símbolo que *encabeza* el término t, o bien que $\mathcal{H}ead(t)$ es el *símbolo raíz* de t. También escribimos $t[u]$ para denotar la etiqueta asociada al árbol del término t en la posición $u \in \mathcal{P}os(t)$: $t[u] = \mathcal{H}ead(t|_u)$. Dados los términos t, s, escribimos $t \prec s$ para indicar que t es un subtérmino *propio* de s, es decir, que existe $u \in \mathcal{P}os(s)$, $u \neq \Lambda$ tal que $t = s|_u$.

Las posiciones en un término están ordenadas por el orden de prefijos definido en el Apartado 2.6.1. El orden $\leq$ sobre posiciones nos permite dar significado a

[4] En este contexto emplearemos con preferencia la denominación *posición*, ya que "ocurrencia" puede confundirse con el subtérmino que ella designa

expresiones como "*símbolo (o término) que se encuentra sobre o encima (debajo) de otro símbolo (o término)*": si t es un término y $u, v \in \mathcal{P}os(t)$ son tales que $u < v$, decimos que el símbolo $t[u]$ (el subtérmino $t|_u$) está sobre (por encima de) $t[v]$ ($t|_v$).

El conjunto $\mathcal{FP}os(t) = \{u \in \mathcal{P}os(t) \mid \mathcal{H}ead(t|_u) \in \mathcal{F}\}$ contiene las posiciones de símbolos *no variables* en t y $\mathcal{VP}os(t) = \mathcal{P}os(t) \setminus \mathcal{FP}os(t)$ es el conjunto de posiciones de variables. Escribimos $\mathcal{P}os_s(t)$ para denotar el conjunto de posiciones en las que ocurre el subtérmino s en t: $\mathcal{P}os_s(t) = \{u \in \mathcal{P}os(t) \mid t|_u = s\}$. Un término es *lineal* si no contiene múltiples ocurrencias de una misma variable.

Emplearemos, de forma auxiliar, las siguientes propiedades y resultados estándar [101]. Consideremos $t_1, t_2, t_3 \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

- Si $u \in \mathcal{P}os(t_1)$ y $v \in \mathcal{P}os(t_2)$ entonces
  - $(t_1[t_2]_u)|_{u.v} = t_2|_v$ (*inmersión*),
  - $t_1[t_2[t_3]_v]_u = t_1[t_2]_u[t_3]_{u.v}$ (*asociatividad*).
- Si $u, v \in \mathcal{P}os(t_1)$ y $u \parallel v$ tenemos que
  - $(t_1[t_2]_u)|_v = t_1|_v$ (*persistencia*) y
  - $t_1[t_2]_u[t_3]_v = t_1[t_3]_v[t_2]_u$ (*conmutatividad*).
- Si $u \leq v$, haciendo $v = u.w$ obtenemos
  - $t_1|_v = (t_1|_u)|_w$ (*cancelación*),
  - $t_1[t_2]_v[t_3]_u = t_1[t_3]_u$ (*dominancia*) y
  - $(t_1[t_2]_v)|_u = (t_1|_u)[t_2]_w$ (*distributividad*).

Proposición 2.6.4 [101] *Sea $\mathcal{F}$ una signatura, $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ y $\sigma \in Subst(\mathcal{F}, \mathcal{X})$. Entonces,*

$$\mathcal{P}os(\sigma(t)) = \mathcal{P}os(t) \cup \bigcup_{t|_u \in \mathcal{X}} \{u.v | v \in \mathcal{P}os(\sigma(t|_u))\}$$

y, por tanto, para toda posición $u \in \mathcal{P}os(t)$:

1. *Si $t|_u = t' \notin \mathcal{X}$, entonces $\sigma(t)|_u = \sigma(t')$;*
2. *Si $t|_u = x \in \mathcal{X}$, entonces $\sigma(t)|_{u.v} = \sigma(x)|_v$ para todo $v \in \mathcal{P}os(\sigma(x))$.*

Proposición 2.6.5 [101] *Sea $\mathcal{F}$ una signatura, t y s términos de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ y $\sigma \in Subst(\mathcal{F}, \mathcal{X})$. Entonces, para toda posición $u \in \mathcal{P}os(t)$, tenemos que $\sigma(t)[\sigma(s)]_u = \sigma(t[s]_u)$.*

Cuando no se precise un tratamiento riguroso de las posiciones utilizaremos el concepto de contexto. Un *contexto* C es un término con cero o más 'huecos', es decir, ocurrencias de un símbolo especial $\Box$ (un nuevo símbolo constante). El propio símbolo $\Box$ es un contexto; se le denomina el contexto *vacío*. Escribimos

$C[\ ]_u$ para poner de manifiesto que existe un hueco $\Box$ (normalmente único) situado sobre la posición u de C. Muchas veces escribiremos simplemente $C[\ ]$ para denotar un contexto arbitrario, dejando que el número y localización concreta de sus huecos sean clarificados en cada momento. Si $C[\ ]$ es un contexto arbitrario y t es un término, el resultado de reemplazar las ocurrencias de $\Box$ por t es el término $C[t]$; se dice que t es un *subtérmino* de $C[t]$.

2.7 Sistemas de Reescritura.

Una *regla de reescritura* es un par ordenado $\langle l, r \rangle$, escrito $l \to r$ (o bien $\alpha : l \to r$ si se desea asociar una etiqueta α a la regla), donde $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ y $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ [59, 112]. Dada una regla de reescritura $l \to r$, l se denomina la *parte izquierda* (lhs, del inglés, *left-hand side*) y r la *parte derecha* (rhs, del inglés, *right-hand side*) de la regla de reescritura. Un *Sistema de Reescritura de Términos*[5] (TRS, del inglés *Term Rewrite System*) es un par $\mathcal{R} = \langle \mathcal{F}, R \rangle$ donde R es un conjunto finito de reglas de reescritura. Se sobrentiende que R no contiene reglas que puedan derivarse de otras por renombramiento (una a una) de sus variables. Por abuso de lenguaje, en futuros capítulos, muchas veces identificaremos el TRS $\mathcal{R}$ con el conjunto de reglas R, sobrentendiendo que la signatura $\mathcal{F}$ está formada por el conjunto de símbolos de función que aparecen en dichas reglas.

El conjunto de las partes izquierdas de un TRS se denota como $L(\mathcal{R})$: $L(\mathcal{R}) = \{l \mid\ l \to r \in R\}$, mientras que $L_f(\mathcal{R}) = \{l \mid\ l \to r \in R \wedge \mathcal{H}ead(l) = f\}$ es el conjunto de las lhs's de las reglas que definen la función f en $\mathcal{R}$. Toda instancia $\sigma(l)$ de una lhs $l \in L(\mathcal{R})$ de una regla es un *redex* (del inglés *reducible expression*). El conjunto de posiciones de redexes en un término t es $\mathcal{P}os_{\mathcal{R}}(t)$.

Las reglas de reescritura de un TRS definen una relación de reducción $\to$ entre los términos.

Definición 2.7.1 (Relación de Reescritura)
Un término t se reescribe *al término s (en la posición p), escrito $t \to_{\mathcal{R}_{p,\alpha}} s$ (o simplemente $t \to_{p,\alpha} s$, cuando el TRS $\mathcal{R}$ se sobrentienda), si existe una posición $p \in \mathcal{P}os(t)$, una regla de reescritura $\alpha : l \to r$ y una substitución σ con $t|_p = \sigma(l)$ y $s = t[\sigma(r)]_p$.*

Decimos que $t|_p$ es el *redex* de t que se ha contraído (o reducido) y que $\sigma(r)$ es su *contracto*. También decimos que t se reduce a s en un *paso de reescritura* $t \to_{p,\alpha} s$. A veces escribiremos también $t \to_p s$ o $t \to s$ cuando no sea necesario especificar la regla y/o la posición que se utilizan para dar el paso. Puede demostrarse que la relación de reescritura es cerrada bajo la aplicación de contextos y substituciones. Esto es, si $t \to s$ entonces para todo contexto

[5]Dada una regla de reescritura $l \to r$, si no se cumple la condición $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, decimos que $l \to r$ es una regla de reescritura con *variables extra*, o también con *variables libres*. La noción estándar de TRS prohíbe la presencia de este tipo reglas. En este trabajo nos limitaremos al estudio de los TRS's compuestos por reglas sin variables extras.

C, posición p en ese contexto y para toda $\sigma \in Subst(\mathcal{F}, \mathcal{X})$, se cumple que $C[\sigma(t)]_p \to C[\sigma(s)]_p$.

El par $\langle \mathcal{T}(\mathcal{F}, \mathcal{X}), \to_\mathcal{R} \rangle$ es el sistema de reducción abstracto asociado al TRS $\mathcal{R}$. Una *secuencia* (o *derivación*) de reescritura (infinita) es una secuencia de reducción (infinita) asociada a la relación de reescritura $\to$. El cierre reflexivo y transitivo $\to^*$ de $\to$ se denomina *relación de derivabilidad* [6] en [59]. Si $t \to^* s$, decimos que t es *reducible* a un término s y que s es un *reducto* de t. El cierre simétrico de $\to$ se denota como $\leftrightarrow$. El cierre reflexivo, simétrico y transitivo de $\to$ se denomina *relación de convertibilidad* y se denota bien como $=_\mathcal{R}$ (siguiendo la notación introducida en la sección 2.1) o como $\overset{*}{\leftrightarrow}$. La relación $\downarrow$ (*joinability*) se define como sigue: $t \downarrow t'$ si existe un s tal que $t \to^* s$ y $t' \to^* s$.

2.7.1 Clases de Sistemas de Reescritura.

Un TRS $\mathcal{R}$ se llama confluente si $\to_\mathcal{R}$ es confluente. Un TRS se llama terminante si $\to_\mathcal{R}$ es terminante. Si un TRS es confluente y terminante, se denomina canónico[7].

Una regla $l' \to r'$ es un *renombramiento* de $l \to r$ si existe una substitución de renombramiento ρ tal que $l' = \rho(l)$ y $r' = \rho(r)$. Dadas dos reglas[8] $\alpha_1 : l_1 \to r_1$ y $\alpha_2 : l_2 \to r_2$ para las que existe una posición $p \in \mathcal{FP}os(l_1)$ tal que $l_1|_p$ y l_2 unifican con mgu σ, entonces el par de reductos $\langle \sigma(r_1), l_1[\sigma(r_2)]_p \rangle$ es un *par crítico* (excluimos el caso trivial $\alpha_1 = \alpha_2$ y cuando $p = \Lambda$). Un par crítico se denomina un *cubrimiento* (*overlay*) si $p = \Lambda$. Un par crítico $\langle t, s \rangle$ es *trivial* si $t \equiv s$. Un par crítico $\langle t, s \rangle$ es *convergente* si $t \downarrow s$.

Una regla $l \to r \in R$ *lineal por la izquierda* (*left-linear*), si l es un término lineal.

Las definiciones anteriores permiten caracterizar una clase importante de TRS's que goza de propiedades interesantes.

Definición 2.7.2 (TRS ortogonal)
Un TRS $\mathcal{R}$ se dice ortogonal *si cumple las siguientes restricciones sintácticas:*

1. Linealidad por la izquierda*: toda regla de $\mathcal{R}$ es lineal por la izquierda.*
2. No ambigüedad (fuerte)*: $\mathcal{R}$ no contiene pares críticos; i.e., las lhs de las reglas de $\mathcal{R}$ no* solapan.

La condición de no ambigüedad (fuerte) puede relajarse para dar lugar a clases, sucesivamente más amplias, de TRS's. Si todos los pares críticos de un TRS son cubrimientos triviales, éste se denomina *cuasi ortogonal* (*almost orthogonal*). Si un TRS sólo tiene pares críticos triviales se denomina *débilmente ortogonal* (*weakly orthogonal*) [113]. Si relajamos por completo la condición de

[6] Otros autores llaman a $\to$ la *relación de reescritura de un paso* para el TRS $\mathcal{R}$, dejando la denominación de *relación de reescritura* para la relación $\to^*$.

[7] Algunos autores utilizan 'convergente' [30, 59, 61, 110] (o 'completo' [112]) para referirse a los TRS's confluentes y terminantes y reservan la palabra 'canónico' para referirse a los TRS's convergentes con algunos requerimientos adicionales [30, 110].

[8] Que no comparten variables, para lo que pueden ser renombradas si es preciso

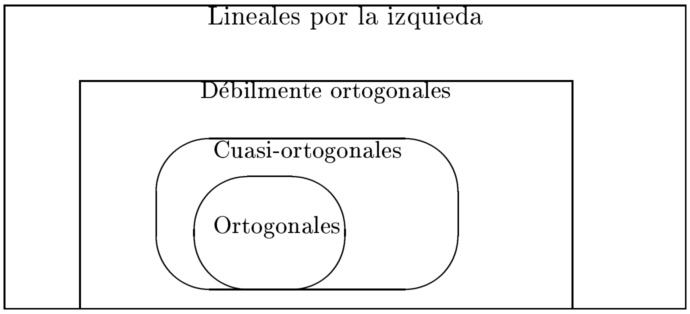

Figura 2.1: Una clasificación de los TRS's lineales por la izquierda.

no ambigüedad y permitimos la existencia de pares críticos obtenemos la clase de los TRS's lineales por la izquierda. i.e., un TRS $\mathcal{R}$ se dice *lineal por la izquierda* (*left-linear*), si para toda $l \in L(\mathcal{R})$, l es un término lineal. La Figura 2.1 ilustra las relaciones entre estas clases de TRS's.

Dado un TRS $\mathcal{R} = \langle \mathcal{F}, R \rangle$, siempre podemos establecer una partición $\mathcal{F} = \mathcal{C} \uplus \mathcal{D}$ de la signatura $\mathcal{F}$ en un conjunto $\mathcal{C}$ de símbolos *constructores* $c \in \mathcal{C}$ que no tienen ninguna regla de reescritura asociada y un conjunto $\mathcal{D}$ de símbolos de función *definidos* (u operaciones) $f \in \mathcal{D}$, que quedan definidos como sigue: $\mathcal{D} = \{f \in \mathcal{F} \mid (\exists\ l \in L(\mathcal{R})).\ \mathcal{H}ead(l) = f\}$ y $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. Denotamos por $\mathcal{T}(\mathcal{C}, \mathcal{X})$ el conjunto de los términos obtenidos empleando únicamente símbolos constructores y variables. Los términos $d \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ se denominan *términos constructores*. Un *patrón* es un término de la forma $f(d_1, \ldots, d_n)$ donde $f \in \mathcal{D}$ y los argumentos $d_1, \ldots, d_n$ son términos constructores o variables.

Al imponer a las reglas de reescritura la *disciplina de constructores* obtenemos una clase importante de TRS's, los TRS's *basados en constructores*. Una regla de reescritura $l \to r \in R$ de un TRS $\mathcal{R} = \langle \mathcal{F}, R \rangle = \langle \mathcal{C} \uplus \mathcal{D}, R \rangle$ cumple la disciplina de constructores si l es un patrón. Si todas las reglas de un TRS $\mathcal{R}$ cumplen la disciplina de constructores, se dice que $\mathcal{R}$ es un TRS basado en constructores (CB, del inglés, *Constructor Based*).

2.7.2 Formas Canónicas.

Un término t es una forma normal en cabeza si no existe niguna derivación $t = t_1 \to t_2 \to \cdots$ que reescriba t a un redex, es decir, tal que t_i es un redex para algún $i \geq 1$. En particular, llamaremos *forma normal en cabeza constructora* a todo término t que sea una variable o $\mathcal{H}ead(t) \in \mathcal{C}$. Reservamos el acrónimo hnf (del ingles "*head normal form*") para las formas normales en cabeza constructoras, dada la importancia que cobran en este trabajo. Las formas normales en cabeza han recibido el nombre de *formas estables* en [141]. En este sentido, un término es *estabilizable* si puede reducirse a una forma estable.

Un término t es *irreducible* o está en *forma normal* si no posee redexes, i.e., $\mathcal{P}os_{\mathcal{R}}(t) = \emptyset$; en caso contrario, decimos que t es *reducible*. Un término t es *normalizable* si posee una forma normal, i.e., si existe una reducción $t \to^* s$, donde s es una forma normal (si bien de él pueden partir secuencias de reducción infinitas). Se denota la forma normal de un término t mediante el símbolo $t\!\downarrow$. Un TRS es *normalizante* si todo término es normalizable.

El concepto de forma normal puede ponerse en combinación con el de substitución. Decimos que una substitución σ está *normalizada*, si para todo $x \in \mathcal{D}om(\sigma)$, $\sigma(x)$ es una forma normal.

2.7.3 Confluencia.

La confluencia es una propiedad muy importante de los TRS's porque asegura la unicidad de las formas normales cuando existen. En general, la confluencia de un TRS es una propiedad indecidible. La caracterización de las propiedades estructurales de los TRS's que permiten asegurar la confluencia de los mismos ha sido estudiada, entre otros, en [101, 102] y en [178].

La confluencia de un TRS puede asegurarse mediante el análisis de la convergencia de sus pares críticos.

Teorema 2.7.3 *[101] Un TRS $\mathcal{R}$ es localmente confluente si y sólo si todo par crítico es convergente.*

Una consecuencia inmediata del Teorema 2.7.3 y del Lema de Newman (Lema 2.5.3) es el siguiente teorema, que permite concluir la confluencia de un TRS.

Teorema 2.7.4 *[101] Un TRS $\mathcal{R}$ terminante es confluente si y sólo si todo par crítico es convergente.*

Otra forma de asegurar la confluencia consiste en imponer restricciones sintácticas a los TRS's que permitan demostrar la confluencia fuerte de la relación de *reescritura paralela*. En reescritura paralela se contraen simultáneamente un conjunto de posiciones disjuntas de redexes del término a reducir. Si $t \;\#\!\!\!\!\to s$ es un paso de reescritura paralela sobre un conjunto de posiciones $U = \{u_1, \ldots, u_n\}$, con $n \geq 0$, entonces existe una derivación $t \equiv t_1 \to_{u_1} t_2 \to_{u_2} \cdots \to_{u_n} t_{n+1} \equiv s$ (puesto que las posiciones de los redexes considerados en cada paso de una derivación son disjuntas, el orden de los mismos es irrelevante). Para los TRS's ortogonales, la confluencia fuerte de la relación de reescritura paralela viene dada por el llamado *parallel moves lemma* [102], que puede entenderse como un corolario (ver [112]) de la conocida propiedad de los TRS's ortogonales de que los *descendientes* de un redex continuan siendo redexes.

Definición 2.7.5 (Descendiente [102]**)**
Sea $\mathcal{R} = \langle \mathcal{F}, R \rangle$ un TRS ortogonal y $A : t \to_{[u,\alpha]} s$ un paso de reescritura, donde $\alpha : l \to r \in R$. Sea $v \in \mathcal{P}os_{\mathcal{R}}(t)$. El conjunto $v \backslash A$ de descendientes (o residuos*) del redex $t|_v$ a través de A es un subconjunto de $\mathcal{P}os(s)$ definido como sigue:*

$$v \backslash A = \begin{cases} \emptyset & \texttt{si } v = u \\ \{v\} & \texttt{si } (v \parallel u \;\vee\; v < u) \\ \{u.w_1.v_1 \mid r|_{w_1} = x\} & \texttt{si } (v = u.w.v_1 \wedge l|_w = x \in \mathcal{X}) \end{cases}$$

Podemos expresar la noción de descendiente de manera más intuitiva siguiendo la definición informal propuesta en [112, 113]: i) marcamos la raíz del redex $t|_v$ que ocupa la posición v que queremos analizar; ii) damos el paso $t \rightarrow_{[u,\alpha]} s$ para obtener s, manteniendo las marcas de los símbolos introducidas; iii) ahora, las posiciones p (respectivamente, los términos $t|_p$) que aparecen marcadas (marcados en su raíz) en el término s son las (los) descendientes de v ($t|_v$). Nótese que un residuo de un redex puede ser modificado por la acción de un paso de reescritura. La noción de descendiente se generaliza fácilmente a derivaciones de reescritura. Para cualquier derivación no elemental B y posición v de un redex en un término t, definimos el conjunto de descendientes de v a través de la derivación B, denotado $v \backslash\!\backslash B$, como sigue: $v \backslash\!\backslash B = \{v\}$, si B es la derivación vacía; y $v \backslash\!\backslash B = \bigcup_{w \in v \backslash A} w \backslash\!\backslash B'$, si $B = (AB')$, donde A es el paso inicial de B. El cálculo de los residuos $v \backslash A$ de una occurrencia redex v se extiende a conjuntos de posiciones disjuntas de redexes P como sigue: $P \backslash A = \bigcup_{v \in P} v \backslash A$.

A partir del *parallel moves lemma* y del hecho de que $\rightarrow^+ = \nrightarrow\!\!\!\!\!\parallel^+$, empleando el Lema 2.5.2, puede concluirse la confluencia de la relación $\rightarrow$ para un TRS ortogonal.

Teorema 2.7.6 *[102] Todo TRS ortogonal es confluente.*

El método anterior puede generalizarse y utilizarse para demostrar la confluencia de los TRS's débilmente ortogonales [112].

2.7.4 Estrategias de Reescritura.

La reescritura es un mecanismo que permite computar valores de expresiones, entendiendo por *valor* una *forma normal* de la expresión. En general, dado un término (básico) que se considera la expresión a evaluar, existen diferentes secuencias de reducción a partir de dicha expresión inicial. La elección del redex que se selecciona para explotarse en cada paso puede afectar no solamente a la eficiencia del cómputo, sino a la posibilidad de encontrar una forma normal, cuando estamos en presencia de un TRS no terminante. Salvo para clases restringidas de TRS's, como los sistemas canónicos, para los que cada término tiene una forma normal única que puede computarse mediante cualquier secuencia de reducción, si queremos asegurarnos de que se computará la forma normal de una expresión (si existe), estamos obligados a construir todas las posibles secuencias de reducción. Naturalmente, esto es computacionalmente inviable en muchos casos. Por consiguiente tenemos que dotarnos de alguna *estrategia* con la que encontrar los redexes que deben ser contraidos en cada paso para alcanzar el resultado deseado. Consideraremos dos tipos de estrategias (deterministas): las *secuenciales* o monopaso (que seleccionan un único redex para ser reducido en un paso de reescritura) y las paralelas (que seleccionan un conjunto de redexes disjuntos para ser contraídos simultáneamente).

Definición 2.7.7
Una estrategia de reescritura monopaso (multipaso) para un TRS $\mathcal{R} = \langle \mathcal{F}, R \rangle$ es una función $F_\mathcal{R} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X})$ que verifica

1. *$F_{\mathcal{R}}(t) = t$ si t es una forma normal y*

2. *$t \to_{\mathcal{R}} F_{\mathcal{R}}(t)$ ($t \to^{+}_{\mathcal{R}} F_{\mathcal{R}}(t)$) en otro caso.*

Si un término t que no es una forma normal verifica $s = F_{\mathcal{R}}(t)$ para algún término s, escribiremos $t \to_{F_{\mathcal{R}}} s$. Una secuencia de reducción $t_1 \to_{F_{\mathcal{R}}} t_2 \to_{F_{\mathcal{R}}} \cdots$ se denomina una $F_{\mathcal{R}}$-secuencia.

Una estrategia $F_{\mathcal{R}}$ es estabilizante (normalizante) si para todo término t estabilizable (normalizable) existe una $F_{\mathcal{R}}$-secuencia tal que para algún $i \geq 1$, $F^{i}_{\mathcal{R}}(t)$ es una forma estable (normal).

Cualquier estrategia de reescritura $F_{\mathcal{R}}$ para un TRS $\mathcal{R} = \langle \mathcal{F}, R \rangle$ puede descomponerse en dos componentes [25, 141]: $F_{\mathcal{R}} = \langle P_{\mathcal{R}}, R_{\mathcal{R}} \rangle$. El componente $P_{\mathcal{R}} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to [Occ]$ selecciona la lista de posiciones de redexes que deben reducirse (una lista unitaria cuando la estrategia es secuencial). El componente $R_{\mathcal{R}} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to [R]$ selecciona la lista de reglas que deben aplicarse [25]. Las funciones $P_{\mathcal{R}}$ y $R_{\mathcal{R}}$ están sometidas a las siguientes restricciones:

1. $P_{\mathcal{R}}(t) = [\,]$ si y sólo si $\mathcal{P}os_{\mathcal{R}}(t) = \emptyset$, es decir, la estrategia siempre reduce algo, salvo cuando t es una forma normal ($\mathcal{P}os_{\mathcal{R}}(t) = \emptyset$).

2. Si $P_{\mathcal{R}}(t) = [u_1, \ldots, u_m]$ entonces, asumiendo que $R_{\mathcal{R}}(t) = [\alpha_1, \ldots, \alpha_n]$, se tiene que $m = n$, $\{u_1, \ldots, u_n\} \subseteq \mathcal{P}os_{\mathcal{R}}(t)$, $u_i \parallel u_j$ para $1 \leq i, j \leq n$, $i \neq j$, y $t|_{u_i} = \sigma_i(l_i)$ para la regla $\alpha_i : l_i \to r_i$ cuando $1 \leq i \leq n$ y $F_{\mathcal{R}}(t) = t[\sigma_1(r_1)]_{u_1} \cdots [\sigma_n(r_n)]_{u_n}$.

Debido a su propiedad de no ambigüedad débil, para los TRS's débilmente ortogonales, la estrategia $F_{\mathcal{R}}$ viene determinada por completo por el componente $P_{\mathcal{R}}$ [25], aunque no de forma unívoca en el caso general. Sólo si el TRS $\mathcal{R}$ es ortogonal, la elección de un redex concreto por $P_{\mathcal{R}}$ fija una y sólo una regla seleccionada por $R_{\mathcal{R}}$ para reducir ese redex. En el caso de los TRS's cuasi ortogonales y débilmente ortogonales todavía persiste cierto indeterminismo, aunque se asegura que, cuando se dé la posibilidad de aplicar varias reglas, el término que reemplazará al redex estará unívocamente determinado, sea cual sea la regla seleccionada para reducirlo. Por tanto, el indeterminismo en la elección de una regla de reescritura puede romperse sin más que imponer un orden total arbitrario sobre el conjunto de las reglas R del TRS y escoger la primera regla de acuerdo con ese orden. Por consiguiente, cuando tratemos con TRS's (débilmente) ortogonales, entenderemos por estrategia de reescritura la restricción al componente $P_{\mathcal{R}}$ de $F_{\mathcal{R}}$. También supondremos que $P_{\mathcal{R}}$ devuelve conjuntos de posiciones en lugar de listas de posiciones, i.e., $P_{\mathcal{R}} : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \wp(Occ)$. Esta formulación de estrategia es más natural y conveniente para el contexto en el que se va a desarrollar el presente trabajo.

Una *posición necesaria* (*needed*) es aquella que señala la aparición de un *redex necesario*. Un redex en un término t es *necesario* si es contraído, él mismo o alguno de sus descendientes, en cualquier secuencia de normalización de t [102, 113]. El resultado fundamental establecido por Huet y Lévy es el siguiente.

Teorema 2.7.8 [102] *Sea $\mathcal{R}$ un TRS ortogonal. Todo término que no es una forma normal contiene un redex necesario para su normalización.*

Este teorema posibilita la definición de estrategias necesarias, es decir, que sólo reduzcan redexes necesarios. Las secuencias de reducción que sólo contraen redexes necesarios se denominan secuencias de *reducción necesaria*. Estas secuencias exhiben una propiedad importante.

Teorema 2.7.9 [102] *Sea $\mathcal{R}$ un TRS ortogonal y t un término normalizable. Toda secuencia de reducción necesaria que parta de t es normalizante.*

Por lo tanto, dado un término que posee una forma normal, la contracción repetida de sus redexes necesarios conducirá a esa forma normal. Lamentablemente dada una posición p de un redex en un término t, en general, es indecidible si $t|_p$ es un redex necesario para normalizar t o no. Sin embargo, Huet y Levy [102] demostraron que en una subclase de los TRS's ortogonales, los llamados TRS's *fuertemente secuenciales*, al menos uno de los redexes necesarios de un término puede computarse de modo efectivo. También demostraron que es decidible (aunque muy complicado) saber si un TRS es fuertemente secuencial. Así pues, la propiedad de secuencialidad fuerte garantiza la existencia de una estrategia (computable) de reducción secuencial normalizante. En [18] se ha introducido una estrategia eficiente para el cómputo de redexes necesarios, para la clase de los TRS's *inductivamente secuenciales* (ver apartado 2.9.4 para una definición de TRS inductivamente secuencial). Recientemente, se ha demostrado que la clase de los TRS's inductivamente secuenciales coincide con la clase de los TRS's CB fuertemente secuenciales [97].

Finalizamos este apartado destacando que las secuencias de reducción necesarias son óptimas, en el sentido de que no realizan pasos de reducción inútiles (en implementaciones basadas en reducción de grafos).

2.8 Sistemas de Reescritura y Razonamiento Ecuacional.

Como hemos mencionado, la *lógica ecuacional* proporciona un soporte sintáctico para la definición de funciones y el razonamiento ecuacional; de ahí su interés. En este apartado se introducen dos temas: la relación entre los TRS's y las teorías ecuacionales, y el problema de la unificación semántica.

Una *ecuación* es un par de términos $\langle s, t\rangle$, que representamos mediante la expresión $s = t$. Cuando los términos contienen variables, éstas se suponen cuantificadas universalmente. Dado un sistema de ecuaciones $\mathcal{E}$, una *teoría ecuacional* es el conjunto de ecuaciones que puede obtenerse por deducción utilizando el siguiente sistema de reglas de inferencia, en el que las ecuaciones de $\mathcal{E}$ son tomadas como axiomas.

Definición 2.8.1 (Reglas de Inferencia de la Lógica Ecuacional)
Sean s, t y r términos en $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Las reglas de inferencia básicas de la lógica ecuacional son:

1. Reflexiva	$\dfrac{}{t=t}$	
2. Simétrica	$\dfrac{s=t}{t=s}$	
3. Transitiva	$\dfrac{s=r, r=t}{s=t}$	
4. Substitución	$\dfrac{s_1=t_1,\ldots,s_n=t_n}{f(s_1,\ldots,s_n)=f(t_1,\ldots,t_n)}$	$(\forall f).\ (f \in \mathcal{F} \wedge ar(f) = n)$
5. Instanciación	$\dfrac{s=t}{\sigma(s)=\sigma(t)}$	$(\forall \sigma).\ \sigma \in Subst(\mathcal{F}, \mathcal{X})$
6. Ecuaciones	$\dfrac{}{s=t}$	$si\ s = t \in \mathcal{E}$

Por abuso de lenguaje, muchas veces hablamos de "la teoría ecuacional $\mathcal{E}$" para hacer referencia a la teoria ecuacional axiomatizada por $\mathcal{E}$. Si una ecuación $s = t$ pertenece a la teoría ecuacional $\mathcal{E}$, se debe a que es deducible a partir del conjunto $\mathcal{E}$, y escribimos $\mathcal{E} \vdash s = t$ o bien $s =_{\mathcal{E}} t$. Puede darse una definición más compacta del sistema de inferencia de la Definición 2.8.1 en términos de la noción de "reemplazamiento de iguales por iguales" (ver [59]). La noción de reemplazamiento conduce al concepto de congruencia. Una relación de equivalencia $\sim$ sobre $\mathcal{T}(\mathcal{F}, \mathcal{X})$ es una *congruencia* si para todo $f \in \mathcal{F}$, con $ar(f) = n$, se cumple que $f(s_1, \ldots, s_n) \sim f(t_1, \ldots, t_n)$ siempre que $s_i \sim t_i$ para $i = 1, \ldots, n$. Es fácil probar que la relación binaria $=_{\mathcal{E}}$ sobre $\mathcal{T}(\mathcal{F}, \mathcal{X})$, definida por el sistema de inferencia de la Definición 2.8.1, es la mínima congruencia sobre $\mathcal{T}(\mathcal{F}, \mathcal{X})$ que contiene al conjunto de ecuaciones $\mathcal{E}$ y es estable bajo substituciones. La relación $=_{\mathcal{E}}$ puede extenderse a substituciones como sigue: $\theta =_{\mathcal{E}} \sigma$ si y sólo si $\theta(x) =_{\mathcal{E}} \sigma(x)$ para todo $x \in \mathcal{X}$. También decimos que la substitución θ es más $\mathcal{E}$-general que la substitución σ, en símbolos $\theta \leq_{\mathcal{E}} \sigma$, si existe una substitución γ tal que $\gamma \circ \theta =_{\mathcal{E}} \sigma$. Definimos $\theta =_{\mathcal{E}} \sigma\ [W]$ y $\theta \leq_{\mathcal{E}} \sigma\ [W]$ igual que antes, pero restringiendo las substituciones al conjunto de variables $W \subseteq \mathcal{X}$.

Como es habitual en lógica, las construcciónes sintácticas de la lógica ecuacional cobran significado cuando se las interpreta en un dominio de discurso. Aquí el dominio de interpretación son las estructuras matemáticas denominadas álgebras. Dada una signatura $\mathcal{F}$, un *$\mathcal{F}$-álgebra* es un par $\langle A, \mathcal{F}_A \rangle$, donde A es un conjunto, denominado *soporte*, y $\mathcal{F}_A$ es un conjunto de operaciones tal que por cada símbolo de función $f \in \mathcal{F}$, existe una operación $f_A : A^{ar(f)} \to A$ en $\mathcal{F}_A$. Por abuso de lenguaje, muchas veces haremos referencia a un $\mathcal{F}$-álgebra $\langle A, \mathcal{F}_A \rangle$ mencionando únicamente el nombre del conjunto de soporte A. Interpretar en la lógica ecuacional consiste en seleccionar un $\mathcal{F}$-álgebra A y realizar una *asignación* de valor a las variables, i.e., definir una aplicación $\rho_A : \mathcal{X} \to A$. Una asignación puede extenderse al conjunto de los términos $\mathcal{T}(\mathcal{F}, \mathcal{X})$ dándoles significado en A. Una ecuación $s = t$ es *verdadera* en un $\mathcal{F}$-álgebra A si y sólo si, para toda asignación ρ_A, se cumple que $\rho_A(s) = \rho_A(t)$; los términos s y t tienen el mismo significado en A, i.e., representan el mismo elemento en A.

También decimos que la $\mathcal{F}$-álgebra A es *modelo* de la ecuación $t = s$ y escribimos $A \models t = s$. Un $\mathcal{F}$-álgebra A es *modelo* de un conjunto de ecuaciones $\mathcal{E}$ si es modelo de cada una de las ecuaciones que lo forman. Denotamos por $Mod(\mathcal{E})$ el conjunto de todas las $\mathcal{F}$-álgebras que son modelo de $\mathcal{E}$. Diremos que una ecuación $t = s$ es (*lógicamente*) *válida* si es verdadera en toda $\mathcal{F}$-álgebra $A \in Mod(\mathcal{E})$, y escribiremos $Mod(\mathcal{E}) \models t = s$. Un $\mathcal{F}$-álgebra de importancia singular es la denominada *álgebra de términos básicos* [9] $\langle \mathcal{T}(\mathcal{F}), \mathcal{F}_{\mathcal{T}(\mathcal{F})} \rangle$, en la cual $\mathcal{F}_{\mathcal{T}(\mathcal{F})} = \{f : \mathcal{T}(\mathcal{F})^{ar(f)} \to \mathcal{T}(\mathcal{F}) \mid f \in \mathcal{F}\}$. La importancia de este álgebra radica en su propiedad de ser inicial para la clase de todas las $\mathcal{F}$-álgebras. Por otra parte, el $\mathcal{F}$-álgebra cociente $(\mathcal{T}(\mathcal{F})/ =_{\mathcal{E}})$ es inicial en la clase $Mod(\mathcal{E})$ de todas las $\mathcal{F}$-álgebras que son modelo del conjunto de ecuaciones $\mathcal{E}$. Un álgebra inicial es un representante prototípico de una clase $\mathcal{C}lass$ de $\mathcal{F}$-álgebras que puede emplearse, en lugar de cualquier otra $\mathcal{F}$-álgebra $A \in \mathcal{C}lass$, para estudiar las propiedades de la clase $\mathcal{C}lass$. Formalmente, un $\mathcal{F}$-álgebra I es *inicial* en una clase $\mathcal{C}lass$ de $\mathcal{F}$-álgebras si y sólo si $I \in \mathcal{C}lass$ y, para toda $\mathcal{F}$-álgebra $A \in \mathcal{C}lass$, existe un único homomorfismo $h : I \to A$.

Un conocido teorema establecido por Birkhoff [40] pone en relación los conceptos semánticos de validez en $Mod(\mathcal{E})$ y deducibilidad. Dicho teorema enuncia que, para todo conjunto de ecuaciones $\mathcal{E}$ y términos s y t en $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $Mod(\mathcal{E}) \models (s = t)$ si y sólo si $\mathcal{E} \vdash (s = t)$. Alternativamente, este resultado también puede formularse en los siguientes términos.

Teorema 2.8.2 (Teorema de la Lógica Ecuacional) *Para todo conjunto de ecuaciones $\mathcal{E}$ se cumple:*

1. *(Corrección) para todo par de términos s y t en $\mathcal{T}(\mathcal{F}, \mathcal{X})$,*

$$\text{Si } \mathcal{E} \vdash (s = t) \text{ entonces } (\mathcal{T}(\mathcal{F})/ =_{\mathcal{E}}) \models (s = t);$$

2. *(Completitud) para todo par de términos s y t en $\mathcal{T}(\mathcal{F})$,*

$$\text{Si } (\mathcal{T}(\mathcal{F})/ =_{\mathcal{E}}) \models (s = t) \text{ entonces } \mathcal{E} \vdash (s = t).$$

Por lo tanto, el teorema de Birkhoff establece la equivalencia entre la posibilidad de deducir la igualdad de dos términos (sin variables) a partir de un sistema ecuacional y la verdad de la correspondiente ecuación en todos los modelos del sistema ecuacional (o, alternativamente, en el algebra inicial cociente).

Ahora nos gustaría relacionar los sistemas ecuacionales y los TRS's, con el fin de dotar a los primeros de un mecanismo de ejecución eficiente. La idea es automatizar la lógica ecuacional, es decir, queremos responder a preguntas de la clase[10]

[9]Llamada *universo de Herbrand* en otros contextos, como la demostración automática de teoremas y la programación lógica.

[10]Este problema recibe el nombre de "*word problem*" en la literatura inglesa. La restricción a términos básicos viene aconsejada por la correspondiente restricción impuesta en la dirección de la completitud en el teorema de Birkhoff. Cuando se elimina la restricción de considerar solamente términos básicos, el problema planteado se denomina "*problema de la validez*".

dada una teoría ecuacional $\mathcal{E}$ y una ecuación $s = t$ formada por términos básicos, ¿$Mod(\mathcal{E}) \models (s = t)$? (o equivalentemente, ¿$s =_{\mathcal{E}} t$?),

utilizando como mecanismo operacional la reducción de expresiones por reescritura. Dado que $=_{\mathcal{E}}$ es la mínima congruencia sobre $\mathcal{T}(\mathcal{F},\mathcal{X})$ que contiene el conjunto de ecuaciones $\mathcal{E}$ y es estable bajo substituciones, se puede comprobar, a partir de la Definición 2.7.1, que para el sistema de reescritura $\mathcal{R}^{\mathcal{E}}$, resultante de imponer cierta direccionalidad a las ecuaciones de $\mathcal{E}$, la relación de convertibilidad $\overset{*}{\leftrightarrow}_{\mathcal{R}^{\mathcal{E}}}$ y $=_{\mathcal{E}}$ coinciden [11]. Esta es ya una relación, sin embargo el hecho de que la reescritura puede ir en las dos direcciones en la derivación de $s \overset{*}{\leftrightarrow}_{\mathcal{R}^{\mathcal{E}}} t$ es una desventaja. La siguiente propiedad y un resultado establecido por Newman vienen en nuestra ayuda.

Definición 2.8.3 (Propiedad de Church-Rosser)
Un TRS $\mathcal{R}$ se dice que posee la propiedad de Church-Rosser *si se cumple que:*

Para todo s,t en $\mathcal{T}(\mathcal{F},\mathcal{X})$, $s \overset{*}{\leftrightarrow} t$ si y sólo si $s \downarrow t$.

Teorema 2.8.4 *Un TRS $\mathcal{R}$ posee la* propiedad de Church-Rosser *si y sólo si $\mathcal{R}$ es confluente.*

Así pues, si el sistema $\mathcal{R}^{\mathcal{E}}$ es canónico, deducir la igualdad de dos términos s y t (sin variables) a partir de un sistema ecuacional será equivalente a comprobar la igualdad sintáctica de las formas normales que se obtienen al reducir los términos s y t en $\mathcal{R}^{\mathcal{E}}$. Naturalmente, si el sistema $\mathcal{R}^{\mathcal{E}}$ es confluente, pero no terminante, el proceso puede no terminar si uno de los términos s o t no posee forma normal.

En el futuro no haremos distinción entre el sistema ecuacional $\mathcal{E}$ y el TRS $\mathcal{R}^{\mathcal{E}}$ que lo representa.

Un problema más amplio que el de la validez de una ecuación compuesta por términos básicos en una teoría ecuacional $\mathcal{E}$, es el problema de la *unificación semántica* o *$\mathcal{E}$-unificación*. Dados dos términos s y t, decimos que son *$\mathcal{E}$-unificables* si existe una substitución σ tal que $\sigma(s) =_{\mathcal{E}} \sigma(t)$. La substitución σ se denomina *$\mathcal{E}$-unificador* de s y t. Por abuso de lenguaje, generalmente a la substitución σ se le llama también *solución* de la ecuación $s = t$. La unificación semántica se corresponde con la unificación sintáctica cuando $\mathcal{E} = \emptyset$. En el contexto de una teoria ecuacional $\mathcal{E}$, el concepto de *mgu* se generaliza al de conjuntos completos de $\mathcal{E}$-unificadores. Un conjunto de substituciones Θ es un *conjunto completo* de $\mathcal{E}$-unificadores de dos términos s y t, si se satisfacen las siguientes condiciones:

[11] En [59], a los sistemas para los que se cumple que $\overset{*}{\leftrightarrow}_{\mathcal{R}^{\mathcal{E}}} = =_{\mathcal{E}}$ se les llama correctos y completos para $\mathcal{E}$. También en [59], se reserva la denominación de "completo" para los sistemas que cumplen los siguientes requisitos: a) finito (estos autores consideran la posibilidad de que los TRS's contengan un número infinito de reglas; cosa que nosotros excluimos por definición); b) terminante; c) Church-Rosser; y d) correcto y completo para $\mathcal{E}$. Nosotros supondremos que tratamos con TRS's que cumplen al menos las propiedades (a), (c) y (d)

- $\mathcal{D}om(\theta) \subseteq (\mathcal{V}ar(s) \cup \mathcal{V}ar(t))$, para toda $\theta \in \Theta$.
- cada $\theta \in \Theta$ es un $\mathcal{E}$-unificador de s y t.
- Si σ es un $\mathcal{E}$-unificador de s y t entonces existe una $\theta \in \Theta$ tal que $\theta \leq_{\mathcal{E}} \sigma\ [\mathcal{V}ar(s) \cup \mathcal{V}ar(t)]$.

Cada conjunto consistente en un *mgu* de los términos s y t es un conjunto completo de $\emptyset$-unificadores de s y t.

En el próximo apartado se introduce el *narrowing* como un algoritmo para la $\mathcal{E}$-unificación en el contexto de los TRS's. Allí estudiaremos las condiciones para las que este algoritmo es *completo* (i.e, si θ es un $\mathcal{E}$-unificador de s y t, entonces el *narrowing* encuentra al menos uno más $\mathcal{E}$-general).

2.9 Programación Lógico–Funcional.

Los lenguajes lógico–funcionales pueden considerarse como extensiones de los lenguajes funcionales con principios derivados de la programación lógica [174]. La mayoría de estos lenguajes integrados utilizan sistemas de reescritura como programas y alguna variante del *narrowing* como principio operacional. El *narrowing* es una generalización del mecanismo operacional de la *reescritura*, empleada en los lenguajes funcionales, con el fin de extender éstos con características de la programación lógica: variables lógicas, estructuras de datos parcialmente definidas y mecanismos de búsqueda deductiva no determinista. El *narrowing* extiende el mecanismo de la reescritura sustituyendo el ajuste de patrones por la unificación, de forma que ambos coinciden cuando se emplean sobre términos que no contienen variables. El *narrowing* proporciona completitud en el sentido de la programación lógica (computación de respuestas) así como también en el de la programación funcional (computación de *valores* o formas normales).

En los próximos apartados profundizaremos en los aspectos sintácticos del lenguaje que vamos a considerar y en los princios operacionales que lo rigen.

2.9.1 Programas.

La sintaxis del lenguaje lógico–funcional que utilizamos en la elaboración de este trabajo, esencialmente, es equivalente a la de (un subconjunto de) Babel [138, 157], $\mathcal{TOY}$ [49] o Curry [96]. Abstraemos las características sintácticas del lenguaje diciendo que los *programas* son TRS's basados en constructores y ortogonales. Como se argumentó en el Apartado 1.1.2, ésta es una clase de programas muy adecuada para la integración de los lenguajes lógicos y funcionales. Los programas con los que tratamos pueden ser no terminantes.

Uno de los intereres primarios de los lenguajes lógico–funcionales es la resolución de ecuaciones. Como ya hemos indicado, una *ecuación* es un par de términos $\langle s, t\rangle$. Con el fin de aumentar la expresividad del lenguaje, extendemos la signatura $\mathcal{F}$ con un conjunto $\mathcal{P}$ de símbolos de función primitivos. El conjunto $\mathcal{P}$ se define como $\mathcal{P} = \{\approx, \wedge, \Rightarrow\}$, lo que permite manipular expresiones complejas conteniendo ecuaciones, que ahora podremos expresar como un término $s \approx t$, conjunciones $b_1 \wedge b_2$, y expresiones condicionales (guardadas) $b \Rightarrow t$, que también expresamos como términos. Así pues, la signatura extendida es $\mathcal{F} = \mathcal{C} \uplus \mathcal{D} \uplus \mathcal{P}$. Usualmente tratamos los símbolos de $\mathcal{P}$ como operadores infijos. La semántica de estos símbolos primitivos viene determinada por el siguiente conjunto de reglas predefinidas, que denotamos por StrEq y que suponemos pertenecen a todo programa:

$$\begin{array}{rcl}
c \approx c & \to & \mathit{true} \quad \% \; c \in \mathcal{C}, ar(c) = 0 \\
c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) & \to & (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) \quad \% \; c \in \mathcal{C}, ar(c) = n \\
\mathit{true} \wedge x & \to & x \\
(\mathit{true} \Rightarrow x) & \to & x
\end{array}$$

Notad que las dos primeras "reglas" son en realidad *esquemas* de reglas, correspondiendo a cada esquema tantas "realizaciones" como símbolos constructores aparecen en $\mathcal{R}$. La extensión de un programa $\mathcal{R}$ con el conjunto de reglas StrEq se denota por $\mathcal{R}_+$, i.e., $\mathcal{R}_+ = (\mathcal{R} \cup \text{StrEq})$. Estas reglas son ortogonales y definen la validez de una ecuación como la *igualdad estricta* entre términos, lo cual es común en los lenguajes funcionales cuando los cómputos pueden no terminar [79, 138, 157]. La igualdad estricta no posee la propiedad reflexiva, i.e., no se cumple que $t \approx t$ para todo término t. La igualdad estricta trata dos términos s y t como idénticos si y solamente si tienen como forma normal el mismo término constructor básico, i.e., si $s \approx t$ se reduce a *true*. La equivalencia entre la reducibilidad al mismo término constructor básico y la reducibilidad a *true*, queda establecida en la siguiente proposición.

Proposición 2.9.1 *[22] Sea $\mathcal{R}$ un programa. Un par de términos s y t son reducibles por reescritura al mismo término constructor básico, utilizando las reglas de $\mathcal{R}$, si y solamente si $s \approx t$ se reescribe a true usando las reglas del programa $\mathcal{R}_+$.*

Notad que, aunque el modelo computacional básico solamente emplea reglas incondicionales, todavía es adecuado para soportar programas lógicos, ya que las reglas de reescritura condicionales $l \to r \Leftarrow C$ pueden simularse mediante reglas incondicionales con expresiones guardadas $l \to (C \Rightarrow r)$ utilizando el símbolo de función primitivo '$\Rightarrow$' al igual que en el lenguaje Babel [157].

Por razones de simplicidad, suponemos que el operador '$\wedge$' es asociativo por la derecha y que '$\approx$' tiene más precedencia (i.e., liga más) que '$\wedge$', y '$\wedge$' que '$\Rightarrow$'. Así, por ejemplo, el término $b_1 \wedge (b_2 \wedge b_3)$ puede escribirse como $b_1 \wedge b_2 \wedge b_3$ y el término $((t_1 \approx s_1) \wedge (t_2 \approx s_2)) \Rightarrow t$ se escribe como $t_1 \approx s_1 \wedge t_2 \approx s_2 \Rightarrow t$.

En el siguiente apartado, para introducir los conceptos relativos al mecanismo del *narrowing* (sin restricciones), consideraremos como programas TRS's más generales desde el punto de vista sintáctico (e.g., TRS's que no cumplen la disciplina de constructores, o cuyas reglas solapan – si bien, manteniendo la propiedad de ser confluentes).

2.9.2 Narrowing y Estrategias de Narrowing.

Esencialmente, el mecanismo del *narrowing* calcula una substitución apropiada, σ, que cuando se aplica sobre el término en consideración, t, éste puede ser reducido en un paso de reescritura [94].

Ejemplo 5 *Sean las reglas que definen el predicado menor o igual "$\leq$" sobre los números naturales representados por los términos formados usando los símbolos constructores "0" y sucesor "s":*

$$\begin{array}{rcl} 0 \leq N & \rightarrow & true \\ s(M) \leq 0 & \rightarrow & false \\ s(M) \leq s(N) & \rightarrow & M \leq N \end{array}$$

El término $s(X) \leq Y$ puede reducirse a true instanciando Y a $s(Y1)$, para aplicar la tercera regla, seguida de la instanciación de X por 0, para aplicar la primera regla:

$$s(X) \leq Y \overset{\{Y/s(Y1)\}}{\leadsto} X \leq Y1 \overset{\{X/0\}}{\leadsto} true$$

Es habitual definir un paso de *narrowing* de la forma siguiente, que se corresponde con la definición de paso de reescritura, pero sustituyendo el mecanismo de *ajuste de patrones* ("*matching*") por el de *unificación*:

Definición 2.9.2 (Paso de Narrowing)
Sea $\mathcal{R}$ un TRS. Sean t y s términos. Decimos que t se reduce por narrowing *a s si existe una posición $p \in \mathcal{FP}os(t)$, una (variante renombrada aparte de una) regla de reescritura $R \equiv l \rightarrow r$ de $\mathcal{R}$ y una substitución σ tal que:*

- $\sigma = mgu(\{t|_p = l\})$ *(i.e., σ es el unificador más general de $t|_p$ y l), y*
- $s \equiv \sigma(t[r]_p)$.

Escribimos que $t \overset{[p,R,\sigma]}{\leadsto} s$ o, simplemente, $t \overset{\sigma}{\leadsto} s$.

Algunos autores [22, 163] no exigen que las substiciones computadas, σ, sean unificadores más generales, sino sólo que sean unificadores. Con este enfoque, $t \overset{[p,R,\sigma]}{\leadsto} s$ es un *paso de narrowing* si p es una posición no variable de t y $\sigma(t) \rightarrow_{p,R} s$. Esta es la definición más general que puede darse de paso de *narrowing*. La Definición 2.9.2 tiene la ventaja de que los unificadores más generales pueden computarse de forma única, mientras que existen muchos unificadores independientes. Sin embargo, eliminar la restricción de que la substitución σ sea un unificador más general es un requisito indispensable en la definición de algunas estrategias de *narrowing* (ver más adelante).

Definición 2.9.3 (Derivación de Narrowing)
Sea $\mathcal{R}$ un TRS y t un término. Decimos que existe una derivación de narrowing *de t a s, si existe una secuencia de pasos $t \equiv t_0 \overset{[p_1,R_1,\sigma_1]}{\leadsto} t_1 \overset{[p_2,R_2,\sigma_2]}{\leadsto} \ldots \overset{[p_n,R_n,\sigma_n]}{\leadsto} t_n \equiv s$. Escribimos que $t \overset{\sigma}{\leadsto}^* s$, donde $\sigma = \sigma_n \circ \ldots \circ \sigma_2 \circ \sigma_1$. Decimos que s es el* resultado *de la derivación, con* respuesta *(parcial) σ. Decimos que el par $\langle s, \sigma \rangle$ es la* salida *de la derivación.*

La Definición 2.9.3 pone de manifiesto que los programas lógico–funcionales permiten tanto el cómputo de un resultado como la obtención de una respuesta. En particular, la semántica esperada de muchos lenguajes lógico–funcionales [49, 79, 96, 157], al igual que sucede con los lenguajes funcionales, se basa en el cómputo de términos constructores básicos (también denominados *valores*[12]). Recuperando una términología proveniente de la programación lógica, en ocasiones hablaremos de *derivación de éxito* para referirnos a las derivaciones $t \overset{\sigma}{\leadsto}^* s$ cuyo resultado s es un valor. Diremos que la correspondiente respuesta, restringida a las variables del término inicial, $\sigma\lceil_{\mathcal{V}ar(s)}$, es una *respuesta computada.* Las derivaciones (finitas) que no conduzcan a un valor se verán como *incompletas*. Hablaremos de *derivación de fallo* cuando, además, el resultado de la derivación es una forma irreducible que no es un valor. Las derivaciones de *narrowing* para un término pueden representarse mediante un árbol de búsqueda (posiblemente infinito) ramificado finitamente.

Definición 2.9.4 (Arbol de Narrowing)
Dado un TRS $\mathcal{R}$ y un término t, un árbol de narrowing *para t en $\mathcal{R}$, denotado $\tau(t, \mathcal{R})$ (o simplemente τ, cuando el término y el TRS sean evidentes por el contexto), es un árbol etiquetado, $\langle N, \leadsto, \Lambda\rangle$, donde N es un conjunto de* nodos *etiquetados por los términos obtenidos al probar pasos de* narrowing, *$\leadsto$ es la relación de* narrowing *en un paso, y Λ es un elemento distinguido de τ, etiquetado por el término t, llamado la* raíz *de τ.*

En lo que sigue no haremos distinción entre los nodos y los términos que los etiquetan. Cada una de las *ramas* del árbol de *narrowing* τ representan una de las posibles derivaciones de *narrowing* a partir del término situado en la raíz. Así pues, las *hojas* de τ representan el resultado de la computación. Siguiendo a Lloyd y Shepherdson [137], adoptamos la convención de que cualquier derivación es potencialmente incompleta y, por lo tanto, también lo serán los árboles a los que den lugar. La Figura 5.1 es un ejemplo de representación de un árbol de *narrowing*, en la que se ha seguido el convenio usual de etiquetar los arcos con el nombre de la regla del TRS con la que se da el paso, señalando con un subrayado el término (en la posición) que se reduce.

Inicialmente el *narrowing* se utilizó como un medio para resolver *ecuaciones*. Fue en los trabajos pioneros de [184] donde se introdujo por primera vez el *narrowing* como medio para la obtención de un conjunto de soluciones a un problema de $\mathcal{E}$-unificación. Lo que autoriza el uso del *narrowing* para la resolución de ecuaciones es su condición de ser un procedimiento correcto y completo para la $\mathcal{E}$-unificación.

Definición 2.9.5
Dado un TRS $\mathcal{R}$ (perteneciente a una clase de TRS's) y la teoría ecuacional $\mathcal{E} = \{l = r \mid (l \to r) \in \mathcal{R}\}$ por él generada, decimos que el algoritmo de narrowing *es:*

[12] En el contexto de la programación funcional, suele utilizarse la palabra *valor* refiriéndose, únicamente, a los términos constructores básicos. Sin embargo, en un contexto lógico–funcional, en ocasiones, extenderemos el concepto de *valor* a los términos constructores (posiblemente con variables).

1. **Correcto**: *Si* $(s \approx t) \overset{\sigma}{\rightsquigarrow}^* true$ *es una derivación de* narrowing *entonces* $\sigma(s) =_{\mathcal{E}} \sigma(t)$ *(i.e.,* σ *es un* $\mathcal{E}$*-unificador de* s *y* t*).*

2. **Completo**: *Si* $\sigma(s) =_{\mathcal{E}} \sigma(t)$ *entonces existe una derivación de* narrowing $(s \approx t) \overset{\theta}{\rightsquigarrow}^* true$ *tal que* $\theta \leq_{\mathcal{E}} \sigma\ [\mathcal{V}ar(s) \cup \mathcal{V}ar(t)]$.

donde s *y* t *son términos.*

La propiedad de completitud indica que, dado un $\mathcal{E}$-unificador σ de s y t, el *narrowing* computa una respuesta θ más $\mathcal{E}$-general. También, esta propiedad puede expresarse diciendo que el *narrowing* computa un conjunto completo de $\mathcal{E}$-unificadores $\mathcal{O}_{\mathcal{R}} = \{\theta_{\lceil(\mathcal{V}ar(s)\cup\mathcal{V}ar(t))} \mid (s \approx t) \overset{\theta}{\rightsquigarrow}^* true\}$. Empleando la nomenclatura de la programación lógica, hablamos del *conjunto de éxitos* (*success set*) del objetivo ecuacional $(s \approx t)$ en el programa $\mathcal{R}$. En general el *narrowing* es correcto, sin embargo, para lograr la completitud deben imponerse ciertas restricciones a los TRS's con los que se tratan, así como al tipo de resultados y respuestas obtenidos. El *narrowing*[13] es completo para TRS's canónicos [103]. En la programación lógico–funcional, es habitual restringir la atención sobre las respuestas normalizadas (i.e., aquellas cuyo rango está formado por términos en forma normal). Considerar solamente las respuestas normalizadas está justificado ya que, en un contexto funcional, los "valores" que estamos buscando son siempre términos constructores y por lo tanto normalizados [58]. La restricción a respuestas constructoras es suficiente en la práctica; de otro modo, al considerar como "solución" substituciones más gerales, podríamos incurrir en el riesgo de que estas contuvieran expresiones no evaluadas o indefinidas. El *narrowing* es completo para TRS's confluentes cuando nos restringimos a substituciones normalizadas [147]. En este caso, además, podremos eliminar el subíndice "$\mathcal{E}$" del orden "$\leq_{\mathcal{E}}$" y emplear el preorden de máxima generalidad "$\leq$" en la Definición 2.9.5.

Estrategia de Narrowing.

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 demasiado amplio. Se han diseñado muchas estrategias para reducir el tamaño del espacio de búsqueda, eliminando algunas derivaciones inútiles. En este contexto, es habitual definir el concepto de estrategia como una aplicación φ que asigna a cada término t un conjunto de posiciones $\varphi(t) \subseteq \mathcal{FP}os(t)$ para las que puede darse un paso de *narrowing*, i.e., una estrategia es una restricción del espacio de búsqueda. Dado que pueden aplicarse varias reglas del TRS sobre una posición p seleccionada por la estrategia φ, es conveniente generalizar el concepto de estrategia

[13] En el caso del *narrowing* sin restricciones $\mathcal{R}_+$, la extensión de $\mathcal{R}$ para tratar con la igualdad, denota el conjunto $\mathcal{R} \cup \{x \approx x \rightarrow true\}$. Esto permite tratar la unificación sintáctica como un paso de *narrowing*, utilizando la regla "$x \approx x \rightarrow true$" para computar *mgu*'s. Entonces $s \approx t \overset{\theta}{\rightsquigarrow} true$ si y sólo si $\sigma = mgu(\{s \approx t\})$.

para devolver ternas formadas por la propia posición p, la regla del TRS R y la substitución σ aplicada al dar el paso de *narrowing*.

Definición 2.9.6 (Estrategia de Narrowing)
Una estrategia de narrowing *es una aplicación φ que, para un término t, computa el conjunto de ternas $\langle p, R, \sigma\rangle$, donde $p \in \mathcal{FP}os(t)$, $R \equiv (l \to r)$ es la regla del TRS utilizada para dar el paso de* narrowing *y σ una substitución unificadora de $t|_p$ y l* [14].

Dado un término t y una regla $R \equiv (l \to r)$ del TRS $\mathcal{R}$, decimos que $t \overset{[p,R,\sigma]}{\leadsto_\varphi} \sigma(t[r]_p)$ es un paso de *narrowing* que respeta la estrategia φ, si $\langle p, R, \sigma\rangle \in \varphi(t)$. Si el conjunto $\varphi(t)$ contiene un solo elemento, decimos que el paso de *narrowing* es *determinista*. De forma semejante, diremos que una derivación $t \equiv t_0 \overset{[p_1,R_1,\sigma_1]}{\leadsto_\varphi} t_1 \overset{[p_2,R_2,\sigma_2]}{\leadsto_\varphi} \ldots \overset{[p_n,R_n,\sigma_n]}{\leadsto_\varphi} t_n \equiv s$, respeta la estrategia φ, y escribimos $t \overset{\sigma}{\leadsto}_\varphi{}^* s$, si cada uno de sus pasos es un paso de *narrowing* que respeta la estrategia φ. Decimos que un término t es *evaluable de manera determinista* bajo la estrategia φ, si cada paso de *narrowing* en una derivación a partir de t que respeta φ, es determinista.

Una propiedad importante que debe cumplir toda estrategia es que siga manteniendo, bajo determinadas condiciones, la completitud del cálculo. Una clasificación de las diferentes estrategias y estudios detallados sobre las condiciones que debe cumplir un programa para que una determinada estrategia sea correcta y completa, pueden encontrarse en [94] y en [149].

Para evitar computaciones innecesarias y posibilitar el empleo de estructuras de datos infinitas, muchos trabajos se han centrado en el estudio de las estrategias de *evaluación perezosa* [23, 79, 96, 138, 157]. En el siguiente apartado introducimos la estrategia de *narrowing perezoso*, que es correcta y completa para la clase de programas definida en el Apartado 2.9.1. Consideraremos que los programas se ejecutan respetando la estrategia de *narrowing* perezoso. Después, estudiaremos la estrategia de *narrowing necesario* a efectos de precisar la relación entre ambas. El *narrowing* necesario es correcto y completo para una subclase de los programas bajo consideración, la subclase de los programas *inductivamente secuenciales*, sobre los cuales se satisface, además, una propiedad de optimalidad que formalizaremos posteriormente. En lo sucesivo, omitiremos el subindice "+" que denota a los programas extendidos con el conjunto de reglas StrEq. Así pues, salvo que se diga lo contrario, $\mathcal{R}$ es indicativo de $\mathcal{R}_+$.

2.9.3 Narrowing Perezoso.

La evaluación perezosa es una característica esencial de los lenguajes lógico–funcionales, ya que facilita una mayor expresividad, al permitir la definición de funciones parciales y no estrictas y la utilización de expresiones infinitas. El

[14] Habitualmente la substitución σ se exige que sea un unificador más general, i.e., $\sigma = mgu(\{l = t|_p\})$. Hacemos esta salvedad para permitir estrategias, que como la estrategia de *narrowing* necesario, computan substituciones que no son necesariamente unificadores más generales.

narrowing perezoso reduce las expresiones comenzando por las posiciones más externas ("outermost") sobre las que se puede dar un paso de *narrowing*. Los pasos de *narrowing* sobre posiciones más internas solamente se realizan si son demandados (por el patrón de la lhs de alguna regla) y contribuyen a un paso de *narrowing* posterior sobre una posición más externa. Dado que la noción de "posición demandada" no es única, se han propuesto diferentes estrategias de *narrowing* perezoso [23, 119, 138, 157, 174]. En lo que sigue, especificamos nuestra estrategia de *narrowing* perezoso, que es similar en lo esencial a la presentada en [157].

Las siguientes definiciones son necesarias para nuestra formalización de la estrategia de *narrowing* perezoso. Esta formalización apareció por vez primera en [8]. Comenzamos caracterizando una clase particular de problema de unificación que se presenta debido al hecho de trabajar con programas CB y lineales por la izquierda.

Definición 2.9.7 (Problema de Unificación Lineal)
Un problema de unificación lineal es un par de términos:

$$\langle f(d_1,\ldots,d_n), f(t_1,\ldots,t_n)\rangle;$$

donde $f(d_1,\ldots,d_n)$ *es un patrón lineal que no comparte variables con* $f(t_1,\ldots,t_n)$*.*

Resolvemos los problemas de unificación lineal utilizando el algoritmo presentado en [157], donde el caso en el que un intento de unificación no tiene éxito debido a que se produce una colisión entre un símbolo constructor c y uno de operación f, no se ve como un fracaso, sino como una demanda de una mayor evaluación de f. Esto difiere con respecto al algoritmo de unificación sintáctica estándar [125]. Lo que sigue es una reformulación del algoritmo de unificación para problemas de unificación lineal presentado en [157]. Debido a la linealidad de los patrones que aparecen en las lhs's de las reglas de los programas, no se precisa de ningún mecanismo de "occur-check".

Definición 2.9.8 (Configuración LU)
Una configuración LU *es un par* (U,σ)*, donde* U *es un conjunto* $\{d_1 \downarrow_1 t_1,\ldots, d_n \downarrow_n t_n\}$*, siendo* $d_1,\ldots,d_n$ *términos constructores lineales que no comparten variables y* σ *una substitución. Los términos* $d_1,\ldots,d_n$ *(respectivamente,* $t_1,\ldots,t_n$*) se denominan* términos lhs *(*términos rhs*) de* U*.*

Generalizamos la aplicación de una substitución σ sobre un conjunto $U \equiv \{d_1 \downarrow_1 t_1,\ldots,d_n \downarrow_n t_n\}$ en la forma obvia: $\sigma(U) = \{\sigma(d_1) \downarrow_1 \sigma(t_1),\ldots,\sigma(d_n) \downarrow_n \sigma(t_n)\}$.

Definición 2.9.9 (Relación de Unificación $\to_{\mathsf{LU}}$)
Definimos la relación de unificación $\to_{\mathsf{LU}}$*, entre configuraciones* LU*, como la relación más pequeña que satisface:*

1. $(\{c(d_1,\ldots,d_m) \downarrow_u c(t_1,\ldots,t_m)\} \cup U, \sigma) \rightarrow_{\mathsf{LU}}$
   $(\{d_1 \downarrow_{u.1} t_1, \ldots, d_m \downarrow_{u.m} t_m\} \cup U, \sigma)$,
   donde $(c/m) \in \mathcal{C}, m \geq 0$.
2. $(\{x \downarrow_u t\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{x/t\}(U), \{x/t\} \circ \sigma)$,
   donde $t \notin V$.
3. $(\{d \downarrow_u x\} \cup U, \sigma) \rightarrow_{\mathsf{LU}} (\{x/d\}(U), \{x/d\} \circ \sigma)$.
4. $(\{c(d_1,\ldots,d_m) \downarrow_u c'(t_1,\ldots,t_p)\} \cup U, \sigma) \quad \rightarrow_{\mathsf{LU}} (\{\mathtt{fail}\}, \sigma)$,
   donde $(c/m), (c'/p) \in \mathcal{C}$, $c \neq c'$, *y* $m, p \geq 0$.

Notad que la relación de unificación $\rightarrow_{\mathsf{LU}}$ está bien definida, en el sentido de que partiendo de una configuración LU que cumple los requisitos de la Definición 2.9.8 se obtiene otra configuración LU que también los cumple.

Definición 2.9.10 (Configuración Inicial LU)
Sea un problema de unificación lineal $\langle f(d_1,\ldots,d_n), f(t_1,\ldots,t_n)\rangle$. *La* configuración inicial LU *es:* $(U_0, \sigma_0) \equiv (\{d_1 \downarrow_1 t_1, \ldots, d_n \downarrow_n t_n\}, id)$.

Un problema de unificación lineal (LU) puede tener éxito (*success*), fallar (*fail*) o quedar suspendido. Cuando queda suspendido, devuelve el conjunto de posiciones que se demandan (*demand*) para una evaluación posterior. Formalmente, dada una configuración LU irreducible, (U, σ), una posición u es *demandada*, si $(c(d_1,\ldots,d_m) \downarrow_u g(t_1,\ldots,t_p)) \in U$.

Definición 2.9.11 (Comportamiento de $\rightarrow_{\mathsf{LU}}$)
Sea $\Gamma \equiv \langle f(d_1,\ldots,d_n), f(t_1,\ldots,t_n)\rangle$ *un problema de unificación lineal. Sea* $(U_0, \sigma_0) \equiv (\{d_1 \downarrow_1 t_1, \ldots, d_n \downarrow_n t_n\}, id) \rightarrow^*_{\mathsf{LU}} (U, \sigma) \not\rightarrow_{\mathsf{LU}}$, *el proceso de unificación lineal que conduce a la configuración irreducible* (U, σ). *Definimos la función* $\mathsf{LU}(\Gamma)$ *como sigue:*

$$\mathsf{LU}(\Gamma) = \begin{cases} (\textsc{Succ}, \sigma) & \mathtt{si}\ U = \emptyset \\ (\textsc{Fail}, \emptyset) & \mathtt{si}\ U = \{\mathtt{fail}\} \\ (\textsc{Demand}, P) & \mathtt{en\ otro\ caso,} \\ & \mathtt{donde}\ P\ \textit{es el conjunto de posiciones demandadas} \end{cases}$$

Como en los procedimientos de prueba de la programación lógica, suponemos que las reglas que se utilizan en el proceso de unificación lineal siempre contienen variables frescas (i.e., las reglas se suponen "*renombradas aparte*"). Esto supone que todas las substituciones computadas son idempotentes. Introducimos el símbolo "$\ll$" para indicar que una regla R perteneciente a un TRS $\mathcal{R}$ se toma como una variante renombrada aparte y escribimos $R \ll \mathcal{R}$.

Los siguientes lemas establecen propiedades interesantes de la relación de unificación $\rightarrow_{\mathsf{LU}}$ introducida en la Definición 2.9.9.

Lema 2.9.12 *Sea* (U, σ) *una configuración* LU. *Sea la transición de un paso* $(U, \sigma) \rightarrow_{\mathsf{LU}} (U', \sigma')$, *donde* $U' \not\equiv \{\mathtt{fail}\}$. *Entonces:*

1. *Las variables en los términos lhs de* U' *están incluidas en el conjunto de variables de los términos lhs de* U.

2. *Si los términos lhs de U no comparten variables con los términos rhs de U o con σ, entonces los términos lhs de U' no comparten variables con los otros términos lhs o con los términos rhs de U', ni con σ'* [15].

Prueba. La primera parte del enunciado de este lema es inmediata por la Definición 2.9.8 de configuración LU y la Definición 2.9.9 de relación de unificación lineal. Para la prueba de la segunda parte consideramos tres casos:

1. Si se aplica la regla (1) de la Definición 2.9.9, entonces el resultado se sigue de forma inmediata.

2. Supongamos que se aplica la regla (2) de la Definición 2.9.9, y consideremos la transición $(U, \sigma) \equiv (\{x \downarrow_u t\} \cup U^\star, \sigma) \to_{\mathsf{LU}} (\{x/t\}(U^\star), \{x/t\} \circ \sigma) \equiv (U', \sigma')$. Ya que los términos lhs de U no comparten variables con los otros términos lhs, por definición de configuración LU, o con los términos rhs de U, entonces $U' = \{x/t\}(U^\star) = U^\star$, y entonces $U' \subseteq U$ (por lo que $\mathcal{V}ar(U') \subseteq \mathcal{V}ar(U)$). Más aún, ya que los términos lhs de U' no comparten variables con σ, x y t, entonces no comparten variables con σ'.

3. Consideremos ahora que se aplica la regla (3) de la Definición 2.9.9, entonces la transición es $(U, \sigma) \equiv (\{d \downarrow_u x\} \cup U^\star, \sigma) \to_{\mathsf{LU}} (\{x/d\}(U^\star), \{x/d\} \circ \sigma) \equiv (U', \sigma')$. Ya que los términos lhs de U no comparten variables con los otros términos lhs, por definición de configuración LU, o con los términos rhs de U, o con σ, entonces la substitución $\{x/d\}$ no instancia los términos lhs de $U^\star$, y así las variables de los términos lhs de U' están incluidas en el conjunto de variables de los términos lhs de U, con más precisión, el conjunto de las variables que aparecen en los términos lhs de U' es la diferencia del conjunto de las variables que aparecen en los términos lhs de U y $\mathcal{V}ar(d)$, de manera que los términos lhs de U' no comparten variables con los otros términos lhs o los términos rhs de U', ni con σ'.

$\square$

Es importante notar que, como consecuencia del Lema 2.9.12, la regla (2) de la Definición 2.9.9 de relación de unificación $\to_{\mathsf{LU}}$ puede simplificarse de la siguiente forma: $(\{x \downarrow_u t\} \cup U, \sigma) \to_{\mathsf{LU}} (U, \{x/t\} \circ \sigma)$, donde $t \notin V$. En efecto, $x \notin \mathcal{V}ar(U)$.

Lema 2.9.13 *Sea $\langle l, s\rangle \equiv \langle f(d_1, \ldots, d_k), f(t_1, \ldots, t_k)\rangle$ un problema de unificación lineal. Sea $(U_0, id) \equiv (\{d_1 \downarrow_1 t_1, \ldots, d_k \downarrow_k t_k\}, id) \to^*_{\mathsf{LU}} (U_n, \sigma_n)$ una derivación LU, donde $U_n \not\equiv \{\mathtt{fail}\}$. Entonces, para cada $x, y \in Dom(\sigma_n\lceil_{\mathcal{V}ar(s)})$, con $x \neq y$, $\sigma_n(x)$ es un término constructor lineal que no comparte variables ni con s ni con $\mathcal{V}ar(\sigma_n(y))$, y $\mathcal{V}ar(\sigma_n(x)) \subseteq \mathcal{V}ar(l)$.*

Prueba. Probamos este lema por inducción en el número de pasos n de la derivación LU. Ya que el caso base $n = 0$ es trivial, pasamos a considerar el caso inductivo $n > 0$.

[15] Esto es, no comparten variables ni con el domino ni con el rango de σ'.

Supongamos $(U_0, id) \rightarrow^*_{\mathsf{LU}} (U_{n-1}, \sigma_{n-1}) \rightarrow_{\mathsf{LU}} (U_n, \sigma_n)$, $n > 0$, con $U_n \not\equiv \{\mathtt{fail}\}$. En primer lugar, ya que las condiciones del Lema 2.9.12 se cumplen para la configuración inicial (U_0, id), mediante la aplicación reiterada de este lema, obtenemos que los términos lhs de U_{n-1} no comparten variables ni con los otros términos lhs ni tampoco con los términos rhs de U_{n-1}, o con σ_{n-1}. Más aún, también por el Lema 2.9.12, las variables en los términos lhs de cada configuración U_i, $0 \leq i \leq n$, provienen del término (estandarizado aparte) l que no comparte variables con s. También notad que los términos lhs de cada configuración U_i son términos constructores lineales, ya que son subtérminos de l que no son instanciados a lo largo de la derivación.

Consideremos el último paso $(U_{n-1}, \sigma_{n-1}) \rightarrow_{\mathsf{LU}} (U_n, \sigma_n)$. Distinguimos tres casos:

1. Si se aplica la regla (1) de la Definición 2.9.9, entonces el resultado se sigue directamente por la hipótesis de indución, ya que $\sigma_n = \sigma_{n-1}$.

2. Supongamos que se aplica la regla (2) de la Definición 2.9.9, y consideremos la transición $(U_{n-1}, \sigma_{n-1}) \rightarrow_{\mathsf{LU}} (U_n, \sigma_n)$, donde $(U_{n-1}, \sigma_{n-1}) \equiv (\{x \downarrow_u t\} \cup U^\star, \sigma_{n-1})$ y $(U_n, \sigma_n) \equiv (\{x/t\}(U^\star), \{x/t\} \circ \sigma_{n-1})$. Ya que $x \notin \mathcal{V}ar(\sigma_{n-1})$, entonces $\sigma_n = \sigma_{n-1} \cup \{x/t\}$ y, puesto que $x \in \mathcal{V}ar(l)$, entonces $\sigma_n = \sigma_{n-1}[\mathcal{V}ar(s)]$, y el resultado se sigue por la hipótesis de indución.

3. Consideremos ahora que se aplica la regla (3) de la Definición 2.9.9, y la transición es $(U_{n-1}, \sigma_{n-1}) \rightarrow_{\mathsf{LU}} (U_n, \sigma_n)$, donde $(U_{n-1}, \sigma_{n-1}) \equiv (\{d \downarrow_u x\} \cup U^\star, \sigma_{n-1})$ y $(U_n, \sigma_n) \equiv (\{x/t\}(U^\star), \{x/t\} \circ \sigma_{n-1})$. Tenemos que considerar dos casos:

   (a) Sea $x \in \mathcal{V}ar(l)$. Por la hipótesis de indución, los términos en el codominio de $\sigma_{n-1\restriction\mathcal{V}ar(s)}$ son términos constructores lineales que no comparten variables ni con s ni tampoco entre ellos mismos. Además, para toda $x \in \mathit{Dom}(\sigma_{n-1\restriction\mathcal{V}ar(s)})$, $\mathcal{V}ar(\sigma_{n-1}(x)) \subseteq \mathcal{V}ar(l)$. Por el Lema 2.9.12, d no comparte variables con σ_{n-1} y, por lo tanto, los términos en el codominio de $\sigma_{n\restriction\mathcal{V}ar(s)}$ son constructores lineales, no comparten variables y, para toda $x \in \mathit{Dom}(\sigma_{n\restriction\mathcal{V}ar(s)})$, $\mathcal{V}ar(\sigma_n(x)) \subseteq \mathcal{V}ar(l)$.

   (b) Sea $x \in \mathcal{V}ar(s)$. Por la hipótesis de indución, la variable x no aparece en ningún término en el codominio de $\sigma_{n-1\restriction\mathcal{V}ar(s)}$, y así $\sigma_n = \sigma_{n-1} \cup \{x/d\}[\mathcal{V}ar(s)]$. Ahora d es un constructor lineal, $\mathcal{V}ar(d) \subseteq \mathcal{V}ar(l)$ y, por el Lema 2.9.12, d no comparte variables con σ_{n-1}. Por consiguiente, el resultado se sigue directamente por hipótesis de inducción.

□

Vamos a definir la estrategia de *narrowing* perezoso que empleamos para computar el conjunto de posiciones perezosas de un término t. Informalmente, la función $\lambda_{lazy}(t)$ devuelve el conjunto de ternas $\langle p, R_k, \sigma \rangle$ tal que $p \in \mathcal{FP}os(t)$

es una posición perezosa de t que puede ser reducida por *narrowing* mediante la regla R_k utilizando la substitución σ. Suponemos que las reglas de $\mathcal{R}$ están numeradas como $R_1, \ldots, R_m$.

Definición 2.9.14 (Estrategia de Narrowing Perezoso)
Definimos la estrategia de narrowing perezoso *como una función λ_{lazy} que, aplicada a un término t, computa el conjunto de ternas $\langle p, R_k, \sigma\rangle$, siendo $p \in \mathcal{FP}os(t)$ una posición perezosa de t, $R_k \equiv (l_k \to r_k)$ es una regla (renombrada aparte) de $\mathcal{R}$ y σ una substitución, como sigue:*

$$
\begin{array}{lcl}
\lambda_{lazy}(t) & = & \bigcup_{k=1}^{m} \lambda\_(t, \Lambda, R_k) \\
\lambda\_(t, p, R_k) & = & \texttt{si } \mathcal{H}ead(l_k) = \mathcal{H}ead(t|_p) \texttt{ entonces} \\
 & & \quad \texttt{en caso de que } \mathsf{LU}(\langle l_k, t|_p\rangle) = \\
 & & \quad \left\{ \begin{array}{ll} (\texttt{Succ}, \sigma): & \{\langle p, R_k, \sigma\rangle\} \\ (\texttt{Fail}, \emptyset): & \emptyset \\ (\texttt{Demand}, P): & \bigcup_{q \in P} \bigcup_{k=1}^{m} \lambda\_(t, p.q, R_k) \end{array} \right. \\
 & & \texttt{sino } \emptyset
\end{array}
$$

Como ya se ha mencionado, existen varias fuentes de indeterminismo en el cálculo de *narrowing*: la elección del subtérmino reducible (redex) señalado por la posición p y la elección de la regla del programa R_k. Ambas, son fuentes de indeterminismo "don't-know", lo que significa que, en general, todos los posibles puntos de elección deben explotarse para asegurar la completitud del proceso. El *narrowing* perezoso es *fuertemente completo* ("strong complete" [91, 150, 151]) con respecto a substituciones constructoras para TRS's CB y ortogonales [157, 94]. La completitud fuerte significa que la selección de un subconjunto de posiciones perezosas dentro de los subtérminos t_i de un término conjuntivo $t_1 \wedge t_2 \wedge \ldots \wedge t_n$ puede realizarse de forma "don't-care". Por lo tanto, podemos seleccionar las posiciones perezosas del subtérmino t_i y desestimar el resto de posiciones perezosas en los subtérminos t_j, con $j \neq i$. En particular, el subtérmino t_i seleccionado para su inspección puede ser el que, poseyendo un subconjunto de posiciones perezosas, se encuentre más a la izquierda en la conjunción. Nosotros simulamos esta selección "don't-care" introduciendo en los programas una única regla para definir el símbolo de función predefinido "$\wedge$" (i.e., $true \wedge x \to x$), que garantiza que se seleccionará el subconjunto de posiciones perezosas situado más a la izquierda en un término conjuntivo.

Después de este trabajo preparatorio, ya estamos en disposición de definir formalmente qué se entiende por *narrowing* perezoso (LN, del inglés *Lazy Narrowing*).

Definición 2.9.15 (Narrowing Perezoso)
Definimos el narrowing perezoso *como un sistema de transición etiquetado cuya relación de transición $\leadsto_{LN} \subseteq (\mathcal{T} \times \lambda_{lazy}(\mathcal{T}) \times \mathcal{T})$ es la relación más pequeña que satisface:*

$$
\frac{\langle p, R, \sigma\rangle \in \lambda_{lazy}(t) \wedge \ R \equiv (l \to r) \ll \mathcal{R}}{t \overset{[p,R,\sigma]}{\leadsto}_{LN} \ \sigma(t[r]_p)}
$$

Si $s_0 \overset{[p_1,R_1,\sigma_1]}{\leadsto}_{LN} s_1 \overset{[p_2,R_2,\sigma_2]}{\leadsto}_{LN} \ldots \overset{[p_n,R_n,\sigma_n]}{\leadsto}_{LN} s_n$, escribimos, $s_0 \overset{\sigma}{\leadsto}_{LN}{}^* s_n$, siendo $\sigma = \sigma_n \circ \ldots \circ \sigma_2 \circ \sigma_1$ y hablamos de que existe una *derivación de narrowing perezoso* para el término s_0 con *resultado* (parcial) s_n y *respuesta* σ. Principalmente, estamos interesados en aquellas derivaciones que conducen a un término constructor y en la restricción de las respuestas a las variables del término inicial.

Ejemplo 6 *Consideremos de nuevo las reglas que definen el predicado "$\leq$" en el Ejemplo 5, junto con las siguientes reglas que definen la adición sobre los números naturales:*

$$\begin{array}{rcl} 0 + N & \to & N \\ s(M) + N & \to & s(M + N) \end{array}$$

Entonces el narrowing *perezoso evalúa el término $X \leq X+X$ aplicando un paso de* narrowing *sobre la posición Λ, con la primera regla de "$\leq$", o aplicando un paso de* narrowing *sobre la posición 2, ya que el argumento $X+X$ es demandado por la segunda y la tercera reglas de "$\leq$". De este modo, pueden darse tres pasos de* narrowing *perezoso diferentes a partir del término inicial:*

$$\begin{array}{lll} X \leq X + X & \overset{\{X \mapsto 0\}}{\leadsto}_{LN} & true \\ X \leq X + X & \overset{\{X \mapsto 0\}}{\leadsto}_{LN} & 0 \leq 0 \\ X \leq X + X & \overset{\{X \mapsto s(M)\}}{\leadsto}_{LN} & s(M) \leq s(M + s(M)) \end{array}$$

Notad que el segundo paso puede considerarse en cierto sentido superfluo, ya que conduce al mismo resultado (true) con la misma respuesta que el primer paso. Este hecho introduce una fuente de ineficiencia en la estrategia de narrowing *perezoso, que también se pone de manifiesto en otros contextos.*

Para programas CB y ortogonales, resolver una ecuación $e \equiv (s \approx t)$ consiste en hallar los valores de las variables que permiten reducir los términos s y t a un mismo término constructor. Por consiguiente, en este contexto, considerando la Proposición 2.9.1, una *solución* de una ecuación e es una substitución σ tal que $\sigma(e)$ se reescribe a *true* usando las reglas del programa. La estrategia de *narrowing* perezoso es completa con respecto a la igualdad estricta para substituciones constructoras:

Teorema 2.9.16 *[157] Sea $\mathcal{R}$ un programa CB y ortogonal, e una ecuación, y σ una substitución constructora que es una solución para e. Entonces existe una derivación de* narrowing *perezoso $e \overset{\sigma'}{\leadsto}_{LN}{}^* true$ tal que $\sigma' \leq \sigma$ $[\mathcal{V}ar(e)]$.*

Notad que hemos hecho desaparecer el subíndice $\mathcal{E}$ de la expresión "$\sigma' \leq_{\mathcal{E}} \sigma$ $[\mathcal{V}ar(e)]$" ya que en nuestro contexto tratamos, solamente, con respuestas constructoras y, por tanto, normalizadas [147].

Para finalizar este apartado, presentamos una propiedad interesante de esta estrategia que utilizaremos más adelante.

Proposición 2.9.17 *Sea un programa $\mathcal{R}$ y un término s. Sea $\mathcal{D} \equiv (s \stackrel{\sigma}{\leadsto}_{LN}{}^* t)$ una derivación de* narrowing *perezoso para s en $\mathcal{R}$. Entonces, para toda $x, z \in Dom(\sigma_{\lceil \mathcal{V}ar(s)})$, con $x \neq z$, se cumple que $\sigma(x)$ y $\sigma(z)$ son términos constructores lineales, y $\mathcal{V}ar(\sigma(x)) \cap \mathcal{V}ar(\sigma(z)) = \emptyset$.*

Prueba. Sea $\mathcal{D} \equiv (s \equiv s_0 \stackrel{[p_1, R_1, \sigma_1]}{\leadsto}_{LN} s_1 \stackrel{[p_2, R_2, \sigma_2]}{\leadsto}_{LN} \ldots \stackrel{[p_n, R_n, \sigma_n]}{\leadsto}_{LN} s_n \equiv t)$ y $\sigma = \sigma_n \circ \ldots \circ \sigma_1$. Primero hay que notar que $\sigma_{\lceil \mathcal{V}ar(s)} = \sigma_{n \lceil \mathcal{V}ar(s_{n-1})} \circ \ldots \circ \sigma_{1 \lceil \mathcal{V}ar(s_0)}$. Ahora, por el Lema 2.9.13, para todo par de variables distintas $x, z \in Dom(\sigma_{i \lceil \mathcal{V}ar(s_{i-1})})$, $\sigma_i(x)$ es un término constructor lineal que no comparte variables ni con s_{i-1} ni con $\sigma_i(z)$, y $\mathcal{V}ar(\sigma_i(x)) \subseteq \mathcal{V}ar(l_i)$ (siendo l_i la lhs de la correspondiente regla R_i), para $i = 1, \ldots, n$. Por consiguiente, los términos en los codominios de $\sigma_{1 \lceil \mathcal{V}ar(s_0)}, \ldots, \sigma_{n \lceil \mathcal{V}ar(s_{n-1})}$ no comparten variables y, por tanto, los términos en el codomino de $\sigma_{\lceil \mathcal{V}ar(s)}$ son también términos constructores lineales que no comparten variables. □

2.9.4 Narrowing Necesario.

En lo que sigue, vamos a resumir los resultados fundamentales relativos a la estrategia de *narrowing* necesario [16, 22, 23]. El *narrowing necesario* es una estrategia que se basa en la selección de las *posiciones necesarias* más externas ("outermost") de un término para dar un paso de *narrowing*, de modo que solamente se dan pasos inevitables para el cómputo de un resultado o la solución de una ecuación. El *narrowing* necesario es destacable por sus propiedades de optimalidad con respecto a la longitud de las derivaciones de éxito (en implementaciones basadas en grafos) y el número de soluciones computadas.

El *narrowing* necesario se define para la clase de los *programas inductivamente secuenciales*. Esta clase de programas es un subconjunto de los programas CB ortogonales y recientemente se ha probado que coincide con la clase de los *programas CB fuertemente secuenciales* [97]. Para proporcionar una definición precisa de esta clase de programas y de la estrategia de *narrowing* necesario, introducimos el concepto de árbol definicional. En lugar de la definición original [18], presentamos una definición "declarativa" introducida en [19] y [20] que es más útil para la demostración de ciertas propiedades que deseamos estudiar.

Definición 2.9.18 (Arbol Definicional)
Un árbol definicional *de un conjunto finito de patrones lineales S, es un conjunto no vacío $\mathcal{P}$ de patrones lineales ordenado por el orden $<$ (de generalidad relativa estricta) y que cumple las siguientes propiedades:*

Propiedad de la raíz: Existe un elemento mínimo $pattern(\mathcal{P})$, que denominamos patrón *del árbol definicional.*

Propiedad de las hojas: Los elementos maximales, llamados hojas*, son los elementos de S. Los elementos no maximales se denominan* ramas*.*

Propiedad de los padres: Si $\pi \in \mathcal{P}$, $\pi \neq pattern(\mathcal{P})$, entonces existe un único $\pi' \in \mathcal{P}$, llamado padre *de π (y π se denomina* hijo *de π'), tal que $\pi' < \pi$ y no existe otro patrón $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ con $\pi' < \pi'' < \pi$.*

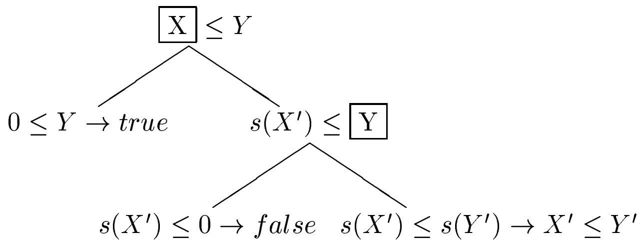

Figura 2.2: Arbol definicional para la función "$\leq$"

Propiedad inductiva: Dado un patrón $\pi \in \mathcal{P}\backslash S$, *existe una posición* o *en* π *con* $\pi|_o \in \mathcal{X}$ *(llamada* posición inductiva*) y constructores* $c_1, \ldots, c_n \in \mathcal{C}$ *con* $c_i \neq c_j$ *para* $i \neq j$, *tal que, para todo* $\pi_1, \ldots, \pi_n$ *que tiene por padre a* π, $\pi_i = \pi[c_i(x_1, \ldots, x_{n_i})]_o$ *(donde* $x_1, \ldots, x_{n_i}$ *son variables nuevas y distintas) para todo* $1 \leq i \leq n$.

Si $\mathcal{R}$ es un TRS ortogonal y f una función definida, con $ar(f) = n$, decimos que $\mathcal{P}$ es un *árbol definicional de* f, si $pattern(\mathcal{P}) = f(x_1, \ldots, x_n)$, donde $x_1, \ldots, x_n$ son variables distintas y las hojas de $\mathcal{P}$ son todas (y únicamente) variantes de las lhs's de las reglas de $\mathcal{R}$ que definen f.

Definición 2.9.19 (Inductivamente Secuencial)
Sea f *una función definida en un TRS CB,* $\mathcal{R}$. *La función* f *se dice que es* inductivamente secuencial *si existe un árbol definicional para* f. *Un TRS* $\mathcal{R}$ *se denomina* inductivamente secuencial *si todas las funciones que se definen en* $\mathcal{R}$ *son inductivamente secuenciales.*

Un TRS inductivamente secuencial puede verse como un conjunto de árboles definicionales, cada uno de los cuales define un símbolo de función. Dado el indeterminismo existente en la formación de los árboles definicionales, puede haber más de un árbol definicional para una función inductivamente secuencial. En lo que sigue, para cada función definida que sea inductivamente secuencial, supondremos fijado su árbol definicional entre uno cualquiera de los varios existentes. Como ya se ha indicado, no todo TRS ortogonal y CB es inductivamente secuencial. Notad, por ejemplo, que el TRS de *Berry* [112]

$$\begin{array}{lcl} f(a,b,X) & \rightarrow & c \\ f(b,X,a) & \rightarrow & c \\ f(X,a,b) & \rightarrow & c \end{array}$$

donde a, b y c se consideran símbolos constructores, es claramente un TRS CB ortogonal pero no es inductivamente secuencial, ya que es imposible construir un árbol definicional para f. Una representación gráfica de los árboles definicionales puede facilitar su entendimiento. Es habitual asociar a cada nodo un patrón y marcar, mediante un recuadro, cada posición inductiva de una rama. Finalmente, las hojas contienen las reglas correspondientes. La Figura 2.2 ilustra el árbol definicional para la función "$\leq$" del Ejemplo 5.

Ahora estamos en disposición de definir qué entendemos por estrategia de *narrowing* necesario (NN, del inglés *Needed Narrowing*). La descripción que realizamos de la estrategia de *narrowing* necesario, extraída de [16] (y similar a la que aparece en [20]), es ligeramente diferente a la presentada en [23], pero conduce a los mismos pasos de *narrowing* necesario en una derivación[16].

Definición 2.9.20 (Estrategia de Narrowing Necesario) *Sea $\mathcal{R}$ un TRS inductivamente secuencial. Sea t un término encabezado por un símbolo de operación y $\mathcal{P}$ un árbol definicional con $pattern(\mathcal{P}) = \pi$ tal que $\pi \leq t$. Definimos una aplicación λ de términos y árboles definicionales a conjuntos de ternas (posición, regla, substitución) como el menor conjunto que satisface las siguientes propiedades. Consideramos dos casos para $\mathcal{P}$:*

1. *Si π es una hoja, i.e., $\mathcal{P} = \{\pi\}$, y $(\pi \to r) \ll \mathcal{R}$, entonces $\lambda(t, \mathcal{P}) = \{\langle \Lambda, \pi \to r, id\rangle\}$.*

2. *Si π es una rama, considerar la posición inductiva o de π y un hijo $\pi_i = \pi[c_i(x_1, \ldots, x_n)]_o \in \mathcal{P}$. Sea $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ el árbol definicional donde todos los patrones son instancias de π_i. Entonces consideramos los siguientes casos para el subtérmino $t|_o$:*

$$\lambda(t,\mathcal{P}) \ni \begin{cases} \langle p, R, \sigma \circ \tau\rangle & \texttt{si } t|_o = x \in \mathcal{X},\ \tau = \{x/c_i(x_1,\ldots,x_n)\}, \\ & \quad \texttt{y } (p,R,\sigma) \in \lambda(\tau(t), \mathcal{P}_i); \\ \langle p, R, \sigma \circ id\rangle & \texttt{si } t|_o = c_i(t_1,\ldots,t_n) \texttt{ y } (p,R,\sigma) \in \lambda(t,\mathcal{P}_i); \\ \langle o.p, R, \sigma \circ id\rangle & \texttt{si } t|_o = f(t_1,\ldots,t_n),\ f \in \mathcal{F} \texttt{ y } (p,R,\sigma) \in \lambda(t|_o,\mathcal{P}') \\ & \quad \texttt{donde } \mathcal{P}' \text{ es un árbol definicional para } f. \end{cases}$$

Informalmente, el *narrowing* necesario aplica una regla si es posible (caso 1) o comprueba los subtérminos correspondientes a las posiciones inductivas de una rama (caso 2): si dicho subtérmino es una variable, queda instanciada con el constructor de un hijo (formando lo que denominamos un enlace de *substitución adelantada*); si ya es un constructor, procede con el hijo que le corresponde (aquél que en la misma posición inductiva presenta un término constructor plano encabezado por el mismo símbolo constructor); si es una función, se evalúa aplicando recursivamente la definición de estrategia de *narrowing* necesario. Así pues, esta estrategia difiere de otras estrategias perezosas en la instanciación de variable libres mediante substituciones adelantadas, lo cual hace posible el cómputo de respuestas que no son unificadores más generales.

Al computar los pasos de *narrowing* necesario para un término t encabezado por un símbolo de función f, tomamos el árbol definicional $\mathcal{P}$ de f y calculamos $\lambda(t, \mathcal{P})$. Entonces, para cada $\langle p, l \to r, \sigma\rangle \in \lambda(t, \mathcal{P})$, el paso $t \overset{p,R,\sigma}{\leadsto}_{NN} \sigma(t[r]_p)$ es un *paso de narrowing necesario.*

[16] La estrategia λ de la Definición 2.9.20 computa las mismas ternas que la estrategia original, definida en [23], si en las substituciones nos restringimos a las variables del término inicial y no tenemos en cuenta los posibles renombramientos de variables. Para que la Definición 2.9.20 se ajuste exactamente a la original, basta hacer que, en el punto (1), $\lambda(t, \mathcal{P})$ devuelva $\{\langle \Lambda, \pi \to r, mgu(t, \pi)\rangle\}$ en lugar de $\{\langle \Lambda, \pi \to r, id\rangle\}$.

Ejemplo 7 *Consideremos, de nuevo, las reglas para "$\leq$" y "$+$" del Ejemplo 6. Entonces, la estrategia λ computa para el término $X \leq X + X$ el siguiente conjunto de ternas:*

$$\{\langle \Lambda, 0 \leq N \to true, \{X/0\}\rangle,\ \langle 2, s(M) + N \to s(M+N), \{X/s(M)\}\rangle\}$$

que se corresponden con los siguientes pasos de narrowing*:*

$$\begin{array}{lll} X \leq X + X & \overset{\{X/0\}}{\leadsto}_{NN} & true \\ X \leq X + X & \overset{\{X/s(M)\}}{\leadsto}_{NN} & s(M) \leq s(M + s(M)) \end{array}$$

La comparación de estos pasos de narrowing *con los de las derivaciones obtenidas en el Ejemplo 6, nos muestra que el* narrowing *necesario solamente da pasos necesarios para el cómputo de una respuesta.*

En lo que sigue, resumimos una serie de propiedades interesantes de la estrategia de *narrowing* necesario que nos serán muy útiles en el futuro. La primera proposición muestra que cada substitución en un paso de *narrowing* necesario instancia solamente variables que aparecen en el término inicial. Antes de presentar esta proposición, reparemos en que, en cada paso recursivo i durante el cómputo de λ, componemos la subtitución actual ϑ_i (que puede ser la identidad) con las computadas en las llamadas recursivas previas $\vartheta_k \circ \cdots \circ \vartheta_{i-1}$. De este modo, cada paso de *narrowing* necesario puede representarse como $(p, R, \vartheta_k \circ \cdots \circ \vartheta_1)$, que denominamos *representación canónica* de un paso de *narrowing* necesario. Como en los procedimientos de prueba de la programación lógica, suponemos que los árboles definicionales siempre contienen variables frescas cuando se emplean en el cómputo de un paso de *narrowing*. Esto supone que todas las substituciones computadas son idempotentes.

Proposición 2.9.21 *[16] Si $(p, R, \vartheta_k \circ \cdots \circ \vartheta_1) \in \lambda(t, \mathcal{P})$ es un paso de* narrowing *necesario, entonces, para $i = 1, \ldots, k$, se tiene que $\vartheta_i = id$ o bien $\vartheta_i = \{x/c(x_1, \ldots, x_n)\}$ (donde $x_1, \ldots, x_n$ son variables distintas) con $x \in \mathcal{V}ar(\vartheta_{i-1} \circ \cdots \circ \vartheta_1(t))$.*

Se debe observar que de la Proposición 2.9.21 se deriva que los términos $c(x_1, \ldots, x_n)$ del rango de las substituciones ϑ_i son constructores planos lineales y, puesto que las variables $x_1, \ldots, x_n$ provienen de reglas renombradas aparte del programa, no comparten variables con otros términos del rango de otras substituciones ϑ_j con $i \neq j$. Notad también que, si $\vartheta_i = \{x/c(x_1, \ldots, x_n)\}$, dado que $x \in \mathcal{V}ar(\vartheta_{i-1} \circ \cdots \circ \vartheta_1(t))$, la variable x puede pertenecer a las variables del rango de los componentes ϑ_j (con $j \leq i$) hallados previamente.

Corolario 2.9.22 *Sea $\mathcal{R}$ un programa inductivamente secuencial, y t y s dos términos. Si $\mathcal{D} \equiv (t \overset{\sigma}{\leadsto}_{NN}^* s)$ es una derivación de* narrowing *necesario, entonces para todo par de variables distintas $x, y \in \mathcal{D}om(\sigma_{\restriction \mathcal{V}ar(t)})$, $\sigma(x)$ y $\sigma(y)$ son términos constructores lineales que no comparten variables.*

Prueba. Por inducción en el número de pasos de la derivación $\mathcal{D}$.

1. Caso base ($n = 1$). La derivación $\mathcal{D}$ es de un sólo paso $t \overset{[p,R,\sigma]}{\rightsquigarrow}_{NN} s$. Por definición de paso de *narrowing* necesario y la Proposición 2.9.21, los términos del rango de la susbtitución σ son constructores lineales que no comparten variables.

2. Caso inductivo ($n > 1$). La derivación $\mathcal{D}$ tiene longitud n

$$t \equiv s_0 \overset{[p_1,R_1,\sigma_1]}{\rightsquigarrow}_{NN} s_1 \overset{[p_2,R_2,\sigma_2]}{\rightsquigarrow}_{NN} \dots \overset{[p_n,R_n,\sigma_n]}{\rightsquigarrow}_{NN} s_n \equiv s,$$

donde $\sigma = \sigma_n \circ \ldots \circ \sigma_2 \circ \sigma_1$. Ahora podemos dividir la derivación $\mathcal{D}$ en dos partes:

$$t \equiv s_0 \overset{\sigma'}{\rightsquigarrow}_{NN}{}^* s_{n-1} \overset{[p_n,R_n,\sigma_n]}{\rightsquigarrow}_{NN} s_n \equiv s,$$

donde $\sigma' = \sigma_{n-1} \circ \ldots \circ \sigma_1$. Notad también que, si $R_i \equiv l_i \to r_i$, entonces $s_{n-1} \equiv \sigma_{n-1}(\ldots(\sigma_2(\sigma_1(s_0[r_1]_{p_1})[r_2]_{p_2}\ldots)[r_{n-1}]_{p_{n-1}})$. Por lo que, si $x \in \mathcal{D}om(\sigma')$, entonces $x \notin \mathcal{V}ar(s_{n-1})$. Por consiguiente, teniendo en cuenta la aclaración anterior y el hecho de que las reglas del programa se renombran aparte, $\mathcal{D}om(\sigma') \cap \mathcal{D}om(\sigma_n) = \emptyset$. Asi pues, por definición de composición de substituciones, $\sigma_{\lceil \mathcal{V}ar(t)} = (\sigma_n \circ (\sigma'_{\lceil \mathcal{V}ar(t)}) \cup \sigma_{n \lceil \mathcal{V}ar(t)}$.

   Por hipotesis de inducción, para todo par de variables x e y distintas de $\mathcal{D}om(\sigma'_{\lceil \mathcal{V}ar(t)})$, los términos $\sigma'(x)$ y $\sigma'(y)$ son constructores lineales que no comparten variables. Además, las variables de los términos del rango de $\sigma'_{\lceil \mathcal{V}ar(t)}$ son variables que provienen de reglas renombradas aparte R_i de $\mathcal{R}$ (con $i < n$).

   Por la Proposición 2.9.21, los términos del rango de $\sigma_{n \lceil \mathcal{V}ar(s_{n-1})}$ son constructores lineales que no comparten variables entre ellos. Tampoco con los términos del rango de $\sigma'_{\lceil \mathcal{V}ar(t)}$, ya que las variables que aparecen en el rango de $\sigma_{n \lceil \mathcal{V}ar(s_{n-1})}$ son variables nuevas que provienen de la regla renombrada aparte R_n. En particular, los términos del rango de $\sigma_{n \lceil \mathcal{V}ar(t)}$ son constructores lineales que no comparten variables entre ellos, ni con los términos del rango de $\sigma'_{\lceil \mathcal{V}ar(t)}$.

   Por otro lado, las variables de $\mathcal{D}om(\sigma_n)$ son variables del término s_{n-1} o variables nuevas que provienen de la regla renombrada aparte R_n. Esto es, $\sigma_n = (\sigma_{n \lceil \mathcal{V}ar(s_{n-1})}) \cup \sigma_{n \lceil \mathcal{V}ar(R_n)}$. Entonces, la composición de $\sigma'_{\lceil \mathcal{V}ar(t)}$ con σ_n siempre introduce subterminos del rango de $\sigma_{n \lceil \mathcal{V}ar(s_{n-1})}$ que, como ya hemos dicho, son constructores lineales que no comparten variables ni con los términos del rango de $\sigma'_{\lceil \mathcal{V}ar(t)}$ ni con los otros términos del rango de $\sigma_{n \lceil \mathcal{V}ar(t)}$. Por consiguiente, los términos del rango de $\sigma_{\lceil \mathcal{V}ar(t)}$ son constructores lineales que no comparten variables.

□

Para programas inductivamente secuenciales, el *narrowing* necesario es correcto y completo con respecto a ecuaciones estrictas y substituciones constructoras como solución de dichas ecuaciones. Más aún, el *narrowing* necesario no computa soluciones redundantes. Estas propiedades se formalizan en el siguiente teorema:

Teorema 2.9.23 *[23] Sea $\mathcal{R}$ un programa inductivamente secuencial y e una ecuación.*

1. *(Corrección) Si $e \overset{\sigma}{\leadsto}_{NN}{}^{*}$ true es una derivación de* narrowing *necesario, entonces σ es una solución para e.*

2. *(Completitud) Para cada substitución constructora σ que es una solución de e, existe una derivación de* narrowing *necesario $e \overset{\sigma'}{\leadsto}_{NN}{}^{*}$ true con $\sigma' \leq \sigma\ [\mathcal{V}ar(e)]$.*

3. *(Minimalidad) Si $e \overset{\sigma}{\leadsto}_{NN}{}^{*}$ true and $e \overset{\sigma'}{\leadsto}_{NN}{}^{*}$ true son dos derivaciones de* narrowing *necesario distintas, entonces σ y σ' son independientes (i.e., existe al menos un $x \in \mathcal{V}ar(e)$ tal que $\sigma(x) \neq \sigma'(x)$).*

2.9.5 Narrowing Normalizante.

Una mejora importante de la relación de *narrowing* se consigue mediante la introducción de pasos de reescritura, generalmente hasta alcanzar una forma normal, después (o antes) de cada paso de *narrowing*. Llamaremos *narrowing normalizante* a esta variante del *narrowing*, que fue denominada simplemente *narrowing* por Fay [68] y más tarde *narrowing normal* por Réty [175].

Definición 2.9.24 *Un paso de* narrowing normalizante *con respecto a un TRS $\mathcal{R}$, denotado $t \overset{\sigma}{\leadsto} t'{\downarrow}$, consiste en un paso de* narrowing *$t \overset{\sigma}{\leadsto} t'$ seguido por una secuencia de normalización $t' \to_{\mathcal{R}}^{*} t'{\downarrow}$.*

La optimización que se acaba de introducir es importante, ya que el *narrowing* normalizante reduce el indeterminismo en las derivaciones de *narrowing*: la normalización de un término puede realizarse de un modo determinista gracias a que cada secuencia de reescritura conduce al mismo resultado, si el TRS es canónico. Además, es útil aplicar pasos de reescritura entre pasos de *narrowing*, siempre que sea posible, ya que los pasos extra de reescritura pueden transformar un espacio de búsqueda infinito en otro finito, ahorrando una gran cantidad de tiempo y espacio de almacenamiento en memoria [93]. Por otra parte, como se ha mostrado en [72], el *narrowing* normalizante permite que los programas lógico–funcionales puedan ser ejecutados más eficientemente que los correspondientes programas lógicos (puros). También, la preferencia por los pasos deterministas de reescritura permite la eliminación de algunas características extralógicas de los lenguajes de programación lógicos, como es el operador de *corte* de PROLOG [93, 96]: en concreto, los pasos de reescritura permiten eliminar, de una forma *segura*[17], algunos *puntos de elección* (también denominados *puntos de vuelta atrás – backtracking*) en una implementación secuencial del algoritmo de *narrowing*, produciéndose un efecto similar al que proporciona el *corte dinámico* (*dynamic cut*) de Loogen y Winkler [139].

[17]Esto es, sin perder soluciones.

La idea de explotar cómputos deterministas mediante la inclusión de pasos de normalización se ha aplicado a la mayoría de las estrategias de *narrowing* sin pérdida de completitud (e.g. *narrowing* básico [99, 175], *narrowing innermost* [72] y *narrowing* básico *innermost* [99]).

Hanus [93] ha mostrado cómo pueden realizarse pasos deterministas de reescritura (que él denomina pasos de *simplificación*) entre pasos indeterministas de *narrowing* perezoso, sin perder la completitud. La simplificación del término a evaluar, aplicando las reglas de reescritura, puede hacerse con cualquier estrategia de reducción, si el conjunto de reglas es terminante. Sin embargo, en presencia de funciones no terminantes, un proceso de simplificación arbitrario puede destruir la completitud de la estrategia de *narrowing* perezoso. Para evitar este problema, una solución muy simple es emplear solamente un subconjunto terminante de las reglas del programa cuando se utilizan para la normalización[18]. Dado que *narrowing* perezoso es completo sin simplificación, no es necesario realizar pasos de reescritura con todas las posibles reglas del programa, de forma que podemos restringirnos a un subconjunto cualquiera; por conveniencia, a uno que sea terminante.

En el ejemplo 15 del Apartado 4.1 se muestra como la inclusión de pasos de normalización en la estrategia de *narrowing* perezoso, no solamente mejora la eficiencia del proceso de evaluación parcial sino que también mejora el grado de especialización obtenido para los programas lógico–funcionales.

[18]En el caso de considerar TRS's condicionales como programas, se necesita el requisito adicional de *reglas decrecientes* (*decreasing*) [93].

Capítulo 3

Evaluación Parcial y Especialización de Programas Lógico-Funcionales.

La *evaluación parcial* (PE, del inglés *Partial Evaluation*) es una técnica de transformación de programas, también conocida con el nombre de *especialización de programas*. Algunos temas centrales de la evaluación parcial son la obtención de *generadores de generadores de programas* así como la transformación de programas para su optimización con respecto a ciertos datos de entrada; de ahí el nombre de *especialización*. Contrariamente a lo que sucede con otras técnicas de transformación de programas [48, 164], el proceso de evaluación parcial puede automatizarse completamente pero, por contra, la evaluación parcial consigue menores mejoras en la eficiencia de los programas especializados. La evaluación parcial ha sido aplicada intensivamente tanto a los lenguajes imperativos como a los declarativos y extensivamente a una gran variedad de problemas concretos (ver [108] para una panorámica general sobre el área y su marco de aplicación).

En este capítulo definiremos con precisión el concepto de evaluación parcial y nos ocuparemos de sus principales técnicas y de los problemas que se plantean, haciendo énfasis en la especialización de programas lógico–funcionales.

3.1 Evaluación Parcial.

La idea de especializar funciones con respecto a uno o varios de sus argumentos es ya vieja en el campo de la teoría de funciones recursivas, donde recibe el nombre de *proyección*. Esta posibilidad está contenida en el teorema s-m-n de Kleene [111] que establece que, dada una función computable $f \in \mathcal{F}^{n+m}$ tal que $f : S^{n+m} \to S$, se cumple que:

1. la función $f_{e_1,\ldots,e_n} : S^m \to S$, tal que

$$f_{e_1,\ldots,e_n}(d_1,\ldots,d_m) = f(e_1,\ldots,e_n,d_1,\ldots,d_m),$$

   es computable;

2. la función de orden superior $f^* : \mathcal{F}^{n+m} \times S^n \to \mathcal{F}^m$, tal que

$$f^*(f,e_1,\ldots,e_n) = f_{e_1,\ldots,e_n},$$

   es computable;

Ejemplo 8 *Dada la función de dos argumentos plus* $: I\!N^2 \to I\!N$, *definida como* $plus(x,y) = x+y$, *puede especializarse si el valor de* x *es conocido e igual a* 2, *transformándose en una función* $plus_2 : I\!N \to I\!N$, *tal que* $plus_2(y) = 2+y$.

Se debe notar que simplemente se substituyen los valores conocidos por los correspondientes parámetros formales de una función; no se realiza ningún cómputo. Además, en este proceso, que recuerda a la *currificación* de funciones en los lenguajes funcionales, el objetivo de lograr una mayor eficiencia está ausente. Los problemas de eficiencia eran irrelevantes para las investigaciones efectuadas por Kleene, centradas en establecer los límites entre lo que es o no es computable. El hecho es que la técnica propuesta por Klenee produce funciones especializadas que se comportan, en algunos casos, "peor" computacionalmente que las originales. Por otro lado, la evaluación parcial trabaja con programas (textos) más bien que con funciones matemáticas. Por consiguiente, la evaluación parcial es una técnica que va más allá de la simple proyección de funciones matemáticas.

La *evaluación parcial* establece cómo ejecutar un programa cuando sólo conocemos parte de sus datos de entrada. De forma más precisa, dado un programa P y parte de sus datos de entrada in_1, el objetivo de la evaluación parcial es construir un nuevo programa P_{in_1} que cuando se le introduce el resto de los datos de entrada in_2, computa el mismo resultado que P produce procesando toda su entrada $(in_1 + in_2)$. Esto último asegura la corrección de la transformación efectuada. El programa que realiza el proceso de evaluación parcial recibe el nombre de *evaluador parcial* y el resultado de la evaluación parcial, el programa P_{in_1}, se denomina *programa especializado*, *programa evaluado parcialmente* o también *programa residual*. La idea que se esconde detrás del proceso de evaluación parcial consiste en: i) realizar tantos cómputos como sea posible en tiempo de evaluación parcial, haciendo uso de los datos de entrada conocidos in_1, también denominados *datos estáticos* (por contraposición con los datos de entrada desconocidos in_2, que son denominados *datos dinámicos*, y que sólo son conocidos en tiempo de ejecución del programa residual); ii) generar código relacionado con aquellos cálculos que no puedan realizarse por depender de los datos de entrada desconocidos. Así pues, un evaluador parcial realiza una mezcla de acciones de cómputo y de generación de código; esta es la razón por la que Ershov denominó a la evaluación parcial *computación mixta* (*mixed computation*).

Para cumplir estos fines la evaluación parcial utiliza, además de la computación simbólica, algunas técnicas bien conocidas provenientes de la transformación de programas [48], procurando su automatización:

1. Definición: permite la introducción de funciones nuevas o la extensión de las existentes; si las funciones están definidas por reglas, da la posibilidad de introducir reglas nuevas, i.e. para un programa $\mathcal{R}$, si no existe ninguna regla $l_1 \to r_1 \in \mathcal{R}$ tal que l_1 solape con l_2 entonces la regla $l_2 \to r_2$ puede emplearse en el proceso de transformación.

2. Instanciación: permite especializar una función asignando datos de entrada conocidos a sus argumentos; si suponemos que la función está definida por reglas de reescritura, hace posible la instanciación de reglas aplicando substituciones, i.e. si $l \to r \in \mathcal{R}$ y θ es una substitución, entonces la regla $\theta(l) \to \theta(r)$ puede emplearse en el proceso de transformación.

3. Desplegado (*unfolding*): permite el reemplazamiento de una llamada a función por su respectiva definición; nuevamente, si la función está definida por reglas de reescritura, tendremos que si $l_1 \to r_1 \in \mathcal{R}$ y $l_2 \to C[\theta(l_1)] \in \mathcal{R}$ entonces la regla $l_2 \to C[\theta(r_1)]$ puede emplearse en el proceso de transformación.

4. Plegado (*folding*): como su nombre indica, es la transformación inversa del desplegado, es decir, el reemplazamiento de cierto fragmento de código por la correspondiente llamada a función; si $l_1 \to r_1 \in \mathcal{R}$ y $l_2 \to C[\theta(r_1)] \in \mathcal{R}$ entonces la regla $l_2 \to C[\theta(l_1)]$ puede emplearse en el proceso de transformación.

De entre estas técnicas, la evaluación parcial hace uso intensivo del desplegado como su herramienta de transformación fundamental. Una técnica específica empleada en la evaluación parcial es la denominada *especialización de puntos de control* del programa (*program point specialization*), que combina definición y plegado. En un lenguaje imperativo un *punto de control* es una etiqueta del programa; en un lenguaje declarativo puede considerarse que es la definición de una función o predicado. La idea es que una etiqueta o una función del programa P pueda aparecer en el programa especializado P_{in_1} en varias versiones especializadas, cada una correspondiente a datos determinados en tiempo de evaluación parcial. Otra de las técnicas empleadas por la evaluación parcial es la *abstracción*, consistente en generalizar una expresión; en cierto sentido, puede verse como la transformación inversa de la instanciación y puede caracterizarse en términos de un proceso de definición al que le sigue uno de plegado [10, 181].

Ejemplo 9 *Sea el fragmento de un programa P que computa la función x^n:*

$$pow(0, X) \to 1$$
$$pow(N, X) \to X * pow(N-1, X)$$

Intuitivamente, podemos considerar que un evaluador parcial es un programa que realiza automáticamente una serie de transformaciones como las definidas *anteriormente y que consigue la optimización del programa original para una clase de datos de entrada. Si queremos especializar el programa para el dato de entrada conocido $N = 3$, los pasos que podría realizar un hipotético evaluador parcial serían:*

`% Definición`
$pow3(X) \to pow(3, X)$
`% Instanciación, desplegado y computación simbólica`
$pow(3, X) \to X * pow(3 - 1, X)$
$pow(3, X) \to X * pow(2, X)$
$pow(3, X) \to X * (X * pow(2 - 1, X))$
$pow(3, X) \to X * (X * pow(1, X))$
$pow(3, X) \to X * (X * (X * pow(1 - 1, X)))$
$pow(3, X) \to X * (X * (X * pow(0, X)))$
$pow(3, X) \to X * (X * (X * 1))$
$pow(3, X) \to X * (X * X)$
`% Desplegado final`
$pow3(X) \to X * (X * X)$

conduciendo al programa especializado P_3

$$pow3(X) \to X * (X * X)$$

que computa la función x^3.

En este ejemplo no ha sido necesaria la especialización de puntos de control, incluyendo pasos de plegado, debido a que el programa especializado no contiene llamadas a función. Tampoco se han requerido pasos de abstracción.

Un tema importante relativo a los evaluadores parciales es su capacidad para reestructurar los puntos de control. La reestructuración de puntos de control (*control restructuring*) tiene que ver con las relaciones que se establecen entre los puntos de control del programa original y los del programa residual. En la nomenclatura de [87] podemos distinguir las siguientes capacidades de reestructuración:

- *Monovariante*: cualquier punto de control del programa original da lugar, como mucho, a un punto de control en el programa residual; "como mucho" quiere decir que un punto de control del programa original puede desaparecer (i.e., da lugar a cero puntos de control) en el programa residual.

- *Polivariante*: cualquier punto de control del programa original puede dar lugar a uno o más puntos de control en el programa residual.

- *Monogenético*: cualquier punto de control del programa residual se produce a partir de un único punto de control del programa original.

- *Poligenético*: cualquier punto de control del programa residual se produce a partir de uno o más puntos de control del programa original.

3.1.1 Corrección de la Evaluación Parcial.

Antes de seguir adelante, conviene introducir uno de los temas más relevantes dentro de la evaluación parcial, el de la *corrección*[1] de la misma. A este respecto hay dos puntos a tratar: la corrección semántica (i.e., la preservación de la semántica operacional) y el control de la terminación.

Comenzaremos discutiendo la corrección semántica de la evaluación parcial. Con el fin de formalizar este concepto en un marco general, se introducen las siguientes suposiciones y notaciones estándares [108], que serán útiles en éste y en el próximo subapartado. Vamos a tratar con diversos lenguajes; emplearemos las letras L, S y T para referirnos, respectivamente, a un lenguaje de implementación, a un lenguaje fuente y a un lenguaje objeto. Denotaremos por D el conjunto de los datos que pueden pasarse como valores a un programa, incluyendo también los textos que forman los programas. Supondremos que suministramos listas de datos como entrada para los programas; denotaremos por D^* el conjunto de todas las listas que pueden formarse a partir de los elementos de D. Suponemos que la semántica de los lenguajes que empleamos está definida por una semántica operacional (no especificada), en la que los cómputos se realizan mediante una secuencia de instrucciones que producen cambios de estado (si los lenguajes son imperativos), o más genéricamente, mediante deducción aplicando ciertas reglas de inferencia. Si P es un programa en un lenguaje L, entonces $[\![P]\!]_L$ denota el significado del programa P escrito en el lenguaje L. El significado del programa se establece como una función parcial $[\![P]\!]_L : D^* \to D \cup \{\perp\}$ tal que si $in \in D^*$ entonces

$$out = [\![P]\!]_L(in)$$

siendo $out \in D$ el resultado de ejecutar P para la entrada in; el valor de out es $\perp$ (indefinido) cuando la ejecución del programa P no termina.

Supongamos un programa fuente P al que se suministra la entrada de datos estática e, conocida previamente, y la entrada de datos dinámica d, conocida con posterioridad. Entonces definimos la evaluación parcial de P, utilizando un evaluador parcial $Spec$ mediante la ecuación:

$$P_e = [\![Spec]\!]_L([P, e]).$$

Definición 3.1.1
Supuesto que el proceso de evaluación parcial termina, decimos que un evaluador parcial es:

1. **Correcto**
   si y sólo si para toda entrada dinámica $d \in D$, $[\![P_e]\!]_T(d) = out$ implica que $[\![P]\!]_S([e, d]) = out$ (i.e., si out es una respuesta computada por P_e entonces también es una respuesta computada por P.).

[1]Se ha empleado la palabra castellana "corrección" como traducción de la palabra inglesa "*correctness*", que engloba los significados de *soundness* (corrección) y *completeness* (completitud). Dado que la palabra inglesa "*soundness*" también se ha traducido al castellano como "corrección", puede crearse cierta confusión semántica en algunos momentos, que el lector sabrá dilucidar por el contexto.

2. **Completo**

si y sólo si para toda entrada dinámica $d \in D$, $[\![P]\!]_S([e,d]) = out$ implica que $[\![P_e]\!]_T(d) = out$ (i.e., si out es una respuesta computada por P entonces también es una respuesta computada por P_e.).

Los conceptos de corrección y completitud se han definido en un sentido *fuerte*[2], de forma que si un evaluador parcial es correcto y completo, podemos establecer la identidad semántica entre el programa especializado y el original. Esto es, se cumple que

$$[\![P_e]\!]_T(d) = [\![P]\!]_S([e,d]),$$

para cada entrada dinámica $d \in D$. En palabras, P y P_e computan las mismas respuestas (salidas).

El control de la terminación es un problema crucial en el ámbito de la evaluación parcial. La no terminación es un comportamiento indeseable para una herramienta que pretende la optimización automática de programas. La no terminación del proceso de evaluación parcial puede producirse por una de las siguientes razones:

1. Un intento de construir una instrucción, una expresión, o en general una estructura infinita.

2. Un intento de construir un programa residual que contenga infinitos puntos de control (etiquetas, procedimientos, funciones definidas o predicados).

La causa última de este comportamiento es la misma: un fallo en la estrategia de desplegado que emplea el evaluador parcial [107]. En cualquier caso, la no terminación hace que un evaluador parcial sea poco aceptable para su uso por parte de un usuario inexperto, y completamente inaceptable para su uso como herramienta de generación automática. Para asegurar la terminación de la evaluación parcial, la mayoría de los evaluadores parciales mantienen durante el proceso de especialización una historia de la computación, que permite su consulta a la hora de tomar decisiones sobre si desplegar una expresión o no; y, en caso de que no, cómo realizar el plegado para especializar la expresión o saber si hay que abstraer, convirtiéndola en una expresión más general. Existen varios compromisos que deben tenerse en consideración cuando realizamos la evaluación parcial: desplegar con suma liberalidad puede llevar al problema (1), dando lugar a un tiempo de especialización infinito y a la no generación de un programa residual; generar expresiones residuales demasiado especializadas (i.e., conteniendo demasiados datos estáticos) puede conducir a un programa residual infinito y, por lo tanto, a la aparición del problema (2); en el otro extremo, generar expresiones residuales demasiado generales puede hacer perder toda la especialización, obteniéndose programas residuales que son (esencialmente) el programa original sin especializar. Ser conscientes de los aspectos que acabamos

[2]En el Apartado 3.3.1 se presenta una noción de corrección y completitud *débil* para el caso concreto de un lenguaje lógico–funcional.

de comentar ha permitido distinguir dos niveles de control en el problema de la terminación de la evaluación parcial [145] que, en buena medida, pueden ser abordados de forma independiente: el llamado *control local*, asociado con el punto (1); y el llamado *control global*, asociado con el punto (2).

Un aspecto relacionado con el problema de la terminación es el de la *precisión* [126], entendiendo por tal la obtención del máximo potencial especializador. Aquí, también podemos distinguir dos niveles: el nivel de *precisión local*, asociado con el desplegado de una expresión y el hecho de que puede perderse potencial para la especialización si se detiene el desplegado de la expresión demasiado pronto (o demasiado tarde); y un nivel de *precisión global* que está asociado al número de puntos de control especializados, en general disponer de un programa residual con el mayor número de puntos de control especializados con respecto a una variedad de datos estáticos conducirá a una mejor especialización.

Como puede apreciarse, el control de la evaluación parcial ofrece aspectos que pueden entrar en conflicto, como son el de la terminación y el de la precisión. Un buen algoritmo de evaluación parcial debe asegurar la corrección y la terminación mientras minimiza las pérdidas de precisión.

Por último, decir que cuando un evaluador parcial, en el supuesto de que termina, genera un programa residual semánticamente equivalente al original, se habla de *corrección parcial*. Si además de generar un programa residual semánticamente equivalente al original, termina cualquiera que sea la circunstancia, se habla de *corrección total*.

Volveremos sobre todos estos problemas más adelante, planteando técnicas concretas que permitan dar solución a los mismos en el contexto de los programas integrados.

3.1.2 Evaluación Parcial y Generación Automática de Programas.

En este apartado formalizamos el concepto de evaluador parcial, lo que nos permite establecer relaciones interesantes entre la evaluación parcial y la generación automática de programas.

Para establecer la propiedad esencial que caracteriza a un evaluador parcial de una manera formal, supongamos un programa fuente P al que se suministra la entrada de datos estática $in1$ y la entrada de datos dinámica $in2$. Entonces podemos describir el cómputo (del resultado) en un paso mediante la ecuación,

$$out = [\![P]\!]_S([in1, in2])$$

y el cómputo (del resultado) en dos pasos, utilizando un evaluador parcial *Spec* mediante las ecuaciones

$$\begin{array}{rcl} P_{in1} & = & [\![Spec]\!]_L([P, in1]) \\ out & = & [\![P_{in1}]\!]_T(in2). \end{array}$$

Combinando estas ecuaciones se obtiene una *definición ecuacional* del evaluador parcial *Spec* :

$$[\![P]\!]_S([in1, in2]) = [\![[\![Spec]\!]_L([P, in1])]\!]_T(in2)$$

donde, si una parte de la ecuación está definida, también lo estará la otra y con el mismo valor. De estas ecuaciones también se desprende que el evaluador parcial puede estar escrito en un lenguaje de implementación L, tener como entrada un programa P escrito en un lenguaje fuente S, y producir como salida un programa especializado escrito en un lenguaje objeto[3] T. Si el evaluador parcial, escrito en un lenguaje L, admite como entrada programas también escritos en L, se dice que es *autoaplicable*. La capacidad de autoaplicación posibilita la escritura de evaluadores parciales que puedan especializarse a sí mismos. Si bien éste es un tema de gran interés dentro del campo de la evaluación parcial, no cae dentro de los objetivos de esta tesis. Cuando solamente se emplea un lenguaje en la discusión, los subíndices pueden eliminarse obteniéndose la siguiente ecuación simplificada:

$$[\![P]\!]([in1, in2]) = [\![[\![Spec]\!]([P, in1])]\!](in2)$$

La definición ecuacional del evaluador parcial nos permite poner en relación los conceptos de interpretación, compilación y evaluación parcial. Primeramente, nótese que un intérprete es un programa Int, escrito en un lenguaje L, que puede ejecutar un programa *Fuente* en un lenguaje S junto con sus datos de entrada in. En símbolos,

$$[\![Fuente]\!]_S(in) = [\![Int]\!]_L([Fuente, in]),$$

que es la ecuación de definición de un *intérprete* Int para S escrito en L. Un compilador es un programa $Comp$, escrito en un lenguaje L, que genera un programa *Objeto* en un lenguaje T. En símbolos,

$$Objeto = [\![Comp]\!]_L(Fuente),$$

que es la ecuación que define un programa *Objeto*. El efecto de ejecutar el programa *Fuente* sobre los datos de entrada in se consigue, una vez realizada la compilación, ejecutando el programa *Objeto* con los datos de entrada in,

$$[\![Fuente]\!]_S(in) = [\![Objeto]\!]_T(in).$$

Combinando estas ecuaciones obtenemos la definición de un *compilador* $Comp$ de S a T escrito en L,

$$[\![Fuente]\!]_S(in) = [\![[\![Comp]\!]_L(Fuente)]\!]_T(in)$$

Ahora pueden resumirse las capacidades de la evaluación parcial para la *generación automática* de programas mediante la siguiente proposición.

[3]Posiblemente un lenguaje máquina, si el interés primordial fuese el incremento de la eficiencia del programa especializado con respecto al original.

Proposición 3.1.2 (Proyecciones de Futamura) *Sean Spec un especializador de L a T escrito en L, e Int un intérprete para S escrito en L.*

- *Primera proyección:* $Objeto = [\![Spec]\!]_L([Int, Fuente])$
- *Segunda proyección:* $Comp = [\![Spec]\!]_L([Spec, Int])$
- *Tercera proyección:* $Cogen = [\![Spec]\!]_L([Spec, Spec])$

Estas ecuaciones, aunque sencillas de probar (ver [108]), no son fáciles de entender intuitivamente. La primera de ellas indica que se puede compilar un programa *Fuente* especializando su intérprete *Int* con respecto a dicho programa *Fuente*. La segunda dice que se puede obtener un compilador mediante autoaplicación del evaluador parcial, i.e., especializando el propio *Spec* con respecto a un intérprete *Int*. La tercera establece que *Cogen* es un generador de compiladores, que transforma un intérprete en un compilador, i.e., $Comp = [\![Cogen]\!]_T(Int)$. Así pues, la evaluación parcial permite la compilación (primera proyección), la generación de compiladores (segunda proyección) y la generación de generadores de compiladores (tércera proyección).

3.1.3 Objetivos de la Evaluación Parcial.

Aunque en este apartado discutiremos diversos objetivos de la evaluación parcial, para nosotros la principal motivación de la evaluación parcial es el aumento en la *eficiencia* (*speedup*) de los programas. Debido a que parte de los cómputos se han realizado previamente, en tiempo de evaluación parcial, esperamos que el programa especializado sea más rápido que el programa original. Es común realizar una medida del aumento de la eficiencia, obteniendo la razón entre el tiempo de ejecución del programa original y del especializado [108, 107]. Antes de describir con precisión el concepto de eficiencia necesitamos la siguiente notación [107]: dado un programa P y una de sus entradas $in \in D$, $time_P(in)$ es el tiempo para computar $[\![P]\!]_S(in)$. Los tiempos de ejecución pueden medirse empleando diferentes unidades, ya sea el número de ciclos consumidos para ejecutar el programa en un determinado computador, el número de operaciones elementales realizadas en la ejecución del programa, o bien el número de milisegundos transcurridos desde que comienza hasta que termina la ejecución del programa.

Definición 3.1.3 (Eficiencia)
Sean P un programa, escrito en un lenguaje S, y $e, d \in D^*$ los datos de entrada estáticos y dinámicos que se le suministran, respectivamente. Sea Spec un evaluador parcial de S a T escrito en L. Si $P_e = [\![Spec]\!]_L([P, e])$ es el programa especializado por Spec para la entrada e, (el aumento en) la eficiencia *obtenida por Spec sobre el programa P y la entrada estática e, denotado $speedup(P, e, d)$, viene dado por la razón:*

$$speedup(P, e, d) = \frac{time_P([e, d])}{time_{P_e}(d)}$$

Nótese, que para un programa P y una entrada estática e fijados, *speedup* es una función que depende exclusivamente de la entrada dinámica d.

A la hora de medir la eficiencia de la evaluación parcial debe considerarse el coste del tiempo de especialización, $time_{Spec}([P, e])$, del programa original (i.e., el tiempo necesario para computar $[\![Spec]\!]_L([P, e])$). La evaluación parcial es claramente ventajosa cuando un (procedimiento de un) programa debe ejecutarse reiteradamente para una porción de su entrada, ya que entonces el coste que pueda suponer la especialización del programa $time_{Spec}([P, e])$ será ampliamente amortizado por las sucesivas ejecuciones del programa P_e, que en concreto será $speedup(P, e, d)$ veces más rápido que P. En [108], Neil D. Jones argumenta que la evaluación parcial puede ser ventajosa incluso para una única ejecución, ya que muchas veces sucede que

$$time_{Spec}([P, e]) + time_{P_e}(d) < time_P([e, d]).$$

Otro objetivo de la evaluación parcial es propiciar la *productividad* en el desarrollo de programas mediante el aumento de la *reusabilidad* y la *modularidad* del código. De todos es sabido que es más fácil establecer el significado declarativo (correcto) para una especificación sencilla de un problema; por contra, la ejecución de esta especificación como programa puede resultar ineficiente. Un evaluador parcial puede facilitar y hacer más ágil el desarrollo de los programas: disponiendo de una biblioteca de plantillas genéricas y simples (cuyo significado declarativo fuese, sin duda, el esperado) que posteriormente se especializarían de forma automática para producir un código más eficiente. La corrección del proceso de evaluación parcial asegura la corrección semántica del programa especializado. Disponer de una biblioteca de plantillas genéricas, para las tareas más comunes, también mejoraría la reusabilidad del código.

Cuando programamos, muchas veces nos encontramos con un conjunto de tareas similares para resolver y que corresponden a diferentes aspectos de un mismo problema. Una forma de afrontar esta situación es escribir un procedimiento específico y eficiente para cada una de estas tareas. Podemos enumerar dos desventajas en esta forma de proceder:

1. Debe de realizarse un sobrexceso de programación, lo que aumenta el coste de creación del programa y de su verificación.

2. El mantenimiento del programa se hace más dificultoso, ya que un cambio en las especificaciones puede requerir el cambio de cada uno de los procedimientos.

Con ser grave la primera de las deficiencias apuntadas, la segunda es la que puede producir mayores costes a largo plazo. Muchos estudios indican que el mayor coste en el *ciclo de vida* de un programa no es el coste inicial de diseño, codificación y verificación, sino el coste posterior asociado al mantenimiento del programa mientras está en producción y uso [170]. Una solución alternativa, que elimina las deficiencias comentadas anteriormente, consiste en escribir un procedimiento altamente parametrizado capaz de solucionar cada uno de los aspectos del problema. Nuevamente, podemos apuntar dos deficiencias:

- La dificultad propia de programar un procedimiento genérico que cubra todas las alternativas de forma eficiente.

- La ineficiencia inherente a este tipo de procedimientos, ya que un procedimiento altamente parametrizado gastará mucho de su tiempo de ejecución en la comprobación e interpretación de los parámetros y relativamente poco en los cómputos que debe realizar.

La evaluación parcial puede ayudarnos a vencer esta disyuntiva. Podemos escribir un procedimiento genérico altamente parametrizado (posiblemente ineficiente) y utilizar un evaluador parcial para especializarlo, suministrando los valores de los parámetros adecuados para cada una de las tareas especificas a resolver, obteniendo automáticamente un conjunto de procedimientos especificos y eficientes, tal y como deseábamos.

Uno de los objetivos más perseguidos en el campo de la evaluación parcial es lograr evaluadores parciales autoaplicables. La autoaplicación permite llevar a la práctica los resultados teóricos formulados en la Proposición 3.1.2, que propician la generación automática de programas. Una de sus aplicaciones más notables es la *generación de compiladores dirigida por la semántica* [196]; por ello entendemos lo siguiente: dada una especificación de un lenguaje de programación, basada en una semántica formal, transformarla automáticamente en un compilador. La motivación para la generación automática de compiladores es clara: el ahorro en esfuerzos de programación que supone la construcción de un compilador, que en ocasiones no es correcto con respecto a la semántica propuesta para el lenguaje que compila. La corrección de la evaluación parcial permite que la transformación automática de una especificación semántica de un lenguaje en un compilador haga desaparecer estos errores. Las tareas de diseñar la especificación de un lenguaje, escribir el compilador y mostrar la corrección del compilador, se reducen a una sola tarea: escribir la especificación del lenguaje en una forma adecuada para ser la entrada de un generador de compiladores.

Como puede apreciarse, la evaluación parcial y la autoaplicación tiene unas posibilidades muy prometedoras, pero todavía se necesita mucho trabajo para entender perfectamente su teoría y sus técnicas prácticas.

3.2 Perspectiva Histórica y Trabajos Relacionados.

En este apartado se da una visión general del desarrollo y estado actual de los trabajos sobre la evaluación parcial que se han realizado en diversas áreas relacionadas.

3.2.1 Programas Funcionales.

Puede decirse que la historia de la evaluación parcial comienza dentro del ámbito de la programación funcional, con los estudios de Lombardi y Raphael (1964) en los que se discute el empleo de Lisp para la computación con información incompleta (*incremental computation*). Es Lombardi quien emplea por vez primera el término "evaluación parcial".

En Suecia, a mediados de los setenta, Sandewall y su grupo desarrollaron *Redfun*, el primer gran evaluador parcial para un subconjunto representativo de *LISP*. Aunque consideraron la posibilidad de la autoaplicación para la contrucción de generadores de compiladores, no fueron capaces de llevarla a la práctica. Posteriormente hay un periodo de desinterés hasta que en 1984, en Dinamarca, Jones, Sestof y Søndergaard construyen el primer evaluador parcial autoaplicable para un lenguaje de ecuaciones recursivas de primer orden. Este evaluador parcial se llamó *mix*, siguiendo la terminología de Ershov que denominaba a la evaluación parcial *computación mixta* (*mixed computation*). Este trabajo lanzó un gran número de proyectos en Dinamarca (*Tradición de Copenhague*).

La evaluación parcial clásica de programas funcionales, tal y como se describe en [108], utiliza técnicas comparables a las de la evaluación parcial de programas imperativos: puntos de control, divisiones, anotaciones y un algoritmo de evaluación parcial *off-line* similar en muchos aspectos. En un contexto funcional se habla de una *división* para hacer referencia a una clasificación de los parámetros de una función a especializar en estáticos y dinámicos. Aquí, un *punto de control* hace referencia a un nombre de una función definida en un programa y un *punto de control especializado* es una definición de función especializada dotada de un nombre de función especializado (f, v_s), donde f es el nombre de la función original y v_s el valor de los parámetros estáticos de f. Un programa especializado es un conjunto de definiciones de función especializadas. Las divisiones suelen suministrarse a un evaluador parcial por medio de *anotaciones*, que son representadas mediante una sintaxis de doble nivel (*two-level syntax*), que posee una versión "estática" y otra "dinámica" para cada una de las construcciones del lenguaje fuente. El evaluador parcial trabaja con definiciones de función anotadas, manipulando un conjunto de funciones *pendientes* de especializar y un conjunto de funciones *marcadas* ya especializadas. El algoritmo, esencialmente, lo que hace es seleccionar y eliminar repetidamente del primer conjunto una función f para especializar con respecto a los valores v_s de sus parámetros estáticos. La especialización se realiza mediante un proceso de "reducción"[4] de las expresiones de los cuerpos de la función, en el que las partes representadas mediante el componente estático de la sintaxis de doble nivel producirán un valor; mientras que la reducción de las partes representadas por componentes dinámicos consistirá en la reducción de sus subexpresiones, para dar lugar a expresiones residuales. La reducción del cuerpo de la función f con respecto a v_s puede requerir de la introducción de nuevas funciones en el conjunto de funciones *pendientes* de especialización. El algoritmo de evaluación parcial termina cuando el conjunto de funciones *pendientes* es vacio, devolviendo el conjunto de funciones *marcadas* como especializadas.

En su tesis doctoral (1984), Wadler describe el *listless transformer* y poste-

[4]Ver el Apartado 5.4.2 de [108] para una descripción detallada de la función `reduce`, el componente esencial de un evaluador parcial para un subconjunto reducido del lenguaje funcional *Scheme*.

riormente el *algoritmo de deforestación* [203], un transformador de programas que, aunque no pueda considerarse estrictamente un evaluador parcial, está especialmente adaptado para la eliminación de estructuras intermedias y, por lo tanto, para la especialización de programas. La deforestación realiza propagación de constantes y no incorpora ningún mecanismo de generalización.

En 1988, Futamura y Nogi [74] introducen el método de *computación parcial generalizada* (GPC, del inglés *Generalized Partial Computation*), que después fue desarrollado con más detalle por Takano [194]. La computación parcial generalizada se aplica a un lenguaje funcional con semántica operacional perezosa y realiza la evaluación parcial apoyándose en un demostrador de teoremas, de forma que cuando encuentra una construcción "else" evalúa un predicado P y propaga P a la rama verdadera de la construcción y $\neg P$ a la rama falsa. Así pues, la computación parcial generalizada es capaz de propagar tanto información positiva como negativa. Sin embargo, no hay noción de información negativa en el lenguaje de [194] debido a la inexistencia de construcciones "else".

A finales de los ochenta, Charles Consel construyó un evaluador parcial autoaplicable llamado *Schism* para un subconjunto (de primer orden) del lenguaje Scheme [52]. Más tarde, Bondorf y Danvy construyeron *Similix*, un evaluador parcial autoaplicable también para un subconjunto (de primer orden) del lenguaje Scheme, que Bondorf extendio posteriormente a un subconjunto de orden superior de Scheme. Un trabajo reciente en esta direción es el de Thiemann [195].

La evaluación parcial también se ha aplicado a diversas clases de λ-cálculo [28, 89, 90, 108].

La evaluación parcial se ha aplicado también a los sistemas de reescritura de términos. Los primeros trabajos sobre evaluación parcial en este campo se centraron en lograr la capacidad de autoaplicación de los evaluadores parciales, antes que en determinar la correccción o la terminación de la transformación.

En Bondorf [43] se presentó un evaluador parcial para un lenguaje funcional intermedio, denominado *Tree*, en el que se representa el sistema de reescritura de términos que se desea especializar. El evaluador parcial se denomina *TreeMix* y es autoaplicable. Las características de *TreeMix* dependen notablemente de las características operacionales del lenguaje. *Tree* es un lenguaje de primer orden sin tipos y con semántica operacional impaciente (orden de reducción *innermost*) y, como reconoce Bondorf, adaptar el evaluador parcial a un modo de evaluación perezosa requeriría una completa revisión del mismo. Por otro lado, no se da ningún resultado de corrección sobre el método de evaluación parcial empleado y la terminación se deja al cuidado del usuario, que debe introducir anotaciones manuales para controlarla.

En Bonacina [42], la autora describe un evaluador parcial basado en el algoritmo de compleción de Knuth-Bendix (también llamado superposición). No se garantiza la terminación de la transformación. Un trabajo relacionado con el anterior es el presentado por Dershowitz y Reddy [63], que también formula una técnica basada en la compleción para la síntesis de programas funcionales, pero requiere de conocimientos heurísticos que aproximan esta técnica de transformación al tipo de las transformaciones de plegado/desplegado más que a la evaluación parcial.

Finalmente, el trabajo de Miniussi y Sherman [152] intenta mejorar la eficiencia de un lenguaje ecuacional en el que los programas se interpretan como TRS's. El sistema de reescritura de términos se utiliza para generar un automata que es implementado en un lenguaje imperativo específico (código EM), siendo este código el que se somete a una transformación mediante un algoritmo de evaluación parcial. El método consigue la eliminación de algunos pasos de reescritura intermedios y la fusión de estructuras de datos intermedias sin riesgo de no terminación.

Los trabajos sobre supercompilación [197, 199] están estrechamente relacionados con la evaluación parcial y son, dentro de la enorme literatura sobre transformación de programas funcionales, los más cercanos a nuestros métodos. La *supercompilación* (SC, del inglés *Supervised Compilation*) es una técnica de transformación que consta de tres componentes básicos: *driving*, *generalización* y *generación* de los programas residuales. El *driving* cubre las actividades de especialización y desplegado en la evaluación parcial. La técnica de *driving* puede entenderse como un mecanismo de transformación de funciones basado en la unificación, que usa un tipo de procedimiento de evaluación comparable al *narrowing* (perezoso) para construir los *árboles de transformación* (posiblemente con infinitos nodos), para un programa y un término dados, a partir de los cuales se extrae el término y el programa especializados. Los trabajos de Turchin describen el supercompilador para el lenguaje *Refal* (*Recursive Function Algorithmic Language*), un lenguaje funcional basado en emparejamiento con una noción poco estándar de patrones. Los primeros trabajos de Valentin Turchin son contemporáneos a las primeras definiciones del *narrowing* [68, 184] y prácticamente desconocidos para la comunidad científica occidental, no solamente porque fueran escritos en ruso, sino por la notación y conceptos matemáticos empleados en su desarrollo. En sus trabajos, Turchin utiliza la siguiente terminología [177]: "emparejamiento de patrones generalizado" para la unificación; "contracciones" para las substituciones computadas por el procedimiento de *narrowing*; y, a menudo, "*driving*" se usa para denominar el propio procedimiento de *narrowing*, es decir, la operación de instanciación de una llamada a función para todos los posibles valores de sus argumentos, seguido por el desplegado de las diferentes ramas. En años recientes se ha realizado un gran esfuerzo para encontrar una nueva formulación de la técnica de supercompilación en términos más familiares [80, 81, 82, 84, 85, 86, 88, 186, 188, 189, 190], estudiando su esencia, los efectos que puede conseguir y su relación con otros métodos de transformación. En [106], Neil Jones ha reelaborado la metodología de Turchin basada en el *driving* formulando sólidos fundamentos semánticos independientes de cualquier lenguaje o estructura de datos particular. En [188, 186] y [190] se presenta una versión simplificada del supercompilador de Turchin, denominada *supercompilador positivo* ya que sólo propaga la información positiva durante la transformación. El supercompilador positivo también puede verse como una variante del algoritmo de deforestación de Wadler, en el que se aumenta la cantidad de información que se propaga. La *supercompilación positiva* (PS, del inglés *Positive Supercompilation*) trata con un lenguaje funcional de primer orden con una semántica operacional perezosa (*call-by-name*) [190]. En este lenguaje, los argumentos de las funciones son parámetros de entrada y sólo se realiza ajuste de patrones para patrones constuctores planos y lineales. Esta formulación se puede generalizar a lenguajes menos restrictivos al coste de tener que utilizar algoritmos de *driving* más complejos[5]. En la supercompilación positiva, no hay distinción explicita entre los niveles de control local y global, tan útiles para el control de la terminación del proceso de especialización, ya que los términos se despliegan un único paso. Por contra, se construye una amplia estructura de evaluación que comprende algo similar a los árboles de búsqueda local y a las configuraciones globales de [145]. Cada llamada t en el grafo del proceso de *driving* produce una función residual. El cuerpo de la definición de la nueva función se deriva a partir de los descendientes de t en el grafo.

Gracias al mecanismo de *driving*, el supercompilador es capaz de conseguir el mismo nivel de propagación de información (basada en la unificación) y de especialización que la evaluación parcial de los programas lógicos. Ambas técnicas son semejantes, permitiendo la eliminación de estructuras intermedias y un grado similar de especialización de los programas. Además, el supercompilador es capaz de propagar tanto la información positiva como la negativa, lo que constituye la principal diferencia con el supercompilador positivo. Tradicionalmente, la evaluación parcial clásica de programas funcionales únicamente realiza propagación de constantes [108]. Asímismo, la supercompilación es capaz de soportar ciertas formas de demostración de teoremas, síntesis e inversión de programas. La supercompilación puede también mejorar un programa incluso si todos los parámetros actuales en las llamadas a función son variables, ya que puede eliminar algunas redundancias provocadas por bucles anidados, variables repetidas, etc.

El proceso de *driving* no siempre termina y tampoco preserva la semántica del programa, dado que puede extender el dominio de las funciones [186, 189]. Algunas técnicas para asegurar la terminación del mecanismo de *driving* se estudian en [188] y en [200]. La idea en [200] consiste en supervisar la construcción del árbol y, en ciertos lugares, plegar un estado a un estado previo, construyendo así un grafo finito. La operación que, en general, posibilita plegar los estados se conoce como *generalización*. En [188], la terminación se garantiza siguiendo una aproximación similar al marco general de Martens y Gallagher para asegurar la terminación de la evaluación parcial de programas lógicos [144]. Por último, Sørensen, inspirado en ideas que aparecen en [165], ha construido un marco (que se pretende) general para el control de la terminación de un proceso de transformación [187]. En particular, Sørensen presenta una instancia del marco que asegura la terminación del proceso de supercompilación positiva.

[5]En el contexto de los lenguajes logico–funcionales, esto no sucede, ya que se dispone de un mecanismo de emparejamento más poderoso via la unificación de todos los argumentos, tanto durante la ejecución como durante la especialización.

3.2.2 Programas Lógicos.

Komorowski introdujo la evaluación parcial en el campo de la Programación Lógica [115]. Después de varios años de olvido, la evaluación parcial ha atraído un interés considerable en este campo [35, 50, 77, 75, 83, 136, 137, 145], donde se le conoce generalmente como *deducción parcial* (PD, del inglés *Partial Deduction*)

En [36] se presenta un algoritmo para la evaluación parcial de programas lógicos, mientras que en [137] se presentan los fundamentos de la deducción parcial en un marco formal. Dentro del marco teórico establecido por Lloyd y Shepherdson en [137] para la deducción parcial, la evaluación parcial de programas lógicos puede describirse como sigue. Dado un programa P y un conjunto de átomos S, el objetivo de la deducción parcial es obtener un nuevo programa P' que compute las mismas respuestas que P para cualquier *objetivo* (*input goal*) que sea una instancia de un átomo perteneciente a un conjunto S de átomos evaluados parcialmente. El programa P' se obtiene agrupando en un conjunto las *resultantes*, que se obtienen de la siguiente manera: para cada átomo A de S, primero construir un árbol-SLD finito (y posiblemente incompleto), $\tau(A)$, para $P \cup \{\leftarrow A\}$; entonces considerar las hojas de las ramas de $\tau(A)$ que no son de fallo, digamos $G_1, \ldots, G_r$, y las substituciones computadas a lo largo de estas ramas, digamos $\theta_1, \ldots, \theta_r$; y, finalmente, construir el conjunto de cláusulas: $\{\theta_1(A) \leftarrow G_1, \ldots, \theta_r(A) \leftarrow G_r\}$; que son las resultantes que constituyen el programa residual P'. La restricción a especializar únicamente átomos $A \in S$ (y no conjunciones de átomos, por ejemplo) está motivada por la necesidad de que las resultantes obtenidas sean cláusulas de Horn.

Para programas y objetivos definidos, Lloyd y Shepherdson han demostrado que la deducción parcial es siempre correcta, pero no necesariamente completa. La completitud se restablece exigiendo que $P' \cup \{G\}$ sea *S-cerrado*, i.e., cada átomo A' en $P' \cup \{G\}$ es una instancia de un átomo B en S (también se dice que el átomo A' está *cubierto*[6] por el átomo B). El requisito anterior recibe el nombre de *condición de cierre*. Por otra parte, el resultado de corrección es *débil*, en el sentido de que si para un objetivo y el programa especializado se computa una respuesta, entonces el programa original computará una más general. Para garantizar que el programa residual P' no produce respuestas adicionales se necesita una condición adicional de *independencia*, que se cumple cuando dos átomos en S no tienen una instancia en común. De esta forma se consigue la corrección en una forma *fuerte* que supone la igualdad de las respuestas obtenidas por el progragrama original P' y el transformado P. La condición de independencia se logra mediante técnicas de renombramiento. Para programas y objetivos normales los resultados son menos satisfactorios. Muestran que la evaluación parcial es, en general, no solamente incompleta, sino, también incorrecta. Sólamente cuando se añade la condición de independencia a la condición de cierre se restablecen las propiedades de corrección y completitud fuertes de la evaluación parcial.

[6] Este concepto se translada fácilmente a los términos de $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Dados dos términos s y t, diremos que s está *cubierto* por t si s es una instancia de t.

Como se ha comentado, los problemas de la terminación del proceso de la evaluación parcial son de primordial importancia y han sido tratados en [47, 143, 144, 145]. Martens y Gallagher abordan el problema de la terminación estructurando el procedimiento en dos niveles: local y global. En el *nivel local*, una *regla de desplegado* (*unfolding rule*) controla la generación de los árboles de resolución de los que se extraen las resultantes, de forma que éstos se mantengan finitos, gracias a la aplicación de un test de parada apropiado (usando habitualmente un orden bien fundado). En el *nivel global* se controla la aplicación recursiva de la regla de desplegado mediante la selección de los átomos que se van a considerar en el paso siguiente. El nivel global debe garantizar una especialización suficiente (mediante la adición de nuevos átomos), a la vez que se asegura la condición de cierre y la terminación del proceso.

Gracias al mecanismo de unificación, la deducción parcial es capaz de propagar información sintáctica sobre los datos de entrada, tal como la estructura de los términos, y no sólo valores constantes, haciendo así la evaluación parcial de los programas lógicos más potente y simple que la evaluación parcial de los programas funcionales.

La relación entre la deducción parcial y la transformación de programas basada en técnicas de plegado/desplegado también ha sido objeto de estudio en años recientes [46, 128, 164, 171, 183]. Como se indica en [164], la evaluación parcial es esencialmente un subconjunto de las transformaciones de plegado y desplegado en la que se hace uso casi exclusivo de la regla de desplegado como herramienta de transformación básica (solamente se obtiene una forma limitada de plegado, al exigir que los programas transformados cumplan la condición de cierre). Por lo tanto, las estrategias basadas en las técnicas de plegado/desplegado consiguen una serie de optimizaciones de los programas (*tupling*, *deforestación* y eliminación de variables innecesarias) que la deducción parcial no puede obtener. Como contrapartida, la deducción parcial es menos compleja computacionalmente y se puede automatizar más fácilmente. Se han realizado esfuerzos por aunar lo mejor de ambas técnicas de transformación. Por ejemplo, [165] presenta un algoritmo de evaluación parcial expresado en términos de las técnicas de transformación de plegado/desplegado. Siguiendo una orientación distinta, [83, 128] consideran la extensión del marco clásico de la deducción parcial [137] para que se puedan especializar conjunciones de átomos, lo que permite conseguir algunos de los beneficios adicionales que obtienen las técnicas de plegado y desplegado. Esta nueva técnica ha recibido el nombre de *deducción parcial conjuntiva* (CPD, del inglés *Conjuntive Partial Deduction*).

3.2.3 Programas Lógico–Funcionales.

No existen en la literatura muchos antecedentes sobre trabajos relacionados directamente con la especialización de programas lógico–funcionales. Hemos encontrado, sin embargo, dos excepciones notables. En [133], Levi y Sirovich definen un procedimiento de evaluación parcial para el lenguaje lógico–funcional TEL, que utiliza un mecanismo de ejecución simbólica, basado en la unificación, que puede entenderse como una forma de *narrowing perezoso*. En [56], Darlington y Pull muestran cómo la unificación permite integrar los pasos de desplegado e instanciación (introducidos en el marco de transformación de programas definido por Burstall y Darlington en [48]), obteniendo así la habilidad del *narrowing* para tratar con variables lógicas. También se esboza un evaluador parcial para el lenguaje funcional HOPE (extendido con unificación). Sin embargo, en ninguno de estos trabajos se han abordado cuestiones de control, terminación o equivalencia semántica. Por consiguiente podemos afirmar que la especialización de programas lógico–funcionales es un área de investigación relativamente nueva.

En [12, 14] y [15] se presentan las bases para el estudio formal de la evaluación parcial de programas para lenguajes lógico–funcionales de primer orden sin tipos. Los autores, siguiendo el estilo de Lloyd y Shepherdson [137], definen los conceptos de *resultante* y *programa evaluado parcialmente*, y las correspondientes condiciones de *cierre* e *independencia* que son necesarias para asegurar la corrección y completitud del método. Todos estos conceptos se extienden convenientemente para tratar algunas características, como las funciones anidadas y la noción de modo de evaluación, que están ausentes en la programación lógica. El resultado es un *marco* general para la evaluación parcial que permite la especialización de programas con respecto a un conjunto S de *llamadas* (términos). El algoritmo de evaluación parcial comienza con el conjunto de llamadas que aparecen en el objetivo inicial, prosigue evaluando parcialmente cada uno de esos términos, usando una regla de desplegado finita (i.e. que asegura la denominada terminación local del proceso). Dicha regla introduce, dinámicamente, nuevos términos en el conjunto S que serán especializados posteriormente. Para asegurar que la construcción del conjunto S se realiza en un tiempo finito (logrando la denominada terminación global), se introduce un operador de *abstracción* que, además, garantiza la *polivarianza* de la especialización. Para controlar la terminación del proceso de evaluación parcial, tanto en su aspecto local como global, se hace uso de un orden de subsumción (*embedding*)[7] y algunas técnicas inspiradas en [145, 188]. Debido a que el proceso de desplegado de los árboles locales se realiza mediante *narrowing*, decimos que el método de evaluación parcial está guiado por *narrowing* (por lo que lo distinguimos mediante el acrónimo NPE, del inglés *Narrowing-driven Partial Evaluation*), siendo una característica distintiva respecto de otras propuestas.

El *marco* de evaluación parcial obtenido es paramétrico con respecto a la estrategia de *narrowing* utilizada para desplegar los árboles así como del operador de *abstracción*, y asegura tanto la terminación del proceso de evaluación parcial como la condición de cierre respecto al conjunto S para el que se especializa el programa. Otro resultado general es que el programa especializado cumple la propiedad de *corrección débil* con respecto al programa original. Sin embargo, no puede probarse la corrección y completitud fuerte del programa transformado independientemente de la estrategia de *narrowing*. Estos teoremas son, por naturaleza, dependientes de la estrategia particular que se considere y requieren pruebas específicas. De hecho, diferentes estrategias de *narrowing* tienen dife-

[7] Ver más adelante la Definición 3.4.1.

rentes propiedades semánticas [94], por lo que son necesarias técnicas de prueba diferentes en cada caso concreto. Un problema adicional es que si no se quiere perder la completitud de la estrategia de *narrowing*, los programas transformados deberán cumplir ciertas restricciones adicionales. Así, en ocasiones se requiere una fase final de renombramiento [8] en la que se asignan nuevos nombres a los símbolos de función, con el objeto de que el programa transformado pueda ejecutarse con el mismo significado. Esta fase no es necesaria en el caso de *narrowing condicional* sin restricciones.

En [14] se demuestra la corrección y completitud fuerte del método NPE basado en *narrowing* condicional sin estrategia, para un programa que cumple las condiciones necesarias para la completitud del *narrowing* condicional básico. También en [14] se explora una instancia del método NPE para *narrowing* condicional con estrategia *innermost* y normalización.

Después de este trabajo de fundamentación de la evaluación parcial de programas lógico-funcionales, las investigaciones han cubierto tanto aspectos teóricos como prácticos:

- En [8] se estudia una nueva instancia del método NPE para *narrowing* condicional con estrategia *perezosa*. En este caso, el desplegado de los términos se detiene al alcanzar una forma constructora en cabeza y es necesario un postproceso adecuado de renombramiento del programa transformado para recuperar la disciplina de constructores y la linealidad por la izquierda del programa original; lo que posibilita su posterior ejecución.

- En [3, 2] se desarrollan técnicas avanzadas de especialización, definiendo una regla de desplegado dinámica y un nuevo operador de abstracción que permiten obtener resultados análogos a los de la *deducción parcial conjuntiva* [83, 128].

- En [17, 16] se estudia una nueva instancia del método NPE para *narrowing* con estrategia *necesaria* y programas *inductivamente secuenciales*. Se demuestra que el método hereda las buenas propiedades del mecanismo de base. En particular, la secuencialidad inductiva de los programas se conserva durante el proceso de especialización.

- En [9] se estudian las condiciones bajo las cuales la evaluación parcial de programas lógico-funcionales basada en *narrowing* con estrategia necesaria y con estrategia perezosa coinciden. Se postula la clase de los programas uniformes [120, 121] como la clase de programas más amplia para la cual ambas estrategias son equivalentes.

- En [4] se presenta un prototipo de evaluador parcial para lenguajes lógico-funcionales, denominado Indy (Integrated Narrowing-Driven specialization sYstem), basado en las ideas desarrolladas en los trabajos anteriores. Indy permite elegir diversas opciones mediante las que se selecciona: la estrategia de *narrowing*, la posibilidad de normalización entre pasos de *narrowing*; la regla de desplegado que controla la construcción de los árboles locales y el operador de abstracción que permite controlar la terminación global. El poder de una técnica de transformación puede medirse comprobando si es capaz de obtener la optimización que es básica en el algoritmo de Knuth-Morris-Pratt (KMP) [114] para la búsqueda de patrones en cadenas de caracteres. Los experimentos con INDY demuestran que el método NPE es capaz de pasar el llamado "*test KMP*" [86, 106]. También puede constatarse que el método NPE consigue incrementos promedio del 100% en la ejecución de los programas especializados con respecto a los programas originales.

- Finalmente, en [5] se estudia la viabilidad de aplicar y extender las técnicas de la NPE a lenguajes lógico-funcionales que, como Curry, combinan los principios operacionales del *narrowing* y la residuación. En estos lenguajes cobra especial importancia el tratamiento de los *retrasos* (*delays*) de las llamadas a función que no están suficientemente instanciadas.

Lafave y Gallagher [123] han presentado un algoritmo de evaluación parcial para el lenguaje lógico–funcional Escher. Como ya se ha comentado, contrariamente a la mayoria de las propuestas existentes, Escher emplea una técnica basada en reescritura como mecanismo operacional para simplificar los términos hasta alcanzar una forma normal, que se considera la salida de la computación, por este motivo denominamos al método de Lafave y Gallagher: evaluación parcial de programas lógico–funcionales basados en reescritura (PER, del inglés *Partial Evaluation of functional–logic Rewrite–based languages*). El algoritmo de evaluación parcial propuesto por estos autores es genérico con respecto a la regla de selección de Escher, el operador de desplegado (que controla la terminación local del proceso de evaluación parcial, y que hace uso de un orden de subsumción y de técnicas semejantes a las desarrolladas en [8, 83, 129, 188]) y el operador de abstracción (que controla la terminación global y la polivarianza de la especialización mediante técnicas de plegado y generalización, desarrolladas previamente en [145, 188]). La estructura del algoritmo está basada en el algoritmo básico de Gallagher para la especialización de programas lógicos [145]. El algoritmo utiliza una técnica de propagación de la información basada en la unificación (semejante al *narrowing* normalizante) que permite obtener una mejor especialización de la que sería posible mediante el uso del mecanismo operacional propio del lenguaje Escher. Sin embargo, los autores reconocen que su método no tiene la potencia especializadora del supercompilador positivo (ya que no es capaz de pasar el test KMP). Posteriormente, estos mismos autores, en [122], han aumentado su evaluador parcial con la posiblilidad de propagar información negativa basada en restricciones. La idea básica de esta extensión del método ha consistido en la especialización de términos junto con un conjunto de restricciones que debe cumplir dicho término. Las restricciones son instanciadas con las respuestas parciales obtenidas en los cómputos realizados en tiempo de evaluación parcial o bien aumentadas con nuevas restricciones (tanto positivas como negativas) provenientes de las condiciones, si el término que se especializa es una expresión condicional. Estas mejoras hacen que su método pueda superar el test KMP.

3.3 Evaluación Parcial de Programas Lógico Funcionales.

Como acabamos de mostrar, la evaluación parcial ha sido aplicada a distintos paradigmas de programación (incluido el imperativo) [108]. También se ha visto que muchas de sus características y de sus problemas pueden abordarse desde un marco teórico muy general. Por otro lado, las diferentes áreas de la evaluación parcial, si bien comparten muchas técnicas básicas, también es cierto que difieren en los aspectos concretos de dichas técnicas. En esta sección describimos un procedimiento genérico para la evaluación parcial de programas lógico–funcionales. Las definiciones de este apartado son sencillas adaptaciones de las que aparecen en [14] y se aplican a TRS's canónicos.

El procedimiento genérico para la evaluación parcial de programas lógico–funcionales que vamos a presentar sigue, principalmente, el marco teórico establecido por Lloyd y Shepherdson en [137] y por Martens y Gallagher en [145] para la deducción parcial de programas lógicos. Sin embargo, un buen número de conceptos y resultados deben generalizarse para poder tratar con funciones y llamadas a función anidadas. Debido a que los programas lógico–funcionales engloban a los programas lógicos y a los funcionales, algunas de las nociones toman una forma más elaborada en comparación con las que aparecen en la evaluación parcial de los lenguajes lógicos y funcionales puros. Por contra, lo que se obtiene es un marco general más limpio que aúna muchos de los desarrollos efectuados en ambos campos.

Comenzamos formalizando las nociones básicas de la evaluación parcial de los programas lógico–funcionales, utilizando el *narrowing* como principio operacional.

3.3.1 Conceptos y Resultados Básicos.

Existen varias alternativas para extender la noción de resultante utilizada en la deducción parcial, e introducida en el Apartado 3.2.2, al campo de la programación lógico–funcional [202]. Dado que el equivalente natural de los átomos en este contexto son las ecuaciones, podemos considerar un *objetivo ecuacional* $g \equiv s \approx t$ y una derivación de *narrowing* $g \overset{\sigma}{\leadsto}_{\varphi}^{*} g'$, donde $g' \equiv s' \approx t'$. Esto se correspondería con la restricción, en el campo de la deducción parcial, de especializar únicamente átomos (individuales). Entonces, siguiendo un camino totalmente paralelo al de la deducción parcial, podríamos definir el concepto de resultante mediante la regla: $\sigma(g) \to (g' \Rightarrow true)$. Esta forma de definir el concepto de resultante no es útil por ciertas razones. Principalmente, porque no se está especializando ninguna función definida por el usuario en el programa, sino el símbolo de la igualdad de la ecuación g. Una alternativa más natural en nuestro contexto es la especialización de términos cualesquiera y no necesariamente de ecuaciones. Esto conduce a la siguiente definición de resultante.

Definición 3.3.1 (Resultante)
Sea s un término y $\mathcal{R}$ un programa. Dada una derivación de narrowing *que*

respeta la estrategia φ, $s \overset{\sigma}{\leadsto}_\varphi^* t$, su resultante *asociada es la regla de reescritura $\sigma(s) \to t$.*

Nótese, sin embargo, que esta definición puede agravar el problema mencionado anteriormente, debido a que ahora podrán aparecer tanto ecuaciones como términos conjuntivos puesto que tratamos la igualdad y la conjunción como operadores booleanos. Además, en este caso pueden darse muy pocas oportunidades de encontrar *regularidades* (i.e., de alcanzar la condición de cierre [165]), de modo que se alcanzará una especialización muy pobre. Tal y como se comentó en el Apartado 3.1.1, esto se debe al mecanismo de generalización (abstracción), que comúnmente se aplica para garantizar la terminación del proceso de evaluación parcial, que fuerza a una generalización excesiva de las expresiones y a veces puede destruir toda posibilidad de especialización (dentro del marco de la NPE, describimos con detalle como se produce el proceso de abstracción en el Apartado 3.4.2). Estos problemas pueden evitarse introduciendo técnicas especiales que sean capaces de reordenar y partir adecuadamente expresiones complejas, que contengan ecuaciones y conjunciones, en "piezas" más simples antes de proceder a su especialización. Algunas de estas técnicas han sido propuestas en [83, 128] dentro del marco de la *deducción parcial conjuntiva* . Volveremos sobre estos problemas en el Capítulo 8, donde investigamos técnicas más potentes que hacen posible la especialización de expresiones complejas dentro del marco de la NPE. Por el momento parece adecuado restringirnos a la especialización de *términos simples* i.e. términos del conjunto $\mathcal{T}(\mathcal{C} \cup \mathcal{D}, \mathcal{X})$. Para no complicar el discurso, durante éste y el próximo capítulo, reservaremos la denominación de "*término*" para referirnos a los términos simples, en contraposición con la denominación "*expresión compleja*" que emplearemos para referirnos a los términos del conjunto $\mathcal{T}(\mathcal{F}, \mathcal{X}) \setminus \mathcal{T}(\mathcal{C} \cup \mathcal{D}, \mathcal{X})$, que son términos que contienen símbolos primitivos.

La intuición que se esconde detrás del concepto de resultante queda patente en el siguiente ejemplo.

Ejemplo 10 *Sea el programa*

$$
\begin{array}{ll}
R_1: & f(X) \;\to\; (g(X) \approx 0 \;\Rightarrow\; X) \\
R_2: & g(0) \;\to\; 0 \\
R_3: & 0 \approx 0 \;\to\; true \\
R_4: & (true \;\Rightarrow\; X) \;\to\; X
\end{array}
$$

Dada la derivación de narrowing *(perezoso):*

$$
\begin{array}{l}
\underline{f(x)} \overset{[\Lambda, R_1, id]}{\leadsto_\varphi} (\underline{g(x)} \approx 0 \;\Rightarrow\; x) \\
\overset{[1.1, R_2, \{X/0\}]}{\leadsto_\varphi} (\underline{0 \approx 0} \;\Rightarrow\; 0) \\
\overset{[1, R_3, id]}{\leadsto_\varphi} (\underline{true \;\Rightarrow\; 0}) \\
\overset{[\Lambda, R_4, id]}{\leadsto_\varphi} 0
\end{array}
$$

Intuitivamente, el término $f(X)$ se ha reducido a 0 al aplicar la substitución $\{X/0\}$, de modo que la resultante asociada a esta derivación es:

$$\{X/0\}(f(X)) \;\to\; 0 \;\; (i.e.,\; f(0) \;\to\; 0).$$

Podemos interpretar que la resultante comprime una serie de pasos de narrowing *convirtiéndolos en una regla de programa, de forma que cuando se emplea para evaluar el término $f(X)$ lo evalúa a 0 en un único paso de* narrowing.

Una vez comprendido el concepto de resultante es fácil definir qué entendemos por evaluación parcial de un programa con respecto a un término s. La idea básica consiste en la construcción de un árbol de búsqueda incompleto para el término inicial s, del que extraemos las resultantes asociadas con las derivaciones que parten de la raíz s y terminan en las hojas que no son de fallo.

Definición 3.3.2 (Evaluación Parcial)
Sea $\mathcal{R}$ un programa y s un término. Sea τ un árbol de narrowing *(posiblemente incompleto) para s en $\mathcal{R}$. Sean $\{t_1, \ldots, t_n\}$ los términos en las hojas de τ. Entonces, el conjunto de resultantes asociadas a las derivaciones de* narrowing $\{s \overset{\sigma_i}{\leadsto}_{\varphi}^{+} t_i \mid i = 1, \ldots, n\}$ *se denomina una* evaluación parcial *de s en $\mathcal{R}$.*

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

La evaluación parcial de un conjunto de términos se considera módulo renombramientos. También, en ocasiones, la evaluación parcial de un conjunto de términos S en $\mathcal{R}$ se denominará evaluación parcial de $\mathcal{R}$ con respecto a S.

Nuestra restricción de especializar términos no impide la especialización de expresiones complejas, si convenimos en especializar sus términos constituyentes por separado. Por ejemplo, si $g \equiv (s_1 \approx t_1) \wedge \ldots \wedge (s_n \approx t_n)$; donde los s_i y t_i, con $i = 1, \ldots, n$, son términos, entonces la evaluación parcial del objetivo ecuacional g es la evaluación parcial del conjunto $\{s_1, \ldots, s_n, t_1, \ldots, t_n\}$ en $\mathcal{R}$.

Los términos simples que constituyen una expresión compleja g, a especializar se denominan en muchas ocasiones "*llamadas*". Denotamos el conjunto de las llamadas de g como g_{calls}. De igual modo, $\mathcal{R}_{calls}$ denota el conjunto de llamadas en las rhs's de las reglas de $\mathcal{R}$. Por ejemplo, si $\mathcal{R}$ contiene una regla $l \to ((s_1 \approx t_1) \wedge \ldots \wedge (s_n \approx t_n) \Rightarrow r)$, el conjunto $\mathcal{R}_{calls}$ contiene las llamadas $s_1, \ldots, s_n, t_1, \ldots, t_n, r$.

Corrección de la Evaluación Parcial Dirigida Por Narrowing.

Como fue establecido por Lloyd y Shepherdson en [137], se necesita una condición de *cierre* (*closedness*) para asegurar la completitud del proceso de evaluación parcial. Nosotros extendemos la noción "plana" de cierre de la deducción parcial introduciendo una condición de cierre recursiva que garantiza que todas las llamadas que puedan ocurrir durante la ejecución del *programa residual* están cubiertas por alguna regla de éste.

Definición 3.3.3 (Cierre)
Sea S un conjunto de términos y t un término. Decimos que el término t es S-cerrado si se cumple closed(S, t), donde el predicado closed se define inductivamente como sigue:

$$closed(S,t) \Leftrightarrow \begin{cases} true & \texttt{si } t \in \mathcal{X} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \texttt{si } t \equiv c(t_1,\ldots,t_n) \\ (\exists s \in S)\ \theta(s) = t \wedge \bigwedge\limits_{t' \in \mathcal{R}an(\theta)} closed(S,t') & \texttt{si } t \equiv f(t_1,\ldots,t_n) \end{cases}$$

donde[8] $c \in \mathcal{C}$, $f \in \mathcal{D}$. *Decimos que un conjunto de términos* T *es* S*-cerrado, en símbolos* $closed(S,T)$*, si se cumple* $closed(S,t)$ *para todo* $t \in T$*, y decimos que un programa* $\mathcal{R}$ *es* S*-cerrado si se cumple* $closed(S, \mathcal{R}_{calls})$*.*

Informalmente, un término t se considera cerrado con respecto a un conjunto S, si t puede descomponerse ("aplanarse") en un conjunto de subtérminos, cada uno de los cuales es una instancia de un término de S. Así pues, contrariamente a la condición de cierre de la deducción parcial, estamos imponiendo no solamente que t quede cubierto por algún término de S, sino que también los términos del rango de θ estén cubiertos recursivamente por S. El siguiente ejemplo ilustra la anterior definición.

Ejemplo 11 *Sea* t *el término* $f(g(0))$ *y* S *el conjunto* $\{f(X), g(X)\}$*. Entonces,* t *es* S*-cerrado ya que:*

1. $(\exists\sigma)\sigma = \{X/g(0)\}$. $t = \sigma(f(X))$, *i.e. el propio término* t *está cubierto por* $f(X)$*.*
2. $(\exists\theta)\theta = \{X/0\}$. $g(0) = \theta(g(X))$, *i.e.* $g(0)$ *está cubierto por* $g(X)$*.*
3. *y* 0 *es constructor que, por definición, está cubierto.*

Esencialmente, la condición de cierre nos indica que el significado de una expresión como $f(g(0))$ puede derivarse de sus constituyentes "planos": $f(x)$, $g(y)$ y 0.

Ahora introducimos la condición de *independencia*. La condición de independencia garantiza que dos términos especializados diferentes no pueden ser confundidos y que el programa residual $\mathcal{R}'$ no produce respuestas adicionales. En el caso de la deducción parcial, para asegurar la condición de independencia y garantizar la corrección fuerte de la transformación, solamente se comprueba la no unificabilidad (dos a dos) de los átomos evaluados parcialmente de S. En nuestro caso, necesitamos una condición más compleja, más concretamente, necesitamos considerar los posibles *solapamientos* (*overlaps*) entre las llamadas especializadas [14].

Definición 3.3.4 (Solapamiento)
Un término s solapa *con un término* t *si existe un subtérmino no variable* $s_{|u}$ *de* s *tal que* $s_{|u}$ *y* t *unifican. Si* $s \equiv t$*, requerimos que* t *sea unificable con un subtérmino propio no variable de* s*.*

[8] El potencial del método de NPE puede extenderse con la posibilidad de manipular expresiones complejas de manera inmediata: simplemente permitiendo que $f \in (\mathcal{D} \cup \mathcal{P})$. Si bien, esto conduce a las dificultades ya comentadas. En el Capítulo 8 se presenta un tratamiento adecuado para la especialización de expresiones que contienen símbolos primitivos dentro del marco de la NPE.

Definición 3.3.5 (Independencia)
Un conjunto de términos S es independiente *si no existen términos s y t en S tal que s solapa con t.*

En [14] se ha estudiado el problema de la corrección y completitud de la NPE cuando se consideran TRS's condicionales como programas y se emplea el *narrowing* sin restricciones como mecanismo operacional. En lo que sigue resumimos los resultados que se aplican al caso de los TRS's (incondicionales) con los que trabajamos. Comenzamos con los resultados relativos a la corrección y completitud débiles.

Teorema 3.3.6 *[14] [Corrección Débil] Sea $\mathcal{R}$ un TRS confluente, S un conjunto finito de términos, y $\mathcal{R}'$ la evaluación parcial de $\mathcal{R}$ con respecto a S. Sea g una expresión (posiblemente compleja). Entonces, si el* narrowing *sin restricciones computa un resultado d en $\mathcal{R}'$ con respuesta computada normalizada θ', también computa el resultado d en $\mathcal{R}$ con una respuesta computada normalizada θ tal que $\theta \leq \theta'$ $[Var(g)]$.*

Teorema 3.3.7 *[14] [Completitud Débil] Sea $\mathcal{R}$ un TRS canónico, g una expresión (posiblemente compleja), S un conjunto de términos y $\mathcal{R}'$ la evaluación parcial de $\mathcal{R}$ con respecto a S tal que ${\mathcal{R}'}_{calls} \cup g_{calls}$ es S-cerrada. Entonces, si el* narrowing *sin restricciones computa un resultado d en $\mathcal{R}$ con respuesta computada normalizada θ, también computa el resultado d en $\mathcal{R}'$ con una respuesta computada normalizada θ' tal que $\theta' \leq \theta$ $[Var(g)]$.*

Nótese que, aunque la semántica a preservar sea la de las respuestas normalizadas, para obtener la completitud no basta con exigir simplemente la confluencia. Esto es debido a que existe una conexión entre la completitud de la NPE y la composicionalidad de la semántica del *narrowing* [14]. Desgraciadamente, el *narrowing* sin restricciones no es composicional en general [11, 202]; por este motivo, para la completitud de la NPE se requieren las condiciones en las que el *narrowing* básico [103], una estrategia de *narrowing* que sí es composicional, es completo, i.e. confluencia y terminación[9]. Para asegurar que el programa original y el especializado computan las mismas respuestas hacen falta restricciones adicionales a la de cierre, como son la independencia (en el caso de la corrección fuerte) y la linealidad (en el caso de la completitud fuerte) del conjunto de términos evaluados parcialmente.

Teorema 3.3.8 *[14] [Corrección Fuerte] Sea $\mathcal{R}$ un TRS confluente, g una expresión (posiblemente compleja), S un conjunto finito e independiente de términos, y $\mathcal{R}'$ la evaluación parcial de $\mathcal{R}$ con respecto a S tal que ${\mathcal{R}'}_{calls} \cup g_{calls}$ es S-cerrada. Entonces, si el* narrowing *sin restricciones computa un resultado d en $\mathcal{R}'$ con respuesta computada normalizada θ', también computa el resultado d en $\mathcal{R}$ con respuesta computada normalizada θ tal que $\theta = \theta'$ $[Var(g)]$.*

[9] Cuando se trata con TRS's condicionales, debe exigirse que el sistema sea decreciente (*decreasing*) – ver [149] para una definición formal. La condición de decrecimiento es la extensión natural de la condición de terminación para TRS's condicionales que asegura la finitud de los cómputos recursivos.

Teorema 3.3.9 *[14] [Completitud Fuerte] Sea $\mathcal{R}$ un TRS canónico, g una expresión (posiblemente compleja), S un conjunto finito de términos lineales y $\mathcal{R}'$ la evaluación parcial de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}'_{calls} \cup g_{calls}$ es S-cerrada. Entonces, si el* narrowing *sin restricciones computa un resultado d en $\mathcal{R}$ con respuesta computada normalizada θ, también computa el resultado d en $\mathcal{R}'$ con una respuesta computada normalizada θ' tal que $\theta' = \theta$ $[Var(g)]$.*

De las pruebas realizadas en [14] se desprende que la completitud fuerte solamente se cumple para las respuestas computadas por derivaciones de *narrowing* básico, incluso bajo el supuesto de que se cumpla la condición de cierre y la linealidad de las llamadas especializadas.

Los teoremas anteriores no responden a la cuestión de cómo se puede computar el conjunto de términos S, para que se cumplan las condiciones de cierre e independencia requeridas, o cómo debe desarrollarse el proceso de evaluación parcial. En los próximos apartados se formaliza un procedimiento de evaluación parcial dirigido por *narrowing* del que se garantiza su terminación y que alcanza la condición de cierre. En el próximo capítulo estudiaremos un postproceso de renombramiento que permite, además, obtener la condición de independencia del conjunto de llamadas especializadas S.

3.3.2 Un Algoritmo Genérico para la Evaluación Parcial Dirigida por Narrowing.

En la literatura sobre la especialización de programas, el control del proceso de evaluación parcial se ha abordado desde dos aproximaciones diferentes. Los evaluadores parciales *on-line* toman todas las decisiones de control durante el mismo proceso de evaluación parcial mientras que los especializadores *off-line* realizan una fase previa de análisis que genera un programa con *anotaciones*, que es el que finalmente se utiliza en la fase de especialización [108]. Los métodos *off-line* obtienen mejores tiempos para el proceso de evaluación parcial (i.e., son más rápidos que los métodos *on-line*) y están mejor adaptados para obtener la terminación [53]. Para conseguir evaluadores parciales autoaplicables, la evaluación parcial clásica de programas funcionales se ha decantado principalmente por las técnicas *off-line*, mientras que la *supercompilación positiva* [188, 190] y la *deducción parcial* se han centrado en las técnicas *on-line*, que habitualmente conducen a programas especializados más eficientes. En esta tesis estamos interesados en los aspectos de la precisión y del control de la terminación de la evaluación parcial, y no en la autoaplicación, por lo que adoptaremos la aproximación *on-line*.

Como se ha avanzado en el Apartado 3.1.1, el control de la evaluación parcial puede ser estructurado en dos niveles: el llamado nivel *local* y el *global*. El problema de la *terminación local* se concreta en nuestro marco en el problema de la terminación del desplegado, i.e., como controlar que la expansión de los *árboles de narrowing locales*, que proporcionan la evaluación parcial de las llamadas en el programa bajo consideración, se mantengan finitos. El problema de la *terminación global* tiene que ver con la selección de un conjunto adecuado S de llamadas a especializar para el cual el programa especializado y el conjunto de llamadas inicial sea S-cerrado; en otras palabras, este problema es el de la terminación del desplegado recursivo, i.e., como parar la construcción de los árboles de *narrowing* locales mientras se garantiza el grado deseado de especialización (precisión) y se alcanza la condición de cierre. Partiendo de un conjunto de llamadas inicial con respecto a las cuales se desea especializar un programa, encontrar el conjunto de términos S que asegura que se cumplirá la condición de cierre, normalmente requiere aumentar el conjunto de llamadas inicial con términos nuevos. Esto introduce el problema de mantener el conjunto S finito durante el proceso de evaluación parcial por medio de un operador de abstracción apropiado que garantice la terminación.

En adelante seguimos esta aproximación al problema, que tiene su origen en el marco introducido por Martens y Gallagher para asegurar la terminación global de la deducción parcial [145], en el que se establece una clara distinción entre control local y global, y que nosotros reformulamos para poder tratar con funciones en combinación con variables lógicas. Esta aproximación contrasta con la adoptada por Glück y Sørensen en [86, 188] y Turchin en [198], donde estos dos niveles no se distinguen explícitamente, ya que solamente se permite realizar un paso de desplegado y se construye una gran estructura que comprende algo similar a nuestros árboles de *narrowing* locales y a los árboles globales de Martens y Gallagher [145]. En el próximo apartado trataremos el problema de encontrar operadores apropiados que resuelvan el problema del control de la terminación mientras se garantiza que se realiza aún todo el desplegado que sea posible para obtener una buena precisión en la especialización.

En este apartado, sin embargo, estamos interesados en formalizar un algoritmo genérico para la evaluación parcial de programas lógico funcionales basado en el *narrowing*, así como definir operadores genéricos, que aseguran que el programa residual está cubierto por el conjunto de términos evaluados parcialmente. El algoritmo es genérico con respecto a:

1. la *estrategia de narrowing* que construye los árboles de búsqueda locales;

2. la regla de desplegado (*unfolding rule*) que determina cuándo y cómo detener la construcción de los árboles locales (atendiendo a algún criterio de parada);

3. el *operador de abstracción* empleado para garantizar la terminación global del proceso de evaluación parcial.

La regla de desplegado determina las expresiones sobre las que se dará un paso de *narrowing* (respetando una estrategia φ) y asegura que el proceso de desplegado termina devolviendo un conjunto finito de resultantes. Esto es, la regla de desplegado garantiza la denominada *terminación local* del proceso de NPE.

Definición 3.3.10 (Regla de Desplegado)
Una regla de desplegado $U_\varphi(s, \mathcal{R})$ *(o simplemente U_φ) es una aplicación que*

devuelve una evaluación parcial para el término s en el programa $\mathcal{R}$ utilizando la relación de narrowing $\leadsto_\varphi$ *(i.e., un conjunto de resultantes).*

Dado un conjunto de términos S, denotamos por $U_\varphi(S,\mathcal{R})$ la unión de los conjuntos $U_\varphi(s,\mathcal{R})$, para cada s en S.

En otras palabras, $U_\varphi(S,\mathcal{R})$ denota una *evaluación parcial de S en $\mathcal{R}$ utilizando* $\leadsto_\varphi$.

El *operador de abstracción* garantiza que el proceso de NPE termina alcanzando la condición de cierre, y asegura la finitud del conjunto de términos evaluados parcialmente así como el adecuado grado de *polivarianza* (i.e., la capacidad para producir una especialización que contiene varias definiciones especializadas a partir de una única definición original).

Definición 3.3.11 (Operador de Abstracción)
Dado un conjunto finito de términos T y un conjunto de términos S (que forman la configuración[10] *actual de términos evaluados parcialmente), un* operador de abstracción *es una función que devuelve un conjunto finito de términos $abstract(S,T)$ tal que:*

1. *si $s \in abstract(S,T)$, entonces existe un $t \in (S \cup T)$ tal que $t|_p = \theta(s)$ para alguna posición p y substitución θ;*

2. *para todo $t \in (S \cup T)$, t es cerrado con respecto al conjunto de términos en $abstract(S,T)$.*

Intuitivamente, la primera condición garantiza que el operador de abstracción no introduce símbolos de función nuevos que no aparezcan en el conjunto de entrada S y en T, mientras que la segunda condición asegura que el conjunto de términos resultante cubre las llamadas especializadas previamente y que la condición de cierre se preserva a traves de las sucesivas operaciones de abstracción.

El algoritmo genérico para la NPE está parametrizado por la regla de desplegado U_φ y el operador de abstracción *abstract*, en el estilo de [75].

Algoritmo 1
Entrada: *un programa $\mathcal{R}$ y un conjunto de términos T*
Salida: *un conjunto de términos S*
Inicialización: $i := 0$; $T_0 := abstract(\emptyset, T)$
Repetir
  1. $\mathcal{R}' := U_\varphi(T_i, \mathcal{R})$;
  2. $T_{i+1} := abstract(T_i, \mathcal{R}'_{calls})$;
  3. $i := i + 1$;

Hasta que $T_i = T_{i-1}$ *(módulo renombramientos)*
Devolver $S := T_i$

De forma semejante a [145],el operador *abstract* se aplica en cada iteración del Algoritmo 1, para ajustar el control de la polivarianza tanto como sea necesario. La polivarianza se obtiene gracias al uso del mecanismo de propagación de la

[10] Para una caracterización más general de *configuración*, ver [14].

información basado en la unificación propio del *narrowing*, que permite computar una variedad de especializaciones independientes para una determinada llamada con respecto a diferentes instancias de sus variables. La NPE también es potencialmente capaz de producir especialización poligenética, mediante la especialización de expresiones complejas anidadas (ya que estos términos, al ser tratados como unidades individuales durante el proceso de evaluación parcial, producen una única definición especializada construida a partir de las definiciones de cada una de las funciones que forman la expresión compleja). Sin embargo, en presencia de símbolos de función primitivos, este tipo de especialización solamente es efectivo cuando se introducen mecanismos de control específicos, como los estudiados en el Capítulo 8.

Es conveniente matizar que si el programa se especializa con respecto a una expresión compleja g, en el contexto actual en el que los términos se especializan por separado, el conjunto de términos inicial que se suministra al Algoritmo 1 es $T = g_{calls}$. Nótese también que la salida del Algoritmo 1, para un programa $\mathcal{R}$, no es una evaluación parcial, sino un conjunto de términos S a partir del cual $U_\varphi(S, \mathcal{R})$ determina, de forma no ambigua, el programa evaluado parcialmente $\mathcal{R}'$ a partir de $\mathcal{R}$ empleando U_φ.

Informalmente, podemos describir el Algoritmo 1 del siguiente modo: dado un conjunto (inicial) de términos T_i y un programa $\mathcal{R}$, construimos una evaluación parcial para T_i en $\mathcal{R}$, utilizando la regla de desplegado U_φ para producir el conjunto de resultantes $\mathcal{R}'$; a continuación se añaden apropiadamente al conjunto T_i los términos que aparecen en las rhs's de las resultantes (que forman el conjunto $\mathcal{R}'_{calls}$) y no están cubiertos por los elementos de T_i, empleando el operador de abstracción para obtener el nuevo conjunto de términos T_{i+1}, que será empleado en la siguiente iteración. El proceso se repite en cada iteración hasta que el algoritmo se detiene porque el conjunto actual de términos evaluados parcialmente coincide con la configuración previa. El siguiente ejemplo ilustra el uso del Algoritmo 1.

Ejemplo 12 *Sea el programa que define la adición de números naturales, construidos usando los operadores* 0 *y* s, $\mathcal{R} = \{X + 0 \to X,\ X + s(Z) \to s(X + Z)\}$, *y el objetivo ecuacional* $g \equiv X + s(s(0)) = y$. *Para aplicar el Algoritmo 1, vamos a considerar una regla de desplegado que simplemente construye los árboles de* narrowing *local desplegandolos un sólo nivel usando el* narrowing *sin restricciones (de forma similar a como actua la supercompilación positiva) y un operador de abstracción que sólo añade a un conjunto* T *aquellos términos (o subtérminos) que no son* T*-cerrados (siguiendo el estilo en el que la deducción parcial define un operador de abstracción). El algoritmo comienza con el conjunto de términos inicial* $T = g_{calls} = \{X + s(s(0))\}$ *y, siguiendo el procedimiento descrito anteriormente, se obtiene:*

- *Iteración* $(i = 0)$.
  $T_0 = \{X + s(s(0))\}$ *y el resultado de aplicar el operador de desplegado es el árbol representado en la Figura 3.1(a), que conduce al programa*

especializado

$$\mathcal{R}' = \{ \quad X + s(s(0)) \to s(X + s(0))\}.$$

Ahora, el operador de abstracción definido más arriba intenta probar si el único término del conjunto $\mathcal{R}'_{calls} = \{s(X + s(0))\}$ *es* T_0*-cerrado; como no lo es, ya que no existe substitución alguna* θ *tal que* $\theta(X + s(s(0))) = X + s(0)$*, se añade al conjunto* T_0 *el subtérmino* $X + s(0)$ *(que impide que* $s(X + s(0))$ *sea* T_0*-cerrado), para formar el conjunto* T_1*.*

- *Iteración* $(i = 1)$.
  $T_1 = \{X + s(s(0)), X + s(0)\}$ *y el operador de desplegado construye los árboles de la Figura 3.1(a) y (b), siendo*

  $$\begin{aligned} \mathcal{R}' = \{ \quad & X + s(s(0)) \to s(X + s(0)) \\ & X + s(0) \to s(X + 0)\}. \end{aligned}$$

  En esta ocasión $\mathcal{R}'_{calls} = \{s(X + s(0)), s(X + 0)\}$*. El término* $s(X + s(0))$ *es* T_1*-cerrado, pero el término* $s(X+0)$ *no lo es. El operador de abstracción añade al conjunto* T_1 *el subtérmino* $X + 0$ *(que impide que* $s(X + 0)$ *sea* T_1*-cerrado), para formar el conjunto* T_2*.*

- *Iteración* $(i = 2)$.
  $T_2 = \{X + s(s(0)), X + s(0), X + 0\}$ *y el operador de desplegado construye los árboles de la Figura 3.1(a), (b) y (c), siendo*

  $$\begin{aligned} \mathcal{R}' = \{ \quad & X + s(s(0)) \to s(X + s(0)) \\ & X + s(0) \to s(X + 0) \\ & X + 0 \to X\}. \end{aligned}$$

  Ahora $\mathcal{R}'_{calls} = \{s(X + s(0)), s(X + 0), X\}$ *y todos sus términos son* T_2*-cerrados, de forma que el operador de abstracción no modifica el conjunto* T_2*.*

La condición de parada del Algoritmo 1 se cumple, ya que $T_3 = T_2$*, con lo que el algoritmo termina devolviendo el conjunto* $S = \{X + s(s(0)), X + s(0), X + 0\}$*.*

La evaluación parcial de S *en* $\mathcal{R}$ *es el programa residual*

$$\begin{aligned} \mathcal{R}' = \{ \quad & X + s(s(0)) \to s(X + s(0)) \\ & X + s(0) \to s(X + 0) \\ & X + 0 \to X\}. \end{aligned}$$

Nótese que $\mathcal{R}' \cup g_{calls}$ *es* S*-cerrado.*

El siguiente teorema enuncia que, suponiendo que el Algoritmo 1 termina, éste computa un conjunto S de términos evaluados parcialmente y un conjunto de resultantes $\mathcal{R}'$ (la evaluación parcial de S en $\mathcal{R}$), tal que la condición de cierre para $\mathcal{R}' \cup T$ se alcanza independientemente de la estrategia de *narrowing*, la regla de desplegado y el operador de abstracción empleados.

$$\begin{array}{ccc} X + s(s(0)) & X + s(0) & X + 0 \\ \Big|\ {\scriptstyle \{z/s(0)\}} & \Big|\ {\scriptstyle \{z/0\}} & \Big|\ {\scriptstyle id} \\ s(X + s(0)) & s(X + 0) & X \\ \text{(a)} & \text{(b)} & \text{(c)} \end{array}$$

Figura 3.1: Arboles locales de *narrowing* para $X + s(s(0))$, $X + s(0)$ y $X + 0$.

Teorema 3.3.12 *[14] Sea $\mathcal{R}$ un TRS y T un conjunto de términos. Si el Algoritmo 1 termina computando el conjunto de términos S, entonces $\mathcal{R}' \cup T$ es S-cerrado, donde el programa especializado viene dado por $\mathcal{R}' = U_\varphi(S, \mathcal{R})$.*

Como un corolario de los teoremas del Apartado 3.3.1 sobre la corrección, el Teorema 3.3.12 establece la corrección parcial del algoritmo de evaluación parcial basado en el *narrowing* sin restricciones.

Es evidente que un operador de desplegado como el definido en el Ejemplo 12 soluciona el problema de la terminación al precio de una pérdida considerable de precisión. Por otra parte, un operador de abstracción ingenuo como el definido en el Ejemplo 12, si bien asegura que se alcanzará la condición de cierre, en general conduce a un problema de no terminación (ver Ejemplo 4.1 en [143]). En el próximo apartado se presenta una solución simple pero útil al problema de la terminación, mediante la introducción de dos operadores de desplegado y abstracción concretos que consiguen un grado de precisión adecuado mientras aseguran que se cumplirá la condición de cierre.

3.4 Control de la Terminación.

Para resolver el problema de la terminación local se han propuesto diversos métodos en la literatura de algunas áreas relacionadas: la imposición de límites a la profundidad de los árboles locales (*depth-bounds*) y la utilización de órdenes bien fundados para controlar el desplegado durante la construcción de los árboles de búsqueda son dos de dichos métodos [47]. La utilización de límites a la profundidad de los árboles locales es una técnica demasiado burda que no garantiza la precisión de la especialización, mientras que la técnica más sofisticada de utilizar órdenes bien fundados puede producir un desplegado no acorde con su punto óptimo. Por otra parte, para el control global, es común considerar operadores de abstracción que utilizan órdenes bien fundados en combinación con la noción de generalización más especifica msg[11]. En un contexto de especialización *on-line*, los órdenes bien fundados son, en ocasiones, demasiado rígidos

[11] El hecho de que los operadores de abstracción suelan estar basados en la noción de msg es la causa de que también reciban el nombre de *operadores de generalización*.

o demasiado complejos. También es bien conocido que un empleo excesivo de la generalización puede conducir a una pérdida de precisión en la especialización. Estas son razones suficientes para que dirijamos nuestra atención a una técnica que ha ganado reciente popularidad: la utilización de buenos preórdenes en lugar de órdenes bien fundados.

Las técnicas que se van a introducir para el control de la terminación del algoritmo de NPE, pueden verse como una adaptación a nuestro marco de otras técnicas ya existentes para el control de la terminación de la deducción parcial y la supercompilación positiva. Para el control de la terminación local seguimos principalmente la aproximación de Sørensen y Glück en [188], que utiliza buenos preórdenes para asegurar la terminación cuando se construyen los llamados *árboles de procesos parciales* (*partial process trees*). En lo que respecta al control de la terminación global, nuestra aproximación toma prestadas algunas ideas del método para asegurar la terminación global de la deducción parcial presentado por Martens y Gallagher en [145]. Las principales diferencias de nuestro método con respecto al de Martens y Gallagher son las siguientes: (1) para controlar la terminación utilizamos, también en este nivel global, un buen preorden, en lugar de un orden bien fundado; y (2) no registramos las relaciones de dependencia entre las expresiones; por contra, preferimos utilizar conjuntos de términos antes que adaptar el tipo de estructuras arborescentes que ellos utilizan. Esto simplifica nuestra formulación manteniéndola útil.

Una herramienta común para probar propiedades de terminación se basa en la noción intuitiva de (pre)órdenes en los que un término que es "sintácticamente más simple" que otro (en algún sentido que se especificará) es menor que ese otro dentro del (pre)orden. La siguiente definición extiende la relación de subsumción homeomórfica (*homeomorphic embedding*) [59] para términos con variables (y formaliza la noción de "sintácticamente más simple"). En las pruebas de terminación para TRS's [60] se han empleado variantes de esta relación y también para asegurar la terminación local de la deducción parcial [41]. A partir de su utilización por parte de Sørensen y Glück en [188], varios trabajos recientes han utilizado esta relación (o variantes de ella) para evitar cómputos infinitos durante el proceso de evaluación parcial (e.g., ver [83, 109, 122, 130, 131, 132, 188, 201]).

Definición 3.4.1 (Relación de Subsumción Homeomórfica) *[188]*
La relación de subsumción homeomórfica $\trianglelefteq$ *sobre términos de* $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ *se define como la relación más pequeña que satisface que* $x \trianglelefteq y$ *para todo* $x, y \in \mathcal{X}$*, y* $s \equiv f(s_1, \ldots, s_m) \trianglelefteq g(t_1, \ldots, t_n) \equiv t$*, si y sólo si*

1. $f \equiv g$ *(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, se dice que $s \trianglelefteq t$ si s puede obtenerse de t borrando ciertos operadores. Por ejemplo, $\sqrt{}\sqrt{}(u \times (u+v)) \trianglelefteq (w \times \sqrt{}\underline{\sqrt{}\sqrt{}}((\sqrt{}\underline{u}+\sqrt{}u)\underline{\times}(\sqrt{}\underline{u+}\sqrt{}\underline{v})))$.

El siguiente resultado es una reformulación del teorema de Kruskal (*Kruskal's Tree Theorem*) [59, 130].

Teorema 3.4.2 (Teorema de Kruskal) *La relación de subsumción homeomórfica $\trianglelefteq$ es un buen preorden sobre el conjunto $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ para signaturas $\mathcal{F}$ finitas.*

Como consecuencia del teorema anterior, $\trianglelefteq$ es un preorden y se cumple que, 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 es *auto-subsumida* (*self-embedding*).

Vamos a utilizar la relación de subsumción homeomórfica $\trianglelefteq$ tanto para el control local como para el global. En el Apartado 3.4.1 se empleará para definir una regla de desplegado que garantice la terminación local, i.e., una condición que impide que los árboles de *narrowing* se expandan infinitamente. En el Apartado 3.4.2, se emplea también para definir un operador de abstración que asegura la terminación global (de la instancia seleccionada) del algoritmo genérico de NPE.

3.4.1 Terminación Local.

En este apartado, se introduce una regla de desplegado que intenta maximizar el desplegado sin comprometer la terminación. La estrategia que se emplea está basada en el uso del preorden de subsumción para detener las derivaciones de *narrowing*. La parada se produce cuando se detecta que los términos en la derivación comienzan a crecer, por hacerse "sintácticamente más complejos", y puede haber riesgo de no terminación. Esta es una idea simple, pero menos cruda que imponer de forma *ad hoc* un limite a la profundidad de los árboles de *narrowing* local, y garantiza un desplegado finito en todos los casos. A continuación precisamos esta idea.

Para producir árboles de *narrowing* locales finitos, introducimos el siguiente criterio: para evitar secuencias de llamadas "divergentes", se compara cada redex del término actual con los redexes que fueron seleccionados en los pasos de *narrowing* anteriores, i.e., con los redexes seleccionados en los (términos) ancestros[12] que están en la misma rama que el término actual considerado; si alguna de las llamadas comparadas están en la relación de subsumción con el redex actual, se detiene la expansión del árbol de *narrowing* local en esa rama y se consideran las otras alternativas; si no, la expansión prosigue. Para formalizar este criterio necesitamos las siguientes definiciones y notaciones.

Definición 3.4.3 (Términos Comparables)
Sean s y t términos. Decimos que s y t son comparables, *escrito $comparable(s, t)$, si y sólo si $\mathcal{H}ead(s) \equiv \mathcal{H}ead(t)$*

[12] Aquí, el concepto de ancestro empleado es el definido en el Apartado 2.4. Nociones más refinadas y restrictivas, como el concepto estándar en la deducción parcial de ancestro (*covering ancestor*) [47] (i.e., átomos seleccionados por la regla de cómputo en un objetivo de una derivación SLD, de los que procede el átomo seleccionado actualmente y que comparten el mismo símbolo de predicado en cabeza), o el concepto de posición dependiente de la Definición 8.2.1, conducen a estrategias de desplegado más liberales. Por el momento, se prefiere postponer este tipo de optimizaciones hasta su discusión en el Capítulo 8.

Definición 3.4.4 (Derivación Admisible)
Sea $D \equiv (t_0 \overset{[u_0,\theta_0]}{\leadsto}_{\varphi} \ldots \overset{[u_{n-1},\theta_{n-1}]}{\leadsto}_{\varphi} t_n)$ *una derivación de* narrowing *para* t_0 *en* $\mathcal{R}$. *Decimos que* D *es* admisible *si y solamente si no contiene un par de redexes comparables incluidos en la relación de subsumción* $\trianglelefteq$. *Formalmente,*

$$\begin{array}{ll} admisible(D) \Leftrightarrow & \forall i = 1, \ldots, n,\ \forall u \in \varphi(t_i),\ \forall j = 0, \ldots, i-1. \\ & (comparable(t_j|_{u_j}, t_i|_u) \Longrightarrow t_j|_{u_j} \ntrianglelefteq t_i|_u). \end{array}$$

Para formular la regla de desplegado concreta, todavía se necesita introducir la siguiente definición auxiliar.

Definición 3.4.5 (Arbol de Narrowing de No-subsumción)
Dado un término t_0 *y un programa* $\mathcal{R}$, *definimos el* árbol de *narrowing* de no-subsumción $\tau_{\varphi}^{\trianglelefteq}(t_0, \mathcal{R})$ *para* t_0 *en* $\mathcal{R}$ *como sigue:*

$$D \equiv [t_0 \overset{[u_0,\theta_0]}{\leadsto}_{\varphi} \ldots \overset{[u_{n-1},\theta_{n-1}]}{\leadsto}_{\varphi} t_n \overset{[u_n,\theta_n]}{\leadsto}_{\varphi} t_{n+1}] \in \tau_{\varphi}^{\trianglelefteq}(t_0, \mathcal{R}),$$

si se cumplen las siguientes condiciones:

1. *la derivación* $(t_0 \overset{[u_0,\theta_0]}{\leadsto}_{\varphi} \ldots \overset{[u_{n-1},\theta_{n-1}]}{\leadsto}_{\varphi} t_n)$ *es admisible y*
2. (a) *la hoja* t_{n+1} *en* D *es un valor, o*
   (b) *la hoja* t_{n+1} *en* D *es un término irreducible que no es un valor, o*
   (c) *existe una posición* $u \in \varphi(t_{n+1})$ *y un número* $i \in \{1, \ldots, n\}$ *tal que* $t_i|_{u_i}$ *y* $t_{n+1}|_u$ *son comparables, y* $t_i|_{u_i} \trianglelefteq t_{n+1}|_u$.

Resumiendo, las derivaciones se cortan cuando: D es una derivación de éxito (2.a); o D es una derivación de fallo (2.b); o bien D es una derivación incompleta y existe riesgo de no terminación porque los redexes en consideración satisfacen la relación de subsumción (2.c). Nótese que, por la forma en la que se construyen los árboles de *narrowing*, los redexes comparables en una rama forman cadenas decrecientes de acuerdo con la relación de no-subsumción. El siguiente resultado, que establece la utilidad de los árboles de *narrowing* de no-subsumción para asegurar la terminación local, es una conseciencia directa del hecho anterior y del Teorema 3.4.2.

Teorema 3.4.6 *[14] Para un TRS* $\mathcal{R}$ *y un término* t, $\tau_{\varphi}^{\trianglelefteq}(t, \mathcal{R})$ *es un árbol de* narrowing *finito (y posiblemente incompleto) para* $\mathcal{R} \cup \{t\}$ *utilizando* $\leadsto_{\varphi}$.

Finalmente, formalizamos la noción de regla de desplegado inducida por la noción de árbol de *narrowing* de no-subsumción.

Definición 3.4.7 (Regla de Desplegado de No-subsumción)
Para un TRS $\mathcal{R}$ *y un término* t, *definimos* $U_{\varphi}^{\trianglelefteq}(t, \mathcal{R})$ *como el conjunto de las resultantes asociadas a las derivaciones en* $\tau_{\varphi}^{\trianglelefteq}(t, \mathcal{R})$.

3.4.2 Terminación Global.

En este apartado definimos un operador de abstracción concreto utilizando una estructura simple (un conjuntos de términos) que manipulamos de tal forma que se garantiza la terminación global del algoritmo génerico de NPE y todavía produce el grado adecuado de polivarianza que permite no perder excesiva precisión a pesar del uso del operador msg. En [145], Martens y Gallagher utilizan estructuras más sofisticadas (como son sus árboles globales) que, si bien pueden mejorar el grado de especialización en algunos casos, son más costosos en consumo de recursos de cómputo.

En cada iteración del Algoritmo 1, la configuración actual de términos parcialmente evaluados $T_i \equiv \{t_1, \ldots, t_n\}$ se transforma para que los términos resultantes de la evaluación parcial de T_i in $\mathcal{R}$ quedan cubiertos por la nueva configuración T_{i+1}. Esta transformación se realiza mediante el siguiente operador de abstracción *abstract**, que emplea la noción de generalización más específica.

Definición 3.4.8 (Operador de Abstracción de No-subsumción) *Sean S una configuración y T un conjunto de términos. Definimos el operador de abstracción de no-subsumción abstract* inductivamente como sigue:* $abstract^*(S, T)$ =

$$\begin{cases} S & si\ T \equiv \emptyset \ \texttt{o}\ (T \equiv \{t\},\ t \in \mathcal{X}) \\ abstract^*(\ldots abstract^*(S, \{t_1\}), \ldots, \{t_n\}) & \texttt{si } T \equiv \{t_1, \ldots, t_n\}, n > 0 \\ abstract^*(S, \{t_1, \ldots, t_n\}) & \texttt{si } T \equiv \{t\},\ t \equiv c(t_1, \ldots, t_n),\ c \in \mathcal{C} \\ abs\_call(S, t) & \texttt{si } T \equiv \{t\},\ \mathcal{H}ead(t) \in \mathcal{D} \end{cases}$$

La función auxiliar abs_call se define como

- $abs\_call(\emptyset, t) = \{t\}$
- $abs\_call(\{s_1, \ldots, s_n\}, t) =$

$$\begin{cases} \{s_1, \ldots, s_n, t\} & \texttt{si } \not\exists i \in \{1, \ldots, n\}.(comparable(s_i, t) \wedge s_i \trianglelefteq t) \\ abstract^*(\{s_1, \ldots, s_n\}, T') & \texttt{si } i\ es\ el\ m\acute{a}ximo\ j \in \{1, \ldots, n\} \texttt{ tal que } (comparable(s_j, t) \wedge s_j \trianglelefteq t), \texttt{ y } \theta(s_i) = t \texttt{ para alguna } \theta, \texttt{ donde } T' = \mathcal{R}an(\theta) \\ abstract^*(S', T') & \texttt{si } i\ es\ el\ m\acute{a}ximo\ j \in \{1, \ldots, n\} \texttt{ tal que } (comparable(s_j, t) \wedge s_j \trianglelefteq t), \texttt{ y } s_i \not\leq t, \texttt{ donde } S' = \{s_1, \ldots, s_n\} \setminus \{s_i\}, msg(s_i, t) = \langle w, \{\theta_1, \theta_2\}\rangle \texttt{ y } T' = \{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2). \end{cases}$$

La definición anterior puede parecer técnicamente compleja; sin embargo, *abstract** procede de una manera bastante simple, como se infiere de la siguiente explicación informal. Dada una configuración S, para añadir a S un nuevo término t (encabezado por un símbolo de función definido), la función *abstract** realiza lo siguiente:

- Si t no subsume ningún término comparable de S, entonces t simplemente se añade a S sin tener en cuenta ninguna otra consideración (e.g., si el término ya está cubierto), pues lo que se persigue es obtener la máxima polivarianza sin incurrir en *problemas de subsumción*[13].

- Si t subsume algún término comparable de S, entonces el término t no debe añadirse a la configuración S porque se violaría la condición de no subsumción. Sin embargo, si queremos preservar la completitud del proceso de evaluación parcial, el término t es una llamada que debe ser cubierta convenientemente. Con esta finalidad, el operador de abstracción selecciona el término s' más a la derecha en la configuración S (que es tratada como una secuencia en la que los términos son numerados según el orden de su inserción) de entre los términos comparables a t que causan problemas de subsumción y distingue dos casos:

  - si t es una instancia de s' con substitución θ, entonces el término t ya está cubierto; sin embargo, eso no basta si queremos que se cumpla la condición de cierre; se necesita que los subtérminos de t también estén cubiertos; por esa causa, el operador de abstracción intenta añadir recursivamente a S los términos en el rango de θ;
  - en otro caso, cuando el término t no está cubierto, con objeto de generar una configuración que siga cumpliendo la condición de cierre, computa el msg de s' y t, digamos $\langle w, \theta_1, \theta_2 \rangle$ y entonces intenta añadir w, así como los términos en el rango de θ_1 y θ_2, a la configuración resultante de borrar s' en S.

Llegados a este punto, son pertinentes las siguientes observaciones:

- En el último caso de *abs_call*, nótese que si el término s' no se borrase de la configuración S, el operador de abstracción podría entrar en un bucle infinito [14].

- El segundo caso de *abstract** y el primero de *abs_call* hacen que las configuraciones se traten, en la práctica, como secuencias de términos.

- Si bien nuestras configuraciones, tratadas como secuencias, pueden parecer más pobres que las estructuras arborescentes de Martens y Gallagher en [145] y Sørensen y Glück en [188], este tipo de configuraciones se usan habitualmente en las implementaciones reales y, en muchos casos, pueden evitar la duplicación de definiciones (de funciones).

- Finalmente, la pérdida de precisión que causa el empleo del oprerador msg es muy razonable y se compensa con la simplicidad del método obtenido.

Concluimos este apartado enunciando una serie de resultados sobre el operador de abstracción concreto que acabamos de definir y el algoritmo de NPE.

[13] Lo que podría ocasionar que el proceso de adición de nuevos términos al conjunto S no terminase nunca.

El siguiente lema establece que *abstract** es una instancia correcta de la noción genérica de operador de abstracción, según se ha caracterizado en la Definición 3.3.11 .

Lema 3.4.9 *[14] La función abstract* es un operador de abstracción.*

El siguiente teorema establece la terminación local y global del algoritmo genérico para la NPE.

Teorema 3.4.10 *[14] El Algoritmo 1 termina para la regla de desplegado de no-subsumción* $U_{\varphi}^{\trianglelefteq}$ *(Definición 3.4.7) y el operador de abstracción de no-subsumción abstract* (Definición 3.4.8).*

Como una consecuencia de los teoremas sobre la corrección del Apartado 3.3.1, el Teorema 3.4.10 establece la corrección total del algoritmo de evaluación parcial basado en *narrowing* sin restricciones.

3.5 Comparación con Otros Metodos de Evaluación Parcial.

En esta sección intentamos establecer una comparación entre el método de NPE y otros métodos de evaluación parcial en áreas relacionadas[14], como son: la evaluación parcial clásica de programas funcionales (PEC) [108], la deforestación (DF) [203], la supercompilación (SC) [197, 199], la supercompilación positiva (PS) [188, 190], la computación parcial generalizada (GPC) [74, 194], la deducción parcial (PD) [35, 75, 137, 145], la deducción parcial conjuntiva (CPD) [83, 128] y la evaluación parcial de programas lógico–funcionales basados en reescritura (PER) [122, 123].

Para efectuar esta comparación, vamos a utilizar como criterio las siguientes propiedades, que pueden servir para caracterizar un evaluador parcial:

Lenguaje fuente. Aquí, llamamos *lenguaje fuente* a aquel que se pretende especializar. También es interesante resaltar la semántica operacional del lenguaje fuente, contrastándola con el mecanismo de base utilizado por el evaluador parcial.

Mecanismo de base. Si entendemos un evaluador parcial como un intérprete extendido con la posibilidad de generar expresiones residuales y capaz de efectuar cómputos con los datos estáticos, llamamos *mecanismo de base* del evaluador parcial al mecanismo operacional utilizado en la realización de dichos cómputos. También distinguiremos la estrategia de evaluación utilizada, cuando ésto sea posible. Es bien conocido que la coincidencia entre la semántica

[14]Para más información sobre las particularidades de estos métodos de evaluación parcial, ver el Apartado 3.2.

Espec.	Leng. Fuente (Sem. Oper.)	Mec. Base (estrategia)	Prop. inform.	Reest. P. Control variante	genético	Test KMP	Elim. Estruc.
PEC	Funcional (reduc., orden aplicativo)	Específico (impaciente)	const.	poli	mono	no	no
DF	Funcional (reduc., orden normal)	Específico (perezoso)	const.	poli	poli	no	si
SC	Refal (reduc., orden aplicativo)	driving (perezoso)	restric.	poli	poli	si	si
PS	Funcional (reduc., orden normal)	driving (perezoso)	unific.	poli	poli	si	si
GPC	Funcional (reduc., orden normal)	demostrador de teoremas (perezoso)	restric.	poli	poli	si	si
PD	Cláusulas Horn (resolución SLD)	resolución SLD (—–)	unific.	poli	mono	si	no
CPD	Cláusulas Horn (resolución SLD)	resolución SLD (—–)	unific.	poli	poli	si	si
PER	Lóg.–funcional (resolución SLD + residuación)	*narrowing* normalizante (impaciente)	unific.	poli	mono	no	si
NPE	Lóg.–funcional (*narrowing* impaciente)	*narrowing* (impaciente)	unific.	poli	mono	si	si

Tabla 3.1: Características de los evaluadores parciales.

operacional del lenguaje fuente y el mecanismo operacional de base de un evaluador parcial facilita las pruebas de corrección [126].

Propagación de la información. En los cómputos, que se realizan en tiempo de evaluación parcial, los datos conocidos se propagan a lo largo de las secuencias (o árboles) de computación permitiendo una mayor concreción de las expresiones. Decimos que hay una *propagación de la información*. El grado de especialización del programa transformado es tanto mejor cuanto mayor sea el grado de propagación de la información. Un evaluador parcial será tanto más potente cuanto mayor sea el grado de información que propague.

La información no solamente se propaga mediante cómputos ordinarios, sino también cuando se realizan comprobaciones de las condiciones de las construcciones "if–then–else" u "otherwise" de algunos lenguajes. Para fijar ideas podemos centrarnos en una construcción "if–then–else" del tipo de las expresiones guardadas de BABEL [157], $((c_1 = c_2) \Rightarrow t_1 \Box t_2)$, y en un evaluador parcial $\mathcal{T}$. En lo que sigue $\mathcal{T}[E]$ designa la especialización de una expresión E (en un programa P), obtenida al aplicar el evaluador parcial $\mathcal{T}$. Distinguimos varios niveles de propagación de la información [87]:

1. Propagación de constantes: Si no hay propagación de la información almacenada en la condición, es decir, cuando $\mathcal{T}[(c_1 = c_2) \Rightarrow t_1 \Box t_2] = ((c_1 = c_2) \Rightarrow \mathcal{T}[t_1] \Box \mathcal{T}[t_2])$.

2. Propagación basada en unificación: Se utiliza unificación para comprobar si se cumple una condición, obteniéndose un mgu, y después se propaga la información almacenada en la substitución mediante la instanciación de la expresión que aparece en la parte "*then*", es decir, $\mathcal{T}[(c_1 = c_2) \Rightarrow t_1 \Box t_2] = ((c_1 = c_2) \Rightarrow \mathcal{T}[\sigma(t_1)] \Box \mathcal{T}[t_2])$, donde $\sigma = mgu(c_1, c_2)$. En este caso se habla de propagación de la información positiva.

3. Propagación basada en restricciones: La condición pasa a formar parte de la expresión de la parte "*then*" como una ecuación y de la parte "*else*" como una desigualdad, esto es, $\mathcal{T}[(c_1 = c_2) \Rightarrow t_1 \Box t_2, R] = ((c_1 = c_2) \Rightarrow \mathcal{T}[t_1, R \cup \{c_1 = c_2\}] \Box \mathcal{T}[t_2, R \cup \{c_1 \neq c_2\}])$, donde R representa el conjunto de restricciones previas. En este caso se habla de que se propaga tanto la información positiva como la negativa.

Reestructuración de los puntos de control. Como ya se ha mencionado en el Apartado 3.1 la reestructuración de los puntos de control tiene que ver con las relaciones que se establecen entre los puntos de control del programa original y el residual. Recordamos que pueden distinguirse las siguientes capacidades de reestructuración en un evaluador parcial: monovariante, polivariante, monogenética y poligenética.

También prestaremos atención a la capacidad del evaluador parcial para superar ciertas pruebas de especialización, como por ejemplo mostrar la habilidad de eliminar estructuras intermedias, tal y como se presenta en el Ejemplo 15, y superar el llamado test KMP (i.e., si es capaz de obtener la optimización que es básica en el algoritmo de Knuth-Morris-Pratt para la búsqueda de patrones en cadenas de caracteres [114] que se presenta en Apartado 4.4).

La Tabla 3.1 resume las características de las distintas técnicas de transformación consideradas[15] y pone de manifiesto las ventajas e inconvenientes de la NPE tal y como se ha formulado hasta el momento. Respecto a los datos presentados en relación a la NPE son pertinentes los siguientes comentarios:

- los datos que aparecen en la tabla hacen referencia a la instancia impaciente (*call-by-value*) presentada en [14], i.e., la concreción del método genérico de la NPE, cuando se utiliza el *narrowing* condicional con estrategia impaciente (*innermost*) para el desplegado de los árboles de búsqueda y el operador de desplegado concreto de la Definición 3.4.7 (particularizado al caso de la estrategia impaciente, para controlar la expansión de los árboles locales) y el de abstracción de la Definición 3.4.8.

- si bien, potencialmente el método NPE permite la especialización poligenética ésta no es efectiva, como ya se ha comentado, por lo que hemos caracterizado el método NPE como monogenético;

- para lograr la eliminación de estructuras intermedias utilizando la instancia impaciente, se requieren pasos de normalización antes de dar un paso de desplegado en la construcción del árbol de *narrowing* local [14].

En los próximos capítulos, el objetivo es estudiar una instancia perezosa (*call-by-name*) del método NPE y dotarlo de la capacidad de realizar una especialización poligenética efectiva.

[15]Respecto a la evaluación parcial de programas lógico–funcionales basados en reescritura (PER), en [122], Lafave y Gallagher han aumentado su evaluador parcial con la posiblilidad de propagar información negativa basada en restricciones, lo que les ha permitido superar el test KMP.

Parte II

Especialización de Programas Lógico–Funcionales Perezosos.

Capítulo 4

Evaluación Parcial Dirigida por Narrowing Perezoso.

En el Apartado 2.9.1 hemos introducido la clase de programas con los que vamos tratar en lo que sigue. Las estrategias de evaluación perezosa permiten evitar (en el caso óptimo) la realización de cómputos innecesarios, con lo que se da al programador la posibilidad de no prestar atención al orden de evaluación de las expresiones, permiten también la definición de funciones parciales y no estrictas, y dan cuenta de computaciones que no terminan. Todas estas razones hacen que la capacidad de realizar evaluación perezosa sea una característica muy apreciada en los lenguajes integrados y justifica el estudio que vamos a emprender para dotar a los mismos de una herramienta de optimización basada en la evaluación parcial (perezosa).

En el capítulo precedente se ha presentado un algoritmo genérico para la evaluación parcial de programas lógico–funcionales, que es paramétrico con respecto a la estrategia de *narrowing* que se emplea en el desplegado de los árboles locales. Se han definido las nociones de *cierre* e *independencia* que son esenciales para probar la equivalencia computacional entre el programa original y el residual, para un conjunto distinguido de términos. Estas condiciones permiten probar la corrección y completitud de la NPE en el caso del *narrowing* sin restricciones, para programas que cumplen (al menos) las condiciones bajo las que es correcto y completo el *narrowing* sin restricciones. Parece razonable esperar que este resultado se traslade a otras estrategias de *narrowing*, i.e., que bajo los requerimientos de cierre e independencia y las condiciones que aseguran que una estrategia de *narrowing* es correcta y completa, se pueda garantizar la corrección de una instancia particular del método de NPE. Lamentablemente, como podremos comprobar, esto no sucede en general y se requieren pruebas y condiciones específicas que son dependientes de la estrategia de *narrowing* particular que se esté considerando. Por otro lado, la condición de independencia, que asegura que el programa residual no produce respuestas adicionales, no se obtiene en general automáticamente, a diferencia de la condición de cierre que puede alcanzarse mediante un operador de abstracción adecuado. Para algunas estrategias de *narrowing*, el programa residual puede incluso incumplir las restricciones (sintácticas) que son necesarias para la completitud de la estrategia considerada.

En este capítulo se formaliza un evaluador parcial perezoso para lenguajes lógico–functionales con una semántica operacional como la presentada en el Apartado 2.9.3. También se define una transformación de renombramiento del programa residual que consigue la condición de independencia. Esto se realiza mediante una fase de postproceso en la que se introducen símbolos de función nuevos y que obtiene un nuevo programa transformado, así como un conjunto de llamadas especializadas renombradas, de acuerdo con los nuevos nombres de función elegidos. Constatamos que la fase de postproceso de renombramiento, no solamente es vital para la obtención de la condición de independencia, sino también para garantizar la corrección de todo el proceso. En general, el programa evaluado parcialmente que se obtiene antes del postproceso podría incumplir alguna de las condiciones que son necesarias para la completitud de la estrategia de *narrowing* perezoso, o aún peor, incumplir alguna condición básica (e.g., la disciplina de constructores) que es imprescindible para que la estrategia perezosa pueda ejecutar el programa residual para las llamadas consideradas. El siguiente ejemplo ilustra este problema:

Ejemplo 13 *Sea el programa* $\mathcal{R} \equiv \{f(X) \to X\}$ *para el cual es posible derivar el siguiente programa residual* $\mathcal{R}'$*:*

$$\begin{array}{rcl} f(0) & \to & 0 \\ f(f(X)) & \to & X \end{array}$$

cuando se especializa con respecto a las llamadas $f(f(X))$ *y* $f(0)$*. Este programa contiene una llamada a función anidada en la lhs de la segunda regla, lo que viola la condición de ser CB y además hace que esta regla solape con la segunda, por lo que el programa tampoco es ortogonal. Se debe observar que el conjunto de llamadas especializadas* $\{f(0), f(f(X))\}$ *no es independiente, ya que contiene dos pares de llamadas que interfieren, la llamada* $f(0)$ *con la* $f(x)$ *y la* $f(x)$ *con la* $f(f(x))$*.*

En este ejemplo resulta bastante sencillo recuperar la disciplina de constructores y la ortogonalidad del programa residual mediante un postproceso de renombramiento, que asigne a la llamada $f(f(X))$ un nuevo nombre, e.g., $h(X)$, que deriva el programa transformado:

$$\begin{array}{rcl} f(0) & \to & 0 \\ h(X) & \to & X \end{array}$$

junto con un conjunto de llamadas parcialmente evaluadas, y convenientemente renombradas, $\{f(0), h(X)\}$ que es independiente.

En los próximos apartados concretamos una instancia del marco genérico para la NPE que se traduce en un método de especialización, en dos fases, para programas integrados con semántica perezosa, y estudiamos las condiciones que garantizan la corrección y completitud del método.

4.1 Un Evaluador Parcial Basado en Narrowing Perezoso.

En este apartado se formula una instancia perezosa del procedimiento genérico de NPE introducido en el Apartado 3.3.2 y mostramos su comportamiento.

Antes de pasar a formalizar el evaluador parcial perezoso, conviene introducir el concepto de evaluación parcial con restricción a hnf, que es fundamental en un contexto de evaluación parcial perezosa.

Definición 4.1.1 (Evaluación Parcial con Restricción a hnf) *Sea $\mathcal{R}$ un programa y s un término. Una evaluación parcial* con restricción *a hnf, $\mathcal{R}'$, de s en $\mathcal{R}$ es una evaluación parcial de s en $\mathcal{R}$ tal que, en cada derivación de* narrowing *perezoso $s \overset{\theta}{\leadsto}_{LN}^{*} s' \overset{\theta}{\leadsto}_{LN}^{*} t$ empleada para obtener una resultante del programa residual $\mathcal{R}'$, ningún término s' en hnf ha sido reducido por* narrowing *(i.e., el término t no proviene de una hnf en la que se han reducido posiciones por debajo de la posición Λ).*

En lo que sigue utilizaremos el acrónimo "hnf-PE" para hacer referencia a una evaluación parcial con restricción a hnf. Informalmente, diremos que en una derivación $s \overset{\theta}{\leadsto}_{LN}^{*} t$, que cumple las condiciones de la derivación de la Definició 4.1.1, el término s no se ha reducido (por *narrowing*) *más allá* de su hnf, o también que no ha *sobrepasado* su hnf. Intuitivamente, la razón para no permitir que la evaluación parcial de un término s sobrepase su hnf es que, en tiempo de ejecución, la evaluación de una llamada anidada $C[s]_p$, que contenga el término s en alguna posición p, podría no *demandar* la evaluación de s más allá de su hnf. Ya que los contextos $C[\ ]$ en los que s aparecerá no son conocidos en tiempo de evaluación parcial, imponiendo esta restricción evitamos interferir con la "naturaleza perezosa" de los cómputos en el programa especializado. El siguiente ejemplo ilustra que el no respetar esta restricción puede provocar la pérdida la completitud del proceso de evaluación parcial.

Ejemplo 14 *Sea el programa*

$$\mathcal{R} = \{ \quad \begin{array}{ll} R_1 : f(c(X)) & \to 0 \\ R_2 : g(c(X)) & \to c(g(X)) \}; \end{array}$$

y el conjunto de términos $S = \{f(X), g(X)\}$. Partiendo de las derivaciones

$$f(c(X)) \overset{[\Lambda, R_1, id]}{\leadsto}_{LN} 0$$

y

$$\begin{array}{llll} g(X) & \overset{[\Lambda, R_2, \{X/c(X_1)\}]}{\leadsto}_{LN} & c(g(X_1)) & \\ & \overset{[1, R_2, \{X_1/c(X_2)\}]}{\leadsto}_{LN} & c(c(g(X_2))) \overset{[1.1, R_2, \{X_2/c(X_3)\}]}{\leadsto}_{LN} & c(c(c(g(X_3)))) \end{array}$$

puede obtenerse el programa residual

$$\mathcal{R}' = \{ \quad \begin{array}{ll} R'_1 : f(c(X)) & \to 0 \\ R'_2 : g(c(c(c(X)))) & \to c(c(c(g(X)))) \}; \end{array}$$

como resultado de la evaluación parcial de $\mathcal{R}$ con respecto a S. Ahora, puede comprobarse que, para el término $f(g(X))$, existe la siguiente derivación perezosa en el programa original $\mathcal{R}$

$$f(g(X)) \overset{[1,R_2,\{X/c(X_1)\}]}{\leadsto_{LN}} f(c(g(X_1))) \overset{[\Lambda,R_1,id]}{\leadsto_{LN}} 0$$

que computa el resultado 0 con respuesta $\{X/c(X_1)\}$, mientras que la única derivación perezosa para esta llamada en $\mathcal{R}'$ es

$$f(g(X)) \overset{[1,R'_2,\{X/c(c(c(X_1)))\}]}{\leadsto_{LN}} f(c(c(c(g(X_1))))) \overset{[\Lambda,R'_1,id]}{\leadsto_{LN}} 0$$

que computa el resultado 0 con respuesta menos general $\{X/c(c(c(X_1)))\}$, siendo imposible computar en $\mathcal{R}'$ la respuesta $\{X/c(X_1)\}$. En otras palabras, se pierde la completitud del proceso de evaluación parcial.

El problema en el Ejemplo 14 proviene de que la llamada especializada $g(X)$ se ha evaluado más allá de una hnf en la derivación que dio lugar a la resultante R'_2 del programa residual $\mathcal{R}'$.

Ahora podemos dar la definición precisa de qué se entiende por un evaluador parcial perezoso

Definición 4.1.2 (Evaluador Parcial Perezoso)
Un evaluador parcial perezoso es una instancia concreta del Algoritmo 1 que se obtiene al elegir:

1. *la estrategia de* narrowing *perezoso, λ_{lazy}, para el desplegado de los árboles locales;*

2. *la regla de desplegado de la Definición 3.4.7 (parametrizada con la estrategia de* narrowing *perezoso, λ_{lazy}) y el preorden de subsumción de la Definición 3.4.1, reforzado con el criterio de parada adicional consistente en detener el desplegado de los árboles de* narrowing *local cuando se alcanza una hnf;*

3. *el operador de abstracción de no-subsumción, abstract*, de la Definición 3.4.8.*

En lo que sigue nos referiremos a esta instancia particular del algoritmo de NPE con el acrónimo LN-PE (del inglés *Lazy Narrowing-driven Partial Evaluation*).

El siguiente ejemplo, el "*append* doble", es un test estándar de eliminación de estructuras de datos intermedias. Este ejemplo demuestra que nuestro método puede eliminar estructuras intermedias y convertir un programa que recorre varias veces una estructura, realizando multiples "pasadas" sobre ésta, en otro programa que sólo recorre una vez la estructura considerada. Este efecto támbien lo consiguen la deforestación [203], el *tupling*[1] [51] y la supercompilación

[1]La transformación de *tupling* elimina recorridos paralelos de estructuras de datos idénticas mediante la fusión de bucles en una nueva función recursiva que se define sobre *tuplas*. Generalmente, el *tupling* es muy complicado y los algoritmos automáticos de *tupling* o bien revierten en un alto coste de ejecución (que impide su utilización en un sistema real), o bien tienen éxito para clases muy restringidas de programas.

positiva [186], entre otras técnicas de transformación (ver la Tabla 3.1), mientras que la evaluación parcial clásica [108] y la deducción parcial [35, 75, 137, 145] no consiguen en general esta optimización. Este ejemplo también muestra que la introducción de secuencias de normalización entre los pasos de *narrowing* puede emplearse como una alternativa *segura* a ciertas técnicas de optimización *ad hoc* utilizadas en otras metodologías para la transformación de programas. Eligiendo un subconjunto terminante de reglas del programa para ser empleadas durante la normalización no se corre el riesgo de entrar en un bucle infinito: así pues, cada paso de *narrowing* normalizante termina y, por lo tanto, los árboles de *narrowing* local se construyen en un tiempo finito. Tampoco se pierde la completitud del proceso, debido a que solamente se descartan ramas alternativas que no producen nuevas soluciones (ver Apartado 2.9.5).

Ejemplo 15 (*Append* Doble) *Consideremos el conocido programa terminante que concatena dos listas*[2]*,*

$$\begin{array}{rcl} app([\,], y_s) & \to & y_s \\ app(x : x_s, y_s) & \to & x : app(x_s, y_s) \end{array}$$

y la llamada inicial $app(app(x_s, y_s), z_s)$, que se pretende especializar. Esta llamada concatena tres listas: comienza concatenando las dos primeras, lo que conduce a una estructura intermedia a la cual añade después la tercera de las listas. Realizamos la evaluación parcial utilizando narrowing *perezoso normalizante. Comenzando con el conjunto inicial de términos $T = \{app(app(x_s, y_s), z_s)\}$, y siguiendo el procedimiento descrito en el Algoritmo 1, se computan los árboles dibujados en la Figura 4.1 para el conjunto de términos evaluados parcialmente $S = \{app(app(x_s, y_s), z_s), app(x_s, y_s)\}$. Entonces se obtiene el siguiente programa residual $\mathcal{R}'$:*

$$\begin{array}{rcl} app(app([\,], y_s), z_s) & \to & app(y_s, z_s) \\ app(app(x : x_s, y_s), z_s) & \to & x : app(app(x_s, y_s), z_s) \\ app([\,], z_s) & \to & z_s \\ app(y : y_s, z_s) & \to & y : app(y_s, z_s) \end{array}$$

que es capaz de concatenar las tres listas recorriéndolas sólo una vez. Nótese que la clave del éxito en este ejemplo ha sido la normalización (reflejada en la Figura 4.1 por una flecha "$\downarrow$" que indica un paso de reescritura). Sin este paso de simplificación, el orden de subsumción se habría satisfecho demasiado pronto, en la rama derecha del primero de los árboles de la Figura 4.1, con lo cual el desplegado del árbol se habría detenido en una posición inadecuada, impidiendo así una especialización óptima. El algoritmo de driving *en [188] (que define la única versión de la supercompilación positiva para la cual se garantiza la terminación) alcanza el mismo efecto por medio de una técnica* ad hoc

[2]En lo que sigue utilizamos una notación para las listas similar a la de Haskell o Curry, donde "[ ]" (la lista vacia) y ':' (el operador de concatenación) son los constructores de listas. La lista "$x : x_s$" consiste en la lista, no vacía, formada por un primer elemento x seguido de la lista x_s. También denotan listas "$[a : b : x_s]$" y "$[a, b, c]$", que son, respectivamente, notaciones abreviadas de "$a : (b : x_s)$" y de "$a : (b : (c : [\,]))$", donde a, b, y c son elementos de un cierto tipo de datos.

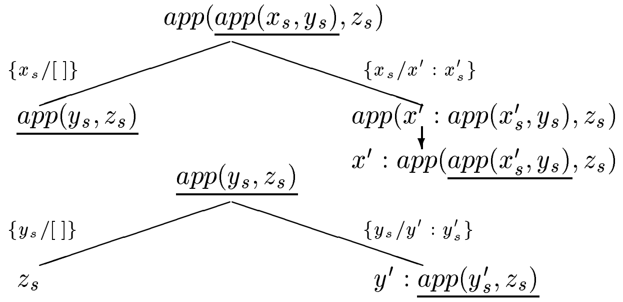

Figura 4.1: Arboles de *narrowing* perezoso normalizante para $app(app(x_s, y_s), z_s)$ y $app(x_s, y_s)$.

conocida como reducciones transitorias[3] *(*transient reductions*) [188], que puede incurrir en no terminación, a diferencia de lo que sucede con nuestro método. En [130], se ha propuesto la utilización de los llamados árboles característicos (*characteristic trees*) como una alternativa segura a las reducciones transitorias que sí asegura la terminación.*

Como se ha mostrado en [14], el efecto beneficioso de la normalización permite la eliminación de estructuras de datos intermedias aún en el caso de que no se adopte una estrategia de evaluación perezosa para obtener este tipo de optimizaciones, contrariamente a lo conjeturado por Sørensen et al. en [87, 189, 190].

Es conveniente notar que la normalización de las llamadas permite implementar una estrategia en la que se computa de un modo determinista tanto tiempo como sea posible, y que ésto es comparable a lo que Gallagher denomina desplegado *determinado*[4] (*determinate unfolding*) [75, 188], que evita la creación de puntos de elección superfluos para reglas alternativas introduciendo una especie de *look ahead* en los árboles de búsqueda, pero cuya terminación no está asegurada [126].

El Ejemplo 15 ilustra el hecho de que, en general, no está garantizado que el programa residual obtenido por el método de NPE sea CB. Esto impide que la estrategia de *narrowing* perezoso pueda emplearse para ejecutar el programa transformado. El Ejemplo 15 también muestra que el conjunto S de términos parcialmente evaluados, si bien garantiza la condición de cierre del programa transformado, en general no es independiente. Concretamente, el primer término solapa con uno de sus propios argumentos y el segundo de los

[3]Esta técnica consiste en realizar un plegado sobre los llamados "términos de tránsito", sin efectuar las debidas comprobaciones con los ancestros que impiden incurrir en riesgos de no terminación.

[4]Un árbol de búsqueda es *determinado* si cada nodo tiene como mucho un hijo. Una regla de desplegado *determinada*, es aquella que para un programa P y un objetivo G devuelve un árbol de búsqueda determinado. Habitualmente se relaja esta restricción para permitir un sólo paso de desplegado no determinado con un objetivo situado o bien en la raíz, o en cualquier punto del árbol, o en el "fondo".

términos solapa con el primero. El problema es que, en el Ejemplo 15, se emplea un mismo símbolo de función para dos especializaciones diferentes de una definición (concretamente, para las llamadas $app(app(x_s, y_s), z_s)$, el "append doble", y $app(x_s, y_s)$, el "append simple"). Así pues, sería necesario algún tipo de transformación para garantizar que no hay interferencias entre los correspondientes conjuntos de reglas derivadas, como podría haberlas si la definición del "append doble" fuese empleada para reducir por *narrowing* la llamada anidada $app(\_,\_)$ dentro de un término $app(\underline{app(\_,\_),\_)}$ (y que, erroneamente, puede dar lugar a respuestas no constructoras).

4.2 Postproceso de Renombramiento.

En este apartado, se formaliza una fase de renombramiento que mientras preserva la semántica de respuestas computadas asegura que la estrategia de *narrowing* perezoso puede ejecutar un término (cerrado con respecto al conjunto de llamadas especializadas) en el programa residual y que no es posible confundir diferentes especializaciones de una misma función.

Definición 4.2.1 (Renombramiento Independiente)
Sea S un conjunto de términos. Definimos un renombramiento independiente *S' de S como sigue:*

$$S' = \{\langle s \mapsto s' \rangle \mid s \in S \ \wedge \ s' = f^s(x_1, \ldots, x_m)\}$$

donde $\{x_1, \ldots, x_m\}$ son las distintas variables que aparecen en el término s en el orden (textual) de su primera aparición, y los f^s's símbolos de función nuevos, que son diferentes de todos los que aparecen en $\mathcal{R}$ y S.

Nótese que la definición anterior utiliza una técnica de *filtrado de argumentos*[5], que "filtra" las constantes y símbolos de función y sólo mantiene las variables como argumentos. La técnica del filtrado de argumentos es capaz de mejorar la eficiencia del programa especializado al realizar una simplificación sintáctica de las estructuras que lo componen.

El postproceso de renombramiento puede definirse formalmente como sigue.

Definición 4.2.2 (Postproceso de Renombramiento)
Sea $\mathcal{R}$ un programa y S un conjunto de términos. Sea $\mathcal{R}'$ una evaluación parcial de $\mathcal{R}$ con respecto a S, y S' un renombramiento independiente de S. Definimos el renombramiento $\mathcal{R}'' = ppren_{lazy}(\mathcal{R}', S')$ de $\mathcal{R}'$ con respecto a S' como sigue:

$$ppren_{lazy}(\mathcal{R}', S') = \quad \bigcup_{\langle s \mapsto s' \rangle \in S'} \ \{r' \mid \begin{array}{l} (\theta(s) \to \rho) \in \mathcal{R}', \ y \\ r' \equiv (\theta(s') \to ren(\rho, S'))\} \end{array}$$

donde la función indeterminista $ren(o, S')$ se define inductivamente como sigue:
$ren(o, S') =$

[5] El filtrado se ha denominado "*pushing-down meta-arguments*" en [140] o "*PDMA*" en [162]. En la programación funcional se ha empleado el término "*arity raising*".

$$\begin{cases} x & \texttt{si } o \equiv x \in \mathcal{X} \\ c(ren(t_1, S'), \ldots, ren(t_n, S')) & \texttt{si } o \equiv c(t_1, \ldots, t_n), c \in \mathcal{C}, n \geq 0 \\ p(ren(t_1, S'), ren(t_2, S')) & \texttt{si } o \equiv p(t_1, t_2) \text{ y } p \in \mathcal{P} \\ \gamma'(s') & \texttt{si } o = \gamma(s),\ \langle s \mapsto s' \rangle \in S',\ \texttt{y} \\ & \gamma' = \{x/ren(\gamma(x), S') \mid x \in \mathcal{D}om(\gamma)\}. \end{cases}$$

Siguiendo la terminología de Benkerimi y Hill en [34], en ocasiones, denominaremos a las expresiones renombradas *formas filtradas*. Informalmente, la Definición 4.2.2 permite derivar procedimientos especializados para cada término de S, introduciendo sistemáticamente definiciones $s' \to s$ nuevas y realizando un proceso de plegado sobre cada llamada a función de $\mathcal{R}'$, en que se reemplaza el término original s (i.e., el cuerpo de la nueva "regla") por su correspondiente renombramiento en S' (i.e., por una llamada a la nueva función definida s'), para producir el programa renombrado $\mathcal{R}''$. La idea que hay detrás de esta transformación es que, cualquiera que sea el término t, S-cerrado, el algoritmo de *narrowing* perezoso debe computar las mismas respuestas para t en $\mathcal{R}$ que para el término que resulta de renombrar t (de acuerdo con S') en $\mathcal{R}''$. Nótese que con total seguridad el proceso de renombramiento termina, ya que las expresiones que manipulamos son finitas.

Las definiciones anteriores se ilustran mediante el siguiente ejemplo.

Ejemplo 16 *Consideremos de nuevo el programa* app *del Ejemplo 15. Un renombramiento independiente S' de $S = \{app(x_s, y_s), app(app(x_s, y_s), z_s)\}$ es:*

$$\begin{aligned} S' = \{ \quad & \langle app(x_s, y_s) \mapsto a_1(x_s, y_s) \rangle, \\ & \langle app(app(x_s, y_s), z_s) \mapsto a_2(x_s, y_s, z_s) \rangle \ \}. \end{aligned}$$

Partiendo del programa residual, la evaluación parcial $\mathcal{R}'$ de $\mathcal{R}$ con respecto a S, (mostrando las substituciones de las resultantes de forma explicita), es:

$$\begin{array}{rcl} \{x'_s/[\,]\}(app(app(x'_s, y_s), z_s))) & \to & app(y_s, z_s) \\ \{x'_s/x : x_s\}\}(app(app(x'_s, y_s), z_s))) & \to & x : app(app(x_s, y_s), z_s) \\ \{x'_s/[\,]\}(app(x'_s, z_s)) & \to & z_s \\ \{x'_s/y : y_s\}(app(x'_s, z_s)) & \to & y : app(y_s, z_s) \end{array}$$

trás aplicar la definición 4.2.2 se obtiene:

$$\begin{array}{rcl} \{x'_s/[\,]\}(a_2(x'_s, y_s, z_s)) & \to & ren(app(y_s, z_s), S')) \\ \{x'_s/x : x_s\}\}(a_2(x'_s, y_s, z_s)) & \to & ren(x : app(app(x_s, y_s), z_s), S') \\ \{x'_s/[\,]\}(a_1(x'_s, z_s)) & \to & ren(z_s, S') \\ \{x'_s/y : y_s\}(a_1(x'_s, z_s)) & \to & ren(y : app(y_s, z_s), S') \end{array}$$

donde es interesante observar que las substituciones de las lhs's no quedan afectadas por el proceso de renombramiento, debido a que son substituciones constructoras. Finalmente, el programa renombrado $\mathcal{R}''$ de $\mathcal{R}'$ con respecto a S' es:

$$\begin{array}{rcl} a_2([\,], y_s, z_s) & \to & a_1(y_s, z_s) \\ a_2(x : x_s, y_s, z_s) & \to & x : a_2(x_s, y_s, z_s) \\ a_1([\,], y_s) & \to & y_s \\ a_1(x : x_s, y_s) & \to & x : a_1(x_s, y_s) \end{array}$$

Es conveniente notar que, para un determinado renombramiento independiente S' de S, la forma filtrada del programa residual puede depender de la estrategia empleada para seleccionar el par de S' que se utiliza para renombrar un término o de S, ya que puede existir, en general, más de un término s en S' que cubre la llamada o (e.g., en el ejemplo anterior, podríamos tener que $ren(x : app(app(x_s, y_s), z_s), S') = x : a_1(a_1(x_s, y_s), z_s)$ si se selecciona el par $\langle app(x_s, y_s) \mapsto a_1(x_s, y_s)\rangle$ de S' en lugar del par $\langle app(app(x_s, y_s), z_s) \mapsto a_2(x_s, y_s, z_s)\rangle$, que fue el que se consideró.). Por consiguiente, la forma filtrada de un programa no es única y una elección inconveniente puede hacer que se pierda algún potencial especializador. Una posible heurística (entre otras) para eliminar este indeterminismo podría ser elegir para renombrar, de entre los posibles pares $\langle s \mapsto s'\rangle$ de S' cuyo término s cubre a o, el par $\langle s \mapsto s'\rangle$ para el que la substitución asociada γ', aquélla tal que $o = \gamma'(s)$, fuese más general o sintácticamente más "simple" en algún sentido que habría que precisar. En el próximo apartado estudiaremos más detenidamente el problema del indeterminismo de renombramiento y de la heurística que puede establecerse para su eliminación.

El postproceso de renombramiento que acabamos de introducir en este apartado puede verse como una extensión del definido en [34] para el caso de expresiones con funciones anidadas. Para expresiones que no contengan funciones anidadas, nuestro postproceso de renombramiento, esencialmente, se reduce a la transformación, más simple, de [34]. Un marco de evaluación parcial en el que también se utilizan técnicas de renombramiento es el de la deducción parcial conjuntiva [83, 128], donde pueden aparecer conjunciones de llamadas (átomos) en las cabezas de las cláusulas de los programas evaluados parcialmente, lo cual es algo comparable a la aparición de símbolos de función anidados en las lhs's de las resultantes. Por consiguiente, en la deducción parcial conjuntiva, como en nuestro marco, la utilización de técnicas de renombramiento es obligatoria para poder derivar un programa ejecutable a partir del programa especializado intermedio. Existen dos diferencias entre nuestro postproceso de renombramiento y el introducido en [128]. En primer lugar, su definición describe unas transformaciones de renombramiento que pueden *borrar* variables dentro de las llamadas renombradas. Como se ha advertido en [128], esta aproximación puede producir transformaciones incorrectas. La segunda diferencia se relaciona con el indeterminismo a la hora de renombrar los cuerpos de las cláusulas. La transformación de renombramiento de [128] es semejante a la nuestra, pero introduce dos fuentes de indeterminismo: i) la primera concierne a la forma en la que se agrupan las conjunciones de los cuerpos de las cláusulas para ser renombradas; ii) la segunda, de forma similar a nuestro caso, surge cuando hay diferentes formas de renombrar las conjunciones de los cuerpos una vez que se ha elegido un renombramiento independiente. En [128] se introduce la idea de usar una función de partición de las conjunciones que decide qué partes (de los cuerpos) deberán renombrarse por separado y cuáles no. Si bien esto puede resolver el indeterminismo del primer tipo, como los propios autores reconocen, no soluciona el indeterminismo del segundo tipo, puesto que una partición particular todavía puede quedar cubierta con distintas opciones de renombramiento.

4.2.1 Indeterminismo del Renombramiento.

El indeterminismo en la Definición 4.2.2 de postproceso de renombramiento está inducido por el de la Definición 3.3.3 de condición de cierre, ya que el postproceso renombra los subtérminos de una expresión en el mismo orden en el que se probó que eran cerrados, de acuerdo a la Definición 3.3.3. Así pues, cuando el conjunto S de llamadas evaluadas parcialmente puede cubrir un término t de diferentes formas, dicho término admitirá también diferentes renombramientos. Para poner en evidencia este hecho y establecer la relación existente entre las posiciones del término original y las posiciones del término renombrado, conviene formalizar las nociones de conjunto de *cubrimientos* y de conjunto de *posiciones de cierre* (o simplemente conjunto de *cierre*, para abreviar) de un término t con respecto a un conjunto S.

Definición 4.2.3 (Conjunto de Cubrimientos y Posiciones de Cierre) *Sea S un conjunto finito de términos y t un término S-cerrado. Definimos el* conjunto de cubrimientos *de t con respecto a S como sigue:*

$$CSet(S,t) = \{O \mid O \in c\_set(S,t),\ \langle u.0, fail\rangle \notin O,\ u \in I\!N^*\}$$

donde la función auxiliar c_set, empleada para computar cada conjunto de cierre *O, se define inductivamente como: $c\_set(S,t) \ni$*

$$\begin{cases} \emptyset & \texttt{si } t \in \mathcal{X} \texttt{ o } t \equiv c \in \mathcal{C}; \\ \bigcup_{i=1}^{n}\{\langle i.p, s\rangle \mid \langle p, s\rangle \in c\_set(S,t_i)\} & \texttt{si } t \equiv c(t_1,\ldots,t_n),\ c \in \mathcal{C}; \\ \bigcup_{i=1}^{2}\{\langle i.p, s\rangle \mid \langle p, s\rangle \in c\_set(S,t_i)\} & \texttt{si } t \equiv p(t_1,t_2),\ p \in \mathcal{P}; \\ \{\langle \Lambda, s\rangle\} \cup \{\langle u.p, s'\rangle \mid s_{|u} = x,\ \langle p, s'\rangle \in c\_set(S,\theta(x))\} & \texttt{si } t \equiv f(t_1,\ldots,t_n),\ f \in \mathcal{D}, \\ & x \in \mathcal{X} \texttt{ y } \exists s \in S.\ \theta(s) = t; \\ \{\langle 0, fail\rangle\} & \texttt{de otro modo.} \end{cases}$$

Nótese que las posiciones que terminan con la marca "0" identifican las situaciones en las que un subtérmino de t no es instancia de ningún término de S. Por lo tanto, un conjunto de posiciones que contiene un par de la forma "$\langle u.0, fail\rangle$" no se considera un conjunto de cierre.

Las posiciones de cierre de un término t con respecto a un conjunto S también se denominarán posiciones S-cerradas. Nótese que la definición anterior ignora las posiciones etiquetadas por variables y símbolos constructores o primitivos, si bien éstas se consideran cerradas trivialmente. Hablando de manera informal, decimos que cada elemento del conjunto de cubrimientos es, a su vez, un conjunto de posiciones de cierre, formado por las posiciones (anidadas) de t que se consideran cuando se usa inductivamente la Definición 3.3.3 para comprobar si t es S-cerrado. Cada uno de los elementos del conjunto de cubrimientos se obtiene ensayando todos los modos posibles de hacer que el término t sea S-cerrado, i.e., explorando todas las posibilidades de cubrir t empleando los elementos de

S. Ahora, el hecho de que un término t pueda probarse S-cerrado de diferentes formas se manifiesta en que la función indeterminista $CSet$ computa varios conjuntos de cierre para el término considerado. El siguiente ejemplo ilustra este punto.

Ejemplo 17
Sea $S = \{f(X), g(X), f(g(X))\}$ y considérese el término $t = f(g(0))$. Entonces, hay dos posibilidades de verificar que t es S-cerrado:

1. *Si se tiene en cuenta el término $f(g(x)) \in S$, entonces $t = \{X/0\}(f(g(X)))$ y, trivialmente, se cumple que $closed(S, \{0\})$ es cierto. Por lo tanto, el conjunto de posiciones de cierre asociado es $O_1 = \{\langle \Lambda, f(g(X)) \rangle\}$.*

2. *Si por contra tomamos el término $f(X) \in S$, entonces $t = \{X/g(0)\}(f(X))$ y se puede verificar que $closed(S, \{g(0)\})$, ya que $g(0) = \{X/0\}(g(X))$, con $g(X) \in S$ y $closed(S, \{0\})$ por definición. En este caso, el conjunto de cierre asociado es $O_2 = \{\langle \Lambda, f(X) \rangle, \langle 1, g(X) \rangle\}$.*

Esto se traduce en que el conjunto de cubrimientos
$CSet(S, t) = \{\{\langle \Lambda, f(g(X)) \rangle\}, \{\langle \Lambda, f(X) \rangle, \langle 1, g(X) \rangle\}\}$.

Nótese que la Definición 4.2.3 también es útil para facilitar la implementación del test de cierre.

Dado un conjunto de llamadas S y un término t, cada uno de los conjuntos de cierre $O_i \in CSet(S, t)$ refleja una posibilidad de renombramiento para t. Por otra parte, la idea de almacenar en cada conjunto de cierre, junto a la posición de cierre p considerada, el término $s \in S$ que la hace cerrada, facilita la tarea del renombramiento. Estos dos aspectos se ponen de manifiesto en el siguiente ejemplo.

Ejemplo 18
Consideremos de nuevo el conjunto S y el término t del Ejemplo 17 con el renombramiento independiente $S' = \{\langle f(X) \mapsto f'(X) \rangle, \langle g(X) \mapsto g'(X) \rangle, \langle f(g(X)) \mapsto h(X) \rangle\}$. Entonces, según se emplee el conjunto de cierre O_1 o el O_2 pueden obtenerse, respectivamente, los siguientes renombramientos para t:

1. *$t_1'' = \{X/0\}(h(X)) = h(0)$, y*

2. *$t_2'' = (\{Y/0\} \circ \{X/(g'(Y))\})(f'(X)) = f'(g'(0))$*

El ejemplo anterior, también pone de manifiesto que el renombramiento más adecuado (intuitivamente, el renombramiento esperado para el término t con respecto a S'), coincide con el que se obtiene al emplear el conjunto de cierre con menor cardinalidad. Esta heurística, precisa y hace operativa la intuición, anteriormente apuntada, de renombrar seleccionando aquellos conjuntos de cierre cuyos pares generan las substituciones más "simples". En la implementación del prototipo de evaluador parcial Indy [13, 4], se ha definido una heurística para eliminar el indeterminismo del renombramiento basada en el concepto de conjunto de cubrimientos. La estrategia de renombramiento de INDY procede del siguiente modo: se computa el conjunto de cubrimientos $CSet(S,t)$; después se selecciona un conjunto de cierre con un número mínimo de pares[6]; finalmente, el renombramiento de t se efectúa siguiendo la partición (en subtérminos) inducida por el conjunto de cierre seleccionado.

Es de destacar que la causa última del indeterminismo es la falta de independencia del conjunto de llamadas especializadas S. Así pues, es de esperar que el indeterminismo desaparezca cuando dicho conjunto resulte ser independiente. El siguiente lema formaliza esta intuición.

Lema 4.2.4 *[202] Si S es un conjunto de términos independientes, entonces el conjunto $CSet(S,t)$ está formado por un único conjunto de cierre.*

En adelante, mientras no se especifique otra cosa, supondremos que los términos (y por consiguiente los programas) S-cerrados sólo admiten una forma de ser renombrados, bien porque S resulte ser un conjunto independiente o bien porque sea posible definir una heurística que permita, para un término t, seleccionar de forma determinista un conjunto de cierre $O \in CSet(S,t)$ como base para el renombramiento.

4.2.2 Corrección de la Transformación de Renombramiento.

Al tratar el problema de la corrección de la transformación de renombramiento es necesario tener en cuenta dos aspectos: por un lado, queremos que se alcancen las condiciones de cierre e independencia deseadas; por otro, es preciso que el programa especializado intermedio y el renombrado computen los mismos resultados con las mismas respuestas.

La siguiente proposición asegura que la Definición 4.2.2 de postproceso de renombramiento permite que se alcance la independencia de las llamadas especializadas y la condición de cierre para el programa y los términos renombrados. También asegura que el programa renombrado resultante es CB y lineal por la izquierda.

Proposición 4.2.5 (Independencia) *Sea $\mathcal{R}$ un TRS CB y ortogonal, S un conjunto finito de términos y g una expresión inicial (posiblemente compleja) S-cerrada. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}'$ es S-cerrado. Sea S' un renombramiento independiente de S, de manera que $\mathcal{R}'' = ppren_{lazy}(\mathcal{R}', S')$ es la forma renombrada de $\mathcal{R}'$ con respeto a S' y $g'' = ren(g, S')$ el renombramiento de g con respecto a S'. Entonces,*

1. *$\mathcal{A} \equiv \{s' \mid \langle s \mapsto s' \rangle \in S'\}$ es independiente,*
2. *$\mathcal{R}''$ es un TRS CB y lineal por la izquierda, y*

[6]Esta heurística se ha probado que es adecuada en la mayoría de los casos. En el Ejemplo 18 se desecha el conjunto de cierre O_2, al que le corresponde la substitución $\{X/(g'(0)), Y/0\}$, seleccionando el conjunto de cierre O_1, al que le corresponde la substitución $\{X/0\}$ sintácticamente más "simple".

3. $\mathcal{R}''_{calls} \cup g''_{calls}$ es $\mathcal{A}$-cerrado.

Prueba. Pasamos a probar cada una de las afirmaciones del teorema:

1. Por la Definición 4.2.1, para cada $\langle s_i \mapsto s'_i \rangle \in S'$, $s'_i = f^i(x_1, \ldots, x_m)$, donde f^i es un símbolo de función nuevo y $\{x_1, \ldots, x_m\}$ son las distintas variables de s_i tomadas en el orden de su primera aparición. Entonces, $\mathcal{A}$ es independiente por construcción.

2. Ahora se prueba que $\mathcal{R}''$ es un TRS CB y lineal por la izquierda. Sea $D_s \equiv (s \overset{[u_1,R_1,\sigma_1]}{\leadsto_{LN}} s_1 \overset{[u_2,R_2,\sigma_2]}{\leadsto_{LN}} \ldots \overset{[u_n,R_n,\sigma_n]}{\leadsto_{LN}} s_n)$ una derivación finita de *narrowing* perezoso para s en $\mathcal{R}$ que produce la resultante $R' \equiv (\sigma(s) \to s_n)$, donde $\sigma = \sigma_n \circ \ldots \circ \sigma_1$. Por la Definición 4.2.2, el renombramiento de la resultante R' es la regla $R'' \equiv (\sigma(s') \to ren(s_n, S'))$, cuya lhs es el término $\sigma(s') \equiv \sigma(f^s(x_1, \ldots, x_m))$, donde $f^s(x_1, \ldots, x_m)$ es lineal. Por otra parte, por el Lema 2.9.17, para cada i, $\sigma(x_i)$ es un término constructor lineal y, para cada $j \neq i$, $\mathcal{V}ar(\sigma(x_i)) \cap \mathcal{V}ar(\sigma(x_j)) = \emptyset$. Por lo tanto, la lhs de la regla R'' es un patrón lineal, tal y como se queria demostrar.

3. Es suficiente demostrar que, para cada término S-cerrado t de $\mathcal{R}'_{calls} \cup g_{calls}$, el término renombrado $t'' = ren(t, S')$, $t'' \in \mathcal{R}''_{calls} \cup g''_{calls}$, es $\mathcal{A}$-cerrado. Esta propiedad se demuestra por inducción estructural sobre t.

   - $t \equiv x \in \mathcal{X}$. Inmediato, por la Definición 3.3.3.
   - $t \equiv c(t_1, \ldots, t_n)$. Ya que t es S-cerrado, por la Definición 3.3.3, cada t_i es S-cerrado. Entonces, por hipótesis de inducción, cada $t''_i = ren(t_i, S')$ es $\mathcal{A}$-cerrado, y la propiedad se sigue de forma inmediata.
   - $t \equiv f(t_1, \ldots, t_n)$. Ya que t es S-cerrado, por la Definición 3.3.3, $t \equiv \gamma(s)$, con $s \in S$, y $\mathcal{R}an(\gamma)$ es S-cerrado. Por consiguiente, por hipótesis de inducción, para cada $l \in \mathcal{R}an(\gamma)$, $ren(l, S')$ es $\mathcal{A}$-cerrado. Por otro lado, tenemos que $t'' = ren(t, S') = \gamma'(s')$, con $\langle s \mapsto s' \rangle \in S'$ y $\gamma' = \{x/ren(\gamma(x), S') \mid x \in \mathcal{D}om(\gamma)\}$. De esto se sigue que el conjunto $\mathcal{R}an(\gamma')$ es $\mathcal{A}$-cerrado. Por tanto, existe un $s' \in \mathcal{A}$ tal que $((\gamma'(s') = t'') \wedge closed(\mathcal{A}, \mathcal{R}an(\gamma')))$, lo cual concluye la prueba.

□

A partir de la Proposición 4.2.5 y del Lema 4.2.4, puede establecerse de forma inmediata el siguiente resultado sobre los términos renombrados que afirma que éstos son cerrados de forma única con respecto al conjunto $\mathcal{A}$.

Corolario 4.2.6 *Sea S un conjunto de llamadas especializadas, t un término S-cerrado y S' un renombramiento independiente de S. Sea $\mathcal{A} \equiv \{s' \mid \langle s \mapsto s' \rangle \in S'\}$. Entonces, dado el término $t'' = ren(t, S')$, el conjunto $CSet(\mathcal{A}, t'')$ está formado por un único conjunto de cierre.*

Es fácil comprobar que las posiciones del término renombrado $t'' = ren(t, S')$ son posiciones de cierre con respecto al conjunto $\mathcal{A}$.

Lema 4.2.7 *Sea S un conjunto de llamadas especializadas, t un término S-cerrado y S' un renombramiento independiente de S. Sea $\mathcal{A} \equiv \{s' \mid \langle s \mapsto s'\rangle \in S'\}$. Sea $t'' = ren(t, S')$ un renombramiento de t. Para cada una de sus posiciones $p'' \in \mathcal{FP}os(t'')$, con $\mathcal{H}ead(t''|_{p''}) \in \mathcal{D}$, se cumple que p'' es una posición $\mathcal{A}$-cerrada.*

Prueba. Se prueba esta propiedad por inducción estructural sobre t.

- $t \equiv x \in \mathcal{X}$ (o $t \equiv c \in \mathcal{C}$). Inmediato, ya que en este caso $t'' \equiv x \in \mathcal{X}$ (o $t'' \equiv c \in \mathcal{C}$) y, por la Definición 4.2.3, el enunciado se cumple por vacuidad.

- $t \equiv c(t_1, \ldots, t_n)$. Dado que t es S-cerrado, por la Definición 3.3.3 se cumple que cada uno de los subtérminos t_i es S-cerrado. Ahora $t'' \equiv c(ren(t_1, S'), \ldots, ren(t_n, S'))$, y la posición Λ se ignora al ser $\mathcal{H}ead(t'') \equiv c \notin \mathcal{D}$ mientras que por hipótesis de inducción el resto de las posiciones p'' de los subtérminos $t''_i = ren(t_i, S')$, con $\mathcal{H}ead(t''_i|_{p''}) \in \mathcal{D}$, son $\mathcal{A}$-cerradas.

- $t \equiv f(t_1, \ldots, t_n)$. Ya que t es S-cerrado, por la Definición 3.3.3, $t \equiv \gamma(s)$, con $s \in S$, y $\mathcal{R}an(\gamma)$ es S-cerrado. Ahora, por la Definición 4.2.1 $\langle s \mapsto s'\rangle \in S'$, con $s' = f^s(x_1, \ldots, x_k)$, donde f^s es un símbolo de función nuevo y $\{x_1, \ldots, x_k\}$ son las distintas variables de s tomadas en el orden de su primera aparición. Por consiguiente, $t'' = \gamma'(f^s(x_1, \ldots, x_k))$, con $\gamma' = \{x/ren(\gamma(x), S') \mid x \in \mathcal{D}om(\gamma)\}$. Ahora, Λ es una posición $\mathcal{A}$-cerrada, por la Definición 4.2.3 y por hipótesis de inducción, cada una de las posiciones p'' de los subtérminos $ren(\gamma(x_i), S')$, con $i = 1, \ldots, k$, para las que se cumple que $\mathcal{H}ead(t''_i|_{p''}) \in \mathcal{D}$, son $\mathcal{A}$-cerradas.

□

También es fácil probar el siguiente lema, que establece la correspondencia precisa entre la condición de cierre de una expresión t y la de su forma renombrada.

Lema 4.2.8 *Sea S un conjunto de llamadas especializadas, t un término y S' un renombramiento independiente de S. Sea $\mathcal{A} \equiv \{s' \mid \langle s \mapsto s'\rangle \in S'\}$. Entonces, $t'' = ren(t, S')$ es $\mathcal{A}$-cerrado si y sólo si t es S-cerrado.*

Prueba. Completamente análoga a la del punto (3) de la Proposición 4.2.5. □

En lo que sigue se aborda el segundo de los aspectos sobre la corrección de la transformación de renombramiento: la equivalencia del programa residual y el renombrado con respecto a las respuestas computadas por el *narrowing* perezoso.

Primero conviene darse cuenta del siguiente hecho interesante, relativo a las posiciones de un término sobre las que se pueden dar pasos de *narrowing* usando las reglas del programa especializado. En general, dichas posiciones son posiciones de cierre. El orden en el que se utilizan las posiciones cerradas viene dictado, como es natural, por la estrategia de *narrowing*. En particular las posiciones perezosas más externas y aquéllas intérnas que contribuyen a la realización de un cómputo más extérno con posterioridad deben ser posiciones de cierre. Esta intuición se formaliza mediante el siguiente lema.

Lema 4.2.9 *Sea S un conjunto de llamadas especializadas, t un término S-cerrado y S' un renombramiento independiente de S. Sea $\mathcal{A} \equiv \{s' \mid \langle s \mapsto s' \rangle \in S'\}$. Sea $t'' = ren(t, S')$ un renombramiento de t. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S y sea $\mathcal{R}'' = ppren_{lazy}(\mathcal{R}', S')$. Entonces,*

1. *Si $p \in \mathcal{FP}os(t)$ es una posición sobre la que se puede dar un paso de* narrowing *(ordinario) en $\mathcal{R}'$, entonces p es una posición S-cerrada.*
2. *Si $p'' \in \mathcal{FP}os(t'')$ es una posición sobre la que se puede dar un paso de* narrowing *(ordinario) en $\mathcal{R}''$, entonces p'' es una posición $\mathcal{A}$-cerrada.*

Prueba.

1. Por definición de evaluación parcial, las reglas de $\mathcal{R}'$ son resultantes de la forma: $R' \equiv (\sigma(s) \to r)$, donde $s \in S$ y σ es una substitución (constructora, en el caso de una LN-PE). Si es posible dar un paso de *narrowing* sobre la posición p es porque existe una regla de $\mathcal{R}'$, dígase la resultante R', tal que:

$$mgu(\{t|_p = \sigma(s)\}) \not\equiv fail.$$

   Esto obliga a que $t|_p \equiv \theta(s)$, de otro modo $t|_p$ y $\sigma(s)$ no serían unificables. Ahora conviene distinguir los siguientes casos:

   - $p = \Lambda$. Entonces, por la Definición 4.2.3, $\langle p, s \rangle$ pertenece a cualquier conjunto de cierre del conjunto de cubrimientos $CSet(S, t)$.
   - $p \neq \Lambda$. Entonces, puesto que t es un término S-cerrado, $t \equiv \gamma(s')$, con $s' \in S$, y $closed(S, \mathcal{R}an(\gamma))$. Por consiguiente, habrá una ocurrencia interna p, con $t|_p \equiv \theta(s)$, tal que $\langle p, s \rangle$ pertenece a algún conjunto de cierre de $CSet(S, t)$. De otro modo $t|_p$, no sería unificable con la lhs $\sigma(s)$ de R', en contra de lo supuesto.

2. Inmediato, ya que, por el Lema 4.2.7, todas las posiciones de t'' encabezadas por un símbolo de función son $\mathcal{A}$-cerradas.

□

De todo lo anterior se desprende que si queremos estudiar la equivalencia de las derivaciones de *narrowing* a partir de un término t en el programa residual $\mathcal{R}'$ (resultado de la LN-PE de $\mathcal{R}$ con respecto a un conjunto de llamadas S) y las correspondientes derivaciones de *narrowing* a partir del término renombrado t'' en el programa renombrado $\mathcal{R}''$, es preciso establecer una relación entre las

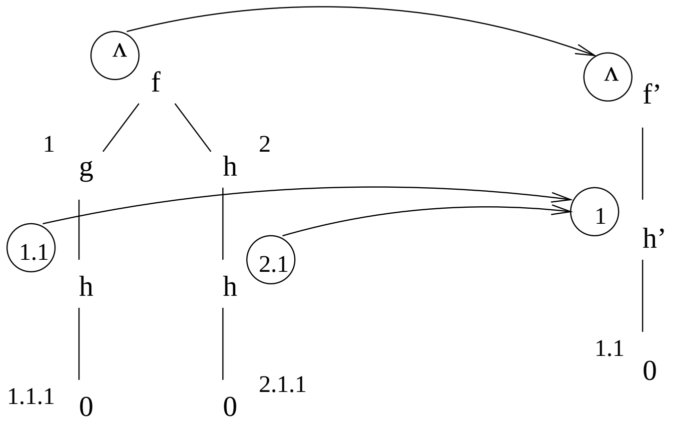

Figura 4.2: Correspondencia entre las posiciones de cierre de un término y su renombrado, en el Ejemplo 19.

posiciones de cierre del término t, $O = \{p \mid \langle p, s\rangle \in O \;\wedge\; O \in CSet(S, t)\}$, y las posiciones de cierre del término t'', $O'' = \{p'' \mid \langle p'', s'\rangle \in c\_set(\mathcal{A}, t'')\}$. Si el conjunto de llamadas especializadas S está formado por términos no lineales, entonces no existe una correspondencia *uno a uno* entre los elementos de O y los de O'', como muestra el siguiente ejemplo.

Ejemplo 19 *Sea el conjunto* $S = \{f(g(X), h(X)), g(X), h(X)\}$ *y el término* $t = f(g(h(0)), h(h(0)))$*. Es sencillo comprobar que* t *es cerrado de forma única respecto a* S*, siendo* $CSet(S, t) = \{\{\langle\Lambda, f(g(X), h(X))\rangle, \langle 1.1, h(X)\rangle, \langle 2.1, h(X)\rangle\}\}$*. Dado el renombramiento independiente*

$$S' = \{\langle f(g(X), h(X)) \mapsto f'(X)\rangle, \langle g(X) \mapsto g'(X)\rangle, \langle h(X) \mapsto h'(X)\rangle\},$$

el conjunto $CSet(S, t)$ *conduce al siguiente término renombrado:* $t'' = f'(h'(0))$*. Ahora, el término* t'' *tiene asociado un cubrimiento*

$$CSet(\mathcal{A}, t'') = \{\{\langle\Lambda, f'(X)\rangle, \langle 1, h'(X)\rangle\}\},$$

siendo $\mathcal{A} = \{f'(X), g'(X), h'(X)\}$*. Claramente, a las posiciones de cierre '1.1' y '2.1' del término* t *les corresponde la posición de cierre '1' en el término* t'' *(ver la Figura 4.2). Por lo tanto, si tuviésemos que definir una aplicación de las posiciones* S*-cerradas del término* t *en las posiciones* $\mathcal{A}$*-cerradas del término renombrado* t''*, dicha aplicación no sería inyectiva y, por lo tanto, su inversa no sería una función. Sin embargo, todavía es posible definir una relación entre las posiciones de* t *y sus correspondientes renombradas.*

Formalizamos la relación entre las posiciones de cierre de un término, (sobre las que se dan los pasos de *narrowing* en el programa residual) y las correspondientes posiciones del término renombrado t'' (sobre las que se dan los pasos de *narrowing* en el programa renombrado) mediante la siguiente definición.

Definición 4.2.10
Sea S un conjunto finito de llamadas, t un término S-cerrado y S' un renombramiento independiente de S. Caracterizamos la relación entre las posiciones de cierre del término t, $O = \{p \mid \langle p, s\rangle \in O \ \wedge \ O \in CSet(S,t)\}$, y las posiciones de cierre del término $t'' = ren(t, S')$, $O'' = \{p'' \mid \langle p'', s'\rangle \in c\_set(\mathcal{A}, t'')\}$, sobre las que se pueden dar pasos de narrowing*, como un subconjunto $n\_\mathcal{P}os(t, S')$ de $\wp(O) \times O''$ que se define inductivamente como sigue: $n\_\mathcal{P}os(t, S') =$*

$$\begin{cases} \emptyset & \texttt{si } t \in \mathcal{X} \ \ \texttt{o} \ \ t \equiv c \in \mathcal{C}; \\ \bigcup_{i=1}^{n} \{\langle i.\mathcal{U}_k, i.p_k\rangle\} & \texttt{si } t \equiv c(t_1, \ldots, t_n), \ c \in \mathcal{C}, \\ & \texttt{y } \langle \mathcal{U}_k, p_k\rangle \in n\_\mathcal{P}os(t_i, S'); \\ \bigcup_{i=1}^{2} \{\langle i.\mathcal{U}_k, i.p_k\rangle\} & \texttt{si } t \equiv p(t_1, t_2), \ c \in \mathcal{P}, \\ & \texttt{y } \langle \mathcal{U}_k, p_k\rangle \in n\_\mathcal{P}os(t_i, S'); \\ \{\langle\{\Lambda\}, \Lambda\rangle\} \cup \ (\bigcup_{i=1}^{n} \bigcup_{i=1}^{n_i} \{\langle \mathcal{U}_i.\mathcal{U}_k, i.p_k\rangle\}) & \texttt{si } t = \theta(s), \langle s \mapsto f^s(x_1, \ldots, x_n)\rangle \in S', \\ & \mathcal{U}_i = \{u_i \mid s|_{u_i} = x_i\}, \\ & \langle \mathcal{U}_k, p_k\rangle \in n\_\mathcal{P}os(\theta(x_i), S') \\ & \texttt{y } |n\_\mathcal{P}os(\theta(x_i), S')| = n_i. \end{cases}$$

Se ha hecho uso de la notación introducida en el Apartado 2.6.1, considerando que si U y W son conjuntos de ocurrencias y $W = \emptyset$ entonces $U.W = \emptyset$. El siguiente ejemplo ilustra el modo en el que opera esta definición.

Ejemplo 20 *Sea el conjunto $S = \{f(g(X), X), h(X)\}$ y el término $t = c(f(g(h(Y)), h(Y))), h(0))$, donde c es un constructor. Puede comprobarse que t es cerrado de forma única respecto a S, siendo*

$$CSet(S,t) = \{\{\langle 1, f(g(X), X)\rangle, \langle 1.1.1, h(X)\rangle, \langle 1.2, h(X)\rangle, \langle 2, h(X)\rangle\}\}.$$

También puede comprobarse que para el renombramiento independiente

$$S' = \{\langle f(g(X), X) \mapsto f'(X)\rangle, \langle h(X) \mapsto h'(X)\rangle\},$$

el conjunto $CSet(S,t)$ conduce al término renombrado $t'' = c(f'(h'(Y)), h'(0))$, cuyo cubrimiento es

$$CSet(\mathcal{A}, t'') = \{\{\langle 1, f'(X)\rangle, \langle 1.1, h'(X)\rangle \langle 2, h'(X)\rangle\}\},$$

siendo $\mathcal{A} = \{f'(X), h'(X)\}$. La relación entre las posiciones de cierre del término t y del renombrado t'' es:

$$n\_\mathcal{P}os(t, S') = \{\langle\{1\}, 1\rangle, \langle\{1.1.1, 1.2\}, 1.1\rangle, \langle\{2\}, 2\rangle\},$$

Debido a que al aplicar la Definición 4.2.10:

1. *$n\_\mathcal{P}os(f(g(h(Y)), h(Y))), S') = \{\langle\{\Lambda\}, \Lambda\rangle, \langle\{1.1, 2\}, 1\rangle\}$, ya que: $f(g(h(Y)), h(Y))) = \{X/h(Y)\}(f(g(X), X))$, $\langle f(g(X), X) \mapsto f'(X)\rangle \in S'$, $\mathcal{U}_i = \{1.1, 2\}$ y $n\_\mathcal{P}os(h(Y), S') = \{\langle\{\Lambda\}, \Lambda\rangle\}$;*

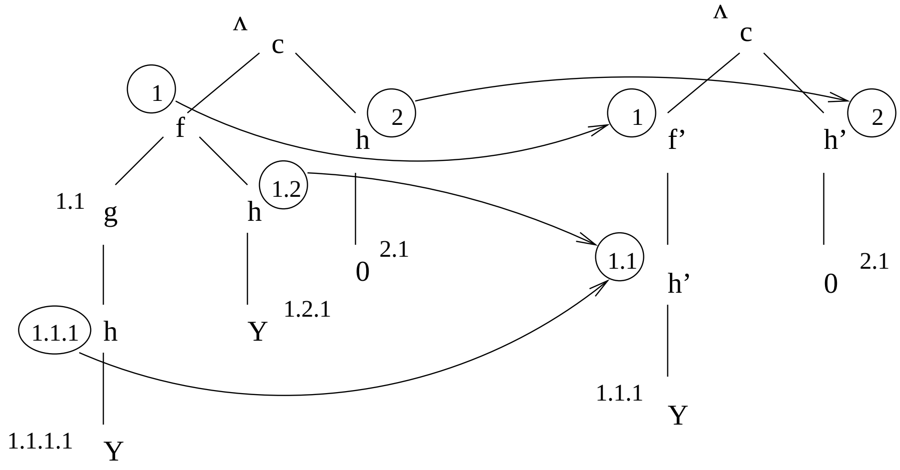

Figura 4.3: Correspondencia entre las posiciones de cierre de un término y su renombrado, en el Ejemplo 20.

2. $n\_\mathcal{P}os(h(0), S') = \{\langle\{\Lambda\}, \Lambda\rangle\}$, *ya que:*

$$h(0) = \{X/0\}(h(X)),\ \langle h(X) \mapsto h'(X)\rangle \in S',\ \mathcal{U}_i = \{1\} \text{ y } n\_\mathcal{P}os(0, S') = \emptyset.$$

La Figura 4.3 visualiza la relación $n\_\mathcal{P}os(t, S')$.

Como se ha podido apreciar, cuando se realiza una LN-PE de un programa $\mathcal{R}$ con respecto a un conjunto S, se obtiene un programa $\mathcal{R}'$ que en general no es CB y tampoco lineal por la izquierda (si S no es lineal). Esto impide que la estrategia de *narrowing* perezoso λ_{lazy} pueda aplicarse para ejecutar el programa $\mathcal{R}'$. Naturalmente, todavía es posible dar pasos de *narrowing* sin restricciones con el programa $\mathcal{R}'$. Hemos demostrado que un proceso de renombramiento como el de la Definición 4.2.2 y la Proposición 4.2.5 aseguran que el programa renombrado $\mathcal{R}''$ podrá ejecutarse empleando la estrategia λ_{lazy}. En lo que sigue establecemos un paralelismo entre las derivaciones de *narrowing* (perezoso) en $\mathcal{R}''$ y las derivaciones de *narrowing* (sin restricciones) en $\mathcal{R}'$ formalizando el concepto de paso de *narrowing cuasiperezoso* en $\mathcal{R}'$. Intuitivamente, lo que buscamos con este concepto es definir un tipo de derivaciones en $\mathcal{R}'$ tal que, cuando se renombran los términos a evaluar y las reglas del programa, se obtiene una derivación de *narrowing* perezoso en $\mathcal{R}''$ que computa el mismo resultado con la misma respuesta; esto nos permitirá demostrar la corrección del renombramiento y de la transformación completa.

La siguiente definición formaliza la noción de *multipaso de narrowing* [23].

Definición 4.2.11 (Multipaso de Narrowing)

Sea $t \overset{[u_i, l_i \to r_i, \sigma_i]}{\leadsto} t_i$, *para* i *perteneciente a algún conjunto de índices* $I = \{1, \ldots, n\}$, *un paso de* narrowing *tal que, para cualquier* i *y* j *en* I, *con* $i \neq j$, $u_i \parallel u_j$ *y* $\sigma_i \circ \sigma_j = \sigma_j \circ \sigma_i$. *Decimos que el término* t *es reducible por* narrowing *a* t' *en un* multipaso de narrowing, *denotado* $t \overset{\langle u_i, l_i \to r_i, \sigma_i\rangle_{i \in I}}{\leadsto} t'$, *si y* *sólo si* $t' = (((t[r_1]_{u_1})[r_2]_{u_2})\ldots[r_n]_{u_n})\sigma$, *donde* $\sigma = \sigma_n \circ \ldots \circ \sigma_1$ *(el orden es irrelevante).*

Siguiendo a [23], cuando se quiera enfatizar la diferencia entre un paso de *narrowing* como el definido en el Apartado 2.9.2 y un multipaso, denominaremos al primero *elemental*. De otro modo, identificamos un paso elemental con un multipaso en el cual el conjunto de posiciones que se reduce por *narrowing* tiene un sólo elemento. Un multipaso de *narrowing* puede entenderse como un conjunto de pasos de *narrowing* elemental que se realizan en paralelo. De hecho, las condiciones impuestas sobre las posiciones y las substituciones de cada paso elemental para definir un multipaso implican que, en un multipaso, el orden en el que se componen las substituciones y se reducen las posiciones por *narrowing* es irrelevante.

Definición 4.2.12 (Paso de Narrowing Cuasiperezoso)
Sea S un conjunto de llamadas especializadas, t un término S-cerrado y S' un renombramiento independiente de S. Sea $t'' = ren(t, S')$ un renombramiento de t. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S y sea $\mathcal{R}'' = ppren_{lazy}(\mathcal{R}', S')$. Sea u'' una posición perezosa de t'' en $\mathcal{R}''$ y $\langle\{u_1, \ldots, u_n\}, u''\rangle \in n\_\mathcal{P}os(t, S')$, definimos un paso de narrowing cuasiperezoso *en $\mathcal{R}'$ como un paso multipaso dado sobre las posiciones $\{u_1, \ldots, u_n\}$.*

Una derivación de narrowing cuasiperezoso *en $\mathcal{R}'$ es una derivación de* narrowing *en $\mathcal{R}'$ cuyos pasos son todos cuasiperezosos.*

Después de este trabajo previo estamos ya en disposición de establecer la equivalencia entre las derivaciones de *narrowing* perezoso en $\mathcal{R}''$ y las derivaciones de *narrowing* cuasiperezoso en $\mathcal{R}'$. Necesitamos dos lemas auxiliares.

El siguiente lema establece una propiedad interesante de los términos cerrados con respecto a un conjunto de llamadas S y sus correspondientes formas renombradas.

Lema 4.2.13 (Lema del Renombramiento) *Sea S un conjunto finito de términos y s, t dos términos S-cerrados. Sea S' un renombramiento independiente de S. Sean $s'' = ren(s, S')$ y $t'' = ren(t, S')$. Se cumple que $\theta'' = mgu(\{s'' = t''\}) \not\equiv fail$ si y sólo si $\theta = mgu(\{s = t\}) \not\equiv fail$; donde $\theta'' = \{(x/ren(\theta(x), S')) \mid x \in Dom(\theta)\}$.*

Prueba. Comenzamos con la prueba de la condición necesaria. La idea es definir un procedimiento constructivo consistente en lo siguiente: dado que θ'' es el mgu de s'' y t'', partimos de la ecuación $\theta''(s'') = \theta''(t'')$ y sometemos ésta a un proceso de "renombramiento inverso" de los términos que la componen, hasta llegar a la ecuación $\theta(s) = \theta(t)$ donde θ es un unificador de s y t; finalmente, probamos que θ es un mgu y que θ y θ'' cumplen la relación que establece el enunciado de este lema.

El procedimiento de renombramiento inverso se concreta en dos fases:

1. Renombramiento inverso de los términos s'' y t''.
   Esto conduce de forma unívoca a los términos s y t. Dado que el renombramiento no afecta a las variables de un término, se cumple que $\mathcal{V}ar(s'') = \mathcal{V}ar(s)$ y $\mathcal{V}ar(t'') = \mathcal{V}ar(t)$. Así pues, $\theta''(s) = \theta''(t)$.

2. Renombramiento inverso de los términos de $\mathcal{R}an(\theta'')$.
   Ya que los términos s y t son S-cerrados, por el Corolario 4.2.6 y el Lema 4.2.7, s'' y t'' son $\mathcal{A}$-cerrados de forma única y todas sus posiciones son de cierre. Además, dado que θ'' es el mgu de s'' y t'', los componentes de $\mathcal{R}an(\theta'')$ son subtérminos de s'' y t'' y por consiguiente, $\mathcal{A}$-cerrados. Por tanto, para cada $t_i'' \in \mathcal{R}an(\theta'')$, existe un t_i tal que $t_i'' = ren(t_i, S')$. También, dado que el renombramiento no afecta a las variables, se cumple que $\mathcal{D}om(\theta'') = \mathcal{D}om(\theta)$. Así pues,

$$\begin{aligned}\theta'' &= \{x_i/t_i'' \mid x_i \in \mathcal{D}om(\theta'')\}\\ &= \{x_i/ren(t_i, S') \mid x_i \in \mathcal{D}om(\theta)\}.\end{aligned}$$

   Si ahora aplicamos el renombramiento inverso a los términos de $\mathcal{R}an(\theta'')$, obtenemos la ecuación $\theta(s) = \theta(t)$, con $\theta = \{x_i/t_i \mid x_i \in \mathcal{D}om(\theta)\}$, lo que indica que los términos s y t unifican con unificador θ. Dado que los términos s y t unifican, por el teorema de unificación, existe un $mgu(\{s = t\}) \not\equiv fail$, que es único (salvo renombramiento de variables). Además, dicho mgu debe coincidir con θ, ya que de otro modo θ'' no sería un mgu de s'' y t''.

La prueba de la condición suficiente se realiza mediante un razonamiento similar al anterior y haciendo uso directo de la Definición 4.2.2. □

El lema siguiente clarifica la relación existente entre los pasos de *narrowing* perezoso en $\mathcal{R}''$ y los pasos de *narrowing* cuasiperezoso en $\mathcal{R}'$.

Lema 4.2.14 *Sea $\mathcal{R}$ un TRS ortogonal y CB. Sea S un conjunto finito de términos y t un término S-cerrado. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}'$ es S-cerrado. Sea S' un renombramiento independiente de S, $\mathcal{R}'' = ppren_{lazy}(\mathcal{R}', S')$ y $t'' = ren(t, S')$. Sea $R'' \equiv (\sigma(s') \to r'') \in \mathcal{R}''$ el renombramiento de la correspondiente resultante $R' \equiv (\sigma(s) \to r) \in \mathcal{R}'$, donde $r'' = ren(r, S')$ y $\langle s \mapsto s' \rangle \in S'$. Entonces, existe un paso de* narrowing *perezoso $t'' \overset{[u'',R'',\theta'']}{\leadsto}_{LN} t_1''$, en $\mathcal{R}''$, si y sólo si existe un paso de* narrowing *cuasiperezoso $t \overset{\langle u_i,R',\theta\rangle_{i\in I}}{\leadsto} t_1$ en $\mathcal{R}'$, donde $\langle U, u'' \rangle \in n\_\mathcal{P}os(t, S')$, $U = \{u_i \mid i \in I\}$ e $I = \{1, \ldots, n\}$. Más aún, se cumple que $t_1'' = ren(t_1, S')$ y $\theta'' = \theta[\mathcal{V}ar(t)]$.*

Prueba. Primero se debe notar que $|U| = 1$ solamente si S es un conjunto de términos lineales. En el caso general, $|U| > 1$ y, debido a que esa cardinalidad mayor que la unidad proviene de la existencia en S de terminos no lineales (i.e., con multiples apariciones de la misma variable), se cumple que $t|_{u_i} \equiv t|_{u_j}$ para cada $i \neq j$.

Si es posible dar un paso de *narrowing* perezoso $t'' \overset{[u'',R'',\theta'']}{\leadsto}_{LN} t_1''$ en $\mathcal{R}''$, entonces $\theta'' = mgu(\{t''|_{u''} = \sigma(s')\}) \not\equiv fail$ y, ya que $t''|_{u''} = ren(t|_{u_i}, S')$ y $\sigma(s') = ren(\sigma(s), S')$, el Lema 4.2.13 es aplicable y podemos decir que $\theta = mgu(\{t|_{u_i} = \sigma(s)\}) \not\equiv fail$ y $\theta'' = \{(x/ren(\theta(x), S')) \mid x \in Dom(\theta)\}$. Por consiguiente, existe el paso de *narrowing* $t \overset{[u_i,R',\theta]}{\leadsto} \theta(t[r]_{u_i})$. Ahora, por la

Definición 4.2.11 y el hecho de que u_i, u_j, con $i \neq j$, son posiciones disjuntas, es posible dar un multipaso de *narrowing* cuasiperezoso de t a t_1.

Además, claramente, $t_1'' = ren(t_1, S')$ y, ya que $\theta''_{\lceil \mathcal{V}ar(t)}$ es una substitución constructora, entonces $\theta'' = \theta[\mathcal{V}ar(t)]$.

La prueba de la condición suficiente se realiza mediante un razonamiento similar al anterior y aplicando el lema del renombramiento (Lema 4.2.13) en el sentido inverso. □

Finalmente, la siguiente proposición establece la corrección de la transformación de renombramiento.

Proposición 4.2.15 *Sea $\mathcal{R}$ un TRS ortogonal y CB, S un conjunto finito de términos y t un término S-cerrado. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}'$ es S-cerrado. Sea S' un renombramiento independiente de S, $\mathcal{R}'' = ppren_{lazy}(\mathcal{R}', S')$ y $t'' = ren(t, S')$. Entonces, existe una derivación de* narrowing *perezoso $t'' \overset{\theta''}{\rightsquigarrow}_{LN}^{*} d$ en $\mathcal{R}''$ si y sólo si existe una derivación de* narrowing *cuasiperezoso $t \overset{\theta}{\rightsquigarrow}^{*} d$ en $\mathcal{R}'$ que emplean las reglas correspondientes sobre las posiciones correspondientes y donde $\theta'' = \theta[\mathcal{V}ar(g)]$.*

Prueba. Trivial, por inducción sobre la longitud de las derivaciones, utilizando el Lema 4.2.14. □

4.3 Corrección de la Evaluación Parcial Dirigida por Narrowing Perezoso.

Utilizando los resultados del apartado anterior, podemos establecer finalmente la corrección (débil) de la transformación LN-PE con el postproceso de renombramiento.

Teorema 4.3.1 (Corrección Débil) *Sea $\mathcal{R}$ un TRS ortogonal y CB, e una ecuación y S un conjunto finito de términos. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}'$ y e son S-cerrados. Sea S' un renombramiento independiente de S, y e'' (resp. $\mathcal{R}''$) un renombramiento de e (resp. $\mathcal{R}'$) empleando S'. Si el procedimiento de* narrowing *perezoso computa la respuesta θ'' para e'' en $\mathcal{R}''$, entonces computa una respuesta θ para e en $\mathcal{R}$, donde $\theta \leq \theta''[\mathcal{V}ar(e)]$.*

Prueba. Con el fin de probar este teorema, primero realizamos las siguientes consideraciones acerca de las substituciones θ, θ'' y el programa $\mathcal{R}$. Dado que (por la Proposición 2.9.17) los términos del rango de las substituciones $\theta_{\lceil \mathcal{V}ar(e)}$ y $\theta''_{\lceil \mathcal{V}ar(e)}$ son términos constructores lineales, se cumple que:

1. El postproceso de renombramiento no afecta a $\theta_{\lceil \mathcal{V}ar(e)}$.
2. Las substituciones $\theta_{\lceil \mathcal{V}ar(e)}$ y $\theta''_{\lceil \mathcal{V}ar(e)}$ son substituciones normalizadas.

Por otra parte, dado que $\mathcal{R}$ es ortogonal y CB, es confluente. La restricción a respuestas normalizadas y la confluencia de $\mathcal{R}$ son los requisitos necesarios para poder aplicar el resultado de corrección débil de la NPE utilizando *narrowing* sin restricciones (Teorema 3.3.6). Hechas estas aclaraciones, estamos en condiciones de iniciar la prueba.

Si existe una derivación de *narrowing* perezoso en $\mathcal{R}''$

$$D_1 \equiv (e'' \overset{\theta''}{\leadsto}_{LN}{}^* \, true),$$

entonces, por la Proposición 4.2.15, existe una derivación de *narrowing* cuasi-perezoso en $\mathcal{R}'$

$$D_2 \equiv (e \overset{\theta''}{\leadsto}{}^* \, true),$$

que emplea las correspondientes resultantes de $\mathcal{R}'$ sobre las correspondientes posiciones de e. Dado que el *narrowing* sin restricciones subsume cualquier derivación de *narrowing*, la anterior puede considerarse una derivación de *narrowing* sin restricciones en $\mathcal{R}'$. Ahora, aplicando la corrección débil de la NPE usando *narrowing* sin restricciones, existe una derivación de *narrowing* sin restricciones en $\mathcal{R}$ tal que

$$D_3 \equiv (e \overset{\theta'}{\leadsto}{}^* \, true), \text{ con } \theta' \leq \theta''[\mathcal{V}ar(e)].$$

Por la corrección del *narrowing*, θ' es una solución para e en $\mathcal{R}$. Ahora, por la completitud del *narrowing* perezoso para respuestas constructoras, existe una derivación de *narrowing* perezoso en $\mathcal{R}$,

$$D_4 \equiv (e \overset{\theta}{\leadsto}_{LN}{}^* \, true), \text{ con } \theta \leq \theta'[\mathcal{V}ar(e)].$$

$\square$

En [14] se presenta una prueba del Teorema 3.3.6 que puede ser parametrizada para cualquier estrategia de *narrowing*, bajo el supuesto de que se cumplen las condiciones de completitud de la estrategia considerada. Aquí hemos preferido mantener el anterior esquema de prueba, no solamente porque pueda generalizarse del mismo modo a otras estrategias de *narrowing*, sino porque revela ciertos aspectos interesantes sobre la dificultad para probar la completitud (débil) del método de NPE para ciertas estrategias de *narrowing*, como la perezosa. La causa principal de que el razonamiento de la prueba anterior no pueda invertirse (para demostrar el resultado de completitud débil) es que, en general, aunque el programa transformado $\mathcal{R}'$ obtenido a través de un proceso de LN-PE es parte de una NPE, lo contrario no es cierto. Por otra parte, el Teorema 3.3.7, que debería emplearse en lugar del Teorema 3.3.6 en la prueba de la completitud, requiere que el programa sea terminante, además de ser confluente (contrariamente al Teorema 3.3.6, que solamente exige la confluencia del programa). Dado que los programas con los que tratamos no son en general terminantes, la completitud (débil) para el proceso de LN-PE debe abordarse empleando técnicas propias.

En lo que resta de apartado nos centramos la prueba de la completitud débil de la LN-PE con el postproceso de renombramiento y discutimos las dificultades para alcanzar resultados más fuertes.

La clave para la prueba de la completitud débil de la LN-PE consiste en establecer que cada secuencia de reescritura en el programa original $\mathcal{R}$, para una ecuación e S-cerrada, puede ser reproducida para la ecuación renombrada en el programa renombrado $\mathcal{R}''$, resultado final de la LN-PE con el postproceso de renombramiento.

Proposición 4.3.2 *Sea $\mathcal{R}$ un TRS ortogonal y CB, e una ecuación y S un conjunto finito de términos. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}'$ y e son S-cerrados. Sea S' un renombramiento independiente de S, y e'' (resp. $\mathcal{R}''$) un renombramiento de e (resp. $\mathcal{R}'$) empleando S'. Si $e \to^* true$ en $\mathcal{R}$ entonces $e'' \to^* true$ en $\mathcal{R}''$.*

Antes de probar este resultado precisamos realizar ciertas consideraciones y demostrar una serie de resultados auxiliares.

Los TRS's ortogonales poseen la interesante propiedad de que los descendientes de los redexes a lo largo de una secuencia de reescritura son también redexes [113]. Más concretamente, Klop indicó en [112] que la contracción de un redex situado bajo otro redex no hace que este último deje de serlo. Recientemente, Middeldorp ha extendido en [146] esta propiedad a los TRS's cuasi ortogonales. Esto se concreta en el siguiente lema.

Lema 4.3.3 *Sea $\mathcal{R} = (\mathcal{F}, R)$ un TRS cuasi ortogonal, t un término tal que $t = \sigma(l)$ para alguna $l \to r \in R$ y $\sigma \in Subst(\mathcal{F}, \mathcal{X})$. Si $u \in \mathcal{P}os_{\mathcal{R}}(t) \backslash \{\Lambda\}$, entonces, para todo término s, existe una $\sigma' \in Subst(\mathcal{F}, \mathcal{X})$ tal que $t[s]_u = \sigma'(l)$.*

La demostración de este lema revela que el redex para el cual se han contraido posiciones más internas, y que continua siendo un redex, lo es con respecto a la misma lhs que lo era antes.

En lo que resta de este apartado consideraremos secuencias de reescritura *outermost*, i.e., secuencias en las que se explotan en paralelo, o mediante un multipaso, los redexes más externos de un término. Se sabe que la estrategia *outermost* es *fair* [113], en el sentido de que si se relaja la condición de explotar las ocurrencias de los redexes más externos en el momento en el que éstos aparecen, y se permite retrasar la evaluación de un redex (más externo) siempre y cuando se garantice que será evaluado eventualmente (él o sus descendientes), la estrategia que se obtiene es normalizante para TRS's ortogonales.

El siguiente lema de *lifting* es una variante del resultado de completitud del *narrowing* perezoso y es útil para la prueba de la Proposición 4.3.2.

Lema 4.3.4 *Sea $\mathcal{R}$ un TRS ortogonal y CB. Sea σ una substitución constructora, V un conjunto finito de variables y s un término encabezado por un símbolo de función, con $\mathcal{V}ar(s) \subseteq V$. Si $\sigma(s) \to_{p_1,R_1} \cdots \to_{p_n,R_n} t$ es una secuencia de reducción* outermost*, entonces existe una derivación de* narrowing *perezoso $s \overset{p_1,R_1,\sigma_1}{\leadsto}_{LN} \cdots \overset{p_n,R_n,\sigma_n}{\leadsto}_{LN} t'$ y una substitución constructora σ' tal que $\sigma'(t') = t$ y $\sigma' \circ \sigma_n \circ \cdots \circ \sigma_1 = \sigma$ $[V]$.*

Prueba. Análoga a prueba del Lema 3.4 que aparece en [148]. □

Este lema puede generalizarse para hacerlo aplicable incluso cuando la substitución que se considera no es constructora, siempre que no introduzca nuevos redexes. Esto resulta muy útil porque nos permite no prestar atención a ciertos subtérminos encabezados por símbolos de función y que son introducidos por instanciación, cuando no se contraen en la derivación considerada. El siguiente resultado es auxiliar:

Lema 4.3.5 *[16] Sea $\mathcal{R}$ un TRS ortogonal y CB, Sea $\theta = \{x_1/s_1, \ldots, x_m/s_m\}$ una substitución idempotente tal que s_i es un término encabezado por un símbolo de función, para todo $i = 1, \ldots, m$. Sea s un término encabezado por un símbolo de función y $\theta(s) = t_0 \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} t_n$ una secuencia de reducción donde $A_i = (t_{i-1} \rightarrow_{p_i,R_i} t_i)$ y $P_i = P_{i-1} \backslash A_i$, para $i = 1, \ldots, n$, $n \geq 0$. Si $p \parallel p_i$ o $p < p_i$ para todo $p \in P_{i-1}$, $i = 1, \ldots, n$, entonces existe una secuencia de reducción $s \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} s'$ tal que $\theta(s') = t_n$.*

Ahora podemos extender el Lema 4.3.4 para substituciones no constructoras que no introducen nuevos redexes.

Proposición 4.3.6 *Sea $\mathcal{R}$ un TRS ortogonal y CB. Sea σ una substitución y V un conjunto finito de variables. Sea s un término encabezado por un símbolo de función y $\mathcal{V}ar(s) \subseteq V$. Sea $\sigma(s) \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} t$ una secuencia de reescritura* outermost *tal que, para todos los redexes* outermost *$\sigma(s)|_p$ de $\sigma(s)$, $p \in \mathcal{FP}os(s)$. Entonces, existe una derivación de* narrowing *perezoso $s \rightsquigarrow_{p_1,R_1,\sigma_1} \cdots \rightsquigarrow_{p_n,R_n,\sigma_n} t'$ y una substitución σ' tal que $\sigma'(t') = t$ y $\sigma' \circ \sigma_n \circ \cdots \circ \sigma_1 = \sigma\ [V]$.*

Prueba. En esta prueba consideramos dos casos:

1. σ es una substitución constructora.

   En este caso, la prueba se sigue directamente del Lema 4.3.4, y σ' es una substitución constructora.

2. σ es una substitución no constructora.

   Entonces, existen unas substituciones θ_1 y θ_2 tales que $\sigma = \theta_2 \circ \theta_1$, donde podemos suponer que la substitución θ_1 es constructora y que, para todo elemento $s' \in \mathcal{R}an(\theta_2)$, s' es un término encabezado por un símbolo de función. Entonces, $\sigma(s) = \theta_2(\theta_1(s))$. Aplicando el Lema 4.3.5, tenemos que $\theta_1(s) \rightarrow_{p_1,R_1} \cdots \rightarrow_{p_n,R_n} s''$ tal que $\theta_2(s'') = t$. Por otro lado, ya que σ no introduce redexes *outermost* (i.e., si $\sigma(s)|_p$ es un redex *outermost*, entonces $p \in \mathcal{FP}os(s)$), entonces la secuencia es una secuencia *outermost*. Ahora, aplicando el Lema 4.3.4 a esta secuencia de reducción, existe una derivación de *narrowing* perezoso $s \overset{p_1,R_1,\sigma_1}{\rightsquigarrow_{LN}} \cdots \overset{p_n,R_n,\sigma_n}{\rightsquigarrow_{LN}} t'$ y una substitución constructora σ'' tal que $\sigma''(t') = s''$ y $\sigma'' \circ \sigma_n \circ \cdots \circ \sigma_1 = \theta_1\ [V]$. Tomando $\sigma' = \theta_2 \circ \sigma''$, tenemos que $\sigma'(t') = (\theta_2 \circ \sigma'')(t') = \theta_2(\sigma''(t')) = \theta_2(s'') = t$. Finalmente, ya que $\sigma'' \circ \sigma_n \circ \cdots \circ \sigma_1 = \theta_1\ [V]$, tenemos que $\theta_2 \circ \sigma'' \circ \sigma_n \circ \cdots \circ \sigma_1 = \theta_2 \circ \theta_1\ [V]$ y, por consiguiente, $\sigma' \circ \sigma_n \circ \cdots \circ \sigma_1 = \sigma\ [V]$.

$\square$

Finalmente estamos en disposición de demostrar la Proposición 4.3.2.

Prueba. Sean $B_1, \ldots, B_m$ todas las secuencias de reducción de e a *true*, y sea k_i el número de redexes contraidos en B_i, $i = 1, \ldots, m$. Realizamos la prueba del enunciado de esta proposición por inducción sobre el número máximo $n = max(k_1, \ldots, k_m)$ de redexes que es necesario contraer para reducir e a *true*.

1. Caso base ($n = 0$):

   Trivial, ya que en este caso e es *true* y $e'' = ren(true, S') = true$.

2. Caso inductivo ($n > 0$):

   Puesto que e es S-cerrado, existe un conjunto de cierre $\{(p_1, s_1), \ldots, (p_m, s_m)\} \in CSet(S, e)$, con $m > 0$, donde $p_i \in \mathcal{FP}os(e)$ y $s_i \in S$, $i = 1, \ldots, m$. Haciendo uso del Lema 4.3.3 y de las buenas propiedades de los TRS's ortogonales, consideramos el siguiente proceso de normalización:

   (a) Seleccionar el subtérmino más interno de e que sea S-cerrado, i.e., $e|_{p_i} = \theta(s_i)$ siendo $\mathcal{R}an(\theta)$ un conjunto de términos S-cerrado, y que cumple las siguientes condiciones:

   - existe al menos una posición $q \in \mathcal{P}os(e|_{p_i})$ tal que $e|_{p_i \cdot q}$ es un redex de e, y
   - para todo redex $e|_{p_i \cdot q'}$ de e, tenemos que $q' \in \mathcal{FP}os(s_i)$.

   Es evidente que un término así existe, ya que de otro modo e no contendría redexes o no sería S-cerrado, en contra de lo supuesto.

   (b) Explotar los sucesivos redexes de $e|_{p_i}$ seleccionando, en orden, los redexes más externos (i.e., reducir siguiendo un orden *outermost*).

   Por consiguiente, podemos considerar la siguiente secuencia de reducción,

   $$e[\theta(s_i)]_{p_i} \rightarrow_{p_i \cdot q_1, R_1} \ldots \rightarrow_{p_i \cdot q_k, R_k} e[t_k]_{p_i} \rightarrow^* true$$

   donde la correspondiente secuencia de reducción para $\theta(s_i)$

   $$\theta(s_i) \rightarrow_{q_1, R_1} \ldots \rightarrow_{q_k, R_k} t_k$$

   es *outermost* y t_k es una hnf, $k > 0$. Sea V un conjunto finito de variables que contiene $\mathcal{V}ar(s_i)$. Por la Proposición 4.3.6, sabemos que existe una derivación de *narrowing* perezoso $s_i \leadsto_{q_1, R_1, \sigma_1} \ldots \leadsto_{q_k, R_k, \sigma_k} t_k$ que contrae las correspondientes posiciones empleando las correspondientes reglas. Por definición de LN-PE, alguna resultante de $\mathcal{R}'$ se ha construido a partir de un prefijo de esta derivación. Supongamos que la siguiente subderivación

   $$s_i \overset{q_1, R_1, \sigma_1}{\leadsto}_{LN} \ldots \overset{q_j, R_j, \sigma_j}{\leadsto}_{LN} t', \quad 0 < j \leq k$$

es la que ha sido usada para construir esa resultante. Sea $\sigma'' = \sigma_j \circ \cdots \circ \sigma_1$. Dado que $\theta(s_i) \to_{q_1,R_1} \ldots \to_{q_j,R_j} t$, nuevamente por la Proposición 4.3.6, existe una substitución σ' tal que $\sigma'(t') = t$ y $\sigma' \circ \sigma'' = \theta\ [V]$. Así, la resultante considerada (una vez renombrada) tiene la forma

$$R'' = (\sigma''(s'_i) \to ren(t', S')),$$

donde $\langle s_i \mapsto s'_i \rangle \in S'$, y la secuencia de reducción que se considera en $\mathcal{R}$ tiene la forma

$$e = e[\theta(s_i)]_{p_i} \to_{p_i.q_1,R_1} \ldots \to_{p_i.q_j,R_j} e[t]_{p_i} \to^* true$$

Ahora es fácil demostrar que la ecuación renombrada $e'' = ren(e, S')$ puede reducirse sobre la posición p''_i utilizando la regla R'', donde p''_i es la posición correspondiente de p_i en e: por hipótesis, $\mathcal{R}an(\theta)$ es S-cerrado; más aún, ya que σ'' es una substitución constructora y $\sigma' \circ \sigma'' = \theta\ [V]$, tenemos que $\sigma'(x)$ es también un término S-cerrado para todo $x \in V$. Entonces, existe una substitución $\theta' = \{x/ren(\sigma'(x), S') \mid x \in V\}$ tal que $\mathcal{R}an(\theta')$ es $\mathcal{A}$-cerrado; por definición de postproceso de renombramiento, $e''|_{p''_i} = ren(e|_{p_i}, S') = ren(\theta(s_i), S') = ren(\sigma' \circ \sigma''(s_i), S') = \theta'(\sigma''(ren(s_i, S'))) = \theta'(\sigma''(s'_i))$; por consiguiente, $e''|_{p''_i}$ es una instancia de la lhs $\sigma''(s'_i)$ y puede darse el paso de reescritura

$$e''|_{p''_i} = \theta'(\sigma''(s'_i)) \to_{\Lambda,R''} \theta'(ren(t', S')) = ren(\sigma'(t'), S') = ren(t, S')$$

y, así $e'' \to_{p''_i,R''} e''[ren(t, S')]_{p''_i}$, de donde resulta inmediato que $e''[ren(t, S')]_{p''_i} = ren(e[t]_{p_i}, S')$.

Con el fin de constatar que se cumplen las condiciones para aplicar la hipótesis de inducción, comprobamos si el término $e[t]_{p_i}$ es S-cerrado: por la Proposición 4.2.5, el término $ren(t', S')$ es $\mathcal{A}$-cerrado y por consiguiente, por el Lema 4.2.8, t' es S-cerrado; puesto que $\sigma'(x)$ es S-cerrado para todo $x \in V$, entonces por definición de la condición de cierre, $\sigma'(t') = t$ es también S-cerrado; por hipótesis, e es una expresión S-cerrada y dado que la posición p_i es una posición de cierre, el contexto $e[\ ]_{p_i}$ es S-cerrado. Ahora, nuevamente por definición de la condición de cierre, $e[t]_{p_i}$ es una expresión S-cerrada.

Ahora $e[t]_{p_i} \to^* true$ es una secuencia de reducción con menos de n redexes por contraer y $e[t]_{p_i}$ es una expresión S-cerrada. Aplicando la hipótesis de inducción, existe una secuencia de reducción $e''[ren(t, S')]_{p''_i} \to^* true$ en $\mathcal{R}''$, con lo que se puede construir la secuencia de reducción

$$e'' \to_{p''_i,R''} e''[ren(t, S')]_{p''_i} \to^* true$$

en $\mathcal{R}''$, lo que constituye el resultado deseado.

□

La completitud débil del método de LN-PE es una consecuencia directa de la Proposición 4.3.2 y de la corrección y completitud del *narrowing* perezoso.

Teorema 4.3.7 (Completitud Débil) *Sea $\mathcal{R}$ un TRS ortogonal y CB, e una ecuación, y S un conjunto finito de términos. Sea $\mathcal{R}'$ una LN-PE de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}'$ y e son S-cerrados. Sea S' un renombramiento independiente de S, y e'' (resp. $\mathcal{R}''$) un renombramiento de e (resp. $\mathcal{R}'$) empleando S'. Sea $\mathcal{R}''$ un TRS no ambiguo. Si el* narrowing *perezoso computa la respuesta θ para e en $\mathcal{R}$, entonces computa una respuesta θ'' para e'' en $\mathcal{R}''$, con $\theta'' \leq \theta[\mathcal{V}ar(e)]$.*

Prueba. Puesto que existe una derivación de *narrowing* perezoso $e \overset{\theta}{\leadsto}_{LN}^{*} true$ en $\mathcal{R}$, por la corrección del *narrowing* perezoso se tiene que $\theta(e) \to^{*} true$ en $\mathcal{R}$. Dado que (por la Proposición 2.9.17) los términos del rango de la substitución $\theta_{\upharpoonright \mathcal{V}ar(e)}$ son constructores, claramente, por definición de la condición de cierre, el término $\theta(e)$ es S-cerrado, y $ren(\theta(e), S') = \theta(e'')$. Ahora, por la Proposición 4.3.2, existe una secuencia de reescritura $\theta(e'') \to^{*} true$ en $\mathcal{R}''$. Por otra parte, dado que el TRS $\mathcal{R}''$ es CB, lineal por la izquierda (Proposición 4.2.5) y no ambiguo, $\mathcal{R}''$ es TRS ortogonal y CB. Por consiguiente, ya que θ es una solución de e'' en $\mathcal{R}''$, por la completitud del *narrowing* perezoso, existe una derivación $e'' \overset{\theta''}{\leadsto}_{LN}^{*} true$ en $\mathcal{R}''$, tal que $\theta'' \leq \theta\ [\mathcal{V}ar(e)]$. □

Una observación importante, respecto a la prueba de la completitud que acabamos de presentar, es que ha sido necesario imponer la condición de que el programa transformado $\mathcal{R}''$ sea no ambiguo. Esto se debe a que, en general, se puede producir una pérdida de la condición de no ambigüedad al derivar el programa especializado $\mathcal{R}'$ que el postproceso de renombramiento no siempre es capaz de recuperar. El siguiente ejemplo ilustra este problema.

Ejemplo 21 *[17] Considérese el siguiente programa:*

$$\begin{array}{rcl} f(0,0) & \to & s(f(0,0)) \\ f(s(N),X) & \to & s(f(N,X)) \\ g(0) & \to & g(0) \\ h(s(X)) & \to & 0. \end{array}$$

El programa hace un uso exhaustivo de las características que hacen apreciable la evaluación perezosa, i.e., la manipulación de estructuras de datos infinitas y llamadas a función no terminantes. Queremos especializar este programa con respecto a la llamada inicial $h(f(X, g(Y)))$. Nótese que este término se reduce a 0 si la variable X se enlaza a $s(\square)$, mientras que no termina si X se enlaza a 0 (debido a una evaluación del segundo argumento que no termina). Si aplicamos el algoritmo de LN-PE y el postproceso de renombramiento (renombrando el término inicial como

$h2(X,Y)$), obtenemos el siguiente programa renombrado[7]*:*

$$\begin{array}{rcl} h1(X) & \rightarrow & h1(X) \\ h1(s(X)) & \rightarrow & 0 \\ h2(X,0) & \rightarrow & h1(X) \\ h2(s(X),Y) & \rightarrow & 0 \\ h2(s(X),0) & \rightarrow & 0 \end{array}$$

Contrariamente al programa original, el programa renombrado no es ortogonal, debido a la aparición de varios pares críticos (aunque todos son convergentes). Es más, ni siquiera es cuasi ortogonal. Aunque el programa renombrado eventualmente podría ejecutarse usando el algoritmo de narrowing *perezoso (ya que los problemas planteados siguen siendo de unificación lineal), la pérdida de la no ambigüedad rompe la completitud de la estrategia.*

Nótese también que el programa filtrado tiene un comportamiento peor que el programa original, respecto a la terminación. Por ejemplo, consideremos el término $h(f(s(0),g(0)))$. La evaluación de este término empleando la estrategia de *narrowing* perezoso da lugar a un árbol de búsqueda finito para el programa original $\mathcal{R}$. Sin embargo, el término renombrado $h2(s(0),0)$ tiene un árbol de búsqueda infinito en el programa filtrado $\mathcal{R}''$. La rama infinita se debe a la aplicación de las reglas $h2(X,0) \rightarrow h1(X)$ y $h1(X) \rightarrow h1(X)$. Asimismo, este ejemplo muestra que la especialización de programas usando *narrowing* perezoso puede destruir las ventajas de la existencia de cómputos de reducción deterministas en el programa original.

En [17] también se ha observado que cuando se realiza una LN-PE es posible la aparición de reglas redundantes en el programa especializado, si bien éste es un inconveniente menor que puede evitarse mediante la eliminación de las reglas duplicadas en una fase de postproceso adicional.

Por otra parte, el hecho de que la estrategia de *narrowing* perezoso no goce de propiedades de optimalidad como las del *narrowing* necesario (ver Apartado 2.9.4) dificulta las pruebas de corrección fuerte. Es inmediato demostrar que, bajo la condición de que la estrategia de evaluación empleada computa respuestas independientes, los teoremas 4.3.1 y 4.3.7 pueden fortalecerse para establecer que el programa original y el renombrado computan las mismas respuestas.

En [17] se demuestra que los inconvenientes que acabamos de relatar no aparecen cuando se realiza una NPE basada en la estrategia de *narrowing* necesario. En los próximos capítulos estudiaremos las condiciones bajo las cuales la NPE basada en la estrategia de *narrowing* perezoso no pierde tampoco las buenas cualidades de su mecanismo de base y goza también de las propiedades de corrección y completitud fuertes.

Para terminar este capítulo, mostramos la potencia del procedimiento de LN-PE estudiando la especialización del programa *match* para la búsqueda de

[7]Este programa se ha obtenido utilizando el sistema INDY. El sistema INDY permite combinar *narrowing* perezoso y simplificación usando un subconjunto terminante de reglas de las reescritura [95]. Si bien la simplificación puede tener un impacto positivo en la estructura de los programas especializados, no puede explotarse de forma efectiva en este ejemplo, ya que la única regla terminante es la que define la función h.

patrones en cadenas introducido en [114]. Entre otros autores, este ejemplo ha sido discutido por [106, 188, 108].

4.4 Búsqueda de Patrones en Cadenas.

Un ejemplo estándar en la literatura sobre evaluación parcial es el problema consistente en derivar un programa eficiente para la búsqueda de patrones en cadenas de caracteres, partiendo de un programa ingenuo y poco eficiente que se especializa, para un patrón determinado, empleando evaluación parcial [86, 108]. El programa original $\mathcal{R}$ (un listado del cual puede verse en la Figura 4.4(a)) comprueba si un patrón p aparece dentro de una cadena s, comparando iterativamente dicho patrón con un prefijo de s. En el caso de que p no ajuste con el prefijo de s considerado, el primer elemento de s se elimina y el proceso comienza de nuevo con el resto de la cadena s. Esta estrategia de comparación no es óptima, ya que pueden realizarse reiteradamente comprobaciones innecesarias sobre los mismos elementos de la cadena. El poder de una técnica de transformación puede medirse comprobando si puede alcanzar, por medios automáticos, el grado de optimización que se obtiene al aplicar el algoritmo de Knuth, Morris y Pratt (KMP) para la búsqueda de patrones, que construye un autómata finito determinista. El llamado "test KMP" se emplea a menudo para comparar la fortaleza de los diferentes especializadores. Este ejemplo es particularmente interesante porque es un tipo de optimización que no son capaces de obtener automáticamente ni la evaluación parcial clásica (de programas funcionales) ni tampoco la deforestación [108]. La deducción parcial de programas lógicos y la supercompilación positiva de programas funcionales sí que pasan el test [108]. El método de LN-PE, aquí presentado, también resuelve satisfactoriamente el problema, como ilustra el siguiente ejemplo.

Ejemplo 22 *Sea $\mathcal{R}$ el programa ingenuo de la Figura 4.4* (a) *para la búsqueda de patrones en cadenas de bits. Supongamos que el patrón es la cadena "001" y que deseamos resolver el problema de búscar la aparición de dicho patrón dentro de una cadena arbitraria s. Si aplicamos el evaluador parcial basado en* narrowing *perezoso al término $match(001, s)$, obtenemos el programa $\mathcal{R}'$*[8] *(Figure 4.4,* (b)*). Después del postproceso de renombramiento se obtiene el programa especializado $\mathcal{R}''$ (Figure 4.4,* (c)*). El grado de especialización obtenido en este programa es esencialmente análogo al de las reglas producidas por el supercompilador positivo de [188]. El programa especializado se comporta como un buscador de patrones de estilo KMP, dando las mismas ventajas, en términos de complejidad, que el algoritmo de KMP cuando se le compara con el programa original. Para un patrón determinado p el programa especializado realiza*

[8]Por simplicidad, se han omitido las reglas que reducen las funciones a "*false*". Se ha empleado la regla de desplegado `ostrans` (ver Apéndice A) para obtener la mejor de las especializaciones posibles. Tendremos que esperar a las mejoras en el control del método de NPE, introducidas en el capítulo 8, para obtener el mismo resultado con estrategias de desplegado más liberales.

(a) $\mathcal{R}$ (Buscador de patrones ingenuo):

$$
\begin{array}{rcll}
match(p,s) & \rightarrow & loop(p,s,p,s) & \\
loop(nil,ss,op,os) & \rightarrow & true & \\
loop(p:pp,nil,op,os) & \rightarrow & false & \\
loop(p:pp,s:ss,op,os) & \rightarrow & (p \approx s \Rightarrow loop(pp,ss,op,os)) & \%\ \texttt{continuar} \\
loop(p:pp,s:ss,op,os) & \rightarrow & ((p \approx s) \approx false \Rightarrow next(op,os)) & \%\ \texttt{saltar cadena} \\
next(op,nil) & \rightarrow & false & \\
next(op,s:ss) & \rightarrow & loop(op,ss,op,ss) & \%\ \texttt{ir a loop}
\end{array}
$$

(b) $\mathcal{R}'$ (LN-PE de $\mathcal{R}$ con respecto a $match(001,s)$):

$$
\begin{array}{rl}
match(001,s) & \rightarrow loop(001,s,001,s) \\
loop(001,0:ss,001,0:ss) & \rightarrow loop(01,ss,001,0:ss) \\
loop(001,s:ss,001,s:ss) & \rightarrow ((0 \approx s) \approx false \\
 & \Rightarrow loop(001,ss,001,ss)) \\
loop(01,0:ss',001,00:ss') & \rightarrow loop(1,ss',001,00:ss') \\
loop(01,s':ss',001,0:s':ss') & \rightarrow ((0 \approx s') \approx false \\
 & \Rightarrow loop(001,ss',001,s':ss')) \\
loop(1,1:ss'',001,001:ss'') & \rightarrow true \\
loop(1,s'':ss'',001,00:s'':ss'') & \rightarrow ((1 \approx s'') \approx false \\
 & \Rightarrow loop(01,s'':ss'',001,0:s'':ss''))
\end{array}
$$

(c) $\mathcal{R}''$ (Renombramiento de $\mathcal{R}'$):

$$
\begin{array}{rcl}
match'(s) & \rightarrow & loop\_001(s) \\
loop\_001(0:ss) & \rightarrow & loop\_01(ss) \\
loop\_001(s:ss) & \rightarrow & ((0 \approx s) \approx false \Rightarrow loop\_001(ss)) \\
loop\_01(0:ss) & \rightarrow & loop\_1(ss) \\
loop\_01(s:ss) & \rightarrow & ((0 \approx s) \approx false \Rightarrow loop\_001(ss)) \\
loop\_1(1:ss) & \rightarrow & true \\
loop\_1(s:ss) & \rightarrow & ((1 \approx s) \approx false \Rightarrow loop\_01(s:ss))
\end{array}
$$

Figura 4.4: Test KMP para programas de búsqueda de patrones en cadenas de bits.

muchas menos comparaciones, más aún conforme la longitud de p se hace más grande (i.e., aumenta su número de bits), la complejidad del programa original también se hace más grande, mientras que para el programa especializado existe una cota superior que es independe de la longitud de p.

Hemos supuesto que la búsqueda se realiza sobre cadenas de bits, i.e., cadenas que contienen solamente ceros o unos. Nótese que la elección de un alfabeto binario finito tiene repercusiones a la hora de alcanzar el éxito en la especialización de este ejemplo. Para obtener resultados similares cuando se manipula un alfabeto general, es necesario el uso de mecanismos que permitan la propagación de información negativa durante el proceso de especialización [190].

4.5 Conclusiones.

En este capítulo, hemos investigado las propiedades de una instancia del método genérico de NPE, descrito en el capítulo 3, basada en el *narrowing* perezoso. El proceso de evaluación parcial se desarrolla en dos fases. A la fase de evaluación parcial propiamente dicha, sigue una fase de renombramiento que restaura la disciplina de constructores del programa original y también alcanza la deseada condición de independencia del conjunto de llamadas evaluadas parcialmente. Se ha demostrado que la transformación de renombramiento es correcta, en el sentido de que el programa residual y su forma renombrada computan las mismas respuestas. También se ha probado la corrección débil del evaluador parcial obtenido y se han discutido las dificultades para conseguir resultados más fuertes. Utilizando los resultados conocidos previamente acerca del *narrowing*, las pruebas presentan una estructura relativamente más simple que para otros procedimientos de transformación.

Desde un punto de vista práctico, se ha constatado que la inclusión de un proceso de normalización entre pasos de *narrowing* no solamente ahorra tiempo y espacio sino que puede conducir a mejores resultados de especialización, ya que evita el desplegado de puntos de elección innecesarios. El uso de la simplificación también mejora la capacidad para eliminar estructuras de datos intermedias en un proceso de evaluación parcial basado en el *narrowing* (perezoso). Se ha mostrado que el evaluador parcial puede obtener los mismos efectos de transformación que otros transformadores descritos en la literatura, sin el empleo de técnicas *ad hoc* como las empleadas generalmente por éstos (por ejemplo, como algunas técnicas de *postunfolding* que eliminan funciones intermedias en la supercompilación positiva [86, 186, 188]). Es ampliamente conocido en el ámbito de la deducción parcial que el empleo del llamado *desplegado determinado* constituye una heurística muy efectiva. En general, la preferencia por los cómputos deterministas evita la explosión de código y es una técnica comparable al criterio basado en el determinismo de [76], que explora los caminos deterministas máximos.

Finalmente, hemos mostrado que el método presentado pasa el denominado test de KMP [86, 106], i.e. consigue la especializacón de un programa de búsqueda de patrones ingenuo y poco eficiente con respecto a un patrón fijo, obteniendo una eficiencia comparable a la del algoritmo de Knuth, Morris y Pratt [114].

Resumiendo, las principales aportaciones de este capítulo son las siguientes:

1. Un procedimiento de evaluación parcial que es aplicable a los lenguajes lógico–funcionales con semántica operacional perezosa más populares, como Babel, $\mathcal{TOY}$ o Curry.

2. Una transformación de renombramiento que garantiza: (a) la independencia del conjunto de términos evaluados parcialmente, (b) la preservación de requisitos que son básicos para la completitud de la estrategia de *narrowing* perezoso y que hacen posible que el programa final transformado y renombrado sea ejecutable. (c) la preservación de la semántica del

programa, es decir, que el programa especializado intermedio y su forma renombrada computan las misma respuestas.

3. La corrección y completitud (débil) del método de NPE basado en el *narrowing* perezoso.

Capítulo 5

Programas Uniformes.

5.1 Introducción.

Como hemos mencionado en otras partes de esta memoria, para evitar computaciones innecesarias y posibilitar el empleo de estructuras de datos infinitas, muchos trabajos en el área de los lenguajes lógico–funcionales se han centrado en el estudio de las estrategias de *evaluación perezosa* [23, 79, 96, 138, 157]. De entre las estrategias de evaluación perezosa, la estrategia de *narrowing* necesario [23] se ha postulado como óptima (ver Teorema 2.9.23) ya que: es correcta y completa, con respecto a ecuaciones estrictas y soluciones constructoras, para la clase de programas inductivamente secuenciales; computa derivaciones necesarias de longitud mínima (en implementaciones basadas en grafos) y no obtiene soluciones redundantes. Sin embargo, el narrowing necesario presenta también inconvenientes derivados de su propio mecanismo de cómputo. El *narrowing* necesario guía los cómputos mediante el empleo de los árboles definicionales [18], que contienen toda la información sobre las reglas del programa. Estas estructuras permiten seleccionar una posición del término que se está evaluando y que señala la aparición de un subtérmino que es inevitable reducir para la obtención de un resultado. Sin embargo, como se muestra en [7, 6], la representación explícita y el uso de los árboles definicionales en la implementación del *narrowing* necesario puede tener un coste elevado, tanto en aspectos de eficiencia (en tiempo de ejecución) como en el consumo de memoria. La introducción de *técnicas incrementales* puede reducir en más de un 50% la cantidad de memoria utilizada en la representación de los árboles definicionales (como se demuestra en la mayor parte de las pruebas efectuadas en [7]); también se ha constatado que se experimenta un aumento de eficiencia en tiempo de ejecución que, si bien no es significativo en todos los casos, no provoca una sobrecarga en ninguno de ellos.

Por otra parte, las características de la evaluación parcial dirigida por *narrowing* dependen de la estrategia de *narrowing* empleada para el desplegado de los árboles locales. Como hicimos notar en el Capítulo 4, la evaluación parcial usando *narrowing* perezoso presenta varios inconvenientes: i) se obtienen reglas redundantes en el programa especializado; ii) puede introducirse un empeoramiento, con respecto a la terminación de los cómputos, en comparación con el comportamiento del programa original; iii) la especialización de programas usando *narrowing* perezoso también puede destruir las ventajas de la existencia de cómputos deterministas en el programa original. iv) se pierde la ortogonalidad del programa original. Esta última deficiencia impide que la estrategia de *narrowing* perezoso pueda emplearse para ejecutar el programa especializado preservando la completitud del cálculo. Como se muestra en [17], estos inconvenientes no aparecen cuando el proceso de evaluación parcial se basa en la estrategia de *narrowing* necesario.

En éste y los proximos capítulos investigamos la relación precisa existente entre la estrategia de *narrowing* necesario y la estrategia de *narrowing* perezoso, que no requiere de las costosas estructuras de los árboles definicionales, y se intenta identificar la clase de programas más amplia para la cual ambas estrategias tienen el mismo comportamiento operacional, evitándose así los problemas encontrados al usar la estrategia de *narrowing* perezoso en el proceso de evaluación parcial. Por consiguiente, el objetivo de esta parte se orienta a conseguir, empleando una estrategia de *narrowing* perezoso, los beneficios de la estrategia de *narrowing* necesario sin el coste de mantener y usar los árboles definicionales.

5.2 Caracterización de los Programas Uniformes.

Los programas uniformes fueron introducidos en [120, 121] para el lenguaje Babel. La *uniformidad* es una restricción sintáctica que permite una implementación eficiente de la estrategia de *narrowing* perezoso. A continuación presentamos una definición de programa uniforme ligeramente diferente a la presentada en [121] y adaptada a la clase de programas de primer orden sin tipos con los que trabajamos[1].

Definición 5.2.1 (Programa uniforme)
Un programa uniforme *consiste en un conjunto de reglas $f(t_1, \ldots, t_n) \to r$ que cumplen las siguientes restricciones:*

1. *Patrones* constructores planos*: cada t_i es una variable x o un constructor $c(x_1, \ldots, x_n)$. En el último caso, se dice que f demanda el i-ésimo argumento y que c es el* constructor demandante.

2. Linealidad por la izquierda*: las partes izquierdas de las reglas, $f(t_1, \ldots, t_n)$, no contienen ocurrencias múltiples de la misma variable.*

[1]Hemos cambiado la condición de *no-ambigüedad débil*, de la definición original en [121], por la condición de *no-ambigüedad (fuerte)*, que prohibe la existencia de reglas que sean un renombramiento una de otra. Este tipo de reglas son las únicas cuyas lhs's pueden unificar cumpliendo la restricción de no-ambigüedad débil en programas uniformes.

3. Restricción de variables libres*: las partes derechas de las reglas, r, no contienen* variables libres *(i.e., variables que no aparecen en la parte izquierda).*

4. Uniformidad*: sean $f(t_1, \ldots, t_n)$ y $f(s_1, \ldots, s_n)$ las partes izquierdas de dos reglas que definen f; entonces, t_i es una variable si y sólo si s_i es una variable.*

5. No-ambigüedad*: Si $l_1 \to r_1$ y $l_2 \to r_2$ son dos reglas distintas cualesquiera que definen f, l_1 y l_2 no son unificables*[2].

Lema 5.2.2 (Estructura de los Programas Uniformes) *Sea $\mathcal{R}$ un programa uniforme y $L_f(\mathcal{R})$ el conjunto de las lhs's de las reglas que definen la función f en $\mathcal{R}$.*

1. Sea una posición $k \in \{1, \ldots, n\}$. Para toda $l \in L_f(\mathcal{R})$, $l|_k$ es una variable, o bien para toda $l \in L_f(\mathcal{R})$, $l|_k$ es un constructor lineal plano.

2. Si en las lhs's de las reglas que definen f aparecen argumentos con términos constructores planos, dadas dos lhs's $l_1 \in L_f(\mathcal{R})$ y $l_2 \in L_f(\mathcal{R})$ correspondientes a reglas distintas cualesquiera, existe un argumento $k \in \{1, \ldots, n\}$ tal que $l_1|_k = c_1(x_1, \ldots, x_{n_1})$ y $l_2|_k = c_2(y_1, \ldots, y_{n_2})$ con $c_1 \neq c_2$. En caso contrario, f está definida por una sola regla $f(x_1, \ldots, x_n) \to r$.

Prueba. Demostremos los dos puntos por separado:

1. Procedemos por reducción al absurdo. Supongamos que existen dos lhs's $l_1 \in L_f(\mathcal{R})$ y $l_2 \in L_f(\mathcal{R})$, tal que en la posición k, $l_1|_k \equiv x$ y $l_2|_k \equiv c(x_1, \ldots, x_n)$. Esto viola la restricción (4) de uniformidad, con lo que $\mathcal{R}$ no puede ser un programa uniforme.

2. Procedemos por casos y por reducción al absurdo en cada caso:

   - Consideremos el caso en el que aparecen constructores planos en algunas de las posiciones $k \in \{1, \ldots, n\}$ de las lhs's de las reglas que definen f. Si f está definido por una sola regla, el punto (2) se cumple trivialmente. Si hay más de dos reglas que definen f, tomemos dos lhs's $l_1 \in L_f(\mathcal{R})$ y $l_2 \in L_f(\mathcal{R})$ cualesquiera. Supongamos que no existe un argumento $k \in \{1, \ldots, n\}$ tal que $l_1|_k = c_1(x_1, \ldots, x_n)$ y $l_2|_k = c_2(y_1, \ldots, y_n)$ con $c_1 \neq c_2$. Entonces, teniendo en cuenta el resultado del punto anterior, l_1 y l_2 son identicas salvo renombramiento, σ, y por tanto unifican, siendo σ el unificador más general. Esto viola la condición (5) de no-ambigüedad.
   - En el caso de que no aparezcan constructores planos en ninguna de las posiciones $k \in \{1, \ldots, n\}$ de las lhs's de las reglas que definen f, procedemos de igual forma. Concluimos que f está definida por una sola regla $f(x_1, \ldots, x_n) \to r$.

[2] Esta definición es una particularización de la dada en los preliminares, ya que para TRS's CB y ortogonales, al ser términos constructores los argumentos de las lhs's de las reglas, la única posibilidad de que dos reglas solapen es que lo hagan sobre la posición Λ.

$\square$

El Lema 5.2.2 pone de manifiesto la estructura de los programas uniformes. En un programa uniforme, las funciones f de aridad n están definidas:

1. por una o más reglas cuyas lhs's son de la forma

$$f(\ldots, c_{k_1}(x_1, \ldots, x_{m_{k_1}}), \ldots, c_{k_p}(y_1, \ldots, y_{m_{k_p}}), \ldots),$$

   siendo $\{k_1, \ldots, k_p\} \subseteq \{1, \ldots, n\}$ posiciones fijas en las que aparecen constructores planos mientras en el resto de las posiciones aparecen variables, o bien

2. por una sola regla de la forma $f(x_1, \ldots, x_n) \to r$.

En el primero de los casos, para cada par de reglas, en uno de los argumentos no variables debe aparecer al menos un término constructor plano con un símbolo constructor diferente. Por razones que quedarán claras más adelante, denominaremos a las posiciones fijas $\{k_1, \ldots, k_p\}$ en las que aparecen constructores planos *posiciones inductivas* de f.

Ejemplo 23 *El programa*

$$\begin{array}{lcl} f(X, a, b, c) & \to & 1 \\ f(X, b, b, c) & \to & 2 \\ f(X, b, b, b) & \to & 3 \end{array}$$

donde a, b y c se consideran símbolos constructores, es un ejemplo de programa uniforme.

5.3 Problemas de Implementación de la Estrategia de Narrowing Perezoso y Programas Uniformes.

La posibilidad de realizar evaluación perezosa es una característica importante de los lenguajes lógico-funcionales. Sin embargo, como se muestra en [104, 156], es difícil combinar una estrategia de evaluación perezosa con el uso de variables lógicas y una implementación secuencial que utilize una técnica de busqueda en profundidad con *vuelta atrás* (*backtracking*). Pueden destacarse los siguientes problemas relacionados con el uso de la evaluación perezosa:

- Por un lado, la computación de un término t puede no terminar si se demanda un subtérmino $t|_p$, en una posición interna p de t, para el que existen infinitos resultados y ninguno de ellos contribuye al cómputo (proporcionando la posibilidad de unificar una regla del programa con el término) más externo t.

- Por otra parte, cuando se retrasa la evaluación de un término, éste puede evaluarse varias veces, en lugar de una sola vez, como ocurriría si se empleara una estrategia de evaluación impaciente (*innermost*).
- Finalmente, la existencia de puntos de vuelta atrás "adicionales", asociados a la existencia de diversos redexes, dificulta una implementación eficiente (y posiblemente completa) de la estrategia de *narrowing* perezoso, que debe gestionar esos puntos de vuelta atrás "adicionales", junto con los habituales (en programación lógica) asociados a la elección de las diferentes reglas.

El primero de los problemas puede solucionarse mediante el empleo de técnicas de evaluación mixta, combinando *narrowing* perezoso con *narrowing* impaciente para el cómputo de los subtérminos demandados. El segundo puede resolverse utilizando compartición (*sharing*) de variables o alguna técnica de reducción basada en grafos. Los programas uniformes fueron introducidos por vez primera en [120] para paliar el último de los inconvenientes mencionados. El siguiente ejemplo aclara y profundiza en este último punto.

Ejemplo 24 *[120] Dado el programa (no uniforme)*

$$\begin{array}{llll} \mathcal{R} = \{ & R_1 : & f(0,0) & \to 0 \\ & R_2 : & f(s(X),0) & \to s(0) \\ & R_3 : & f(X,s(s(Y))) & \to s(s(0))\} \end{array}$$

y el término $t \equiv f(f(X,Y),Z)$, puede construirse el árbol de búsqueda de la Figura 5.1, donde se han subrayado los distintos redexes y se han etiquetado las ramas con las reglas de $\mathcal{R}$ empleadas en cada paso de narrowing *perezoso. Puede apreciarse que, cuando se evalúa el término t, las reglas R_1 y R_2 de $\mathcal{R}$ demandan el primer argumento, ya que $f(X,Y)$ no está suficientemente evaluado. Por lo tanto, se postpone la aplicación de las reglas R_1 y R_2 hasta haberse evaluado el argumento $f(X,Y)$. Sin embargo, la regla R_3 si puede ser aplicada directamente sobre t. Al evaluar el término demandado $f(X,Y)$, nuevamente debe considerarse la aplicación de todas las reglas.*

Este ejemplo muestra la aparición de diversos redexes demandados por distintas reglas, lo que obliga a considerar, en una implementación secuencial de la estrategia de *narrowing* perezoso, los puntos de elección de vuelta atrás asociados a las reglas así como asociados a la existencia de diferentes redexes (para los que no existe algo análogo en el contexto de la programación lógica y que provocan un nuevo intento de unificación con las diferentes reglas del programa). Esta singularidad plantea problemas de eficiencia, no solamente porque ambos tipos de puntos de elección deben ser gestionados en una implementación secuencial de la estrategia de *narrowing* perezoso, sino porque también pueden aparecer cómputos redundantes. Se debe notar que, en el ejemplo anterior, las dos ramas más a la derecha computan el mismo resultado con la misma respuesta. La principal dificultad radica en el hecho de que pueda darse un paso de *narrowing* perezoso sobre una posición de t (Λ) con una regla (R_3), mientras que no es posible dar un paso de *narrowing* perezoso con las otras reglas sobre dicha posición.

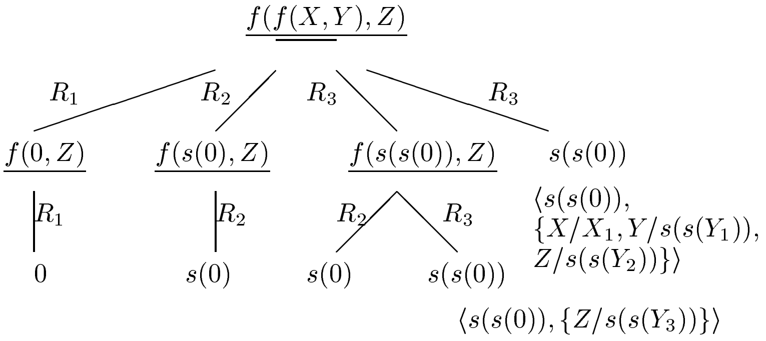

Figura 5.1: Arbol de búsqueda para el término $f(f(X,Y),Z)$ utilizando LN (Se muestran las salidas redundantes).

Los programas uniformes se introdujeron para evitar este tipo de situaciones, en el que diferentes reglas demandan diferentes redexes de un mismo término, lo que conduce a la existencia de puntos de vuelta atrás asociados a diferentes redexes que deben ser explotados. La idea es remplazar los puntos de vuelta atrás debidos a los diferentes redexes por puntos de vuelta atrás debidos a la elección de las diferentes reglas del programa.

En [120] se presenta un algoritmo que transforma cualquier programa (en el sentido definido en el Apartado 2.9.1) en un programa uniforme. Denotaremos este algoritmo mediante el símbolo $\mathcal{U}_B$. La transformación $\mathcal{U}_B$ es idempotente, esto es $\mathcal{U}_B(\mathcal{U}_B(\mathcal{R})) = \mathcal{U}_B(\mathcal{R})$. El siguiente ejemplo muestra las ventajas de transformar un programa en un programa uniforme.

Ejemplo 25 *La implementación del lenguaje lógico-funcional BABEL, propuesta en [120], transforma el programa $\mathcal{R}$ del Ejemplo 24 en el programa uniforme:*

$$\mathcal{U}_B(\mathcal{R}) = \quad \{ \begin{array}{ll} f(X,0) & \to h(X) \\ f(X,s(Y)) & \to g(Y) \\ g(s(Y)) & \to s(s(0)) \\ h(0) & \to 0 \\ h(s(X)) & \to s(0)\} \end{array}$$

donde g y h son símbolos de función nuevos, que no aparecen en el programa original. Ahora, para la evaluación del término $f(f(X,Y),Z)$, todas las reglas del programa que definen f se aplican sobre la posición Λ. En general, para un término $f(s_1,s_2)$, siendo s_2 un término encabezado por un símbolo de función, todas las reglas del programa que definen f demandan el segundo argumento s_2.

Para el programa del Ejemplo 25, los puntos de vuelta atrás debidos a diferentes redexes han sido eliminados. También puede comprobarse una mejora de la eficiencia debido a la desaparición de salidas redundantes. El programa del Ejemplo 24 computa las salidas $\langle s(s(0)), \{X/X_1, Y/s(s(Y_1)), Z/s(s(Y_2))\}\rangle$ y $\langle s(s(0)), \{Z/s(s(Y_3))\}\rangle$, donde claramente la última respuesta es más general que la primera, sin embargo el programa del Ejemplo 25 sólo computa la salida $\langle s(s(0)), \{Z/s(s(Y_3))\}\rangle$.

A pesar de las ventajas de los programas uniformes, no siempre se consigue con ellos la eliminación de los puntos de vuelta atrás asociados a diferentes redexes demandados, en un mismo término, por diferentes reglas del programa. Esto se pone de manifiesto más adelante, en el Ejemplo 30.

Por otra parte, la transformación $\mathcal{U}_B$ presenta deficiencias, algunas de las cuales ya fueron mencionadas en [120], que detallamos a continuación. La transformación $\mathcal{U}_B$ consta de dos fases: i) *aplanamiento*, que produce reglas cuyas lhs's son patrones constructores planos; ii) *obtención de uniformidad*, que asegura que se cumpla la correspondiente restricción de uniformidad sin violar el resto de las restricciones que debe respetar un programa uniforme. Sin embargo, $\mathcal{U}_B$ no siempre logra preservar la ortogonalidad (débil) del programa original, dando lugar a programas que violan la restricción de no-ambigüedad. Por este motivo en [120] se introduce una tercera fase consistente en "*fusionar*" adecuadamente las reglas que poseen la misma lhs, salvo renombramiento de variables, y que, por lo tanto, incumplen la restricción de no-ambigüedad (débil) que debe respetar todo programa uniforme. Esta tercera fase ha sido empleada para obtener el programa del Ejemplo 25, a partir de un programa transformado intermedio en el que se había perdido la ortogonalidad (débil) del programa original y que, por lo tanto, no era uniforme. Desgraciadamente, no siempre es posible eliminar el problema comentado a través de la fusión de reglas realizada en la fase final de la transformación sin introducir otros problemas. El ejemplo siguiente pone de manifiesto este hecho.

Ejemplo 26 *Dado el programa ortogonal y CB:*

$$\begin{array}{lll} \mathcal{R} = \{ & f(a,b,X) & \to 1 \\ & f(c,X,c) & \to 2 \\ & f(X,a,b) & \to 3\}. \end{array}$$

Por ser ortogonal, $\mathcal{R}$ es lineal por la izquierda y no-ambiguo. Asimismo, este programa cumple la restricción de patrones constructores planos. Por lo tanto la primera fase de la transformación $\mathcal{U}_B$ devuelve el programa original $\mathcal{R}$. Si aplicamos la segunda fase de la transformación $\mathcal{U}_B$ a $\mathcal{R}$, obtenemos el programa

$$\begin{array}{lll} \{ & f(Y,Z,X) & \to f_{iv}(Y,Z,X) \\ & f_{iv}(Y,b,X) & \to f_i(Y,b,X) \\ & f_i(a,b,X) & \to 1 \\ & f(Y,X,Z) & \to f_v(Y,X,Z) \\ & f_v(Y,X,c) & \to f_{ii}(Y,X,c) \\ & f_{ii}(c,X,c) & \to 2 \\ & f(X,Y,Z) & \to f_{vi}(X,Y,Z) \\ & f_{vi}(X,Y,c) & \to f_{iii}(X,Y,c) \\ & f_{iii}(X,a,b) & \to 3\}, \end{array}$$

donde los símbolos de función $f_i, \ldots, f_{vi}$ son símbolos nuevos que no aparecen en el programa original. Podemos observar que, ahora, el programa obtenido cumple la restricción de uniformidad pero incumple la restricción de no-ambigüedad, ya que las lhs's de las reglas primera, cuarta y septima unifican. Por lo tanto, el programa no es uniforme. Podemos intentar recuperar la restricción de no-ambigüedad del programa original $\mathcal{R}$ mediante la fusión adecuada de las reglas que plantean el conflicto. Tras la fusión de dichas reglas se obtiene el programa

$$\begin{array}{lll}
\mathcal{R}_1 = \{ & f(X,Y,Z) & \to f_1(X,Y,Z) \\
& f_1(Y,b,X) & \to f_i(Y,b,X) \\
& f_i(a,b,X) & \to 1 \\
& f_1(Y,X,c) & \to f_{ii}(Y,X,c) \\
& f_{ii}(c,X,c) & \to 2 \\
& f_1(X,Y,c) & \to f_{iii}(X,Y,c) \\
& f_{iii}(X,a,b) & \to 3\},
\end{array}$$

que tampoco es uniforme, ya que la función definida f_1, introducida para fundir las reglas en conflicto y eliminar el problema de la no-ambigüedad, incumple la restricción de uniformidad. El resultado es un programa que debe ser sometido a un nuevo proceso de transformación. Aplicando un proceso de transformación como el anteriormente descrito al programa $\mathcal{R}_1$, obtenemos el programa

$$\begin{array}{lll}
\mathcal{R}_2 = \{ & f(X,Y,Z) & \to f_1(X,Y,Z) \\
& f_1(X,Y,Z) & \to f_2(X,Y,Z) \\
& f_2(Y,b,X) & \to f_i(Y,b,X) \\
& f_i(a,b,X) & \to 1 \\
& f_2(Y,X,c) & \to f_{ii}(Y,X,c) \\
& f_{ii}(c,X,c) & \to 2 \\
& f_2(X,Y,c) & \to f_{iii}(X,Y,c) \\
& f_{iii}(X,a,b) & \to 3\},
\end{array}$$

Podemos apreciar que el programa $\mathcal{R}_2$ simplemente reproduce el programa $\mathcal{R}_1$ (pero ahora con la nueva función definida f_2 y la regla redundante $f_1(X,Y,Z) \to f_2(X,Y,Z)$). Esta claro que después de transformar este programa y fusionar sus reglas n veces obtendriamos el programa

$$\begin{array}{lll}
\mathcal{R}_n = \{ & f(X,Y,Z) & \to f_1(X,Y,Z) \\
& f_1(X,Y,Z) & \to f_2(X,Y,Z) \\
& . & \\
& . & \\
& . & \\
& f_{n-1}(X,Y,Z) & \to f_n(X,Y,Z) \\
& f_n(Y,b,X) & \to f_i(Y,b,X) \\
& f_i(a,b,X) & \to 1 \\
& f_n(Y,X,c) & \to f_{ii}(Y,X,c) \\
& f_{ii}(c,X,c) & \to 2 \\
& f_n(X,Y,c) & \to f_{iii}(X,Y,c) \\
& f_{iii}(X,a,b) & \to 3\},
\end{array}$$

que sigue sin ser uniforme.

La característica destacable del programa del Ejemplo 26 es que no es inductivamente secuencial, contrariamente a lo que sucede con el programa del Ejemplo 24. A la vista de este ejemplo podemos concluir que la transformación $\mathcal{U}_B$, compuesta de las tres fases anteriomente mencionadas, no termina obteniendo un programa uniforme cuando se parte, en general, de un programa (débilmente) ortogonal y CB. Conjeturamos que $\mathcal{U}_B$ solamente es correcta cuando se usa sobre programas inductivamente secuenciales.

5.4 Propiedades y Taxonomía de los Programas Uniformes.

Recientemente, Zartmann ha introducido una nueva clase de programas uniformes, obtenidos mediante una transformación, definida en [205], a partir de programas inductivamentes secuenciales [18, 23]. En lo que sigue denotaremos la transformación de Zartmann mediante el símbolo $\mathcal{U}_Z$. La transformación $\mathcal{U}_Z$ es idempotente. Esta transformación conduce a programas que llevan "precompilada" la información almacenada en los árboles definicionales, en el sentido de que, para ellos, el *narrowing* perezoso y el *narrowing* necesario coinciden (como mostraremos más adelante). Los programas uniformes definidos por Zartmann se caracterizan porque en las lhs's de las reglas que definen f puede haber como máximo un argumento en la posición $k \in \{1, \dots, n\}$ en el que aparece un constructor plano, siendo el resto de los argumentos con $i \neq k$, argumentos en los que aparecen variables. Atendiendo a esta característica, denominaremos a los programas uniformes definidos por Zartmann, *programas uniformes simples*, ya que sólo poseen una posición inductiva. La clase de programas uniformes simples es un subconjunto de los programas uniformes, como se ilustra en la Figura 5.3.

Ejemplo 27 *El programa*

$$\begin{array}{lcl} f(X,a,Y,Z) & \to & 1 \\ f(X,b,Y,Z) & \to & 2 \\ f(X,c,Y,Z) & \to & 3 \end{array}$$

donde a, b y c se consideran símbolos constructores, es un ejemplo de programa uniforme simple.

Los programas uniformes simples son inductivamente secuenciales, i.e., cada una de las funciones definidas en el programa tienen asociado un árbol definicional.

Ejemplo 28 *La función f definida en el programa del Ejemplo 27 tiene asociada el árbol definicional de la Figura 5.2.*

Notad que este árbol es único y sólo posee un nivel, en correspondencia con la única posición inductiva que aparece en la función f definida en el programa

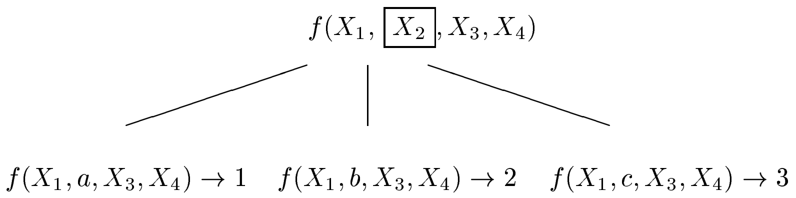

Figura 5.2: Arbol definicional para la función "f" definida en el Ejemplo 27.

del Ejemplo 27. De esta forma, los subárboles que cuelgan de la raíz del árbol definicional son hojas. Este hecho es generalizable al resto de los árboles definicionales asociados a las funciones definidas en programas uniformes simples.

La siguiente proposición pone de manifiesto que los programas uniformes también son inductivamente secuenciales. La Figura 5.3 ilustra la relación existente entre éstas y otras clases de programas ortogonales.

Proposición 5.4.1 *Todo programa uniforme es inductivamente secuencial.*

Prueba. Basta probar que se puede asociar un árbol definicional a cada una de las funciones f definidas en un programa uniforme $\mathcal{R}$. Para realizar la prueba, definiremos un procedimiento que permita la construccción de un árbol cuya raíz sea el término $f(x_1, \ldots, x_n)$ y cuyas hojas sean los términos del conjunto $L_f(\mathcal{R})$ de las lhs's de las reglas que definen la función f en $\mathcal{R}$. Finalmente, comprobaremos que el árbol construido cumple las restricciones de la Definición 2.9.18 de árbol definicional.

Sea $I = \{k \mid\ l \in L_f(\mathcal{R}) \wedge l|_k = c_k(x_1, \ldots, x_{m_k})\}$ el conjunto de posiciones inductivas de f que, por el Lema 5.2.2, está fijado por igual para todas las reglas de $\mathcal{R}$ que definen f. Nuestro procedimiento constructivo es el siguiente:

Algoritmo 2
Entrada: *El conjunto $L_f(\mathcal{R})$ de las lhs's que definen f.*
El conjunto I de posiciones en las que aparece un constructor plano.
Salida: *Un conjunto $\mathcal{P}$ de patrones lineales.*
Initialización: $\mathcal{P} = \{\pi_0\}$,
siendo $\pi_0 \equiv f(x_1, \ldots, x_n)$ y $x_1, \ldots, x_n$ variables nuevas
Mientras $I \neq \emptyset$ haced
1) *Seleccionar un $k \in I$ arbitrario; $I := I \setminus \{k\}$;*
2) $H = \{\pi \mid\ \pi$ *es una hoja de* $\mathcal{P}\}$;
3) **Repetir**
3.1) *Seleccionar una hoja $\pi \in H$; $H := H \setminus \{\pi\}$;*
3.2) $L_\pi = \{l \mid\ l \in L_f(\mathcal{R}) \wedge \pi \leq l\}$;
3.3) $C = \{c_i(x_1, \ldots, x_{m_i}) \mid\ l_i \in L_\pi \wedge \mathcal{H}ead(l_i|_k) = c_i\}$,
donde las variables $x_1, \ldots, x_{m_i}$ son nuevas
3.4) *Para cada $c_i(x_1, \ldots, x_{m_i}) \in C$,*
\- *formar una nueva hoja $\pi_i = \pi[c_i(x_1, \ldots, x_{m_i})]_k$,*
\- $\mathcal{P} := \mathcal{P} \cup \{\pi_i\}$;
Hasta que $H = \emptyset$

FinMientras
Devolver $\mathcal{P}$

Hacemos notar que el algoritmo anterior trata todos los casos posibles relativos a la definición de una función f por las reglas del programa. En particular, cuando la función f está definida por una sola regla $f(x_1, \ldots, x_n) \to r$, entonces $L_f(\mathcal{R}) = \{f(x_1, \ldots, x_n)\}$ e $I = \emptyset$, obteniéndose de forma inmediata el árbol $\mathcal{P} = \{f(x_1, \ldots, x_n)\}$ constituido por un solo nodo.

El conjunto de patrones lineales $\mathcal{P}$ construido por el algoritmo que acabamos de describir está ordenado por el orden $<$ (de generalidad relativa estricta) y cumple las propiedades que caracterizan un árbol definicional:

- *Propiedad de la raíz*: Efectivamente, $\mathit{pattern}(\mathcal{P}) = f(x_1, \ldots, x_n)$ es un elemento mínimo, ya que $f(x_1, \ldots, x_n) < \pi$ para todo $\pi \in \mathcal{P}$.

- *Propiedad de las hojas*: Por construcción, las hojas de $\mathcal{P}$ son los elementos del conjunto $L_f(\mathcal{R})$. Estos se han obtenido, a partir del elemento minimal $f(x_1, \ldots, x_n)$, instanciando cada posición inductiva, $k \in I$, con los correspondientes constructores planos (pasos 3.2 a 3.4). Por consiguiente, los elementos de $L_f(\mathcal{R})$ son instancias de sus respectivos ancestros en el árbol construido y, por consiguiente, los elementos maximales de $\mathcal{P}$

- *Propiedad de los padres*: Por construcción, a un nodo $\pi = f(\ldots, x_k, \ldots)$ dado le corresponden diferentes hijos $\pi_i = f(\ldots, c_i(x_1, \ldots, x_{m_i}), \ldots)$. Por consiguiente, a cada nodo $\pi_i \in \mathcal{P}$ distinto de $\mathit{pattern}(\mathcal{P})$ le corresponde un único padre $\pi < \pi_i$, no pudiendo existir otro padre π' tal que $\pi < \pi' < \pi_i$, ya que la variable x_k es sustituida por un constructor plano $c_i(x_1, \ldots, x_{m_i})$, cuando formamos π_i.

- *Propiedad inductiva*: Dado un nodo $\pi \in \mathcal{P} \setminus L_f(\mathcal{R})$, las posiciones inductivas son elementos de I. El paso 3.4 del algoritmo que construye $\mathcal{P}$ asegura que los hijos π_i de π cumplen la propiedad inductiva.

$\square$

Se debe observar que la Proposición 5.4.1 asegura que, dado un programa uniforme es posible construir un árbol definicional para cualquier función definida en el programa, haciendo uso del Algoritmo 2, partiendo de una posición inductiva arbitraria. Esto no sucede para los programas inductivamente secuenciales no uniformes (e.g., es imposible construir el árbol definicional de la función "$\leq$" definida en el Ejemplo 5, si seleccionamos como posición inductiva el segundo argumento).

Ejemplo 29 *El Algoritmo 2 construye el árbol definicional de la Figura 5.4 para la función f definida en el programa del Ejemplo 23. La figura se ha obtenido concretando la regla de selección de posiciones inductivas, de modo que seleccionamos la posición más a la izquierda de entre las del conjunto I.*

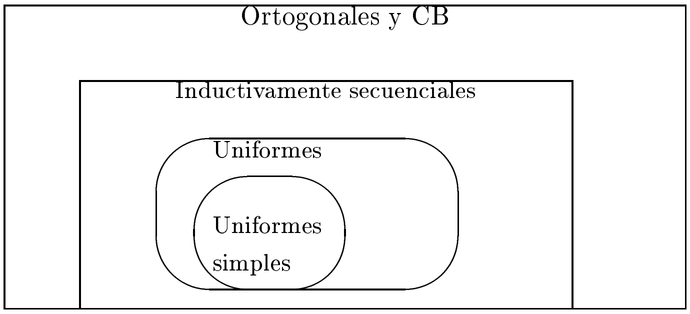

Figura 5.3: Una clasificación de los programas ortogonales y CB.

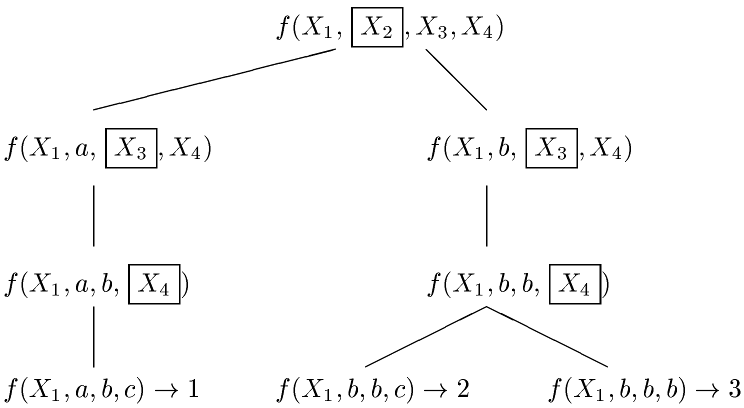

Figura 5.4: Arbol definicional para la función "f" definida en el Ejemplo 23.

Nótese que el número de niveles y posiciones inductivas del árbol definicional es igual al número de posiciones inductivas que aparecen en las lhs's de las reglas que definen f. Por construcción, este es un hecho generalizable a todos los árboles definicionales asociados a los programas uniformes. Ahora debe de quedar clara la razón por la que elegimos el nombre de "posición inductiva" para las posiciones, dentro del patrón de una regla del programa, en las que aparecen términos constructores planos. Como hemos visto, estas posiciones se corresponden con las posiciones inductivas del árbol definicional.

En lo que sigue, diremos que un árbol definicional se construye de la *forma estándar* cuando en el Algoritmo 2 se concreta la regla de selección de posiciones inductivas de modo que seleccionamos la posición más a la izquierda de entre las del conjunto I (i.e., seleccionamos de izquierda a derecha los argumentos que son posiciones inductivas de las funciones definidas por el programa).

5.5 Conclusiones.

En este capítulo, partiendo de la clase de programas uniformes de [119, 121], hemos caracterizado formalmente los programas uniformes como una subclase de los inductivamente secuenciales (Proposición 5.4.1). Esta caracterización nos va a permitir realizar, en el próximo capítulo, un estudio formal y detallado de la relación existente entre la estrategia de *narrowing* necesario [23] y la estrategia de *narrowing* perezoso presentadas en el Apartado 2.9.3.

También hemos discutido alguna de las ventajas de la utilización del *narrowing* perezoso sobre programas uniformes (e.g., la eliminación de los puntos de vuelta atrás asociados a los diferentes redexes demandados, en un mismo término, por diferentes reglas del programa), habiendo puesto de manifiesto que esa pretendida ventaja no se cumplen en general, como se aprecia en el Ejemplo 30 del Capítulo 6.

Por último, comentar que el estudio del Ejemplo 26 nos ha permitido demostrar que el algoritmo de transformación de programas ortogonales a programas uniformes presentado en [120] no es correcto en el caso general.

Capítulo 6

Estrategias de Evaluación Perezosa en Programas Uniformes.

En este apartado estudiamos la equivalencia entre las estrategias de *narrowing* perezoso y *narrowing* necesario para programas uniformes. Como resultado de este estudio, definimos un refinamiento de la estrategia de *narrowing* perezoso que denominamos *narrowing perezoso uniforme*, para el que probamos sus propiedades de corrección y completitud. Comenzamos con una serie de resultados sobre programas uniformes simples que servirán como punto de partida.

6.1 Resultados para Programas Uniformes Simples.

Zartmann ha establecido la correspondencia entre las derivaciones que respetan una estrategia de *narrowing* necesario en un programa inductivamente secuencial $\mathcal{R}$ y las que respetan una estrategia perezosa en el correspondiente programa transformado $\mathcal{U}_Z(\mathcal{R})$.

Teorema 6.1.1 *[205] Sea $\mathcal{R}$ un programa inductivamente secuencial y $\mathcal{U}_Z(\mathcal{R})$ el correpondiente programa uniforme simple. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$. Existe una derivación de* narrowing *necesario $t \overset{\sigma}{\rightsquigarrow}_{NN}^{*} s$ en $\mathcal{R}$ a una hnf s si y sólo si existe una derivación de* narrowing *perezoso $t \overset{\sigma}{\rightsquigarrow}_{LN}^{*} s$ en $\mathcal{U}_Z(\mathcal{R})$.*

El resultado anterior puede entenderse como un resultado de corrección y completitud de la transformación $\mathcal{U}_Z$, en el sentido de que el programa uniforme simple transformado $\mathcal{U}_Z(\mathcal{R})$ tiene idéntica semántica, cuando se ejecuta empleando *narrowing* perezoso, que la que caracteriza al programa original $\mathcal{R}$ cuando se ejecuta empleando la estrategia de *narrowing* necesario.

El siguiente corolario establece la relación existente entre las derivaciones de *narrowing* necesario y *narrowing* perezoso en un programa uniforme simple.

Corolario 6.1.2 *Sea $\mathcal{R}$ un programa uniforme simple. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$ y s un término en hnf. Existe una derivación de* narrowing *necesario* $t \overset{\sigma}{\leadsto}_{NN}^{*} s$ *en $\mathcal{R}$ si y sólo si existe una derivación de* narrowing *perezoso* $t \overset{\sigma}{\leadsto}_{LN}^{*} s$ *en $\mathcal{R}$.*

Prueba. Inmediata, por el Teorema 6.1.1 y el hecho de que $\mathcal{U}_Z(\mathcal{R}) = \mathcal{R}$. □

En el caso de programas uniformes, esta correspondencia en general no es biunívoca, como muestra el siguiente ejemplo.

Ejemplo 30 *Sea el programa uniforme*

$$\begin{array}{lll} R_1: & f(X,a,b) & \to 1 \\ R_2: & f(X,a,c) & \to 2 \\ R_3: & g(a) & \to a \\ R_4: & g(b) & \to b \end{array}$$

Para el término $t \equiv f(g(X), g(Y), g(Z))$, existen las siguientes derivaciones empleando la estrategia de narrowing *perezoso:*

$$\begin{array}{lll} f(g(X), g(Y), g(Z)) & \overset{[2,R_3,\{Y/a\}]}{\leadsto}_{LN} & f(g(X), a, g(Z)) \\ & \overset{[3,R_4,\{Z/b\}]}{\leadsto}_{LN} & f(g(X), a, b) \\ & \overset{[\Lambda,R_1,id]}{\leadsto}_{LN} & 1, \end{array}$$

y

$$\begin{array}{lll} f(g(X), g(Y), g(Z)) & \overset{[3,R_4,\{Z/b\}]}{\leadsto}_{LN} & f(g(X), g(Y), b) \\ & \overset{[2,R_3,\{Y/a\}]}{\leadsto}_{LN} & f(g(X), a, b) \\ & \overset{[\Lambda,R_1,id]}{\leadsto}_{LN} & 1, \end{array}$$

mientras que solamente es posible una derivación empleando narrowing *necesario (cuando empleamos el árbol definicional para la función "f" que aparece en la Figura 6.1):*

$$\begin{array}{lll} f(g(X), g(Y), g(Z)) & \overset{[2,R_3,\{Y/a\}]}{\leadsto}_{NN} & f(g(X), a, g(Z)) \\ & \overset{[3,R_4,\{Z/b\}]}{\leadsto}_{NN} & f(g(X), a, b) \\ & \overset{[\Lambda,R_1,id]}{\leadsto}_{NN} & 1. \end{array}$$

Así pues, la estrategia de narrowing *perezoso, incluso cuando se aplica a programas uniformes, sigue generando derivaciones redundantes (si bien, ambas computan idénticos resultados con las respuestas correspondientes y explotando únicamente posiciones necesarias).*

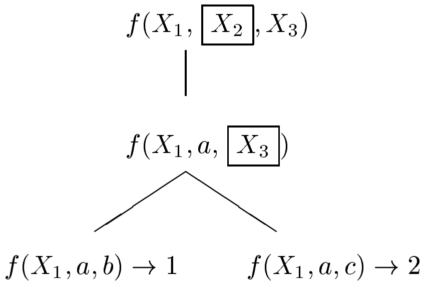

Figura 6.1: Arbol definicional para la función "f" del Ejemplo 30.

El Ejemplo 30 pone de manifiesto una cierta ineficiencia (ya comentada) de la estrategia de *narrowing* perezoso frente a la de *narrowing* necesario. La causa de ésta radica en el hecho de que, al emplear la estrategia de *narrowing* perezoso sobre programas uniformes, varias reglas que definen una misma función pueden demandar la evaluación de posiciones diferentes de un término. Este era el caso del término $t \equiv f(g(X), g(Y), g(Z))$, para el que se demandan las posiciones 2 y 3, asociadas con las correspondientes posiciones inductivas de las reglas que definen f. En los programas uniformes simples, dado que sólo puede aparecer una posición inductiva en cada lhs de una regla (i.e., un argumento con un término constructor plano y en la misma posición para todas las reglas que definen una determinada función), la estrategia de *narrowing* perezoso sólo demandará una posición de un término t. Esta posición será una posición necesaria para la evaluación del término t, de lo cual se deriva que los pasos de *narrowing* perezoso y de *narrowing* necesario deben coincidir en programas uniformes simples. Este resultado se presenta formalmente en la siguiente proposición, que establece la completa equivalencia entre las estrategias de *narrowing* necesario y de *narrowing* perezoso para programas uniformes simples.

Proposición 6.1.3 *Sea $\mathcal{R}$ un programa uniforme simple. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$ y $\mathcal{P}$ el árbol definicional de f. Entonces $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t)$.*

Prueba. Por inducción sobre la complejidad del término $t \equiv f(t_1, \ldots, t_n)$, medida por el número de símbolos de función definida y constructores que aparecen en t. Las substituciones se consideran módulo renombramiento de variables y restringidas a las variables del término t.

1. Caso base ($n = 1$): En este caso, no aparecen en t otros símbolos de función definida o constructores que el símbolo f que lo encabeza. Siguiendo la Definición 2.9.20 de la estrategia de *narrowing* necesario, λ, distinguimos las siguientes posibilidades.

   (a) *pattern*$(\mathcal{P}) = \pi$ es una hoja. En este caso, la función f está definida por una única regla $R \equiv f(x_1, \ldots, x_n) \to r$ y es inmediato comprobar que $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t) = \{\langle \Lambda, R, id \rangle\}$.

(b) $pattern(\mathcal{P}) = \pi$ es una rama. Dado que todos los argumentos del término t son necesariamente variables para la única posición inductiva o de $\mathcal{P}$ se cumple que $t|_o \equiv x$.

Por ser $\mathcal{R}$ uniforme simple, los subárboles $\mathcal{P}_k$ que cuelgan de la posición inductiva o son tales que $pattern(\mathcal{P}_k) = l_k$ es una hoja (i.e., la lhs de una de las m reglas $R_k \equiv l_k \to r_k$ que definen f) y por lo tanto $\lambda(\tau_k(t), \mathcal{P}_k) = \{(\Lambda, R_k, id)\}$, donde $\tau_k = \{x/c_k(x_1, \ldots, x_n)\}$. Aplicando la definición de estrategia de *narrowing* necesario, λ, es inmediato derivar que $\lambda(t, \mathcal{P}) = \bigcup_{k=1}^{m}\{\langle\Lambda, R_k, \tau_k\rangle\}$.

Por otro lado, $\lambda_{lazy} = \bigcup_{k=1}^{m} \lambda\_(t, \Lambda, k)$. Por ser $\mathcal{R}$ uniforme simple, las reglas que definen f tienen todos sus argumentos variables, salvo el situado en la posición inductiva o, que presenta el constructor plano $c_k(x_1, \ldots, x_n)$. Por consiguiente, el problema de unificación lineal planteado es tal que $\mathsf{LU}(\langle l_k, t\rangle) = (\textsc{Succ}, \tau_k)$, módulo renombramiento de variables cuando nos restringimos a las variables del término t. De esta forma, tenemos que $\lambda\_(t, \Lambda, k) = \{\langle\Lambda, R_k, \tau_k\rangle\}$ y concluimos que $\lambda_{lazy}(t) = \lambda(t, \mathcal{P})$.

2. Caso inductivo ($n > 1$): Siguiendo la definición de la estrategia de *narrowing* necesario, λ, distinguimos las siguientes posibilidades.

(a) $pattern(\mathcal{P}) = \pi$ es una hoja. En este caso, la función f está definida por una única regla $R \equiv f(x_1, \ldots, x_n) \to r$ y $\pi = f(x_1, \ldots, x_n)$.

Por definición, $\lambda(t, \mathcal{P}) = \{\langle\Lambda, R, id\rangle\}$ y $\pi \le t$. Por lo tanto, existe una substitución σ tal que $t = \sigma(\pi)$. Entonces, el problema de unificación lineal planteado entre los términos π y t debe de tener éxito, de forma que $\mathsf{LU}(\langle\pi, t\rangle) = (\textsc{Succ}, id)$, cuando la substitución unificadora más general σ se restringe a las variables del término t y se toma módulo renombramiento de variables. Así pues, $\lambda_{lazy}(t) = \{\langle\Lambda, R, id\rangle\}$.

(b) $pattern(\mathcal{P}) = \pi$ es una rama. Ahora distinguimos entre las siguientes posibilidades, según que sobre la posición inductiva o de $\mathcal{P}$ aparezca en t una variable, un término encabezado por un constructor o un término encabezado por un símbolo de función definida:

i. $t|_o \equiv x$.

En este caso, $\langle p, R_k, \sigma \circ \tau_k\rangle \in \lambda(t, \mathcal{P})$, donde $R_k \equiv l_k \to r_k$ es una de las m reglas que definen f, si $\langle p, R_k, \sigma\rangle \in \lambda(\tau_k(t), \mathcal{P}_k)$, donde $\tau_k = \{x/c_k(x_1, \ldots, x_{n_k})\}$. Dado que no puede aplicarse la hipótesis de inducción, ya que en el cómputo de $\lambda(\tau_k(t), \mathcal{P}_k)$ la complejidad del término $\tau_k(t)$ ha aumentado, procedemos de manera análoga a la del caso base (b). Concluimos que $\lambda_{lazy}(t) = \lambda(t, \mathcal{P}) = \bigcup_{k=1}^{m}\{\langle\Lambda, R_k, \tau_k\rangle\}$.

ii. $t|_o \equiv c(t_1, \ldots, t_n)$.

Razonamos de manera similar al caso anterior, pero ahora t sólo puede unificar con una de las reglas R_k que definen f, aquélla cuya lhs, l_k, cumple que $\mathcal{H}ead(l_k|_o) = c$. Así que, $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t) = \{\langle\Lambda, R_k, id\rangle\}$.

iii. $t|_o \equiv g(t_1, \ldots, t_n)$.
Entonces, $\langle o.p, R, \sigma\rangle \in \lambda(t, \mathcal{P})$ si $\langle p, R, \sigma\rangle \in \lambda(t|_o, \mathcal{P}')$, donde $\mathcal{P}'$ es el árbol definicional asociado a la operación g.
Por ser $\mathcal{R}$ un programa uniforme simple, en todas las lhs's l_k de las reglas $R_k \equiv l_k \to r_k$ que definen f aparece un término constructor plano en la posición inductiva o, siendo el resto de los argumentos variables. Al evaluar $\lambda_{lazy}(t)$, obtenemos que $\mathsf{LU}(\langle l_k, t\rangle) = (\textsc{Demand}, \{o\})$, cualquiera que sea la lhs l_k. Además, por ser $\mathcal{R}$ uniforme, Λ no puede ser una posición perezosa de t, ya que en tal caso la posición o no habría sido demandada. Por lo tanto, para computar $\lambda_{lazy}(t)$ es necesario computar previamente $\lambda_{lazy}(t|_o)$, cumpliéndose que si $\langle p, R, \sigma\rangle \in \lambda_{lazy}(t|_o)$ entonces $(o.p, R, \sigma) \in \lambda_{lazy}(t)$. Naturalmente, en cualquier circunstancia, si $\langle o.p, R, \sigma\rangle \in \lambda_{lazy}(t)$ entonces $\langle p, R, \sigma\rangle \in \lambda_{lazy}(t|_o)$. Por hipótesis de inducción $\lambda(t|_o, \mathcal{P}') = \lambda_{lazy}(t|_o)$, luego $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t)$.

$\square$

Haciendo uso de la Proposición 6.1.3, es inmediato establecer la siguiente relación entre derivaciones que respetan la estrategia de *narrowing* perezoso y las que respetan la estrategia de *narrowing* necesario.

Corolario 6.1.4 *Sea $\mathcal{R}$ un programa uniforme simple. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$ y s un término cualquiera. Existe una derivación por* narrowing *necesario $t \overset{\sigma}{\leadsto}_{NN}^{*} s$ en $\mathcal{R}$ si y sólo si existe una derivación por* narrowing *perezoso $t \overset{\sigma}{\leadsto}_{LN}^{*} s$ en $\mathcal{R}$. Estas derivaciones utilizan las mismas reglas sobre las mismas posiciones de t y sus descendientes.*

En este punto, es interesante observar que el resultado obtenido en el Corolario 6.1.4 es más fuerte que el establecido en el Corolario 6.1.2, que sólo asegura la equivalencia para derivaciones a una hnf.

6.2 Resultados para Programas Uniformes.

El problema cuando tratamos con programas uniformes es que la correspondencia entre derivaciones de *narrowing* necesario y *narrowing* perezoso no se puede obtener paso a paso, como vuelve a poner de manifiesto el siguiente ejemplo.

Ejemplo 31 *Sea el programa uniforme*

$$\begin{array}{lll} R_1: & f(a,b) & \to c \\ R_2: & g(c) & \to b \end{array}$$

Para el término $t \equiv f(X, g(Y))$ y sendos árboles definicionales de f y g construidos en la forma estándar, existe la siguiente derivación empleando narrowing *necesario:*

$$f(X, g(Y)) \overset{[2,R_2,\{X/a,Y/c\}]}{\leadsto_{NN}} f(a,b) \overset{[\Lambda,R_1,id]}{\leadsto_{NN}} c$$

mientras que la derivación empleando la estrategia de narrowing *perezoso es:*

$$f(X, g(Y)) \overset{[2,R_2,\{Y/c\}]}{\leadsto}_{LN} f(x, b) \overset{[\Lambda,R_1,\{X/a\}]}{\leadsto}_{LN} c$$

Sin embargo, existe todavía una relación interesante entre ambas estrategias, que formalizamos en la Proposición 6.2.3. Pero antes necesitamos un lema previo y precisar el concepto de *substitución adelantada*. Como ya se ha comentado, una característica de la estrategia de *narrowing* necesario es que adelanta substituciones, con ello queremos decir que utiliza los árboles definicionales asociados a un programa, no solamente como una guía para obtener posiciones demandadas de un término, sino que además guarda los enlaces necesarios para acceder a dicha posición. El resultado es que, aplicadas las substituciones adelantadas y una vez evaluado el término que ocupa la posición demandada, se reduce el indeterminismo que entraña toda derivación de *narrowing*. La estrategia de *narrowing* necesario utiliza el cómputo de las substituciones adelantadas para forzar pasos de *narrowing* sobre las posiciones necesarias, evitando el cómputo de algunas salidas redundantes que la estrategia de *narrowing* perezoso computa. La utilización de substituciones adelantadas por parte de la estrategia de *narrowing* necesario constituye la principal diferencia con la estrategia de *narrowing* perezoso. Por lo tanto, si deseamos establecer una relación entre ambas, es fundamental formalizar el concepto de substitución adelantada. El siguiente lema establece que cuando el paso de *narrowing* tiene lugar sobre la posición Λ, tanto la estrategia de *narrowing* necesario como la de *narrowing* perezoso computan la misma respuesta con la misma regla. Intuitivamente, dado que la parte de substitución adelantada en un paso de *narrowing* necesario es aquélla que no se computa en el correspondiente paso de *narrowing* perezoso, podemos interpretar este resultado en el sentido de que la estrategia de *narrowing* necesario no adelanta substituciones cuando el cómputo se realiza sobre la posición Λ.

Lema 6.2.1 *Sea $\mathcal{R}$ un programa uniforme y R una regla de $\mathcal{R}$. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$ y $\mathcal{P}$ el árbol definicional de f. Entonces, $\langle \Lambda, R, \sigma \rangle \in \lambda(t, \mathcal{P})$ si y sólo si $\langle \Lambda, R, \sigma \rangle \in \lambda_{lazy}(t)$.*

Prueba. En esta prueba consideraremos las substituciones módulo renombramiento de variables y restringidas a las variables del término t.

Supongamos que $\langle \Lambda, R, \sigma \rangle \in \lambda(t, \mathcal{P})$. Nótese que, debido a que $\langle \Lambda, R, \sigma \rangle \in \lambda(t, \mathcal{P})$, los argumentos de t sólo pueden ser variables o términos encabezados por un constructor. De otro modo, Λ no sería una posición necesaria de t. Siguiendo la Definición 2.9.20 de la estrategia de *narrowing* necesario λ, distinguimos las siguientes posibilidades.

1. $pattern(\mathcal{P}) = \pi$ es una hoja. En este caso, la función f está definida por una única regla $R \equiv f(x_1, \ldots, x_n) \to r$ y es inmediato comprobar que $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t) = \{\langle \Lambda, R, id \rangle\}$.

2. $pattern(\mathcal{P}) = \pi$ es una rama.

Por ser $\mathcal{R}$ un programa uniforme, existe una o más posiciones inductivas para cada árbol definicional $\mathcal{P}$. Dada una posición inductiva o_i de $\mathcal{P}$, tendremos dos posibilidades, $t|_{o_i} \equiv x_{k_i}$ o $t|_{o_i} \equiv c_{k_i}(s_1, \ldots, s_{n_i})$, ya que los argumentos del término t son variables o términos encabezados por un constructor. El Lema 5.2.2 asegura que, para cada lhs de las reglas que definen f, en las posiciones inductivas o_i aparece un término constructor plano. Sea $R_k \equiv l_k \to r_k$ una de las reglas que definen f, para las que $l_k|_{o_i} \equiv c_{k_i}(x_1, \ldots, x_{n_i})$. Es fácil comprobar que

$$\langle \Lambda, R_k, \tau_{k_q} \circ \ldots \circ \tau_{k_1} \rangle \in \lambda(t, \mathcal{P})$$

donde q es el número de posiciones inductivas de $\mathcal{P}$ y $\tau_{k_i} = id$ o bien $\tau_{k_i} = \{x_{k_i}/c_{k_i}(x_1, \ldots, x_{n_i})\}$.

Por otro lado, al computar $\lambda_{lazy}(t)$, el problema de unificación lineal planteado con la regla R_k es $\Gamma \equiv \langle l_k, t \rangle$. Una vez construida la correspondiente configuración inicial LU, (U_0, σ_0) podemos comprobar que existe la derivación LU:

$$(U_0, \sigma_0) \to^*_{\mathsf{LU}} (\{l_k|_{o_1} \downarrow_{o_1} t|_{o_1}, \ldots, l_k|_{o_q} \downarrow_{o_q} t|_{o_q}\}, id) \to^*_{\mathsf{LU}} (\emptyset, \tau_{k_q} \circ \ldots \circ \tau_{k_1})$$

si sólo tenemos en consideración las variables del término t y que los elementos del tipo $c_{k_i}(x_1, \ldots, x_{n_i}) \downarrow_{o_i} c_{k_i}(s_1, \ldots, s_{n_i})$ se reducen a la substitución id. Entonces, $\mathsf{LU}(\langle l_k, t \rangle) = (\textsc{Succ}, \tau_{k_q} \circ \ldots \circ \tau_{k_1})$, de forma que $\langle \Lambda, R_k, \tau_{k_q} \circ \ldots \circ \tau_{k_1} \rangle \in \lambda_{lazy}(\tau(t))$.

La prueba en el sentido de dirección opuesto es completamente análoga. □

El Lema 6.2.1 justifica la siguiente formalización del concepto de substitución adelantada en un paso de *narrowing* necesario, que se ilustra a continuación.

Definición 6.2.2 (Substitución Adelantada)
Sea $\mathcal{R}$ un programa inductivamente secuencial. Sea t un término encabezado por un símbolo de función definida y $\mathcal{P}$ un árbol definicional con $pattern(\mathcal{P}) = \pi$ tal que $\pi \leq t$. Las partes de substitución adelantada en el cómputo de $\lambda(t, \mathcal{P})$ se obtienen mediante la aplicación α, de términos y árboles definicionales a substituciones, definida inductivamente como sigue:

$$\alpha(t,\mathcal{P}) \ni \begin{cases} id & \texttt{si existe } \textit{una regla } R \texttt{ y } \textit{una substitución } \sigma \\ & \texttt{tales que } \ \langle \Lambda, R, \sigma \rangle \in \lambda(t,\mathcal{P}), \\ \tau' \circ \tau & \texttt{si } t|_o = x \in \mathcal{X},\ \tau = \{x/c_i(x_1,\ldots,x_n)\} \\ & \texttt{y } \tau' \in \alpha(\tau(t), \mathcal{P}_i); \\ \tau' \circ id & \texttt{si } t|_o = c_i(t_1,\ldots,t_n) \texttt{ y } \tau' \in \alpha(t,\mathcal{P}_i); \\ \tau' \circ id & \texttt{si } t|_o = g(t_1,\ldots,t_n),\ g \in \mathcal{F} \texttt{ y } \tau' \in \alpha(t|_o, \mathcal{P}') \\ & \texttt{donde } \mathcal{P}' \textit{ es un árbol definicional para } g. \end{cases}$$

donde o es la posición inductiva de π, $\pi_i = \pi[c_i(x_1, \ldots, x_n)]_o \in \mathcal{P}$ un hijo de π, y $\mathcal{P}_i = \{\pi' \in \mathcal{P} \mid \pi_i \leq \pi'\}$ el árbol definicional donde todos los patrones son instancias de π_i.

Ejemplo 32 *Sea el programa uniforme*

$$\begin{array}{lll} R_1: & f(a,b) & \to c \\ R_2: & f(a,c) & \to b \\ R_3: & g(a) & \to c \\ R_4: & g(c) & \to b \end{array}$$

Para el término $t \equiv f(X, f(Y, g(Z)))$ y los árboles definicionales $\mathcal{P}$ de f y $\mathcal{P}'$ de g (construidos seleccionando las posiciones inductivas de izquierda a derecha) es fácil comprobar que $\langle 2.2, R_4, id \circ \{Z/c\} \circ id \circ \{Y/a\} \circ id \circ \{X/a\}\rangle \in \lambda(t, \mathcal{P})$. En este caso, la Definición 6.2.2 computa como substitución adelantada: $\tau = id \circ id \circ \{Y/a\} \circ id \circ \{X/a\} \in \alpha(t, \mathcal{P})$.

Intuitivamente, la parte de substitución adelantada τ se compone de los enlaces establecidos en el cómputo de $\lambda(t, \mathcal{P})$ antes de encontrar la ocurrencia demandada 2.2 y calcular: $\langle \Lambda, R_4, id \circ \{Z/c\}\rangle \in \lambda(f(a, f(a, g(c)))|_{2.2}, \mathcal{P}')$, que computa la parte de substitución no adelantada.

En el ejemplo anterior, puede comprobarse que, para el término lineal $t \equiv f(X, f(Y, g(Z)))$, $\langle 2.2, R_4, \{Z/c\}\rangle \in \lambda_{lazy}(t)$. Así pues, en ese caso, lo que computa la Definición 6.2.2 coincide con nuestro concepto intuitivo de substitución adelantada (aquélla parte de la substitución computada en un paso de *narrowing* necesario que no computa el correspondiente paso de *narrowing* perezoso). Sin embargo, el comportamiento de la aplicación α no siempre es tan reconfortante. Basta utilizar en los cómputos un término no lineal, como el término $s \equiv f(Z, f(Y, g(Z)))$, para que nos demos cuenta de que el concepto definido por la aplicación α no se corresponde exactamente con el concepto intuitivo de substitución adelantada.

Ejemplo 33 *Para el programa uniforme del Ejemplo 32 y el término $s \equiv f(Z, f(Y, g(Z)))$ obtenemos que $\langle 2.2, R_3, id \circ id \circ id \circ \{Y/a\} \circ id \circ \{Z/a\}\rangle \in \lambda(t, \mathcal{P})$. En este caso, la Definición 6.2.2 computa como substitución adelantada: $\tau = id \circ id \circ \{Y/a\} \circ id \circ \{Z/a\} \in \alpha(t, \mathcal{P})$, mientras que $\langle 2.2, R_3, \{Z/a\}\rangle \in \lambda_{lazy}(s)$. Vemos que el correspondiente paso de* narrowing *perezoso no "desecha" alguno de los enlaces establecidos en el cómputo de $\lambda(s, \mathcal{P})$ antes de encontrar la ocurrencia demandada* 2.2.

Sin embargo, esto no es un problema ya que lo que pretendemos con la Definición 6.2.2 es disponer de una herramienta operacional con la que establecer una relación entre un paso de *narrowing* necesario y el correspondiente paso de *narrowing* perezoso.

Proposición 6.2.3 *Sea $\mathcal{R}$ un programa uniforme y R una regla de $\mathcal{R}$. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$ y $\mathcal{P}$ el árbol definicional de f. Entonces, $\langle p, R, \sigma \circ \tau\rangle \in \lambda(t, \mathcal{P})$ si y sólo si $\langle p, R, \sigma\rangle \in \lambda_{lazy}(\tau(t))$, donde $\tau \in \alpha(t, \mathcal{P})$ es la parte de substitución adelantada en el cómputo de $\lambda(t, \mathcal{P})$.*

Prueba. Por inducción sobre la complejidad del término $t \equiv f(t_1, \ldots, t_n)$, medida por el número de símbolos de función definida que aparecen en t, e inducción estructural sobre el árbol definicional $\mathcal{P}$. Las substituciones se consideran módulo renombramiento de variables y restringidas a las variables del término t.

1. Caso base ($n = 1$): En este caso, no aparecen en t otros símbolos de función que f, así que los argumentos de t son variables o términos constructores. Por consiguiente, la única posibilidad para que $\langle p, R, \sigma \circ \tau\rangle \in \lambda(t, \mathcal{P})$ es que $p = \Lambda$. El Lema 6.2.1 y la Definición 6.2.2 de substitución adelantada nos aseguran que $\langle p, R, \sigma \circ \tau\rangle \in \lambda_{lazy}(t)$ y que $\tau = id$ es la parte de substitución adelantada (i.e., no hay substitución adelantada), por lo que el enunciado se cumple trivialmente.

2. Caso inductivo ($n > 1$): Siguiendo la definición de la estrategia de *narrowing* necesario, λ, distinguimos los siguientes casos.

   (a) $pattern(\mathcal{P}) = \pi$ es una hoja. En este caso, la función f está definida por una única regla $R \equiv f(x_1, \ldots, x_n) \to r$ y es inmediato comprobar que $\lambda(t, \mathcal{P}) = \lambda_{lazy}(t) = \{\langle \Lambda, R, id\rangle\}$. La idempotencia de la substitución identidad nos permite afirmar que $\langle p, R, id \circ id\rangle \in \lambda(t, \mathcal{P})$ si y sólo si $\langle p, R, id\rangle \in \lambda_{lazy}(t)$.

   (b) $pattern(\mathcal{P}) = \pi$ es una rama. Sea o la posición inductiva de π. Ahora distinguimos entre las siguientes posibilidades:

      i. $t|_o \equiv x$.
      Sea $\tau = x/c_i(x_1, \ldots, x_{n_i})$. Entonces $\langle p, R, \sigma' \circ \tau' \circ \tau\rangle \in \lambda(t, \mathcal{P})$ si $\langle p, R, \sigma' \circ \tau'\rangle \in \lambda(\tau(t), \mathcal{P}_i)$, donde τ' es la parte substitución adelantada en el cómputo de $\lambda(\tau(t), \mathcal{P}_i)$. Por hipótesis de inducción $\langle p, R, \sigma'\rangle \in \lambda_{lazy}(\tau'(\tau(t)))$, donde por Definición 6.2.2, $\tau' \circ \tau$ es la parte substitución adelantada en el cómputo de $\lambda(t, \mathcal{P})$.

      ii. $t|_o \equiv c(t_1, \ldots, t_n)$.
      Si $\langle p, R, \sigma' \circ \tau'\rangle \in \lambda(t, \mathcal{P}_i)$, siendo τ' la parte de substitución adelantada en el cómputo de $\lambda(t, \mathcal{P}_i)$, entonces $\langle p, R, \sigma' \circ \tau' \circ id\rangle \in \lambda(t, \mathcal{P})$. Por hipótesis de inducción $\langle p, R, \sigma'\rangle \in \lambda_{lazy}(\tau'(t))$, donde por la Definición 6.2.2, $\tau' \circ id = \tau'$ es la parte de substitución adelantada en el cómputo de $\lambda(t, \mathcal{P})$.

      iii. $t|_o \equiv g(t_1, \ldots, t_n)$.
      Entonces, $\langle o.p, R, \sigma' \circ \tau' \circ id\rangle \in \lambda(t, \mathcal{P})$ si $\langle p, R, \sigma' \circ \tau'\rangle \in \lambda(t|_o, \mathcal{P}')$, donde $\mathcal{P}'$ es el árbol definicional asociado a la operación g y τ' es la parte de substitución adelantada en el cómputo de $\lambda(t|_o, \mathcal{P}')$. Al evaluar $\lambda_{lazy}(t)$, por ser $\mathcal{R}$ uniforme, obtenemos que $\mathsf{LU}(\langle l_k, t\rangle) = (\textsc{Demand}, P)$, cualquiera que sea la lhs l_k. Además, $o \in P$ y no existe una posición $u < o$ que sea una posición perezosa de t, ya que entonces o no habría sido demandada. Por lo tanto para computar $\lambda_{lazy}(t)$ es necesario computar previamente $\lambda_{lazy}(t|_o)$, cumpliéndose que $\langle p, R, \sigma\rangle \in \lambda_{lazy}(t|_o)$ si y sólo si $\langle o.p, R, \sigma\rangle \in \lambda_{lazy}(t)$. Por hipótesis de inducción $\langle p, R, \sigma'\rangle \in \lambda_{lazy}(\tau'(t|_o))$ y por tanto $\langle o.p, R, \sigma'\rangle \in \lambda_{lazy}(\tau'(t))$, donde por la

Definición 6.2.2, $\tau' \circ id = \tau'$ es la parte de substitución adelantada en el cómputo de $\lambda(t, \mathcal{P})$.

□

Ahora es fácil establecer el análogo de la Proposición 6.1.3 para los pasos de *narrowing* perezoso y los pasos de *narrowing* necesario sobre programas uniformes.

Corolario 6.2.4 *Sea $\mathcal{R}$ un programa uniforme y R una regla de $\mathcal{R}$. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$ y $\mathcal{P}$ el árbol definicional de f. Entonces, existe el paso de* narrowing *necesario $t \overset{[p,R,\sigma\circ\tau]}{\leadsto}_{NN} s$ si y sólo si existe el paso de* narrowing *perezoso $\tau(t) \overset{[p,R,\sigma]}{\leadsto}_{LN} s$, donde $\tau \in \alpha(t, \mathcal{P})$ es la parte de substitución adelantada en la computación $\lambda(t, \mathcal{P})$.*

Prueba. Si $t \overset{[p,R,\sigma\circ\tau]}{\leadsto}_{NN} s$, por definición de paso de *narrowing*, $s \equiv \sigma(\tau(t[r]_p))$. Por la Proposición 6.2.3, $\langle p, R, \sigma\rangle \in \lambda_{lazy}(\tau(t))$. Por tanto puede darse el paso de *narrowing* $\tau(t) \overset{[p,R,\sigma]}{\leadsto}_{LN} \sigma(\tau(t)[r]_p)$. Dado que la substitución adelantada τ está restringida a las variables del término t, se cumple que

$$\sigma(\tau(t)[r]_p) = \sigma(\tau(t)[\tau(r)]_p) = \sigma(\tau(t[r]_p)) \equiv s.$$

La prueba en el otro sentido puede realizarse mediante un razonamiento completamente similar al anterior. □

La idea intuitiva subyacente en el enunciado del Corolario 6.2.4 se ilustra mediante el siguiente ejemplo.

Ejemplo 34 *Volviendo al Ejemplo 31 y centrándonos en el primer paso de la derivación de* narrowing *necesario para el término $t \equiv f(X, g(Y))$, tenemos que:*

$$f(X, g(Y)) \overset{[2,R_2,id\circ\{Y/c\}\circ id\circ\{X/a\}]}{\leadsto}_{NN} f(a, b)$$

donde la respuesta computada se ha representado en forma canónica, para poner de manifiesto la relación existente entre este paso y el correspondiente paso empleando la estrategia de narrowing *perezoso:*

$$f(X, g(Y)) \overset{[2,R_2,\{Y/c\}]}{\leadsto}_{LN} f(X, b).$$

La parte de subtitución adelantada en el paso de narrowing *necesario ha sido $\tau \equiv id\circ\{x/a\} \in \alpha(t, \mathcal{P})$, donde $\mathcal{P}$ es el árbol definicional de f. Si aplicamos esta subtitución τ al término t y damos un paso de* narrowing *perezoso, obtenemos:*

$$f(a, g(Y)) \overset{[2,R_2,\{Y/c\}]}{\leadsto}_{LN} f(a, b).$$

en concordancia con lo enunciado en el Corolario 6.2.4.

Naturalmente, la relación expresada por el Corolario 6.2.4 también se cumple para términos no lineales, a pesar de que el concepto intuitivo de substitución adelantada no se corresponde con lo que computa la aplicación α, como ilustra el siguiente ejemplo.

Ejemplo 35 *Para el programa uniforme del Ejemplo 32 y el término objetivo* $s \equiv f(Z, f(Y, g(Z)))$*, hemos visto que* $\langle 2.2, R_3, id \circ id \circ id \circ \{Y/a\} \circ id \circ \{Z/a\}\rangle \in \lambda(s, \mathcal{P})$ *donde* $\tau = id \circ \{Y/a\} \circ id \circ \{Z/a\}$ *es la parte de substitución adelantada computada por la aplicación* α*. Es fácil comprobar que, empleando el* narrowing *necesario, puede darse el siguiente paso:*

$$f(Z, f(Y, g(Z)))\overset{[2.2, R_3, id \circ id \circ id \circ \{Y/a\} \circ id \circ \{Z/a\}]}{\leadsto_{NN}} f(a, f(a, c))$$

donde, nuevamente, la respuesta computada se representa en forma canónica, para poner de manifiesto la relación existente entre este paso y el correspondiente paso empleando la estrategia de narrowing *perezoso cuando aplicamos la subtitución* τ *al término* s*. El paso de* narrowing *perezoso correspondiente es:*

$$f(a, f(a, g(a)))\overset{[2.2, R_3, id]}{\leadsto_{LN}} f(a, f(a, c))$$

en concordancia con lo enunciado en el Corolario 6.2.4.

Adviértase que la Proposición 6.2.3 fue demostrada para un árbol definicional fijado arbitrariamente. Terminamos este apartado haciendo notar que existe una identidad entre los pasos de *narrowing* perezoso y los pasos de *narrowing* necesario cuando seleccionamos un árbol definicional adecuado. Se cumple que, dado un término t, para cada paso de *narrowing* perezoso dado sobre una posición p de t, con una regla R y una substitución σ, existe un árbol definicional que permite dar un paso de *narrowing* necesario sobre la misma posición p de t, con la misma regla R y substitución σ. Motivamos este resultado mediante el siguiente ejemplo.

Ejemplo 36 *Consideremos nuevamente el programa uniforme del Ejemplo 31:*

$$\begin{array}{lll} R_1: & f(a, b) & \to c \\ R_2: & g(c) & \to b \end{array}$$

Para el término $t \equiv f(X, g(Y))$*, puede darse el siguiente paso de* narrowing *perezoso*

$$f(X, g(Y))\overset{[2, R_2, \{Y/c\}]}{\leadsto_{LN}} f(X, b).$$

Ahora bien, si empleamos los árboles definicionales que se muestran en la Figura 6.2, en lugar de emplear árboles definicionales para f *y* g *construidos en la forma estándar, puede darse el mismo paso empleando* narrowing *necesario:*

$$f(X, g(Y))\overset{[2, R_2, \{Y/c\}]}{\leadsto_{NN}} f(X, b).$$

Así pues, existe una representación adecuada de los árboles definicionales para la cual la estrategia de *narrowing* necesario coincide con la de *narrowing* perezoso en programas uniformes.

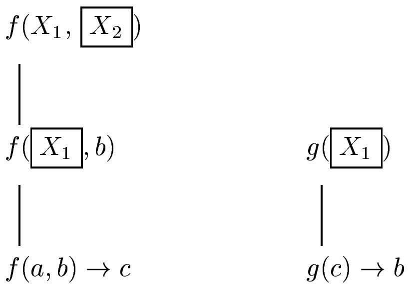

Figura 6.2: Arboles definicionales para las funciones "f" y "g" delEjemplo 36.

Proposición 6.2.5 *Sean $\mathcal{R}$ un programa uniforme y R una regla de $\mathcal{R}$. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$. Si $\langle p, R, \sigma\rangle \in \lambda_{lazy}(t)$, entonces existe un árbol definicional $\mathcal{P}$ de f tal que $\langle p, R, \sigma\rangle \in \lambda(t, \mathcal{P})$.*

Prueba. Realizamos esta prueba mostrando la posibilidad de construir un árbol definicional $\mathcal{P}$ para el que se cumple el enunciado. Emplearemos inducción sobre la estructura de la posición p.

1. Caso base ($p = \Lambda$): Por el Lema 6.2.1, $\langle \Lambda, R, \sigma\rangle \in \lambda_{lazy}(t)$ si y sólo si $\langle \Lambda, R, \sigma\rangle \in \lambda(t, \mathcal{P})$, cualquiera que sea el árbol definicional $\mathcal{P}$ de f. Por lo tanto, el enunciado de la proposición se cumple de forma inmediata.

2. Caso inductivo ($p > \Lambda$): Por ser $\mathcal{R}$ un programa uniforme, la posición perezosa p es de la forma: $p = i.q$, donde $i \in \{1, \ldots, n\}$ es una posición inductiva de f. Dado que según el Algoritmo 2, la posición inductiva sobre la que se comienza a formar el árbol definicional en un programa uniforme puede ser elegida de forma arbitraria, tiene sentido considerar que $\mathcal{P}$ es el árbol definicional resultante de elegir como primera posición inductiva la posición i.

   Para que p sea una posición perezosa, debe cumplirse que $t|_i \equiv g(t_1, \ldots, t_m)$. Por otro lado, como se ha argumentado en el caso inductivo (b-iii) de la Proposición 6.2.3, por ser $\mathcal{R}$ un programa uniforme tenemos que $\langle q, R, \sigma\rangle \in \lambda_{lazy}(t|_i)$ si y sólo si $\langle i.q, R, \sigma\rangle \in \lambda_{lazy}(t)$. Por hipótesis de inducción, existe un árbol definicional $\mathcal{P}'$ de g tal que $\langle q, R, \sigma\rangle \in \lambda(t|_i, \mathcal{P}')$. Por la definición de estrategia de *narrowing* necesario λ, si $t|_i \equiv g(t_1, \ldots, t_m)$ y $\langle q, R, \sigma\rangle \in \lambda(t|_i, \mathcal{P}')$ entonces $\langle i.q, R, \sigma\rangle \in \lambda(t, \mathcal{P})$.

   Así pues existe posiblemente un conjunto de árboles definicionales, aquellos que tienen como primera posición inductiva a la posición i, para los que se cumplirá el enunciado.

$\square$

El siguiente corolario expresa el resultado anterior en términos de pasos de *narrowing*.

Corolario 6.2.6 *Sean $\mathcal{R}$ un programa uniforme y R una regla de $\mathcal{R}$. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$. Si $t \overset{[p,R,\sigma]}{\leadsto}_{LN} s$, entonces existe un árbol definicional $\mathcal{P}$ de f tal que $t \overset{[p,R,\sigma]}{\leadsto}_{NN} s$.*

6.3 Narrowing Perezoso Uniforme.

Los resultados del apartado anterior, en especial los que se expresan en las proposiciones 6.2.3 y 6.2.5, ponen de manifiesto que, para programas uniformes, las posiciones de un término computadas por la estrategia λ_{lazy} son *posiciones necesarias* computadas por la estrategia λ para dicho término. Esta importante observación permite introducir un refinamiento de la estrategia de *narrowing* perezoso, para el cual ésta puede probarse todavía completa. La idea intuitiva detrás de este refinamiento es que si las posiciones que computa la estrategia de *narrowing* perezoso son posiciones necesarias en el sentido de la estrategia λ, entonces no es preciso explotar todas ellas de una forma "don't know", basta con elegirlas de una forma "don't care", ya que eventualmente todas ellas deberán explotarse para poder alcanzar el resultado. Debido a las propiedades de completitud del *narrowing* necesario, el orden en el que sean explotadas podrá generar derivaciones distintas, pero sin afectar a los resultados ni a las respuestas obtenidas.

En [64, 65], Echahed ha introducido el concepto de estrategia de *narrowing uniforme* para TRS canónicos. Intuitivamente, este tipo de estrategia permite dar pasos de *narrowing* seleccionando solamente una posición del término a evaluar, lo que reduce drásticamente el espacio de búsqueda. Es en este mismo sentido que denominamos *narrowing perezoso uniforme* (ULN, del inglés *Uniform Lazy Narrowing*) al refinamiento de la estrategia de *narrowing* perezoso que consiste en seleccionar el subconjunto de las ternas asociadas a una posición perezosa (por ejemplo, la más a la izquierda) para dar los correspondientes pasos de *narrowing*. Formalizamos este concepto mediante la siguiente definición:

Definición 6.3.1 (Estrategia de Narrowing Perezoso Uniforme)
Sea $\mathcal{R}$ un programa uniforme. Definimos la estrategia de *narrowing* perezoso uniforme *como una función λ_{ulazy} que, aplicada a un término t, computa el siguiente conjunto de ternas $\langle p, R_k, \sigma\rangle$, siendo $p \in \mathcal{FP}os(t)$ una posición perezosa de t, $R_k \equiv (l_k \to r_k)$ una regla (renombrada aparte) de $\mathcal{R}$ y σ una substitución:*

$$
\begin{array}{lcl}
\lambda_{ulazy}(t) & = & \bigcup_{k=1}^{m} \lambda\_(t, \Lambda, R_k) \\
\lambda\_(t, p, R_k) & = & \texttt{si } \mathcal{H}ead(l_k) = \mathcal{H}ead(t|_p) \texttt{ entonces} \\
 & & \quad \texttt{en caso de que } \mathsf{LU}(\langle l_k, t_{|}p\rangle) = \\
 & & \quad \left\{ \begin{array}{ll} (\texttt{Succ}, \sigma): & \{\langle p, R_k, \sigma\rangle\} \\ (\texttt{Fail}, \emptyset): & \emptyset \\ (\texttt{Demand}, P): & \bigcup_{k=1}^{m} \lambda\_(t, p.q, R_k) \\ & \texttt{con } q = \mathit{select\_don't\_care}(P) \end{array} \right. \\
 & & \texttt{sino } \emptyset
\end{array}
$$

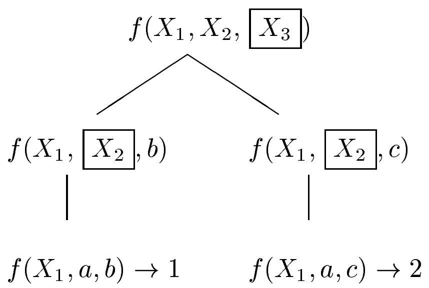

Figura 6.3: Arbol definicional para la función "f" del Ejemplo 37.

donde la función $select\_don't\_care(S)$ elige arbitrariamente un elemento del conjunto S.

Dado un término t y una regla $R \equiv (l \to r)$ del programa uniforme $\mathcal{R}$, decimos que $t \overset{[p,R,\sigma]}{\leadsto}_{ULN} \sigma(t[r]_p)$ es un *paso de narrowing uniforme*, si $\langle p, R, \sigma \rangle \in \lambda_{ulazy}(t)$.

La Definición 6.3.1 asegura que solamente se seleccionará, de forma "don't care", el subconjunto de las ternas del *narrowing* perezoso asociado a una posición perezosa. La estrategia que acabamos de introducir rompe la discrepancia entre el indeterminismo "don't care" en la selección inicial de los árboles definicionales que emplea la estrategia de *narrowing* necesario y el hecho de que las posiciones perezosas demandadas por la estrategia de *narrowing* perezoso se exploten siempre de forma "don't know".

Ejemplo 37 *Reconsideremos el programa uniforme del Ejemplo 30. Si empleamos el árbol definicional para la función "f" representado en la Figura 6.3, en lugar de emplear el que ilustra la Figura 6.1, puede obtenerse la siguiente derivación empleando* narrowing *necesario:*

$$\begin{array}{lll} f(g(X), g(Y), g(Z)) & \overset{[3,R_4,\{Z/b\}]}{\leadsto}_{NN} & f(g(X), g(Y), b) \\ & \overset{[2,R_3,\{Y/a\}]}{\leadsto}_{NN} & f(g(X), a, b) \\ & \overset{[\Lambda,R_1,id]}{\leadsto}_{NN} & 1. \end{array}$$

Podemos apreciar que esta derivación se corresponde paso a paso con la derivación de narrowing *perezoso "redundante", obtenida en el Ejemplo 30.*

El ejemplo anterior muestra que la derivación de *narrowing* perezoso que era "redundante" en el Ejemplo 30, se corresponde con la que calcularía la estrategia de *narrowing* necesario eligiendo la representación para el árbol definicional alternativo de f mostrada en la Figura 6.3. Por lo tanto, cada una de las derivaciones "redundantes" de *narrowing* perezoso que aparecen al computar un término en un programa uniforme están asociadas a una de las posibles derivaciones de *narrowing* necesario para un árbol definicional fijado.

Justificamos el refinamiento introducido mediante los siguientes resultados. El primero de ellos indica que para programas uniformes, fijado un término y un árbol definicional para la raíz de dicho término, la estrategia de *narrowing* necesario computa una única posición necesaria (a la que posiblemente hay asociadas varias ternas correspondientes a diferentes reglas del programa).

Proposición 6.3.2 *Sea $\mathcal{R}$ un programa uniforme. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$ y $\mathcal{P}$ un árbol definicional de f. Si $\lambda(t, \mathcal{P}) = \{\langle p_1, R_1, \sigma_1\rangle, \ldots, \langle p_k, R_k, \sigma_k\rangle\}$, entonces $p_1 = \ldots = p_k = p$.*

Prueba. Inmediata, apartir de la Definición 5.2.1 y la estructura de un programa uniforme (Lema 5.2.2). □

La Proposición 6.2.5 y la Proposición 6.3.2 aseguran que, para programas uniformes, las posiciones perezosas de un término t computadas por la estrategia λ_{lazy} son disjuntas, y los subconjuntos de ternas asociados, corresponden a pasos de *narrowing* necesario $\lambda(t, \mathcal{P})$ dados con uno de los posibles árboles definicionales $\mathcal{P}$ de f, siendo f el símbolo que encabeza el término t.

Proposición 6.3.3 *Sean $\mathcal{R}$ un programa uniforme y t un término. Si $\langle p, R, \sigma\rangle \in \lambda_{lazy}(t)$ y $\langle p', R', \sigma'\rangle \in \lambda_{lazy}(t)$ entonces $p = p'$ o bien $p \parallel p'$.*

Prueba. Inmediata, haciendo uso de la Proposición 6.2.5 y de la Proposición 6.3.2. □

La Proposición 6.3.2 y la Proposición 6.3.3 justifican la corrección del refinamiento propuesto: dado que la estrategia de *narrowing* necesario es correcta y completa para programas inductivamente secuenciales (de los que los programas uniformes son una subclase), al emplear la estrategia de *narrowing* perezoso basta explotar uno de los subconjuntos de ternas disjuntos asociados a las posiciones perezosas.

A modo de conclusión, podemos decir que la estrategia de *narrowing* perezoso uniforme es una estrategia "intermedia" entre la de *narrowing* necesario y la de *narrowing* perezoso, que tiene las buenas propiedades de evitar: i) el exceso de instanciación del *narrowing* necesario; ii) las derivaciones redundantes del *narrowing* perezoso. La Figura 6.4 sintetiza estas propiedades.

6.4 Equivalencia de las Estrategias de Evaluación Perezosa y Corrección del Narrowing Perezoso Uniforme.

Hasta el momento hemos establecido la relación existente entre 'pasos' correspondientes a las estrategias de *narrowing* perezoso y de *narrowing* necesario. A continuación investigaremos la relación existente entre derivaciones de longitud arbitraria. Aunque en este apartado nos ceñimos a las relaciones entre la estrategia de *narrowing* necesario y la de *narrowing* perezoso uniforme, queremos hacer notar que los principales resultados pueden extenderse a la estrategia de *narrowing* perezoso, si bien, en este caso, a una derivación de *narrowing* necesario podrán corresponderle varias derivaciones de *narrowing* perezoso (sólo distinguibles por el orden en el que se dan los pasos en la derivación, como se ilustra en la Figura 6.4).

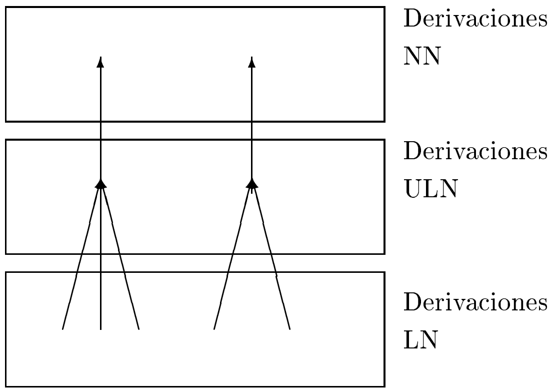

Figura 6.4: Relación entre las estrategias de evaluación perezosa sobre programas uniformes.

Nuestro interes primordial es probar la equivalencia entre estas estrategias y demostrar la corrección y completitud de la estrategia de *narrowing* perezoso uniforme introducida en el apartado anterior. El siguiente ejemplo muestra que, para árboles definicionales fijados arbitrariamente, estas estrategias no son equivalentes (en el sentido de que no computan la misma salida) a menos que evaluemos un término hasta alcanzar la hnf.

Ejemplo 38 *Sea el programa uniforme*

$$\begin{array}{lll} R_1: & f(c(X), a, b) & \to b \\ R_2: & g(c(Y)) & \to Y \\ R_3: & h(b) & \to b \end{array}$$

Para el término $t \equiv f(X, g(Z), h(W))$ existe una única derivación empleando narrowing *perezoso uniforme (seleccionando el subconjunto de ternas asociado a la posición perezosa más a la izquierda):*

$$f(X, g(Z), h(W)) \overset{[2,R_2,\{Z/c(Y)\}]}{\leadsto_{ULN}} f(X, Y, h(W)) \overset{[3,R_3,\{W/b\}]}{\leadsto_{LN}} f(X, Y, b),$$

que produce la salida $\langle f(X, Y, b), \{W/b, Z/c(Y)\}\rangle$. Por el contrario, la derivación obtenida construyendo los árboles definicionales de las funciones f, g y h en la forma estándar y empleando la estrategia de narrowing *necesario es:*

$$\begin{array}{lll} f(X, g(Z), h(W)) & \overset{[2,R_2,\{Z/c(Y_2)\}\circ\{X/c(X_4)\}]}{\leadsto_{NN}} & f(c(X_4), Y_2, h(W)) \\ & \overset{[3,R_3,\{W/b\}\circ\{Y_2/a\}]}{\leadsto_{NN}} & f(c(X_4), a, b), \end{array}$$

que produce la salida $\langle f(c(X_4), a, b), \{X/c(X_4), W/b, Z/c(a)\}\rangle$. Sin embargo, al dar el siguiente paso (que lleva los términos a una hnf), ambas derivaciones computan el mismo resultado con la misma respuesta (salvo renombramientos).

Hacemos notar también que la estrategia de narrowing *perezoso permite construir una derivación adicional en la que primero se da un paso con la regla R_3 y a continuación con la R_2.*

El siguiente teorema establece la equivalencia de la semántica operacional, para respuestas computadas, cuando empleamos las estrategias de *narrowing* necesario y *narrowing* perezoso uniforme.

Teorema 6.4.1 *Sea $\mathcal{R}$ un programa uniforme. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$. Entonces,*

1. *existe una derivación de* narrowing *necesario $\mathcal{D} \equiv (t \overset{\sigma}{\rightsquigarrow}_{NN}{}^{*} s)$, donde s es una hnf, si y sólo si existe una derivación de* narrowing *perezoso uniforme $\mathcal{D}' \equiv (t \overset{\sigma}{\rightsquigarrow}_{ULN}{}^{*} s)$;*

2. *$\mathcal{D}$ y $\mathcal{D}'$ son derivaciones con el mismo número de pasos y que emplean las mismas reglas sobre las mismas posiciones en los pasos correspondientes de cada derivación.*

Para demostrar el Teorema 6.4.1 necesitaremos varios resultados axiliares.

Primero probamos que, para programas uniformes, se cumple la siguiente propiedad entre los distintos componentes de la representación canónica de un paso de *narrowing* necesario.

Lema 6.4.2 *Sea $\mathcal{R}$ un programa uniforme. Sea t un término encabezado por el símbolo de función f y $\mathcal{P}$ el árbol definicional de f. Si $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_1 \rangle \in \lambda(t, \mathcal{P})$ es un paso de* narrowing *necesario, entonces:*

1. *para $i, j \in \{1, \ldots, k\}$, con $i \neq j$, $\mathcal{V}ar(\vartheta_i) \cap \mathcal{V}ar(\vartheta_j) = \emptyset$;*

2. *para $i \in \{1, \ldots, k\}$, si $\vartheta_i = \{x/c_i(x_1, \ldots, x_{n_i})\}$ entonces $x \in \mathcal{V}ar(t)$.*

Prueba. Procedemos por inducción sobre k e inducción estructural sobre el árbol definicional $\mathcal{P}$

1. Caso base ($k = 1$): En este caso, necesariamente $pattern(\mathcal{P}) = \pi$ es una hoja; de otro modo $k > 1$, en contra de lo supuesto. Por la Definición 2.9.20 de estrategia de *narrowing* necesario, $\lambda(t, \mathcal{P}) = \{\langle \Lambda, \pi \to r, id \rangle\}$ y el enunciado del lema se cumple trivialmente.

2. Caso inductivo ($k > 1$): De acuerdo con la Definición 2.9.20 de estrategia de *narrowing* necesario, tenemos que $pattern(\mathcal{P}) = \pi$ es una rama y distinguimos las siguientes posibilidades. Sea o_1 la posición inductiva correspondiente a esta rama. Entonces $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_1 \rangle \in \lambda(t, \mathcal{P})$ si

   (a) $t|_{o_1} \equiv x$, $\vartheta_1 = \{x/c_1(x_1, \ldots, x_{n_1})\}$ y $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_2 \rangle \in \lambda(\vartheta_1(t), \mathcal{P}_1)$. Por hipótesis de inducción, las substituciones ϑ_j con $j \in \{2, \ldots, k\}$ satisfacen el enunciado. Además, $x \in \mathcal{V}ar(t)$, con lo que ϑ_1 cumple también cumple el punto (2) del enunciado y, por lo tanto, se cumple en el caso general. Para probar que se satisface el punto (1), razonamos del siguiente modo:

Por ser $\mathcal{R}$ un programa uniforme, las posiciones inductivas de $\mathcal{P}$ se corresponden con los argumentos en los que aparecen constructores planos de los términos lhs que definen f (Proposición 5.4.1). Por lo tanto, si o es una posición inductiva y o' es una posición de t con $o < o'$, entonces o' no es una posición inductiva. En otras palabras las posiciones inductivas en $\mathcal{P}$ se corresponden con posiciones disjuntas del término t.

Sea o_2 una posición inductiva de $\mathcal{P}$ y, por lo tanto, disjunta de o_1. Entonces, es imposible que $\vartheta_1(t)|_{o_2} \equiv y$ con $y \in \{x_1, \dots, x_{n_1}\}$. Se presentan las siguientes posibilidades:

i. $t|_{o_2} \equiv x$.
Entonces, $\vartheta_1(t)|_{o_2} \equiv c_1(x_1, \dots, x_{n_1})$ y por consiguiente $\vartheta_2 = id$. Así que, de manera trivial ϑ_1 no comparte variables con ϑ_2.
Nótese que debe existir un patrón $\pi' \in \mathcal{P}_1$, tal que $\pi'|_{o_2} \equiv c_1(y_1, \dots, y_{n_1})$, ya que en otro caso $\lambda(\vartheta_1(t), \mathcal{P}_1)$ no sería un paso de *narrowing* necesario y, por lo tanto, tampoco lo sería $\lambda(t, \mathcal{P})$.

ii. $t|_{o_2} \equiv y$.
Entonces, $\vartheta_1(t)|_{o_2} \equiv y$. En este caso, si existe un patrón $\pi' \in \mathcal{P}_1$, tal que $\pi'|_{o_2} \equiv c_2(y_1, \dots, y_{n_2})$, $\vartheta_2 = \{y/c_2(y_1, \dots, y_{n_2})\}$ y, por construcción del árbol definicional $\mathcal{P}$, las variable $y_1, \dots, y_{n_2}$ son frescas. Por tanto, ϑ_1 no comparte variables con ϑ_2.

iii. $t|_{o_2} \equiv c_2(t_1, \dots, t_{n_2})$.
Entonces, $\vartheta_1(t)|_{o_2} \equiv c_2(\vartheta_1(t_1), \dots, \vartheta_1(t_{n_2}))$ y por consiguiente $\vartheta_2 = id$. Por tanto, de manera trivial ϑ_1 no comparte variables con ϑ_2.
Nótese que el caso en el que $t|_{o_2} \equiv c(t_1, \dots, t_n)$, con $c \neq c_2$, no puede darse, ya que $\lambda(t, \mathcal{P})$ no sería un paso de *narrowing* necesario.

iv. $t|_{o_2} \equiv g(t_1, \dots, t_{n_2})$.
Idéntico al caso anterior.

Ahora podemos repetir este razonamiento, de forma que podemos probar que ϑ_1 no comparte variables con el resto de las substituciones ϑ_j que forman la representación canónica de $\lambda(t, \mathcal{P})$.

(b) $t|_{o_1} \equiv c_1(t_1, \dots, t_n)$, $\vartheta_1 = id$ y $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_2 \rangle \in \lambda(t, \mathcal{P}_1)$.
Por hipótesis de inducción, las substituciones ϑ_j con $j \in \{2, \dots, k\}$ satisfacen el enunciado. Por otro lado, en este caso trivialmente ϑ_1 no comparte variables con el resto de los ϑ_j, para $j \in \{2, \dots, k\}$, y el punto (2) del enunciado se cumple por vacuidad para ϑ_1.

(c) $t|_{o_1} \equiv g(t_1, \dots, t_n)$, $\vartheta_1 = id$ y $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_2 \rangle \in \lambda(t|_{o_1}, \mathcal{P}')$, donde $\mathcal{P}'$ es un árbol definicional para g.
Idéntico al caso anterior.

$\square$

Como consecuencia del lema anterior, los elementos de la substitución computada en un paso de *narrowing* necesario, representada en forma canónica, están restringidos a las variables del término t. Además, debido a que sus componentes no comparten variables, puede entenderse que forman una partición, de manera que esos componentes pueden ser reordenados sin afectar al resultado final de la composición. En otras palabras, los operadores de composición "$\circ$" y de unión de conjuntos "$\cup$" son intercambiables para los componentes de una substitución computada en un paso de *narrowing* necesario. Este resultado se formaliza mediante el siguiente corolario del cual haremos uso extensivo en adelante.

Corolario 6.4.3 *Sea $\mathcal{R}$ un programa uniforme. Sea t un término encabezado por el símbolo de función f y $\mathcal{P}$ el árbol definicional de f. Si $\langle p, R, \vartheta_k \circ \cdots \circ \vartheta_1 \rangle \in \lambda(t, \mathcal{P})$ es un paso de* narrowing *necesario, entonces $\vartheta_k \circ \cdots \circ \vartheta_1 = \vartheta_k \cup \cdots \cup \vartheta_1$*

Al comentar el Ejemplo 33 indicamos que algunas de las substituciones computadas por la aplicación α no eran "desechadas" en un paso de *narrowing* perezoso. Apoyándonos en el Corolario 6.4.3 podemos escribir $(\tau_2 \cup \tau_1) \in \alpha(t, \mathcal{P})$ distinguiendo entre la parte que es desechada, τ_1, y la que no es desechada, τ_2, en un paso de *narrowing* perezoso a partir de t. El siguiente lema pone de manifiesto que la parte desechada en el paso de *narrowing* perezoso vuelve a ser computada por la estrategia λ en el paso siguiente.

Lema 6.4.4 *Sea $\mathcal{R}$ un programa uniforme. Sea t un término encabezado por el símbolo de función f y $\mathcal{P}$ el árbol definicional de f. Sea $(\tau_2 \cup \tau_1) \in \alpha(t, \mathcal{P})$. Si $\langle p, R, \sigma \cup \tau_2 \cup \tau_1 \rangle \in \lambda(t, \mathcal{P})$ y $t \overset{[p,R,\sigma\cup\tau_2]}{\leadsto_{LN}} t'$ es un paso de* narrowing *perezoso, siendo $p \not\equiv \Lambda$, entonces $\langle p', R', \sigma' \cup \tau_1 \rangle \in \lambda(t', \mathcal{P})$.*

Prueba. Inmediata, sin más que observar que las posiciones disjuntas o por encima de la posición p permanecen inalteradas después del paso de *narrowing*, salvo las posibles instanciaciones de las variables $x \in \mathcal{D}om(\sigma \cup \tau_2)$. Además, por el Lema 6.4.2 $\mathcal{D}om(\sigma \cup \tau_2) \cap \mathcal{D}om(\tau_1) = \emptyset$. Por lo tanto, $\lambda(t', \mathcal{P})$ computa de nuevo la parte de substitución τ_1. $\square$

El siguiente lema relaciona las derivaciones de *narrowing* perezoso uniforme a partir de un término t y la correspondiente derivación obtenida cuando al término t se le aplica, previamente, la porción desechada de la substitución adelantada $\alpha(t, \mathcal{P})$.

Lema 6.4.5 *Sea $\mathcal{R}$ un programa uniforme. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$. Sea $\mathcal{P}$ el árbol definicional de f. Sea $\tau = (\tau_2 \cup \tau_1) \in \alpha(t, \mathcal{P})$ la substitución adelantada en un paso de* narrowing *necesario. Sea $\langle p_1, R_1, \sigma_1 \cup \tau_2 \cup \tau_1 \rangle \in \lambda(t, \mathcal{P})$ y $\langle p_1, R_1, \sigma_1 \cup \tau_2 \rangle \in \lambda_{ulazy}(t)$. Existe una derivación de* narrowing *perezoso uniforme $t \overset{\theta\circ\tau_1}{\leadsto}_{ULN}^{*} s$ si y sólo si existe una derivación de* narrowing *perezoso uniforme $\tau_1(t) \overset{\theta}{\leadsto}_{ULN}^{*} s$, donde el término s es una hnf y $\theta = \sigma \circ (\sigma_1 \cup \tau_2)$.*

Prueba. Por inducción sobre el número de pasos, n, de las derivaciones. Las substituciones se consideran módulo renombramiento de variables y restringidas a las variables del término t.

1. Caso base ($n = 1$): En este caso, el paso de *narrowing* perezoso, $t \overset{[p_1, R_1, \sigma_1 \cup \tau]}{\leadsto}_{ULN} s$, tiene que darse sobre la posición $p_1 = \Lambda$; en otro caso, el paso se daría sobre una posición $p_1 > \Lambda$ y, por definición de paso de *narrowing*, el término s estaría encabezado por un símbolo de función f, en contra de lo supuesto en el enunciado. Por el Lema 6.2.1, $\tau = id$, $\langle \Lambda, R, \sigma_1 \rangle \in \lambda(t, \mathcal{P})$ y $\langle \Lambda, R, \sigma_1 \rangle \in \lambda_{ulazy}(t)$. Así pues, el enunciado del lema se cumple trivialmente.

2. Caso inductivo ($n > 1$): Si existe la derivación de *narrowing* perezoso uniforme $t \overset{\theta \circ \tau_1}{\leadsto}_{ULN}{}^{*} s$, podemos dividir esta derivación en dos partes, de la siguiente forma:

$$t \overset{[p_1, R_1, \sigma_1 \cup \tau_2]}{\leadsto}_{ULN} t_1 \overset{\sigma \circ \tau_1}{\leadsto}_{ULN}{}^{*} s.$$

El caso en el que $p_1 = \Lambda$ es inmediato ya que, por el Lema 6.2.1, $\tau = id$. Consideremos por tanto el caso en el que $p_1 \neq \Lambda$. Por el Lema 6.4.4, $\langle p_2, R_2, \sigma_2 \cup \tau_1 \rangle \in \lambda(t_1, \mathcal{P})$, donde p_2 y R_2 son, respectivamente, la posición y la regla que se usan en el segundo paso de *narrowing* de la derivación. Ahora distinguimos dos casos, según se cumpla:

(a) $\langle p_2, R_2, \sigma_2 \cup \tau_1 \rangle \in \lambda_{ulazy}(t_1)$.
Entonces, el enlace adelantado se computa en este paso y de forma evidente, por definición de paso de *narrowing* y estar τ_1 restringida a las variables de t, puede darse el paso de *narrowing* $\langle p_2, R_2, \sigma_2 \rangle \in \lambda_{ulazy}(\tau_1(t_1))$. Por lo tanto, si la derivación $t_1 \overset{\sigma \circ \tau_1}{\leadsto}_{ULN}{}^{*} s$ está constituida por los pasos de *narrowing*

$$t_1 \overset{[p_2, R_2, \sigma_2 \cup \tau_1]}{\leadsto}_{ULN} t_2 \overset{\sigma'}{\leadsto}_{ULN}{}^{*} s,$$

donde $\sigma = \sigma' \circ \sigma_2$, ahora puede formarse la derivación alternativa

$$\tau_1(t_1) \overset{[p_2, R_2, \sigma_2]}{\leadsto}_{ULN} t_2 \overset{\sigma'}{\leadsto}_{ULN}{}^{*} s.$$

Esto es, existe la derivación $\tau_1(t_1) \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$.

(b) $\langle p_2, R_2, \sigma_2 \rangle \in \lambda_{ulazy}(t_1)$.
Entonces, el enlace adelantado se computará en alguno de los pasos de *narrowing* posteriores. En este caso se cumplen las condiciones para aplicar la hipótesis de inducción. Por lo tanto, dado que existe la derivación $t_1 \overset{\sigma \circ \tau_1}{\leadsto}_{ULN}{}^{*} s$ y tiene menos de n pasos, por hipótesis de inducción existe la derivación $\tau_1(t_1) \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$.

Por otro lado, dado que se puede dar el paso de *narrowing* perezoso uniforme $t \overset{[p_1, R_1, \sigma_1 \cup \tau_2]}{\leadsto}_{ULN} t_1 = \sigma_1(\tau_2((t[r_1]_{p_1})))$, supueto que $R_1 \equiv l_1 \to r_1$, podemos asegurar que el correspondiente problema de unificación lineal tiene éxito, esto es, $\mathsf{LU}(\langle l_1, t|_{p_1}\rangle) = (\textsc{Succ}, \sigma_1 \cup \tau_2)$. En lo que sigue comprobaremos que $\mathsf{LU}(\langle l_1, \tau_1(t)|_{p_1}\rangle) = (\textsc{Succ}, \sigma_1 \cup \tau_2)$:

(a) Primero se debe notar que τ_1 no introduce nuevos redexes cuando se aplica al término t, por tratarse de una substitución constructora lineal (Proposición 2.9.21). Por consiguiente, el problema de unificación lineal $\langle l_1, \tau_1(t)|_{p_1}\rangle$ también tendrá éxito.

(b) Sean $l_1 = f(d_1, \ldots, d_k)$ y $t|_{p_1} = f(s_1, \ldots, s_k)$. Dado que el problema de unificación lineal $\langle l_1, t|_{p_1}\rangle$ tiene éxito, sabemos que la configuración inicial LU

$$(\{d_1 \downarrow_1 s_1, \ldots, d_k \downarrow_k s_k\}, id)$$

es reducible a la configuración irreducible $(\emptyset, \sigma_1 \cup \tau_2)$ mediante la relación $\rightarrow_{\mathsf{LU}}$. Estudiemos las características de esta reducción, fijándonos en el elemento $d_i \downarrow_i s_i$ de la configuración LU inicial. Por ser $\mathcal{R}$ un programa uniforme, d_i es una variable o un término constructor plano y lineal. En el primer caso, si $d_i \equiv z_i$, obtenemos la substitución $\{z_i/s_i\}$ (i.e., id, cuando nos restringimos a las variables del término t). En el segundo caso, si $d_i \equiv c_i(z_1, \ldots, z_m)$, pueden darse las siguientes posibilidades:

   i. $s_i \equiv x \in \mathcal{V}ar(t)$.
   Obtenemos la substitución $\{x/c_i(z_1, \ldots, z_m)\}$. Dado que l_1 es un patrón lineal cuyos argumentos son variables o constructores planos y las reglas se toman renombradas aparte, las variables $z_1, \ldots, z_m$ no forman parte del dominio de ninguna otra substitución parcial computada en la derivación LU. Así pues, esta substitución es un elemento de $\sigma_1 \cup \tau_2$.

   ii. $s_i \equiv c_i(s'_1, \ldots, s'_m)$, con $c_i \in \mathcal{C}$.
   En este caso, el elemento $d_i \downarrow_i s_i$ de la configuración LU inicial se transforma en $c_i(z_1, \ldots, z_m) \downarrow_i c_i(s'_1, \ldots, s'_m)$. Después de un paso de reducción LU, la configuración LU inicial quedaría:

$$(\{d_1 \downarrow_1 s_1, \ldots, z_1 \downarrow_{i.1} s'_1, \ldots, z_m \downarrow_{i.m} s'_m, \ldots, d_k \downarrow_k s_k\}, id).$$

   En el siguiente paso obtendríamos las substituciones $\{z_i/s'_i\}$ (i.e., id, cuando nos restringimos a las variables del término t).

   Nótese que los casos en los que $s_i \equiv c_j(s'_1, \ldots, s'_m)$, con $c_j \in \mathcal{C}$ y $c_j \neq c_i$, o $s_i \equiv g(s'_1, \ldots, s'_m)$, con $g \in \mathcal{F}$, no pueden darse, ya que entonces el problema de unificación lineal no tendria éxito, en contra de lo supuesto.

(c) Si aplicamos la substitución τ_1 al término $t|_{p_1}$, el problema de unificación lineal planteado $\langle l_1, \tau_1(t|_{p_1})\rangle$ conduce a la configuración inicial LU

$$(\{d_1 \downarrow_1 \tau_1(s_1), \ldots, d_k \downarrow_k \tau_1(s_k)\}, id)$$

que debe ser reducible a una configuración ireducible mediante la relación $\to_{\mathsf{LU}}$. Estudiemos, nuevamente, las características de esta reducción, fijándonos en un elemento arbitrario $d_i \downarrow_i \tau_1(s_i)$ de la configuración LU inicial y realizando un análisis de casos como el anterior. Si $d_i \equiv z_i$ obtenemos la substitución $\{z_i/\tau_1(s_i)\}$ (i.e., id, cuando nos restringimos a las variables del término t). Si $d_i \equiv c_i(z_1, \ldots, z_m)$, pueden darse las siguientes posibilidades:

i. $s_i \equiv x \in \mathcal{V}ar(t)$, con $x \notin \mathcal{D}om(\tau_1)$.
En este caso, $\tau_1(s_i) = x$ y se obtiene la substitución $\{x/c_i(z_1, \ldots, z_m)\}$. Dado que l_1 es un patrón lineal cuyos argumentos son variables o constructores planos y las reglas se toman renombradas aparte, esta substitución es el elemento de $\sigma_1 \cup \tau_2$ que se computó en el punto (2.b.i).

ii. $s_i \equiv x \in \mathcal{V}ar(t)$, con $x \in \mathcal{D}om(\tau_1)$.
Entonces $\tau_1(s_i) = \tau_1(x)$, con lo que el elemento $d_i \downarrow_i \tau_1(s_i)$ de la configuración LU inicial se transforma en $c_i(z_1, \ldots, z_m) \downarrow_i \tau_1(x)$. Como el problema de unificación lineal tiene éxito, $\tau_1(x)$ debe ser un término encabezado por el símbolo constructor c_i. Esto es, $\tau_1(x) = c_i(s'_1, \ldots, s'_m)$ y la configuración LU inicial, después de un paso de reducción LU, quedaría:

$$(\{d_1 \downarrow_1 \tau_1(s_1), \ldots, z_1 \downarrow_{i.1} s'_1, \ldots, z_m \downarrow_{i.m} s'_m, \ldots, d_k \downarrow_k \tau_1(s_k)\}, id).$$

En el siguiente paso obtendríamos las substituciones $\{z_i/s'_i\}$ (i.e., id, cuando nos restringimos a las variables del término t).

iii. $s_i \equiv c_i(s'_1, \ldots, s'_m)$, con $c_i \in \mathcal{C}$.
También obtendríamos la substitución id, cuando nos restringimos a las variables del término t.

Mediante este análisis, podemos concluir que $\mathsf{LU}(\langle l_1, \tau_1(t)|_{p_1}\rangle\rangle) = (\textsc{Succ}, \sigma_1 \cup \tau_2)$ y, por lo tanto, que $\langle p_1, R_1, \sigma_1 \cup \tau_2\rangle \in \lambda_{ulazy}(\tau_1(t))$. Es decir, podemos dar el paso de *narrowing* perezoso uniforme

$$\begin{aligned}
\tau_1(t) \overset{[p_1, R_1, \sigma_1 \cup \tau_2]}{\leadsto_{ULN}} \quad & \sigma_1(\tau_2((\tau_1(t)[r_1]_{p_1}))) \\
&= \sigma_1(\tau_2(\tau_1(t[r_1]_{p_1}))) \\
&= \tau_1(\sigma_1(\tau_2(t[r_1]_{p_1}))) \\
&= \tau_1(t_1)
\end{aligned}$$

ya que la substitución τ_1 está restringida a las variables del término t y σ_1, y τ_1 y τ_2 son substituciones constructoras lineales que no comparten variables entre sí (Lema 6.4.2), lo que nos permite conmutar las substituciones. Ahora, podemos construir la siguiente derivación de *narrowing* perezoso:

$$\tau_1(t) \overset{[p_1, R_1, \sigma_1 \circ \tau_2]}{\leadsto_{ULN}} \tau_1(t_1) \overset{\sigma'}{\leadsto_{ULN}}{}^{*} s.$$

La prueba en el sentido de dirección opuesto es completamente análoga. □

Corolario 6.4.6 *Sea $\mathcal{R}$ un programa uniforme. Sea t un término encabezado por un símbolo de función definida $f \in \mathcal{F}$. Sea $\mathcal{P}$ el árbol definicional de f. Sea $\tau \in \alpha(t, \mathcal{P})$ la substitución adelantada en un paso de* narrowing *necesario. Existe una derivación de* narrowing *perezoso uniforme $t \overset{\sigma\circ\tau}{\leadsto}_{ULN}{}^{*}\, s$, donde el término s es una hnf, si y sólo si existe una derivación de* narrowing *perezoso uniforme $\tau(t) \overset{\sigma}{\leadsto}_{ULN}{}^{*}\, s$.*

Prueba. Inmediata, por el Lema 6.4.5 y la definición de paso de *narrowing*.

Sea $\tau = (\tau_2 \cup \tau_1) \in \alpha(t, \mathcal{P})$, donde τ_2 representa los enlaces que se establecen en el primer paso de *narrowing* perezoso uniforme y que, por lo tanto, no son desechados. Por el Lema 6.4.5, dado que existe la derivación de *narrowing* perezoso uniforme $t \overset{[\theta\circ\tau_1]}{\leadsto}_{ULN}{}^{*}\, s$ existe también una derivación de *narrowing* perezoso uniforme $\tau_1(t) \overset{\theta}{\leadsto}_{ULN}{}^{*}\, s$, donde el término s es una hnf, $\theta = \sigma \circ (\sigma_1 \cup \tau_2)$, $\langle p_1, R_1, \sigma_1 \cup \tau_2 \cup \tau_1\rangle \in \lambda(t, \mathcal{P})$ y $\langle p_1, R_1, \sigma_1 \cup \tau_2\rangle \in \lambda_{ulazy}(t)$. Ahora bien, por definición de paso de *narrowing*, si existe el paso de *narrowing* perezoso uniforme

$$\tau_1(t) \overset{[p_1, R_1, \sigma_1\cup\tau_2]}{\leadsto}_{ULN}\ t_2 \equiv \sigma_1(\tau_2(\tau_1(t)[r_1]_{p_1}))$$

entonces existe el paso de *narrowing* perezoso uniforme

$$\begin{aligned} \tau_2(\tau_1(t)) \overset{[p_1, R_1, \sigma_1]}{\leadsto}_{ULN}\ & \sigma_1(\tau_2(\tau_1(t))[r_1]_{p_1}) \\ &= \sigma_1(\tau_2(\tau_1(t)[r_1]_{p_1})) \\ &= t_2 \end{aligned}$$

ya que la substitución τ_1 está restringida a las variables del término t. Ahora, podemos construir la siguiente derivación de *narrowing* perezoso uniforme:

$$\tau_2(\tau_1(t)) \overset{[p_1, R_1, \sigma_1]}{\leadsto}_{ULN}\ t_2 \overset{\sigma}{\leadsto}_{ULN}{}^{*}\, s.$$

La prueba en el sentido contrario se puede establecer de forma analoga. □

Finalmente estamos en disposición de probar el Teorema 6.4.1 y de establecer la equivalencia entre las semánticas operacionales por *narrowing* necesario y por *narrowing* perezoso uniforme.

Prueba.

1. Demostramos el punto (1) del enunciado por inducción sobre el número de pasos, n, de la derivación $\mathcal{D}$. Las substituciones se consideran módulo renombramiento de variables.

   (a) Caso base ($n = 1$):
   Idéntico al caso base del Lema 6.4.5.

   (b) Caso inductivo ($n > 1$):
   Sea

$$\mathcal{D} \equiv (t \overset{[p_1, R_1, \sigma_1\circ\tau_1]}{\leadsto}_{NN}\ t_1 \overset{[p_2, R_2, \sigma_2\circ\tau_2]}{\leadsto}_{NN}, \ldots \overset{[p_{n-1}, R_n, \sigma_n\circ\tau_n]}{\leadsto}_{NN}\ s),$$

donde $\theta_i = (\sigma_i \circ \tau_i)$ y $\tau_i \in \alpha(s_{i-1}, \mathcal{P}_{i-1})$ es la substitución adelantada en cada paso $i \in \{1, \ldots, n\}$. La derivación $\mathcal{D}$ puede dividirse en dos partes como, indicamos a continuación:

$$t \overset{[p_1,R_1,\theta_1]}{\leadsto}_{NN} t_1 \overset{\theta_n \circ \cdots \circ \theta_2}{\leadsto}_{NN}{}^{*} s.$$

Dado que existe el paso de *narrowing* necesario $t \overset{[p_1,R_1,\theta_1]}{\leadsto}_{NN} t_1$, por el Corolario 6.2.4 existe el paso de *narrowing* perezoso uniforme $\tau_1(t) \overset{[p_1,R_1,\sigma_1]}{\leadsto}_{ULN} t_1$. También, dado que existe la derivación de *narrowing* necesario, $t_1 \overset{\theta_n \circ \cdots \circ \theta_2}{\leadsto}_{NN}{}^{*} s$, que tiene menos de n pasos, por hipótesis de induccción existe también la derivación de *narrowing* perezoso uniforme $t_1 \overset{\theta_n \circ \cdots \circ \theta_2}{\leadsto}_{ULN}{}^{*} s$. Ahora, podemos formar la derivación de *narrowing* perezoso uniforme:

$$\tau_1(t) \overset{[p_1,R_1,\sigma_1]}{\leadsto}_{ULN} t_1 \overset{\theta_n \circ \cdots \circ \theta_2}{\leadsto}_{ULN}{}^{*} s,$$

y, por el Corolario 6.4.6, existe la derivación de *narrowing* perezoso uniforme $t \overset{\theta_n \circ \cdots \circ \theta_2 \circ \theta_1}{\leadsto}_{ULN}{}^{*} s$.

La prueba en el sentido de dirección opuesto puede realizarse fácilmente ya que el razonamiento empleado es completamente reversible.

2. El punto (2) del enunciado de este teorema es una consecuencia inmediata de la Proposición 6.2.3.

□

Corolario 6.4.7 *Sea $\mathcal{R}$ un programa uniforme. Sea e una ecuación. Existe una derivación de* narrowing *necesario $e \overset{\sigma}{\leadsto}_{NN}{}^{*}$ true si y sólo si existe una derivación de* narrowing *perezoso uniforme $e \overset{\sigma}{\leadsto}_{ULN}{}^{*}$ true.*

Prueba. Sean t_1 y t_2 términos y $e \equiv t_1 \approx t_2$. Por definición de solución de una ecuación, existe una derivación de *narrowing* necesario $e \overset{\sigma}{\leadsto}_{NN}{}^{*}$ *true* si y sólo si existen las derivaciones de *narrowing* necesario $t_1 \overset{\sigma}{\leadsto}_{NN}{}^{*} s$ y $t_2 \overset{\sigma}{\leadsto}_{NN}{}^{*} s$, donde s es un término constructor. Por el Teorema 6.4.1, existen las derivaciones de *narrowing* necesario $t_1 \overset{\sigma}{\leadsto}_{NN}{}^{*} s$ y $t_2 \overset{\sigma}{\leadsto}_{NN}{}^{*} s$, si y sólo si existen las derivaciones *narrowing* perezoso uniforme $t_1 \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$ y $t_2 \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$. Finalmente, usando de nuevo la definición de solución de una ecuación, existen las derivaciones de *narrowing* perezoso uniforme $t_1 \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$ y $t_2 \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$ si y sólo si existe una derivación de *narrowing* perezoso uniforme $e \overset{\sigma}{\leadsto}_{ULN}{}^{*}$ *true*. □

Los resultados establecidos hasta el momento permiten demostrar que, para programas uniformes, la estrategia de *narrowing* perezoso uniforme es correcta y completa con respecto a ecuaciones estrictas y substituciones constructoras.

Teorema 6.4.8 *Sea $\mathcal{R}$ un programa uniforme y e una ecuación.*

1. *(Corrección) Si $e \overset{\sigma}{\leadsto}_{ULN}{}^{*}$ true es una derivación de* narrowing *perezoso uniforme, entonces σ es una solución para e.*

2. *(Completitud) Para cada substitución constructora σ que es una solución de e, existe una derivación de* narrowing *perezoso uniforme* $e \rightsquigarrow^{\sigma'}_{ULN}{}^{*}$ *true con* $\sigma' \leq \sigma$ $[\mathcal{V}ar(e)]$.

Prueba. Inmediato, por el Corolario 6.4.7 y el Teorema 2.9.23. □

Llegado a este punto, es conveniente hacer notar que la Definición 6.3.1 junto con el resultado anterior caracterizan a la estrategia de *narrowing* perezoso uniforme como una estrategia que goza de la propiedad de completitud fuerte.

6.5 Conclusiones.

En este capítulo hemos realizado un estudio exhaustivo de la relación existente entre la estrategia de *narrowing* necesario [23] y la estrategia de *narrowing* perezoso presentada en el Apartado 2.9.3, que en esencia es una formalización de la aparece en [120]. Algunas de las contribuciones originales se resumen en los siguientes puntos:

- Para programas uniformes simples, las estrategias de *narrowing* perezoso y de *narrowing* necesario son equivalentes paso a paso (Proposición 6.1.3).
- Hemos dado una definición operacional del concepto de substitución adelantada en un paso de *narrowing* necesario.
- Hemos precisado la relación existente entre los pasos de *narrowing* necesario y los pasos de *narrowing* perezoso, para programas uniformes (Proposición 6.2.4 y Proposición 6.2.5).
- Hemos caracterizado una nueva estrategia de evaluación perezosa para programas uniformes, que hemos denominado *narrowing* perezoso uniforme.
- Hemos probado la equivalencia de las estrategias de *narrowing* perezoso uniforme y de *narrowing* necesario para programas uniformes (Teorema 6.4.1).
- Hemos demostrado que la estrategia de *narrowing* perezoso uniforme es correcta y fuertemente completa para programas uniformes (Teorema 6.4.8).

Estos resultados ponen de manifiesto que los programas uniformes constituyen la clase de programas más amplia para la cual se puede lograr una implementación eficiente de la estrategia de *narrowing* perezoso, sin perder las buenas propiedades de la estrategia de *narrowing* necesario.

Estos resultados tienen aplicación inmediata a la optimización de las técnicas de evaluación parcial, como estudiamos en el próximo capítulo.

Capítulo 7

Evaluación Parcial de Programas Uniformes.

En este capítulo formulamos una instancia del marco de evaluación parcial introducido en el Capítulo 3 que consiste, esencialmente, en particularizar el Algoritmo 1 del Apartado 3.3.2 para la estrategia de *narrowing* perezoso uniforme y estudiamos las ventajas y problemas que plantea.

7.1 Un Evaluador Parcial Basado en Narrowing Perezoso Uniforme.

La instancia que se propone se justifica recordando una de las dificultades que presenta la evaluación parcial de programas cuando se emplea la estrategia de *narrowing* perezoso [17]: la pérdida de la ortogonalidad del programa original. La evaluación parcial de programas utilizando la estrategia de *narrowing* perezoso puede conducir a programas que no son siquiera ortogonales, aun en el caso de partir de programas uniformes, como muestra el siguiente ejemplo.

Ejemplo 39 *Sea $\mathcal{R}$ el programa uniforme del Ejemplo 30. La evaluación parcial de $\mathcal{R}$ con respecto al término $f(g(X), g(a), g(b))$, usando una regla de desplegado de un paso, conduce al siguiente programa especializado*[1]*:*

$$\begin{array}{lll} \mathcal{R}' = U_{\lambda_{lazy}}(S, \mathcal{R}) = & \{f(g(X), g(a), g(b)) & \to & f(g(X), a, g(b)) \\ & f(g(X), g(a), g(b)) & \to & f(g(X), g(a), b) \\ & f(g(X), a, g(b)) & \to & f(g(X), a, b) \\ & f(g(X), g(a), b) & \to & f(g(X), a, b) \\ & f(g(X), a, b) & \to & 1\}. \end{array}$$

[1]Después de la tercera iteración, el conjunto de términos evaluados parcialmente es: $S = \{f(g(X), g(a), g(b)),\ f(g(X), a, g(b)),\ f(g(X), g(a), b),\ f(g(X), a, b)\}$.

que tras el postproceso de renombramiento, adoptando el siguiente renombramiento independiente

$$
\begin{array}{rl}
S = \{ & \langle f(g(X), g(a), g(b)) \mapsto h(X) \rangle, \\
& \langle f(g(X), a, g(b)) \mapsto h_1(X) \rangle, \\
& \langle f(g(X), g(a), b) \mapsto h_2(X) \rangle, \\
& \langle f(g(X), a, b) \mapsto h_3(X) \rangle \},
\end{array}
$$

produce el programa renombrado

$$
\begin{array}{rlcl}
\mathcal{R}'' = & \{h(X) & \to & h_1(X) \\
& h(X) & \to & h_2(X) \\
& h_1(X) & \to & h_3(X) \\
& h_2(X) & \to & h_3(X) \\
& h_3(X) & \to & 1\}.
\end{array}
$$

que no es ortogonal ya que no cumple la condición de no ambigüedad[2]*. Evidentemente, dicho programa derivado tampoco es uniforme.*

El resultado obtenido en el ejemplo anterior es consecuencia de que la estrategia de *narrowing* perezoso computa derivaciones redundantes, en las que el orden en el que se usan las reglas simplemente se intercambia cuando se aplican sobre los mismos redexes perezosos. De esta forma, derivaciones parciales que finalmente computarían el mismo resultado con la misma respuesta, dan lugar a distintas resultantes. Este ejemplo muestra la necesidad de emplear una estrategia de *narrowing* perezoso refinada durante el proceso de evaluación parcial, como la estrategia de *narrowing* perezoso uniforme, que no presenta este tipo de problemas en programas uniformes.

Ejemplo 40 *Consideremos de nuevo el programa uniforme $\mathcal{R}$ del Ejemplo 30. La evaluación parcial de $\mathcal{R}$ con respecto al término $f(g(X), g(a), g(b))$, usando también una regla de desplegado de un paso, pero aplicando la estrategia de* narrowing *perezoso uniforme en lugar de la estrategia de* narrowing *perezoso, produce el siguiente programa especializado*

$$
\begin{array}{rlcl}
\mathcal{R}'' = & \{h(X) & \to & h_1(X) \\
& h_1(X) & \to & h_3(X) \\
& h_3(X) & \to & 1\}.
\end{array}
$$

que es un programa uniforme.

A la luz de los dos últimos ejemplos, proponemos como instancia concreta la que consiste en elegir la estrategia de *narrowing* perezoso uniforme y un criterio de parada basado en el orden de subsumción (Definición 3.4.1) para definir la regla de desplegado y operador de abstracción. Para preservar la completitud del proceso de evaluación parcial imponemos la condición de no desplegar los árboles locales de *narrowing* más alla de un término en hnf. De forma precisa, la definición de evaluador parcial perezoso uniforme es:

[2]Adviértase que tampoco cumple la condición de no ambigüedad débil

Definición 7.1.1 (Evaluador Parcial Perezoso Uniforme) *Un evaluador parcial perezoso uniforme es una instancia concreta del Algoritmo 1 que se obtiene al elegir:*

1. *la estrategia de* narrowing *perezoso uniforme, λ_{ulazy}, para el desplegado de los árboles locales;*

2. *la regla de desplegado de la Definición 3.4.7 (parametrizada con la estrategia de* narrowing *perezoso uniforme, λ_{ulazy}) y el preorden de subsumción de la Definición 3.4.1, reforzado con el criterio de parada adicional consistente en detener el desplegado de los árboles de* narrowing *local cuando se alcanza una hnf;*

3. *el operador de abstracción de no-subsumción, abstract*, de la Definición 3.4.8.*

En lo que sigue nos referiremos a esta instancia particular del algoritmo de NPE con el acrónimo ULN-PE (del inglés *Uniform Lazy Narrowing-driven Partial Evaluation*), para distinguirla de la instancia presentada en el Capítulo 4 a la que hemos denominado LN-PE.

Como estudiamos en el Capítulo 4, en ocasiones, una LN-PE de un programa puede hacer perder la disciplina de constructores en el programa transformado, lo que impide que la estrategia de *narrowing* perezoso pueda emplearse para evaluar términos en el programa transformado. Se necesita un postproceso de renombramiento tanto para recuperar la disciplina de constructores del programa original y asegurar que podrá ejecutarse en el programa transformado, como para lograr la condición de independencia, es decir que no habrá interferencias entre las reglas del programa transformado. Lograr la independencia del conjunto de términos parcialmente evaluados es vital para la corrección (fuerte) de la LN-PE.

Para programas uniformes, se necesita además del postproceso de renombramiento un postproceso adicional, como pone de relieve el siguiente ejemplo, en el que se muestra que debido al anidamiento de constructores en las lhs de las reglas se puede perder la uniformidad del programa especializado.

Ejemplo 41 *Sea el programa uniforme*

$$\begin{array}{llll} \mathcal{R} = \{ & R_1 : & f(c(X)) & \to g(X) \\ & R_2 : & g(a) & \to a\} \end{array}$$

Cuando se emplea la estrategia de narrowing *perezoso uniforme, para el término $t \equiv f(X)$, solamente existe la siguiente derivación que conduce a un término constructor en cabeza:*

$$f(x) \overset{[\Lambda, R_1, \{X/c(Y)\}]}{\leadsto_{ULN}} g(Y) \overset{[\Lambda, R_2, \{Y/a\}]}{\leadsto_{ULN}} a,$$

que produce el programa especializado

$$\begin{array}{llll} \mathcal{R}' = \{ & R_1' : & f(c(a)) & \to a\} \end{array}$$

después de aplicar una iteración del algoritmo de evaluación parcial. Este programa $\mathcal{R}'$ (que coincide con el programa renombrado $\mathcal{R}''$) no es un programa uniforme, ya que incumple la primera restricción de la Definición 5.2.1 de programa uniforme.

Adviértase que este mismo problema también afecta a la estrategia de *narrowing* necesario ya que, para el programa $\mathcal{R}$ del Ejemplo 41, la estrategia de *narrowing* necesario conduce al mismo programa especializado $\mathcal{R}'$

Si queremos ejecutar el programa transformado utilizando *narrowing* perezoso uniforme, es importante recuperar la uniformidad del programa original lo cual es indispensable para asegurar la corrección del proceso de ULN-PE. A continuación proponemos una fase de postproceso adicional que utiliza el siguiente algoritmo para aplanar las lhs's de las reglas del programa evaluado parcialmente.

Algoritmo 3 (Aplanamiento, F)
Entrada: *Un programa $\mathcal{R}$.*
Salida: *Un programa en forma plana $F(\mathcal{R})$.*

$$
\begin{array}{lll}
F(\mathcal{R}) = & \texttt{Si} \ \textit{las lhs's de las reglas de } \mathcal{R} \textit{ son planas} \ \texttt{entonces} \ \mathcal{R} & \\
& \texttt{sino Sea} \ [R \equiv (f(t_1, \ldots, t_n) \to (E \Rightarrow r)) \in \mathcal{R} \ \textit{tal que} & \\
& \quad (\exists i, j, m : 1 \leq i \leq n, 1 \leq j \leq m). \ (t_i = c(s_1, \ldots, s_m) \land r \notin \mathcal{X}); & \\
& \quad \texttt{y} \ \mathcal{R}' \ = \ (\mathcal{R} \setminus \{R\}) \cup & \\
& \qquad \{f(t_1, \ldots, t_{i-1}, c(z_1, \ldots, z_m), t_{i+1}, \ldots, t_n) \to & \\
& \qquad (z_1 \approx s_1 \land \ldots \land z_m \approx s_m \land E \Rightarrow r)\}] & \\
& \quad \texttt{En} \ F(\mathcal{R}'). &
\end{array}
$$

Las variables $z_1, \ldots, z_m$ son variables nuevas y E es una condición, posiblemente vacía.

Ejemplo 42 *Volviendo al Ejemplo 41, si aplicamos el Algoritmo 3 al programa especializado*

$$\mathcal{R}' = \{ \quad R_1' : \quad f(c(a)) \quad \to a\},$$

obtenemos el programa

$$\mathcal{R}'' = F(\mathcal{R}') = \{ \quad R_1'' : \quad f(c(X)) \quad \to (X = a \Rightarrow a)\},$$

para el que se ha recuperado la restricción de patrones constructores planos y, por lo tanto, la uniformidad del programa original.

Hacemos notar que en este ejemplo un algoritmo de aplanamiento como el que aparece en [120] restablecería la uniformidad al precio de perder toda la especialización conseguida en el proceso de ULN-PE, ya que obtendríamos nuevamente el programa original.

7.2 Corrección y Completitud Fuerte de la Evaluación Parcial Dirigida por Narrowing Perezoso Uniforme.

En este apartado demostramos la corrección y completitud fuerte del método ULN-PE. La prueba hace uso de todo el trabajo desarrollado en el Capitulo 6 para establecer la equivalencia del *narrowing* perezoso uniforme y del del *narrowing* necesario en programas uniformes. Sin embargo, todavía necesitamos de un resultado previo, que formalizamos de la siguiente manera.

Lema 7.2.1 *Sean $\mathcal{R}$ un programa uniforme y R una regla de $\mathcal{R}$. Sean t y s términos. Si $t \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$, entonces existe una selección de árboles definicionales de las funciones definidas en $\mathcal{R}$ tales que $t \overset{\sigma}{\leadsto}_{NN}{}^{*} s$.*

Prueba. Por inducción sobre el número de pasos, n, de la derivación.

1. Caso base ($n = 1$): Inmediato por el Corolario 6.2.6.

2. Caso inductivo ($n > 1$): Si $t \overset{\sigma}{\leadsto}_{ULN}{}^{*} s$, entonces podemos formar la derivación

   $$t \overset{[p,R,\sigma]}{\leadsto}_{ULN} t_1 \overset{\sigma'}{\leadsto}_{ULN}{}^{*} s.$$

   Dado que $t_1 \overset{\sigma'}{\leadsto}_{ULN}{}^{*} s$ es una derivación con menos de n pasos, por la hipótesis de inducción existe una selección apropiada de árboles definicionales tales que $t_1 \overset{\sigma'}{\leadsto}_{NN}{}^{*} s$. También, por el Corolario 6.2.6 existe un árbol definicional (que pasa a formar parte de la selección) tal que $t \overset{[p,R,\sigma]}{\leadsto}_{NN} t_1$. Ahora podemos formar la derivación

   $$t \overset{[p,R,\sigma]}{\leadsto}_{NN} t_1 \overset{\sigma'}{\leadsto}_{NN}{}^{*} s,$$

   y el lema queda probado.

□

Teorema 7.2.2 (Corrección y Completitud Fuerte)
Sea $\mathcal{R}$ un programa uniforme, e una ecuación y S un conjunto finito de términos. Sea $\mathcal{R}'$ una hnf-PE de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}' \cup \{e\}$ es S-cerrado. Sea S' un renombramiento independiente de S, $\mathcal{R}''$ un renombramiento de $\mathcal{R}'$ con respecto a S', seguido de un aplanamiento, y $e'' = ren(e, S')$. Entonces, θ es una respuesta computada para e en $\mathcal{R}$ si y sólo si θ es una respuesta computada para e'' en $\mathcal{R}''$.

Prueba. Por el Lema 7.2.1, si $\mathcal{R}''$ es una ULN-PE entonces es también una NN-PE. Ahora, el resultado se sigue por la corrección y completitud fuerte de la NN-PE [17]. □

7.3 Resultados Experimentales.

En este apartado presentamos una serie de experimentos para medir el impacto que las diferentes estrategias de *narrowing* perezoso tienen en la eficiencia del proceso de evaluación parcial.

Los experimentos se han realizado con el sistema INDY[3] cuyo uso y características se describen en [4, 13]. El sistema INDY es un evaluador parcial dirigido por *narrowing* para lenguajes lógico–funcionales, que puede especializar programas utilizando las estrategias de *narrowing* perezoso y de *narrowing* necesario. Para realizar nuestro estudio, también se ha incorporado al sistema la posibilidad de utilizar la estrategia de *narrowing* perezoso uniforme. En la especialización de los programas de prueba se han considerado las siguientes opciones:

- Estrategia de evaluación: Todas las pruebas se han repetido empleando las diferentes estrategias de *narrowing* perezoso (sin normalización).
- Regla de desplegado: Hemos probado dos alternativas:
    1. `emb_goal`: Expande las derivaciones mientras los nuevos objetivos no subsumen un objetivo comparable que haya aparecido previamente en la misma rama del árbol de desplegado;
    2. `emb_redex`: La regla de desplegado concreta definida en el Apartado 8.2.1 que implementa el test *dependency_clash* comprobando la subsumción homeomórfica entre ancestros comparables de los redexes seleccionados para asegurar la finitud del proceso de desplegado (se debe notar que en el contexto actual, en el que no tratamos con símbolos primitivos – ver el Capítulo 8 –, esta opción difiere de la opción `emb_goal` simplemente en que el test de subsumción homeomórfica se realiza sobre redexes en vez de sobre objetivos completos).
- Operador de abstracción: El proceso de abstracción se realiza siempre utilizando el operador de abstracción definido en el Apartado 8.2.2, sin la posibilidad de partir las expresiones complejas que contienen símbolos de función primitivos (ver el Capítulo 8), ya que en el contexto actual nos restringimos a la evaluación parcial de expresiones simples que no contienen símbolos de función primitivos y, por lo tanto, se hace innecesario el empleo de este tipo de técnicas de partición.

Los *programas de prueba* considerados en la realización de los experimentos han sido los siguientes: `ackermann`, la clásica función de Ackermann; `allones`, que transforma los elementos de una lista de números en 1's; `applast`, que añade

[3] En el Apéndice A puede encontrarse un resumen de las principales características del sistema INDY.

un elemento al final de una lista y devuelve el último elemento de la lista resultante de esta operación; `exam`, el programa del Ejemplo 21; `fibonacci`, la conocida función de Fibonacci; `match-kmp`, un programa ingenuo de búsqueda de patrones en cadenas; `palindrome`, un programa para comprobar si una lista dada es o no un palíndromo; `sumprod`, que obtiene la suma y el producto de los elementos de una lista; `matmul`, un programa que multiplica matrices; y `sumleq`, un programa que contiene las reglas que definen las funciones "$+$","$-$", y "$\leq$". Algunos de estos programas son ejemplos típicos de programas de prueba extraidos del ámbito de la deducción parcial (ver [124, 127]) y adaptados a una sintaxis lógico-funcional, mientras que otros provienen de la literatura asociada al área de la transformación de programas funcionales, tal como la supercompilación positiva [108], las transformaciones de plegado/desplegado [48, 55] y la deforestación [203]. En el Apéndice B aparece el código de los programas utilizados en los experimentos, así como las llamadas evaluadas parcialmente. Las llamadas se han seleccionado para que den lugar a tiempos de especialización razonablemente largos. En algunos casos ha sido necesario transformar los programas a forma uniforme con el fin de poder aplicar la estrategia de *narrowing* perezoso uniforme.

Los resultados experimentales se resumen en las Tablas 7.1, 7.2 y 7.3. Los tiempos se midieron en un computador HP 9000/712, ejecutando el sistema operativo HP Unix vB.10.20. Están expresados en milisegundos y se corresponden con el promedio de 5 ejecuciones.

Las Tablas 7.1 y 7.2 muestran el tiempo de especialización y el aumento de la eficiencia alcanzada con respecto al tiempo de especialización que obtiene el sistema INDY cuando se emplea la estrategia de *narrowing* perezoso para el desplegado de los árboles locales. También se da información sobre la bondad de la especiación alcanzada en términos del número de reglas del programa especializado. Más precisamente, las columnas "Rw" indican el número de reglas de reescritura de los programas originales o transformados, "T_{INDY}" representa los tiempos de especialización para las diferentes estrategias perezosas y "Mejora" se ha obtenido como el cociente entre el tiempo de especialización transcurrido cuando se emplea la estrategia de *narrowing* perezoso y el tiempo de especialización invertido cuando se emplea la estrategia de *narrowing* perezoso uniforme o de *narrowing* necesario.

Del análisis de los tiempos que se presentan en las Tablas 7.1 y 7.2, se desprende la superioridad de la estrategia de *narrowing* necesario aún en los casos en los que el programa original está en forma uniforme. Los mejores resultados para la estrategia de *narrowing* necesario se obtienen cuando se emplea la regla de desplegado `emb_redex` debido a que esta regla permite un desplegado más liberal y, por lo tanto, da lugar a un mayor número de pasos de *narrowing*. Entonces, el ligero aumento en los índices de mejora sería indicativo de una mayor eficiencia de la estrategia de *narrowing* necesario con respecto a la estrategia de *narrowing* perezoso uniforme. Estos resultados apuntan a la hipótesis de que, en una

	Orig.	LN-PE		ULN-PE			NN-PE		
Programas	Rw	Rw	T_{INDY}	Rw	T_{INDY}	Mej.	Rw	T_{INDY}	Mej.
`ackermann`	5	23	38010	23	26170	1.45	23	29230	1.30
`allones`	7	5	420	5	290	1.45	5	290	1.45
`applast`	6	4	408	4	326	1.25	4	324	1.26
`exam`	5	5	420	5	380	1.10	5	374	1.12
`fibonacci`	6	11	2370	11	1636	1.45	11	1645	1.44
`match-kmp`	13	14	3920	14	3548	1.10	14	3080	1.27
`matmult`	10	19	2550	19	1830	1.39	19	1640	1.55
`palindrome`	12	19	4500	19	3270	1.38	19	2910	1.55
`sumleq`	8	8	480	8	436	1.10	8	408	1.18
`sumprod`	9	14	4280	14	3062	1.39	14	2913	1.47
Promedio	8.1	12.2	5693.8	12.2	4094.8	1.31	12.2	4281.4	1.36

Tabla 7.1: Comparación de LN-PE, ULN-PE, y NN-PE: tiempo de especialización (en ms.) y tamaño de los programas considerando la estrategia desplegado `emb_goal`.

	Orig.	LN-PE		ULN-PE			NN-PE		
Programas	Rw	Rw	T_{INDY}	Rw	T_{INDY}	Mej.	Rw	T_{INDY}	Mej.
`ackermann`	5	35	63010	35	30190	2.09	35	33850	1.86
`allones`	7	4	480	4	328	1.46	4	304	1.57
`applast`	6	6	778	6	612	1.27	6	608	1.28
`exam`	5	3	350	3	290	1.17	3	268	1.30
`fibonacci`	6	7	797	7	578	1.38	7	562	1.42
`match-kmp`	13	14	5910	14	5520	1.07	14	4920	1.20
`matmult`	10	15	3112	15	1914	1.63	15	1700	1.83
`palindrome`	12	19	6180	19	4840	1.28	19	4470	1.38
`sumleq`	8	8	600	8	567	1.06	8	513	1.17
`sumprod`	9	18	3710	18	3047	1.22	18	2820	1.32
Promedio	8.1	12.9	8492.7	12.9	4789.4	1.36	12.9	5001.5	1.43

Tabla 7.2: Comparación de LN-PE, ULN-PE, y NN-PE: tiempo de especialización (en ms.) y tamaño de los programas considerando la estrategia desplegado `emb_redex`.

implementación como la del sistema INDY, el uso y mantenimiento de los árboles definicionales no resulta gravoso.

El sistema INDY implementa la estragia de evaluación de *narrowing* perezoso dejando el control de los cómputos al mecanismo de búsqueda indetermista del PROLOG; i.e., se realiza una búsqueda a ciegas, sin información específica suministrada por las características propias del problema (e.g., la secuencialidad inductiva del programa). Como estamos viendo, ésto penaliza la eficiencia de los cómputos. Sin embargo, el sistema INDY implementa la estragia de evaluación de *narrowing* necesario haciendo uso explícito de los árboles definicionales. Los árboles definicionales contienen toda la información sobre las reglas del programa y permiten seleccionar la posición del término con la que se dará el paso de *narrowing* de forma eficiente. La estrategia de *narrowing* perezoso uniforme podemos considerar que actúa de una forma intermedia, en la cual la

	LN-PE	ULN-PE	
Términos:	T_{INDY}	T_{INDY}	Mejora
$f(X, g(a), g(b))$	180	70	2.57
$f(X, g(Y), g(Z))$	220	90	2.44
$f(a, g(g(g(X))), g(g(g(Y))))$	3890	118	32.97
$f(a, g(g(g(g(X)))), g(g(g(g(Y)))))$	81970	700	117.10
$f(a, g(f(a, g(X), g(Y))), g(f(a, g(X), g(Y))))$	65410	460	142.20

Tabla 7.3: Comparación de LN-PE y ULN-PE: evaluación parcial del programa del Ejemplo 30 para diferentes términos; el tiempo de especialización se ha medido en ms; la estrategia desplegado seleccionada ha sido `emb_redex`.

información de los árboles definicionales se almacena "compilada" en forma de reglas en un programa transformado uniforme dejando, nuevamente, el control de los cómputos al mecanismo de búsqueda indeterminista del PROLOG. Los experimentos realizados indican que los árboles definicionales son una excelente heurística a la hora de guiar una evaluación perezosa.

Por otra parte, cuando se emplea la estrategia de *narrowing* perezoso uniforme, los experimentos confirman una mejora en los tiempos de especialización, resultantes del proceso de evaluación parcial, con respecto a los conseguidos con la estrategia de *narrowing* perezoso original. También confirman una mejora en la clase de los programas especializados obtenidos, evitandose tanto la aparición de reglas redundantes como la pérdida de la secuencialidad inductiva. La Tabla 7.3 resume los tiempos de especialización y las mejoras obtenidas en la evaluación parcial del programa del Ejemplo 30 con respecto a diferentes términos. Este es un programa uniforme en el que se ponen de relieve las ventajas de la estrategia de *narrowing* perezoso uniforme con respecto a la estrategia de *narrowing* perezoso original.

7.4 Conclusiones.

En [17], Alpuente et al. demuestran que la evaluación parcial basada en *narrowing* necesario (NN-PE) hereda las buenas propiedades del mecanismo de base (e.g., preserva la estructura inductiva de los programas, las computaciones deterministas, las propiedades de optimalidad, etc). Por otro lado, la LN-PE puede aplicarse a una clase más amplia de programas, si bien presenta varios inconvenientes (e.g., se obtienen reglas redundantes en el programa especializado y se pierde la ortogonalidad del programa original). Los resultados del capítulo 6 caracterizan los programas uniformes como la clase más amplia de programas sobre la que pueden obtenerse ventajas comparables a las de la NN-PE sin la necesidad de usar estructuras como los árboles definicionales que utiliza el *na-*

rrowing necesario para guiar la ejecución. Aunque la representación explícita de los árboles definicionales puede tener un coste elevado para la eficiencia de la ejecución de los programas, así como para el consumo de memoria, los experimentos realizados en el Apartado 7.3 indican que para una implementación como la del sistema INDY, el uso y mantenimiento de los árboles definicionales no se manifiesta más gravoso que otros procesos. Sin embargo, nuestra intuición nos dice que para evaluadores parciales que admitan programas de un mayor tamaño y, por lo tanto, árboles definicionales más complejos, la situación podría inversirse resultando el balance favorable para la estrategia de *narrowing* perezoso uniforme. Así pues, la idea es hacer uso de la estrategia de *narrowing* perezoso uniforme para evitar el problema que introduce la gestión de los árboles definicionales obteniendo, todavía, las ventajas de la evaluación parcial basada en *narrowing* necesario. Esta decisión estaría en la línea de la última implementación del lenguaje Curry [24] en la que, en lugar de emplear árboles definicionales, las funciones se definen mediante expresiones *case* que "compilan" la información de los árboles definicionales en el propio código del programa.

En este capítulo, explotando las buenas propiedades de los programas uniformes, hemos conseguido:

- definir una nueva instancia del marco general para la evaluación parcial de programas lógico–funcionales que denominamos ULN-PE;

- definir un postproceso de aplanamiento (Algoritmo 3) que restablece la uniformidad del programa transformado sin perdida de la especialización conseguida durante el proceso de evaluación parcial; y

- probar que la ULN-PE goza de las propiedades de corrección y completitud fuerte y preserva la uniformidad del programa original (tras el postproceso de aplanamiento).

Los experimentos realizados con el sistema INDY, extendido con las adaptaciones del mecanismo operacional de base necesarias para aplicar la estrategia de *narrowing* perezoso uniforme, constatan la importancia práctica que la nueva estrategia supone para las técnicas de evaluación parcial de programas integrados perezosos con respecto a la estrategia de *narrowing* perezoso original. Nuestra intención es seguir profundizando en esta línea de investigación, estudiando la mejora que los resultados obtenidos suponen para las técnicas de evaluación parcial de programas integrados perezosos.

Parte III

Técnicas Avanzadas de Especialización.

Capítulo 8

Mejora del control en la Evaluación Parcial Dirigida por Narrowing.

Las técnicas clásicas de la deducción parcial de programas lógicos computan evaluaciones parciales para cada uno de los átomos diferentes que constituyen un objetivo de forma independiente. Recientemente, en [83, 128] se ha introducido una técnica para la deducción parcial de conjunciones de átomos, que ha recibido el nombre de *deducción parcial conjuntiva*. Esta técnica se ha diseñado con el objetivo de superar las limitaciones relativas al tratamiento de conjunciones de átomos, conectados por variables compartidas. Básicamente, el método de la deducción parcial conjuntiva puede verse como una mejora del proceso de abstracción de la deducción parcial clásica consistente en generalizar una conjunción de átomos bien directamente o bien partiendo la conjunción en subconjunciones antes de generalizar (o empleando una combinación de ambas técnicas). En la deducción parcial clásica, una conjunción siempre se divide en sus componentes atómicos. La deducción parcial conjuntiva consigue ciertas optimizaciones difíciles de los programas, tales como (alguna forma de) *tupling* y *deforestación*, que se obtienen habitualmente a través de técnicas de transformación más complejas, como las técnicas de plegado/desplegado, que son difíciles de automatizar y que no pueden obtenerse a través de la deducción parcial clásica. La supercompilación positiva también es capaz de producir efectos similares.

El método NPE introducido en el Capítulo 3 es capaz de producir especialización *poligenética* (i.e., es capaz de extraer definiciones especializadas que combinan varias definiciones de función del programa original). Esto significa que la NPE tienen un potencial de especialización similar al de la deducción parcial o la supercompilación positiva dentro del paradigma considerado. Como ya se ha adelantado en el Apartado 3.3.1, esto es debido a que el método genérico de la NPE permite manipular ecuaciones y conjunciones durante el proceso de especialización, simplemente considerando los operadores de igualdad y de conjunción como símbolos de función *primitivos* del lenguaje. Desafortunadamente, el uso de funciones primitivas complica notablemente la naturaleza del los problemas de especialización y, a menudo, se necesita alguna forma de *tupling*[1] para lograr la especialización de expresiones que contienen llamadas conjuntivas. El algoritmo de NPE no incorpora un tratamiento específico para los símbolos primitivos, que son considerados en pie de igualdad con el resto de símbolos de función definidos. Este hecho hace que se desperdicien muchas oportunidades para alcanzar la condición de *cierre* y el método se ve forzado a una excesiva generalización de las llamadas, dando lugar a una pérdida (casi total) de la especialización conseguida hasta el momento y resultando en una especialización monogenética en muchos casos interesantes (ver el Ejemplo 43). Inspirados por los resultados obtenidos por la deducción parcial conjuntiva [83], en este capítulo extendemos el algoritmo básico de la NPE (Algoritmo 1) incorporando nuevas opciones de control y comprobamos experimentalmente la efectividad de las mismas.

8.1 Motivación y Conceptos Preparatorios.

En este apartado generalizamos algunos conceptos básicos relacionados con las técnicas de NPE de los programas lógico–funcionales (perezosos) tal y como se han presentado hasta el momento. Estas nuevas nociones generales se muestran de gran utilidad a la hora de permitir la definición de operadores de desplegado y de abstracción más potentes, capaces de manipular convenientemente los símbolos de función primitivos. En el marco original para la NPE que se ha presentado en el Capítulo 3 no se hace ninguna distinción entre símbolos de función definidos y primitivos durante el proceso de especialización. Por ejemplo, una conjunción $b_1 \wedge b_2$ se considera como una entidad unitaria cuando se comprueba si está o no cubierta por el conjunto de llamadas especializadas. En el caso general, esto conlleva una drástica generalización de las llamadas envueltas en el proceso de especialización, que puede causar una pérdida total de especialización. El siguiente ejemplo ilustra este punto y motiva el resto del capítulo.

Ejemplo 43 *Sea el fragmento de programa:*

$$\begin{array}{l} \mathit{sorted\_bits}(X:[\,]) \;\rightarrow\; \mathit{true} \\ \mathit{sorted\_bits}(X_1:X_2:Xs) \;\rightarrow\; \mathit{sorted\_bits}(X_2:Xs) \;\wedge\; X_1 \leq X_2 \\ 0 \leq 0 \;\rightarrow\; \mathit{true} \qquad 0 \leq 1 \;\rightarrow\; \mathit{true} \qquad 1 \leq 1 \;\rightarrow\; \mathit{true} \end{array}$$

cuyo significado declarativo es que sorted_bits es verdadero si la lista de bits que se suministra como argumento está ordenada de acuerdo a la relación orden "$\leq$" definida por las reglas del programa. Consideremos la llamada $\mathit{sorted\_bits}(X:Xs) \wedge 1 \leq X$. *La Figura 8.1 muestra el árbol de* narrowing *local construido*

[1] Tal y como se define en [166] para los programas lógicos

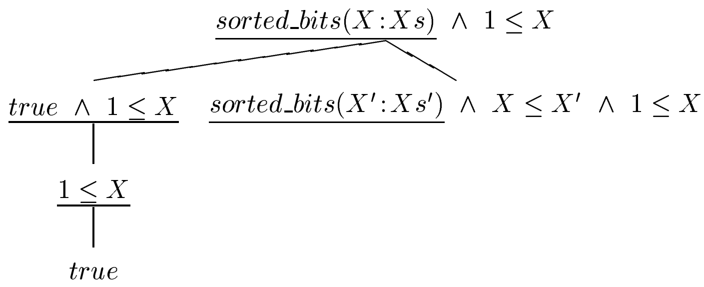

Figura 8.1: Arbol de *narrowing* incompleto para la expresión ($sorted\_bits(X\!:\!Xs) \;\wedge\; 1 \leq X$).

empleando la regla de desplegado de la Definición 3.4.7, parametrizada utilizando la estrategia de narrowing *perezoso y un criterio de parada basado en el orden de subsumción homeomórfico. Intuitivamente, al utilizar este criterio de parada, las derivaciones se expanden mientras que se cumpla que los nuevos redexes considerados no son sintácticamente más grandes que alguno de los redexes* comparables *(i.e., redexes con el mismo símbolo de función en cabeza) que le preceden en la misma rama. En el árbol de la Figura 8.1, pueden identificarse dos debilidades del algoritmo básico de NPE (Algoritmo 1):*

- *La expansión de la rama de la derecha se detiene debido a que el redex más a la izquierda* $sorted\_bits(X' : Xs')$ *en la hoja del árbol subsume el redex* $sorted\_bits(X : Xs)$ *seleccionado en el paso previo, incluso aunque no se han realizado reducciones en otros elementos de la conjunción, lo cual no parece muy conveniente ni equitativo.*

- *De acuerdo con la distinción entre el control local y global, cuando se corta la expansión de los árboles locales y el control pasa de nuevo al nivel global, el algoritmo intenta probar que la llamada* $sorted\_bits(X' : Xs') \;\wedge\; X \leq X' \;\wedge\; 1 \leq X$ *que aparece en la hoja del árbol esta cubierta por la llamada inicial* $sorted\_bits(X : Xs) \;\wedge\; 1 \leq X$ *(plegado); de otro modo, se realiza una generalización de las llamadas comparables. Ya que las llamadas consideradas son comparables y la llamada* $sorted\_bits(X' : Xs') \wedge X \leq X' \wedge 1 \leq X$ *de la hoja del árbol subsume (pero no es cubierta por) la llamada especializada* $sorted\_bits(X : Xs) \;\wedge\; 1 \leq X$*, se computa el msg* $sorted\_bits(X : Xs) \;\wedge\; Z$*, provocando una excesiva generalización que conduce a la pérdida de toda la especialización conseguida.*

A continuación comentamos las adaptaciones realizadas en el marco genérico de NPE para facilitar una manipulación correcta y adecuada de los símbolos primitivos.

En el ejemplo anterior se ha empleado una definición de la estrategia de *narrowing* perezoso que supone una regla selección fija, en el orden de izquierda a derecha, de las expresiones dentro de una conjunción. Como ya hemos comentado, la estrategia de *narrowing* perezoso goza de la propiedad de *completitud fuerte* con respecto a substituciones constructoras en los TRS's CB y ortogonales. La completitud fuerte significa que, dado un término conjuntivo $t_1 \wedge t_2 \wedge \ldots \wedge t_n$, cualquiera de las posiciones perezosas relativas al subtérmino t_i se pueden seleccionar de forma "don't-care", ignorando el resto de posiciones perezosas pertenecientes a términos t_j, con $j \neq i$, sin afectar a la completitud del proceso de evaluación. En particular, al utilizar la Definición 2.9.14 y la definición de programa que aparece en el Apartado 2.9.1, el subtérmino t_i seleccionado para su inspección es aquél que, conteniendo un subconjunto de posiciones perezosas, se encuentra más a la izquierda en la conjunción. Se ha simulado esta selección "don't-care" introduciendo en los programas una única regla para definir el símbolo de función primitivo "$\wedge$" (i.e., $true \wedge x \to x$), lo cual fuerza la selección del subconjunto de posiciones perezosas situado más a la izquierda en cada término conjuntivo. Esta forma de selección es demasiado rígida en la situación actual. Se necesita una estrategia de *narrowing* que sea más flexible y capaz de seleccionar todas las posiciones perezosas que puedan existir en un término conjuntivo, de forma que posteriormente el interprete pueda adoptar cualquier política de planificación (*scheduling*). Esto será útil a la hora de definir una regla de desplegado concreta que permita desplegar los árboles de *narrowing* local de forma más equilibrada, lo cual es crucial a la hora de alcanzar una especialización efectiva. Una forma de conseguir el efecto deseado con el mínimo coste es aumentar los programas con una nueva regla para definir el símbolo de función primitivo "$\wedge$": $x \wedge true \to x$. De esta forma, podemos simular la selección de todas las ocurrencias perezosas de un término, ya que la primera regla demanda las posiciones perezosas que aparecen en la parte izquierda de una conjunción y esta segunda regla demanda las posiciones que aparecen en la parte derecha. Por consiguiente, en el resto de este capítulo supondremos (mientras no se haga explícito otra cosa) que todo programa contiene el siguiente conjunto de reglas predefinidas:

$$\begin{array}{rcl} c \approx c & \to & true \\ c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) & \to & (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) \\ true \wedge x & \to & x \\ x \wedge true & \to & x \\ (true \Rightarrow x) & \to & x \end{array}$$

Por otra parte, también se hace necesario definir una condición recursiva de *cierre* más flexible, que impida una generalización excesiva de los términos que contienen símbolos de función primitivos y todavía garantize que cada llamada que pueda ocurrir durante la ejecución del programa residual estará cubierta por una regla de programa. La nueva definición de cierre se formaliza comprobando, de manera inductiva, que las diferentes llamadas de las reglas están cubiertas por las funciones especializadas.

Definición 8.1.1 (Cierre Generalizado)
Sea S un conjunto finito de términos y t un término. Decimos que un término t es S-cerrado si se cumple closed(S, t), donde el predicado closed se define inductivamente como sigue:

$$closed(S,t) \Leftrightarrow \begin{cases} true & \texttt{si } t \in \mathcal{X} \\ closed(S,t_1) \wedge \ldots \wedge closed(S,t_n) & \texttt{si } t \equiv c(t_1,\ldots,t_n) \\ \exists s \in S^+.\ s\theta = t \wedge \bigwedge_{t' \in \mathcal{R}an(\theta)} closed(S,t') & \texttt{si } t \equiv f(t_1,\ldots,t_n) \end{cases}$$

donde $c \in \mathcal{C}$, $f \in (\mathcal{D} \cup \mathcal{P})$ y $S^+ = S \cup \{p(x,y) \mid p \in \mathcal{P}\}$. Decimos que un conjunto de términos T es S-cerrado, escrito $closed(S,T)$, si se cumple $closed(S,t)$ para todo $t \in T$, y decimos que un programa $\mathcal{R}$ es S-cerrado si se cumple que $closed(S,\mathcal{R}_{calls})$ es verdadero, siendo $\mathcal{R}_{calls}$ el conjunto de los términos rhs de las reglas de $\mathcal{R}$.

Informalmente, un término t encabezado por un símbolo de función definido es cerrado con respecto a un conjunto de llamadas S, si es una instancia de un término de S y los términos del rango de la substitución de emparejamiento son también cerrados (recursivamente) con respecto a S. La novedad con respecto a la Definición 3.3.3 es que se puede probar que una expresión compleja encabezada por un símbolo de función primitivo, tal como una conjunción, es cerrada con respecto a S bien comprobando que es una instancia de una llamada de S (seguido de una comprobación inductiva para los subtérminos), o bien dividiendo la expresión en dos conjunciones e intentando emparejarlas entonces con términos más simples de S (lo que sucede cuando el emparejamiento se intenta primero con respecto a una de las llamadas planas $p(x,y)$ de S^+). Esta extensión es segura debido al hecho de que las reglas que definen las funciones primitivas se incorporan automáticamente a cada programa; así, las llamadas a estos símbolos están cubiertas regularmente en el programa especializado. Esta sencilla extensión de la condición de cierre permite formular operadores de abstracción en los que los términos que contienen símbolos de función primitiva (posiblemente) se parten antes de intentarse un plegado o una generalización. Nótese que los términos conjuntivos pueden partirse, de forma indeterminista, en conjunciones de un número arbitrario de elementos, debido a la asociatividad de '$\wedge$'.

El siguiente ejemplo ilustra la Definición 8.1.1 y pone de manifiesto las diferencias entre la vieja condición de cierre y la nueva.

Ejemplo 44 *Sea t el término $(f(g(0)) \approx 0 \ \wedge \ g(X) \approx 1)$ y S el conjunto $\{f(X), g(X), (f(0) \approx 0 \ \wedge \ X \approx 1)\}$. Intuitivamente, con la nueva Definición 8.1.1 de cierre, un término es cerrado con respecto a un conjunto S si puede partirse en componentes cerradas. Entonces, t es S-cerrado ya que, después de partir convenientemente el término t:*

1. *$f(g(0))$ es S-cerrado, porque está cubierto por $f(X)$;*
2. *$g(X)$ es S-cerrado, porque está cubierto por $g(X)$.*

Si el término t no hubiese podido partirse, entonces habría sido imposible demostrar que es cerrado con respecto al conjunto S, ya que no existe ninguna substitución θ tal que $t = \theta(s_i)$, para algún $s_i \in S$. En un proceso de abstracción, esta situación podría conducir a una pérdida de especialización.

El Algoritmo 1 es un algoritmo genérico para la NPE. El modo en el que se construyen las evaluaciones parciales concretas viene dado por una *regla de desplegado*, que determina las expresiones que serán reducidas (respetando una estrategia de *narrowing* φ) y que decide cómo parar la expansión de los árboles locales de *narrowing*. Por otra parte, el *operador de abstracción* garantiza la terminación global del proceso de NPE asegurando la finitud de los conjuntos de términos para los que se produce una evaluación parcial así como garantizando que no es posible que una misma llamada sea sucesivamente introducida y eliminada del conjunto, lo cual produciría igualmente la no terminación aun si el conjunto de llamadas no creciese infinitamente. El primer inconveniente mostrado en el Ejemplo 43 motiva la definición de una regla de desplegado más sofisticada que sea capaz de conseguir una evaluación *equilibrada* de la expresión reduciendo por *narrowing* los redexes apropiados (e.g., utilizando alguna forma de estrategia de planificación dinámica que tenga en cuenta los ancestros reducidos por *narrowing* en la misma rama). El segundo inconveniente sugiere la definición de un operador de abstracción más flexible que sea capaz de dividir términos complejos antes de intentar el plegado o la generalización. En el próximo apartado formalizamos estas técnicas refinadas de control que permiten mejorar la calidad de la especialización alcanzada por el método NPE.

8.2 Mejora del Control en el Método NPE.

La inclusión de símbolos de función primitivos (y, en particular, el operador de conjunción) en las llamadas que deben evaluarse parcialmente, plantea problemas de control específicos que no se han considerado previamente. En el nivel de control local, la reducción de elementos en una conjunción puede realizarse haciendo uso del indeterminismo *don't care* que permite la completitud fuerte de la estrategia de *narrowing* perezoso, pero el orden en el que se seleccionan los elementos durante la construcción del árbol de *narrowing* local es crucial para alcanzar una buena especialización. El Ejemplo 45 pone de manifiesto que una selección ingenua puede hacer perder toda la especialización conseguida hasta ese instante. Se necesita alguna clase de estrategia de selección dinámica, que tenga en cuenta los ancestros reducidos en la misma derivación, si queremos aumentar significativamente la eficiencia de la especialización. En el nivel de control global, mejorar el algoritmo genérico para la NPE requiere desarrollar técnicas particulares para la partición de términos complejos, al mismo tiempo que se mantienen tan unidos como sea posible aquéllos términos que pueden comunicar estructuras de datos entre ellos. Lamentablemente, esto último no siempre es posible debido al hecho de que, para asegurar la terminación del método NPE, algunas veces es necesario forzar la partición de los términos en posiciones inadecuadas.

En el próximo apartado, estudiamos la mejora del control local fijando una estrategia de desplegado diseñada específicamente para la "especialización conjuntiva". En el Apartado 8.2.2, se introduce, para el control global, un tratamiento específico de los símbolos de función primitivos '$\approx$', '$\wedge$' y '$\Rightarrow$' que

$$\underline{app(X,Y)} \approx W \;\wedge\; app(W,Z) \approx V$$

$$\vdots \qquad \underline{X' : app(Xs',Y) \approx W} \;\wedge\; app(W,Z) \approx V$$

$$\underline{X' \approx W'} \;\wedge\; app(Xs',Y) \approx Ws' \;\wedge\; app(W' : Ws',Z) \approx V$$

$$\underline{true \;\wedge\; app(Xs',Y) \approx Ws' \;\wedge\; app(W' : Ws',Z) \approx V}$$

$$\underline{app(Xs',Y)} \approx Ws' \;\wedge\; app(W' : Ws',Z) \approx V$$

Figura 8.2: Control local ingenuo para la llamada
$(app(X,Y) \approx W \;\wedge\; app(W,Z) \approx V)$.

produce una especialización poligenética más efectiva, en comparación con la NPE clásica.

8.2.1 Mejora del Control Local.

La regla de desplegado introducida en la Definición 3.4.7 simplemente explota los redexes seleccionados por la estrategia de *narrowing* perezoso λ_{lazy} (empleando una regla de selección estática fijada, que determina el siguiente elemento a reducir dentro de la conjunción) siempre que ninguno de ellos subsuma un redex (comparable) existente en la misma rama. El ejemplo siguiente revela que esta estrategia no es lo suficientemente elaborada como para especializar las llamadas que puedan contener símbolos primitivos como la conjunción, existiendo un riesgo de perder toda la especialización conseguida en el proceso de NPE.

Ejemplo 45 *Consideremos de nuevo el programa append:*

$$\begin{array}{rcl} app([\,],Y) & \rightarrow & Y \\ app(X : Xs,Y) & \rightarrow & X : app(Xs,Y) \end{array}$$

junto con el objetivo de entrada "$app(X,Y) \approx W \wedge app(W,Z) \approx V$". Utilizando la regla de desplegado de la Definición 3.4.7, en la primera fase de la especialización obtenemos el árbol representado[2] *en la Figura 8.2 (utilizando una regla de selección de izquierda a derecha dentro de las conjunciones). A partir de este árbol local, no se puede obtener una definición especializada apropiada para el objetivo inicial, ya que el término de la hoja no puede plegarse utilizando la llamada inicial de la raíz siendo necesario una generalización (que causa la pérdida de toda la especialización, como en el Ejemplo 43). Nótese que el subtérmino*

[2] Se ha adoptado la optimización estándar que hace uso del mecanismo de unificación interno (*built-in*) para resolver ecuaciones estrictas $s \approx t$ (suponiendo que solamente se producen enlaces constructores), evitándose así la creación de puntos superfluos de elección.

$app(W' : Ws', Z) \approx V$ dentro de la conjunción que aparece en la hoja del árbol de la Figura 8.2 no se ha evaluado suficientemente como para permitir que pueda plegarse con la llamada inicial. Nuevamente, el problema se resume en que no se consigue una evaluación equilibrada de la llamada inicial.

Es interesante notar que el método NPE clásico triunfa en este ejemplo cuando la llamada con respecto a la cual se desea realizar la evaluación parcial del programa se escribe como una expresión anidada $app(app(X, Y), Z)$. Esto se debe a que explotar la capacidad de anidamiento de la sintaxis funcional permite transformar el problema de *tupling*[3] original ilustrado por el Ejemplo 45 en un problema más sencillo de deforestación, que el método NPE original soluciona fácilmente. De hecho, se ha probado que el método NPE clásico está especialmente bien adaptado para realizar la deforestación, mientras que no es capaz de conseguir automáticamente la transformación de *tupling* en cualquier situación (ver [14]).

La regla de desplegado dinámica que vamos a introducir intenta alcanzar una evaluación equilibrada de los términos de entrada, antes que una evaluación en profundidad de algunos subtérminos concretos, dejando el resto de los otros subtérminos inalterados. Esto puede evitar el corte prematuro de algunas derivaciones y, en algunos casos, produce una mejor especialización que cuando se usa una regla de selección "ciega". La idea es que la nueva regla de desplegado seleccione dinámicamente las posiciones que deben reducirse dentro de una conjunción dada, seleccionando las posiciones perezosas más apropiadas de entre aquéllas sujetas a una elección *don't care*, explotando para ello alguna información sobre las dependencias funcionales entre redexes recogida a lo largo de la derivación. Esta nueva regla de desplegado se comporta de manera similar a la de [83] pero la nueva definición es más amplia, ya que se ha formulado para resolver el problema general de la especialización de términos complejos que contienen posiblemente (diferentes clases de) operadores anidados. También debe ocuparse de mantener el comportamiento perezoso del lenguaje lógico-funcional con el que tratamos.

La siguiente noción de *posición dependiente* se emplea para trazar las *dependencias funcionales* entre redexes del árbol de *narrowing* que se construye durante el proceso de NPE.

Definición 8.2.1 (Posiciones Dependientes)
Sea $D \equiv (s \leadsto_{p,l\to r,\sigma} t)$ un paso de narrowing*. El conjunto de* posiciones dependientes *de una posición q de s en D, escrito $q\backslash\backslash D$, es:*

$$q\backslash\backslash D = \begin{cases} \{q.u \mid u \in \mathcal{FP}os(r) \wedge \mathcal{H}ead(r|_u) \notin \mathcal{C}\} & \texttt{si } q = p \\ \{q\} & \texttt{si } q \parallel p \\ \{p.u'.v \mid r|_{u'} = x\} & \texttt{si } (q = p.u.v \ \wedge \ l|_u = x \in \mathcal{X}) \end{cases}$$

Esta noción se extiende a las derivaciones de *narrowing* de forma natural. La noción de dependencia para términos proviene directamente de la noción correspondiente para posiciones. Nótese que la definición anterior es estrictamente una

[3] Aquí nos referimos al concepto de *tupling* tal y como se entiende en la programación lógica, que engloba las transformaciones de deforestación y *tupling* de la programación funcional [166].

extensión de la noción estándar de *descendiente* de la programación funcional (ver la Definición 2.7.5). La misma técnica de “seguir la pista” de los redexes, subrayándolos para determinar los descendientes de un redex, puede emplearse para hacer más accesible el concepto de posición dependiente. Intuitivamente, una posición q' de t depende de una posición q de s (en D) si q' es un descendiente de q (segundo y tercer caso) o si la posición q' ha sido introducida por la rhs de la regla aplicada en la reducción por *narrowing* de la posición q precedente y señala un subtérmino encabezado por un símbolo de función definido (primer caso). El primer caso refleja las dependencias funcionales entre posiciones que aparecen debido a la aplicación de una regla del programa, mientras el segundo y tercer casos especifican cómo se propagan a lo largo de los pasos de la computación las posiciones que no son reducidas por *narrowing*. Nótese que esta noción es una extensión del concepto estándar de *ancestro* (*covering ancestor* [47]) de la deducción parcial adaptado convenientemente al marco lógico-funcional. Por abuso de lenguaje, si q' depende de q en una derivación D, diremos que el término apuntado por la ocurrencia q es un *ancestro* del término apuntado por la ocurrencia q'. Si s es un ancestro de t en D y $\mathcal{H}ead(s) = \mathcal{H}ead(t)$, decimos que s es un *ancestro comparable* de t en D.

Ejemplo 46 *Consideremos el árbol de narrowing local que muestra la Figura 8.2. La llamada $app(X,Y)$ que aparece en el objetivo inicial es un ancestro de la llamada $app(Xs',Y)$ que aparece en la hoja de dicho árbol (ya que la llamada a app ha sido introducida por la rhs de la segunda regla que define el símbolo de función app). Sin embargo, no es un ancestro de la llamada más a la derecha $app(W' : Ws', Z)$ que aparece en la hoja (cuyo ancestro en el objetivo inicial es la llamada $app(W,Z)$).*

En lo que sigue, formalizamos como se realiza la selección dinámica de los redexes (perezosos).

Definición 8.2.2 (Regla de Selección Dinámica)
Sea $D \equiv (t_0 \leadsto t_1 \leadsto \ldots \leadsto t_n)$, con $n \geq 0$, una derivación de narrowing *perezoso. Definimos la* regla de selección dinámica $\varphi_{dynamic}$ *como sigue:*

$\varphi_{dynamic}(t_n, D) = select(t_n, \Lambda, D)$,

donde la función auxiliar select es:

$$
\begin{array}{lll}
select(t,p,D) = \texttt{si } \langle p,k,\sigma\rangle \in \lambda_{lazy}(t) & & \\
\quad \texttt{entonces sea } t|_p = f(s_1,\ldots,s_n) \texttt{ y} & & \\
\qquad O_{args} = \bigcup_{i=1}^{n} select(t,p.i,D) \texttt{ en} & & \\
\qquad [\texttt{si } dependency\_clash(t|_p, D) & & \\
\qquad \texttt{entonces } \{\bot\} \texttt{ sino } \{\langle p,k,\sigma\rangle\} \cup O_{args}] & & \\
\quad \texttt{sino en caso de que } t|_p \texttt{ sea} & & \\
\qquad x \in \mathcal{V}\text{:} & \emptyset & \\
\qquad s_1 \wedge s_2\text{:} & \texttt{sea } O_i = select(t,p.i,D),\ i \in \{1,2\} \texttt{ en} & \\
 & [\texttt{si } \exists i.\ (\bot \not\in O_i\ \wedge\ O_i \not\equiv \emptyset) \texttt{ entonces } O_i & \\
 & \texttt{sino si } (O_1 \equiv O_2 \equiv \emptyset) \texttt{ entonces } \emptyset \texttt{ sino } \{\bot\}] & \\
\qquad \texttt{De otro modo}\text{:} & \texttt{sea } t|_p = f(s_1,\ldots,s_n) \texttt{ y} & \\
 & O_{args} = \bigcup_{i=1}^{n} select(t,p.i,D) \texttt{ en} & \\
 & [\texttt{si } \bot \in O_{args} \texttt{ entonces } \{\bot\} \texttt{ sino } O_{args}] &
\end{array}
$$

donde $dependency\_clash(s, D)$ es una función booleana genérica que inspecciona los ancestros de s en D de acuerdo con un criterio específico para determinar si existe riesgo de no terminación.

Por simplicidad, en el resto de este apartado consideramos que se verifica el test $dependency\_clash(s, D)$ siempre que exista un ancestro comparable del redex seleccionado s en D; nos referimos a este criterio como "test *comp_redex*". Otra aproximación, que investigaremos posteriormente en los experimentos, es comprobar si el redex seleccionado s subsume homeomórficamente a algún ancestro comparable; nos referimos a este criterio como "test *emb_redex*". Como ya se ha mencionado, todas las políticas de planificación son admisibles para un intérprete que implementa *narrowing* perezoso. Hablando de manera informal, la regla de selección dinámica analiza recursivamente la estructura del objetivo y determina el conjunto de posiciones perezosas que admiten una selección *don't-care* al estar dentro de un subtérmino conjuntivo. La regla de selección dinámica escoge exactamente uno de esos subconjuntos de posiciones perezosas (de entre aquéllos que no incurren en un *dependency_clash*); posteriormente, las posiciones seleccionadas deben ser obviamente desplegadas de forma *don't-know*.

Introducimos una regla de desplegado dinámica $U_{dynamic}(t, \mathcal{R})$ que simplemente expande los árboles locales de *narrowing* perezoso de acuerdo con la regla de selección dinámica $\varphi_{dynamic}$.

Definición 8.2.3 (Regla de Desplegado Dinámica)
Definimos la regla de desplegado dinámica $U_{dynamic}(t, \mathcal{R})$ *(o simplemente $U_{dynamic}$) como una aplicación que devuelve una evaluación parcial para el término t en el programa $\mathcal{R}$ obtenida a partir del árbol de* narrowing *local τ construido según la regla de selección dinámica $\varphi_{dynamic}$.*

Dado un conjunto de términos S, denotamos por $U_{dynamic}(S, \mathcal{R})$ la unión de los conjuntos $U_{dynamic}(s, \mathcal{R})$, para s en S.

La "marca" $\perp$ de la Definición 8.2.2 se emplea como una alarma para avisarnos del momento en el que una derivación debe cortarse porque se ha producido una situación en la que el test *dependency_clash* es positivo. Esto es, la expansión de cada rama D del árbol se detiene cuando $\varphi_{dynamic}(t, D) = \{\perp\}$ o el término t (de la hoja) está en hnf. Así pues, la regla de desplegado dinámica explota con ventaja la información proporcionada por las dependencias funcionales dentro del objetivo a la hora de calibrar qué alternativa es mejor.

Ejemplo 47 *Consideremos de nuevo el programa y el objetivo del Example 45. Utilizando la regla de desplegado dinámica $U_{dynamic}$, obtenemos el árbol representado en la Figura 8.3. A partir de ese árbol puede derivarse una definición especializada (recursiva) para la llamada inicial, suponiendo que existe un mecanismo de partición conveniente que permite extraer, de la hoja del árbol, una subconjunción adecuada, como lo es $app(Xs', Y) \approx Ws' \wedge app(Ws', Z) \approx Vs'$, que está cubierta por la llamada inicial en la raíz del árbol (ver Ejemplo 50).*

Antes de finalizar este apartado, es interesante comentar que la Definición 8.2.2 puede generalizarse haciendo que la estrategia de *narrowing* sea un

$$\underline{app(X,Y)} \approx W \;\wedge\; app(W,Z) \approx V$$

$$\vdots \qquad \underline{X' : app(Xs',Y) \approx W} \;\wedge\; app(W,Z) \approx V$$

$$X' \approx W' \;\wedge\; app(Xs',Y) \approx Ws' \;\wedge\; \underline{app(W' : Ws', Z)} \approx V$$

$$X' \approx W' \;\wedge\; app(Xs',Y) \approx Ws' \;\wedge\; \underline{W' : app(Ws', Z) \approx V}$$

$$X' \approx W' \;\wedge\; app(Xs',Y) \approx Ws' \;\wedge\; W' \approx V' \;\wedge\; app(Ws', Z) \approx Vs'$$

Figura 8.3: Mejora del control local para la llamada
$(app(X,Y) \approx W \wedge app(W,Z) \approx V)$.

parámetro de la definición, siempre y cuando se consideren estrategias fuertemente completas. Otro aspecto interesante es que la regla de desplegado dinámica es conservativa, en el sentido de que cuando la llamada inicial no contiene símbolos de función primitivos (y se elige una definición adecuada de la función *dependency_clash*) se comporta como la regla de desplegado estática.

8.2.2 Mejora del Control Global.

En presencia de símbolos de función primitivos como '$\wedge$' o '$\approx$', el empleo de operadores de abstracción que respetan la estructura de los términos (como el de la Definición 3.4.8) no es muy efectivo, ya que la generalización de dos conjunciones (resp. ecuaciones) podría dar lugar a un término de la forma $x \approx y \wedge z$ (resp. $x \approx y$) en la mayoría de los casos, tal y como se ha mostrado anteriormente en los Ejemplos 43 y 45. La drástica solución adoptada en los Capítulos 3 y 4, consistente en descomponer una expresión compleja en subtérminos simples, que contienen únicamente una llamada a función, puede evitar el problema pero tiene la consecuencia negativa de que se pierde parte de la especialización deseada. Esto desemboca en un método que en la práctica sólo realiza especialización monogenética. En este apartado introducimos un operador de abstracción más sofisticado, inspirado en las técnicas de partición de la deducción parcial conjuntiva [83, 128]. El control de nivel global en [83, 128] se basa en el empleo de *árboles globales*, tal y como fueron introducidos en [130, 145]. Si bien esta técnica puede mejorar la especialización en muchos aspectos, aquí se ha preferido utilizar una estrategia de control global más simple pero menos costosa (en términos de representación y gestión de la memoria) basada en el empleo de conjuntos.

Durante el proceso de abstracción, puede resultar necesario partir las expresiones complejas bajo consideración, para continuar el proceso de especialización del mejor modo posible sin incurrir en el riesgo de no terminación. La noción de *términos que mejor se ajustan* (*best matching terms*), o simplemente *mejores ajustes*, para simplificar, tiene como objetivo evitar la pérdida de especialización debida a una excesiva generalización. Esta noción, que se define a continuación, es una generalización apropiada a nuestras necesidades del concepto de *conjunción que mejor se ajusta* (*best matching conjunction*) que aparece en [83]. Antes de formalizar este concepto, necesitamos de una noción auxiliar.

Definición 8.2.4 (Elemento Mínimamente General)
Dado un conjunto $W = \{w_1, \ldots, w_n\}$, se dice que w_i es un elemento mínimamente general *de W si y sólo si, para todo $w_j \in W$, con $j \neq i$, no existe una substitución θ, diferente de la substitución identidad o de un renombramiento, tal que $w_j = \theta(w_i)$ (i.e., si no existe otro elemento, diferente de w_i, que sea una instancia estricta de w_i).*

Ejemplo 48 *Sea el conjunto $W = \{f(g(X)), f(g(Y)), f(Z)\}$. El elemento $f(Z)$ no es mínimamente general ya que tanto $f(g(X))$ como $f(g(Y))$ son instancias estrictas de $f(Z)$. Sin embargo, $f(g(X))$ es un elemento mínimamente general ya que, aunque $f(g(X)) = \theta(f(g(Y)))$, con $\theta = \{Y/X\}$, la substitución θ es un renombramiento. Lo mismo puede decirse acerca de $f(g(Y))$.*

Definición 8.2.5 (Mejores Ajustes)
Sea el conjunto de términos $S = \{s_1, \ldots, s_n\}$ y un término t. Consideremos el conjunto de términos $W = \{w_i \mid \langle w_i, \{\theta_{i1}, \theta_{i2}\}\rangle = msg(\{s_i, t\}),\ i = 1, \ldots, n\}$. Los mejores ajustes *$BMT(S,t)$ para t en S son aquéllos términos $s_j \in S$ tales que el correspondiente w_j es un elemento mínimamente general de W.*

Nótese que la función BMT devuelve un subconjunto de términos del conjunto S que mejor se ajustan al término t y, en general, $|BMT| > 1$ lo que introduce cierto grado de indeterminismo. El siguiente ejemplo clarifica definición la anterior.

Ejemplo 49 *Sea $S = \{f(g(X)), f(g(a)), f(Z)\}$ y $t \equiv f(g(b))$. Para computar los mejores ajustes para t en S, primero obtenemos el conjunto:*

$$\begin{array}{rcl} W & = & \{msg(\{f(g(X)), f(g(b))\}), msg(\{f(g(a)), f(g(b))\}), \\ & & msg(\{f(Z), f(g(b))\})\} \;=\; \{f(g(X)), f(g(Y)), f(Z)\}. \end{array}$$

Después, hallamos los elementos mínimamente generales de W, que son $f(g(X))$ y $f(g(Y))$, a los cuales les corresponden los términos de S, $f(g(X))$ y $f(g(a))$, respectivamente. Así se obtine que $BMT(S,t) = \{f(g(X)), f(g(a))\}$. La Figura 8.4 ilustra el proceso.

A continuación se define un nuevo operador de abstracción que permite una generalización adecuada de los términos que contienen símbolos de función primitivos haciendo uso del concepto de "mejores ajustes". La noción de BMT se emplea en dos fases del proceso de abstracción (ver la Definición 8.2.6:

1. Cuando se selecciona, de entre los elementos de S, el término que mejor se ajusta con aquéllos respecto de los cuales existen problemas de subsumción, procediéndose entonces a la generalización.

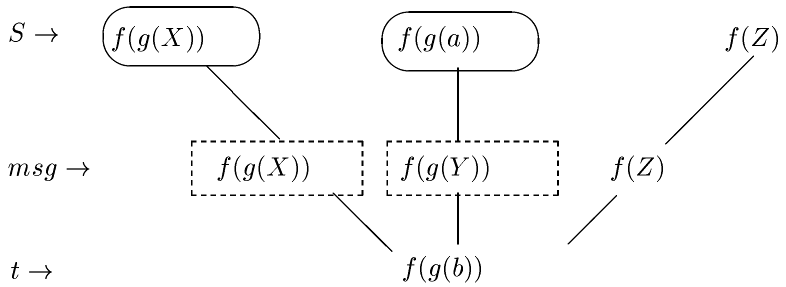

Figura 8.4: Mejores ajustes para el término $t \equiv f(g(b))$ en el conjunto $S = \{f(g(X)), f(g(a)), f(Z)\}$.

2. Cuando se determina si la nueva llamada t encabezada por un símbolo de función primitivo puede añadirse de forma segura al conjunto actual de llamadas especializadas S o si debe partirse. Si el término s es un mejor ajuste del término t en T y los términos s y t tienen la misma estructura, entonces no hay peligro al generalizar, ya que no se perderá demasiada información. Por el contrario, si los términos s y t no comparten la misma estructura, es mejor partir convenientemente el término t antes de generalizar (aunque en la definición no se especifica como) .

Definición 8.2.6 (Operador de Abstracción Concreto)
Sean S y T dos conjuntos de términos. Se define el operador abstract$_\trianglelefteq(S,T)$ inductivamente como sigue: $abstract_\trianglelefteq(S,T) =$

$$\left\{\begin{array}{ll} S & \texttt{si } T \equiv \emptyset \texttt{ o } (T \equiv \{t\},\ t \in \mathcal{X}) \\ abstract_\trianglelefteq(\ldots abstract_\trianglelefteq(S,\{t_1\}),\ldots,\{t_n\}) & \texttt{si } T \equiv \{t_1,\ldots,t_n\}, n > 0 \\ abstract_\trianglelefteq(S,\{t_1,\ldots,t_n\}) & \texttt{si } T \equiv \{t\},\ t \equiv c(t_1,\ldots,t_n),\ c \in \mathcal{C} \\ abs\_def(S,T',t) & \texttt{si } T \equiv \{t\},\ \mathcal{H}ead(t) \in \mathcal{D} \\ abs\_prim(S,T',t) & \texttt{si } T \equiv \{t\},\ \mathcal{H}ead(t) \in \mathcal{P} \end{array}\right.$$

donde $T' = \{s \in S \mid \mathcal{H}ead(s) = \mathcal{H}ead(t) \wedge s \trianglelefteq t\}$. *Las funciones abs_def y abs_prim se definen como sigue:*

$$\begin{array}{lcl} abs\_def(S,\emptyset,t) & = & abs\_prim(S,\emptyset,t) \ = S \cup \{t\} \\ abs\_def(S,T,t) & = & abstract_\trianglelefteq(S \setminus \{s\}, \{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)) \\ & & \quad \texttt{si } \langle w, \{\theta_1,\theta_2\}\rangle = msg(\{s,t\}),\ \texttt{con } s \in BMT(T,t) \\ abs\_prim(S,T,t) & = & \left\{\begin{array}{l} abs\_def(S,T,t) \ \texttt{ si } \exists s \in BMT(T,t) \texttt{ t.q. } def(t) = def(s) \\ abstract_\trianglelefteq(S,\{t_1,t_2\}) \ \texttt{ de otro modo, donde } t \stackrel{\triangle}{=} p(t_1,t_2) \end{array}\right. \end{array}$$

siendo $def(t)$ una secuencia formada por los símbolos de función definidos *que aparecen en t en orden lexicográfico y donde $\stackrel{\triangle}{=}$ denota la igualdad de términos, salvo reordenamiento de los elementos en una conjunción.*

Esencialmente, el modo en el que actúa el operador de abstracción es simple. Distinguiremos entre los siguientes casos, dependiendo de que la expresión t considerada sea: i) una variable, ii) un término encabezado por un símbolo constructor, iii) por un símbolo de función definido, o iv) por un símbolo primitivo. Las acciones que realiza el operador de abstracción son, respectivamente: i) ignorar la llamada, dejando el conjunto de términos evaluados parcialmente S inalterado, ii) inspeccionar recursivamente los subtérminos, iii) si no hay problemas de subsumción con los términos del conjunto S se añade la llamada t al conjunto S; si no, se generaliza la llamada t con respecto a alguno de sus mejores ajustes, inspeccionando recursivamente el msg w y los subtérminos de θ_1, θ_2 no cubiertos por la generalización, y iv) proceder de la misma forma que en (iii) pero considerando la posibilidad de partir la llamada t antes de generalizarla cuando no existe un mejor ajuste s con la misma estructura que t, i.e., cuando $def(t) \neq def(s)$ (lo cual evita que se pierda algún símbolo de función debido a la aplicación prematura del operador msg).

Para finalizar este apartado, presentamos un ejemplo que atestigua que el operador de abstracción $abstract_{\trianglelefteq}$ se comporta bien con relación al Ejemplo 47.

Ejemplo 50 *Consideremos de nuevo el árbol dibujado en la Figura 8.3. Aplicando el Algoritmo 1 a este problema, se realizan las siguientes llamadas al operador* $abstract_{\trianglelefteq}$*:*

$$
\begin{aligned}
T_0 &= abstract_{\trianglelefteq}(\emptyset, \{app(X,Y) \approx W \ \wedge \ app(W,Z) \approx V\}) \\
&= \{app(X,Y) \approx W \ \wedge \ app(W,Z) \approx V\} \\[1ex]
T_1 &= abstract_{\trianglelefteq}(T_0, \{(X' \approx Y' \ \wedge \ app(X_s,Y_s) \approx Z_s), \\
&\qquad\qquad (X' \approx W' \ \wedge \ app(X'_s,Y) \approx W'_s \ \wedge \\
&\qquad\qquad W' \approx V' \ \wedge \ app(W'_s,Z) \approx V'_s)\}) \\
&= \{(app(X,Y) \approx W \ \wedge \ app(W,Z) \approx V), \\
&\quad\ (X' \approx Y' \ \wedge \ app(X_s,Y_s) \approx Z_s)\} \\[1ex]
T_2 &= T_1
\end{aligned}
$$

Por consiguiente, después de dos iteraciones, se obtiene el siguiente conjunto de términos evaluados parcialmente:

$$\{(app(X,Y) \approx W \ \wedge \ app(W,Z) \approx V), (X' \approx Y' \ \wedge \ app(X_s,Y_s) \approx Z_s)\}.$$

Asociando el renombramiento independiente $dapp(X,Y,W,Z,V)$ *a la llamada especializada* $app(X,Y) \approx W \ \wedge \ app(W,Z) \approx V$*, el método obtiene una regla recursiva de la forma:*

$dapp(X\!:\!X_s, Y, W\!:\!W_s, Z, V\!:\!V_s) \ \rightarrow \ X \approx W \wedge W \approx V \wedge dapp(X_s, Y, W_s, Z, V_s)$

que contiene la especialización óptima esperada para este ejemplo.

8.3 Terminación del método NPE.

En el presente apartado se estudia la terminación del proceso de NPE cuando se emplea el operador de desplegado $U_{dynamic}$ y el de abstracción $abstract_{\trianglelefteq}$. Primero nos centramos en la terminación local.

8.3.1 Terminación Local.

El siguiente resultado es una consecuencia sencilla de la completitud fuerte de la estrategia de *narrowing* perezoso y de que el número de símbolos distintos en la signatura es finito.

Proposición 8.3.1 *Sea $\mathcal{R}$ un programa y t un objetivo. Entonces $U_{dynamic}(t,\mathcal{R})$ produce un árbol de* narrowing *perezoso finito (posiblemente incompleto) para t en $\mathcal{R}$.*

Prueba. La regla de desplegado $U_{dynamic}(t,\mathcal{R})$ expande los árboles locales de acuerdo a la estrategia de *narrowing* perezoso dinámica. Evidentemente, la estrategia de selección $\varphi_{dynamic}(t,\mathcal{R})$ produce un árbol de *narrowing* perezoso (posiblemente incompleto) ya que, en cada paso, se consideran todas las posiciones que deben explotarse de forma *don't know* y solamente se desechan aquéllas que pueden seleccionarse en forma *don't-care*, conforme a la propiedad de completitud fuerte de que goza el *narrowing* perezoso. El empleo de la función $dependency\_clash(t|_p,\mathcal{D})$ asegura la finitud del árbol de *narrowing* construido. Comprobamos que las dos clases diferentes de tests considerados, *comp_redex* y *emb_redex*, son apropiadas y dan lugar a un criterio de parada adecuado.

1. *comp_redex*: test en el que se comprueba si existe un ancestro comparable del redex seleccionado en $\mathcal{D}$.
   Procedemos por contradicción. Supongase que construimos un árbol de *narrowing* perezoso infinito para el término t en $\mathcal{R}$ utilizando $U_{dynamic}$. Ya que el término t es finito, el número de redexes de t es finito también y el árbol de *narrowing* de t en $\mathcal{R}$ está ramificado de forma finita. Entonces, por el lema de König (Lema 2.4.2), existe una rama infinita en el árbol. Ahora el resultado se sigue trivialmente ya que no existe una rama infinita que pueda contener redexes dependientes encabezados por símbolos de función diferentes a menos que la signatura sea infinita.

2. *emb_redex*: test de subsumción homeomórfica sobre ancestros comparables del redex selecionado.
   En este caso, también procedemos por contradicción. Supongamos que existe una rama infinita en el árbol de *narrowing* perezoso. Ya que tanto la signatura como el término t son finitos, existe un número finito de subsecuencias de subtérminos comparables dependientes en la rama. Por consiguiente, al menos una de esas subsecuencias será infinita. Dado que las subsecuencias de subtérminos comparables dependientes construidos por $U_{dynamic}$, utilizando el test *emb_redex*, son subsecuencias de no subsumción, tenemos una secuencia infinita de términos $t_1, t_2, \ldots$ tal que ningún término t_j subsume a un término precedente t_i, con $i < j$, lo que contradice el teorema de Kruskal (Teorema 3.4.2).

$\square$

8.3.2 Terminación Global.

En este apartado nos centramos en la terminación global del proceso de NPE. Para demostrar los resultados de esta sección, necesitamos introducir algunas notaciones y conceptos previos.

La *profundidad* de un término t es el número máximo de símbolos anidados en t.

Definición 8.3.2 (Profundidad)
La profundidad *de un término t, denotado por $depth(t)$, se define inductivamente como sigue:*

$$depth(t) = \begin{cases} 1 & \texttt{si } t \in \mathcal{X} \\ 1 + max(\{depth(t_1), \ldots, depth(t_n)\}) & \texttt{si } t \equiv f(t_1, \ldots, t_n), f \in \mathcal{F} \end{cases}$$

donde la función max calcula el máximo de un conjunto de números.

Para facilitar las pruebas, introducimos el concepto de *complejidad* $\mathcal{M}_S$ de un conjunto de términos S.

Definición 8.3.3 (Complejidad)
Sea S un conjunto de términos. La complejidad $\mathcal{M}_S$ del conjunto S es el multiconjunto finito de números naturales correspondiente a la profundidad de los elementos de S: $\mathcal{M}_S = \{depth(s) \mid s \in S\}$.

Como se indicó en el Apartado 2.3, el *orden multiconjunto*, $<_{mul}$, es un orden sobre el conjunto de los multiconjuntos finitos de números naturales $M(\mathbb{N})$. El orden multiconjunto sobre $M(\mathbb{N})$ es bien fundado (i.e., no existen en él cadenas decrecientes infinitas) y, por consiguiente, el método de prueba por inducción completa puede aplicarse sobre $M(\mathbb{N})$; muchas de las pruebas de este apartado se han realizado empleando esta técnica.

El siguiente lema establece la transitividad de la relación de cierre (Definición 3.3.3) y es necesaria para probar la Proposición 8.3.5.

Lema 8.3.4 *Si el término t es S_1-cerrado, y S_1 es S_2-cerrado, entonces t es S_2-cerrado.*

Prueba. Por inducción estructural sobre t. Ver el Lema B.1 de [14]. □

Proposición 8.3.5 *La función $abstract_{\trianglelefteq}$ es un operador de abstracción en el sentido de la Definición 3.3.11.*

Prueba. Siguiendo la Definición 3.3.11 de operador de abstracción, necesitamos probar:

1. Si $s \in abstract(S,T)$, entonces existe un término $t \in (S \cup T)$ tal que $t|_p = \theta(s)$ para alguna posición p y substitución θ, y

2. Para todo $t \in (S \cup T)$, t es cerrado con respecto al conjunto de términos $abstract(S,T)$.

La condición (1) se satisface trivialmente porque el operador de abstracción se basa en el empleo de *msg*'s, lo que impide la introducción de nuevos símbolos de función que no aparezcan en S o T.

Ahora nos centramos en la prueba de la condición (2) empleando inducción *bien fundada* sobre la complejidad de $S \cup T$.

Si $S \cup T \equiv \emptyset$, entonces $abstract_{\trianglelefteq}(\emptyset, \emptyset) = \emptyset$, y la prueba está hecha ya que la propiedad (2) se cumple por vacuidad.

Consideremos el caso inductivo $S \cup T \neq \emptyset$. Si $T \equiv \emptyset$, entonces $abstract_{\trianglelefteq}(S, \emptyset) = S$, y la propiedad (2) se cumple trivialmente. Supongamos por lo tanto que el conjunto T no es vacío. Entonces, $T \equiv T_0 \cup \{t\}$, con $T_0 = \{t_1, \ldots, t_{n-1}\}$. Por hipótesis de inducción, la propiedad (2) se cumple para todo par de conjuntos $S_{\trianglelefteq}$ y $T_{\trianglelefteq}$ tales que $\mathcal{M}_{S_{\trianglelefteq} \cup T_{\trianglelefteq}} <_{mul} \mathcal{M}_{S \cup T}$. Siguiendo la definición de $abstract_{\trianglelefteq}$, consideramos cuatro casos:

1. $t \in \mathcal{X}$.
$$\begin{array}{lcl} S' & = & abstract_{\trianglelefteq}(S, T) \\ & = & abstract_{\trianglelefteq}(S, T_0 \cup \{t\}) \\ & = & abstract_{\trianglelefteq}(abstract_{\trianglelefteq}(\ldots abstract_{\trianglelefteq}(S, \{t_1\}), \ldots, \{t_{n-1}\}), \{t\}) \\ & = & abstract_{\trianglelefteq}(abstract_{\trianglelefteq}(S, T_0), \{t\}) \\ & = & abstract_{\trianglelefteq}(S, T_0). \end{array}$$
Ya que $\mathcal{M}_{S \cup T_0} <_{mul} \mathcal{M}_{S \cup T_0 \cup \{t\}} = \mathcal{M}_{S \cup T}$ entonces, por hipótesis de inducción, $S \cup T_0$ es cerrado con relación a los términos de S' y, dado que una variable siempre es cerrada con respecto a un conjunto (por definición de cierre), también lo es $S \cup T_0 \cup \{t\} = S \cup T$.

2. $t \equiv c(s_1, \ldots, s_m)$, $c \in \mathcal{C}$, $m \geq 0$.
$$\begin{array}{lcl} S' & = & abstract_{\trianglelefteq}(S, T) \\ & = & abstract_{\trianglelefteq}(S, T_0 \cup \{t\}) \\ & = & abstract_{\trianglelefteq}(abstract_{\trianglelefteq}(S, T_0), \{c(s_1, \ldots, s_m)\}) \\ & = & abstract_{\trianglelefteq}(abstract_{\trianglelefteq}(S, T_0), \{s_1, \ldots, s_m\}) \\ & = & abstract_{\trianglelefteq}(\ldots abstract_{\trianglelefteq}(abstract_{\trianglelefteq}(S, T_0), \{s_1\}), \ldots, \{s_m\}) \\ & = & abstract_{\trianglelefteq}(S, T_0 \cup \{s_1, \ldots, s_m\}). \end{array}$$
Ya que $\mathcal{M}_{S \cup T_0 \cup \{s_1, \ldots, s_m\}} <_{mul} \mathcal{M}_{S \cup T_0 \cup \{c(s_1, \ldots, s_m)\}} = \mathcal{M}_{S \cup T}$ entonces, por hipótesis de inducción, $S \cup T_0 \cup \{s_1, \ldots, s_m\}$ es cerrado con respecto al conjunto de términos S' y, por definición de cierre, también lo es $S \cup T_0 \cup \{c(s_1, \ldots, s_m)\} = S \cup T$.

3. $t \equiv f(s_1, \ldots, s_m)$, $f \in \mathcal{D}$, $m \geq 0$.
Entonces, por definición de $abstract_{\trianglelefteq}$,
$$\begin{array}{lcl} S' & = & abstract_{\trianglelefteq}(S, T) \\ & = & abstract_{\trianglelefteq}(S, T_0 \cup \{t\}) \\ & = & abstract_{\trianglelefteq}(abstract_{\trianglelefteq}(S, T_0), \{t\}) \\ & = & abs\_def(abstract_{\trianglelefteq}(S, T_0), E, t). \end{array}$$
donde $E = \{s \in abstract_{\trianglelefteq}(S, T_0) \mid \mathcal{H}ead(s) = \mathcal{H}ead(t) \wedge s \trianglelefteq t\}$.
Supongamos que $abstract_{\trianglelefteq}(S, T_0)$ devuelve un conjunto $S'' \equiv \{s_1, \ldots, s_p\}$, con $p \geq 1$ (ya que el caso para el que $abstract_{\trianglelefteq}(S, T_0)$ es $\emptyset$ es inmediato).

Aquí distinguimos dos casos, que se corresponden con los de la definición de la función *abs_def* en $abstract_{\trianglelefteq}$:

(a) $E = \emptyset$.

$$\begin{array}{rcl} S' & = & abs\_def(abstract_{\trianglelefteq}(S, T_0), E, t) \\ & = & S'' \cup \{t\} \end{array}$$

Ya que $\mathcal{M}_{S \cup T_0} <_{mul} \mathcal{M}_{S \cup T}$ entonces, por hipótesis de inducción, $S \cup T_0$ es cerrado con respecto a los términos de $S'' = abstract_{\trianglelefteq}(S, T_0)$; por consiguiente, también lo es con respecto a $S'' \cup \{t\}$. Así pues, trivialmente, dado que t está en $S'' \cup \{t\}$, el conjunto $S \cup T = S \cup T_0 \cup \{t\}$ es cerrado con respecto a $S'' \cup \{t\} = S'$.

(b) $E \neq \emptyset$.

$$\begin{array}{rcl} S' & = & abs\_def(abstract_{\trianglelefteq}(S, T_0), E, t) \\ & = & abstract_{\trianglelefteq}(\{s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_p\}, S^*) \\ & = & abstract_{\trianglelefteq}(S''', S^*) \end{array}$$

donde existe un $i \in \{1, \ldots, p\}$ con $msg(\{s_i, t\}) = \langle w, \{\theta_1, \theta_2\}\rangle$, y $S^* = \{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$.

Primero, tenemos que $\mathcal{M}_{S \cup T_0} <_{mul} \mathcal{M}_{S \cup T_0 \cup \{t\}}$ y entonces, por hipótesis de inducción, $S \cup T_0$ es S''-cerrado y, por consiguiente, $S \cup T_0$ es $S''' \cup S^*$-cerrado también (ya que todo término cubierto por s_i está cubierto por la generalización más específica w de s_i y t).

Ahora necesitamos probar que $\mathcal{M}_{S''' \cup S^*} <_{mul} \mathcal{M}_{S''' \cup \{s_i\} \cup \{t\}}$ utilizando la definición de orden multiconjunto. Distinguimos dos casos:

i. $depth(w) = depth(s_i)$.
Sean M y M' las complejidades de $S''' \cup S^*$ y $S''' \cup \{s_i\} \cup \{t\}$, respectivamente. Sean X y X' las complejidades de $\mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ y $\{t\}$, respectivamente. Sean X_1 y X_2 las complejidades de $\mathcal{R}an(\theta_1)$ y $\mathcal{R}an(\theta_2)$, respectivamente.
Puesto que t subsume a s_i, entonces es immediato comprobar que $depth(s_i) \leq depth(t)$. Ya que s_i y w son comparables y $s_i = \theta_1(w)$, entonces $depth(d) < depth(s_i)$ para todo $d \in \mathcal{R}an(\theta_1)$. Por consiguiente, $depth(d) < depth(t)$ para todo $d \in \mathcal{R}an(\theta_1)$ y, de aquí, para todo $n \in X_1$, existe un $n' \in X'$ tal que $n < n'$.
Dado que t y w son términos comparables y $t = \theta_2(w)$, entonces se cumple que para todo $n \in X_2$, existe un $n' \in X'$ tal que $n < n'$, con lo que finaliza la prueba.

ii. $depth(w) < depth(s_i)$.
Este caso puede probarse de manera similar, considerando que X y X' denotan las complejidades de $\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$ y $\{s_i\} \cup \{t\}$, respectivamente.

Ya que hemos probado que $\mathcal{M}_{S''' \cup S^*} <_{mul} \mathcal{M}_{S''' \cup \{s_i\} \cup \{t\}}$, y puesto que $\mathcal{M}_{S''' \cup \{s_i\} \cup \{t\}} <_{mul} \mathcal{M}_{S \cup T}$, por hipótesis de inducción, $S''' \cup S^*$ es cerrado con respecto a los términos de S'. De este último resultado y del resultado afirmado en primer lugar de que $S \cup T_0$ es $S''' \cup S^*$-cerrado, aplicando el Lema 8.3.4, deducimos que $S \cup T_0$ es también S'-cerrado. Finalmente, como t es cerrado con relación a $\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2)$, por definición de msg, el resultado se sigue usando de nuevo el Lema 8.3.4.

4. $\mathcal{H}ead(t) \in \mathcal{P}$.

$$S' \quad = \quad abs\_prim(abstract_{\trianglelefteq}(S, T_0), E, t).$$

Consideramos dos alternativas, según exista un término s adecuado perteneciente al conjunto de los mejores ajustes $BMT(E, t)$ o no. En la primera, el cálculo de abs_prim es equivalente a la aplicación de abs_def y la prueba es completamente análoga a la del caso 3. En la segunda, se decide la eliminación del símbolo primitivo en cabeza de t y se parte el término t en dos componentes t_1 y t_2, con lo cual

$$S' \quad = \quad (abstract_{\trianglelefteq}(S, T_0, \{t_1, t_2\}),$$

y la prueba prosigue de forma completamente análoga a la del caso 2.

□

Proposición 8.3.6 *El cómputo del operador de abstracción de la definición 8.2.6 termina.*

Prueba. Se prueba que el cómputo de $abstract_{\trianglelefteq}(S, T)$ termina por inducción bien fundada sobre $S \cup T$. La estructura de la prueba es similar a la de la Proposición 8.3.5.

Si $S \cup T \equiv \emptyset$, entonces $abstract_{\trianglelefteq}(\emptyset, \emptyset) = \emptyset$ y el cómputo termina, concluyéndose el resultado.

Consideremos ahora el caso inductivo $S \cup T \neq \emptyset$ y supongamos que T no es vacío (de otro modo, la prueba resulta inmediata). Entonces, $T \equiv T_0 \cup \{t\}$, con $T_0 = \{t_1, \ldots, t_{n-1}\}$. Por hipótesis de inducción, se cumple la propiedad para todo $S_{\trianglelefteq}$ y para todo $T_{\trianglelefteq}$ tal que $\mathcal{M}_{S_{\trianglelefteq} \cup T_{\trianglelefteq}} <_{mul} \mathcal{M}_{S \cup T}$. Basándonos en la definición de $abstract_{\trianglelefteq}$, consideramos los siguientes casos:

1. $t \in \mathcal{X}$.

$$S' \quad = \quad abstract_{\trianglelefteq}(S, T_0).$$

Ya que $\mathcal{M}_{S \cup T_0} <_{mul} \mathcal{M}_{S \cup T_0 \cup \{t\}} = \mathcal{M}_{S \cup T}$ entonces, por hipótesis de inducción, el cómputo de $S' = abstract_{\trianglelefteq}(S, T)$ termina.

2. $t \equiv c(s_1, \ldots, s_m)$, $c \in \mathcal{C}$, $m \geq 0$.

$$S' \quad = \quad abstract_{\trianglelefteq}(S, T_0 \cup \{s_1, \ldots, s_m\})$$

Ya que, $\mathcal{M}_{S \cup T_0 \cup \{s_1, \ldots, s_m\}} <_{mul} \mathcal{M}_{S \cup T_0 \cup \{c(s_1, \ldots, s_m)\}} = \mathcal{M}_{S \cup T}$ entonces, por hipótesis de inducción, el cómputo de S' termina.

3. $t \equiv f(s_1, \ldots, s_m)$, $f \in \mathcal{D}$, $m \geq 0$.

$$S' = abs\_def(abstract_{\trianglelefteq}(S, T_0), E, t).$$

donde $E = \{s \in abstract_{\trianglelefteq}(S, T_0) \mid \mathcal{H}ead(s) = \mathcal{H}ead(t) \wedge s \trianglelefteq t\}$

Supongamos que $abstract_{\trianglelefteq}(S, T_0)$ devuelve $S'' \equiv \{s_1, \ldots, s_p\}$, con $p \geq 1$ (ya que el caso en el que $abstract_{\trianglelefteq}(S, T_0) = \emptyset$ es inmediato). Aquí distinguimos dos casos, que se corresponden con los dos modos de computar la función abs_def en la definición de $abstract_{\trianglelefteq}$, dados por $abs\_def(S'', E, t) =$

   (a) $abstract_{\trianglelefteq}(S, T_0) \cup \{t\}$ cuando $E = \emptyset$.
   Ya que $\mathcal{M}_{S \cup T_0} <_{mul} \mathcal{M}_{S \cup T}$ entonces, por hipótesis de inducción, el cómputo de $abstract_{\trianglelefteq}(S, T_0)$ termina y lo mismo sucede con el de S'.

   (b) $abstract_{\trianglelefteq}(\{s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_p\}, S)$, donde existe un $i \in \{1, \ldots, p\}$ tal que $msg(\{s_i, t\}) = \langle w, \{\theta_1, \theta_2\}\rangle$ y $S = (\{w\} \cup \mathcal{R}an(\theta_1) \cup \mathcal{R}an(\theta_2))$, cuando $E \neq \emptyset$.
   Utilizando el mismo argumento que en el caso (b) de la Proposición 8.3.5 puede probrarse que $\mathcal{M}_{S'' \cup S} <_{mul} \mathcal{M}_{S'' \cup \{s_i\} \cup \{t\}}$. Ahora, nuevamente usando la hipótesis de inducción, podemos asegurar que el cómputo de S' termina.

4. $\mathcal{H}ead(t) \in \mathcal{P}$.

$$S' = abs\_prim(abstract_{\trianglelefteq}(S, T_0), E, t).$$

Ya que el resultado de abs_prim es equivalente a la aplicación de abs_def o a la eliminación del símbolo más externo de t cuando se decide partir el término t, entonces la prueba de este caso es completamente análoga a la de los casos 2 y 3.

□

Los siguientes lemas son auxiliares para la prueba del Teorema 8.3.10. Comenzamos definiendo la propiedad de no subsumción.

Definición 8.3.7 (Propiedad de no Subsumción)
Sea S el conjunto de términos obtenidos en un paso de cómputo del Algoritmo 1. Se dice que S satisface la propiedad de no subsumción si se cumple que:

$$\forall l, k \text{ en } \{1, \ldots, n\}, \text{ con } l < k \;.\; (\mathcal{H}ead(t_l) = \mathcal{H}ead(t_k) \Rightarrow t_l \not\trianglelefteq t_k)$$

donde $(t_1, \ldots, t_n)$ es la secuencia de elementos construida al disponer los términos (de S) en el orden de su inclusión en el conjunto S.

Lema 8.3.8 *Sea S un conjunto de términos que satisface la propiedad de no subsumción, y sea T un conjunto arbitrario de términos. Entonces, $S' = abstract_{\trianglelefteq}(S, T)$ satisface la propiedad de no subsumción.*

Prueba. Se prueba que el cómputo de $abstract_{\trianglelefteq}(S,T)$ satisface la propiedad de no subsumción por inducción bien fundada sobre $S \cup T$. La prueba es idéntica a la de la Proposición 8.3.6.

$\square$

Lema 8.3.9 *Sea $T_0, \ldots, T_n$ la secuencia de conjuntos obtenida al ejecutar el Algoritmo 1, donde $T_{i+1} = abstract_{\trianglelefteq}(T_i, \mathcal{R}^i_{calls})$, siendo $\mathcal{R}^i = U_{dynamic}(T_i, \mathcal{R})$ para un programa $\mathcal{R}$. Entonces, para cada $i = 0, \ldots, n$, T_i satisface la propiedad de no subsumción.*

Prueba. Por inducción sobre el número n de pasos en la computación.

Si $n = 0$, entonces $T_0 = \emptyset$ y la prueba se sigue por vacuidad.

Consideremos el caso inductivo $n > 0$. Por hipótesis de inducción, la propiedad se cumple para todo T_j tal que $j < n$. Es inmediato que la propiedad de no subsumción se cumple para

$$T_n = abstract_{\trianglelefteq}(T_{n-1}, S)$$

por aplicación del Lema 8.3.8 ya que, por hipótesis de inducción, T_{n-1} satisface la propiedad de no subsumción. $\square$

El siguiente resultado establece la corrección y la terminación del proceso global de NPE utilizando el operador de desplegado de la Definición 8.2.3 y el operador de abstracción de la Definición 8.2.6.

Teorema 8.3.10 *El Algoritmo 1 termina para la regla de desplegado $U_{dynamic}$ y el operador de abstracción $abstract_{\trianglelefteq}$.*

Prueba.

Los hechos siguientes son suficientes para la prueba del teorema:

- El número de conjuntos de términos incomparables que pueden formarse empleando un número finito de símbolos de función es finito.
- Por el teorema de Kruskal y la propiedad de no subsumción establecida anteriormente (Lema 8.3.9), los subconjuntos de términos comparables de cualquier conjunto computado por el Algoritmo 1 son finitos.
- Por la Proposición 8.3.1, los conjuntos $\mathcal{R}^i_{calls}$, $i \geq 1$, son todos finitos y, por consiguiente, el cómputo de cada conjunto de términos termina en un tiempo finito.
- La función $abstract_{\trianglelefteq}$ está bien definida, en el sentido de que el cómputo de una nueva configuración siempre termina (Proposición 8.3.6).

$\square$

En el siguiente apartado se comentan una serie de resultados experimentales obtenidos con un evaluador parcial que implementa las estrategias presentadas en este trabajo.

8.4 Resultados Experimentales.

Los refinamientos elaborados en los anteriores apartados se han incorporado al prototipo de evaluador parcial dirigido por *narrowing* INDY[4] que se describe en [4, 13].

Para comprobar la viabilidad práctica de nuestra aproximación, hemos evaluado la eficiencia y la calidad de la especialización conseguida al extender la implementación de INDY para incluir la regla de desplegado dinámica de la Definición 8.2.3 y el operador de abstracción de la Definición 8.2.6. Los programas de prueba utilizados en el análisis han sido: `applast`, que añade un elemento al final de una lista y devuelve el último elemento de la lista resultante de esta operación; `double_app`, el programa del Ejemplo 45; `double_flip`, que invierte la estructura de un árbol dos veces, devolviendo de nuevo el árbol original como resultado; `fibonacci`, la conocida función de Fibonacci; `heads&legs`, que computa el numero de cabezas y patas que corresponden a un número dado de pájaros y gatos; `match-app`, una versión extremamente ingenua de un algoritmo de búsqueda de patrones en cadenas, basado en el empleo de `append`; `match-kmp`, un programa algo menos ingenuo de búsqueda de patrones en cadenas; `maxlength`, que devuelve como resultado el máximo y la longitud de una lista; `palindrome`, un programa para comprobar si una lista dada es o no un palíndromo; y `sorted_bits`, el programa del Ejemplo 43. Algunos de estos programas son ejemplos típicos de programas de prueba extraidos del ámbito de la deducción parcial (ver [124, 127]) y adaptados a una sintaxis lógico-funcional, mientras que otros provienen de la literatura asociada al área de la transformación de programas funcionales, tal como la supercompilación positiva [108], las transformaciones de plegado/desplegado [48, 55] y la deforestación [203]. En el Apéndice B aparece el código de los programas utilizados en los experimentos, así como las llamadas evaluadas parcialmente.

Para la especialización de los programas de prueba se han considerado las siguientes opciones:

- Estrategia de evaluación: Todas las pruebas se han realizado empleando la estrategia de *narrowing* perezoso (sin normalización)[5].

- Regla de desplegado: Hemos comprobado tres alternativas para el control local:

    1. `emb_goal`: Expande las derivaciones mientras los nuevos objetivos no subsumen un objetivo[6] comparable que haya aparecido previamente en la misma rama del árbol de desplegado;

    2. `emb_redex`: La regla de desplegado concreta definida en el Apartado 8.2.1 que implementa el test *dependency_clash* comprobando la

[4]En el Apéndice A puede encontrarse más información sobre el sistema INDY.

[5]No se ha usado *narrowing* perezoso uniforme por simplicidad, al no ser uniformes los programas de partida.

[6]La palabra "objetivo" hace referencia al término completo y no a un redex del mismo.

Tabla 8.1: Influencia de las nuevas estrategias de control en la mejora de la eficiencia de los programas especializados.

	Original		`emb_goal`		`emb_redex`		`comp_redex`	
Benchmarks	Rw	RT	Rw	Speedup	Rw	Speedup	Rw	Speedup
`applast`	10	90	13	1.32	28	2.20	13	1.10
`double_app`	8	106	39	1.63	61	1.28	15	3.12
`double_flip`	8	62	26	1.51	17	1.55	17	1.55
`fibonacci`	5	119	11	1.19	7	1.08	7	1.08
`heads&legs`	8	176	24	4.63	22	2.41	21	2.48
`match-app`	8	201	12	1.25	20	2.75	23	2.79
`match-kmp`	12	120	14	3.43	14	3.64	13	3.43
`maxlength`	14	94	51	1.17	20	1.27	18	1.25
`palindrome`	10	119	19	1.25	10	1.35	10	1.35
`sorted_bits`	8	110	16	1.15	31	2.89	10	2.68
Promedio	9.1	119.7	22.5	1.85	23	2.04	14.7	2.08
$T_{\rm INDY}$ medio			1881		7441		5788	

subsumción homeomórfica entre ancestros comparables de los redexes seleccionados para asegurar la finitud del proceso de desplegado (se debe notar que esta opción difiere de la opción `emb_goal` en que `emb_redex` implementa una planificación dinámica, frente a la regla de selección estática implementada por `emb_goal`, y que el test de subsumción homeomórfica se realiza sobre simples redexes en vez de sobre objetivos completos);

3. `comp_redex`: La regla de desplegado concreta definida en la Sección 8.2.1, pero empleando la forma más simple del test de *dependency_clash* en la que la existencia de ancestros comparables de los redexes seleccionados detiene el proceso de desplegado.

- Operador de abstracción: El proceso de abstracción se realiza siempre utilizando el operador abstracción concreto de la Definición 8.2.6.

La Tabla 8.1 resume los resultados de los experimentos. Las dos primeras columnas miden el número de reglas de reescritura (Rw) y el tiempo absoluto de ejecución (RT) para los programas originales. Las otras columnas muestran el número de reglas de reescritura y la mejora en la *eficiencia*[7] (*speedup*) promedio alcanzada para los programas especializados, obtenidos empleando respectivamente las tres reglas de desplegado consideradas: `emb_goal`, `emb_redex` y `comp_redex`. La última fila de la tabla $T_{\rm INDY}$ muestra el *tiempo de especialización*[8] promedio para cada una de las reglas de desplegado consideradas. Los tiempos de ejecución se midieron en una estación de trabajo HP 712/60, bajo

[7]La razón entre el tiempo de ejecución del programa original y el del especializado para el objetivo considerado; ver la definición 3.1.3.

[8]Tiempo empleado, por el evaluador parcial, para obtener los programas especializados.

el sistema operativo HP Unix v10.01. Los tiempos están expresados en milisegundos y son el resultado de calcular la media para 10 ejecuciones. Las medidas de la eficiencia se computaron ejecutando los programas original y especializado con el lenguaje lógico–funcional perezoso $\mathcal{TOY}$ [49] (que es de disposición pública). Los objetivos que se emplearon en las respectivas ejecuciones de los programas y se recogen en el Apéndice B se seleccionaron para que los tiempos de ejecución fuesen razonablemente grandes.

Los datos de la Tabla 8.1 demuestran que los refinamientos introducidos en los diferentes niveles de control del algoritmo de NPE y que se han incorporado al sistema INDY proporcionan un aumento satisfactorio de la eficiencia en todas las pruebas realizadas (lo cual es muy esperanzador, sobre todo teniendo en cuenta el hecho de que no se han proporcionado datos de entrada parciales en estos ejemplos, excepto para `match-app`, `match-kmp` y `sorted_bits`). Este resultado es cierto incluso en los ejemplos más complicados, como es el caso de `maxlength`, un programa de prueba típico que sirve para medir si un evaluador parcial puede producir la transformación de *tupling*, si bien es necesario tratar el operador $\leq$ como un predicado *built-in* cuya posible selección se retrasa hasta que sus argumentos están suficientemente instanciados (al igual que se hace, por ejemplo, en el sistema ECCE para la deducción parcial conjuntiva [109]). También deseamos poner de relieve que, con las mejoras introducidas, el método NPE consigue pasar el test KMP sin necesidad de hacer uso de estrategias un tanto *ad hoc* como la regla de desplegado `ostrans`[9]; basta con usar `emb_redex` o `comp_redex` con normalización. Aunque en el ejemplo de prueba anterior el uso de normalización es determinante, los experimentos realizados no son concluyentes a este respecto, habiendo casos en los que introducir normalización puede ser perjudicial ya que conduce a un desplegado excesivo del árbol de *narrowing* local (e.g. el programa `double_app` con objetivo conjuntivo). Para otros ejemplos el uso de normalización es indiferente a la hora de alcanzar una buena especialización (e.g., el programa `double_app` con objetivo funcional, cuando se emplean las nuevas estrategias de desplegado `emb_redex` o `comp_redex`). Por otra parte, las extensiones realizadas son *conservativas*, en el sentido de que no hay una penalización con respecto a la especialización alcanzada por el sistema original sobre objetivos no conjuntivos escritos en un estilo funcional puro (si bien, algunos tiempos de especialización son ligeramente superiores debido al procesamiento mas complejo que se realiza). Es interesante observar que, a juzgar por los datos de la Tabla 8.1 sobre la eficiencia, puede parecer que no hay una diferencia significativa entre las estrategias `emb_redex` y `comp_redex`. Sin embargo, cuando consideramos los tiempos de especialización (T_{INDY}) y el tamaño de los programas especializados (Rw), encontramos que `comp_redex` tiene mejor comportamiento en general. Así, aunque la eficiencia alcanzada por estas estrategias es similar, `emb_redex` es inherentemente más compleja y, muy a menudo, expande los árboles de *narrowing* más allá de su punto "óptimo".

Hemos comparado nuestros resultados con los de los evaluadores parciales *on-line* más sofisticados, tales como el sistema de deducción parcial conjuntiva

[9]Para una descripción de esta regla de desplegado un tanto *ad hoc* ver el Apéndice A.

Ecce [109, 127, 131], que es la referencia más reciente y apropiada (lamentablemente, no existe una implementación a disposición pública del supercompilador positivo). Nuestros resultados muestran que el método NPE puede conducir a mejoras de rendimiento significativas, con relación al sistema Ecce para algunos programas, particularmente aquéllos que se utilizan para comprobar la capacidad de eliminación de estructuras intermedias, i.e., para ver si se consigue el efecto de la *deforestación* (e.g., `double_app` y `double_flip`). Esto confirma que el método NPE esta mejor dotado que el sitema Ecce para realizar la deforestación y eliminar los recorridos múltiples de una misma estructura. Esto es debido a que esta clase de mejoras se benefición notablemente tanto de la componente lógica como de la funcional de nuestro sistema. El sistema Indy también produce aumentos de eficiencia razonables para los otros programas de prueba, que están próximos a los que alcanza Ecce y algunas veces son mejores (e.g. para el `double_app`, `double_flip` y `kmp`, ver [109]).

8.5 Conclusiones.

En los lenguajes lógico–funcionales, las expresiones pueden escribirse explotando la capacidad de *anidamiento* propia de la sintaxis funcional (e.g., $app(app(x,y),z) \approx r$) pero, en muchos casos, puede ser apropiado (o necesario) descomponer las expresiones anidadas y escribirlas en la sintaxis de la programación lógica (e.g., $app(x,y) \approx w \ \wedge \ app(w,z) \approx r$). Esto es cierto particularmente en el caso en el que hay que realizar comprobaciones sobre los valores de estructuras intermedias, como por ejemplo, cuando es necesario realizar el test $sorted\_bits(w)$ sobre una lista intermedia w. El sistema original Indy, desprovisto de las mejoras introducidas en este capítulo, se comporta bien al especializar objetivos escritos en una sintaxis funcional "pura" [13]. Sin embargo, no es capaz de producir una buena especialización sobre los programas de prueba de la Tabla 8.1 cuando los objetivos se expresan como conjunciones de subobjetivos (y comúnmente se produce una pérdida de eficiencia). Por consiguiente, con el método NPE clásico no pueden obtenerse algunas de la transformaciones estándar pero difíciles de alcanzar, tales como el *tupling* [51]. Contrariamente a lo que sucede en el marco de la deducción parcial clásica, en la que solamente se puede realizar el plegado sobre átomos individuales, el algoritmo de NPE puede realizar el plegado de expresiones complejas (que contienen un número arbitrario de llamadas a función y símbolos primitivos). Sin embargo, esto no es suficiente para alcanzar los efectos del *tupling* en la práctica, ya que las expresiones complejas se generalizan más allá de lo conveniente y se pierde toda la especialización conseguida, reproduciéndose en la mayoría de los casos el programa original (no especializado). Los resultados presentados en este capítulo demuestran que es posible complementar el algoritmo de NPE genérico con opciones de control apropiadas que posibilitan la especialización de expresiones complejas que contienen símbolos de función primitivos, proporcionando un marco más potente de especialización poligenética sin el empleo de técnicas *ad hoc*. También se ha demostrado que la introducción de estas mejoras no afecta a la terminación del método.

Hasta donde alcanza nuestro conocimiento, este es el primer marco práctico para la especialización de lenguajes lógico–funcionales modernos (con semántica perezosa), tales como Babel[157], Curry[10][96] y $\mathcal{TOY}$[49], que emplea planificación dinámica y técnicas de partición. Considerando el grado de especialización alcanzado y el bajo coste de implementación, los resultados obtenidos en los experimentos son muy satisfactorios y esperanzadores (e.g. el mecanismo de abstracción utilizado por INDY es más simple que el de ECCE, dando buenos resultados aún a pesar de estar en un marco más complejo, como es el de la evaluación parcial de programas lógico–funcionales con semántica no estricta). Estos resultados indican que todavía queda mucho margen para el progreso, pudiéndose conseguir mejoras que propicien un mayor aumento del rendimiento dentro de nuestro marco, tales como la introducción de operadores de abstracción más poderosos, basados en métodos de análisis estático, que nos permiten determinar el modo óptimo en el que deben partirse las expresiones (intentando no dañar la comunicación entre las estructuras de datos que comparten variables) y considerando el problema de dar un tratamiento particularizado a cada uno de los símbolos primitivos que intervienen en el lenguaje. También, es posible introducir un control global más refinado, mediante el empleo de árboles globales [145], que propiciaría un incremento de la eficiencia del programa especializado (si bien al precio de un aumento en el tiempo de especialización).

Resumiendo, algunas de las conclusiones y contribuciones originales presentadas en este capítulo son las siguientes:

1. Se ha introducido una regla de desplegado dinámica y un operador de abstracción novedoso dotado de técnicas de partición simples. Ambos operadores pueden considerarse, en cierto sentido, independientes de la estrategia de *narrowing*[11] y aumentan considerablemente la capacidad de especialización del método NPE.

2. Las nuevas opciones introducidas mejoran el algoritmo de NPE, permitiendo la manipulación de expresiones que contienen símbolos de función primitivos sin artificios *ad hoc*, y no afectan a la terminación del proceso de especialización.

3. Las nuevas estrategias de control se han incorporado al sistema INDY[4] y se ha utilizado éste para la especialización de algunos programas de prueba representativos, obteniéndose versiones especializadas más precisas y eficientes que con la implementación previa. También se ha comprobado la calidad de las mejoras introducidas mediante la especialización de algunos ejemplos que la NPE clásica no trataba convenientemente.

4. Nuestro método es aplicable a algunos lenguajes lógico–funcionales modernos más populares, habiéndose obtenido un algoritmo de evaluación

[10]En [5] se aborda la extensión del marco de la NPE para incluir el principio de residuación utilizado en la semántica operacional del lenguaje Curry.

[11]No se debe olvidar que la regla de desplegado dinámica requiere de estrategias fuertemente completas.

parcial que engloba tanto la especialización de los programas lógicos convencionales como de los programas funcionales (de semántica no estricta). De hecho, desde el punto de vista práctico, este trabajo puede verse como un intento de construir las bases para el desarrollo de un evaluador parcial para los lenguajes integrados que sea competitivo en eficiencia con los especializadores funcionales y lógicos tradicionales.

Capítulo 9

Conclusiones.

9.1 Conclusiones.

El objetivo principal de esta tesis ha sido investigar las técnicas de especialización de programas lógico-funcionales perezosos dentro del marco establecido en [14]. En ella se han introducido:

- `Mejoras en el mecanismo de base`.

  Se ha definido un procedimiento de evaluación parcial basado en el *narrowing* perezoso que es aplicable a los lenguajes lógico–funcionales con semántica no estricta más populares (como Babel [157], $\mathcal{TOY}$ [49] o Curry [96]) y para el que se ha demostrado su corrección y completitud. Se ha identificado la clase de los programas uniformes [120, 121] como aquélla sobre la cual es posible refinar la estrategia de *narrowing* perezoso, sin pérdida de completitud. La nueva estrategia se ha denominado *narrowing* perezoso uniforme y se ha demostrado que es computacionalmente equivalente a la estrategia de *narrowing* necesario sobre la mencionada clase de programas. Más aún, el *narrowing* perezoso uniforme posee las mismas buenas propiedades que el *narrowing* necesario cuando se emplea como mecanismo de base con el que parametrizar el algoritmo génerico de NPE. Esto permite obtener un evaluador parcial que se ha probado que es fuertemente correcto y para el que se consiguen mejores prestaciones que con la estrategia de *narrowing* perezoso original.

- `Técnicas avanzadas de especialización`.

  Se ha mejorado tanto el procedimiento de control local como el de control global del algoritmo genérico de NPE introduciendo, respectivamente: i) una nueva regla de desplegado dinámica y ii) un operador de abstracción que emplea técnicas de partición. Estas innovaciones han permitido la especialización de programas con respecto a expresiones complejas y alcanzar una especialización poligenética sin el empleo de artificios *ad hoc*, todo ello sin afectar a la terminación del proceso de evaluación parcial.

También se ha estudiado la efectividad de las técnicas desarrolladas para el caso concreto de (fragmentos de) lenguajes lógico-funcionales modernos, como Curry o $\mathcal{TOY}$. Las nuevas estrategias de control se han incorporado al sistema Indy [4], obteniendose mejoras en las versiones especializadas de la mayoría de los programas de prueba considerados, que resultan ser más precisas y eficientes que las obtenidas en la implementación previa.

El método de evaluación parcial obtenido es aplicable a los lenguajes lógico–funcionales modernos (específicamente, aquéllos con semántica operacional no estricta) y engloba tanto la especialización de los programas lógicos convencionales como de los programas funcionales (de semántica no estricta). De hecho, desde el punto de vista práctico, este trabajo puede verse como un punto de partida para el desarrollo de un evaluador parcial para los lenguajes integrados que sea competitivo en eficiencia con los especializadores funcionales y lógicos tradicionales. También desde el punto de vista práctico, la motivación última de esta tesis ha sido dotar a los lenguajes integrados con una herramienta automática para el desarrollo de programas eficientes y seguros, que les permita competir con el resto de los lenguajes de programación y haga realidad las promesas que encierra la programación declarativa.

9.2 Trabajo Futuro.

Finalizamos esta tesis resumiendo algunas líneas de trabajo abiertas y que no se han podido completar antes de la conclusión de esta memoria por limitaciones temporales.

- Estudiar la viabilidad práctica de extender los beneficios del *narrowing* perezoso uniforme a clases de programas más amplias.

  Los sistemas de reescritura *forward branching* son una subclase de los sistemas de reescritura ortogonales y fuertemente secuenciales [192]. En [180], Salinier y Strandh han presentado un algoritmo de transformación que convierte cualquier sistema de reescritura *forward branching* en uno ortogonal basado en constructores y fuertemente secuencial, i.e., un sistema de reescritura inductivamente secuencial [97]. El algoritmo de transformación de Salinier y Strandh es correcto y completo, probándose la equivalencia semántica entre el sistema de reescritura original y el transformado. Estos resultados permiten la utilización inmediata del *narrowing* perezoso uniforme con aquéllos programas que pudiesen identificarse con los sistemas de reescritura *forward branching* [153].

- Introducir nuevas mejoras en el marco genérico de NPE mediante el uso de técnicas de análisis basadas en interpretación abstracta [54].

  El Algoritmo 1 debe considerarse como el esquema de un evaluador parcial dirigido por *narrowing*. En la segunda parte de esta tesis, simplemente se ha iniciado el camino para mejorar los mecanismos de control pero, como se ha argumentado, todavía hay mucho margen para incrementar esas mejoras. Por ejemplo, el operador de abstracción puede refinarse mediante el empleo de métodos de análisis estático, que nos permitan determinar el modo óptimo en el que deben partirse las expresiones (intentando no dañar la comunicación entre las estructuras de datos que comparten variables) y considerando el problema de dar un tratamiento particularizado a cada uno de los símbolos primitivos que intervienen en el lenguaje. Por otra parte, los programas especializados pueden optimizarse eliminando las reglas y los argumentos redundantes que pueden aparecer durante el proceso de evaluación parcial. Por lo tanto, es preciso desarrollar técnicas de análisis que detecten la aparición de dichas reglas y argumentos redundantes. Una posibilidad consistiría en intentar adaptar técnicas estándar como las presentadas en Benkerimi y Hill [34], Benkerimi y Lloyd [36] y Gallagher et al. [75, 76]. Como se ha visto, dado que en el marco de la NPE se permiten términos especializados que contienen símbolos de función anidados, las lhs's de las reglas del programa residual también pueden contener símbolos de función anidados, lo cual hace que se incumplan las restricciones sintácticas necesarias para la completitud de algunas de las estrategias de *narrowing* (en particular la estrategia de *narrowing* perezoso). Para evitar este inconveniente, en el Apartado 4.2 se ha introducido una fase de renombramiento del programa residual que restaura la disciplina de constructores del programa original. Sin embargo, el postproceso de renombramiento adolece de una falta de determinismo que es necesario remediar. La búsqueda de nuevas heurísticas, que mejoren la introducida en el Apartado 4.2.1, todavía es un problema de investigación abierto.

- Evaluación parcial con restricciones de desigualdad.

  La propagación de información negativa puede mejorar la especialización de los programas al restringir la clase de valores que pueden tomar las variables. Por ejemplo, la propagación de información negativa permite remediar un problema observado en la especialización de los programas de búsqueda de patrones que suelen dar lugar a programas especializados que realizan ciertas comprobaciones innecesarias cuando se especializan utilizando una técnica de propagación basada simplemente en unificación. Entre las técnicas de evaluación parcial que permiten la propagación de ambos tipos de información y consiguen este tipo de optimizaciones se encuentra el *supercompilador* de Turchin [197, 199]. El supercompilador pasa la información mediante "*entornos*"[1], de forma que no solamente propaga la información positiva, por el efecto de aplicar unificadores, sino que también propaga información negativa que restringe los valores que pueden tomar las variables [198, 189]. Una de las carencias del método NPE [12, 8] es la posibilidad de propagar información negativa. Creemos

[1]Si bien en la literatura relacionada con la evaluación parcial un *entorno* es una aplicación de las variables a los valores, y puede considerarse como una substitución, aquí se entiende por *entorno* una estructura que contiene un conjunto de restricciones sobre las variables del programa.

que esto puede conseguirse empleando algún tipo de *narrowing* con (restricciones de) desigualdad. Por lo tanto, una de nuestras prioridades en el futuro será estudiar el comportamiento de la extensión del método NPE (corrección, eficiencia, grado de especialización del programa resultante, etc.) cuando se emplean lenguajes que puedan incluir desigualdades en el cuerpo de las reglas así como en los objetivos [27, 37, 69, 70, 172].

Apéndice A

El sistema INDY.

El sistema INDY (Integrated Narrowing-Driven specialization sYstem [4, 13]) es un evaluador parcial para lenguajes lógico–funcionales que incorpora las técnicas desarrollas en [3, 8, 14] y [17]. Una de sus características distintivas es el empleo del mecanismo de propagación de la información basado en unificación propio del *narrowing*. El sistema INDY está implementado en SICStus Prolog v3.6 y está puesto a disposición pública a través de internet [4]. La implementación, en su estado actual, consta de alrededor de 800 cláusulas (unas 4000 líneas de código). El evaluador parcial se compone de 95 cláusulas y el metaintérprete de 415 cláusulas (incluyendo el código necesario para manejar la *ground representation*). En la implementación del analizador sintáctico y otras utilidades se han empleado 90 cláusulas y en el postproceso de renombramiento 45. La implementación sólo considera programas incondicionales. Los programas condicionales se tratan, tal y como se hace en esta memoria, mediante el empleo de las funciones predefinidas `and`, `if_then_else`, y también `case_of`, que se reducen empleando reglas de definición estándar bien establecidas en la literatura (ver por ejemplo, [157]). El interfaz del evaluador parcial permite al usuario seleccionar una estrategia de evaluación perezosa (*call-by-name*) o impaciente (*call-by-value*), así como la posibilidad de normalizar los objetivos entre los pasos de *narrowing* (usando un subconjunto terminante de las reglas del programa). El usuario también puede elegir otras opciones, que se comentan más adelante. El sistema realiza automáticamente un postproceso de renombramiento que es útil para alcanzar la condición de independencia y asegura que los programas especializados son CB.

En este apéndice estamos interesados, principalmente, en describir la variedad de opciones que el usuario puede seleccionar y que determinan el comportamiento del sistema. También estamos interesados en poner de relieve el grado en el que han influido las aportaciones de esta tesis en la implementación de esas facilidades. Para más información sobre el uso del sistema, ver [4].

Estrategias de Narrowing.

El algoritmo de NPE utiliza el *narrowing* para construir los árboles de búsqueda locales. La estrategia de *narrowing* concreta puede seleccionarse utilizando el comando `nwing`. En la actualidad, están disponibles las siguientes opciones:

- `innermost`: Implementa la estrategia de *narrowing* impaciente (*innermost*) de [72], que se caracteriza porque un paso de *narrowing* se da sobre el redex *innermost* más a la izquierda del término considerado. Se requiere el empleo de programas CB y completamente definidos para preservar la completitud de esta estrategia.

- `lazy`: Implementa la estrategia de *narrowing* perezoso definida en el Apartado 2.9.3.

- `ulazy`: Implementa la estrategia de *narrowing* perezoso uniforme definida en el Apartado 6.3. Esta opción no está activa en la versión 1.8 del sistema INDY descrita en [4], que es la versión que se encuentra a disposición pública a través de internet, y se ha incorporado para realizar los experimentos del Apartado 7.3.

- `needed`: Implementa la estrategia de *narrowing* necesario definida en el Apartado 2.9.4.

Tanto las estrategias de *narrowing* impaciente como las perezosas pueden combinarse con el empleo de pasos de normalización[1]. Esta opción puede seleccionarse utilizando el comando `normal`.

Control Local.

La regla de desplegado determina cómo se construyen los árboles de *narrowing* local y cuándo se detiene su construcción. La regla de desplegado puede seleccionarse utilizando el comando `unfold`. En la actualidad, están disponibles las siguientes opciones:

- `emb_goal`: Expande las derivaciones mientras los nuevos objetivos no subsuman un objetivo comparable que haya aparecido previamente en la misma rama del árbol de desplegado;

- `emb_redex`: La regla de desplegado concreta definida en la Sección 8.2.1 que implementa el test *dependency_clash* comprobando la subsumción homeomórfica entre ancestros comparables de los redexes seleccionados para asegurar la finitud del proceso de desplegado (se debe notar que esta opción difiere de la opción `emb_goal` en que `emb_redex` implementa una planificación dinámica, frente a la regla de selección estática implementada

[1] Nótese que en el caso de las estrategias de *narrowing* perezosas no se requiere que los programas sean terminantes. Por lo tanto, si se quiere garantizar la finitud del proceso de evaluación parcial cuando se emplea la normalización, es preciso seleccionar un subconjunto terminante de las reglas del programa.

por `emb_goal`, y que el test de subsumción homeomórfica se realiza sobre simples redexes en vez de sobre objetivos completos);

- `comp_redex`: La regla de desplegado concreta definida en la Sección 8.2.1, empleando la forma más simple del test de *dependency_clash* en la que la existencia de ancestros comparables de los redexes seleccionados detiene el proceso de desplegado.

- `depthk`[2]: Una estrategia que despliega los árboles de *narrowing* local hasta alcanzar un nivel de profundidad k, i.e., las derivaciones que constituyen las ramas de los árboles de *narrowing* local tienen una longitud menor o igual que k.

En versiones anteriores del sistema INDY también era posible seleccionar una regla de desplegado de naturaleza un tanto *ad hoc* que se describe a continuación:

- `ostrans`: Una regla de desplegado que simplemente construye los árboles de *narrowing* local desplegandolos un sólo nivel pero que permite pasos de desplegado adicionales para las expresiones cuyo símbolo en cabeza es un símbolo de función primitivo (e.g. `and`, `eq`, o `if_then_else`). Esta estrategia no garantiza la terminación.

Esta regla está inspirada en las reducciones transitorias (*transient reductions*) de [188]. La regla `ostrans` se menciona en varios puntos de la presente memoria de tesis y posibilita pasar el test de KMP cuando se emplean reglas de desplegado que no hacen uso de las dependencias funcionales de los redexes.

Control Global.

El sistema INDY implementa el operador de abstracción definido en el apartado 8.2.2, con y sin la posibilidad de partir las expresiones complejas que contienen símbolos de función primitivos. Cuando se selecciona la opción de partir las expresiones complejas, el operador de abstracción analiza dichas expresiones juzgando si es conveniente partirlas para evitar generalizaciones innecesarias.

La estrategia de partición no es una opción fijada por defecto y, si se desea, es necesario activarla mediante el comando `split`.

[2]Esta opción no puede seleccionarse si previamente se ha elegido la estrategia de *narrowing* necesario.

Apéndice B

Programas de Prueba y Llamadas Especializadas

En este apéndice se incluye el código de los programas utilizados en los experimentos presentados a lo largo de este trabajo, así como como las llamadas evaluadas parcialmente. Para una especificación formal de la sintaxis de los programas, ver [4].

Programas de Prueba y Llamadas Especializadas en los Experimentos del Apartado 7.3.

1. Programa `ackermann`:

   $$\begin{array}{l} ackermann(N) \ \rightarrow \ ack(s(s(0)), N) \\ ack(0, N) \ \rightarrow \ s(N) \\ ack(s(N), M) \ \rightarrow \ ack0(s(N), M) \\ ack0(s(M), 0) \ \rightarrow \ ack(M, s(0)) \\ ack0(s(M), s(N)) \ \rightarrow \ ack(M, ack(s(M), N)) \end{array}$$

   Llamada especializada: $ackermann(N)$

2. Programa `allones`:

   $$\begin{array}{l} f(L) \ \rightarrow \ allones(length(L)) \\ allones(0) \ \rightarrow \ Y \\ append([X|R], Y) \ \rightarrow \ [\,] \\ allones(s(N)) \ \rightarrow \ [1, allones(N)] \\ length([\,]) \ \rightarrow \ 0 \\ length([H|T]) \ \rightarrow \ (s(0) + length(T)) \\ 0 + X \ \rightarrow \ X \\ s(X) + Y \ \rightarrow \ s(X + Y) \end{array}$$

   Llamada especializada: $f(L)$

3. Programa `applast`:

$$
\begin{array}{l}
applast(L,X) \;\to\; last(append(L,[X])) \\
last([X|Y]) \;\to\; last0(X,Y) \\
last([X|Y]) \;\to\; last(Y) \\
last0(X,[\,]) \;\to\; X \\
append([\,],Y) \;\to\; Y \\
append([X|R],Y) \;\to\; [X|append(R,Y)]
\end{array}
$$

Llamada especializada: $applast(L,X)$

4. Programa `exam`:

$$
\begin{array}{l}
f(0,Y) \;\to\; f0(0,Y) \\
f(s(X),Y) \to s(f(X,Y)) \\
f0(0,0) \to s(f(0,0)) \\
g(0) \to g(0) \\
h(s(X)) \to 0
\end{array}
$$

Llamada especializada: $h(f(X,g(Y)))$

5. Programa `fibonacci`:

$$
\begin{array}{l}
fib(0) \;\to\; s(0) \\
fib(s(X)) \;\to\; fib0(X) \\
fib0(0) \;\to\; s(0) \\
fib0(s(N)) \;\to\; (fib(s(N)) + fib(N)) \\
0 + X \;\to\; X \\
s(X) + Y \;\to\; s(X+Y)
\end{array}
$$

Llamada especializada: $fib(N)$

6. Programa `match-kmp`:

$$
\begin{array}{l}
match(P,S) \ \rightarrow \ loop(P,S,P,S) \\
loop([\,],SS,OP,OS) \ \rightarrow \ true \\
loop([P|PP],S,OP,OS) \ \rightarrow \ loop0([P|PP],S,OP,OS) \\
loop0([P|PP],[\,],OP,OS) \ \rightarrow \ false \\
loop0([P|PP],[S|SS],OP,OS) \ \rightarrow \ \ \texttt{if}\ P \approx S \ \texttt{then}\ loop(PP,SS,OP,OS) \\
\ \texttt{else}\ next(OP,OS) \\
next(OP,[\,]) \ \rightarrow \ false \\
next(OP,[S|SS]) \ \rightarrow \ loop(OP,SS,OP,SS) \\
\texttt{if}\ true\ \texttt{then}\ A\ \texttt{else}\ B \ \rightarrow \ A \\
\texttt{if}\ false\ \texttt{then}\ A\ \texttt{else}\ B \ \rightarrow \ B \\
a \approx a \ \rightarrow \ true \\
b \approx b \ \rightarrow \ true \\
a \approx b \ \rightarrow \ false \\
b \approx a \ \rightarrow \ false
\end{array}
$$

Llamada especializada: $match([a,a,b],S)$

7. Programa `matmult`:

$$
\begin{array}{l}
matmult([X|Xs],Y) \ \rightarrow \ [rowmult(X,Y)|matmult(Xs,Y)] \\
matmult([\,],Y) \ \rightarrow \ [\,] \\
rowmult(X,[Y|Ys]) \ \rightarrow \ [dotmult(X,Y)|rowmult(X,Ys)] \\
rowmult(X,[\,]) \ \rightarrow \ [\,] \\
dotmult([X|Xs],[Y|Ys]) \ \rightarrow \ (mult(X,Y) + dotmult(Xs,Ys)) \\
dotmult([\,],[\,]) \ \rightarrow \ 0 \\
0 + X \ \rightarrow \ X \\
s(X) + Y \ \rightarrow \ s(X+Y) \\
0 \times X \ \rightarrow \ 0 \\
s(X) \times Y \ \rightarrow \ (X \times Y) + Y
\end{array}
$$

Llamada especializada: $matmult([X,Y,Z],W)$

8. Programa `palindrome`:

$$
\begin{array}{l}
palindrome(L) \ \rightarrow \ reverse(L) \approx_l L \\
reverse(L) \ \rightarrow \ rev(L,[\,]) \\
rev([\,],L) \ \rightarrow \ L \\
rev([X|L],Y) \ \rightarrow \ rev(L,[X|Y]) \\
[\,] \approx_l [\,] \ \rightarrow \ true \\
[X|RX] \approx_l [Y|RY] \ \rightarrow \ X \approx_e Y \ \wedge \ RX \approx_l RY \\
a \approx_e a \ \rightarrow \ true \\
b \approx_e b \ \rightarrow \ true \\
c \approx_e c \ \rightarrow \ true \\
true \ \wedge \ X \ \rightarrow \ X
\end{array}
$$

Llamada especializada: $palindrome([s(0)|L])$

9. Programa `sumleq`:

$$
\begin{array}{l}
sum(0,X) \;\to\; X \\
sum(s(X),0) \;\to\; s(X) \\
sum(s(X),s(Y)) \;\to\; s(s(sum(X,Y))) \\
leq(0,X) \;\to\; true \\
leq(s(N),M) \;\to\; leq0(s(N),M) \\
leq0(s(N),0) \;\to\; false \\
leq0(s(N),s(M)) \;\to\; leq(N,M)
\end{array}
$$

Llamada especializada: $leq(X, sum(X,Y))$

10. Programa `sumprod`:

$$
\begin{array}{l}
sumprod(L) \;\to\; sum(sumlist(L), prodlist(L)) \\
sumlist([\,]) \;\to\; 0 \\
sumlist([H|T]) \;\to\; sum(H, sumlist(T)) \\
prodlist([H|T]) \;\to\; prod(H, prodlist(T)) \\
sum(0,Y) \;\to\; Y \\
sum(s(X),Y) \;\to\; s(sum(X,Y)) \\
prod(0,Y) \;\to\; 0 \\
prod(s(X),Y) \;\to\; sum(prod(X,Y),Y)
\end{array}
$$

Llamada especializada: $sumprod(L)$

Programas de Prueba y Llamadas Especializadas en los Experimentos del Apartado 8.4.

1. Programa `applast`:

$$
\begin{array}{l}
applast(L,T,X) \;\to\; append(L,[X]) \approx_l T \,\wedge\, last(T) \approx_e X \\
last([X]) \;\to\; X \\
last([X|Y]) \;\to\; last(Y) \\
append([\,],Y) \;\to\; Y \\
append([X|R],Y) \;\to\; [X|append(R,Y)] \\
[\,] \approx_l [\,] \;\to\; true \\
[X|RX] \approx_l [Y|RY] \;\to\; X \approx_e Y \,\wedge\, RX \approx_l RY \\
a \approx_e a \;\to\; true \\
b \approx_e b \;\to\; true \\
true \wedge X \;\to\; X
\end{array}
$$

Llamada especializada: $append(L,[X]) \approx_l T \,\wedge\, last(T) \approx_e X$

2. Programa `double_app`:

$$
\begin{array}{l}
dapp(X,Y,W,Z,R) \;\rightarrow\; append(X,Y) \approx_l W \;\wedge\; append(W,Z) \approx_l R \\
append([\,],Y) \;\rightarrow\; Y \\
append([X|R],Y) \;\rightarrow\; [X|append(R,Y)] \\
[\,] \approx_l [\,] \;\rightarrow\; true \\
[X|RX] \approx_l [Y|RY] \;\rightarrow\; X \approx_e Y \;\wedge\; RX \approx_l RY \\
a \approx_e a \;\rightarrow\; true \\
b \approx_e b \;\rightarrow\; true \\
true \;\wedge\; X \;\rightarrow\; X
\end{array}
$$

Llamada especializada: $append(X,Y) \approx_l W \;\wedge\; append(W,Z) \approx_l R$

3. Programa `double_flip`:

$$
\begin{array}{l}
doubleflip(T,U,R) \;\rightarrow\; flip(T) \approx_t U \;\wedge\; flip(U) \approx_t R \\
flip(leaf(N)) \rightarrow leaf(N) \\
flip(tree(L,N,R)) \rightarrow tree(flip(R),N,flip(L)) \\
leaf(N) \approx_t leaf(M) \rightarrow N \approx_e M \\
tree(A,leaf(B),C) \approx_t tree(D,leaf(E),F)) \rightarrow A \approx_t D \;\wedge\; B \approx_e E \;\wedge\; C \approx_e F \\
a \approx_e a \rightarrow true \\
b \approx_e b \rightarrow true \\
true \;\wedge\; X \rightarrow X
\end{array}
$$

Llamada especializada: $flip(T) \approx_t U \;\wedge\; flip(U) \approx_t R$

4. Programa `fibonacci`:

$$
\begin{array}{l}
fib(0) \;\rightarrow\; s(0) \\
fib(s(0)) \;\rightarrow\; s(0) \\
fib(s(s(N))) \;\rightarrow\; fib(s(N)) + fib(N) \\
0 + Y \;\rightarrow\; Y \\
s(X) + Y \;\rightarrow\; s(X+Y)
\end{array}
$$

Llamada especializada: $fib(N)$

5. Programa `heads&legs`:

$headslegs(M, C, H, L) \rightarrow M + C \approx H \wedge double(M) + double(double(C)) \approx L$
$0 + X \rightarrow X$
$s(X) + Y \rightarrow s(X + Y)$
$double(0) \rightarrow 0$
$double(s(X)) \rightarrow s(s(double(X)))$
$0 \approx 0 \rightarrow true$
$s(X) \approx s(Y) \rightarrow X \approx Y$
$true \wedge X \rightarrow X$

Llamada especializada: $M + C \approx H \wedge double(M) + double(double(C)) \approx L$

6. Programa `match-app`:

$match(P, S) \rightarrow append(A, B) \approx_l S \wedge append(C, P) \approx_l A$
$append([\,], Y) \rightarrow Y$
$append([X|R], Y) \rightarrow [X|append(R, Y)]$
$[\,] \approx_l [\,] \rightarrow true$
$[X|RX] \approx_l [Y|RY] \rightarrow X \approx_e Y \wedge RX \approx_l RY$
$a \approx_e a \rightarrow true$
$b \approx_e b \rightarrow true$
$true \wedge X \rightarrow X$

Llamada especializada: $append(A, B) \approx_l S \wedge append(C, [a, a, b]) \approx_l A$

7. Programa `match-kmp`:

$match(P, S) \rightarrow loop(P, S, P, S)$
$loop([\,], SS, OP, OS) \rightarrow true$
$loop([P|PP], [\,], OP, OS) \rightarrow false$
$loop([P|PP], [S|SS], OP, OS) \rightarrow$ `if` $P \approx S$ `then` $loop(PP, SS, OP, OS)$
`else` $next(OP, OS)$
$next(OP, [\,]) \rightarrow false$
$next(OP, [S|SS]) \rightarrow loop(OP, SS, OP, SS)$
`if` $true$ `then` A `else` $B \rightarrow A$
`if` $false$ `then` A `else` $B \rightarrow B$
$a \approx a \rightarrow true$
$b \approx b \rightarrow true$
$a \approx b \rightarrow false$
$b \approx a \rightarrow false$

Llamada especializada: $match([a, a, b], S)$

8. Programa `maxlength`:

$$\begin{array}{l}
maxlen(X,M,L) \ \rightarrow \ max(X,0) \approx M \ \wedge \ length(X) \approx L \\
length([\,]) \ \rightarrow \ 0 \\
length([X|R]) \ \rightarrow \ s(length(R)) \\
max([\,],M) \ \rightarrow \ M \\
max([X|R],N) \ \rightarrow \ \texttt{if}\ X \leq N \ \texttt{then}\ max(R,N)\ \texttt{else}\ max(R,X) \\
\texttt{if}\ true\ \texttt{then}\ A\ \texttt{else}\ B \ \rightarrow \ A \\
\texttt{if}\ false\ \texttt{then}\ A\ \texttt{else}\ B \ \rightarrow \ B \\
0 \leq 0 \ \rightarrow \ true \\
0 \leq s(M) \ \rightarrow \ true \\
s(N) \leq 0 \ \rightarrow \ false \\
s(N) \leq s(M) \ \rightarrow N \leq M \\
0 \approx 0 \ \rightarrow \ true \\
s(X) \approx s(Y) \ \rightarrow \ X \approx Y \\
true \wedge X \ \rightarrow \ X
\end{array}$$

Llamada especializada: $max(X,0) \approx M \ \wedge \ length(X) \approx L$

9. Programa `palindrome`:

$$\begin{array}{l}
palindrome(L) \ \rightarrow \ reverse(L) \approx_l L \\
reverse(L) \ \rightarrow \ rev(L,[\,]) \\
rev([\,],L) \ \rightarrow \ L \\
rev([X|L],Y) \ \rightarrow \ rev(L,[X|Y]) \\
[\,] \approx_l [\,] \ \rightarrow \ true \\
[X|RX] \approx_l [Y|RY] \ \rightarrow \ X \approx_e Y \ \wedge \ RX \approx_l RY \\
a \approx_e a \ \rightarrow \ true \\
b \approx_e b \ \rightarrow \ true \\
c \approx_e c \ \rightarrow \ true \\
true \wedge X \ \rightarrow \ X
\end{array}$$

Llamada especializada: $palindrome(L)$

10. Programa `sorted_bits`:

$$\begin{array}{l}
sortedbits([\,]) \ \rightarrow \ true \\
sortedbits([X]) \ \rightarrow \ true \\
sortedbits([A,B|C]) \ \rightarrow \ sortedbits([B|C]) \ \wedge \ A \leq B \\
0 \leq 0 \ \rightarrow \ true \\
0 \leq 1 \ \rightarrow \ true \\
1 \leq 1 \ \rightarrow \ true \\
1 \leq 0 \ \rightarrow \ false \\
true \wedge X \ \rightarrow \ X
\end{array}$$

Llamada especializada: $sortedbits([1|L])$

Bibliografía

[1] H. Aït-Kaci and R. Nasr. Integrating Logic and Functional Programming. *Lisp and Symbolic Computation*, 2(1):51–89, 1989.

[2] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. Technical Report DSIC-II/2/97, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[3] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In G. Levi, editor, *Proc. of Static Analysis Symposium, SAS'98*, pages 262–277. Springer LNCS 1503, 1998.

[4] E. Albert, M. Alpuente, M. Falaschi, and G. Vidal. INDY User's Manual. Technical Report DSIC-II/12/98, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[5] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning, LPAR'99*, pages 376–395. Springer LNAI 1705, 1999.

[6] M. Alpuente, S. Escobar, and S. Lucas. Incremental Needed Narrowing. In P. Tarau and K. Sagonas, editors, *Proc of the Int'l Workshop on Implementation of Declarative Languages, IDL'99*, 1999.

[7] M. Alpuente, S. Escobar, and S. Lucas. UPV-Curry: an Incremental Curry Interpreter. In J. Pavelka, G. Tel, and M. Bartosek, editors, *Proc. of 26th Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM'99*, volume 1725 of *Lecture Notes in Computer Science*, pages 327–335, Milovy, Czech Republic, November 1999. Springer-Verlag, Berlin.

[8] 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.

[9] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Técnicas de Evaluación Parcial Perezosa en Programas Uniformes. Technical Report DSIC-II/18/99, UPV, 1999. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[10] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A transformation system for lazy functional logic programs. Technical report, DSIC, 1999.

[11] M. Alpuente, M. Falaschi, and G. Vidal. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. *Theoretical Computer Science*, 165(1):97–131, 1996.

[12] 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.

[13] M. Alpuente, M. Falaschi, and G. Vidal. Experiments with the Call-by-Value Partial Evaluator. Technical Report DSIC-II/13/98, UPV, 1998. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[14] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.

[15] M. Alpuente, M. Falaschi, and G. Vidal. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es, 1998.

[16] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. Technical report DSIC-II/7/99, DSIC, 1999. Submitted for publication. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[17] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of inductively sequential functional logic programs. In *Proc. of the International Conference on Functional Programming (ICFP'99)*, pages 273–283. ACM Press, 1999.

[18] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 143–157. Springer-Verlag, Berlin, 1992.

[19] S. Antoy. Needed Narrowing in Prolog. Technical Report Technical Report 96-2, Portland State University, 1996. Full version of extended abstract in [21].

[20] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, volume 1298 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, Berlin, 1997.

[21] S. Antoy. Needed Narrowing in Prolog. In *Proc. of the 8th Int'l Symp. on Prog. Lang., Implementations, Logics, and Programs*, Lecture Notes in Computer Science, pages 473–474. Aachen University, Germany, Sept. 1996.

[22] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. Technical report MPI-I-93-243, Max-Planck-Institut für Informatik, Saarbrücken, 1993.

[23] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.

[24] S. Antoy, M. Hanus, J. Koj, P.Ñiederau, R. Sadre, and F. Steiner. Pacs 1.1 : The Portland Aachen Curry System User Manual. Technical Report Version of December, 9, RWTH Aachen, Germany, 1999. Available from URL: `http://www-i2.informatik.rwth-aachen.de/hanus/curry`.

[25] S. Antoy and A. Middeldorp. A Sequential Reduction Strategy. *Theoretical Computer Science*, 165:75–95, 1996.

[26] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, Mass., 1990.

[27] P. Arenas, A. Gil, and F. López. Combining Lazy Narrowing with Disequality Constraints. In *Proc. of PLILP'94*, pages 385–399. Springer LNCS 844, 1994.

[28] K. Asai, H. Masuhara, and A. Yonezawa. Partial Evaluation of call-by-value λ-calculus with side-effects. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pages 12–21. ACM, New York, 1997.

[29] F. Baader and T.Ñipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[30] L. Bachmair and N. Dershowitz. Equational inference, canonical proofs and proof orderings. *Journal of the ACM*, 41(2):236–276, 1994.

[31] R. Barbutti, M. Bellia, G. Levi, and M. Martelli. LEAF: A Language which Integrates Logic, Equations and Functions. In D. de Groot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations.* Prentice-Hall, 1986.

[32] H. Barendregt. λ-calculus and Functional Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 321–363. Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990.

[33] M. Bellia and G. Levi. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[34] K. Benkerimi and P.M. Hill. Supporting Transformations for the Partial Evaluation of Logic Programs. *Journal of Logic and Computation*, 3(5):469–486, 1993.

[35] K. Benkerimi and J.W. Lloyd. A Procedure for the Partial Evaluation of Logic Programs. Technical Report TR-89-04, Department of Computer Science, University of Bristol, Bristol, England, May 1989.

[36] K. Benkerimi and J.W. Lloyd. A Partial Evaluation Procedure for Logic Programs. In S. Debray and M. Hermenegildo, editors, *Proc. of the 1990 North American Conf. on Logic Programming*, pages 343–358. The MIT Press, Cambridge, MA, 1990.

[37] D. Bert and R. Echahed. On the Operational Semantics of the Algebraic and Logic Programming Language LPG. In *Recent Trends in Data Type Specifications*, pages 132–152. Springer LNCS 906, 1995.

[38] M. Bidoit, H. J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella. Algebraic System Specification and Development. In *Lecture Notes in Computer Science*, volume 501. Springer-Verlag, Berlin, 1991.

[39] R. Bird. *Introducción a la Programación Funcional con Haskell.* Prentice Hall, Madrid, 2000.

[40] G. Birkhoff and S. MacLane. *A Survey of Modern Algebra.* third edition, Macmillan, New York, 1965.

[41] R. Bol. Loop Checking in Partial Deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.

[42] M. P. Bonacina. Partial evaluation by completion. In G. Italiani et al., editor, *Conferenza dell'Associazione italiana per il calcolo automatico*, 1988.

[43] A. Bondorf. A Self-Applicable Partial Evaluator for Term Rewriting Systems. In J. Diaz and F. Orejas, editors, *Proc. of Int'l Conf. on Theory and Practice of Software Development, Barcelona, Spain*, pages 81–95. Springer LNCS 352, 1989.

[44] P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[45] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87, 1993.

[46] A. Bossi, N. Cocco, and S. Dulli. A Method for Specializing Logic Programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

[47] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding. *New Generation Computing*, 11(1):47–79, 1992.

[48] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[49] R. Caballero-Roldán, F.J. López-Fraguas, and J. Sánchez-Hernández. User's manual for Toy. Technical Report SIP-5797, UCM, Madrid (Spain), April 1997.

[50] D. Chan and M. Wallace. A Treatment of Negation during Partial Evaluation. In H. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 299–318. The MIT Press, Cambridge, MA, 1989.

[51] W. Chin. Towards an Automated Tupling Strategy. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 119–132. ACM, New York, 1993.

[52] C. Consel. New Insights into Partial Evaluation: The Schism Experiment. In In H. Ganzinger, editor, *ESOP'88, 2nd European Symposium on Programming*, pages 236–246. LNCS, volume 300, Springer-Verlag, Berlin, 1988.

[53] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

[54] 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.

[55] J. Darlington. Program transformation. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 193–215. Cambridge University Press, 1982.

[56] J. Darlington and H. Pull. A Program Development Methodology Based on a Unified Approach to Execution and Transformation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 117–131. North-Holland, Amsterdam, 1988.

[57] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations and Equations.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[58] N. Dershowitz. Goal Solving as Operational Semantics. In J.W. Lloyd, editor, *Proc. of ILPS'95*, pages 3–17. The MIT Press, Cambridge, MA, 1995.

[59] 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.

[60] N. Dershowitz and J.-P. Jouannaud. Notations for Rewriting. *Bulletin of the European Association of Theoretical Computer Science*, 43:162–172, 1991.

[61] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.

[62] N. Dershowitz and A. Plaisted. Equational Programming. *Machine Intelligence*, 11:21–56, 1988.

[63] N. Dershowitz and U. Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494, 1993.

[64] R. Echahed. On completeness of narrowing strategies. In *Proc. of CAAP'88*, pages 89–101. Springer LNCS 299, 1988.

[65] R. Echahed. Uniform narrowing strategies. In *Proc. of ICALP'92*, pages 259–275. Springer LNCS 632, 1992.

[66] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[67] A.P. Ershov. On the Partial Compitation Principle. *Information Processing Letters*, 6(2):38–41, 1977.

[68] M. Fay. First Order Unification in an Equational Theory. In *Proc of 4th Int'l Conf. on Automated Deduction*, pages 161–167, 1979.

[69] M. Fernández. Narrowing Based Procedures for Equational Disunification. *Applicable Algebra in Engineering, Communication and Computing*, 3:1–26, 1992.

[70] M. Fernández. Narrowing based procedures for equational disunification. Technical Report 764 LRI, Computer Science Department, University of Paris Sud, July 1992.

[71] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, Reading, MA, 1987.

[72] 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.

[73] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[74] Y. Futamura. Program Evaluation and Generalized Partial Computation. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 1–8. ICOT, Tokyo, 1988.

[75] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. ACM, New York, 1993.

[76] J. Gallagher and M. Bruynooghe. Some Low-Level Source Transformations for Logic Programs. In M. Bruynooghe, editor, *Proc. of 2nd Workshop on Meta-Programming in Logic*, pages 229–246. Department of Computer Science, KU Leuven, Belgium, 1990.

[77] J.P. Gallagher. Transforming Logic Programs by Specializing Interpreters. In *Proc. of 7th European Conf. on Artificial Intelligence ECAI'86*, pages 109–122, 1986.

[78] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel LEAF: A Logic plus Functional Language. Technical Report TR-2/89, Dipartimento di Informatica, Università degli studi di Pisa, 1989.

[79] 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.

[80] R. Glück. On the Generation of S→R-Specializers. Technical report, Technical University, Vienna, Austria, 1991.

[81] R. Glück. Towards Multiple Self-Application. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, 26(9), Sep. 1991)*, pages 309–320. ACM, New York, 1991.

[82] R. Glück. Projections for Knowledge Based Systems. In R. Trappl, editor, *Cybernetics and Systems Research'92. Vol. 1*, pages 535–542. World Scientific, Singapore, 1992.

[83] 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.

[84] 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.

[85] R. Glück and A.V. Klimov. Metacomputation as a Tool for Formal Linguistic Modeling. In *Cybernetics and Systems'94*, volume 2, pages 1563–1570, Singapore, 1994. World Scientific.

[86] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *Proc. Int'l Symp. on Programming Language Implementation and Logic Programming, PLILP'94*, pages 165–181. Springer LNCS 844, 1994.

[87] 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.

[88] R. Glück and V.F. Turchin. Application of Metasystem Transition to Function Inversion and Transformation. In *Proc. of the Int'l Symp. on Symbolic and Algebraic Computation, ISSAC'90*, pages 286–287. ACM, New York, 1990.

[89] C.K. Gomard. A Self-Applicable Partial Evaluator for the Lambda Calculus: Correctness and Pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.

[90] C.K. Gomard and N.D. Jones. A Partial Evaluator for the Untyped Lambda-Calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[91] M. Hamada and A. Middeldorp. Strong Completeness of a Lazy Conditional Narrowing Calculus. In *Proc. of the Second Fuji International Workshop on Functional and Logic Programming, Shonan Village*, 1996. To appear.

[92] 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.

[93] M. Hanus. Combining Lazy Narrowing with Simplification. In *Proc. of 6th Int'l Symp. on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

[94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[95] M. Hanus. Lazy narrowing with simplification. *Computer Languages*, 23(2–4):61–85, 1997.

[96] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[97] M. Hanus, S. Lucas, and A. Middeldorp. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8, 1998.

[98] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www-i2.informatik.rwth-aachen.de/hanus/curry`, 1999.

[99] S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNAI 353, 1989.

[100] P. Hudak. Conception, Evolution and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, Sept. 1989.

[101] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27(4):797–821, 1980.

[102] 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.

[103] J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[104] J.A. Jiménez-Martin, J. Mariño-Carballo, and J.J. Moreno-Navarro. Efficient Compilation of Lazy Narrowing into Prolog. In *Proc. of LOPSTR'92*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1992.

[105] N.D. Jones. Automatic Program Specialization: A Re-Examination from Basic Principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, Amsterdam, 1988.

[106] 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.

[107] N.D. Jones. What *not* to Do When Writing an Interpreter for Specialisation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 216–237. Springer LNCS 1110, 1996.

[108] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[109] Jesper Jørgensen, Michael Leuschel, and Bern Martens. Conjunctive Partial Deduction in Practice. In John Gallager, editor, *Proceedings of the Int'l Workshop on Logic Program Synthesis and Transformation, LOPSTR'96*, pages 59–82. Springer LNCS 1207, 1996.

[110] J.-P. Jouannaud. Rewrite proofs and computations. In H. Schwichtenberg, editor, *NATO Series F: Computer and Sciences*, volume 139, pages 173–218. Springer-Verlag, Berlin, 1994.

[111] S.C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, NJ, 1952.

[112] 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.

[113] J.W. Klop and A. Middeldorp. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195, 1991.

[114] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal of Computation*, 6(2):323–350, 1977.

[115] H.J. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proc. of 9th ACM Symp. on Principles of Programming Languages*, pages 255–267, 1982.

[116] J. Komorowski. An Introduction to Partial Deduction. In A. Pettorossi, editor, *Meta-Programming in Logic, Uppsala, Sweden*, pages 49–69. Springer LNCS 649, 1992.

[117] R. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.

[118] R.A. Kowalski. Predicate Logic as a Programming Language. In *Information Processing 74*, pages 569–574. North-Holland, 1974.

[119] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph-based Implementation of a Functional Logic Language. In *Proc. of ESOP'90*, pages 279–290. Springer LNCS 432, 1990.

[120] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Lazy Narrowing in a Graph Machine. In *Proc. of the Int'l Conf. on Algebraic and Logic Programming*, volume 463 of *Lecture Notes in Computer Science*, pages 298–317, 1990.

[121] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a Functional Logic Language with Disequality Constrains. In K. Apt, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming, JICSLP'93*, pages 207–221. The MIT Press, Cambridge, MA, 1993.

[122] L. Lafave and J.P. Gallagher. Constraint-based Partial Evaluation of Rewriting-based Functional Logic Programs. In *Proc. of LOPSTR'97*, pages 168–188. Springer LNCS 1463, 1997.

[123] 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.

[124] J. Lam and A. Kusalik. A Comparative Analysis of Partial Deductors for Pure Prolog. Technical report, Department of Computational Science, University of Saskatchewan, Canada, May 1991. Revised April 1991.

[125] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[126] M. Leuschel. *Advanced Techniques for Logic Program Specialisation*. PhD thesis, Departement Computerwetenschappen, Katholieke Universiteit Leuven, May 1997.

[127] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks. Technical report, Accessible via `http://www.cs.kuleuven.ac.be/~lpai`, 1998.

[128] 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 the Joint International Conference and Symposium on Logic Programming, JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.

[129] M. Leuschel and B. Martens. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. Technical Report CW-220, Department of Computer Science, K.U. Leuven, Belgium, December 1995.

[130] M. Leuschel and B. Martens. Global Control for Partial Deduction through Characteristic Atoms and Global Trees. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 263–283. Springer LNCS 1110, 1996.

[131] M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.

[132] Michael Leuschel. Extending homeomorphic embedding in the context of logic programming. Technical Report CW 252, Departement Computerwetenschappen, K.U. Leuven, Belgium, June (revised August) 1997.

[133] G. Levi and F. Sirovich. Proving Program Properties, Symbolic Evaluation and Logical Procedural Semantics. In *Proc. of MFCS'75*, pages 294–301. Springer LNCS 32, 1975.

[134] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[135] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.

[136] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. Technical Report CS-87-09, Computer Science Department, University of Bristol, 1987. Revised version 1989, in [137].

[137] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[138] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93, Tallinn (Estonia)*, pages 184–200. Springer LNCS 714, 1993.

[139] R. Loogen and S. Winkler. Dynamic Detection of Determinism in Functional and Logic Programming Languages. In *Proc. of the 3rd Int. Symp. on Programming Language Implementation and Logic Programming*, pages 335–346. Springer LNCS 528, 1991.

[140] L.Sterling and R.D.Beer. Metainterpreters for Expert System Construction. *Journal of Logic Programming*, 6((1 & 2)):163–178, 1989.

[141] S. Lucas-Alba. *Reescritura con Restricciones de Reemplazamiento.* PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Octubre 1998.

[142] S. Lucas-Alba. Apuntes de programación funcional. Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 1999.

[143] B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122:97–117, 1994.

[144] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. Technical Report CSTR-94-16, Computer Science Department, University of Bristol, December 1994.

[145] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In L. Sterling, editor, *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.

[146] A. Middeldorp. Call by Need Computations to Root-Stable Form. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 94–105. ACM, New York, 1997.

[147] A. Middeldorp and E. Hamoen. Counterexamples to Completeness Results for Basic Narrowing. Technical report, Centre for Mathematics and Computer Science, Amsterdam, December 1991.

[148] A. Middeldorp and E. Hamoen. Counterexamples to Completeness Results for Basic Narrowing. In H. Kirchner and G. Levi, editors, *Proc. of Third Int'l Conf. on Algebraic and Logic Programming*, pages 244–258. Springer LNCS 632, 1992.

[149] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

[150] A. Middeldorp and S. Okui. A Deterministic Lazy Narrowing Calculus. In *Proc. of the Fuji Int'l Workshop on Functional and Logic Programming*, pages 104–118. World Scientific, 1995.

[151] A. Middeldorp, S. Okui, and T. Ida. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130, 1996.

[152] A. Miniussi and D. J. Sherman. Squeezing Intermediate Construction in Equational Programs. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 284–302. Springer LNCS 1110, February 1996.

[153] A. Miniussi and R. Strandh. An Efficient Algorithm for Recognizing the Forward-Branching Class of Term-Rewriting Systems. In *In M. Falaschi, M. Navarro, and A. Policriti, editors, Proc. of 1997 Joint Conference on Declarative Programming, APPIA-GULP-PRODE'97*, pages 445–456. Università di Udine, 1997.

[154] J.J. Moreno-Navarro. *Babel: diseño, semántica e implementación de un lenguaje que integra la programación funcional y lógica.* PhD thesis, Facultad de Informática, Universidad Complutense de Madrid, 1989.

[155] J.J. Moreno-Navarro. Expressivity of Functional-Logic Languages and their Implementation. In R. Barbuti I. Ramos, M. Alpuente, editor, *Tutorials of the 1994 Joint Conf. on Declarative Programming, GULP-PRODE94*, pages 11–42. Servicio de Publicaciones de la Universidad Politécnica de Valencia, SPUPV-94.2049, 1994.

[156] J.J. Moreno-Navarro, H. Kuchen, J. Mariño-Carballo, S. Winkler, and W. Hans. Efficient Lazy Narrowing using Demandedness Analysis. In

J. Penjam and M. Bruynooghe, editors, *Proc. of PLILP'93, Tallinn (Estonia)*, pages 167–183. Springer LNCS 714, 1993.

[157] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[158] M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Ann. Math.*, 43:223–243, 1942.

[159] M. O'Donnell. *Computing in Systems Described by Equations.* Springer LNCS 58, 1977.

[160] M. J. O'Donnell. *Equational Logic as a Programming Language.* The MIT Press, Cambridge, MA, 1985.

[161] M. J. O'Donnell. Equational Logic Programming. In D. Gabbay, editor, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5 on Logic Programming, Chapter 2. Oxford Science Publications, 1994.

[162] S. Owen. Issues in the Partial Evaluation of Meta-Interpreters. In H. Abramson and M.H. Rogers, editors, *Proc. of Meta-Programming in Logic Programming*, pages 319–340. The MIT Press, Cambridge, MA, 1989.

[163] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1988.

[164] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[165] A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, pages 355–385. Springer LNCS 1110, 1996.

[166] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414, 1996.

[167] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, Englewood Cliffs, N.J., 1987.

[168] D.A. Plaisted. Equational Reasoning and Term Rewriting Systems. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume I, Logical Foundations, pages 274–364. Oxford University Press, Oxford, 1996.

[169] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, MA,, 1993.

[170] T.W. Pratt and M.V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice Hall, Englewood Cliffs, N.J., 1996.

[171] M. Proietti and A. Pettorossi. The Loop Absorption and the Generalization Strategies for the Development of Logic Programs and Partial Deduction. *Journal of Logic Programming*, 16(1&2):123–161, 1993.

[172] M.J. Ramírez and M. Falaschi. Conditional Narrowing with Constructive Negation. In E. Lamma and P. Mello, editors, *Proc. of Third Int'l Workshop on Extensions of Logic Programing ELP'92*, pages 59–79. Springer LNAI 660, 1993.

[173] C. Reade. *Elements of Functional Programming*. Addison-Wesley, Reading, MA,, 1993.

[174] 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.

[175] P. Réty. Improving basic narrowing techniques. In *Proc. of the Conf. on Rewriting Techniques and Applications*, pages 228–241. Springer LNCS 256, 1987.

[176] J.A. Robinson and E.E. Sibert. LOGLISP: Motivation, Design and Implementation. In K.L. Clark, editor, *Logic Programming*, pages 299–314. Academic Press, 1982.

[177] A. Romanenko. Inversion and metacomputation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 12–22. Sigplan Notices, 26(9), ACM, New York, September 1991.

[178] B.K. Rosen. Tree manipulating systems and church-rosser theorems. *Journal of the ACM*, 20(1):160–187, January 1973.

[179] K.R. Rosen. *Discrete Mathematics and its Applications*. MacGraw-Hill, 1991.

[180] B. Salinier and R. Strandh. Efficient Simulation of Forward-Branching Systems with Constructor Systems. *Journal of Symbolic Computation*, 15:1–19, 1996.

[181] D. Sands. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234, March 1996.

[182] D. Scott. *Outline of a Mathematical Theory of Computation*. Technical Monograph PRG-2. Oxford University Computing Laboratory, November 1970.

[183] H. Seki. Unfold/fold Transformation of General Logic Programs for the Well-Founded Semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.

[184] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[185] R. Socher-Ambrosius and P. Johann. *Deduction Systems*. Springer-Verlag,, 1997.

[186] M.H. Sørensen. Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation. Technical Report 94/7, Master's Thesis, DIKU, University of Copenhagen, Denmark, 1994.

[187] M.H. Sørensen. Convergence of program transformers in the metric space of trees with a reflection on the set of reals. Technical Report DIKU, University of Copenhagen, Denmark., 1997.

[188] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In J.W. Lloyd, editor, *Proc. of ILPS'95*, pages 465–479. The MIT Press, Cambridge, MA, 1995.

[189] M.H. Sørensen, R. Glück, and N.D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In D. Sannella, editor, *Proc. of the 5th European Symp. on Programming, ESOP'94*, pages 485–500. Springer LNCS 788, 1994.

[190] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6 (6):811–838, November1996.

[191] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, MA, 1977.

[192] R. Strandh. Classes of Equational Programs that Compile into Efficient Machine code. In *In N. Dershowitz, editor, Proc. of 3rd International Conference on Rewriting Techniques and Applications, RTA'89*, pages 355:449–461. LNCS, Springer-Verlag, Berlin, 1989.

[193] P.A. Subrahmanyam and J.H. You. FUNLOG = Fuctions + Logic: a Cputational Model Integrating Functional and Logic Programming. In *Proc. of First IEEE Int'l Symp. on Logic Programming*, pages 144–153. IEEE, New York, 1984.

[194] A. Takano. Generalized Partial Computation for a Lazy Functional Language. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut (Sigplan Notices, 26(9), Sep. 1991)*, pages 1–11. ACM, New York, 1991.

[195] P. Thiemann. Towards Partial Evaluation of Full Scheme. In *Proceedings of Reflection'96*, pages 105–115, April 1996.

[196] M. Tofte. *Compiler Generators: What They Can Do, What They Might Do, and What They Will Probably Never Do*, volume 19 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.

[197] V.F. Turchin. Semantic definitions in Refal and automatic production of compilers. In N.D. Jones, editor, *Semantics-Directed Compiler Generation*, pages 441–474. Springer LNCS 94, 1980.

[198] V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[199] V.F. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, 1985*, pages 257–281. Springer LNCS 217, 1986.

[200] V.F. Turchin. The Algorithm of Generalization in the Supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, Amsterdam, 1988.

[201] V.F. Turchin. Metacomputation: Metasystem Transitions plus Supercompilation. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 481–509. Springer LNCS 1110, 1996.

[202] G. Vidal. *Semantics-Based Analysis and Transformation of Functional Logic Programs*. PhD thesis, DSIC-UPV, Sept. 1996. In spanish.

[203] P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[204] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 674–788. Elsevier, Amsterdam and The MIT Press, Cambridge, MA, 1990.

[205] F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In P. Van Hentenryck, editor, *Proc. of the 4th Int'l Static Analysis Symposium, SAS'97*, pages 141–159. Springer LNCS 1302, 1997.

Índice de Materias