Reglas y Estrategias de Transformación para Programas Lógico–Funcionales

Ginés Moreno Valverde
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
Germán Vidal Oriola

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

El problema de la integración de la programación lógica y funcional está considerado como uno de los más importantes en el área de investigación sobre programación declarativa. Para que los lenguajes declarativos sean útiles y puedan utilizarse en aplicaciones reales, es necesario que el grado de eficiencia de su ejecución se aproxime al de los lenguajes imperativos, tal y como se ha conseguido con el lenguaje Prolog. Para ello, es imprescindible el desarrollo de herramientas potentes para la *manipulación* de los programas, capaces de optimizar las implementaciones realizadas. En general, es deseable sustituir las aproximaciones *ad-hoc* por tratamientos más sistemáticos para los problemas de transformación de programas, de forma análoga a como se realizan en este área las labores de análisis. Puesto que la semántica de los lenguajes lógico–funcionales ha sido objeto de numerosos estudios y está matemáticamente bien formalizada, surge el interés por el desarrollo de métodos y técnicas formales para la formulación de optimizaciones, basadas en la semántica, que preserven las propiedades computacionales del programa. Esta tesis se centra en el desarrollo de tales técnicas, adoptándose aproximación conocida como "reglas + estrategias" para la optimización de programas en un contexto lógico y funcional unificado.

De entre la gran variedad de técnicas existentes para la transformación de programas, las técnicas basadas en "reglas + estrategias" son de las que mayor interés han despertado en las dos últimas décadas. Su utilidad reside en que, a pesar de que su grado de automatización no siempre es total, con ellas pueden alcanzarse cotas de optimización mucho más elevadas que con otras técnicas de transformación, como por ejemplo los métodos de evaluación parcial. Las reglas básicas de transformación que se usan en estos marcos son el plegado y el desplegado que consisten, básicamente, en la contración y expansión, respectivamente, de expresiones en un programa utilizando definiciones equivalentes existentes en el mismo o en otro programa precedente en la secuencia de transformación. La combinación de estas reglas con otras para la introducción, eliminación, abstracción o reemplazamiento de expresiones, puede producir programas transformados mucho más eficientes que los originales, siempre y cuando el proceso de transformación sea dirigido por estrategias de transformación adecuadas (como son la composición o la formación de tuplas).

En el contexto de la transformación de programas lógico–funcionales, la propagación bidireccional de parámetros realizada a través del mecanismo de unificación del *narrowing*, es capaz de producir optimizaciones muy apreciables. En este trabajo investigamos diferentes formas de plegado/desplegado basadas en distintos refinamientos de *narrowing* que disfrutan, en muchos casos, de propiedades estrictamente mejores que las de sus homólogas en los contextos funcional y lógico puros. Las ventajas se obtienen al explotar la sinergia existente entre la sintaxis funcional (anidamientos funcionales, posibilidad de evaluaciones perezosas, reducción determinista, declaraciones locales, etc.) y la versatilidad del mecanismo de unificación y las variables lógicas, típicos de los lenguajes lógicos, a través del *narrowing*. Esta visión unificada de ejecución y transformación de programas nos permite explotar los resultados conocidos de ambos campos, funcional y lógico, y desarrollar un esquema simple y potente para mejorar el programa original con respecto a su capacidad para computar tanto valores de funciones como respuestas a objetivos. El esquema presentado en este trabajo constituye la primera aproximación correcta y completa a la transformación de programas por plegado y desplegado en un contexto lógico–funcional integrado.

En la tesis presentamos también algunas aplicaciones de las reglas de plegado/desplegado (como la obtención de semánticas por desplegado o su utilidad para realizar otro tipo de transformaciones como es la evaluación parcial) en el contexto de los lenguajes integrados, centrándonos en la formulación e implementación de un marco de transformación basado en "reglas + estrategias" que incluye transformaciones muy potentes –como son, entre otras, la composición de funciones y la formación de tuplas– que no pueden obtenerse por otros métodos.

Agradecimientos

En estos últimos años dedicados a la elaboración de la presente tesis doctoral, he atravesado algunas de las etapas más interesantes y fecundas de mi vida. A pesar de las dificultades y riesgos que siempre se sufren al enfrentarse a un proyecto (por lo demás nunca fácil) de esta envergadura, hoy puedo decir con satisfacción que este tipo de aventura también está plagada de recompensas y momentos gratificantes, y que merece con mucho la pena asumir el reto de entregarse a ella y disfrutarla. Más aún si, como es mi caso, se tiene la suerte de haber encontrado en el camino a tantas personas dispuestas a ayudar y convertir esta experiencia en una empresa llevadera, agradable y rentable tanto a nivel profesional como humano. Hablo de compañeros de investigación y trabajo, colegas, amigos, familia,... y a todos ellos quiero expresar mi más sincero agradecimiento:

- A los compañeros del Departamento de Informática de la Universidad de Castilla-La Mancha, y en especial a Jose Luis Sánchez, mi tutor de tesis, y a Juan Antonio Guerrero, Fernando Cuartero y Valentín Valero, con quienes he compartido docencia en diferentes asignaturas y que, además de haberme animado en todo momento, nunca han dudado en impartir alguna de mis clases para facilitar mi asistencia a jornadas, congresos o reuniones con mis directores de tesis.

- Quiero agradecer la hospitalidad, amistad y colaboración que me han prestado Isidro Ramos, Salvador Lucas, María José Ramírez, Javier Oliver, Elvira Albert, Cesar Ferri, Santiago Escobar y, en general, todos los integrantes del grupo de Programación Lógica e Ingeniería del Software de la Universidad Politécnica de Valencia.

- A Moreno Falaschi, buen amigo y excelente profesional, con el que siempre ha sido un placer trabajar y realizar estancias de investigación en universidades italianas en las que él me ha introducido en equipos de trabajo de tan alta calidad científica como calidez humana.

- A Germán Vidal, que tantos quebraderos de cabeza me ha proporcionado en todos estos años a base de contraejemplos a mis teorías, hasta hacerme comprender que para aprender previamente se ha de errar. Además de sus sabios consejos, enriquecedoras charlas y prolongados debates, le agradezco la gran amistad que hemos podido desarrollar en diversas estancias de investigación en Italia y Japón.

- Por último, y muy especialmente, a María Alpuente, que me acogió sin reparos en su grupo de investigación de programación declarativa cuando yo era un novato en la materia. Sin su ancha experiencia, gran profesionalidad y mayor dedicación al desarrollo de esta tesis, no me hubiese sido posible concluirla. Más aún, le agradezco la amabilidad y accesibilidad que me ha demostrado constantemente, propiciando un cómodo y cordial ambiente de trabajo.

Quiero agradecer también a aquellos organismos e instituciones que, de una forma u otra, han colaborado en el desarrollo de los trabajos contenidos en esta tesis: a la Universidad de castilla-La Mancha, donde llevo a cabo mi labor docente e investigadora; a las Universidades Politécnica de Valencia y de Udine, donde he realizado diversas estancias trabajando con excelentes grupos de investigación; y, finalmente, al Ministerio de Educación y Ciencia, la Comisión Interministerial de Ciencia y Tecnología, y al Consejo Nacional de Investigación Italiano, que han financiado en gran medida los trabajos del autor.

Finalmente, un agradecimiento muy especial para Pascual Julián, con quien he recorrido este camino codo con codo desde el principio, y también para Yolanda, María, Miguel, Pura, Carlos, Flor, Luis, Benito, Alicia... y todos mis buenos amigos por soportar algunos de mis empachos científicos. Su complicidad y confianza demostrada me han servido para reponer fuerzas en los momentos más difíciles.

Esta tesis está dedicada a toda mi familia, especialmente a mis padres, que nunca han querido perderse ninguna fase del proceso de maduración del proyecto y, claro está, son también ahora los que más satisfechos se sienten con la culminación del mismo.

Índice General

Capítulo 1

Introducción

Después de más de dos décadas de investigación en el campo de la Programación Declarativa, han sido muchas las contribuciones aportadas por la comunidad científica del área, que ha dedicado grandes esfuerzos al desarrollo de esta disciplina. Como consecuencia de esto se han producido grandes avances, tanto en los aspectos más teóricos como en los de aplicación más inmediata. Sin embargo, aún queda por recorrer un considerable trecho en lo que podrían considerarse los fundamentos de la programación lógica y funcional, en particular en lo referente a su combinación y a la integración con otros paradigmas. Una ventaja importante de los lenguajes lógico-funcionales es que se eliminan muchas de las impurezas del lenguaje, introduciendo a cambio conceptos puramente declarativos. Este aspecto es especialmente atractivo en tanto que puede redundar en beneficio de la combinación con otros paradigmas diferentes. Por ejemplo, cuanto más declarativo es el lenguaje, más sencilla resulta su paralelización [Lloyd, 1995].

En un lenguaje lógico-funcional integrado se pueden escribir y ejecutar programas funcionales puros tanto como programas similares a los que se escriben en Prolog. De la programación lógica, los lenguajes integrados obtienen la potencia de las variables lógicas, un mecanismo de búsqueda automática y las estructuras de datos parciales. De la programación funcional, se obtiene la expresividad de las funciones y tipos y un mecanismo de evaluación más eficiente, gracias a la reducción determinista de expresiones funcionales . De esta forma, los lenguajes lógico-funcionales aúnan el poder expresivo de los lenguajes lógicos (debido al empleo de variables lógicas) junto a la eficiencia en la reducción del espacio de búsqueda propiciada por el empleo de la técnica determinista de la reescritura [Hanus, 1994b,a]. Esto evita la necesidad de introducir características extralógicas en el lenguaje –como es el corte de Prolog– para reducir el espacio de búsqueda.

Muchas propuestas para la integración usan sistemas de reescritura de términos condicionales (SRTC) como programas y (alguna variante del) estrechamiento (*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 emparejamiento (*pattern matching*) por unificación en el procedimiento de reducción. En un paso de *narrowing condicional* se unifica un subtérmino del objetivo con la parte izquierda de una regla y se reemplaza el subtérmino del objetivo por la parte derecha instanciada de la regla, añadiendo también al objetivo las correspondientes instancias de las condiciones de la regla utilizada. El uso de *narrowing* como mecanismo operacional para realizar las computaciones supone una extensión muy potente de los programas lógicos tradicionales y el modelo resultante tiene buenas oportunidades para explotar el paralelismo implícito en el lenguaje. La técnica de *narrowing*, como medio para la obtención de un conjunto de soluciones a un problema de E-unificación, fue descrita en los trabajos pioneros de Slagle [1974] y Fay [1979].

En general, el procedimiento de *narrowing* tiene un grado de indeterminismo muy alto, 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: *narrowing* "innermost" (más interno o impaciente) [Fribourg, 1985], *narrowing* básico [Hullot, 1980; Middeldorp y Hamoen, 1994], *narrowing* selección [Bosco *et al.*, 1988], *narrowing* perezoso [Hanus, 1994a; Moreno-Navarro y Rodríguez-Artalejo, 1992; Reddy, 1985], *narrowing* necesario [Antoy *et al.*, 1993, 1994] etc. Cada una de estas estrategias sigue manteniendo, bajo determinadas condiciones, la completitud del cálculo. En [Aït-Kaci y Nasr, 1989; Bellia y Levi, 1986; Dershowitz y Plaisted, 1988; Moreno-Navarro, 1989; Reddy, 1985] se puede encontrar una revisión, análisis y clasificación de éstas y otras propuestas para la integración. La colección de DeGroot y Lindstrom [1986] constituye una referencia estándar en el campo. La panorámica más actualizada se presenta en [Hanus, 1994b]. En [Middeldorp y Hamoen, 1994; Hanus, 1994b] pueden encontrarse resúmenes sobre las condiciones que debe cumplir un programa para que una determinada estrategia sea correcta y completa.

A pesar de que, potencialmente, los distintos paradigmas de programación declarativa ofrecen el soporte más adecuado para el desarrollo rápido y a bajo coste de aplicaciones correctas, así como para la generación de herramientas sofisticadas de ayuda al desarrollo de software, estos lenguajes no están difundidos suficientemente ni han conseguido imponerse a nivel industrial. Esta situación se debe, en gran medida, a la ausencia de un lenguaje lógico-funcional integrado *estándar* (aceptado por toda la comunidad), al carácter *experimental* de las implementaciones existentes (inmaduras aún para aplicaciones reales) y a la carencia de herramientas de desarrollo potentes, capaces de optimizar dichas implementaciones y de asistir al programador en la manipulación, transformación y optimización de los programas.

1.1 Transformación de programas

En la actualidad, existe una demanda creciente de software seguro y fiable. Desafortunadamente, el diseño de un programa correcto, es decir, un programa que se ajusta a su especificación, no siempre produce un programa eficiente. La transformación automática de programas no es más que un método para derivar programas correctos y eficientes partiendo de una especificación ingenua y menos eficiente del problema. Esto permite concentrarse primero en los aspectos de corrección, postergando las consideraciones de eficiencia para etapas posteriores. Más formalmente, por transformación de programas entendemos el problema de, dado un programa P, generar un programa P' que resuelve el mismo problema y que es semánticamente equivalente a P, pero que goza de mejor comportamiento respecto a cierto criterio de evaluación [Bossi y Cocco, 1993].

En la literatura se ha propuesto una gran variedad de transformaciones para mejorar el código sin perder la corrección de las computaciones con respecto a distintos observables. En el contexto de la transformación de programas, se han abordado problemas tales como la *síntesis* y la *especialización* de programas. Se han desarrollado algunas herramientas semiautomáticas para facilitar el proceso de transformación. Una buena revisión de los principales sistemas y técnicas de síntesis y especialización de programas se encuentra en [Dershowitz y Reddy, 1993; Jones *et al.*, 1993].

Las ideas básicas de una metodología sobre transformación de programas se presentaron inicialmente a principios de los 70 para validar ciertas técnicas de manipulación de código, como por ejemplo la eliminación de la recursión a cambio de la iteración [Darlington, 1972; Walker y Strong, 1972]. Sin embargo, su formalización se llevó a cabo algunos años después por Burstall y Darlington [1977]. Su uso extensivo posterior fue propiciado por el desarrollo de los lenguajes lógicos y los funcionales, ya que en estos lenguajes se pueden efectuar manipulaciones usando herramientas sencillas como son el razonamiento ecuacional y la deducción lógica [Clark y Sickel, 1977; Hogger, 1981]. Los esquemas de transformación más difundidos en este contexto hacen uso de dos reglas de transformación elementales: *plegado* y *desplegado* [Burstall y Darlington, 1977]. De alguna forma, estas operaciones pueden considerarse antagónicas: el desplegado de una regla (cláusula) en un programa funcional (lógico) consiste en el reemplazamiento de una llamada a función (predicado) por su respectiva definición, aplicando la correspondiente sustitución, mientras que el plegado es la transformación inversa, es decir, el reemplazamiento de cierto fragmento de código por la llamada a función (predicado) apropiada. Es decir, intuitivamente, la operación de plegado realiza la transformación opuesta a la operación de desplegado, en el sentido de que un paso de desplegado seguido de otro de plegado (y viceversa) puede devolver el programa original. Esta operación se usa generalmente en sistemas de transformación con el objetivo de plegar definiciones recursivas implícitas. También se utiliza en la definición de la mayoría de las técnicas de evaluación parcial [Pettorossi y Proietti, 1994], donde se relaciona con la condición de cierre (*closedness*) que garantiza la completitud de la transformación.

La operación de plegado es la más costosa de definir, incluso en el caso de lenguajes lógicos o funcionales puros, si se pretende que sea conservativa. Un ejemplo clásico es el programa lógico $\mathcal{P} = \{\mathtt{p(x)} \Leftarrow \mathtt{q(x).},\ \mathtt{q(a).}\}$. Si plegamos la primera cláusula de $\mathcal{P}$ con respecto a sí misma, el programa transformado $\mathcal{P}'$ contiene la regla $\mathtt{p(x)} \Leftarrow \mathtt{p(x)}$, y el objetivo $\Leftarrow \mathtt{p(x)}$ entra en ciclo en $\mathcal{P}'$, mientras que tiene éxito en $\mathcal{P}$ con respuesta computada $\{\mathtt{x} \mapsto \mathtt{a}\}$. De forma similar, el plegado del programa funcional $R = \{\mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(n+1)} \to \mathtt{f(n)}\}$ genera el programa transformado $R' = \{\mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(n+1)} \to \mathtt{f(n+1)}\}$, que deja indefinido $\mathtt{f(n)}$ para todo $\mathtt{n} > \mathtt{0}$.

En la literatura relacionada con transformaciones de programas lógicos, se encuentran diferentes formulaciones de la operación de plegado. Las diferencias dependen principalmente de la forma de derivar la equivalencia entre la conjunción de átomos que van a ser plegados y las llamadas correspondientes. El caso más simple se presenta cuando tal equivalencia se deriva directamente de una cláusula que pertenece al mismo programa sobre el que se realiza la operación de plegado [Gardner y Shepherdson, 1991; Maher, 1987a; Pettorossi y Proietti, 1994]. El *plegado reversible* que se define en [Pettorossi y Proietti, 1994] requiere que los conjuntos de reglas a plegar y las reglas usadas para realizar el plegado sean disjuntos, además de pertenecer al mismo programa. Los pasos de plegado de este tipo son *reversibles* es decir, después de un paso de plegado, siempre existe otro de desplegado que restituye el programa inicial. Las condiciones que se requieren para garantizar la corrección del proceso son sintácticas y muy fáciles de comprobar pero, a menudo, son muy restrictivas ya que, en un sistema de transformación de programas, las reglas que deben ser plegadas pueden haber sido modificadas o eliminadas del programa por transformaciones previas. Por todo ello, en un buen número de propuestas para la transformación de programas lógicos [Pettorossi y Proietti, 1994, 1998; Seki, 1993; Tamaki y Sato, 1984], los dos conjuntos de cláusulas utilizados en una operación de plegado pueden pertenecer a programas distintos. O, en el caso más restrictivo de todos, las cláusulas a plegar deben pertenecer al primer programa de una secuencia obtenida por sucesivas transformaciones.

Una propuesta que está cobrando un creciente interés en el contexto de la transformación de programas declarativos, es la llamada aproximación de "reglas + estrategias", que consiste en definir un conjunto de reglas de transformación elementales (incluyendo invariablemente las operaciones básicas de plegado y desplegado), que se aplican sobre un programa siguiendo una determinada estrategia. Por estrategia entendemos un conjunto de meta–reglas que dirigen de forma automática, guiada por una heurística o asistida por el usuario, el proceso de aplicación de las reglas elementales para construir la secuencia de transformaciones [Pettorossi y Proietti, 1994, 1996b]. Esta técnica puede conseguir resultados tales como: la eliminación de variables innecesarias [Proietti y Pettorossi, 1993], evitar la construcción de estructuras de datos intermedias o el recorrido múltiple de una misma estructura de datos [Burstall y Darlington, 1977], convertir especificaciones (ineficientes) en programas (eficientes) [Komorowski, 1991], realizar la compilación de un programa fuente (es el caso del compilador del lenguaje funcional *Haskell* [Peyton-Jones, 1996]), así como efectuar la especialización de programas respecto a parte de sus datos de entrada. Respecto a este último punto, puede observarse que se comparte objetivo con las técnicas de evaluación parcial. Esencialmente, las técnicas de transformación basadas en la aproximación "reglas + estrategias" son más potentes, aunque poseen un menor grado de automatización (en [Pettorossi y Proietti, 1994, 1996a] se puede encontrar una comparativa detallada entre ambos tipos de técnicas). La principal ventaja de un marco general de transformación de programas estriba en que su potencia es mucho más alta que la de cualquier evaluador parcial (de hecho, las técnicas de especialización pueden verse como un caso particular de las técnicas de transformación [Pettorossi y Proietti, 1996b]) y, aunque su grado de automatización puede presentar ciertas limitaciones, el uso de heurísticas apropiadas y la posibilidad de interacción con el usuario, consigue paliar de forma efectiva estas deficiencias.

Aparte de las reglas básicas de plegado y desplegado, existe en la literatura una extensa gama de definiciones de reglas de transformación elementales de menor rango que las anteriores. Estas reglas son necesarias en un marco genérico de transformación de programas si realmente se pretende derivar programas eficientes. Aunque existe una gran cantidad de variantes con diferentes nomenclaturas (*replacement, instantiation, abstraction, pruning, thinning, fattening*, etc.) y significados [Pettorossi y Proietti, 1994; Bossi *et al.*, 1990], la idea básica subyacente a todas ellas consiste en añadir, modificar o eliminar porciones del programa o del objetivo original. En posteriores apartados detallaremos la esencia de tales operaciones.

En este trabajo estudiaremos, a un doble nivel teórico/práctico, la adaptación del esquema "reglas + estrategias" a un contexto de lenguajes integrados. En la vertiente más teórica nos proponemos investigar las propiedades semánticas de las técnicas de transformación por plegado/desplegado de programas lógicos-funcionales con una semántica operacional basada en diferentes formas de *narrowing*. En el aspecto más pragmático, pretendemos implementar una herramienta aceptablemente robusta donde se apliquen los resultados estudiados anteriormente. Con dicha aplicación generaremos programas transformados que conserven el mismo significado que los originales, pero con niveles mayores de eficiencia, siempre y cuando se apliquen buenas estrategias de transformación. En la literatura especializada en el tema, tanto en el campo lógico [Bossi y Cocco, 1993; Bossi *et al.*, 1990; Gardner y Shepherdson, 1991; Pettorossi y Proietti, 1994, 1996a; Tamaki y Sato, 1984] como en el funcional [Burstall y Darlington, 1977; Kott, 1985; Nielson y Nielson, 1990; Sands, 1995; Scherlis, 1981; Zhu, 1994], se da buena cuenta de las ventajas que aporta en la práctica el uso sistemático de las técnicas comentadas. Dado el gran número de resultados relevantes que las técnicas de transformación de programas han generado en el caso de los lenguajes lógicos y funcionales puros, parece razonable pensar que puede convertirse en una herramienta muy útil también para la optimización de los lenguajes lógico-funcionales y conducir a avances sustanciales en el campo. En la actualidad, todavía no existe ninguna propuesta basada en la aproximación *reglas + estrategias* para el caso de los lenguajes lógico-funcionales, lo que nos invita a investigar, en este entorno, cuáles pueden ser las estrategias de control más refinadas, cómo se pueden adaptar las técnicas existentes para un mecanismo de ejecución eficiente para lenguajes integrados, así como la forma de definir heurísticas apropiadas para guiar el proceso de transformación.

Una tarea prioritaria e imprescindible en el área de la transformación de programas consiste en precisar el tipo de propiedades semánticas que se pretende preservar o mantener a través de la transformación, así como identificar las condiciones de aplicabilidad que aseguran el comportamiento correcto de los programas transformados. La semántica declarativa estándar de un programa lógico-funcional $\mathcal{R}$ viene dada por el conjunto de todas las ecuaciones *ground* o básicas, i.e., sin variables (y generalmente formadas sólo por símbolos constructores), que son ciertas en la teoría subyacente [Hölldobler, 1989]. Sin embargo, esta semántica es incapaz de dar cuenta del observable que mejor captura toda la esencia de una computación lógica: las *respuestas computadas* [Falaschi *et al.*, 1989]. El propósito de este trabajo es considerar técnicas de transformación de programas lógico-funcionales que preserven distintos observables, entre ellos el conjunto de las respuestas computadas (particularmente las constructoras) asociadas a todas las derivaciones de éxito del programa. Formalizaremos diferentes técnicas de transformación para al caso lógico-funcional y estudiaremos las condiciones bajo las cuales se preserva esta semántica para estrategias de *narrowing* diferentes. Finalmente, describiremos un marco de transformación de programas lógico-funcionales donde, además de las reglas básicas de plegado/desplegado, se incluyan otras operaciones de transformación elementales similares a las introducidas en los contextos funcionales y lógicos puros [Burstall y Darlington, 1977; Sands, 1996; Bossi *et al.*, 1990; Pettorossi y Proietti, 1998]. Nuestro objetivo último será aplicar todo el material teórico generado al desarrollo de una herramienta práctica y potente que permita transformar programas declarativos escritos en un lenguaje lógico-funcional moderno.

A continuación resumimos el estado actual en el que se encuentran los estudios sobre las técnicas de transformación, poniendo el énfasis en el caso de los lenguajes lógico-funcionales. Los diversos trabajos que se citan inciden de forma diferente en los dos aspectos esenciales que se consideran en un sistema de transformación potente y efectivo:

1. la corrección de los programas obtenidos, así como las condiciones que se exigen para preservar diferentes semánticas definidas sobre determinados observables

2. y la ganancia en eficiencia que se obtiene al ejecutar el código transformado y mejorado.

1.1.1 Programas funcionales

Como ya se ha dicho, las transformaciones de plegado indextransformación!por plegado/desplegado—seetransformación basada en "reglas + estrategias" y desplegado son las técnicas más simples (y al mismo tiempo las más potentes) que forman el núcleo de la mayoría de los marcos de transformación de programas. Estas operaciones fueron definidas por primera vez en los trabajos pioneros de Burstall y Darlington [1977], que se refieren a programas funcionales. El desplegado consiste en el reemplazamiento de una llamada a función por su respectiva definición (aplicando la correspondiente sustitución). 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 emparejamiento. En [Kott, 1985] se detallan diferentes resultados de corrección de los sistema de plegado/desplegado con respecto a diferentes semánticas propuestas para los programas funcionales puros.

En [Burstall y Darlington, 1977] se desarrolla un sistema general de transformación de programas escrito en un lenguaje funcional basado en ecuaciones recursivas, donde las funciones se definen por emparejamiento de patrones (*pattern matching*) y se asume un sistema de evaluación de llamada por nombre (*call-by-name*). Además de las operaciones de plegado y desplegado, se añaden cuatro nuevas reglas:

1. *Definición.* Introduce una nueva ecuación recursiva cuya parte izquierda no es una instancia de la parte izquierda de alguna definición previa.

2. *Instanciación.* Introduce una instancia de una ecuación existente.

3. *Abstracción.* Reemplaza y enlaza ocurrencias de expresiones a variables mediante el uso de declaraciones locales.

4. *Leyes de primitivas.* Permiten la reescritura usando propiedades de funciones primitivas.

Un punto importante que da un poder significativo al método, está en el hecho de que el plegado puede hacer uso de cualquier definición de la función que va ser plegada. El desplegado también disfruta de la misma propiedad en el marco original, pero aporta menos potencia en la práctica [Sands, 1995; Leuschel *et al.*, 1996]. Todas las reglas de transformación presentadas preservan ciertos resultados de corrección parcial [1] y el sistema global goza del potencial necesario para la síntesis y optimización de programas.

En [Scherlis, 1981] se investiga la especialización de programas por medio de técnicas de transformación. El método presentado es menos general que el anterior pero tiene la ventaja de que se preserva la corrección total de la transformación sin necesidad de exigir restricciones globales. La idea consiste en extender un lenguaje (funcional) de ecuaciones recursivas con un procedimiento para la manipulación de expresiones (*expression procedures*), capaz de dar información no sólo sobre los elementos individuales del programa sino también sobre sus relaciones. Se definen cuatro operaciones de transformación que preservan la equivalencia fuerte entre el programa original y los derivados. Sin embargo, en este caso, los mejores resultados sobre los programas transformados solamente se obtienen cuando se restringe el dominio de los programas originales, de forma similar a lo que ocurre con las técnicas de evaluación parcial.

En [Zhu, 1994] se discute por primera vez, de forma rigurosa, el poder de los sistemas de transformación de programas funcionales basados en reglas de plegado y desplegado al estilo de Burstall y Darlington. El trabajo estudia qué tipos de programas pueden obtenerse tras la transformación de uno dado, y se da una cota de la ganancia máxima en eficiencia que se puede esperar tras una transformación. Es aquí donde se justifica la necesidad de introducir nuevas reglas de transformación (como las originales del marco de Burstall y Darlington -*definición, instanciación, abstracción, aplicación de leyes de primitivas*- y otras, como la selección de un subconjunto completo de reglas equivalentes al programa original) aparte de las operaciones básicas de plegado y desplegado, para obtener programas transformados con mejor complejidad inherente que la de los originales. Al igual que en [Nielson y Nielson, 1990], se muestra que el uso de *eurekas* (similares a las llamadas *definiciones* de otros contextos, o conjuntos de ecuaciones que se añaden a un programa para obtener una transformación eficiente del mismo), es necesario si se quiere obtener versiones realmente mejoradas.

[1] A pesar de que los pasos locales de plegado/desplegado preservan el significado del programa transformado, no siempre se puede garantizar que una secuencia de transformaciones será capaz de producir un programa equivalente.

En [Sands, 1995, 1996] se presentan los mejores resultados sobre transformación de programas funcionales donde, al mismo tiempo que se consigue incrementar la eficiencia, se preserva el significado fuerte (incluyendo la propiedad de terminación) de los programas. En estos trabajos se presenta una condición que garantiza la corrección total de las transformaciones sobre programas recursivos que, por primera vez, utilizan al mismo tiempo funciones de orden superior y estructuras de datos perezosas. El punto de partida es el marco original de Burstall y Darlington [1977] sobre el que se realizan ciertas adaptaciones con el objetivo de extenderlo para dotarlo de la capacidad de trabajar con funciones de orden superior. Básicamente, un paso de transformación se inicia añadiendo nuevas definiciones al programa de partida. A continuación, se transforman las partes derechas de un conjunto de definiciones mediante el uso reiterado de las reglas de instanciación, desplegado y reescritura con leyes de primitivas, hasta obtener un conjunto de expresiones que se simplifican por abstracción y plegado. Estas expresiones serán utilizadas posteriormente en el siguiente paso de transformación hasta obtener el grado de optimización deseado.

Finalmente, Pettorossi y Proietti [1996b] abordan el tema de los sistemas de transformación de programas funcionales en conexión con las técnicas desarrolladas en el área de programación lógica.

1.1.2 Programas lógicos

La literatura relativa a la corrección de los sistemas de transformación en programación lógica es extensa, debido a que en este campo el número de observables y semánticas de interés es muy grande. Además, en el caso de programación lógica, la diferencia entre *síntesis* y *transformación* es tan pequeña [Pettorossi y Proietti, 1994] que la literatura de ambos campos prácticamente se funde.

La aproximación basada en las transformaciones de plegado/desplegado fue adaptada inicialmente a la programación lógica en [Tamaki y Sato, 1984], reemplazando emparejamiento por unificación en las reglas de transformación. El desplegado de un programa lógico consiste, por tanto, en aplicar un paso de resolución a un subobjetivo en el cuerpo de una cláusula de todas las formas posibles y reemplazar la cláusula original por las cláusulas desplegadas. La operación de plegado procede en el sentido contrario, de la misma forma que ocurre en la programación funcional.

El sistema de Tamaki y Sato se diseñó inicialmente sobre programas definidos. Se preserva la equivalencia natural entre programas, es decir, la inducida por la semántica del modelo mínimo de Herbrand. Posteriormente, otros investigadores han demostrado que el sistema básico (basado exclusivamente en transformaciones de desplegado y plegado en el estilo de Tamaki y Sato) también es correcto con respecto a otras muchas semánticas como veremos posteriormente: la semántica de respuestas computadas [Kawamura y Kanamori, 1990; Bossi y Cocco, 1993], la semántica de modelo perfecto [Seki, 1993], la semántica bien fundada (*Well-Founded*) [Seki, 1991] y la semántica por modelo estable [Seki, 1990]. Incluso se preserva la aciclicidad de los programas [Bossi y Etalle, 1994].

Seki [1993] modifica el método anterior restringiendo las condiciones de aplicabilidad. El sistema redefinido disfruta de todas las propiedades del de Tamaki y Sato [1984], pero además preserva el conjunto de los fallos finitos del programa original y es correcto con respecto a la semántica de Kunen [Sato, 1992]. También se han definido condiciones que preservan la terminación universal de los programas [Bossi y Cocco, 1993; Pettorossi y Proietti, 1994].

En [Bossi *et al.*, 1990] se elabora un marco de transformación de programas lógicos con operaciones definidas en trabajos previos que, básicamente, son simples adaptaciones y/o ligeras variaciones para el caso lógico de otras operaciones que ya comentamos para programas funcionales (definición, inserción, borrado y mezcla de objetivos, mezcla de funciones, inserción y borrado de cláusulas y leyes de primitivas [Tamaki y Sato, 1984; Sato y Tamaki, 1984]). Además, se incluyen de forma novedosa otras operaciones nuevas como:

1. *Thinning*. Elimina un átomo en el cuerpo de una cláusula.
2. *Fattening*. Inversamente, añade un átomo al cuerpo de una cláusula.
3. *Pruning*. Elimina una cláusula redundante en un programa.
4. *Constraining*. Restringe un programa al subprograma dado por el cierre transitivo de un átomo.

Todas las reglas son conservativas y se usan con el objetivo de especializar programas dentro de un dominio restringido y prefijado. Se trata de un método de especialización de programas lógicos que permite restringir un programa general a una serie de casos particulares por medio de predicados de restricción (*constraint predicates*). El método usa las operaciones anteriores para restringir el dominio de un programa y propagar la información de las restricciones a través del mismo con el objetivo de simplificarlo siempre que sea posible. La eficiencia se obtiene porque algunas partes de la computación general se convierten en redundantes en la versión especializada y pueden omitirse, o bien pueden precomputarse otras partes. El sistema de transformación es interactivo y guiado por el usuario y hace uso de ciertas heurísticas para dirigir el proceso.

En [Bossi y Cocco, 1993; Bossi *et al.*, 1992], las autores consideran la corrección con respecto a las así llamada *s-semántica* (o semántica de respuestas computadas) y desarrollan condiciones de aplicabilidad para las operaciones de plegado, desplegado, reemplazamiento de objetivos, *thinning* y *fattening* que aseguran la preservación de tal semántica a lo largo de la transformación. En [Bossi y Etalle, 1994; Bossi *et al.*, 1995] se investiga la correción del reemplazamiento de objetivos y de sus operaciones derivadas (básicamente todas ellas consisten en la sustitución de una conjunción de átomos por otra equivalente en el cuerpo de una cláusula), con respecto a las principales semánticas sobre programas con negación, tales como la semántica de Fitting y la semántica de Kunen.

Una de las propiedades fundamentales de los programas que es deseable preservar tras una transformación es ciertamente la terminación. En [Bossi y Cocco, 1993], los autores consideran la terminación universal y muestran como preservarla mediante transformaciones de plegado y desplegado de programas lógicos definidos. Además, en [Bossi y Etalle, 1994] se prueba que cuando el programa inicial, en una secuencia de transformaciones (de plegado/desplegado), es acíclico, entonces también lo es el programa final resultante. Este hecho tiene repercusiones obvias y directas sobre la preservación de la terminación ya que cada programa acíclico es terminante y cada programa definido y terminante es acíclico.

Finalmente, Proietti y Pettorossi han señalado en numerosos textos ([Proietti y Pettorossi, 1990, 1993; Pettorossi y Proietti, 1994, 1996a,b, 1998]) la necesidad de introducir heurísticas o estrategias que dirijan el proceso de transformación, al tiempo que adaptan ciertas estrategias desarrolladas en programación funcional que ayudan a identificar una definición (predicado eureka) mediante la inspección de los árboles de desplegado. En [Pettorossi y Proietti, 1994], por ejemplo, se introducen algunas reglas de transformación (introducción y eliminación de definiciones y reemplazamiento de objetivos) que, junto con las operaciones de plegado y desplegado, permiten construir un sistema automático de transformación de programas lógicos, definidos y normales, al tiempo que se discuten las condiciones de aplicabilidad de tales operaciones así como las semánticas que se preservan en cada caso. Estos autores revisan los fundamentos teóricos de la así llamada aproximación *reglas + estrategias* a la transformación de programas lógicos. En [Pettorossi y Proietti, 1994], establecen un marco unificado para formular y comparar las diferentes reglas propuestas en la literatura. La idea básica es distinguir tres fases en el proceso de transformación: 1) ejecución simbólica, 2) búsqueda de regularidades y 3) extracción del programa transformado. El marco es paramétrico con respecto a las semánticas que se preservan durante la transformación. Se presentan varios conjuntos de reglas de transformación y sus correspondientes resultados de corrección con respecto a las siguientes semánticas: modelo mínimo de Herbrand, semántica de respuestas computadas, fallo finito y semántica de Prolog. También se considera el caso de los programas lógicos normales y, usando el mismo marco, se presentan las reglas que preservan las semánticas de fallo finito, conjunto de éxitos y complección de Clark. Con este marco unificado, es posible describir algunos de los métodos de transformación de programas lógicos más significativos propuestos en la literatura. Aquí se tratan algunas estrategias como la formación de tuplas de predicados, absorción de bucles y generalización, y se muestra que algunas técnicas básicas relacionadas con el control de la compilación, la composición de programas, el cambio de la representación de los datos, la evaluación parcial y la especialización de programas pueden verse como aplicaciones concretas de estas estrategias. Básicamente, las tres estrategias citadas se formulan en forma de algoritmos sucesivamente refinados que, de forma automática, guían el proceso de generación de los predicados eureka. Estos predicados (que se definen de forma recursiva) no aparecen en los programas originales y son los que realmente consiguen evitar la evaluación repetida de algunos subtérminos comunes, visitas múltiples a las mismas estructuras de datos, construcción de enlaces intermedios, etc.

Recientemente, Roychoudhury *et al.* [1999] han propuesto un marco genérico para las transformaciones de plegado/desplegado de los programas lógicos definidos. El sistema es paramétrico con respecto a una serie de argumentos bien escogidos de tal forma que la mayor parte de los sistemas de transformación propuestos previamente en la literatura para este tipo de programas pueden verse como instancias del marco paramétrico de Roychoudhury *et al.* [1999] (por ejemplo, [Tamaki y Sato, 1984], [Tamaki y Sato, 1986], [Kawamura y Kanamori, 1990] y [Kanamori y Fujita, 1987]).

La panorámica más actualizada sobre síntesis y transformación de los programas lógicos puede encontrarse en [Bossi (ed.), 1999].

1.1.3 Programas lógico-funcionales

Como hemos argumentado, existe un creciente interés en programación declarativa por la mecanización de las estrategias de transformación de código, la producción de herramientas interactivas para implementar los transformadores de programas y el desarrollo de compiladores optimizados que hagan uso de las técnicas desarrolladas. Dado el número de resultados relevantes que las técnicas de transformación de programas han generado en el caso de los lenguajes lógicos y funcionales puros, parece razonable pensar que puede convertirse en una herramienta muy útil para la optimización de los lenguajes lógico-funcionales integrados.

Puesto que los lenguajes integrados combinan de una forma práctica y simple las mejores características de los lenguajes funcionales y lógicos modernos, su tratamiento unificado simplifica notablemente los desarrollos en las áreas de aplicación ya existentes. Por ejemplo, las aplicaciones lógicas se mejoran notablemente por el uso de las abstracciones propias de la programación funcional y por la mejora de rendimiento derivada de su principio de ejecución eficiente (determinismo y evaluación perezosa).

Por otro lado, las aplicaciones de la programación funcional se simplifican por el uso de las técnicas de la programación lógica, como la posibilidad de computar con información parcial. Una de las ventajas que se deriva del desarrollo de técnicas de transformación para programas integrados es que pueden dar lugar a avances interesantes en ambos campos y evitar la duplicidad de esfuerzos que ha caracterizado el desarrollo de éstos. Y, a la inversa, es factible también reutilizar y adaptar aquí muchos de los desarrollos producidos en ambas áreas (que a veces solo difieren en aspectos de forma) y para los que se dispone de una extensa literatura difundida y consolidada.

La transformación de programas lógico-funcionales integrados es un área de investigación relativamente nueva. Algunas técnicas dentro de este ámbito, como son la *especialización* y la *evaluación parcial* de programas lógico-funcionales, han sido estudiadas recientemente por Alpuente *et al.* [1997a, 1996b, 1998a, 1999c,g]; Albert *et al.* [1998a]. Sin embargo, el estudio formal y detallado de las técnicas básicas de transformación de programas lógico-funcionales (como son las operaciones de plegado y desplegado) no ha recibido todavía una atención particular en la literatura especializada, a excepción de nuestros trabajos en [Alpuente *et al.*, 1997c,b,d, 1999e,d,b,a]

En [Alpuente *et al.*, 1997c] presentamos una primera aproximación al problema, considerando inicialmente una extensión directa, al caso lógico-funcional, de las transformaciones de plegado/desplegado que hace uso de la relación de *narrowing condicional* para definir el paso de computación elemental. En dicho trabajo, se demuestra que una adaptación ingenua y directa al caso lógico-funcional de las definiciones básicas desarrolladas en los contextos lógico y/o funcional puros no es aplicable, ya que no se preserva el observable de las respuestas computadas (ni siquiera las irreducibles o normalizadas), como ilustra el siguiente ejemplo.

Ejemplo 1 Consideremos el siguiente programa $\mathcal{R}$:

$$\begin{array}{rcl} \mathtt{X} + \mathtt{0} & \to & \mathtt{X} \\ \mathtt{X} + \mathtt{s(Y)} & \to & \mathtt{s(X + Y)} \end{array}$$

Si desplegamos la llamada $\mathtt{s(X + Y)}$ que aparece en la parte derecha de la segunda regla usando las dos reglas del programa:

$$\begin{array}{ll} \mathtt{s(X + Y)} \leadsto_{\{\mathtt{Y}\mapsto\mathtt{0}\}} \mathtt{s(X)} & \text{(usando la primera regla),} \\ \mathtt{s(X + Y)} \leadsto_{\{\mathtt{y}\mapsto\mathtt{s(Z)}\}} \mathtt{s(s(X + Z))} & \text{(usando la segunda regla),} \end{array}$$

entonces obtenemos el siguiente programa residual $\mathcal{R}'$:

$$\begin{array}{rcl} \mathtt{X} + \mathtt{0} & \to & \mathtt{X} \\ \mathtt{X} + \mathtt{s(0)} & \to & \mathtt{s(X)} \\ \mathtt{X} + \mathtt{s(s(Z))} & \to & \mathtt{s(s(X + Z))} \end{array}$$

Ahora, el objetivo $\mathtt{X} + \mathtt{s(Y)} = \mathtt{Z}$ calcula la respuesta (normalizada) $\{\mathtt{Z} \mapsto \mathtt{s(X + Y)}\}$ en $\mathcal{R}$, mientras que el programa desplegado es incapaz de calcular esa respuesta ni ninguna otra más general para el objetivo planteado.

Es importante notar que la transformación ni siquiera preserva la semántica (más débil) de las consecuencias ecuacionales (*ground*) del programa original, como demuestra el siguiente ejemplo.

Ejemplo 2 Consideremos el siguiente programa $\mathcal{R}$ compuesto por una sola regla:

$$\mathtt{f(c(X))} \to \mathtt{f(X)}$$

La parte derecha de la única regla del programa, $\mathtt{f(X)}$, únicamente puede ser reducida a $\mathtt{f(Y)}$ con substitución $\{\mathtt{X} \mapsto \mathtt{c(Y)}\}$. Así obtenemos el siguiente programa $\mathcal{R}'$:

$$\mathtt{f(c(c(Y)))} \to \mathtt{f(Y)}$$

Ahora, la ecuación $\mathtt{f(c(a))} = \mathtt{f(a)}$ solamente es cierta en el programa original.

Debido a los problemas comentados, en [Alpuente *et al.*, 1997c] introducimos una definición generalizada de desplegado más elaborada. Una característica importante de esta nueva definición es que puede ser formulada en términos de evaluación parcial [Alpuente *et al.*, 1996a]. Gracias a esto, las condiciones que aseguran la completitud del proceso de evaluación parcial pueden ser reutilizadas de forma inmediata para formalizar condiciones suficientes que aseguren la completitud del desplegado con respecto al observable de las respuestas computadas por *narrowing*. Se debe notar que la situación es muy diferente en comparación con los casos lógico o funcional puros, donde no se requieren condiciones de aplicabilidad para producir programas transformados equivalentes.

Cuando se utilizan técnicas de *narrowing* refinadas, es necesario adaptar apropiadamente la definición básica de la regla de desplegado y reforzar las condiciones de aplicabilidad para asegurar la corrección de dicha transformación dependiendo de la variante concreta de *narrowing* que se trate. Por ejemplo, en [Alpuente *et al.*, 1997c] presentamos una operación de desplegado basada en *narrowing* impaciente que se define de la manera más natural posible. Con esta estrategia de *narrowing* refinada, el desplegado de programas lógico-funcionales también es correcto y completo bajo condiciones razonables, preservándose el conjunto de las respuestas computadas en programas basados en constructores, completamente definidos y canónicos, o confluentes y terminantes (condiciones usuales que se exigen para la completitud del *narrowing* impaciente). En cierta manera, no es de extrañar que el desplegado impaciente sea el más sencillo y potente dentro del contexto lógico-funcional, debido a la equivalencia de esta estrategia con el mecanismo operacional (SLD-resolución) de los programas lógicos puros ([Bosco *et al.*, 1988]). Desafortunadamente, esta variante de *narrowing* no constituye en la actualidad un punto de referencia para la implementación de lenguajes lógico-funcionales potentes, debido a sus fuertes limitaciones y al hecho de que no permite trabajar con funciones parciales o con funciones que no terminan. De hecho, este refinamiento de *narrowing* sólo es completo con respecto a soluciones constructoras básicas (*ground*) en programas canónicos que siguen la disciplina de constructores (CB) y están completamente definidos (CD) [Fribourg, 1985].

Muchos lenguajes lógico-funcionales modernos como *BABEL* [Moreno-Navarro y Rodríguez-Artalejo, 1992], *Curry* [Hanus *et al.*, 1995] o *TOY* [Caballero-Roldán *et al.*, 1997] utilizan variantes del *narrowing perezoso* como principio operacional. Debido a su carácter perezoso, sólo se evalúan los argumentos de una función si es realmente necesario (lo que permite una evaluación más eficiente de las expresiones funcionales). Uno de los objetivos de esta tesis es definir un marco de transformaciones para lenguajes lógico-funcionales perezosos. La elección de esta estrategia plantea problemas de investigación específicos que no han sido estudiados previamente en la literatura. Por ejemplo, el desplegado de la parte derecha de la cabeza de una regla no debe ir más allá de una forma constructora en cabeza, ya que en caso contrario se corre el riesgo de perder respuestas computadas, como muestra el siguiente ejemplo:

Ejemplo 3 Consideremos el siguiente programa $\mathcal{R}$ basado en constructores:

$$\begin{array}{rcll} \mathtt{f(0)} & \to & \mathtt{0} & (R_1) \\ \mathtt{f(c(X))} & \to & \mathtt{c(f(X))} & (R_2) \\ \mathtt{g(c(X))} & \to & \mathtt{c(0)} & (R_3) \end{array}$$

Obsérvese que la parte derecha de la cabeza de la regla R_2, $\mathtt{c(f(X))}$, está encabezada por un constructor. Si la desplegamos obtenemos el siguiente programa $\mathcal{R}'$:

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

y el término $s = \mathtt{g(f(X))}$ puede ser evaluado en $\mathcal{R}$ a la forma constructora $\mathtt{c(0)}$ con respuesta computada (constructora) $\theta = \{\mathtt{X} \mapsto \mathtt{c(X')}\}$, mientras que no existe ninguna otra respuesta computada θ' para el mismo objetivo s en $\mathcal{R}'$ tal que $\theta' \leq \theta[\mathcal{V}ar(s)]$.

En [Alpuente *et al.*, 1999e] definimos un marco de transformación de programas lógico-funcionales perezosos donde usamos una regla de desplegado basada en *narrowing* necesario. En esta tesis repasaremos las principales aportaciones de esta formulación y la compararemos con otras variantes perezosas de potencia mucho más limitada.

Por otra parte, la transformación de plegado se complica enormemente en el contexto lógico-funcional. Por ejemplo, cuando para su definición se usa la relación de *narrowing* sin refinar, las condiciones que se requieren para preservar el conjunto de las respuestas computadas son tan fuertes que prácticamente ningún programa las cumple. Sin embargo, cuando se usan estrategias de *narrowing* más elaboradas, se pueden conseguir transformaciones de plegado correctas bajo condiciones razonables.

En [Alpuente *et al.*, 1997c] presentamos una operación de plegado basada en *narrowing* impaciente. La definición se inspira en la presentada en [Pettorossi y Proietti, 1994] y, además de ser reversible, es capaz de preservar el conjunto de las respuestas computadas bajo las condiciones usuales (ya mencionadas anteriormente) que se exigen para la completitud del *narrowing* impaciente.

En contextos perezosos, el plegado presenta muchas más complicaciones (como también ocurriera con el desplegado) y su tratamiento no ha sido objeto de ningún estudio específico con anterioridad. En [Alpuente *et al.*, 1999e] presentamos nuestra propuesta de plegado al estilo de Tamaki y Sato [1984] para programas indutivamente secuenciales, que constituye una base ideal para optimizar programas de esta clase cuando se combina con desplegado necesario y el conjunto de reglas auxiliares (introducción/eliminación de definiciones, abstracciones y reemplazamiento algebraico) descritas en el mismo texto. En esta tesis describiremos este marco de transformación (y su implementación en el sistema SYNTH) así como su potencia para transformar programas por composición y formación de tuplas, dos de las estrategias de transformación más potentes que se conocen en los contextos funcional y lógico puros.

En la literatura solamente hemos encontrado tres formulaciones explícitas del plegado/desplegado de programas lógico-funcionales que se basan en alguna forma de *narrowing*. En [Darlington y Pull, 1988], los autores muestran como la *instanciación* (la operación del marco de Burstall y Darlington [1977] que introduce una instancia de una ecuación existente) puede ser incorporada dentro de los pasos de desplegado para conseguir la capacidad (del *narrowing*) de trabajar con variables lógicas mediante el uso de la unificación. De forma similar, los pasos de plegado pueden verse como pasos de *narrowing* usando las ecuaciones invertidas, es decir, orientadas de derecha a izquierda. Sin embargo, en [Alpuente *et al.*, 1997c] mostramos que los pasos de plegado deben generalizar (o desinstanciar) las llamadas en vez de instanciarlas, como ocurre en la programación lógica, en contraste con la forma de actuar del *narrowing* o la SLD-resolución. Es decir, como es de esperar, la operación de plegado actúa verdaderamente en sentido contrario a los pasos de *narrowing* ya que primero se desinstancian o generalizan las reglas plegadas, posteriormente se eliminan las condiciones de las reglas que se utilizaron para hacer el plegado, y finalmente se efectúa un paso de reducción contra las cabezas invertidas de estas últimas reglas. En [Darlington y Pull, 1988] no se da ningún resultado de completitud, aún cuando es necesario imponer ciertas condiciones de aplicabilidad a la transformación.

Otra aproximación muy relacionada con la materia de esta tesis es la presentada en [Dershowitz y Reddy, 1993], donde se formula una técnica basada en la reescritura para la síntesis de programas funcionales que hace uso de la regla de instanciación.

Sin embargo, las manipulaciones necesarias para llevar a cabo las transformaciones de plegado/desplegado son a menudo mucho más complejas que simples instanciaciones, y precisan definiciones de funciones auxiliares e inducción.

Finalmente, los *forward closures* de Dershowitz [1987] producen un tipo de desplegado de reglas de un programa que se puede utilizar para formular condiciones que aseguren la terminación del mismo. En esta misma línea, en [Kurtz, 1992] se introduce el mismo concepto pero referido ahora a un tipo de *narrowing* básico refinado. En esencia, el *basic forward closure* de un programa $\mathcal{R}$ se obtiene como la unión de una secuencia de programas que comienza con el programa original y en la que cualquier otro programa de la secuencia se obtiene desplegando (vía *narrowing* básico) en un solo paso todas las reglas del programa inmediatamente anterior en la secuencia de los programas desplegados. Se verifica que cada uno de los programas en la secuencia es finito, mientras que la secuencia misma no tiene por qué serlo. De hecho, normalmente el *basic forward closure* de un programa (definido como el límite o unión de todos los programas que aparecen en la secuencia de transformaciones) contiene un número infinito de reglas. Por otra parte, así definido, el *basic forward closure* de $\mathcal{R}$ coincide con la semántica por desplegado de $\mathcal{R}$. En [Alpuente *et al.*, 1997c] mostramos un ejemplo de aplicación de la técnica de desplegado de programas lógico-funcionales que está muy relacionada con esta última idea. Siguiendo una técnica similar a la que se utiliza para obtener el *forward closure* de un programa dado, definimos una semántica por desplegado impaciente que modela las respuestas computadas. Dicha semántica es equivalente a la semántica operacional estándar que modela el mismo observable, y consiste básicamente en un conjunto (posiblemente infinito) de reglas incondicionales que se computan como el límite de una secuencia de desplegado que parte del programa original.

1.2 Objetivos de la tesis

A modo de resumen, nuestro objetivo en esta tesis es abordar el primer estudio teórico y la posterior aplicación práctica de las técnicas basadas en la aproximación *reglas + estrategias* para la transformación de programas lógico-funcionales. Los principales resultados que se presentan son los siguientes:

1. **Definición y análisis formal de las transformaciones elementales de plegado/desplegado basadas en *narrowing*.** Como ya se ha indicado, en [Alpuente *et al.*, 1997c] iniciamos la primera línea de actuación en este sentido, tomando la estrategia de *narrowing* condicional ordinario como base. Allí únicamente estudiamos las operaciones elementales de plegado y desplegado, empezando por el estudio sistemático de dichas operaciones para el caso más general del *narrowing* sin restricciones (donde la extensión respecto al caso lógico

o funcional puro no es trivial, como hemos visto). Esto nos permite entender y caracterizar los problemas más comunes causados por el mecanismo de base, ignorando así en un primer estadio las complicaciones propias de una estrategia refinada particular.

2. **Optimización de las reglas básicas de plegado/desplegado usando refinamientos del *narrowing*.** En una segunda etapa de investigación, estudiamos las reglas de plegado/desplegado basadas en refinamientos del *narrowing* sofisticados, como son las estrategias impaciente [Fribourg, 1985], perezosa [Moreno-Navarro y Rodríguez-Artalejo, 1992; Reddy, 1985] y necesaria [Antoy *et al.*, 1993; Hanus, 1994a]. Como señalamos en [Alpuente *et al.*, 1997b, 1999e], se demuestra que, en algunos de estos casos, el poder optimizador de las transformaciones se incrementa notablemente, al tiempo que es posible relajar las condiciones de aplicabilidad sin perder las propiedades de corrección ni completitud. En particular, los mejores resultados se obtienen al considerar como soporte de las transformaciones el *narrowing* necesario, una estrategia de evaluación óptima para programas inductivamente secuenciales que se utiliza en el lenguaje *Curry* [Hanus *et al.*, 1995] y que también es la base del principio de ejecución del lenguaje *TOY* [Caballero-Roldán *et al.*, 1997]).

3. **Semánticas formales**. Otro objetivo de la tesis es investigar las transformaciones bajo estudio en conexión con las semánticas de valores y respuestas computadas (constructoras) de los programas lógico-funcionales, tomando como semánticas operacionales de referencia los diferentes refinamientos del *narrowing* comentados. En particular, y como presentamos por primera vez en [Alpuente *et al.*, 1997c,d], mostraremos que es inmediato construir una semántica por desplegado impaciente que es completamente equivalente a la semántica de respuestas computadas estándar para la clase de programas (basados en constructores, canónicos y completamente definidos) que se ejecutan de forma correcta y completa con *narrowing* impaciente.

4. **Plegado/desplegado y evaluación parcial**. Mostraremos la relación precisa existente entre las técnicas de evaluación parcial de programas lógico-funcionales y las reglas de plegado/desplegado [Alpuente *et al.*, 1997c, 2000b]. Más concretamente, demostramos que, de la misma forma que el desplegado constituye el núcleo de todo algoritmo de evaluación parcial, el plegado puede usarse para simular un postproceso de renombramiento sobre los programas especializados que sirve para incrementar el grado de optimización (e incluso restituir la clase original) de los mismos. Por otra parte, veremos que la estrategia de composición (también conocida como *fusión* y *deforestación*) constituye una vía directa para guiar la secuencia de transformaciones de plegado/desplegado de forma que

conduzca a una versión especializada que puede obtenerse de forma equivalente mediante evaluación parcial de un programa lógico-funcional.

5. **Reglas y estrategias de transformación para los lenguajes integrados**. Una vez investigados todos los aspectos anteriores, nuestra última contribución, que se erige como el objetivo último de esta tesis, es el desarrollo de un marco de transformación de programas lógico-funcionales (perezosos) basado en la aproximación "reglas + estrategias". Para ello, además de las reglas básicas de plegado/desplegado, incluiremos operaciones para la introducción y eliminación de definiciones, así como la abstracción y el reemplazamiento algebraico de reglas en un programa. Veremos que se requieren esfuerzos adicionales de adaptación de estas reglas auxiliares (en especial la de abstracción) a nuestro contexto integrado. Con este conjunto de operaciones obtenemos un sistema completo de transformación que extiende el original de Burstall y Darlington [1977] y permite explotar las estrategias de composición y formación de tuplas para la optimización de programas inductivamente secuenciales que se ejecutan con *narrowing* necesario. Un primer esquema de este sistema de transformación apareció publicado en [Alpuente *et al.*, 1999d,a] y más detalladamente en [Alpuente *et al.*, 2000a].

6. **Evaluación experimental del sistema de transformación**. En la vertiente más aplicada de esta tesis, desarrollamos una herramienta experimental donde se aplican los principales resultados de investigación mencionados anteriormente. Debido a que la automatización total de un sistema de transformaciones no siempre es viable (como ocurre con la transformación de formación de tuplas) o, en el mejor de los casos, no produce programas con una mejora sustancial respecto de los originales, nuestra propuesta consiste en introducir, dentro del entorno de transformación, ciertas heurísticas o estrategias (formuladas especialmente para el contexto lógico-funcional) que dirijan el proceso de transformación, dejando siempre la posibilidad de que el usuario interactúe con el sistema a la hora de tomar decisiones sobre el proceso de optimización. Se trata, en definitiva, de construir y evaluar experimentalmente un transformador (semi-)automático de programas lógico-funcionales presentado como una herramienta interactiva que ayude al programador en las tareas de confeccionar programas eficientes transformados (o especializados) partiendo de versiones originales (no optimizadas) de los mismos. Parte de este trabajo se corresponde con el desarrollo del sistema SYNTH, cuya descripción detallada puede encontrarse en [Alpuente *et al.*, 1999b] y [Alpuente *et al.*, 2000a].

1.3 Organización de la memoria

Antes de pasar a detallar la organización de este documento, puede resultar conveniente realizar unos comentarios relativos al lenguaje utilizado en el mismo. El autor ha hecho un esfuerzo consciente por cuidar su castellano y emplear términos lo más correctos posible dentro de las dificultades de tal empeño, habida cuenta de que muchos de los conceptos presentados han sido acuñados en inglés y no tienen una traducción al castellano consensuada por la comunidad informática. Concretamente, se ha mantenido la terminología inglesa de aquellos conceptos para los que no existe una traducción comúnmente aceptada que refleje la esencia de la actividad denotada (es el caso, por ejemplo, de los tipos de indeterminismo *don't know* o *don't care*). En estos casos, se ha optado por utilizar el término inglés escribiéndolo con letras cursivas. La primera aparición del término inglés viene acompañada, entre paréntesis o en nota a pie de página, de una explicación o traducción al castellano cuando ésta empieza a tener una connotación similar a la del término original o está despuntando su uso en nuestro contexto (pero no es aún popular). Este es el caso del principio operacional de "estrechamiento" (*narrowing*) que nosotros optamos por mantener en inglés en el documento. Finalmente, en ocasiones traducimos de forma directa algunos términos ingleses por aquellos vocablos castellanos más comúnmente utilizados, aún cuando existan otras traducciones que se ajustan más fielmente al significado pretendido. Este es el caso del termino inglés *instance* que nosotros traducimos por "instancia", en vez de "concrección" o "particularización", que sería más correcto. Una vez aclaradas las convenciones sintácticas que utilizaremos en este documento, a continuación pasamos a detallar su organización.

En el Capítulo 2 hemos agrupado los principales conceptos técnicos que se utilizan en el resto de la memoria. Revisamos los conceptos fundamentales de la programación lógico-funcional, incluyendo una descripción genérica del algoritmo del *narrowing*, el mecanismo de ejecución estándar para los programas considerados. A continuación presentamos tres instancias del mismo que pueden verse como refinamientos del mecanismo original. Estas estrategias se referieren a las optimizaciones conocidas como *narrowing* impaciente, perezoso y necesario. El capítulo se cierra con la extensión de los conceptos presentados previamente al caso más general de los programas condicionales.

En el Capítulo 3 nos centramos en el estudio de la regla de desplegado para los programas lógico-funcionales. Ya que esta regla suele plantearse en términos del mismo mecanismo operacional que el del paradigma de programación considerado, en nuestro caso utilizamos el cálculo del *narrowing* para definir la operación. Así, partiendo de una definición genérica de la transformación, estudiamos las distintas instancias de la regla que corresponden a los diferentes refinamientos del *narrowing* vistos en el capítulo anterior. Al analizar el comportamiento de la regla de desplegado para el caso general del *narrowing* condicional sin restricciones, salen a la superficie las fuertes limitaciones que acompañan a la regla sin refinar, al tiempo que se demuestra la íntima relación existente entre esta regla y las técnicas de evaluación parcial de los programas lógico-funcionales. Por otra parte, la transformación de desplegado impaciente se presenta mucho más interesante al disfrutar de resultados de corrección y completitud fuertes bajo condiciones mucho más razonables. Explotando estos resultados, es posible presentar una caracterización alternativa de la semántica operacional de un programa en base a una secuencia reiterada de desplegados impacientes. Al considerar programas no terminantes, la regla de desplegado admite varias versiones, que expresamos en términos de *narrowing* perezoso y necesario, respectivamente. Sin embargo, únicamente en el segundo caso (desplegado necesario) se obtienen resultados (óptimos) de corrección y completitud fuertes, al tiempo que la transformación preserva la clase natural de programas (inductivamente secuenciales) sobre los que se define la propia estrategia de *narrowing* necesario. Este último tipo de desplegado encuentra una aplicación natural en el sistema de transformación de programas cuyo desarrollo constituye el objetivo central de esta tesis y que se presenta completo en el Capítulo 5.

La regla de plegado constituye, junto con la regla de desplegado, el núcleo de todo sistema de transformación basado en la contracción o expansión, respectivamente, de subexpresiones dentro de un programa usando las definiciones del mismo (o de otro programa precedente en la secuencia de transformación). En el Capítulo 4 abordamos el estudio de esta regla que, en general, pretende ser la inversa de la transformación de desplegado. Comenzamos este capítulo presentando tres refinamientos de la regla en función de su grado de reversibilidad. En primer lugar proponemos una versión totalmente reversible, capaz de plegar varias reglas dentro de un único programa. A continuación relajamos la condición de inversibilidad exigiendo que sea una sola regla la que se pliega y efectuando la transformación en el seno de un único programa. Finalmente, mostramos una versión mucho más potente de plegado que es capaz de actuar sobre reglas pertenecientes a cualquier programa de una secuencia de transformación. En este último caso se consideran en particular los programas inductivamente secuenciales, y se muestra que la combinación de esta regla junto con la de desplegado necesario constituye una plataforma ideal para desarrollar un sistema efectivo de transformación para esta clase de programas. A modo de ejemplo, se ilustra cómo es posible simular un proceso de evaluación parcial con renombramiento mediante la aplicación del par de reglas de plegado/desplegado guiadas por una estrategia de composición.

En el Capítulo 5 completamos el conjunto de reglas del sistema de transformación para programas inductivamente secuenciales. Aparte de las reglas básicas de desplegado necesario y plegado, incorporamos al sistema otras reglas auxiliares para la introducción y eliminación de definiciones, operaciones de abstracción y, finalmente, ciertas leyes para primitivas o leyes de reemplazamiento algebraico, obteniendo un conjunto de transformaciones para programas lógico-funcionales que goza de las propiedades deseadas de corrección y completitud fuertes con respecto a la semántica de valores y respuestas computadas. A continuación demostramos que este conjunto de reglas es suficiente para implementar en nuestro contexto la mayor parte de las estrategias de transformación más potentes, incluyendo las de composición y formación de tuplas. En el terreno má práctico, presentamos una serie de resultados experimentales que revalidan el interés de nuestra propuesta. El capítulo se cierra presentando las características principales del entorno SYNTH, una herramienta experimental implementada en Prolog que permite optimizar programas escritos en el lenguaje *Curry*, un lenguaje declarativo multiparadigma basado en *narrowing* necesario [Hanus *et al.*, 1995; Hanus (ed.), 1999]. El sistema implementa las reglas de transformación descritas anteriormente y permite aplicar la estrategia de composición de forma completamente automática, así como la de formación de tuplas (de forma semi-automática).

Finalmente, la tesis termina con un capítulo de conclusiones donde comentamos las principales líneas de trabajo futuro.

Capítulo 2

Preliminares

En este capítulo revisamos los conceptos fundamentales de la programación lógico-funcional que se necesitan en esta tesis. Tras introducir una serie de nociones preliminares en la Sección 2.1, se introduce, en la Sección 2.2, el algoritmo de *narrowing*, el mecanismo de ejecución estándar de los programas lógico-funcionales. En los siguientes apartados se presentan refinamientos (*narrowing* impaciente, perezoso y necesario, subsecciones 2.2.1, 2.2.2 y 2.2.3, respectivamente) de la estrategia general de *narrowing*. El capítulo se cierra presentado una extensión de todos los conceptos definidos previamente para el caso de los programas condicionales (Sección 2.3). Las cuestiones más específicas referentes a las diferentes reglas y estrategias de transformación de programas serán introducidas, posteriormente, al principio del capítulo correspondiente. Para un mayor detalle sobre los conceptos introducidos en este capítulo, se puede consultar [Baader y Nipkow, 1998; Dershowitz y Jouannaud, 1990; Hölldobler, 1989; Klop, 1992; Hanus, 1994b; Antoy *et al.*, 1993].

2.1 Conceptos básicos

Términos y ecuaciones

Denotamos por $\mathcal{X}$ un conjunto infinito (numerable) de variables y por Σ un conjunto de símbolos de función f/n, cada uno con una aridad n asociada. $\mathcal{T}(\Sigma, \mathcal{X})$ y $\mathcal{T}(\Sigma)$ denotan, respectivamente, el conjunto de términos y términos *básicos* (sin variables) construidos sobre $\Sigma \cup \mathcal{X}$ y Σ, respectivamente. Asumimos que el *alfabeto o signatura* $\Sigma = \mathcal{C} \uplus \mathcal{F}$ está particionado en un conjunto $\mathcal{C}$ de *constructores* y un conjunto $\mathcal{F}$ de *funciones* (definidas) u *operaciones*. Escribimos $c/n \in \mathcal{C}$ y $f/n \in \mathcal{F}$ para referirnos a constructores y operaciones n-arios, respectivamente. El conjunto de términos *constructores con variables* de $\mathcal{X}$ se denota por $\mathcal{T}(\mathcal{C}, \mathcal{X})$.

Asumimos la existencia de, al menos, un tipo primitivo *Bool* que contiene los constructores booleanos constantes (de aridad 0) $true$ y $false$. Asumimos además que Σ contiene también el símbolo primitivo de función binario "=" para representar la igualdad, escrito en notación infija, que nos permite interpretar las ecuaciones $s = t$ como términos, con $s, t \in \mathcal{T}(\Sigma, \mathcal{X})$. El término $true$ se considera también una ecuación. En algunos contextos (debido a la presencia de funciones no terminantes) se asume la presencia de un nuevo tipo de símbolo primitivo de función binario "$\approx$" para representar la *igualdad estricta* entre dos términos, que también se escribe en notación infija. La igualdad estricta $\approx$ considera dos términos iguales sólo si éstos se reducen a un mismo término básico y formado únicamente por símbolos constructores.

El conjunto de variables que aparecen en una expresión e se denota por $\mathcal{V}ar(e)$. Un término t es *básico* si $\mathcal{V}ar(t) = \varnothing$. Un término es *lineal* si no contiene apariciones múltiples de una misma variable. Denotamos por $\overline{o_n}$ a la *lista de objetos sintácticos* $o_1, \ldots, o_n$. Un *patrón* es un término de la forma $f(\overline{d_n})$ donde $f/n \in \mathcal{F}$ y $d_1, \ldots, d_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$

Los términos se pueden ver como árboles etiquetados de la forma habitual. Usamos la función estándar *depth* para denotar la máxima profundidad de un término. Así,

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

Las *posiciones* $(p, q, ...)$ de un término t se representan por secuencias (posiblemente vacías) de números naturales que sirven para denotar los subtérminos de t. Las posiciones están ordenadas por el orden de *prefijo*: $p \leq q$ si existe w tal que $p.w = q$. Denotamos por Λ la secuencia vacía. Si p y q son posiciones, escribimos $p \leq q$ si p está *encima* o es un *prefijo* de q, mientras que escribimos $p \perp q$ si p y q son posiciones disjuntas (i.e., no verifican $p \leq q$ ni $q \leq p$). $\mathcal{P}os(t)$ denota el conjunto de posiciones de un término t, que se define recursivamente como sigue:

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

$\mathcal{FP}os(t)$ denota el conjunto de posiciones *no variables* de un término t, mientras que $\mathcal{VP}os(t)$ denota el conjunto de posiciones *variables* de t (i.e., $\mathcal{VP}os(t) = \mathcal{P}os(t) - \mathcal{FP}os(t)$). $t|_p$ denota el subtérmino a la posición p de t, mientras que $t[s]_p$ denota el término t cuyo subtérmino a la posición p ha sido reemplazado por s. $t[p]$ denota al símbolo de función que encabeza al subtérmino $t|_p$, i.e., $t[p] = f$ si $t|_p = f(\overline{t_n})$. Para un conjunto de posiciones disjuntas $P = \{p_1, \ldots, p_n\}$, extendemos el reemplazamiento de subtérminos dentro de un término como $t[s_1, \ldots, s_n]_P = (((t[s_1]_{p_1})[s_2]_{p_2}) \ldots [s_n]_{p_n})$.

Sustituciones y unificadores sintácticos

Denotamos por $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ a la *sustitución* σ con $x_i\sigma = t_i$ para $i = 1, \ldots, n$ (donde $x_i \neq x_j$ si $i \neq j$), y $x\sigma = x$ para cualquier otra variable x. El conjunto $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid x\sigma \neq x\}$ se denomina el *dominio* de σ. Una sustitución σ es *constructora (básica)*, si $x\sigma$ es un término constructor (básico) para todo $x \in \mathcal{D}om(\sigma)$. La sustitución identidad se denota por ϵ. Las sustituciones se extienden a morfismos sobre términos como $f(\overline{t_n})\sigma = f(\overline{t_n\sigma})$ para todo término $f(\overline{t_n})$. Consideramos el *preorden* $\leq$ (*subsumción*) habitual entre sustituciones: $\theta \leq \sigma$ sii $\exists\gamma.\ \sigma = \theta\gamma$ y decimos que θ es *más general* que σ.Este preorden induce un (pre-)orden parcial sobre términos dado por $t \leq t'$ si y sólo si $\exists\gamma.\ t' = t\gamma$.

Un *renombramiento* es una sustitución ρ para la cual existe la inversa ρ^{-1} tal que $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. Dos términos t y t' son *variantes* (uno de otro), si existe un renombramiento ρ tal que $t\rho = t'$. Dada una sustitución θ y un conjunto de variables $W \subseteq \mathcal{X}$, denotamos por $\theta_{\restriction W}$ la sustitución obtenida de θ restringiendo su dominio $Dom(\theta)$ a las variables de W. Escribimos $\theta = \sigma\ [W]$ si $\theta_{\restriction W} = \sigma_{\restriction W}$, y $\theta \leq \sigma\ [W]$ denota la existencia de una sustitución γ tal que $\theta\gamma = \sigma\ [W]$. Denotamos por $Ran(\theta)$ las variables que aparecen en el rango de θ, i.e., si $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, entonces $Ran(\theta) = \mathcal{V}ar(t_1) \cup \ldots \cup \mathcal{V}ar(t_n)$.

Un *unificador* de dos términos s y t es una sustitución σ tal que $s\sigma = t\sigma$. Un conjunto de ecuaciones E es unificable, si existe una sustitución ϑ tal que para toda ecuación $s = t$ en E tenemos que $s\vartheta = t\vartheta$. Decimos que ϑ es un *unificador* de E. Un *unificador más general (mgu)* de un par de términos s y t (resp. un conjunto o conjunción de ecuaciones E) $\theta = mgu(\{s = t\})$ (resp. $\theta = mgu(E)$) siempre existe (bajo renombramiento), i.e., $\theta \leq \sigma$ para cualquier otro unificador σ de $s = t$ (resp. E) (e.g., ver [Lassez *et al.*, 1988]).

Programas

En este trabajo presentamos un programa lógico-funcional mediante un sistema de reescritura de términos.

Un *sistema de reescritura de términos* (SRT) es un conjunto de reglas de reescritura (o reglas de reducción) de la forma $l \to r$, tal que $l \notin \mathcal{X}$ y $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Los términos l y r se denominan *parte izquierda* y *parte derecha* de la regla, respectivamente. Un término s se *reescribe* a un término t, y lo denotamos por $s \to_{p,R} t$, si existe una regla $R = (l \to r) \in \mathcal{R}$, una posición $p \in \mathcal{P}os(s)$ y una sustitución σ tal que $s|_p = l\sigma$ y $t = s[r\sigma]_p$. La parte izquierda instanciada $l\sigma$ de una regla de reducción $(l \to r)$ se denomina *redex* (*reducible expression*). Un redex $t|_p$ de un término t es un *redex más externo* si no existe otro redex $t|_q$ de t tal que $q < p$. Un paso de reescritura dado sobre un redex más externo se denomina *paso de reescritura más externo*. La reescritura (más externa) es la base operacional de los lenguajes funcionales (perezosos). $\rightarrow^+$ y $\rightarrow^*$ denotan el cierre transitivo y el cierre reflexivo y transitivo de la relación $\rightarrow$, respectivamente. Un término s es *irreducible* o está en *forma normal* si no existe ningún término t tal que $s \rightarrow t$. Denotamos por $s\downarrow$ la forma normal de s. Un término s está en *forma normal en cabeza* si no se reduce a un redex[1]. Una sustitución σ se dice *normalizada* si $x\sigma$ está en forma normal para todo $x \in Dom(\sigma)$. Un SRT $\mathcal{R}$ se dice *noetheriano* cuando no existe una secuencia infinita de términos de la forma $s_0 \rightarrow_{\mathcal{R}} s_1 \rightarrow_{\mathcal{R}} \ldots$; $\mathcal{R}$ se dice *confluente* si para todo término s tal que $s \rightarrow^*_{\mathcal{R}} t_1$ y $s \rightarrow^*_{\mathcal{R}} t_2$, entonces $t_1 \downarrow t_2$ (i.e., existe un término t tal que $t_1 \rightarrow^*_{\mathcal{R}} t$ y $t_2 \rightarrow^*_{\mathcal{R}} t$). Un SRT $\mathcal{R}$ noetheriano y confluente se denomina *canónico* o completo.

Un SRT $\mathcal{R}$ es *lineal por la izquierda* si el término l es lineal para toda regla $l \rightarrow r \in \mathcal{R}$. Un SRT está *basado en constructores (CB)* si para toda regla $l \rightarrow r \in \mathcal{R}$, l es un patrón. Diremos que $\mathcal{R}$ es *ortogonal* si, además de ser lineal por la izquierda, para toda regla $l \rightarrow r \in \mathcal{R}$ y para cada subtérmino no variable $l|_p$ de l no existe ninguna regla $l' \rightarrow r' \in \mathcal{R}$ tal que $l|_p$ y l' unifiquen (*no solapamiento*)[2]. En lo que sigue, consideraremos como *objetivo* cualquier expresión arbitraria (término, ecuación o conjunción) posiblemente con variables.

2.2 El procedimiento de *narrowing* y sus refinamientos

En este apartado presentamos el mecanismo de computación asociado al lenguaje que nos permite obtener las soluciones para un objetivo en un programa dado. A continuación recordaremos como se puede extender el cálculo de reescritura sustituyendo el emparejamiento por la unificación de términos. Consideremos un programa $\mathcal{R}$. Si queremos evaluar una función cuyos argumentos son básicos (*ground*), basta emparejar la llamada a función $f(t_1, \ldots, t_n)$ con la parte izquierda l de alguna regla $(l \rightarrow r)$ del programa, realizando una secuencia de pasos de reescritura. Por otro lado, si tenemos una llamada a función cuyos argumentos contienen posiblemente variables (un "objetivo"), entonces generalmente es necesario instanciar estas variables a los términos apropiados para poder aplicar un paso de reescritura. Esto se puede llevar a cabo usando unificación en lugar de emparejamiento en el paso de reescritura, y recibe el nombre de *narrowing* [Fay, 1979; Lankford, 1975; Slagle, 1974]. Informalmente, reducir por *narrowing* una expresión consiste en aplicarle una sustitución tal

[1] Nótese que todo término encabezado por un símbolo constructor está obviamente en forma normal en cabeza. En esta tesis será suficiente con considerar esta noción particular de forma normal en cabeza.

[2] Cuando $\mathcal{R}$ es CB, entonces la condición de ortogonalidad se simplifica, ya que al ser patrones las partes izquierdas de todas sus reglas, es suficiente con que no existan pares de reglas $l \rightarrow r$ y $l' \rightarrow r'$ en $\mathcal{R}$ tal que l y l' unifiquen.

que la expresión resultante se pueda reducir, y entonces reducirla [Hullot, 1980]. El procedimiento de *narrowing* no sólo subsume la reescritura, sino también la resolución SLD de los programas lógicos. Formalmente, decimos que un término t se reduce por *narrowing* a un término t' si:

1. p es una posición no variable de t (i.e., $p \in \mathcal{FP}os(t)$);
2. $(l \to r)$ es una regla de $\mathcal{R}$;
3. σ es una sustitución tal que $t\sigma \to_{p,l\,\to\,r} t'$

En este caso, escribimos $t \leadsto_{p,l\to r,\sigma} t'$, $t \leadsto_{p,\sigma} t'$ o simplemente $t \leadsto_{\sigma} t'$, si queda claro por el contexto. Generalmente, un paso de *narrowing* se consigue unificando un subtérmino (no variable) del objetivo con la parte izquierda de la cabeza de una regla de reescritura, y reemplazando entonces el subtérmino instanciado por la correspondiente parte derecha de la regla instanciada. Veamos un ejemplo.

Ejemplo 4 Dado el programa $\mathcal{R}$:

$$\begin{array}{rcl} \mathtt{inc(X)} & \to & \mathtt{s(X)} \\ \mathtt{dec(s(Y))} & \to & \mathtt{Y} \end{array}$$

se puede construir la siguiente secuencia de pasos de narrowing para el término $\mathtt{inc(dec(Z))}$[3]:

$$\underline{\mathtt{inc(dec(Z))}} \leadsto_{\{\mathtt{X}\mapsto\mathtt{dec(Z)}\}} \mathtt{s(}\underline{\mathtt{dec(Z)}}\mathtt{)} \leadsto_{\{\mathtt{Z}\mapsto\mathtt{s(Y)}\}} \mathtt{s(Y)}$$

con sustitución computada $\{\mathtt{Z} \mapsto \mathtt{s(Y)}\}$ (restringida a las variables del objetivo inicial). Nótese que esta secuencia de reducciones no puede obtenerse usando reescritura, ya que la expresión $\mathtt{dec(Z)}$ no empareja con (no es instancia de) la parte izquierda de ninguna regla.

A menudo se impone que la sustitución σ sea el *mgu* de $t|_p$ y l en la definición de *narrowing*, aunque dicha condición puede relajarse en ocasiones exigiendo únicamente que σ sea un unificador arbitrario (no necesariamente el más general) de $t|_p$ y l. Concretamente, Padawitz [1988] distingue entre *narrowing* y *narrowing más general* dependiendo de si σ es un unificador arbitrario o necesariamente el más general. La restricción de que σ sea un *mgu* es la base de la mayoría de las variantes de *narrowing* conocidas (como las que veremos en los apartados 2.2.1, 2.2.2 y 2.3) debido fundamentalmente a la ventaja de que los unificadores más generales son computables de forma determinista, mientras que en general pueden existir muchos unificadores independientes [Antoy *et al.*, 1993]. Sin embargo, eliminar la restricción de que los

[3]En este ejemplo, y en el resto del trabajo, se utilizará la convención de subrayar los subtérminos considerados para realizar el siguiente paso de *narrowing*.

unificadores utilizados en los pasos de *narrowing* sean los más generales es crucial en la definición de la estrategia (más eficiente y refinada) de *narrowing* necesario que estudiaremos en la Sección 2.2.3.

Como hemos visto, un paso de *narrowing* sobre un objetivo concreto siempre considera tres elementos: posición, regla y sustitución. Básicamente, se trata de dar un paso de reescritura sobre una posición concreta del objetivo una vez que previamente se ha aplicado una determinada sustitución sobre el mismo. Formalizamos el procedimiento de *narrowing* usando un sistema de transición etiquetado cuya relación de transición $\leadsto$ formaliza los pasos de computación.

Definición 2.2.1 (narrowing $\leadsto$) Sea $\mathcal{R}$ un SRT y g un objetivo. Definimos la relación de narrowing $\leadsto$ como la menor relación que satisface:

$$\frac{p \in \mathcal{FP}os(g) \ \wedge \ R = (l \ \rightarrow \ r) \ll \mathcal{R} \ \wedge \ \sigma = mgu(\{g|_p = l\})}{g \ \leadsto_{p,R,\sigma} \ (g[r]_p)\sigma}$$

donde $R \ll \mathcal{R}$ denota que R es una variante nueva de una regla de $\mathcal{R}$, tal que R no contiene variables usadas previamente en la computación ("estandarizada" aparte).

Una *derivación de narrowing* se define como: $g \leadsto_\theta^* g'$ sii $\exists\theta_1, \ldots, \exists\theta_n.\, g \leadsto_{\theta_1} \ \ldots \leadsto_{\theta_n} g'$ y $\theta = \theta_1 \ldots \theta_n$. Decimos que dicha derivación tiene longitud n. Si $n = 0$, entonces $\theta = \epsilon$. Para tratar la unificación sintáctica como un paso de *narrowing*, añadimos al SRT la regla $(x = x \ \rightarrow \ true)$, $x \in \mathcal{X}$. Así, $(s = t) \leadsto_\sigma true$ sii $\sigma = mgu(\{s = t\})$. Denotamos por $\mathcal{R}_+$ la extensión de un SRT $\mathcal{R}$ con la regla $(x = x \ \rightarrow \ true)$. Una *derivación de éxito* para una ecuación e en $\mathcal{R}$ es una derivación de la forma $e \leadsto_\theta^* true$ que usa las reglas de $\mathcal{R}_+$.

Cuando los programas no son terminantes es frecuente considerar que todos los símbolos de la igualdad en los objetivos son la igualdad estricta $\approx$. La semántica de la relación $\approx$ se puede definir mediante el siguiente conjunto confluente StrEq de reglas de definición [Hanus (ed.), 1999; Moreno-Navarro y Rodríguez-Artalejo, 1992]:

$$\begin{array}{rcll} c \approx c & \rightarrow & true & \forall c/0 \in \mathcal{C} \\ c(x_1, \ldots, x_n) \approx c(y_1, \ldots, y_n) & \rightarrow & true \Leftarrow x_1 \approx y_1 \ \wedge \ldots \wedge \ x_n \approx y_n & \forall c/n \in \mathcal{C} \\ true \wedge true & \rightarrow & true & \end{array}$$

Nótese que la igualdad estricta no posee la propiedad reflexiva $t \approx t$ para todos los términos t. Cuando se considera la noción de igualdad estricta en un programa $\mathcal{R}$, su extensión para tratar la igualdad sintáctica es el programa extendido ($\mathcal{R} \cup$ StrEq) que, por abuso, denotaremos también como $\mathcal{R}_+$. Para dar la noción completitud del procedimiento de *narrowing* es preciso introducir previamente los siguientes conceptos. Cada SRT o, equivalentemente, cada teoría de Horn ecuacional $\mathcal{E}$ genera una relación de congruencia más pequeña $=_\mathcal{E}$, llamada *$\mathcal{E}$-igualdad*, sobre el conjunto de términos $\mathcal{T}(\Sigma, \mathcal{X})$ (la menor teoría ecuacional que contiene todas las consecuencias lógicas de $\mathcal{E}$ bajo la relación $\models$ y que obedece los axiomas de la igualdad para $\mathcal{E}$). Dado un conjunto de ecuaciones E, decimos que es *$\mathcal{E}$-unificable* sii existe una sustitución σ tal que, para toda ecuación $(s = t) \in E$, se cumple $s\sigma =_{\mathcal{E}} t\sigma$, i.e. $\mathcal{E} \models (s\sigma = t\sigma)$. La sustitución σ se denomina un $\mathcal{E}$-unificador de E (por abuso de notación, a menudo se denomina simplemente *solución*). Dado un conjunto de variables $W \subseteq \mathcal{X}$, la $\mathcal{E}$-igualdad se extiende a sustituciones de la forma estándar: $\sigma =_{\mathcal{E}} \theta\,[W]$ sii $x\sigma =_{\mathcal{E}} x\theta$, $\forall x \in W$. W se omitirá cuando sea igual a $\mathcal{X}$. Decimos que σ es una *$\mathcal{E}$-instancia* de σ' (y, por tanto, que σ' es más general que σ) bajo W, y lo denotamos por $\sigma' \leq_{\mathcal{E}} \sigma[W]$, sii $\exists\gamma.\ \sigma =_{\mathcal{E}} \sigma'\gamma[W]$. Un conjunto de $\mathcal{E}$-unificadores S de un conjunto de ecuaciones E se dice *completo* sii todo $\mathcal{E}$-unificador σ de E se puede factorizar como $\sigma =_{\mathcal{E}} \theta\gamma\ [\mathcal{V}ar(E)]$, donde $\theta \in S$.

Un algoritmo de *narrowing* se dice *completo* si genera un conjunto completo de $\mathcal{E}$-unificadores para cualquier sistema de ecuaciones de entrada. Formalmente, un procedimiento de *narrowing* es completo para una clase de programas si se cumple la siguiente condición:

$$\begin{array}{lll} \text{si } \mathcal{E} \models g\sigma & \text{entonces} & \text{existe una derivación } g \leadsto^*_\theta \mathit{true} \\ & & \text{tal que } \theta \leq_{\mathcal{E}} \sigma\ [\mathcal{V}ar(g)]. \end{array}$$

El subíndice $\mathcal{E}$ puede eliminarse de la expresión $\theta \leq_{\mathcal{E}} \sigma\ [\mathcal{V}ar(g)]$ cuando sólo consideremos completitud con respecto a sustituciones normalizadas [Middeldorp y Hamoen, 1994]. Por extensión, dicho subíndice también desaparece al considerar sustituciones constructoras, ya que este tipo de sustituciones son un caso particular de las sustituciones normalizadas en sistemas basados en constructores. Se ha demostrado que el procedimiento de *narrowing* es un algoritmo de $\mathcal{E}$-unificación completo para sistemas de reescritura de términos que satisfacen diferentes restricciones [Hanus, 1994b; Hölldobler, 1989; Middeldorp y Hamoen, 1994]. Por ejemplo, lo es para programas canónicos y también para programas confluentes, si nos restringimos sólo a sustituciones normalizadas. Ya que el procedimiento de *narrowing* ordinario genera un espacio de búsqueda enorme, se han desarrollado multitud de estrategias para controlar la selección de los *redexes*, mejorando así la eficiencia de *narrowing* al eliminar derivaciones innecesarias, pero sin perder la completitud del cálculo. Un componente muy importante para conseguir una implementación eficiente del *narrowing* es utilizar una estrategia de selección de *redexes* adecuada, ya que el *narrowing* ordinario (Definición 2.2.1) tiene un alto grado de indeterminismo *don't know*[4]. Más concretamente, la ineficiencia del *narrowing* es el resultado de la combinación de los diferentes grados de libertad del cálculo: 1) la elección del *redex* y 2) la elección de la regla de reescritura. Una *estrategia de narrowing* consiste, entonces, en sustituir algunas elecciones *don't*

[4]Preservamos la terminología en inglés para denotar los dos tipos estándar de indeterminismo: indeterminismo *don't know*, cuando todas las opciones deben ser consideradas, e indeterminismo *don't care*, cuando basta con seleccionar una de las opciones e ignorar el resto.

know por elecciones *don't care* [Cheong y Fribourg, 1992] (concretamente, las que se relacionan con el punto 1). Presentamos a continuación un cálculo de *narrowing* genérico con estrategia, a partir del cual se pueden definir como instancias distintos refinamientos del *narrowing* (y, por supuesto, el propio procedimiento de *narrowing* ordinario). La definición del *narrowing* genérico con estrategia es paramétrica con respecto a la función φ que, dado un programa $\mathcal{R}$, asigna a cada objetivo g un conjunto de tuplas de la forma $\langle p, R, \sigma\rangle$ donde $p \in \mathcal{FP}os(g)$, $R = (l \to r) \ll \mathcal{R}_+$ y $g|_p\sigma = l\sigma$. La motivación para la función φ es permitir la explotación de un subconjunto de los *redexes* del objetivo, en lugar de explotarlos todos, así como detallar que tipo de igualdad se considera y cuál es el unificador que se usa en el paso de *narrowing* (que no siempre tiene porqué ser el más general). De esta forma, se puede reducir el espacio de búsqueda manteniendo, bajo diferentes condiciones, la completitud del cálculo.

El cálculo se define como un sistema de transición etiquetado cuya relación de transición $\overset{\varphi}{\leadsto}$ formaliza los pasos de computación.

Definición 2.2.2 (*narrowing* genérico con estrategia $\overset{\varphi}{\leadsto}$) Dado un SRT $\mathcal{R}$, definimos la relación de narrowing (condicional) genérico con estrategia $\overset{\varphi}{\leadsto}$ como la menor relación que satisface:

$$\frac{\langle p, (l \to r), \sigma\rangle \in \varphi(g)}{g \overset{\varphi}{\leadsto}_{p,(l \to r),\sigma} (g[r]_p)\sigma}$$

donde φ denota una estrategia de *narrowing* arbitraria.

Informalmente, el cálculo de *narrowing* presentado en la Definición 2.2.1, puede verse como una instancia de esta relación de *narrowing* genérico con estrategia, cuando la estrategia de selección se define como:

$$\begin{array}{lll}\varphi_{\mathcal{R}}(g) = \{\langle p, R, \sigma\rangle \mid & p \in \mathcal{FP}os(g) \wedge \\ & R = (l \to r) \ll (\mathcal{R} \cup \{x = x \to true\}) \wedge \\ & \sigma = mgu(\{g|_p = l\})\}\end{array}$$

El número de refinamientos del procedimiento de *narrowing* que se han propuesto en la literatura es enorme. En [Hanus, 1994b] se citan hasta 18 tipos distintos de estrategias de *narrowing*, la mayor parte de las cuales se pueden agrupar en tres clases:

- Sin ninguna estrategia: se deben explotar todos los *redexes* del objetivo en cada paso (e.g., *narrowing* ordinario [Slagle, 1974] y *narrowing* condicional ordinario [Hussman, 1985; Kaplan, 1987]). La definición de *narrowing* vista hasta ahora (para el caso incondicional) y su extensión al caso condicional que veremos posteriormente en la Sección 2.3 caen dentro de esta clase.

- Dar prioridad a los *redexes* más internos: e.g., *narrowing* condicional "innermost" o impaciente [Fribourg, 1985]. Este tipo de estrategias se denominan también *voraces*. En la Sección 2.2.1 describiremos una estrategia de este tipo.

- Dar prioridad a los *redexes* más externos: e.g., *narrowing* perezoso [Reddy, 1985], *narrowing* dirigido por la demanda [Moreno-Navarro y Rodríguez-Artalejo, 1992], y *narrowing* necesario [Antoy *et al.*, 1994]). Dentro de las variantes perezosas, dedicaremos dos secciones de este capítulo a estudiar el refinamiento propuesto por Moreno-Navarro y Rodríguez-Artalejo [1992] (Sección 2.2.2) y, sobre todo, al *narrowing* necesario de Antoy *et al.* [1994] (Sección 2.2.3). Los principales resultados de esta tesis se obtienen con esta última estrategia de *narrowing*, considerada en la actualidad como una de las más prometedoras en el área.

Pasemos a analizar las principales estrategias refinadas de *narrowing* que usaremos en este trabajo.

2.2.1 *Narrowing* impaciente

Presentamos en primer lugar una estrategia de *narrowing* en la que los pasos de computación se realizan sólo sobre las ocurrencias más internas (*innermost*) del objetivo ecuacional. Esta estrategia se corresponde con la estrategia de evaluación de Prolog y con la evaluación impaciente (*eager*) de los lenguajes funcionales, también conocida como *llamada por valor (call-by-value)*, debido a que todos los parámetros de una llamada a función deben ser evaluados antes de poder evaluar la propia función. La mayor parte de los conceptos presentados en este punto son una adaptación de [Fribourg, 1985].

En esta sección consideraremos programas basados en constructores (CB). Esto implica que no pueden haber anidamientos en las partes izquierdas de las cabezas de las cláusulas ni axiomas entre los constructores. Ésta es una clase razonable de programas desde el punto de vista de la programación funcional. Muchas teorías ecuacionales que ocurren en la práctica siguen esta disciplina, e.g., en la especificación de los tipos abstractos de datos. Un símbolo de función se dice completamente definido si no ocurre en ningún término básico en forma normal, es decir, si todos los símbolos de función son reducibles sobre todos los posibles términos básicos (del género apropiado[5]). Un SRT $\mathcal{R}$ se dice *completamente definido* (CD, "completely defined") si todos los símbolos de función definidos (i.e., pertenecientes a $\mathcal{F}$) están completamente definidos. En un SRT completamente definido, el conjunto de términos básicos en forma normal coincide con el conjunto de términos irreducibles (o constructores)

[5] Ya que los géneros no son relevantes para los objetivos de este trabajo, los omitimos por simplicidad y sólo consideramos signaturas con un género. La extensión a signaturas heterogéneas es inmediata [Padawitz, 1988].

$\mathcal{T}(\mathcal{C})$ sobre $\mathcal{C}$. En teorías con un sólo género, es extraño encontrar programas completamente definidos; sin embargo, es usual cuando se usan tipos y cada función se encuentra definida sobre todos los constructores pertenecientes al tipo de sus argumentos. En una estrategia impaciente, la función de selección de redexes φ, asigna a cada objetivo g una de las posiciones más internas asociadas a un patrón de g, según el orden prefijo $\leq$. Sin pérdida de generalidad, de las posibles posiciones más internas, seleccionamos aquélla que aparezca más a la izquierda (*leftmost innermost*). Formalmente, definimos el conjunto de posiciones más internas $Inn(g)$ de un objetivo g como:

$$Inn(g) = \{p \in \mathcal{FP}os(g) \mid g|_p \text{ es un patrón } \}.$$

Además, definimos un orden $\triangleleft$ entre las posiciones de $Inn(g)$ de la forma:

$$p \triangleleft q \Leftrightarrow \exists p_0, p_1, p_2 \in I\!N^*,\ \exists i, j > 0.\ p = p_0.i.p_1 \ \wedge\ q = p_0.j.p_2 \ \wedge\ i < j.$$

Intuitivamente, $p \triangleleft q$ implica que la posición p señala un término que aparece más a la izquierda del término señalado por q.

Definición 2.2.3 (*narrowing innermost* $\overset{\text{NI}}{\leadsto}$) La relación de *narrowing innermost* $\overset{\text{NI}}{\leadsto}$ se define como una instancia de la relación de narrowing genérico con estrategia donde la estrategia φ^{NI} se define como sigue:

$$\begin{aligned}\varphi^{NI}_{\mathcal{R}}(g) = \{\langle p, R, \sigma\rangle \mid \quad & p \in Inn(g) \ \wedge\ \forall q \in Inn(g).\ (p \neq q \ \Rightarrow\ p \triangleleft q) \ \wedge \\ & R = (l \ \to\ r) \ll \mathcal{R}_+ \ \wedge\ \sigma = mgu(\{g|_p = l\})\}\ .\end{aligned}$$

Las computaciones se realizan sobre el programa extendido $\mathcal{R}_+ = \mathcal{R} \cup \{x = x \to true\}$.

Como la estrategia *innermost* será la única variante impaciente de *narrowing* que trataremos en esta tesis, en lo que sigue (y sin posibilidad de confusión) la denotaremos genéricamente por *narrowing* impaciente. Al formular la completitud del *narrowing* impaciente, Fribourg [1985] considera sustituciones constructoras. Como ya hemos comentado, esto no es suficiente para enunciar un resultado de completitud incluso cuando las reglas del programa son confluentes y terminantes, debido a los problemas que plantea la estrategia impaciente ante funciones definidas parcialmente. Fribourg presenta diferentes condiciones adicionales para asegurar la completitud del cálculo. La más importante consiste en considerar únicamente funciones totalmente definidas, lo que implica que cualquier término básico irreducible es constructor. El siguiente ejemplo de Hanus [1994b] muestra la incompletitud del *narrowing* impaciente ante la presencia de funciones parciales.

Ejemplo 5 Consideremos las siguientes reglas, donde a y b son constructores:

$$\begin{aligned}\texttt{f(a,Y)} &\ \to\ \texttt{a} \\ \texttt{g(b)} &\ \to\ \texttt{b}\end{aligned}$$

Si queremos resolver la ecuación $\mathtt{f(X,g(X))} = \mathtt{a}$, entonces existe la siguiente derivación de éxito (donde restringimos las sustituciones a las variables del objetivo) usando *narrowing*:

$$\underline{\mathtt{f(X,g(X))}} = \mathtt{a} \leadsto_{\{\mathtt{X}\mapsto\mathtt{a}\}} \underline{\mathtt{a} = \mathtt{a}} \leadsto \mathtt{true}$$

aplicando la primera regla al objetivo original, i.e., $\{\mathtt{X} \mapsto \mathtt{a}\}$ es una solución para la ecuación inicial. Sin embargo, esta derivación no es impaciente, mientras que la única derivación impaciente no es de éxito:

$$\mathtt{f(X,}\underline{\mathtt{g(X)}}\mathtt{)} = \mathtt{a} \overset{\mathtt{NI}}{\leadsto}_{\{\mathtt{X}\mapsto\mathtt{b}\}} \mathtt{f(b,b)} = \mathtt{a}$$

Por tanto, *narrowing* impaciente no puede calcular la solución.

La siguiente proposición, original de Fribourg [1985], establece la completitud del procedimiento de *narrowing* impaciente para programas canónicos que sigan la disciplina CB-CD.

Proposición 2.2.4 (completitud del *narrowing* impaciente)
Sea $\mathcal{R}$ un SRT canónico CB-CD, g un objetivo ecuacional y σ una solución básica constructora tal que $\mathcal{V}ar(g) \subseteq \mathcal{D}om(\sigma)$. Entonces, existe una sustitución de respuesta computada θ para $\mathcal{R} \cup \{g\}$ usando $\overset{\mathtt{NI}}{\leadsto}$ tal que $\theta \leq \sigma$ $[\mathcal{V}ar(g)]$.

La condición $\mathcal{V}ar(g) \subseteq \mathcal{D}om(\sigma)$ en la premisa de la proposición anterior garantiza que $g\sigma$ sea básico. En [Vidal, 1996] encontramos el siguiente ejemplo que revela que esta condición no puede ser eliminada, lo cual se ignora erróneamente, por ejemplo, en [Hanus, 1994b; Hölldobler, 1989].

Ejemplo 6 Consideremos el siguiente programa canónico CB-CD $\mathcal{R}$:

$$\begin{array}{rcl} \mathtt{f(0,Y)} & \to & \mathtt{Y} \\ \mathtt{g(0)} & \to & \mathtt{0} \end{array}$$

La sustitución básica constructora $\sigma = \{\mathtt{X} \mapsto \mathtt{0}\}$ es una solución de la ecuación $\mathtt{f(X,g(Y))} = \mathtt{g(Y)}$. Sin embargo, narrowing impaciente no es capaz de computar una respuesta más general (de hecho, sólo computa la respuesta $\{\mathtt{X} \mapsto \mathtt{0}, \mathtt{Y} \mapsto \mathtt{0}\}$). En este caso, la sustitución σ no satisface la condición $\mathcal{V}ar(g) \subseteq \mathcal{D}om(\sigma)$.

Resulta fácil extender esta estrategia a teorías con funciones que no están completamente definidas, añadiendo simplemente la llamada *regla de reflexión*, que permite ignorar una llamada a función más interna cuando ésta no puede ser reducida [Hölldobler, 1989]. Por simplicidad, asumimos que todas las funciones están completamente definidas y, por tanto, *narrowing* impaciente es suficiente para computar todas las soluciones. Este tipo de estrategia, con algunos refinamientos, es la base de lenguajes lógico-funcionales como ALF [Hanus, 1990], *eager-BABEL* [Kuchen *et al.*, 1990a], SLOG [Fribourg, 1985] o LPG [Bert y Echahed, 1986, 1994]. Como ya hemos comentado, algunas de las restricciones sobre los programas (e.g., estar completemente definidos), se pueden eliminar realizando ligeras variaciones en la estrategia φ^{NI} del cálculo. Sin embargo, la exigencia de que las reglas del programa sean terminantes, no puede ser eliminada sin pérdida de completitud. Esto puede ser un inconveniente cuando se quieren explotar técnicas típicas de la programación funcional como, por ejemplo, las que manipulan estructuras de datos infinitas. En el siguiente punto, presentamos una estrategia de *narrowing* perezoso que preserva la completitud del *narrowing* para programas no terminantes.

2.2.2 *Narrowing* perezoso

La estrategia de evaluación perezosa (*lazy evaluation*) en los lenguajes funcionales consiste, básicamente, en retrasar la evaluación de los argumentos de una función hasta que su evaluación sea necesaria para computar el resultado. Este tipo de estrategia se relaciona con el concepto de *llamada por nombre (call-by-name)* debido a que (algunos de) los parámetros de una llamada a función se pasan como una referencia a su nombre y no a sus valores (que únicamente son evaluados si son "demandados" para poder evaluar la propia función). Las estrategias perezosas poseen varias propiedades importantes. En primer lugar, permiten trabajar con estructuras de datos infinitas, ya que los términos no necesitan ser siempre evaluados completamente. Por otra parte, dado un término t, si existe alguna estrategia de evaluación que permite computar la forma normal de t de manera finita, la estrategia perezosa también termina computando dicho valor. Dado que el concepto de posición "demandada" tiene varias interpretaciones posibles, la adaptación de esta estrategia a los lenguajes lógico-funcionales ha dado lugar a distintas estrategias perezosas de *narrowing* (ver, por ejemplo, [Antoy *et al.*, 1994; Darlington y Guo, 1989; Echahed, 1988; Kuchen *et al.*, 1990b; Moreno-Navarro y Rodríguez-Artalejo, 1992; Reddy, 1985; You, 1989]). En nuestro caso, vamos a definir una estrategia de *narrowing* perezoso guiado por la demanda en el estilo de Reddy [1985] y, por tanto, similar al mecanismo operacional del lenguaje lógico-funcional *BABEL* [Moreno-Navarro y Rodríguez-Artalejo, 1992]. Informalmente, esta estrategia sólo permite realizar pasos de *narrowing* sobre *redexes* internos del objetivo cuando estos son *demandados* por la parte izquierda de alguna regla del programa.

Debido a la presencia de funciones no terminantes, los resultados de completitud para *narrowing* perezoso se establecen con respecto a la igualdad estricta "$\approx$", que como ya hemos dicho anteriormente, considera la identidad entre objetos finitos, i.e., dos términos son iguales sólo si éstos se reducen a un mismo término básico y formado únicamente por símbolos constructores. La igualdad estricta es la única definición adecuada de la igualdad para el caso de programas no terminantes. Cuando trabajamos con la relación de *narrowing* perezoso, consideramos que todos los símbolos de la igualdad en los objetivos son $\approx$. Puesto que no se exige que las reglas de los programas sean terminantes, la propiedad de confluencia de los programas se vuelve indecidible. Pese a todo, existen una serie de condiciones suficientes (decidibles) que garantizan la confluencia en el caso de programas CB, incluso en ausencia de la propiedad de terminación: linealidad por la izquierda y no solapamiento de reglas [González-Moreno *et al.*, 1992; Moreno-Navarro y Rodríguez-Artalejo, 1992]. A continuación, formulamos un procedimiento de *narrowing* perezoso que es completo para programas que satisfacen esta doble condición, que como ya hemos avanzado anteriormente, es conocida también como *ortogonalidad* [Huet y Lévy, 1992; Klop, 1992].

Cuando trabajamos con programas CB lineales por la izquierda, la unificación entre las partes izquierdas de las reglas y los términos del objetivo posee siempre una serie de características comunes que, en general, denominaremos como *problema de unificación lineal*.

Definición 2.2.5 (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 $f(d_1, \ldots, d_n)$ y $f(t_1, \ldots, t_n)$ no comparten variables y $f(d_1, \ldots, d_n)$ es un patrón lineal.

Los problemas de unificación lineal se pueden resolver mediante cualquier versión del algoritmo de unificación (ver, por ejemplo, [Lassez *et al.*, 1988]). Siguiendo [Moreno-Navarro y Rodríguez-Artalejo, 1992], distinguimos el caso en el que la unificación no tiene éxito debido a un conflicto entre un constructor c y un símbolo de función f, ya que tal situación se puede considerar como una demanda de mayor evaluación de f. El siguiente algoritmo es una reformulación del algoritmo de unificación para el caso de los problemas de unificación lineales de Moreno-Navarro y Rodríguez-Artalejo [1992]. Nótese que, debido a la linealidad por la izquierda, la comprobación conocida como "occur-check" no es necesaria.

Definición 2.2.6 (configuración LU) Una configuración LU es un par (U, σ), donde U es un conjunto y σ es una sustitución.

Definición 2.2.7 (relación de unificación $\to_{\mathsf{LU}}$) Definimos la relación de unificación $\to_{\mathsf{LU}}$ como la menor relación que satisface:

1. $(\{c(d_1, \ldots, d_n) \downarrow_p c(t_1, \ldots, t_n)\} \cup U, \sigma) \to_{\mathsf{LU}} (\{d_1 \downarrow_{p.1} t_1, \ldots, d_n \downarrow_{p.n} t_n\} \cup U, \sigma)$, donde $c/n \in \mathcal{C}$, $n \geq 0$.

2. $(\{x \downarrow_p t\} \cup U, \sigma) \to_{\mathsf{LU}} (U\{x \mapsto t\}, \sigma\{x \mapsto t\})$, donde $t \notin \mathcal{X}$.

3. $(\{d \downarrow_p x\} \cup U, \sigma) \to_{\mathsf{LU}} (U\{x \mapsto d\}, \sigma\{x \mapsto d\})$.

4. $(\{c(d_1, \ldots, d_n) \downarrow_p c'(t_1, \ldots, t_m)\} \cup U, \sigma) \to_{\mathsf{LU}} (\{\mathtt{fail}\}, \sigma)$, donde $c/n, c'/m \in \mathcal{C}$, $c \neq c'$, y $n, m \geq 0$.

Es fácil observar que las computaciones $\rightarrow^*_{\mathsf{LU}}$ pueden terminar de tres formas distintas: computando el mgu de los términos de entrada, devolviendo `fail` (en el caso de que los términos no unifiquen) o devolviendo un conjunto de pares $c(d_1, \ldots, d_n) \downarrow_p f(t_1, \ldots, t_n)$, en el que cada posición p indica una posición demandada. La siguiente definición formaliza dicho comportamiento.

Definición 2.2.8 (comportamiento de $\rightarrow_{\mathsf{LU}}$)
Sea $\Gamma = \langle f(d_1, \ldots, d_n), f(t_1, \ldots, t_n)\rangle$ un problema de unificación lineal. Dada la computación[6]:

$$(\{d_1 \downarrow_1 t_1, \ldots, d_n \downarrow_n t_n\}, \epsilon) \rightarrow^*_{\mathsf{LU}} (U, \sigma) \not\rightarrow_{\mathsf{LU}}$$

definimos la función $\mathsf{LU}(\Gamma)$ como sigue:

$$\mathsf{LU}(\Gamma) = \begin{cases} (\textsc{Success}, \sigma) & \text{si } U = \varnothing \\ (\textsc{Fail},) & \text{si } U = \{\texttt{fail}\} \\ (\textsc{Demand}, P) & \text{en cualquier otro caso, donde} \\ & P = \{p \mid (c(d'_1, \ldots, d'_m) \downarrow_p g(t'_1, \ldots, t'_k)) \in U\} \\ & \text{es el conjunto de posiciones demandadas.} \end{cases}$$

Informalmente, una estrategia perezosa debe seleccionar, en cada paso de computación, la ocurrencia del término completo, excepto cuando hayan subtérminos demandados de acuerdo con el algoritmo de unificación lineal. De esta forma, una estrategia de *narrowing* perezoso solamente selecciona los redexes demandados de un término. A continuación formulamos la relación de *narrowing perezoso* de una forma similar a como lo hemos hecho para el *narrowing* impaciente, por medio de un sistema de transición que es instancia de la relación genérica de *narrowing* con estrategia.

Definición 2.2.9 (*narrowing* perezoso $\overset{\text{NP}}{\leadsto}$) La relación de narrowing perezoso $\overset{\text{NP}}{\leadsto}$ se define como una instancia de la relación de narrowing genérico con estrategia donde la estrategia φ^{NP} se define de manera inductiva como sigue:

$$\begin{array}{lcl} \varphi^{NP}(g) & = & \bigcup_{k=1}^{m} \varphi\text{\_}(g, \Lambda, k) \\ \varphi\text{\_}(g, p, k) & = & \text{si } l_k[\Lambda] = g[p] \text{ entonces} \\ & & \begin{cases} \{\langle p, R_k, \sigma\rangle\} & \text{si } \mathsf{LU}(\langle l_k, g|_p\rangle) = (\textsc{Success}, \sigma) \\ \bigcup_{p' \in P} \bigcup_{k'=1}^{m} \varphi\text{\_}(g, p.p', k') & \text{si } \mathsf{LU}(\langle l_k, g|_p\rangle) = (\textsc{Demand}, P) \\ \varnothing & \text{si } \mathsf{LU}(\langle l_k, g|_p\rangle) = (\textsc{Fail},) \end{cases} \end{array}$$

donde $R_k = (l_k \rightarrow r_k) \ll \mathcal{R}_+$, $1 \leq k \leq m$. Las computaciones se realizan usando las reglas del programa extendido $\mathcal{R}_+ = (\mathcal{R} \cup \textsc{StrEq})$.

[6]Por $(U, \sigma) \not\rightarrow_{\mathsf{LU}}$ indicamos que no es posible realizar una transición $\rightarrow_{\mathsf{LU}}$ a partir del estado (U, σ).

Tal y como se expone en [Moreno-Navarro y Rodríguez-Artalejo, 1992], la sustitución computada σ en cada paso de *narrowing* perezoso puede verse como la unión de dos sustituciones $\sigma_g \cup \sigma_r$ representando las restricciones de σ a las variables del objetivo y a las variables de la regla empleada, respectivamente. Es interesante destacar que, debido a la linealidad por la izquierda y a la disciplina de constructores de los programas, la sustitución σ_g no contiene nunca símbolos de función definidos [Moreno-Navarro y Rodríguez-Artalejo, 1992], i.e., es una sustitución constructora. La siguiente proposición, original de Moreno-Navarro y Rodríguez-Artalejo [1992], establece la completitud del cálculo.

Proposición 2.2.10 (completitud del narrowing perezoso) *Sea $\mathcal{R}$ un programa CB ortogonal, g un objetivo ecuacional y σ una sustitución básica constructora tal que, para toda ecuación $s \approx t$ en g, se cumple $(s\sigma)\!\downarrow = (t\sigma)\!\downarrow$, donde $(t\sigma)\!\downarrow \in \mathcal{T}(\mathcal{C})$. Entonces, existe una sustitución de respuesta computada θ para $\mathcal{R} \cup \{g\}$ usando $\overset{\text{NP}}{\leadsto}$ tal que $\theta \leq \sigma$ $[\mathcal{V}ar(g)]$.*

A diferencia de lo que ocurre en las derivaciones de reescritura perezosas, la aplicación de diferentes reglas a una posición (demandada) particular puede conducir a diferentes soluciones cuando se consideran derivaciones de *narrowing* perezosas. Además, al intentar aplicar diferentes reglas a un objetivo, puede darse el caso de que los argumentos a evaluar requeridos sean distintos. Como consecuencia, las estrategias simples de *narrowing* perezoso corren el riesgo de realizar pasos de computación innecesarios en derivaciones de *narrowing*. El siguiente ejemplo ilustra este punto [Hanus, 1994b].

Ejemplo 7 Consideremos el siguiente conjunto de reglas que definen la relación $\leq$ y la suma sobre los números naturales:

$$\begin{array}{rcllcl}
\mathtt{0} & \leq & \mathtt{N} & \to & \mathtt{true} \\
\mathtt{s(M)} & \leq & \mathtt{0} & \to & \mathtt{false} \\
\mathtt{s(M)} & \leq & \mathtt{s(N)} & \to & \mathtt{M} \leq \mathtt{N} \\
\mathtt{0} & + & \mathtt{N} & \to & \mathtt{N} \\
\mathtt{s(M)} & + & \mathtt{N} & \to & \mathtt{s(M+N)}
\end{array}$$

Ahora, pretendemos resolver la ecuación $\mathtt{X} \leq \mathtt{X} + \mathtt{Y} \approx \mathtt{B}$ por *narrowing* perezoso. Una primera solución podría computarse aplicando la primera regla para $\leq$ sin evaluar el subtérmino $\mathtt{X} + \mathtt{Y}$:

$$\underline{\mathtt{X} \leq \mathtt{X} + \mathtt{Y}} \approx \mathtt{B} \overset{\text{NP}}{\leadsto}_{\{\mathtt{X} \mapsto \mathtt{0}\}} \underline{\mathtt{true} \approx \mathtt{B}} \overset{\text{NP}}{\leadsto}_{\{\mathtt{B} \mapsto \mathtt{true}\}} \mathtt{true}$$

De esta forma tenemos que $\{\mathtt{X} \mapsto \mathtt{0}, \mathtt{B} \mapsto \mathtt{true}\}$ es una solución para la ecuación inicial. A la hora de calcular nuevas soluciones, la alternativa consistiría en aplicar la segunda y tercera reglas para $\leq$, pero en ambos casos es necesario evaluar el subtérmino $\mathtt{X} + \mathtt{Y}$.

Si escogemos la regla $\mathtt{0} + \mathtt{N} \to \mathtt{N}$ para dar el primer paso de evaluación, obtenemos la siguiente derivación de *narrowing* perezoso:

$$\mathtt{X} \leq \underline{\mathtt{X} + \mathtt{Y}} \approx \mathtt{B} \overset{\text{NP}}{\rightsquigarrow}_{\{\mathtt{X}\mapsto\mathtt{0}\}} \underline{\mathtt{0} \leq \mathtt{Y}} \approx \mathtt{B} \overset{\text{NP}}{\rightsquigarrow}_{\{\}} \underline{\mathtt{true} \approx \mathtt{B}} \overset{\text{NP}}{\rightsquigarrow}_{\{\mathtt{B}\mapsto\mathtt{true}\}} \mathtt{true}$$

En el segundo paso sólo la primera regla para $\leq$ es aplicable. La solución computada $\{\mathtt{X} \mapsto \mathtt{0}, \mathtt{B} \mapsto \mathtt{true}\}$ es idéntica a la anterior, pero ahora se han dado pasos superfluos, ya que para alcanzar esta solución no es necesario evaluar el término $\mathtt{X} + \mathtt{Y}$.

Para evitar pasos innecesarios en derivaciones de *narrowing* perezosas es posible cambiar el criterio de instanciación de variables y aplicación de reglas. En el ejemplo anterior, la evaluación de un término de la forma $\mathtt{X} \leq \mathtt{t}$ podría comenzar por instanciar la variable $\mathtt{X}$ bien a $\mathtt{0}$ o bien a $\mathtt{s}(\Box)$ y evaluar t únicamente en el segundo caso, cuando es estrictamente necesario. Este refinamiento de *narrowing* perezoso se conoce como *narrowing* necesario y constituye la base del lenguaje *Curry* [Antoy *et al.*, 1994; Hanus *et al.*, 1995]. La nueva estrategia disfruta de propiedades importantes, como la optimalidad e independencia de soluciones, y puede verse como un refinamiento del *narrowing* perezoso [Julián, 2000].

2.2.3 *Narrowing* necesario

La estrategia de *narrowing* necesario [Antoy *et al.*, 1994] es completa para programas *inductivamente secuenciales*, al tiempo que únicamente computa pasos *inevitables* para resolver objetivos. Dicha estrategia es óptima con respecto a la longitud de las derivaciones y al número de soluciones computadas. Como resumen, diremos que las principales ventajas de esta estrategia frente a otras existentes en la literatura incluyen: la gran clase de sistemas de reescritura sobre los que es aplicable (sistemas inductivamente secuenciales), la optimalidad o minimalidad en la longitud de las derivaciones cuando se trabaja con grafos, la independencia de los unificadores que calcula y la facilidad con que puede ser implementada [Antoy *et al.*, 1993].

La noción de *inevitable* o *necesario* es bien conocida en reescritura [Huet y Lévy, 1992]. Los sistemas *ortogonales* tienen la propiedad de que, para todo término t que no esté en forma normal, existe un redex, llamado *necesario*, que eventualmente debe ser reducido para calcular la forma normal de t [Huet y Lévy, 1992; Klop y Middeldorp, 1991; O'Donnell, 1977]. Más aún: la reducción reiterada de redexes necesarios es suficiente para calcular la forma normal de un término, si ésta existe. Es decir, en sistemas ortogonales, es suficiente considerar redexes necesarios. Sin embargo, este concepto es indecidible en el caso general y se han propuesto familias restringidas de programas sobre las que es posible decidir cuando el conjunto de redexes es necesario. A continuación, pasamos a extender estos resultados al caso de la relación de *narrowing* sobre programas inductivamente secuenciales (que son una subclase de los sistemas ortogonales sobre la que el concepto de redex necesario es decidible)[7]. Al intentar extender algunos conceptos de la reescritura necesaria al caso de *narrowing*, nos encontramos con el hecho de que, ahora, debemos considerar la instanciación de un término antes de que éste pueda ser reducido. Afortunadamente, este problema tiene una solución eficiente sobre programas inductivamente secuenciales, que pasa por relajar el requerimiento (exigido hasta ahora en todas las variantes de *narrowing* estudiadas) de que los unificadores usados en los pasos de *narrowing* fuesen los más generales. Los unificadores que se calculan ahora, de alguna forma "anticipan" ciertos enlaces para las variables que, de todos modos, se hubieran computado más adelante en la derivación. Esto complica ligeramente la definición de la nueva estrategia, pero en contrapartida permite obtener el refinamiento de *narrowing* más eficiente conocido, que servirá de base para los resultados de transformación más importantes de esta tesis.

Sea $A = (t \to_{u,l\to r} t')$ un paso de reescritura aplicado sobre una posición u de un término arbitrario t usando una regla $l \to r$ y obteniendo un nuevo término t'. El conjunto de *descendientes* [Huet y Lévy, 1992] de una posición v tras el paso A, denotado por $v\backslash A$ es:

$$v\backslash A = \begin{cases} \varnothing & \text{si } u = v, \\ \{v\} & \text{si } u \not\leq v, \\ \{u.p'.q \mid r|_{p'} = x\} & \text{si } v = u.p.q \text{ and } l|_p = x, \text{ donde } x \in \mathcal{X}. \end{cases}$$

Todos los descendientes $\{v_1, \ldots, v_n\}$ de una misma posición v se denominan *hermanos* entre sí, mientras que v se denomina *antecedente* de todos ellos. El conjunto de *descendientes* [Huet y Lévy, 1992] de una posición v tras una secuencia de reducción B se define inductivamente como sigue:

$$v\backslash B = \begin{cases} \{v\} & \text{si } B \text{ es una derivación nula,} \\ \displaystyle\bigcup_{w \in v\backslash B'} w\backslash B'' & \text{si } B = B'B'', \text{ donde } B' \text{ es el paso inicial de } B. \end{cases}$$

Dado un conjunto de posiciones P, definimos $P\backslash B = \bigcup_{p\in P} p\backslash B$. Decimos que una posición p de un término t es *necesaria* si y sólo si en toda derivación de reescritura que reduce t a su forma normal, algún descendiente de $t|_p$ es reducido a la posición raíz (i.e., el paso de reescritura se aplica sobre la posición Λ de $t|_p$). Como una posición identifica unívocamente a un subtérmino dentro un término, la noción de descendiente para (sub)términos se extiende directamente a partir de la correspondiente noción para posiciones.

Una noción equivalente (y más intuitiva) de descendiente de una posición se propone en [Klop y Middeldorp, 1991]. Sea $t \to^* t'$ una secuencia de reducción y s un

[7] Los sistemas inductivamente secuenciales son una clase de programas basados en constructores incluidos en los sistemas *fuertemente secuenciales* que, a su vez, son ampliamente utilizados para implementar eficazmente cómputos necesarios [Huet y Lévy, 1992].

subtérmino de t. Los descendientes de s en t' se calculan como sigue: subrayar la raíz de s y realizar la secuencia de reducción $t \to^* t'$. Entonces, cada subtérmino de t' con la raíz subrayada es un descendiente de s.

Ejemplo 8 Consideremos la operación que duplica su argumento mediante una suma (las reglas para la suma aparecen en el Ejemplo 7).

$$\mathtt{R} \;=\; \mathtt{double(X)} \;\to\; \mathtt{X} + \mathtt{X}$$

En la siguiente reducción de $\mathtt{double}(\mathtt{0}+\mathtt{0})$ mostramos, como términos subrayados, los descendientes de $\mathtt{0}+\mathtt{0}$:

$$\mathtt{double}(\underline{\mathtt{0}+\mathtt{0}}) \;\to_{\;\mathtt{1,R}}\; (\underline{\mathtt{0}+\mathtt{0}}) + (\underline{\mathtt{0}+\mathtt{0}})$$

El conjunto de descendientes de la posición 1 en la reducción anterior es $\{1,2\}$.

Para dar una definición precisa de la clase de programas inductivamente secuenciales y de la estrategia de *narrowing* necesario, presentamos a continuación el concepto de *árbol definicional* [Antoy, 1992]. Se dice que $\mathcal{P}$ es un árbol definicional con patrón π si y sólo si la profundidad de $\mathcal{P}$ es finita, π es un patrón y se da alguno de los siguientes casos:

$\mathcal{P} = rule(\pi \;\to\; r),$ donde $\pi \;\to\; r$ es una variante de alguna regla de $\mathcal{R}$.

$\mathcal{P} = branch(\pi, o, \pi_1, \ldots, \pi_n),$ donde o es una posición de una variable en π (la *posición inductiva*), $c_1, \ldots, c_n \in \mathcal{C}$ son diferentes constructores, para algún $n > 0$, y, para todo $\pi_1, \ldots, \pi_n$ tenemos que $\pi_i = \pi[c_i(\overline{x_{n_i}})]_o$, donde n_i es la aridad de c_i y $\overline{x_{n_i}}$ son variables nuevas.

El *árbol definicional* de un conjunto finito de patrones lineales (para una función)[8] S es un conjunto no vacío $\mathcal{P}$ de patrones lineales parcialmente ordenados por el orden de subsumción, que tiene las siguientes propiedades:

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

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

Propiedad del padre: Si $\pi \in \mathcal{P}$, $\pi \neq pattern(\mathcal{P})$, existe un único $\pi' \in \mathcal{P}$, llamado el *padre* de π (siendo π el *hijo* de π'), tal que $\pi' < \pi$ y no existe otro patrón $\pi'' \in \mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ que verifique $\pi' < \pi'' < \pi$.

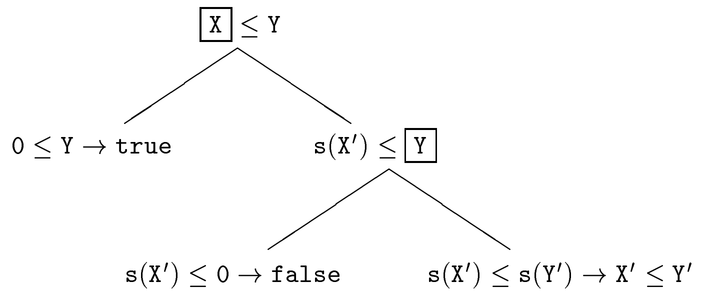
[8] En contraste con la definición original de Antoy [1992], en lo que sigue utilizaremos la siguiente definición "declarativa" de Antoy [1997] ya que resulta más apropiada para demostrar los resultados y propiedades de las transformaciones basadas en *narrowing* necesario que estudiaremos más adelante.

Propiedad de inducción: Dado $\pi \in \mathcal{P}\backslash S$, existe una posición o de $pattern(\mathcal{P})$ con $pattern(\mathcal{P})|_o \in \mathcal{X}$ (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 como padre a π, $\pi_i = \pi[c_i(\overline{x_{n_i}})]_o$ (donde $\overline{x_{n_i}}$ son variables nuevas distintas) para todo $1 \leq i \leq n$.

Si $\mathcal{R}$ es un programa o sistema de reescritura de términos ortogonal y f/n es un símbolo de función definido, decimos que $\mathcal{P}$ es un *árbol definicional de f* si $pattern(\mathcal{P}) = f(\overline{x_n})$ para una serie de variables distintas $\overline{x_n}$, siendo las hojas de $\mathcal{P}$ todas y cada una de las variantes de las partes izquierdas de las reglas de $\mathcal{R}$ que definen f. Debido a la ortogonalidad de $\mathcal{R}$, podemos asignar a cada hoja una sola regla que define f. Una función definida se llama *inductivamente secuencial* si tiene un árbol definicional asociado, lo que intuitivamente se corresponde con una definición "por casos" de la misma. Un sistema de reescritura $\mathcal{R}$ es *inductivamente secuencial* si todas sus funciones definidas son inductivamente secuenciales. Un SRT inductivamente secuencial puede verse como un conjunto de árboles definicionales, definiendo cada uno de ellos un símbolo de función distinto. Como en general pueden existir más de un árbol definicional para una misma función inductivamente secuencial, en lo que sigue asumimos que existe un árbol definicional fijo para cada función definida. Alternativamente, tal y como se demuestra en [Hanus *et al.*, 1998], un sistema inductivamente secuencial puede verse como un sistema "fuertemente secuencial" basado en constructores, según la noción de *strongly sequential system* de Huet y Lévy [1992]. De forma parecida, la noción de árbol definicional mantiene una estrecha correspondencia con la de *matching dag* de Huet y Lévy [1992] y ésta, a su vez, con la de *index tree* de Strandh [1989] (ver [Hanus *et al.*, 1998] y [Durand, 1994] para un estudio detallado de la relación precisa entre estos conceptos).

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

Ejemplo 9 El siguiente gráfico muestra un árbol definicional para la función "$\leq$" del Ejemplo 7:



A su vez, un árbol definicional para la función "+" del mismo Ejemplo 7 puede ilustrarse como sigue:

$$\boxed{\mathtt{X}} + \mathtt{Y}$$

$$\mathtt{0} + \mathtt{Y} \to \mathtt{Y} \qquad\qquad \mathtt{s(X')} + \mathtt{Y} \to \mathtt{s(X'} + \mathtt{Y)}$$

La siguiente proposición muestra que cualquier función definida por una sola regla es siempre inductivamente secuencial.

Proposición 2.2.11 [Alpuente *et al.*, 1999f] *Si $f(\overline{t_n})$ es un patrón lineal, entonces existe un árbol definicional para el conjunto $\{f(\overline{t_n})\}$ con patrón $f(\overline{x_n})$.*

Para definir el cálculo de *narrowing* necesario asumimos que t es un término encabezado por un símbolo definido y que $\mathcal{P}$ es un árbol definicional con patrón $pattern(\mathcal{P}) = \pi$ tal que $\pi \leq t$ [9]. Definimos una función λ de términos y árboles definicionales en conjuntos de tuplas (posición, regla, sustitución) como la menor relación que satisface las siguientes propiedades. Consideramos dos casos para $\mathcal{P}$ [10]:

1. Si π es una hoja, i.e., $\mathcal{P} = \{\pi\}$, y $\pi \to r$ es una variante de una regla del programa, entonces

$$\lambda(t, \mathcal{P}) = \{(\Lambda, \pi \to r, \epsilon)\} \ .$$

2. Si π es una rama, consideramos la posición inductiva o de π y algún hijo $\pi_i = \pi[c_i(\overline{x_n})]_o \in \mathcal{P}$. 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 tenemos los siguientes casos para el subtérmino $t|_o$:

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

En términos informales, *narrowing* necesario aplica una regla si es posible (caso 1), o busca el subtérmino correspondiente a la posición inductiva de la rama (caso 2): si es una variable, ésta se instancia al constructor de algún hijo; si ya es un constructor,

[9] Cuando t está encabezado por un símbolo constructor, la estrategia se generaliza de forma sencilla, como se indica en [Antoy *et al.*, 1994].

[10] Esta descripción dada en [Alpuente *et al.*, 1999f] es ligeramente diferente a la mostrada por Antoy *et al.* [1994] pero genera el mismo conjunto de pasos de *narrowing* necesario.

procedemos con el hijo correspondiente; si es una función, entonces la evaluamos recursivamente aplicando *narrowing* necesario. De esta forma, la nueva estrategia difiere de los lenguajes funcionales perezosos sólo en la (posible) instanciación de las variables libres.

Obsérvese que durante la computación de λ componemos la sustitución calculada en cada paso recursivo (que puede ser la identidad) con la sustitución obtenida en pasos anteriores. Así, cada paso de *narrowing* necesario puede representarse como $(p, R, \varphi_1 \cdots \varphi_k)$, donde cada φ_j es o bien la sustitución identidad o bien el reemplazamiento (computado en cada paso recursivo) de una sola variable. A esta notación se le llama *representación canónica* de un paso de *narrowing* necesario. Al igual que en los procedimientos de demostración de la programación lógica, asumimos que los árboles definicionales siempre contienen variables nuevas cuando se usan en pasos sucesivos de *narrowing*. Esto implica que todas las sustituciones computadas son idempotentes (asumiremos implícitamente esta propiedad en lo que sigue).

Definición 2.2.12 (*narrowing* necesario $\overset{\text{NN}}{\leadsto}$) La relación de narrowing necesario $\overset{\text{NN}}{\leadsto}$ se define como una instancia de la relación de narrowing genérico con estrategia donde la estrategia $\varphi^{NN}(t) = \lambda(t, \mathcal{P})$, siendo t un término encabezado por un símbolo de función definido con árbol definicional $\mathcal{P}$. Las computaciones se realizan usando las reglas del programa extendido $\mathcal{R}_+ = (\mathcal{R} \cup \textsc{StrEq})$.

De esta forma tenemos que para todo $(p, R, \sigma) \in \lambda(t, \mathcal{P})$, $t \overset{\text{NN}}{\leadsto}_{p,R,\sigma} t'$ es un paso de *narrowing* necesario. Decimos, además, que este paso es *determinista* si $\lambda(t, \mathcal{P})$ contiene exactamente un solo elemento. El siguiente ejemplo ilustra la definición anterior, al tiempo que muestra algunas de las ventajas de las que disfruta la relación de *narrowing* necesario frente al *narrowing* perezoso presentado en la sección anterior.

Ejemplo 10 Consideremos de nuevo los árboles definicionales del Ejemplo 9. La función λ computa el siguiente conjunto para el término inicial $\mathtt{X} \leq \mathtt{X} + \mathtt{X}$:

$$\{(\Lambda, 0 \leq \mathtt{N} \to \mathtt{true}, \{\mathtt{X} \mapsto 0\}),\ (2, \mathtt{s(M)} + \mathtt{N} \to \mathtt{s(M+N)}, \{\mathtt{X} \mapsto \mathtt{s(M)}\})\}$$

Lo que corresponde a los siguientes pasos de *narrowing* necesario:

$$\begin{array}{llll} \underline{\mathtt{X} \leq \mathtt{X} + \mathtt{X}} & \overset{\text{NN}}{\leadsto}_{\{\mathtt{X}\mapsto 0\}} & \mathtt{true} & (*) \\ \mathtt{X} \leq \underline{\mathtt{X} + \mathtt{X}} & \overset{\text{NN}}{\leadsto}_{\{\mathtt{X}\mapsto \mathtt{s(M)}\}} & \mathtt{s(M)} \leq \mathtt{s(M + s(M))} & \end{array}$$

Obsérvese que la estrategia de *narrowing* perezoso generaría un paso más de la forma:

$$\mathtt{X} \leq \underline{\mathtt{X} + \mathtt{X}} \quad \overset{\text{NP}}{\leadsto}_{\{2, 0+\mathtt{N}\to 0, \{\mathtt{X}\mapsto 0\}\}} \quad 0 \leq 0$$

Este paso (perezoso, pero no necesario) es obviamente redundante, ya que conduce a una derivación de éxito (en dos pasos) en la que se repite la solución ya obtenida tras el paso $(*)$.

Los siguientes resultados muestran algunas propiedades importantes del *narrowing* necesario. En primer lugar, observamos que cada sustitución computada en un paso de *narrowing* necesario instancia solamente variables del término inicial.

Proposición 2.2.13 [Alpuente *et al.*, 1999f] *Si* $(p, R, \varphi_1 \cdots \varphi_k) \in \lambda(t, \mathcal{P})$ *es un paso de narrowing necesario, entonces, para* $i = 1, \ldots, k$, $\varphi_i = \epsilon$ *o* $\varphi_i = \{x \mapsto c(\overline{x_n})\}$ *(donde* $\overline{x_n}$ *son variables diferentes) con* $x \in \mathcal{V}ar(t\varphi_1 \cdots \varphi_{i-1})$.

El siguiente lema muestra que, para pasos de *narrowing* necesario diferentes (que computan sustituciones diferentes), siempre existe una variable que es instanciada a constructores diferentes:

Lema 2.2.14 [Alpuente *et al.*, 1999f] *Sea* t *un término encabezado por un símbolo de función definido,* $\mathcal{P}$ *un árbol definicional con* $pattern(\mathcal{P}) \leq t$ *y* $(p, R, \varphi_1 \cdots \varphi_k)$, $(p', R', \varphi'_1 \cdots \varphi'_{k'}) \in \lambda(t, \mathcal{P})$, $k \leq k'$. *Entonces, para todo* $i \in \{1, \ldots, k\}$,

- *o bien* $\varphi_1 \cdots \varphi_i = \varphi'_1 \cdots \varphi'_i$, *o bien*
- *existe algún* $j < i$ *tal que*
  1. $\varphi_1 \cdots \varphi_j = \varphi'_1 \cdots \varphi'_j$, *y*
  2. $\varphi_{j+1} = \{x \mapsto c(\cdots)\}$ *y* $\varphi'_{j+1} = \{x \mapsto c'(\cdots)\}$ *con* $c \neq c'$.

Para programas inductivamente secuenciales, *narrowing* necesario es correcto y completo con respecto a ecuaciones estrictas y sustituciones constructoras como soluciones. Además, *narrowing* necesario nunca calcula soluciones redundantes, es decir, todas las respuestas computadas son *independientes*. Dos sustituciones θ y σ son independientes sobre un conjunto de variables V sii existe alguna variable $x \in V$ tal que $x\theta$ y $x\sigma$ no son unificables.

Teorema 2.2.15 (corrección, completitud y minimalidad [Antoy *et al.*, 1994]**)** *Sea* $\mathcal{R}$ *un programa inductivamente secuencial y* e *una ecuación.*

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

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

3. *(Minimalidad) Si* $e \overset{\text{NN}}{\leadsto}{}^*_\sigma$ *true y* $e \overset{\text{NN}}{\leadsto}{}^*_{\sigma'}$ *true son dos derivaciones de narrowing necesario distintas, entonces* σ *y* σ' *son independientes sobre* $\mathcal{V}ar(e)$.

Para finalizar este apartado, presentamos a continuación una última e interesante propiedad de la estrategia de *narrowing* necesario. En general, una ventaja importante de los lenguajes lógico-funcionales frente a los lenguajes lógicos puros es su mejor comportamiento operacional al evitar pasos de computación no deterministas. Una razón por la que se cumple este hecho está en que las estrategias (necesarias) dirigidas por la demanda pueden evitar la evaluación de expresiones potencialmente indeterministas [Alpuente *et al.*, 1999f]. Por ejemplo, consideremos las reglas del Ejemplo 7 y el término $\mathtt{0} \leq \mathtt{X} + \mathtt{X}$. *Narrowing* necesario evalúa este término a `true` mediante un solo paso determinista. En el programa lógico equivalente, este mismo término anidado debe ser *aplanado* y expresado como una conjunción de dos llamadas a predicado como $+(\mathtt{X},\mathtt{X},\mathtt{Z}) \wedge \leq(\mathtt{0},\mathtt{Z},\mathtt{B})$, lo que produce una evaluación indeterminista debido a la llamada al predicado $+(\mathtt{X},\mathtt{X},\mathtt{Z})$. Otro hecho que muestra el mejor comportamiento operacional de los lenguajes lógico-funcionales es la capacidad de ciertas estrategias de evaluación, como es el caso de *narrowing* necesario, para evaluar términos básicos de una forma completamente determinista, lo cual es muy importante para asegurar una implementación eficiente de las evaluaciones funcionales puras. Esta propiedad, que es obvia en la definición del *narrowing* necesario, se formaliza en la siguiente proposición. Con este fin, decimos que un término t es evaluable *deterministamente* (con respecto a *narrowing* necesario) si cada paso en una derivación de *narrowing* para t es determinista. Además, un término t es normalizable *deterministamente* a un término constructor c (con respecto a *narrowing* necesario) si t es evaluable deterministamente y existe una derivación de *narrowing* necesario $t \overset{\text{NN}}{\leadsto}{}^*_\epsilon\, c$ (i.e., c es la forma normal de t).

Proposición 2.2.16 [Alpuente *et al.*, 1999f] *Sea $\mathcal{R}$ un programa inductivamente secuencial y t un término.*

1. *Si $t \overset{\text{NN}}{\leadsto}{}^*_\epsilon\, c$ es una derivación de narrowing necesario, entonces t es normalizable deterministamente a c.*

2. *Si t es básico, entonces t es evaluable deterministamente.*

2.3 Programas condicionales

Finalizamos este capítulo extendiendo los conceptos presentados en secciones anteriores al caso de los programas condicionales haciendo uso del formalismo de los sistemas de reescritura de términos condicionales.

Un *sistema de reescritura de términos condicional* (SRTC en lo sucesivo) es un par $(\Sigma, \mathcal{R})$, donde $\mathcal{R}$ es un conjunto finito de (esquemas de) reglas de reescritura (o reglas de reducción) de la forma $(l \to r \Leftarrow C)$, donde $l, r \in \mathcal{T}(\Sigma, \mathcal{X})$ y $l \notin \mathcal{X}$.

Cuando $\mathcal{V}ar(r) \cup \mathcal{V}ar(C) \subseteq \mathcal{V}ar(l)$ se dice que los programas son de tipo 1 o 1-CTRS. Por su parte, los 2-CTRS satisfacen $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ y los 3-CTRS cumplen $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(C)$. La condición C es una secuencia (posiblemente vacía) de ecuaciones $e_1, \ldots, e_n$, $n \geq 0$. Cuando existen variables en r o C que no aparecen en l, éstas reciben el nombre de *variables extra* (ver [Middeldorp y Hamoen, 1994]). Cuando una regla de reescritura no posee condición, escribimos simplemente $(l \to r)$ siguiendo el convenio visto en el caso de los programas incondicionales. Cuando las condiciones de todas las reglas de $\mathcal{R}$ son vacías, tenemos que $(\Sigma, \mathcal{R})$ es un sistema de reescritura de términos incondicional (SRT).

Las nociones de objetivo ecuacional, posición, etc., se pueden definir ahora directamente sobre secuencias de ecuaciones de forma natural, donde, por ejemplo, el conjunto de posiciones de una secuencia de ecuaciones $g = (e_1, \ldots, e_n)$ se define como: $\mathcal{P}os(g) = \{i.u \mid u \in \mathcal{P}os(e_i),\ i = 1, \ldots, n\}$. Usaremos $\top$ como notación genérica para una secuencia de la forma $true, \ldots, true$. A continuación, pasamos a extender las nociones de reescritura y *narrowing* a este tipo de programas.

Un término s se *reescribe* condicionalmente a un término t, y lo denotamos por $s \to_{\mathcal{R}} t$, si existe una regla $(l \to r \Leftarrow s_1 = t_1, \ldots, s_n = t_n) \in \mathcal{R}$, una posición $p \in \mathcal{P}os(s)$, una sustitución σ tal que $s|_p = l\sigma$, $t = s[r\sigma]_p$ y, para todo $i = 1, \ldots, n$, existe un término q_i tal que $s_i\sigma \to^*_{\mathcal{R}} q_i$ y $t_i\sigma \to^*_{\mathcal{R}} q_i$, donde $\to^+_{\mathcal{R}}$ y $\to^*_{\mathcal{R}}$ denotan respectivamente el cierre transitivo y el cierre reflexivo y transitivo de la relación $\to_{\mathcal{R}}$. Una clase importante de sistemas de reescritura condicionales que usaremos en esta tesis son los llamados *sistemas decrecientes*. Decimos que un SRTC es decreciente si existe una extensión bien fundada $\succ$ de la relación de reescritura $\to_{\mathcal{R}}$ con las siguientes propiedades:

1. $\succ$ cumple la *propiedad de subtérmino*, i.e., $t \succ t|_p$ para todo $p \in \mathcal{P}os(t) - \{\Lambda\}$, y

2. si $(l \to r \Leftarrow C) \in \mathcal{R}$ y σ es una sustitución, entonces $l\sigma \succ r\sigma$ y, para todo $s = t \in C$, $l\sigma \succ s\sigma$ y $l\sigma \succ t\sigma$.

Por su parte, la extensión del cálculo genérico de *narrowing* (con estrategia) al caso condicional se define de la forma obvia. Lo que sigue es la instancia correspondiente al *narrowing* condicional ordinario.

Definición 2.3.1 (narrowing condicional $\leadsto$) Sea $\mathcal{R}$ un SRTC y g un objetivo ecuacional. Definimos la relación de narrowing condicional $\leadsto$ como la menor relación que satisface:

$$\frac{p \in \mathcal{FP}os(g) \ \wedge\ R = (l \to r \Leftarrow C) \ll \mathcal{R}_+ \ \wedge\ \sigma = mgu(\{g|_p = l\})}{g \ \leadsto_{p,R,\sigma} \ (C, g[r]_p)\sigma}$$

En este caso, una *derivación de éxito* para g en $\mathcal{R}$ es una derivación de la forma $g \leadsto_\theta^* \top$. El resto de nociones sobre derivaciones de *narrowing* se siguen de forma natural a partir del caso incondicional. *Narrowing* condicional es completo para 1-CTRS's canónicos, para 1-CTRS's confluentes respecto a soluciones normalizadas y para 2-CTRS's y 3-CTRS's terminantes y confluentes por niveles (ver [Middeldorp y Hamoen, 1994]). Se debe notar que, en el marco de los programas incondicionales, todavía es posible tratar con reglas con condiciones y objetivos formados por conjunciones de ecuaciones, pero en este caso las condiciones en las reglas deben tratarse usando las funciones predefinidas $\wedge$ y $\Rightarrow$ (a las que pueden añadirse también las construcciones `if_then_else` y `case_of`) que se reducen usando reglas de definición estándar [Hanus (ed.), 1999; Moreno-Navarro y Rodríguez-Artalejo, 1992], i.e.:

$$\begin{array}{ll} \mathit{true} \wedge \mathit{true} & \to \mathit{true} \\ \mathit{true} \Rightarrow x & \to x \end{array}$$

Por ejemplo, en el caso de programas ortogonales o inductivamente secuenciales (con funciones no terminantes que se evalúan de forma perezosa) es habitual el considerar cada regla condicional de la forma $l \to r \Leftarrow C$ como otra incondicional equivalente $l \to (C \Rightarrow r)$, donde la secuencia de ecuaciones (estrictas) $e_1, \ldots, e_n$ en C se expresa como un solo término con conjunciones de la forma $e_1 \wedge \ldots \wedge e_n$.

Capítulo 3

Desplegado

El desplegado de un programa declarativo se entiende en general como el proceso en virtud del cual una regla (o cláusula) del mismo es sustituida por un nuevo conjunto de reglas obtenidas a partir de la original, sobre la que se realiza uno (o varios) pasos de ejecución *simbólica*. Este tipo de ejecución se refiere a un tipo de evaluación donde, en ausencia de datos reales, se ejecutan llamadas "simbólicas". Más concretamente, la idea básica subyacente a toda operación de desplegado consiste en efectuar esta clase de evaluación simbólica ya no sobre un objetivo, sino sobre el cuerpo (o la parte derecha de la cabeza) de una regla del programa. Dependiendo del tipo de paradigma declarativo que estemos considerando (lógico puro, funcional puro o lógico-funcional) se trabaja con un mecanismo operacional diferente (resolución SLD, reescritura o *narrowing*, respectivamente) sobre el que basar (de alguna manera) la computación simbólica e implementar el desplegado.

En este capítulo nos centraremos en la regla de desplegado sobre programas lógico-funcionales, estudiando diversas variantes basadas en diferentes refinamientos de *narrowing*, así como las principales aplicaciones de las mismas. Comenzamos haciendo un breve recorrido histórico que explora los antecedentes del desplegado en contextos funcionales y lógicos puros y esbozando el problema de la adaptación de esta regla a un contexto integrado.

3.1 Motivación y antecedentes

El primer trabajo sobre transformación de programas basado en las reglas de plegado y desplegado es original de Burstall y Darlington [1977]. Se trata de un trabajo pionero en esta línea, referido a programas funcionales puros definidos como un conjunto de ecuaciones recursivas. El desplegado se define en [Burstall y Darlington, 1977] textualmente como sigue:

> *Si $E = E'$ y $F = F'$ son ecuaciones y existe alguna ocurrencia de F' que es instancia de E, entonces se reemplaza por la instancia correspondiente de E', obteniendo F'' y añadiendo la ecuación $F = F''$.*

Cabe destacar que este proceso puede verse como un paso de reescritura sobre la parte derecha de la ecuación a desplegar. En esta primera aproximación no se hace explícito el hecho de que la regla desplegada debe retirarse del conjunto resultante de ecuaciones y no se dan tampoco resultados de corrección ni completitud para la transformación.

En trabajos posteriores sobre transformación de programas funcionales [Scherlis, 1981; Kott, 1985; Zhu, 1994; Sands, 1996], la operación de desplegado se formula prácticamente sin variaciones, destacando siempre el hecho de que esta regla se basa en la reescritura y sólo involucra emparejamiento. Puesto que las llamadas a función en el cuerpo de las reglas del programa pueden contener variables, las transformaciones de Burstall y Darlington hacen uso de una operación previa de *instanciación*. De esta forma, si una llamada a función no empareja con ninguna de las funciones definidas en el programa, sus variables pueden instanciarse para permitir así su desplegado (ver, por ejemplo, la Sección 17.3 de [Jones *et al.*, 1993] y, en particular, el ejemplo de desplegado de la función de Ackermann). Sin embargo, hemos encontrado en la literatura pocas técnicas de transformación automáticas basadas en las transformaciones de instanciación + desplegado, las cuales se recogen en [Alpuente *et al.*, 1998a].

Por todo ello, se puede considerar que el proceso de transformación básico en los programas funcionales consta realmente de instanciación + desplegado. En particular, la ejecución de una llamada a función conteniendo variables (valores simbólicos), recibe el nombre de ejecución simbólica. Así, en el caso de los programas funcionales el mecanismo de ejecución (desplegado) y el mecanismo de ejecución simbólica (instanciación + desplegado) no coinciden, lo que significa que hay que desarrollar técnicas *ad-hoc* para los métodos de transformación, a diferencia de lo que ocurre en los contextos lógicos puros y lógico-funcionales, como veremos posteriormente.

La aproximación más reciente a las técnicas de transformación de programas funcionales es la presentada por Sands [1995, 1996]. En general, las transformaciones de plegado/desplegado, definidas por Burstall y Darlington [1977], no garantizan ni la mejora en eficiencia ni la corrección total de la transformación. En [Sands, 1995] se introduce una condición (semántica) para la corrección total de las transformaciones sobre programas funcionales perezosos de orden superior. El principal resultado técnico consiste en que, si los pasos locales de transformación se realizan siguiendo un cierto criterio, entonces se puede asegurar la corrección total de la transformación. En [Sands, 1996], se muestra cómo el resultado anterior permite demostrar la corrección total de algunas de las principales técnicas automáticas de transformación de programas, incluyendo la deforestación de Wadler [1990] y la supercompilación de

Turchin [1986a].

Sin embargo, en todas las propuestas de desplegado de programas funcionales aparece un grave inconveniente. Concretamente, la instanciación no restringida de ciertas ecuaciones para posibilitar su posterior desplegado, se muestra problemática a la hora de preservar la equivalencia con el programa original. De hecho, este escollo nunca se resuelve de forma transparente en los contextos funcionales. El propio Sands no considera explícitamente la regla de instanciación en [Sands, 1996] (solamente muestra que esta regla puede simularse en términos de leyes de distribución sobre expresiones `case_of`), a pesar de que su propuesta puede considerarse como la más rigurosa y moderna en el área.

En resumen, la técnica de desplegado sobre programas funcionales consiste, esencialmente, en la sustitución (dentro de una ecuación que, posiblemente, ha sido previamente instanciada) de una llamada a función por su respectiva definición, aplicando la correspondiente sustitución. Ya que en el caso de los programas funcionales las técnicas de plegado/desplegado sólo involucran emparejamiento, el proceso previo de instanciación suele ser necesario para posibilitar el desplegado de una regla en un programa.

La primera adaptación de las reglas de plegado/desplegado al caso de los programas lógicos se debe a Tamaki y Sato, quienes presentaron sus resultados a mediados de la década de los 80. En [Tamaki y Sato, 1984] se extienden las ideas originales de Burstall y Darlington [1977] reemplazando el emparejamiento por la unificación en las reglas de transformación. El desplegado de un programa lógico consiste, por tanto, en aplicar un paso de resolución a un subobjetivo en el cuerpo de una cláusula de todas las formas posibles. Gracias al mecanismo de unificación, el desplegado de los programas lógicos permite propagar información sintáctica (como es la estructura de los términos) y no sólo valores constantes (como en el caso del desplegado de los programas funcionales). De esta forma, se obtiene un mecanismo más potente que, además, no requiere la presencia de una regla de instanciación independiente, ya que dicho proceso va implícito en la propia regla de desplegado basada en el mecanismo de resolución.

Como se indica en [Pettorossi y Proietti, 1996b], el desplegado de los programas lógicos no sólo se diferencia del de los programas funcionales en el reemplazamiento del emparejamiento de patrones por la unificación sintáctica, sino también en la cantidad de reglas desplegadas que genera como consecuencia de ellos. Ya que en un programa funcional es frecuente considerar funciones definidas mediante reglas cuyas partes izquierdas no solapan, el desplegado (previa instanciación) de una de ellas genera una y solamente una regla nueva, a diferencia del desplegado de una cláusula en un programa lógico que, en general, devuelve un conjunto de cláusulas.

Para el caso de los programas lógicos se han propuesto en la literatura diversas aproximaciones a la regla de desplegado (ver por ejemplo los diferentes trabajos de Proietti y Petorossi para un estudio extenso y recopilatorio del tema [Proietti y Pettorossi, 1993; Pettorossi y Proietti, 1994, 1996b, 1998]). Todas ellas son básicamente similares a la original de Tamaki y Sato [1984], aunque en ocasiones se permite que la regla a desplegar y las reglas desplegantes pertenezcan a programas distintos [Pettorossi y Proietti, 1994], o que puedan ser varias las reglas desplegadas simultáneamente [Levi y Mancarella, 1988; Bossi *et al.*, 1994], etc.

Cuando consideramos programas lógico-funcionales como sistemas de reescritura y una semántica operacional basada en *narrowing* como método de resolución de ecuaciones, no resulta inmediato decidir cuál puede ser la noción de *desplegado* más apropiada. Siguiendo una aproximación similar a la de Alpuente *et al.* [1998a], donde se describe un marco genérico para la evaluación parcial de programas lógico-funcionales, podemos dar una primera aproximación al problema como sigue. Por simplicidad, consideramos de momento sistemas de reescritura incondicionales. Dado un programa $\mathcal{R}$ y una regla del mismo $R = (l \to r) \in \mathcal{R}$, el desplegado de $\mathcal{R}$ con respecto a la regla R consiste en:

1. Construir las derivaciones usando (alguna variante de) *narrowing* en un paso para el objetivo r, obteniendo $\{r \leadsto_{\theta_i} r_i\}$

2. Formar el conjunto de reglas de reescritura $\mathcal{S} = \{l\theta_i \to r_i\}$.

3. Obtener el programa transformado $\mathcal{R}'$ a partir de $\mathcal{R}$ eliminado de éste último la regla desplegada R, y añadiendo todas las pertenecientes a $\mathcal{S}$.

Ya que el mecanismo de ejecución de los programas lógico-funcionales (*narrowing*) en el que hemos basado esta primera definición de desplegado, utiliza la unificación para el paso de parámetros en las llamadas a función, se consigue de forma automática el efecto de la regla de instanciación de los programas funcionales (de forma similar a lo que ocurre con los programas lógicos). El desplegado de programas lógico-funcionales basado en el mecanismo operacional del *narrowing* aporta así una visión unificada de los mecanismos de ejecución y transformación para este tipo de programas, al tiempo que, debido a la posibilidad de explotar su componente funcional, el desplegado de un programa lógico-funcional puede resultar más efectivo que el de un programa lógico equivalente (al poder utilizar, por ejemplo, el anidamiento de funciones que permite la sintaxis funcional). De esta forma podemos decir que el desplegado de programas lógico-funcionales basado en (alguna variante del) *narrowing* subsume (y mejora) los procesos de instanciación y desplegado por reescritura de los programas funcionales puros, y también el desplegado basado en resolución SLD de los programas lógicos puros, como formalizaremos más adelante.

Como antecedentes de las técnicas descritas podemos mencionar los siguientes. En [Levi y Sirovich, 1975] se define un procedimiento de evaluación parcial basado en desplegado para el lenguaje funcional TEL, usando un mecanismo de ejecución simbólica basado en unificación que puede llegar a entenderse como una forma de *narrowing* perezoso. [Darlington y Pull, 1988] muestran cómo la unificación permite integrar los pasos de desplegado e instanciación, obteniendo así la habilidad del *narrowing* para tratar con variables lógicas. Sin embargo, en ninguno de estos trabajos se han abordado cuestiones de equivalencia semántica. Otro ejemplo del uso de unificación en la técnica de desplegado se puede encontrar en [Dershowitz y Reddy, 1993], donde se utiliza para formular una aproximación a la síntesis de programas funcionales.

Los trabajos sobre supercompilación de Turchin [1986a] son, de entre la extensa literatura sobre transformación de programas, los que utilizan un tipo de desplegado más cercano a nuestra propuesta. La supercompilación (*supervised compilation*) es una técnica de transformación para programas funcionales que consta de tres elementos básicos: *driving*, generalización y generación de programas residuales. La supercompilación no especializa el programa original, sino que construye un programa nuevo para la (especialización de la) llamada a función inicial, usando el mecanismo de *driving* [Glück y Sørensen, 1994]. El procedimiento de *driving* puede considerarse un mecanismo de desplegado de funciones basado en unificación y reducción. Concretamente, hace uso de un tipo de maquinaria de evaluación similar a *narrowing* (perezoso) para construir "árboles de estados" (posiblemente infinitos) para un programa de entrada y una llamada dada. En los trabajos de Turchin sobre supercompilación se usa la siguiente terminología [Romanenko, 1991]: "emparejamiento generalizado" para la unificación, "contracciones" para las sustituciones de *narrowing* y, muy a menudo en los textos, "driving" para el propio mecanismo de *narrowing*, es decir, la operación de instanciación de una llamada a función para todos los casos posibles, seguida por el desplegado de las distintas ramas. Gracias al procedimiento de *driving*, el supercompilador puede conseguir la misma cantidad de propagación de información (basada en unificación) y especialización del programa que en la evaluación parcial de los programas lógicos [Glück y Sørensen, 1994].

Los trabajos de Turchin describen el supercompilador para Refal (*Recursive Function Algorithmic Language* [Turchin, 1989]), un lenguaje funcional basado en emparejamiento con una noción de patrones poco usual. La semántica de Refal se da en términos de un intérprete de reescritura (con una estrategia de evaluación impaciente), mientras que el supercompilador (en el que se encuentra incorporado el procedimiento de *driving*) posee una estrategia de evaluación perezosa. La supercompilación subsume, entre otros, el procedimiento de deforestación de Wadler [1988], la evaluación parcial y otras transformaciones estándar de programas funcionales [Sørensen *et al.*, 1994]. Por ejemplo, la supercompilación es capaz de realizar cierto tipo de demostración de teoremas, síntesis e inversión de programas (ver [Sørensen, 1994] para una exposición detallada). La supercompilación puede mejorar un programa incluso si todos los argumentos de las llamadas a función son variables, eliminando las redundancias debidas a los anidamientos funcionales o a las variables repetidas. En [Jones, 1994], se formaliza la metodología de Turchin basada en *driving* sobre unos sólidos fundamentos semánticos, que no están ligados a ningún lenguaje de programación o estructura de datos particular.

El proceso de *driving* puede ser infinito y, en general, no preserva la semántica del programa, ya que puede extender el dominio de las funciones (ver, e.g., [Glück y Sørensen, 1994; Jones *et al.*, 1993; Sørensen *et al.*, 1994]). En [Sørensen y Glück, 1995; Turchin, 1988] se estudian distintas técnicas para asegurar la terminación del proceso de supercompilación. La idea de Turchin [1988] consiste en supervisar la construcción del árbol y, bajo determinadas condiciones, realizar una "vuelta atrás", i.e., plegar las configuraciones a uno de los estados previos, obteniendo de esta forma un grafo finito. A menudo, es necesario disponer de una operación de *generalización* para que el plegado a una configuración anterior sea posible. En [Sørensen y Glück, 1995] se estudia la terminación de la supercompilación *positiva*, una versión simplificada del algoritmo de Turchin en la que no se considera la propagación de información negativa en el proceso de especialización.

En el contexto lógico-funcional, no existen en la literatura muchos trabajos relacionados con el desplegado de este tipo de programas. En [Alpuente *et al.*, 1997c,d, 1999e,d] presentamos diversas variantes de desplegado (y otras reglas de transformación) basadas en *narrowing*, donde el propio método de evaluación estándar de los lenguajes integrados se revela muy adecuado para implementar la regla de desplegado. De la misma forma que este mecanismo operacional admite diferentes refinamientos, se pueden plantear diferentes modelos de desplegado utilizando diferentes variantes del *narrowing* para definir la regla de transformación. Esto es, es posible dar una definición genérica de desplegado de programas lógico-funcionales de forma paramétrica con respecto a la estrategia de *narrowing* que se use en cada caso. Cada instancia exige una serie de condiciones de aplicabilidad y disfruta de una serie de propiedades que iremos revisando en los siguientes apartados.

3.2 Desplegado basado en *narrowing* condicional

Antes de dar nuestra definición genérica del concepto de desplegado de un programa, es interesante introducir la noción más simple de desplegado de una regla dentro de un programa. Al tratarse del caso condicional, en la definición es necesario considerar que el término a desplegar dentro de una regla pueda pertenecer a la parte derecha de su cabeza (r) o bien a las condiciones (C). Esto se resuelve fácilmente considerando el objetivo ecuacional $g = (C, r = y)$, donde y es una variable nueva.

Definición 3.2.1 (desplegado de una regla dentro de un programa)
Sea $\mathcal{R}$ un SRTC y $R = (l \to r \Leftarrow C) \ll \mathcal{R}$ una regla (renombrada) del mismo. Sea $\{g \stackrel{\theta_i}{\leadsto} (C'_i, r'_i = y)\}_{i=1}^{n}$ el conjunto de todas las derivaciones de *narrowing* de un solo paso sobre el objetivo $g = (C, r = y)$ en $\mathcal{R}$. El resultado de desplegar la regla R en el programa $\mathcal{R}$ es el programa $Unf_{\mathcal{R}}(R)$ definido como sigue:

$$Unf_{\mathcal{R}}(R) = \{(l\theta_i \to r'_i \Leftarrow C'_i) \ \mid \ i = 1 \ldots n\} \ .$$

Obsérvese que el desplegado de una regla nunca contiene la regla original que se despliega, ya que en nuestra definición se prohiben expresamente las derivaciones de *narrowing* de un solo paso de la forma $(C, r = y) \stackrel{\theta_i}{\leadsto} true$. Nótese que estas derivaciones únicamente pueden aparecer cuando el objetivo $(C, r = y)$ unifique sintácticamente. Aunque este hecho tendrá repercusiones a la hora de plantear las condiciones de aplicabilidad bajo las cuales se puede garantizar la corrección y la completitud de la transformación, la razón que nos lleva a evitar estas derivaciones en nuestra definición se corresponde con las idea más intuitiva de desplegado que se tiene en mente al inspirarnos en el desplegado de programas funcionales o lógicos puros (concretamente con el hecho de que la regla desplegada, o una versión comprimida de la misma, nunca se repite en el programa transformado). En la sección 3.6 nos detendremos más en este hecho y daremos una definición generalizada del desplegado que permita replicar en el programa transformado una versión comprimida de la regla desplegada.

Basándonos en la definición anterior, podemos definir ahora la noción de desplegado de un programa con respecto a una regla del mismo.

Definición 3.2.2 (desplegado de un programa con respecto a una regla)
Sea $\mathcal{R}$ un SRTC y $R = (l \to r \Leftarrow C) \ll \mathcal{R}$ una regla (renombrada) del mismo. El desplegado del programa $\mathcal{R}$ con respecto a la regla R es el programa $\mathcal{R}'$ definido como sigue:

$$\mathcal{R}' = (\mathcal{R} - \{R\}) \cup Unf_{\mathcal{R}}(R) \ .$$

El siguiente ejemplo ilustra nuestra definición de desplegado al tiempo que muestra la situación en que éste no es practicable.

Ejemplo 11 Consideremos el siguiente programa $\mathcal{R}$:

$$\begin{array}{rcll}
\mathtt{nat(0)} & \to & \mathtt{true} & (R_1)\\
\mathtt{nat(s(X))} & \to & \mathtt{nat(X)} & (R_2)\\
\mathtt{inc(X)} & \to & \mathtt{s(X)} \Leftarrow \mathtt{nat(X)} = \mathtt{true} & (R_3)
\end{array}$$

En primer lugar, observamos que la regla R_1 no puede ser desplegada debido al hecho de que no existen derivaciones de *narrowing* condicional para el objetivo $(\mathtt{true} = \mathtt{Y})$, aparte de la que unifica sintácticamente el objetivo y que hemos prohibido explícitamente en la Definición 3.2.1. Por otra parte, al desplegar la tercera regla obtenemos $Unf_{\mathcal{R}}(R_3)$:

$$\begin{array}{rcl@{\quad}c@{\quad}l} \mathtt{inc(0)} & \rightarrow & \mathtt{s(0)} & \Leftarrow & \mathtt{true} = \mathtt{true} \\ \mathtt{inc(s(X))} & \rightarrow & \mathtt{s(s(X))} & \Leftarrow & \mathtt{nat(X)} = \mathtt{true} \end{array}$$

De esta forma el desplegado de $\mathcal{R}$ con respecto a la regla R_3 es el programa $\mathcal{R}'$:

$$\begin{array}{rcl@{\quad}c@{\quad}l} \mathtt{nat(0)} & \rightarrow & \mathtt{true} & & \\ \mathtt{nat(s(X))} & \rightarrow & \mathtt{nat(X)} & & \\ \mathtt{inc(0)} & \rightarrow & \mathtt{s(0)} & \Leftarrow & \mathtt{true} = \mathtt{true} \\ \mathtt{inc(s(X))} & \rightarrow & \mathtt{s(s(X))} & \Leftarrow & \mathtt{nat(X)} = \mathtt{true} \end{array}$$

Tal y como avanzamos en el capítulo de introducción, la operación de desplegado que acabamos de ver no preserva en general el significado computacional de los programas originales, incluso aún restringiéndonos a requerimientos tan razonables (*a priori*) como son respuestas normalizadas y programas confluentes.

A continuación investigamos las condiciones que aseguran la corrección de la transformación. Básicamente, pretendemos demostrar que la transformación es siempre correcta, pero que las propiedades de *confluencia, decrecimiento* [Bockmayr y Werner, 1995] y cierta condición de "cierre" [Alpuente *et al.*, 1998a] son necesarias para asegurar la completitud de la transformación. Con el objetivo de motivar estas condiciones pasamos a considerar primeramente algunos ejemplos simples.

Ejemplo 12 Nuestro primer ejemplo parte del siguiente programa canónico pero no decreciente $\mathcal{R}$:

$$\begin{array}{lcl@{\quad}c@{\quad}l} \mathtt{f(X)} & \rightarrow & \mathtt{g(X,X)} & & \\ \mathtt{a} & \rightarrow & \mathtt{b} & & \\ \mathtt{g(a,b)} & \rightarrow & \mathtt{c} & & \\ \mathtt{g(X,X)} & \rightarrow & \mathtt{c} & \Leftarrow & \mathtt{f(a)} = \mathtt{c} \end{array}$$

El desplegado de $\mathcal{R}$ con respecto a su primera regla es el programa transformado $\mathcal{R}'$:

$$\begin{array}{lcl@{\quad}c@{\quad}l} \mathtt{f(X)} & \rightarrow & \mathtt{c} & \Leftarrow & \mathtt{f(a)} = \mathtt{c} \\ \mathtt{a} & \rightarrow & \mathtt{b} & & \\ \mathtt{g(a,b)} & \rightarrow & \mathtt{c} & & \\ \mathtt{g(X,X)} & \rightarrow & \mathtt{c} & \Leftarrow & \mathtt{f(a)} = \mathtt{c}. \end{array}$$

Ahora la ejecución del objetivo $\mathtt{f(a)} = \mathtt{c}$ calcula la respuesta (normalizada) ϵ en $\mathcal{R}$, mientras que entra en bucle infinito en $\mathcal{R}'$.

Ejemplo 13 Consideremos ahora el siguiente SRTC confluente y decreciente $\mathcal{R}$:

$$\begin{array}{rcl@{\quad}l} \mathtt{f(0)} & \rightarrow & \mathtt{0} & (R_1) \\ \mathtt{f(c(X))} & \rightarrow & \mathtt{c(f(X))} & (R_2) \end{array}$$

Nótese que la llamada $\mathtt{f(X)}$ en R_2 no está "cubierta" por $\mathcal{R}$, i.e., no es instancia de la parte izquierda de la cabeza de ninguna regla de $\mathcal{R}$. Ahora, si desplegamos $\mathcal{R}$ con respecto a su segunda regla obtenemos el programa $\mathcal{R}'$:

$$\begin{array}{rcl} \mathtt{f(0)} & \rightarrow & \mathtt{0} \\ \mathtt{f(c(0))} & \rightarrow & \mathtt{c(0)} \\ \mathtt{f(c(c(X)))} & \rightarrow & \mathtt{c(c(f(X)))} \end{array}$$

y el objetivo $g = (\mathtt{f(c(X))} = \mathtt{Y})$ obtiene la respuesta (normalizada) $\theta = \{\mathtt{Y} \mapsto \mathtt{c(f(X))}\}$ en $\mathcal{R}$, mientras que no existe ninguna respuesta computada θ' para g en $\mathcal{R}'$ tal que $\theta' \leq \theta$ $[\mathcal{V}ar(g)]$.

Antes de caracterizar formalmente el conjunto de restricciones y condiciones de aplicabilidad (que aseguren la completitud) del desplegado, pasamos a mostrar la estrecha conexión existente entre la corrección de la evaluación parcial de programas lógico-funcionales (ver [Alpuente *et al.*, 1996b, 1998a]) y la corrección de la transformación de la operación de desplegado. Un análisis más detallado de estas conexiones se presenta en la Sección 3.6. Las siguientes definiciones, originales de Alpuente *et al.* [1998a], son necesarias para mostrar esta relación, al tiempo que nos ayudarán a perfilar las condiciones necesarias para garantizar la completitud del desplegado.

La función $terms(O)$ extrae las llamadas que aparecen en el conjunto de reglas o ecuaciones O de la siguiente forma.

Definición 3.2.3 Sea O una expresión. Definimos $terms(O)$ como sigue:

$$terms(O) = \begin{cases} \bigcup_{i=1}^{n} terms(o_i) & \text{si } O = \{o_1, \ldots, o_n\} \\ terms(\{e_1, \ldots, e_n\}) \cup \{r\} & \text{si } O = (l \rightarrow r \Leftarrow e_1, \ldots, e_n) \\ \{s, t\} & \text{si } O = (s = t) \end{cases}$$

Nótese que la definición de la función $terms(O)$ nunca considera la parte izquierda de la cabeza de una regla de programa.

Definición 3.2.4 (cierre) Sean S y T dos conjuntos finitos de términos. Decimos que T es S-cerrado si $closed(S, T)$, donde el predicado $closed$ se define inductivamente como sigue:

$$closed(S, O) \Leftrightarrow \begin{cases} true & \text{si } O = \emptyset \text{ o } O = x \in \mathcal{X} \\ closed(S, t_1) \wedge \ldots \wedge closed(S, t_n) & \text{si } O = \{t_1, \ldots, t_n\} \\ closed(S, \{\overline{t_n}\}) & \text{si } O = c(\overline{t_n}),\ c \in \mathcal{C} \\ (\exists s \in S.\ s\theta = O) \wedge closed(S, terms(\widehat{\theta})) & \text{si } O = f(\overline{t_n}),\ f \in \mathcal{F} \end{cases}$$

Decimos que un término t es S-cerrado si $closed(S, t)$, y, por extensión, decimos que un programa $\mathcal{R}$ es S-cerrado si $closed(S, terms(\mathcal{R}))$.

La noción de cierre fue introducida por primera vez en Alpuente *et al.* [1996b, 1998a] para asegurar la completitud de la evaluación parcial de programas lógico-funcionales, una técnica de especialización que, dado un programa $\mathcal{R}$ y un objetivo g, genera un programa parcialmente evaluado $\mathcal{R}'$ que produce exactamente las mismas respuestas que $\mathcal{R}$ para g y cualquier otro objetivo que satisfaga algunos requerimientos específicos, incluyendo las condición de cierre. Intuitivamente, la condición de cierre garantiza que todas las posibles llamadas que puedan surgir durante la ejecución de g están "cubiertas" por alguna regla de $\mathcal{R}'$.

Un tipo especial de cierre para programas es el L-cierre, que tendrá importantes repercusiones en los resultados de corrección del desplegado que veremos posteriormente y que se define como sigue.

Definición 3.2.5 (L-cierre) Un programa $\mathcal{R} = \{l_i \to r_i \Leftarrow C_i\}_{i=1}^n$ es *L-cerrado* si $\mathcal{R}$ es cerrado con respecto al conjunto L de términos que componen las partes izquierdas de las cabezas de todas sus reglas. Es decir, $\mathcal{R}$ es L-cerrado si $closed(L, terms(\mathcal{R}))$, donde $L = \{l_1, \ldots, l_n\}$.

A continuación pasamos a formular los resultados de corrección de la regla de desplegado que acabamos de definir. Para facilitar la demostración de que el desplegado basado en *narrowing* condicional sin restricciones es parcialmente correcto, introducimos la siguiente definición auxiliar, que puede ser vista como una instancia particular de la Definición 3.2.1.

Definición 3.2.6 (desplegado con respecto a una posición y regla fijas) Sea $\mathcal{R}$ un SRTC y $R = (l \to r \Leftarrow C) \ll \mathcal{R}$ una regla del mismo. Sea $p \in \mathcal{FP}os(R)$ una posición de la regla R y $R' \ll \mathcal{R}$. Si $(C, r = y) \rightsquigarrow_{p,R',\theta} (C'', r'' = y)$ es un paso de *narrowing* condicional en $\mathcal{R}$, entonces definimos el desplegado de la regla R con respecto a la posición p y la regla R' como sigue:

$$unfold(R, R', p) = \{(l\theta \to r'' \Leftarrow C'')\}\ .$$

Obsérvese que $Unf_{\mathcal{R}}(R) = \bigcup_{R' \in \mathcal{R}} \bigcup_{p \in \mathcal{FP}os(R)} unfold(R, R', p)$. La definición anterior, junto con el lema siguiente, nos permitirán demostrar fácilmente la corrección parcial del desplegado. En lo que sigue denotamos por $\mathcal{FP}os(R)$ el conjunto de posiciones no variables de la parte derecha de la cabeza y el cuerpo de una regla condicional R. Asimismo, consideramos la representación ecuacional de una sustitución $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ como el conjunto de ecuaciones $\widehat{\theta} = \{x_1 = t_1, \ldots, x_n = t_n\}$. Finalmente, definimos el operador conmutativo de composición paralela $\Uparrow$, que extiende la noción de mgu de dos sustituciones θ_1 y θ_2, como $\theta_1 \Uparrow \theta_2 = mgu(\widehat{\theta_1} \cup \widehat{\theta_2})$ [Palamidessi, 1990]. La siguiente proposición muestra una interesante propiedad del operador $\Uparrow$ [Lassez *et al.*, 1988; Palamidessi, 1990].

Proposición 3.2.7 *Sean θ_1 y θ_2 dos sustituciones. Entonces,*

$$\theta_1 \Uparrow \theta_2 = \theta_1 mgu(\widehat{\theta_2}\theta_1) = \theta_2 mgu(\widehat{\theta_1}\theta_2)$$

Lema 3.2.8 *Sea $\mathcal{R}$ un SRTC, g_0 un objetivo y $R, R' \ll \mathcal{R}$ dos reglas del programa. Entonces, $g_0 \leadsto_{q,R,\theta} g \leadsto_{p',R',\theta'} g'$, donde p' es la posición correspondiente a alguna posición $p \in \mathcal{FP}os(R)$ en $\mathcal{FP}os(g)$, sii $g_0 \leadsto_{q,R'',\theta''} g''$, donde $R'' \in unfold(R, R', p)$, $g' = g''$ y $\theta\theta' = \theta''$ $[\mathcal{V}ar(g_0)]$.*

Demostración.
($\Rightarrow$) Sea $R = (l \to r \Leftarrow C)$ y $R' = (l' \to r' \Leftarrow C')$. Por comodidad, asumamos que $p \in \mathcal{FP}os(C)$ y por tanto que $p' = p$ (el caso en que $p \in \mathcal{FP}os(r)$ es perfectamente análogo). Como $g_0 \leadsto_{q,R,\theta} g \leadsto_{p',R',\theta'} g'$, entonces tenemos que $\theta = mgu(\{g_0|_q = l\})$, $g = (C, g_0[r]_q)\theta$, $\theta' = mgu(\{C\theta|_p = l'\})$, y $g' = (C', C[r']_p, g_0[r]_q)\theta\theta'$.
Ahora, si consideramos que $\sigma = mgu(\{C|_p = l'\})$ entonces se verifican las siguientes equivalencias:

$$\begin{array}{lll}
\theta\theta' & = & \\
\theta mgu(\{C\theta|_p = l'\}) & = & \text{(ya que } Dom(\theta) \notin \mathcal{V}ar(R')\text{)} \\
\theta mgu(\widehat{mgu}(\{C|_p = l'\})\theta) & = & \\
\theta mgu(\widehat{\sigma}\theta) & = & \text{(por la Proposición 3.2.7)} \\
\theta \Uparrow \sigma & = & \text{(por la Proposición 3.2.7)} \\
\sigma mgu(\widehat{\theta}\sigma) & = & \\
\sigma mgu(\widehat{mgu}(\{g_0|_q = l\})\sigma) & = & \text{(ya que } Dom(\sigma) \notin \mathcal{V}ar(g_0)\text{)} \\
\sigma mgu(\{g_0|_q = l\sigma\}). & &
\end{array}$$

Además, como $\theta\theta' \neq fail$, entonces $\sigma \neq fail$ y por tanto existe una regla $R'' = (l \to r \Leftarrow C', C[r']_p)\sigma \in unfold(R, R', p)$. Ahora, como $mgu(\{g_0|_q = l\sigma\}) \neq fail$, podemos garantizar que existe el siguiente paso de *narrowing*:

$$g_0 \leadsto_{q,R'',\theta''} g''$$

donde $\theta'' = mgu(\{g_0|_q = l\sigma\})$ y $g'' = (C'\sigma, C[r']_p\sigma, g_0[r\sigma]_q)\theta''$. Finalmente, ya que $\theta\theta' = \sigma\theta''$ y $Dom(\sigma) \notin \mathcal{V}ar(g_0)$, tenemos que $g' = g''$ y $\theta\theta' = \theta''$ $[\mathcal{V}ar(g_0)]$, como queríamos demostrar.

($\Leftarrow$) Este caso se puede demostrar fácilmente de manera similar al caso anterior, explotando de nuevo la equivalencia entre $\theta\theta'$ y $\sigma\theta''$. $\square$

A continuación procedemos a enunciar la corrección parcial del desplegado. Su demostración se deriva fácilmente del resultado anterior.

Teorema 3.2.9 (corrección fuerte) *Sea $\mathcal{R}$ un SRTC y $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$.*

Sea g un objetivo. Si $g \rightsquigarrow^*_\theta$ true es una derivación de narrowing condicional en $\mathcal{R}'$, entonces existe una derivación de narrowing condicional $g \rightsquigarrow^*_{\theta'}$ true en $\mathcal{R}$ tal que $\theta = \theta'$ $[\mathcal{V}ar(g)]$.

DEMOSTRACIÓN. La idea de nuestra demostración consiste en probar que cada paso de *narrowing* en $\mathcal{R}'$ también puede realizarse usando una o dos reglas de $\mathcal{R}$, ya que el teorema es consecuencia directa de este hecho.
Sea $g_0 \rightsquigarrow_{R'',\theta''} g''$ un paso de *narrowing* condicional sin restricciones en $\mathcal{R}'$. Si $R'' \in \mathcal{R}$, entonces el resultado se cumple trivialmente. En otro caso, por la Definición 3.2.1, existen dos reglas $R, R' \in \mathcal{R}$ y una posición $p \in \mathcal{FP}os(R)$ tal que $R'' \in unfold(R, R', p)$. A su vez, por el Lema 3.2.8 sabemos que existe la derivación de *narrowing* $g_0 \rightsquigarrow_{R,\theta} g \rightsquigarrow_{R',\theta'} g'$ en $\mathcal{R}$, donde $g'' = g'$ y $\theta'' = \theta\theta'$ $[\mathcal{V}ar(g_0)]$, lo que demuestra el teorema. □

A continuación damos una serie de resultados auxiliares que nos ayudarán a probar la completitud de la operación de desplegado.

Proposición 3.2.10 *Sea $\mathcal{R}$ un SRTC y sea $\mathcal{R}'$ el resultado de desplegar $\mathcal{R}$ con respecto a alguna de sus reglas $R \in \mathcal{R}$. Si $\mathcal{R}$ es lineal por la izquierda y L-cerrado, entonces $\mathcal{R}'$ es L-cerrado.*

DEMOSTRACIÓN. La idea consiste en demostrar que, para toda regla $R = (l \to r \Leftarrow C) \in \mathcal{R}'$, el conjunto $terms(R)$ es L-cerrado. Si $R \in \mathcal{R}$ entonces el resultado es trivialmente cierto, ya que $\mathcal{R}$ es L-cerrado. En otro caso, R responde a alguna de las siguientes formas:

1. $R = (l' \to r'[r'']_p \Leftarrow C'', C')\sigma$, donde $R' = (l' \to r' \Leftarrow C') \ll \mathcal{R}$, $R'' = (l'' \to r'' \Leftarrow C'') \ll \mathcal{R}$, $p \in \mathcal{FP}os(r')$ y $\sigma = mgu(\{r'|_p = l''\}) \neq fail$.
2. $R = (l' \to r' \Leftarrow C'', C'[r'']_p)\sigma$ donde $R' = (l' \to r' \Leftarrow C') \ll \mathcal{R}$, $R'' = (l'' \to r'' \Leftarrow C'') \ll \mathcal{R}$, $p \in \mathcal{FP}os(C')$ y $\sigma = mgu(\{C'|_p = l''\}) \neq fail$.

Consideremos el primer caso. Como l'' es (trivialmente) L-cerrado, y $r'|_p$ también es L-cerrado (ya que el término r' pertenece a una regla que es L-cerrada), entonces tenemos que, para todo enlace $\{x \mapsto t\} \in \sigma$, el término t es L-cerrado. Así, σ sólo introduce términos L-cerrados en R. Como r' y r'' son términos L-cerrados, entonces $r'[r'']_p$ es también L-cerrado. Finalmente, sabemos que las condiciones de las reglas R' y R'' (i.e., C' y C'') son L-cerradas ya que ambas reglas pertenecen al programa original $\mathcal{R}$ que es L-cerrado. Por tanto, R es L-cerrada.

La demostración del segundo caso ($R = (l' \to r' \Leftarrow C'', C'[r'']_p)\sigma$) es perfectamente análoga al caso anterior. □

Obsérvese que, si $\mathcal{R}' = \{l'_j \to r'_j \Leftarrow C'_j\}_{j=1}^m$ y $L' = \{l'_1, \ldots, l'_m\}$, entonces $\mathcal{R}'$ no solamente es un programa L-cerrado, sino también L'-cerrado. De esta forma, como consecuencia directa del L-cierre de $\mathcal{R}'$, tenemos que $\mathcal{R}'$ es también L'-cerrado. Así, esta propiedad del desplegado garantiza que el desplegado repetido dentro de una secuencia de transformaciones que parte de un programa inicial $\mathcal{R}$ que es L-cerrado, genera programas desplegados que son cerrados no sólo con respecto al conjunto de términos dados por las partes izquierdas de las cabezas de las reglas del programa original, sino también con respecto al conjunto de términos dados por las partes izquierdas de las cabezas de las reglas de cada programa desplegado.

A continuación pretendemos demostrar que, además del L-cierre, la propiedad de decrecimiento también se preserva tras efectuar una operación de desplegado. Para ello introducimos la siguiente definición auxiliar, original de Bockmayr y Werner [1995], que nos permite caracterizar la relación de reescritura sin evaluación de las condiciones en un programa condicional.

Definición 3.2.11 (reducción sin evaluación de las condiciones)
Sea $\mathcal{R}$ un SRTC. Para secuencias de ecuaciones g y g', la relación de reescritura de términos condicional sin evaluación de la premisa $\hookrightarrow_{\mathcal{R}}$ se define como: $g \hookrightarrow_{\mathcal{R}} g'$ si existe una posición $p \in \mathcal{P}os(g)$, una variante $(l \to r \Leftarrow C) \ll \mathcal{R}$ de una regla de $\mathcal{R}$ y una sustitución σ tal que $g|_u = l\sigma$ y $g' = (C\sigma, g[r\sigma]_p)$.

El siguiente resultado establece la relación existente entre la noción anterior y la relación de *narrowing*.

Lema 3.2.12 [Bockmayr y Werner, 1995; Middeldorp y Hamoen, 1994] *Sea $\mathcal{R}$ un SRTC y sean g y g' dos secuencias de ecuaciones. Si $g \leadsto_{\theta}^{*} g'$ entonces $g\theta \hookrightarrow_{\mathcal{R}}^{*} g'$.*

A continuación probamos que el carácter decreciente de un programa se mantiene al ser desplegado.

Proposición 3.2.13 *Sea $\mathcal{R}$ un SRTC y sea $\mathcal{R}'$ el resultado de desplegar $\mathcal{R}$ con respecto a alguna de sus reglas $R \in \mathcal{R}$. Si $\mathcal{R}$ es decreciente, entonces $\mathcal{R}'$ es decreciente.*

Demostración. Por la Definición 3.2.1, cada regla $R = (l \to r \Leftarrow C)$ de $\mathcal{R}'$ también pertenece a $\mathcal{R}$, y en ese caso el resultado es cierto trivialmente, o bien se obtiene por desplegado a través de una derivación de *narrowing* de la forma:

$$(C_1, r_1 = y) \leadsto_{p,R_2,\theta_2} (C_2, C_1[r_2]_p, r_1 = y)\theta_2$$

donde $R_1 = (l_1 \to r_1 \Leftarrow C_1) \ll \mathcal{R}$, $R_2 = (l_2 \to r_2 \Leftarrow C_2) \ll \mathcal{R}$, y $R = (l_1 \to r_1 \Leftarrow C_2, C_1[r_2]_p)\theta_2$. Nótese que el caso en el que la posición explotada por *narrowing* pertenece a r_1 es completamente similar.

Ahora, podemos construir una nueva derivación de *narrowing* como sigue:

$$f(x_1, \ldots, x_n) = y \leadsto_{1.1,R_1,\theta_1} (C_1, r_1 = y) \leadsto_{p,R_2,\theta_2} (C_2, C_1[r_2]_p, r_1 = y)\theta_2$$

donde (f/n) es el símbolo de función más externo de l_1 y $f(x_1, \ldots, x_n)\theta_1 = l_1$. Entonces, R puede escribirse también como: $(f(x_1, \ldots, x_n) \to r_1 \Leftarrow C_2, C_1[r_2]_p)\theta_1\theta_2$. Como $\mathcal{R}$ es un programa decreciente, entonces existe un orden bien fundado monotóno $\succ$ sobre $\mathcal{T}(\Sigma, \mathcal{X})$ que es estable bajo sustitución (es decir, verifica que si $l \succ r$, entonces $l\sigma \succ r\sigma$ para toda sustitución σ), con las siguientes propiedades: 1) $\succ$ verifica la *propiedad de subtérminos*, i.e., $t \succ t|_p$ para todo $p \in \mathcal{P}os(t) - \{\Lambda\}$, y 2) si $(l \to r \Leftarrow s_1 = t_1, \ldots, s_k = t_k) \in \mathcal{R}$ y σ es una sustitución, entonces $l\sigma \succ r\sigma$ y $l\sigma \succ s_i\sigma, l\sigma \succ t_i\sigma$ para todo $i \in \{1, \ldots, k\}$. Además, como $\mathcal{R}$ es decreciente, $\hookrightarrow_{\mathcal{R}}$ es noetheriana [Bockmayr y Werner, 1995]. Por tanto podemos extender $\succ$ como sigue: $\ggcurly = (\succ \cup \hookrightarrow_{\mathcal{R}})$. Por el Lema 3.2.12, podemos dar el siguiente paso de reducción sin evaluación de la premisa en $\mathcal{R}$:

$$f(x_1, \ldots, x_n)\theta_1\theta_2 = y \hookrightarrow^*_{\mathcal{R}} (C_2, C_1[r_2]_p, r_1 = y)\theta_1\theta_2$$

Finalmente, es inmediato comprobar que $\ggcurly$ está bien fundado y disfruta de la propiedad de subtérmino. Esto nos permite demostrar el decrecimiento de R, ya que $f(x_1, \ldots, x_n)\theta_1\theta_2 \ggcurly r\theta_1\theta_2$ y $f(x_1, \ldots, x_n)\theta_1\theta_2 \ggcurly (C_2, C_1[r_2]_p)\theta_1\theta_2$, lo que completa la demostración. $\square$

Como consecuencia directa de las dos últimas proposiciones tenemos que, si $\mathcal{R}$ es un SRTC decreciente y L-cerrado, entonces el programa desplegado $\mathcal{R}'$ también lo es. Este importante resultado puede formalizarse como sigue:

Lema 3.2.14 *Sea $\mathcal{R}$ un SRTC lineal por la izquierda y sea $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Si $\mathcal{R}$ es decreciente y L-cerrado entonces $\mathcal{R}'$ también es decreciente y L-cerrado*

DEMOSTRACIÓN. Inmediato por las Proposiciones 3.2.10 y 3.2.13. $\square$

Finalmente, para demostrar la completitud del desplegado, recogemos aquí la noción de una estrategia de *narrowing* correcta y completa conocida como *narrowing* condicional normalizante

(ver, por ejemplo, Bockmayr y Werner [1995]). Esta relación es una forma especial de *narrowing* condicional sin restricciones donde a cada paso de *narrowing* le sigue un proceso de normalización. Este cálculo simplifica nuestra demostración de la completitud del desplegado, ya que las derivaciones de *narrowing* normalizante son más fácilmente simulables en los programas desplegados.

Teorema 3.2.15 (completitud) *Sea $\mathcal{R}$ un SRTC confluente, lineal por la izquierda, decreciente y L-cerrado. Sea $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea g un objetivo. Si $g \leadsto^*_{\theta}$ true es una derivación de narrowing condicional en $\mathcal{R}$, donde $\theta_{\lceil \mathcal{V}ar(g)}$ es normalizada, entonces existe una derivación de narrowing condicional $g \leadsto^*_{\theta'}$ true en $\mathcal{R}'$ tal que $\theta' \leq \theta$ $[\mathcal{V}ar(g)]$ y $\theta'_{\lceil \mathcal{V}ar(g)}$ es normalizada.*

DEMOSTRACIÓN. Sea $\mathcal{D}$ una derivación de éxito para g en $\mathcal{R}$ con sustitución normalizada θ. Como *narrowing* normalizante es correcto y completo para programas confluentes y decrecientes [Bockmayr y Werner, 1995], existe una derivación de éxito $\mathcal{D}'$ usando *narrowing* normalizante para g en $\mathcal{R}$ con susbtitución normalizada θ', tal que $\theta' \leq \theta\ [\mathcal{V}ar(g)]$.

Nuestro objetivo ahora consiste en demostrar que la derivación completa de *narrowing* normalizante $\mathcal{D}'$ puede simularse paso a paso usando el programa desplegado $\mathcal{R}'$ en vez del programa original $\mathcal{R}$. Hacemos la demostración por inducción sobre el número n de pasos en la derivación $\mathcal{D}'$.

Sea $n = 1$. Si $g \leadsto_\theta true$ en $\mathcal{R}$, entonces $g \leadsto_\theta true$ en $\mathcal{R}'$ ya que solo se ha realizado un paso de unificación sintáctica, sin necesidad de tener en cuenta ninguna regla de programa.

Consideremos ahora el caso inductivo donde $n > 1$. Sea $R = (l \to r \Leftarrow C) \in \mathcal{R}$. Asumamos que $g \leadsto_{p,R,\theta} (C, g[r]_p)\theta \to^* (C, g[r]_p)\theta\downarrow\ \leadsto^*_{\theta'} true$. En este momento debemos considerar dos posibilidades. Si $R \in \mathcal{R}'$, entonces, por la hipótesis de inducción, la derivación puede simularse también en $\mathcal{R}'$.

En caso contrario, si $R \notin \mathcal{R}'$, entonces $Unf_{\mathcal{R}}(R) \neq \varnothing$. Como $\mathcal{R}$ es L-cerrado y $Unf_{\mathcal{R}}(r) \neq \varnothing$ entonces $\exists q \in \mathcal{FP}os(C, r = y)$ de tal forma que $(C, r = y)|_q$ no está en forma normal. Por tanto, tenemos que el objetivo $(C, g[r]_p)\theta$ no está normalizado. Esto implica la existencia de una regla $R' = (l' \to r' \Leftarrow C') \ll \mathcal{R}$ tal que $l' \leq (C, r = y)|_q$. Como estamos considerando una derivación de *narrowing* normalizante, el segundo paso de $\mathcal{D}'$ es un paso de reescritura. Por otra parte, al ser $\mathcal{R}$ un programa confluente y decreciente, entonces podemos asumir sin pérdida de generalidad que ese paso de reescritura se da usando la regla R' sobre la ocurrencia p' del objetivo $(C, g[r]_p)\theta$ (siendo $p' = q$ si $q \in \mathcal{FP}os(C)$ o $p' = p.q$ en otro caso). Además, como *narrowing* subsume la reescritura, tenemos que $\mathcal{D}'$ puede expresarse como $g \leadsto_{p,R,\theta]} (C, g[r]_p)\theta \leadsto_{p',R',\theta'} g' \to^* g'\downarrow\ \leadsto^*_\sigma true$ en $\mathcal{R}$. Por el Lema 3.2.8, tenemos que los dos primeros pasos de la derivación pueden efectuarse en $\mathcal{R}'$ dando un solo paso de derivación con la regla $R'' \in unfold(R, R', p)$, es decir, verificando que $g \leadsto_{p,R'',\theta''} g''$ donde $g' = g''$ y $\theta\theta' = \theta''[\mathcal{V}ar(g)]$. Por la hipótesis de inducción el resto de la derivación puede simularse en el programa desplegado. Por tanto, el resultado se cumple para la derivación completa, lo que implica la completitud del desplegado, como queríamos demostrar. $\square$

Como acabamos de ver, el hecho de incorporar normalización en el seno del mecanismo básico de *narrowing* condicional nos ha permitido simular dentro de un programa desplegado $\mathcal{R}'$ cualquier derivación de éxito (para todo objetivo g) que pueda verificarse en el correspondiente programa original $\mathcal{R}$. O más exactamente, el conjunto de respuestas normalizadas asociado al conjunto de todas las derivaciones de éxito para cualquier objetivo g coincide en el programa original $\mathcal{R}$ y en el desplegado $\mathcal{R}'$.

Esto implica que, con este primer refinamiento de *narrowing* es posible demostrar un resultado de corrección y completitud fuertes para la transformación de desplegado como el que se formaliza en el siguiente corolario.

Corolario 3.2.16 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un SRTC confluente, lineal por la izquierda, decreciente y L-cerrado. Sea $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea g un objetivo. Entonces, $g \leadsto_{\theta}^{*}$ true es una derivación de narrowing condicional normalizante en $\mathcal{R}$, donde $\theta\lceil_{\mathcal{V}ar(g)}$ es normalizada, sii $g \leadsto_{\theta'}^{*}$ true es una derivación de narrowing condicional normalizante en $\mathcal{R}'$, donde $\theta'\lceil_{\mathcal{V}ar(g)}$ es normalizada, tal que $\theta' = \theta\ [\mathcal{V}ar(g)]$.*

Demostración. La demostración para el caso ($\Leftarrow$) es perfectamente análoga a la dada para el Teorema 3.2.9, mientras que para el segundo caso, ($\Rightarrow$), la demostración se sigue como un caso particular de la desarrollada para el Teorema 3.2.15. $\square$

Además, como consecuencia de los Teoremas 3.2.9 y 3.2.15, la transformación de desplegado basado en *narrowing* condicional sin restricciones es correcta con respecto a la semántica del $\mathcal{E}$-modelo mínimo de Herbrand (es decir, el conjunto de ecuaciones básicas que se reducen a $true$) bajo las condiciones de confluencia, linealidad por la izquierda, decrecimiento y L-cierre de los programas originales.

En términos sencillos, la condición de que $\mathcal{R}$ sea L-cerrado requiere que las llamadas en las partes derechas y en las condiciones de las reglas del programa cuyo símbolo más externo sea un símbolo de función definido, no estén en forma normal. Además, la necesidad adicional de que el programa sea decreciente asegura que dichas llamadas pueden ser normalizadas finitamente. Estas condiciones son suficientes para garantizar que la desaparición de la regla desplegada en el programa transformado y su sustitución por el conjunto de reglas que se obtienen al desplegarse ésta puede hacerse de forma segura, sin correr el riesgo de perder completitud.

Obsérvese también que la combinación de linealidad por la izquierda y L-cierre garantiza que las llamadas en el programa transformado $\mathcal{R}'$ seguirán siendo cerradas recursivamente con respecto al conjunto correspondiente de partes izquierdas de las cabezas de las reglas de $\mathcal{R}'$. Esta característica es interesante de cara a realizar posteriores desplegados sobre programas previamente transformados, ya que se mantiene a lo largo de toda la cadena de transformaciones (ver de nuevo la Proposición 3.2.10). Además, es interesante exigir siempre ambas condiciones ya que en la mayoría de aplicaciones y situaciones practicas (evaluación parcial, sistemas de transformación, semánticas por desplegado, etc.) el desplegado es una operación que, en general, no se realiza una sola vez de forma aislada, sino en etapas sucesivas y de forma encadenada. En otro caso, si no estamos interesados en realizar repetidas operaciones de desplegado, dichos requerimientos pueden retirarse del conjunto de premisas del Teorema

3.2.15 y del Corolario 3.2.16, dejando en su lugar la condición más simple y relajada (que ya no tiene por qué preservarse a lo largo de toda una secuencia de desplegados) de que todas las llamadas en $\mathcal{R}$ sean instancias de los términos de L.

Ejemplo 14 Consideremos el siguiente programa $\mathcal{R}$:

$$\begin{array}{rcll} \mathtt{f(X)} & \to & \mathtt{g(h(X))} & (R_1) \\ \mathtt{g(X))} & \to & \mathtt{X} & (R_2) \\ \mathtt{h(0)} & \to & \mathtt{0} & (R_3) \end{array}$$

Obsérvese que el programa $\mathcal{R}$ no es L-cerrado debido a que la parte derecha de la primera regla (el término $\mathtt{g(h(X))}$) no es cerrado con respecto al conjunto $L = \{\mathtt{f(X)}, \mathtt{g(X)}, \mathtt{h(0)}\}$. Por otra parte, nótese que aún no siendo $\mathtt{g(h(X))}$ un término L-cerrado, sí es instancia de la parte izquierda de R_2.

El resultado de desplegar $\mathcal{R}$ con respecto a R_1 devuelve el siguiente programa transformado $\mathcal{R}'$:

$$\begin{array}{rcll} \mathtt{f(X)} & \to & \mathtt{h(X)} & (R_4) \\ \mathtt{g(X))} & \to & \mathtt{X} & (R_2) \\ \mathtt{h(0)} & \to & \mathtt{0} & (R_3) \end{array}$$

Ahora $\mathcal{R}'$ además de no ser L-cerrado, contiene una regla (la regla R_4, que potencialmente es la única desplegable) cuya parte derecha no es instancia de la parte izquierda de ninguna otra regla de $\mathcal{R}'$, lo que prohibe la posibilidad de que $\mathcal{R}'$ pueda ser nuevamente desplegado (con respecto a ninguna de sus reglas) de forma segura.

Como es bien conocido, la completitud del cálculo de *narrowing* (condicional) puede garantizarse sin necesidad de exigir condiciones de terminación, siempre y cuando se exija en contrapartida que el observable de interés se centre únicamente en soluciones normalizadas [Middeldorp y Hamoen, 1994]. Ya que en nuestro caso la propiedad de terminación no puede ser eliminada puesto que es esencial para garantizar la completitud de la transformación, podría parecer lógico el plantearse si la restricción a soluciones normalizadas del Teorema 3.2.15 y del Corolario 3.2.16 puede ser relajada de alguna forma. Sorprendentemente, el siguiente ejemplo responde negativamente a esta cuestión, salvo que, en compensación, debilitemos la relación entre las respuestas computadas a $\leq_{\mathcal{E}}$.

Ejemplo 15 Consideremos el siguiente programa confluente y decreciente $\mathcal{R}$:

$$\begin{array}{rcll} \mathtt{f(X)} & \to & \mathtt{X} & (R_1) \\ \mathtt{f(c(X))} & \to & \mathtt{c(f(X))} & (R_2) \end{array}$$

que es L-cerrado, ya que es cerrado con respecto al conjunto de términos $L = \{\mathtt{f(x)}, \mathtt{f(c(x))}\}$. Ahora, si desplegamos el programa original con respecto a su segunda regla obtenemos el programa $\mathcal{R}'$:

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

Dado el objetivo $g = (\mathtt{f(c(c(X)))} = \mathtt{Y})$ se obtiene la respuesta (no normalizada) $\theta = \{\mathtt{Y} \mapsto \mathtt{c(f(c(X)))}\}$ en $\mathcal{R}$, mientras que no existe ninguna respuesta más general para g en $\mathcal{R}'$. Unicamente, las soluciones más "$\mathcal{E}$-generales" ($\leq_{\mathcal{E}}$) $\{\mathtt{Y} \mapsto \mathtt{f(c(c(X)))}\}$, $\{\mathtt{Y} \mapsto \mathtt{c(c(f(X)))}\}$ y $\{\mathtt{Y} \mapsto \mathtt{c(c(X))}\}$ pueden ser calculadas para dicho objetivo en $\mathcal{R}'$.

El uso de formas eficientes de *narrowing* puede mejorar significativamente la eficacia de cualquier método de transformación e incrementar la eficiencia de los programas resultantes. Más aún, si para el caso del desplegado basado en *narrowing* condicional las condiciones necesarias para garantizar la completitud del desplegado son excesivamente restrictivas, es de esperar que al utilizar refinamientos de *narrowing* en el mecanismo de desplegado tal vez se puedan relajar tales requerimientos a otros más razonables y sencillos (es decir, con menos restricciones adicionales respecto a las que se necesitan para la completitud del propio refinamiento de *narrowing* empleado). En el siguiente apartado mostramos una variedad del desplegado que, al estar expresada en términos de *narrowing* impaciente, disfruta de mejores propiedades.

3.3 Desplegado basado en *narrowing* impaciente

Como vimos en la Sección 2.2.1, la estrategia de *narrowing* (condicional) impaciente es un refinamiento de *narrowing* que, para programas canónicos, basados en constructores y completamente definidos (CB-CD), es correcto y completo con respecto al conjunto de soluciones constructoras básicas. Esto significa que, para todo programa $\mathcal{R}$ que cumpla las condiciones anteriores, dado un objetivo g y una solución σ para g, con $\mathcal{V}ar(g) \subseteq Dom(\sigma)$, existe una sustitución de respuesta computada θ para $\mathcal{R} \cup \{g\}$ usando *narrowing* condicional impaciente $\overset{\mathtt{NI}}{\leadsto}$ tal que $\theta \leq \sigma$ $[\mathcal{V}ar(g)]$ [Fribourg, 1985].

A continuación veremos como adaptar y propagar todas estas propiedades al marco de la transformación de desplegado basada en por dicha estrategia de *narrowing*. En esta sección comprobaremos que la transformación de desplegado disfruta de las mismas propiedades de corrección total con respecto a la misma clase de programas (y el mismo tipo de soluciones) para los cuales ya es correcta y completa la propia estrategia de *narrowing* impaciente.

Una característica importante de la estrategia de *narrowing* impaciente es que la función de selección de la posición más interna que es reducida en cada paso presenta un indeterminismo de tipo *don't care*. Ello implica que es suficiente con considerar una sola posición más interna en cada paso, independientemente de que un objetivo concreto disponga de varias posiciones de este tipo. Nótese que aunque sólo se selecciona un único subtérmino más interno en cada paso, es necesario contrastar e intentar unificar dicho subtérmino con todas y cada una de las reglas del programa (indeterminismo *don't know*), lo que implica que tras un desplegado el conjunto de reglas resultantes puede contener más de una regla. Algo similar ocurre por ejemplo con la resolución SLD, el mecanismo operacional de los lenguajes lógicos puros. En particular, de entre dichas posiciones más internas puede seleccionarse la que aparece más a la izquierda. En cualquier caso y sin perdida de generalidad (gracias al indeterminismo "don't-care" de la función de selección), preferimos dejar sin fijar esta posición en nuestras definiciones tanto de *narrowing* como de desplegado impaciente. Así pues, en la Definición 3.3.1, nos limitaremos simplemente a indicar que existe una ocurrencia más interna seleccionada en cada operación de desplegado, sin necesidad de establecer un criterio que fije cuál de ellas se elige en cada caso.

El desplegado basado en *narrowing* impaciente se define como una instancia del desplegado genérico basado en *narrowing* condicional simplemente concretando la relación $\leadsto$ como $\overset{\texttt{NI}}{\leadsto}$ en dicha definición.

Definición 3.3.1 (desplegado impaciente) Sea $\mathcal{R}$ un SRTC y $R = (l \to r \Leftarrow C) \ll \mathcal{R}$ una regla (renombrada) del mismo. Sea $p \in \mathcal{FP}os(R)$ una posición asociada a un subtérmino más interno de la regla R. El desplegado impaciente del programa $\mathcal{R}$ con respecto a la regla R y la posición p es el programa $\mathcal{R}'$ definido como sigue:

$$\mathcal{R}' = (\mathcal{R} - \{R\}) \cup \{(l\theta \to r' \Leftarrow C') \mid (C, r = y) \overset{\texttt{NI}}{\leadsto}_{p,\theta} (C', r' = y)\} \ .$$

Obsérvese que, tal y como hemos avanzado anteriormente, esta definición únicamente presenta dos novedades con respecto a definición genérica de desplegado:

1. la sustitución de la relación $\leadsto$ por $\overset{\texttt{NI}}{\leadsto}$ y
2. la consideración de una posición más interna particular de entre todas las posibles.

El siguiente ejemplo ilustra la definición de desplegado impaciente.

Ejemplo 16 Consideremos el siguiente programa $\mathcal{R}$ que es canónico y CB-CD:

$$\begin{array}{rcll}
\texttt{height(X)} & \to & \texttt{0} \Leftarrow \texttt{X} = \texttt{nil} & (R_1)\\
\texttt{height(tree(L,N,R))} & \to & \texttt{succ(max(height(L),height(R)))} & (R_2)\\
\texttt{balanced(nil)} & \to & \texttt{true} & (R_3)\\
\texttt{balanced(tree(L,N,R))} & \to & \texttt{true} \Leftarrow \texttt{height(L)} = \texttt{height(R)}, & \\
 & & \qquad \texttt{balanced(L)} = \texttt{true}, & \\
 & & \qquad \texttt{balanced(R)} = \texttt{true} & (R_4)
\end{array}$$

El desplegado impaciente del programa $\mathcal{R}$ con respecto a la regla R_4 en la posición 2.1 de la expresión $(\mathtt{height(L)} = \mathtt{height(R)},\ \mathtt{balanced(L)} = \mathtt{true},\ \mathtt{balanced(R)} = \mathtt{true},\ \mathtt{true} = \mathtt{Y})$ (que se refiere al subtérmino más interno $\mathtt{balanced(L)}$ del cuerpo de la regla), es el siguiente programa $\mathcal{R}'$ (por simplicidad, omitimos las ecuaciones de la forma $\mathtt{true} = \mathtt{true}$):

$$
\begin{array}{rcl}
\mathtt{height(nil)} & \to & \mathtt{0} \\
\mathtt{height(tree(L,N,R))} & \to & \mathtt{succ(max(height(L),height(R)))} \\
\mathtt{balanced(nil)} & \to & \mathtt{true} \\
\mathtt{balanced(tree(nil,N,R))} & \to & \mathtt{true} \Leftarrow \mathtt{height(nil)} = \mathtt{height(R)}, \\
 & & \qquad \mathtt{balanced(R)} = \mathtt{true} \\
\mathtt{balanced(tree(tree(L',N',R'),N,R))} & \to & \mathtt{true} \Leftarrow \mathtt{height(L')} = \mathtt{height(L')}, \\
 & & \qquad \mathtt{balanced(L')} = \mathtt{true}, \\
 & & \qquad \mathtt{balanced(R')} = \mathtt{true}, \\
 & & \qquad \mathtt{balanced(R)} = \mathtt{true}, \\
 & & \qquad \mathtt{height(tree(L',N',R'))} = \\
 & & \qquad\qquad \mathtt{height(R)}
\end{array}
$$

Como puede verse, resulta fácil comprobar que ambos programas producen exactamente las mismas respuestas computadas para cualquier objetivo cuando se ejecutan con *narrowing* impaciente.

El siguiente teorema establece la equivalencia computacional entre un programa y cualquiera de sus desplegados impacientes bajo los requerimientos estándar que aseguran la completitud del *narrowing* impaciente. La prueba de este resultado se basa en la relación precisa existente entre la EP y el desplegado de programas basados en *narrowing* impaciente. Postponemos a la Sección 3.6 esta demostración, ya que es allí donde profundizaremos en la relación entre ambas técnicas de transformación.

Teorema 3.3.2 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un SRTC canónico y CB-CD. Sea $R \in \mathcal{R}$ una regla del mismo y sea $p \in \mathcal{FP}os(R)$ la posición de un subtérmino más interno de R, tal que el resultado de desplegar $\mathcal{R}$ con respecto a la posición p de R es el programa transformado $\mathcal{R}'$. Sea g un objetivo. Entonces, $g \overset{\text{NI}}{\leadsto}{}^*_{\theta}$ true es una derivación de narrowing impaciente en $\mathcal{R}$, sii $g \overset{\text{NI}}{\leadsto}{}^*_{\theta'}$ true es una derivación de narrowing impaciente en $\mathcal{R}'$, donde $\theta' = \theta\ [\mathcal{V}ar(g)]$.*

Como hemos visto, el desplegado impaciente disfruta de correción y completitud fuertes bajo una serie de condiciones razonables. Tales condiciones coinciden con las que garantizan la completitud de la propia estrategia de *narrowing* impaciente que se usa para dirigir el desplegado. Esta propiedad es ciertamente muy interesante, y contrasta con el caso de desplegado basado en *narrowing* condicional, donde los requerimientos exigidos para su correcta aplicabilidad son realmente muy restrictivos en comparación con los que garantizan la completitud de la estrategia. En general, estamos interesados en aquellas operaciones de desplegado que se muestren totalmente correctas bajo las mismas condiciones que garanticen la completitud de la estrategia de *narrowing* en que se base la transformación (como ocurre con el *narrowing* impaciente pero no con el *narrowing* condicional).

Si hasta ahora nos hemos centrado en el desplegado de un tipo genérico de programas que necesariamente estaban marcados por su carácter inherentemente terminante, en el siguiente apartado intentaremos refinar la noción de desplegado con el objetivo de poder trabajar con programas no terminantes. Para ello utilizaremos dos variantes perezosas de *narrowing* con las que obtendremos resultados bastante desiguales. En particular, las mayores discrepancias surgirán en la línea que comentábamos en el párrafo anterior, es decir, en la relación existente entre las condiciones de aplicabilidad que aseguren la corrección y completitud del desplegado y la estrategia de *narrowing* en que se basa éste.

3.4 Desplegados perezosos

En esta sección presentamos dos variantes perezosas del desplegado capaces de transformar de forma segura programas no terminantes, utilizando las estrategias perezosas de *narrowing* definidas en las secciones 2.2.2 (*narrowing* perezoso) y 2.2.3 (*narrowing* necesario). Por una parte, veremos que al definir el desplegado en términos del *narrowing* perezoso (aquél conducido por la demanda de forma ingenua) sobre el tipo de programas para los cuales el cálculo es correcto y completo (programas ortogonales), la transformación genera programas que no están dentro de la misma clase, con lo que se corre el riesgo de que las ejecuciones perezosas sobre los mismos no sean siempre correctas. Únicamente al restringir fuertemente las condiciones de aplicabilidad del desplegado a programas uniformes, se recuperará la corrección y completitud (débiles) de la transformación.

Por otro lado, al dirigir el desplegado con una estrategia refinada de *narrowing* perezoso, como es el *narrowing* necesario, veremos que se obtienen resultados óptimos de corrección y completitud fuertes, al tiempo que la transformación preserva la clase natural de programas (inductivamente secuenciales) sobre los que se define la propia estrategia de *narrowing* necesario.

3.4.1 Desplegado basado en *narrowing* perezoso

Antes de dar una definición del desplegado basado en el *narrowing* perezoso consideremos el siguiente ejemplo, donde se aprecia la pérdida de completitud de la transformación al definir ésta como una simple instancia de la definición genérica de desplegado usando la estrategia de *narrowing* perezoso.

Ejemplo 17 Consideremos el siguiente programa $\mathcal{R}$:

$$\begin{array}{rcll} \mathtt{f(0)} & \to & \mathtt{0} & (R_1) \\ \mathtt{g(X)} & \to & \mathtt{s(f(X))} & (R_2) \\ \mathtt{h(s(X))} & \to & \mathtt{s(0)} & (R_3)\ \}. \end{array}$$

Al desplegar la segunda regla (cuya parte derecha está encabezada por un símbolo constructor) de $\mathcal{R}$, obtenemos el programa $\mathcal{R}'$:

$$\begin{array}{rcll} \mathtt{f(0)} & \to & 0 & (R_1) \\ \mathtt{g(0)} & \to & \mathtt{s(0)} & (R_4) \\ \mathtt{h(s(X))} & \to & \mathtt{s(0)} & (R_3) \end{array}$$

Ahora, el objetivo $\mathtt{h(g(s(0)))} \approx \mathtt{X}$ admite la siguiente derivación de *narrowing* perezoso en $\mathcal{R}$:

$$\begin{array}{lll} \mathtt{h(\underline{g(s(0))})} \approx \ \mathtt{X} & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1.1,R_2},\epsilon} & \underline{\mathtt{h(s(f(s(0))))}} \approx \mathtt{X} \\ & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1,R_3},\epsilon} & \mathtt{s(0)} \approx \mathtt{X} \\ & \overset{\mathtt{NP}}{\leadsto}^{*}_{\{\mathtt{X}\mapsto\mathtt{s(0)}\}} & \mathtt{true} \end{array}$$

mientras que falla en el programa transformado $\mathcal{R}'$. El problema se debe al hecho de que, mientras en $\mathcal{R}$ la función $\mathtt{g}$ está definida para cualquier valor $\mathtt{X}$, en $\mathcal{R}'$ su dominio queda reducido al valor 0. Este fenómeno viene originado por el hecho de desplegar un término que está encabezado por un símbolo constructor en la parte derecha de una regla. Este fenómeno es común a cualquier tipo de desplegado perezoso, incluido también el necesario que veremos después.

En general, al desplegar de forma perezosa una regla que define a una función f, se aplica una sustitución que puede afectar a las variables de su parte izquierda, reduciendo su dominio de aplicación. En programas ortogonales, la restricción de este dominio no afecta a las reducciones que llevan a la forma normal un objetivo encabezado por f, pero no ocurre lo mismo cuando consideramos reducciones que conducen a la correspondiente forma normal en cabeza.

En el ejemplo anterior, se ve claramente que cualquier objetivo de la forma $\mathtt{g(t)}$ puede alcanzar su forma normal $\mathtt{s(0)}$ únicamente si el término $\mathtt{t}$ es (o puede reducirse a) 0, tanto en el programa original como en el transformado. Sin embargo, para ese mismo objetivo, la forma normal en cabeza $\mathtt{s(\Box)}$ se alcanza en $\mathcal{R}$ con tan sólo un paso de reescritura para cualquier valor de t, mientras que en $\mathcal{R}'$ el dominio restringido de la regla R_4 influye de forma similar a cuando se calculan formas normales.

La restricción del dominio de una función desplegada afecta pues a su capacidad para computar formas normales en cabeza, lo cual es fundamental cuando se consideran estrategias de *narrowing* perezosas. Sin embargo, este problema desaparece cuando no se permite desplegar la parte derecha de una regla si ésta está encabezada por un constructor ya que, obviamente, en este caso el término desplegado se encuentra en forma normal en cabeza (aunque no necesariamente en forma normal).

Por otro lado, el problema comentado desaparece cuando se trabaja con programas completamente definidos (lo cual es bastante frecuente en contextos heterogéneos) ya que en este caso todas las funciones están definidas sobre todo el dominio (como no es el caso de `f` en el ejemplo anterior) y el desplegado tampoco altera esta característica. Este interesante resultado será tenido en cuenta en la aplicación de algunas estrategias de transformación (como la *formación de tuplas*) más adelante.

En cualquier caso y dado que la restricción a programas completamente definidos es raramente contemplada (por su carácter fuertemente restrictivo) en el contexto de computaciones perezosas sobre programas no terminantes, en lo que sigue preferimos imponer en la definición de desplegado perezoso la condición de que las partes derechas de las reglas a desplegar estén encabezadas por un símbolo definido (lo que elimina la necesidad de imponer que los programas a desplegar sean completamente definidos).

La siguiente definición introduce nuestra regla de desplegado de una forma muy similar a como ya lo hemos hecho en las variantes consideradas anteriormente, con la particularidad de que aquí hacemos uso de la relación de *narrowing* perezoso y que consideramos el caso (más simple) incondicional (que es el caso habitual en la literatura sobre este tipo de programas).

Definición 3.4.1 (desplegado perezoso) Sea $\mathcal{R}$ un SRT y $R = (l \to r) \ll \mathcal{R}$ una regla (renombrada) del mismo, tal que r está encabezado por un símbolo definido. El desplegado perezoso del programa $\mathcal{R}$ con respecto a la regla R es el programa $\mathcal{R}'$ definido como sigue:

$$\mathcal{R}' = (\mathcal{R} - \{R\}) \cup \{(l\theta \to r') \mid r \overset{\text{NP}}{\leadsto}_{\theta} r'\} .$$

Nótese que, a diferencia de la definición de desplegado impaciente (Definición 3.3.1), el desplegado perezoso de un programa $\mathcal{R}$ con respecto a una de sus reglas R debe considerar todas las posibles posiciones demandadas (por la estrategia perezosa) de la parte derecha de R, en vez de explotar tan sólo una sola posición de entre todas las posibles ocurrencias (asociadas a subtérminos más internos) que considera la estrategia impaciente. Este hecho, que también se verifica en el desplegado condicional sin restricciones, viene impuesto por el indeterminismo de tipo *don't know* que presenta la función de selección de la posición que debe explotarse en cada paso de *narrowing*. En general, el indeterminismo de tipo *don't know* que presenta la función de selección de posiciones a explotar es común a todas las estrategias de *narrowing* (si realmente se pretende preservar la completitud del cálculo), excepto para algunos refinamientos, como el impaciente, donde esta clase de indeterminismo se reduce a uno de tipo *don't care*. Por esta razón, mientras que en la Definición 3.3.1 es necesario indicar explícitamente cuál es el término más interno que se despliega en cada caso, en la definición de desplegado perezoso (y también en la que se basa en *narrowing* condicional sin restricciones), se omite este hecho, lo que implícitamente implica que todas las posiciones demandadas de la parte derecha de una regla deben ser desplegadas para no perder completitud en la transformación. El siguiente ejemplo da cuenta de este hecho.

Ejemplo 18 Consideremos el siguiente programa basado en constructores $\mathcal{R}$:

$$\begin{array}{rcll}
\mathtt{g(0,X)} & \rightarrow & \mathtt{0} & (R_1) \\
\mathtt{g(X,0)} & \rightarrow & \mathtt{0} & (R_2) \\
\mathtt{h(0)} & \rightarrow & \mathtt{0} & (R_3) \\
\mathtt{f(c(X))} & \rightarrow & \mathtt{c(f(X))} & (R_4) \\
\mathtt{ok(X)} & \rightarrow & \mathtt{g(f(X),h(X))} & (R_5)
\end{array}$$

Nótese que, siguiendo la Definición 3.4.1, no es posible desplegar la regla R_4 ya que $\mathtt{c(f(X))}$ está encabezado por un constructor. Por otra parte, si desplegamos (todas las posiciones demandadas de) la regla R_5 en el programa $\mathcal{R}$ obtenemos el siguiente programa transformado $\mathcal{R}'$:

$$\begin{array}{rcll}
\mathtt{g(0,X)} & \rightarrow & \mathtt{0} & (R_1) \\
\mathtt{g(X,0)} & \rightarrow & \mathtt{0} & (R_2) \\
\mathtt{h(0)} & \rightarrow & \mathtt{0} & (R_3) \\
\mathtt{f(c(X))} & \rightarrow & \mathtt{c(f(X))} & (R_4) \\
\mathtt{ok(c(X))} & \rightarrow & \mathtt{g(c(f(X)),h(c(X)))} & (R_6) \\
\mathtt{ok(0)} & \rightarrow & \mathtt{g(f(0),0)} & (R_7)
\end{array}$$

Obsérvese que, tal y como hemos dicho, es necesario desplegar todas las posiciones demandadas en la parte derecha (que está encabezada por un símbolo definido) de la regla R_5 para asegurar la completitud de la transformación. En otro caso, si por ejemplo no desplegamos el término $\mathtt{h(X)}$ demandado en la regla R_5, entonces el objetivo $\mathtt{ok(0)} \approx \mathtt{0}$ (que puede ser evaluado a $\mathtt{true}$ en el programa original $\mathcal{R}$) no podría ser reducido a $\mathtt{true}$ en $\mathcal{R}'$ (ya que habríamos perdido la regla R_7).

A continuación pasamos a demostrar la corrección parcial de la regla de desplegado.

Teorema 3.4.2 (corrección) *Sea $\mathcal{R}$ un SRT CB y ortogonal y sea $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea g un objetivo. Si $g \overset{\mathtt{NP}}{\leadsto}{}^{*}_{\theta}$ true es una derivación de narrowing perezoso en $\mathcal{R}'$, entonces existe una derivación de narrowing perezoso $g \overset{\mathtt{NP}}{\leadsto}{}^{*}_{\theta'}$ true en $\mathcal{R}$ tal que $\theta' \leq \theta$ $[\mathcal{V}ar(g)]$.*

DEMOSTRACIÓN. Consideremos que $\mathcal{D} : g \overset{\text{NP}}{\leadsto}{}^*_\theta\ true$ es una derivación de éxito para un objetivo g en $\mathcal{R}'$. Nuestra demostración se basa en la construcción de una nueva derivación de éxito $\mathcal{D}'$ para g en $\mathcal{R}$ que entregue la misma respuesta computada que $\mathcal{D}$. La construcción de $\mathcal{D}'$ se basa en la estructura de $\mathcal{D}$.

1. Cada paso de *narrowing* que se da en $\mathcal{D}$ usando una regla que pertenece tanto a $\mathcal{R}$ como a $\mathcal{R}'$ se computa de forma idéntica en $\mathcal{D}'$.

2. Por otro lado, aquellos pasos que se dan en $\mathcal{D}$ con la regla desplegada $R'' \in \mathcal{R}'$ (donde $R'' \notin \mathcal{R}$) se simulan en $\mathcal{D}'$ explotando el par de reglas $R = l \to r \in \mathcal{R}$ y $R' = l' \to r' \in \mathcal{R}$, tal que el resultado de desplegar R usando R' es R''. Por el Lema 3.2.8, tenemos que un paso de *narrowing* perezoso en $\mathcal{D}$ de la forma $g_i \overset{\text{NP}}{\leadsto}_{q,R'',\theta'} g_{i+1}$ puede simularse en $\mathcal{D}'$ mediante el par de pasos de *narrowing* (no necesariamente perezoso) siguientes: $g_i \leadsto_{q,R,\sigma} g' \leadsto_{R',\sigma'} g_{i+1}$ donde $\theta' = \sigma\sigma'\ [\mathcal{V}ar(g_i)]$. Obsérvese que, con un argumento similar al que utilizábamos en la demostración del Teorema 3.3.2, el Lema 3.2.8 es aplicable en nuestro caso ya que todo paso de *narrowing* perezoso también lo es (en particular de) *narrowing* condicional sin restricciones.

De esta forma sabemos que $\mathcal{D}'$ es una derivación de éxito para g en $\mathcal{R}$ que calcula la misma respuesta computada (restringida a las variables de g) por $\mathcal{D}$ en $\mathcal{R}'$.

Como hemos visto, la derivación de *narrowing* $\mathcal{D}'$ no es necesariamente perezosa en general. Sin embargo, al ser $\mathcal{R}$ un programa CB y ortogonal (por las condiciones de aplicabilidad exigidas en el teorema), sabemos que la completitud en $\mathcal{R}$ está asegurada cuando se resuelven objetivos usando *narrowing* perezoso. Ello conlleva, en nuestro caso particular, la existencia de una derivación perezosa de éxito $\mathcal{D}''$ para g en $\mathcal{R}$ que devuelve una respuesta igual o más general que la computada por $\mathcal{D}'$.

Por tanto podemos concluir que si $\mathcal{D}$ es un derivación perezosa para el objetivo g en $\mathcal{R}'$ con respuesta computada θ, entonces seguro que existe otra derivación perezosa $\mathcal{D}''$ para el mismo objetivo g en $\mathcal{R}$ con respuesta computada igual o más general θ', o lo que es lo mismo $\theta' \leq \theta\ [\mathcal{V}ar(g)]$, como queríamos demostrar. □

La tarea de probar la completitud de la regla de desplegado mediante la simulación en un programa desplegado de todas las derivaciones que tienen éxito en el programa original correspondiente, no es ni mucho menos una labor trivial. El siguiente ejemplo muestra un nuevo tipo de dificultad en este sentido.

Ejemplo 19 Consideremos el siguiente programa CB y ortogonal $\mathcal{R}$:

$$
\begin{array}{rcll}
\texttt{f(X)} & \to & \texttt{g(X,X)} & (R_1) \\
\texttt{g(0,X)} & \to & \texttt{0} & (R_2) \\
\texttt{h(0)} & \to & \texttt{0} & (R_3)
\end{array}
$$

El desplegado de $\mathcal{R}$ con respecto a la regla R_1 es el siguiente programa transformado $\mathcal{R}'$:

$$\begin{array}{rcll} \mathtt{f(0)} & \to & \mathtt{0} & (R_4) \\ \mathtt{g(0,X)} & \to & \mathtt{0} & (R_2) \\ \mathtt{h(0)} & \to & \mathtt{0} & (R_3) \end{array}$$

Nótese que únicamente existe la siguiente derivación perezosa de éxito $\mathcal{D}$ para el objetivo $\mathtt{f(h(X))} \approx \mathtt{0}$ en $\mathcal{R}$ con respuesta computada $\{\mathtt{X} \mapsto \mathtt{0}\}$:

$$\begin{array}{lll} \underline{\mathtt{f(h(X))}} \approx \mathtt{0} & \overset{\texttt{NP}}{\leadsto}_{\mathtt{1.1},\mathtt{R_1},\epsilon} & \mathtt{g(\underline{h(X)},h(X))} \approx \mathtt{0} \\ & \overset{\texttt{NP}}{\leadsto}_{\mathtt{1.1.1},\mathtt{R_3},\{\mathtt{X}\mapsto\mathtt{0}\}} & \underline{\mathtt{g(0,h(0))}} \approx \mathtt{0} \\ & \overset{\texttt{NP}}{\leadsto}_{\mathtt{1.1},\mathtt{R_2},\epsilon} & \underline{\mathtt{0} \approx \mathtt{0}} \\ & \overset{\texttt{NP}}{\leadsto} & \mathtt{true} \end{array}$$

Por otra parte, la única derivación perezosa $\mathcal{D}'$ para el mismo objetivo y con la misma respuesta computada en $\mathcal{R}'$ es:

$$\begin{array}{lll} \mathtt{f(\underline{h(X)})} \approx \mathtt{0} & \overset{\texttt{NP}}{\leadsto}_{\mathtt{1.1.1},\mathtt{R_3},\{\mathtt{x}\mapsto\mathtt{0}\}} & \underline{\mathtt{f(0)}} \approx \mathtt{0} \\ & \overset{\texttt{NP}}{\leadsto}_{\mathtt{1.1},\mathtt{R_4},\epsilon} & \underline{\mathtt{0} \approx \mathtt{0}} \\ & \overset{\texttt{NP}}{\leadsto} & \mathtt{true} \end{array}$$

Obsérvese que no existe relación entre los pasos de ambas derivaciones (a pesar de que las dos obtienen los mismos resultados). Sin embargo, si "expandimos" la segunda derivación, sustituyendo el uso de la regla desplegada R_4 de $\mathcal{R}'$ por las reglas originales R_1 y R_2 de $\mathcal{R}$, tenemos la siguiente derivación (no perezosa) $\mathcal{D}''$:

$$\begin{array}{lll} \mathtt{f(\underline{h(X)})} \approx \mathtt{0} & \leadsto_{\mathtt{1.1.1},\mathtt{R_3},\{\mathtt{X}\mapsto\mathtt{0}\}} & \underline{\mathtt{f(0)}} \approx \mathtt{0} \\ & \leadsto_{\mathtt{1.1},\mathtt{R_1},\epsilon} & \underline{\mathtt{g(0,0)}} \approx \mathtt{0} \\ & \leadsto_{\mathtt{1.1},\mathtt{R_2},\epsilon} & \underline{\mathtt{0} \approx \mathtt{0}} \\ & \leadsto & \mathtt{true} \end{array}$$

Esta derivación tiene bastante semejanza con $\mathcal{D}'$, ya que el primer y último paso en ambas derivaciones es el mismo, y los dos pasos centrales en $\mathcal{D}''$ que se dan con las reglas R_1 y R_2 respectivamente, se simulan perfectamente en $\mathcal{D}'$ con la regla R_4 que proviene del desplegado de R_1 con respecto a R_2. A su vez, $\mathcal{D}''$ puede verse como una versión reordenada (no necesariamente perezosa) de $\mathcal{D}$, donde simplemente se ha intercambiado el orden de sus dos primeros pasos. Finalmente haremos notar un último detalle: curiosamente, a pesar de que la derivación reordenada $\mathcal{D}''$ no es una derivación perezosa en $\mathcal{R}$, sí es simulada perezosamente por $\mathcal{D}'$ en $\mathcal{R}'$.

En lo que sigue, nuestro objetivo será simular, en los programas desplegados, versiones posiblemente reordenadas de derivaciones que tienen éxito en los programas originales. El siguiente par de proposiciones auxiliares son necesarias para conmutar pasos en derivaciones perezosas sin perder respuestas computadas. En primer lugar, comenzamos reescribiendo un interesante resultado original de Réty [1987] que nos permitirá conmutar pasos de *narrowing* (sin restricciones) dentro de una derivación que cumple una serie de restricciones. En la siguiente proposición utilizaremos la notación $s_0 \rightarrow^*_{[q_1,\ldots,q_n,R]} s_n$ para referirnos a una secuencia de n pasos de reescritura que se dan utilizando la misma regla R sobre subtérminos idénticos asociados a una serie de posiciones disjuntas $\{q_1, ..., q_n\}$ de un término inicial s_0 hasta obtener s_n, es decir, $s_0 \rightarrow_{[q_1,R]} s_1 \rightarrow \ldots \rightarrow_{[q_n,R]} s_n$.

Proposición 3.4.3 [Réty, 1987] *Sea $\mathcal{R}$ un SRT y sean*

$$s \leadsto_{p,R,\sigma} t \leadsto_{q',R',\theta} s'' \qquad (1)$$

dos pasos de narrowing dados a partir de un término s tal que:

1. *q' admite al menos un antecedente en s (que denotamos con $q_0, ..., q_{m-1}$),*
2. *todo antecedente de q' es una posición no variable de s,*
3. *$\mathcal{V}ar(l') = \mathcal{V}ar(r')$ o l es lineal.*

Entonces (1) puede ser conmutado por:

$$s \leadsto_{q_0,R',\theta'_0} t'_1 \leadsto. \;...... \leadsto_{q_{m-1},R',\theta'_{m-1}} t'_m \leadsto_{[p,r,\sigma'} s' \qquad (2)$$

tal que

- $\theta'_0 ...... \theta'_{m-1}\sigma' = \sigma\theta\ [\mathcal{V}ar(s)]$
- $s'' \rightarrow^*_{[q'_0,\ldots,q'_n,R']} s'$ *donde $q'_0, ..., q'_n$ son los hermanos de q', es decir, los residuos o descendientes de $q_0, ..., q_{m-1}$ en t.*

Por comodidad, a continuación mostramos una versión adaptada a nuestro caso de la proposición anterior.

Lema 3.4.4 (lema de conmutación) *Sea $\mathcal{R}$ un programa CB y ortogonal y sean $R, R' \ll \mathcal{R}$ dos reglas del mismo. Sea s un término y $p, q \in \mathcal{FP}os(s)$ dos de sus posiciones no variables tal que $p \leq q$ y $p \neq q$. Si $s \leadsto_{p,R,\sigma} t \leadsto_{q',R',\theta} s'' \rightarrow^*_{[q'_0,\ldots,q'_n,R']} s'$ donde q' tiene como antecedente a q en s, y como hermanos a $q'_0, ..., q'_n$ en t, entonces $s \leadsto_{q,R',\theta'} t' \leadsto_{p,R,\sigma'} s'$, tal que $\sigma\theta = \theta'\sigma'$ $[\mathcal{V}ar(s)]$.*

Demostración. La demostración de nuestro lema de conmutación consiste simplemente en comprobar que es un caso particular de la Proposición 3.4.3 ya que se cumplen todas sus condiciones. En efecto:

1. por definición q es un antecedente de q' en s, y además es el único, ya que $\mathcal{R}$ es un programa ortogonal y por tanto lineal por la izquierda,

2. q es una ocurrencia no variable de s ya que $q \in \mathcal{FP}os(s)$,

3. y finalmente l es lineal, ya que $\mathcal{R}$ es lineal por la izquierda.

De esta forma,

$$s \leadsto_{p,R,\sigma} t \leadsto_{q',R',\theta} s''$$

puede conmutarse como:

$$s \leadsto_{q,R',\theta'} t' \leadsto_{p,R,\sigma'} s'$$

tal que

- $\theta'\sigma' = \sigma\theta \ [\mathcal{V}ar(s)]$
- $s'' \to^*_{[q'_0,\ldots,q'_n,R']} s'$ donde $q'_0, ..., q'_n$ son los hermanos de q'.

  O lo que es lo mismo:

  $$s \leadsto_{p,R,\sigma} t \leadsto_{q',R',\theta} s'' \to^*_{[q'_0,\ldots,q'_n,R']} s'$$

  puede conmutarse como:

  $$s \leadsto_{q,R',\theta'} t' \leadsto_{p,R,\sigma'} s'$$

  tal que $\sigma\theta = \theta'\sigma' \ [\mathcal{V}ar(s)]$, como queríamos demostrar.

□

La siguiente definición nos ayudará a detectar en qué puntos de una derivación perezosa será necesario conmutar pasos para obtener una derivación (no perezosa) fácilmente simulable en programas desplegados.

Definición 3.4.5 (pasos críticos) Decimos que dos pasos consecutivos que parten de un término s en una derivación perezosa $\mathcal{D}$ donde se usan reglas de un programa $\mathcal{R}$ son un *par de pasos críticos con respecto a una regla* $R \in \mathcal{R}$ si se verifican las siguientes condiciones:

1. el primer paso se da con la regla R y

2. el segundo paso se da sobre una posición que admite un antecedente en s.

Intuitivamente, un par de pasos críticos es un par de pasos consecutivos en una derivación de *narrowing*, donde el primero se da con una regla concreta y fijada R (que será la que se va a desplegar) y posteriormente el siguiente paso reduce algún descendiente de una posición más interna a la explotada en el paso anterior.

Para poder conmutar un par de pasos críticos será necesario aplicar pasos de reescritura sobre los hermanos (si existen) de la ocurrencia desplegada en el segundo paso del par. Como ya hemos mencionado anteriormente, esta intercalación de pasos de reescritura en una derivación perezosa tiene la completitud y la corrección aseguradas siempre que se trabaje con programas CB y ortogonales [Hanus, 1994a, 1997], como ocurre en el siguiente ejemplo.

Ejemplo 20 Volvamos al programa del Ejemplo 19 y consideremos de nuevo la siguiente derivación $\mathcal{D}$ para el objetivo $\mathtt{f(h(X))} \approx \mathtt{0}$ en $\mathcal{R}$ con respuesta computada $\{\mathtt{X} \mapsto \mathtt{0}\}$:

$$
\begin{array}{lll}
\underline{\mathtt{f(h(X))}} \approx \mathtt{0} & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1,R_1,\epsilon}} & \mathtt{g(\underline{h(X)},h(X))} \approx \mathtt{0} \\
 & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1.1,R_3,\{X\mapsto 0\}}} & \underline{\mathtt{g(0,h(0))}} \approx \mathtt{0} \\
 & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1,R_2,\epsilon}} & \underline{\mathtt{0} \approx \mathtt{0}} \\
 & \overset{\mathtt{NP}}{\leadsto} & \mathtt{true}
\end{array}
$$

En primer lugar, se observa que los dos primeros pasos de $\mathcal{D}$ son un par de pasos críticos con respecto a la regla R_1. Para poderlos conmutar, tendremos que intercalar pasos de reescritura sobre los hermanos de la posición seleccionada en el segundo paso. En este caso concreto, tenemos un solo hermano por reducir, que coincide con el subtérmino $\mathtt{h(0)}$ en $\mathtt{g(0,h(0))}$. Nos queda:

$$
\begin{array}{lll}
\underline{\mathtt{f(h(X))}} \approx \mathtt{0} & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1,R_1,\epsilon}} & \mathtt{g(\underline{h(X)},h(X))} \approx \mathtt{0} \\
 & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1.1,R_3,\{X\mapsto 0\}}} & \mathtt{g(0,\underline{h(0)})} \approx \mathtt{0} \\
 & \rightarrow_{\mathtt{1.1.2,R_3}} & \underline{\mathtt{g(0,0)}} \approx \mathtt{0} \\
 & \overset{\mathtt{NP}}{\leadsto}_{\mathtt{1.1,R_2,\epsilon}} & \underline{\mathtt{0} \approx \mathtt{0}} \\
 & \overset{\mathtt{NP}}{\leadsto} & \mathtt{true}
\end{array}
$$

Al ser $\mathcal{D}$ una derivación perezosa, también es en particular una derivación de *narrowing* sin restricciones, lo que permite aplicar el Lema 3.4.4 ya que $\mathcal{R}$ es un programa CB y ortogonal. Así pues, aplicando el Lema de conmutación 3.4.4 sobre los primeros pasos dados en $\mathcal{D}$ obtenemos una derivación que a pesar de no ser perezosa, sí es de éxito y devuelve la misma respuesta computada que $\mathcal{D}$:

$$
\begin{array}{lll}
\mathtt{f(\underline{h(X)})} \approx \mathtt{0} & \leadsto_{\mathtt{1.1.1,R_3,\{X\mapsto 0\}}} & \underline{\mathtt{f(0)}} \approx \mathtt{0} \\
 & \leadsto_{\mathtt{1.1,R_1,\epsilon}} & \underline{\mathtt{g(0,0)}} \approx \mathtt{0} \\
 & \leadsto_{\mathtt{1.1,R_2,\epsilon}} & \underline{\mathtt{0} \approx \mathtt{0}} \\
 & \leadsto & \mathtt{true}
\end{array}
$$

Obsérvese que esta derivación coincide con la derivación $\mathcal{D}''$ del Ejemplo 19, cuya simulación estaba asegurada con la derivación $\mathcal{D}'$ en el programa desplegado $\mathcal{R}'$. Nótese también que en el mismo ejemplo el programa $\mathcal{R}'$ se obtiene al desplegar la regla R_1 en $\mathcal{R}$. Esta regla coincide justamente con la regla que nos ha servido de referencia para identificar el par de pasos críticos que tras ser conmutados adecuadamente generaban la derivación $\mathcal{D}''$ fácilmente simulable en $\mathcal{R}'$.

La siguiente proposición asegura la existencia de derivaciones comutadas con respecto a una regla de programa concreta partiendo de una derivación original de éxito.

Proposición 3.4.6 *Sea $\mathcal{R}$ un programa CB y ortogonal, $R \in \mathcal{R}$ una regla del mismo y g un objetivo. Si $\mathcal{D} : g \overset{\text{NP}}{\leadsto}{}^{*}_{\sigma}$ true es una derivación perezosa en $\mathcal{R}$ entonces existe una derivación (no necesariamente perezosa) $\mathcal{D}' : g \leadsto^{*}_{\sigma'}$ true en $\mathcal{R}$ tal que:*

- *$\mathcal{D}'$ carece de pares de pasos críticos con respecto a R,*
- *$\mathcal{D}$ y $\mathcal{D}'$ usan las mismas reglas de $\mathcal{R}$ aunque posiblemente en diferente número y orden y*
- *$\sigma' \leq \sigma\ [\mathcal{V}ar(g)]$.*

Demostración. La demostración se basa simplemente en intercalar pasos de reescritura sobre los hermanos de las posiciónes reducidas en los segundos pasos de todos los pares de pasos críticos que aparecen en $\mathcal{D}$ y finalmente conmutarlos mediante la aplicación del Lema 3.4.4. De esta manera desaparecen todos los pares de pasos críticos presentes en $\mathcal{D}$ quedando en $\mathcal{D}'$ pasos que usan las mismas reglas pero que ya no forman pares de pasos críticos porque se ha invertido su orden de aplicación. Finalmente, al ser completa la intercalación de pasos de reescritura en derivaciones perezosas sobre programas CB y ortogonales [Hanus, 1997], sabemos que la derivación reordenada $\mathcal{D}'$ obtiene una respuesta igual o más general que la computada en la derivación original $\mathcal{D}$. $\square$

Antes de entrar en los detalles técnicos que nos permitan demostrar la completitud del desplegado perezoso, pasemos a comentar una última situación de riesgo que se presenta en el caso que estamos considerando. La pregunta surge al plantearnos si el desplegado de un programa ortogonal devuelve siempre otro programa de la misma clase. Esta cuestión está en la línea que ya comentábamos en apartados anteriores dedicados al desplegado de programas terminantes: no sólo estamos interesados en aquellas operaciones de desplegado que se muestren totalmente correctas bajo las mismas (o menos) condiciones que las que garantizan la completitud de la estrategia de *narrowing* en que se base la transformación (como ocurre con *narrowing* impaciente), sino que además nos interesa comprobar en qué grado se propagan estas condiciones a lo largo de sucesivas transformaciones (como es el caso del decrecimiento o el L-cierre en el desplegado basado en el *narrowing* condicional).

El siguiente ejemplo responde negativamente al planteamiento anterior para el caso del desplegado perezoso. La perdida de ortogonalidad durante la transformación tiene serias implicaciones a la hora de garantizar la corrección de las computaciones en los programas transformados y la posibilidad de generar secuencias sucesivas de desplegados. Además, también afecta a nuestros esquemas de demostración de la completitud de la transformación: el desplegado perezoso es correcto y también completo siempre y cuando tanto el programa original como el transformado sean ortogonales.

Ejemplo 21 Consideremos el siguiente programa CB y ortogonal $\mathcal{R}$:

$$\begin{array}{rcll} \texttt{test(X,Y)} & \to & \texttt{h(f(X,g(Y)))} & (R_1) \\ \texttt{f(0,0)} & \to & \texttt{s(f(0,0)} & (R_2) \\ \texttt{f(s(N),X)} & \to & \texttt{s(f(N,X))} & (R_3) \\ \texttt{g(0))} & \to & \texttt{g(0)} & (R_4) \\ \texttt{h(s(X))} & \to & \texttt{0} & (R_5) \end{array}$$

El desplegado perezoso de $\mathcal{R}$ con respecto a su primera regla R_1 devuelve el siguiente programa transformado $\mathcal{R}'$:

$$\begin{array}{rcll} \texttt{test(X,0)} & \to & \texttt{h(f(X,g(0)))} & (R_6) \\ \texttt{test(s(X),Y)} & \to & \texttt{h(s(f(X,g(Y))))} & (R_7) \\ \texttt{f(0,0)} & \to & \texttt{s(f(0,0)} & (R_2) \\ \texttt{f(s(N),X)} & \to & \texttt{s(f(N,X))} & (R_3) \\ \texttt{g(0))} & \to & \texttt{g(0)} & (R_4) \\ \texttt{h(s(X))} & \to & \texttt{0} & (R_5) \end{array}$$

Obsérvese que el programa desplegado no es ortogonal, ya que las partes izquierdas de las cabezas de las nuevas reglas R_6 y R_7 (que son justo las que se han generado por desplegado de la regla R_1) unifican.

Teniendo en cuenta este último problema, en este momento ya estamos en condiciones de demostrar la completitud de la regla de desplegado (bajo la condición de que el programa transformado preserve la ortogonalidad del programa original).

Teorema 3.4.7 (completitud) *Sea $\mathcal{R}$ un SRT CB y ortogonal. Sea $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea g un objetivo. Si $\mathcal{R}'$ es un programa ortogonal y $g \overset{\text{NP}}{\leadsto}{}^*_\theta$ true es una derivación de narrowing perezoso en $\mathcal{R}$ entonces existe una derivación de narrowing perezoso $g \overset{\text{NP}}{\leadsto}{}^*_{\theta'}$ true en $\mathcal{R}'$ tal que $\theta' \leq \theta\ [\mathcal{V}ar(g)]$.*

Demostración. Sea $\mathcal{D} : g \overset{\text{NP}}{\leadsto}{}^*_\sigma$ *true* una derivación perezosa para un objetivo g en el programa CB y ortogonal $\mathcal{R}$, que pretendemos simular en el programa desplegado $\mathcal{R}'$. Por la Proposición 3.4.6 sabemos que existe otra derivación $\mathcal{D}' : g \overset{\text{NP}}{\leadsto}{}^*_\theta$ *true* no perezosa y sin pasos críticos respecto a la regla desplegada R en $\mathcal{R}$ que devuelve una respuesta igual o más general que la computada por $\mathcal{D}$.

Para demostrar el teorema es suficiente con probar que esta segunda derivación $\mathcal{D}'$ puede ser simulada paso a paso en el programa desplegado $\mathcal{R}'$. La demostración se hace por inducción sobre el número de pasos n de la derivación $\mathcal{D}'$.

Sea $n = 1$. Entonces $\mathcal{D}' : (g \leadsto_\theta true)$ en $\mathcal{R}$. En este caso, la única regla que se usa en $\mathcal{D}'$ no puede contener símbolos definidos en su parte derecha. Por tanto, la regla considerada no ha sido desplegada en $\mathcal{R}$ y de esta forma sabemos que también pertenece al programa desplegado, lo que implica que la derivación $\mathcal{D}'$ también se da en $\mathcal{R}'$ como queríamos probar.

Ahora consideremos el caso inductivo donde $n > 1$. En este punto distinguimos dos posibilidades:

- Si el primer paso de $\mathcal{D}'$ se da con una regla distinta a la desplegada, y que por tanto pertenece tanto a $\mathcal{R}$ como a $\mathcal{R}'$, entonces, por la hipótesis de inducción, el resto de la derivación también se verifica en ambos programas.

- En caso contrario, la regla utilizada en el primer paso de $\mathcal{D}'$ no es otra que la regla $R = (l \to r) \ll \mathcal{R}$ que se ha desplegado y que, por tanto, no aparece en $\mathcal{R}'$. De esta forma tenemos que el primer paso de $\mathcal{D}'$ tiene el siguiente aspecto:

  $$g \leadsto_{p,R,\theta} (g[r]_p)\theta$$

  Por la definición 3.4.1, sabemos que r no está encabezado por un símbolo constructor. Por tanto $r\theta$ (en $(g[r]_p)\theta$) tampoco está encabezado por un símbolo constructor. Si de forma similar a [Loogen *et al.*, 1993; Moreno-Navarro y Rodríguez-Artalejo, 1992; Caballero-Roldán *et al.*, 1997] asumimos que todo paso en una derivación perezosa dado sobre una posición demandada de un término concreto siempre contribuye a algún paso posterior en la misma derivación hasta que éste queda encabezado por un símbolo constructor, tenemos que el siguiente paso de $\mathcal{D}'$ se da sobre una posición de la forma $p.p'$, donde p' puede ser una posición vacía (y, en ese caso, coinciden las posiciones reducidas en los dos primeros pasos) o cualquier otra posición (en el caso de que $p.p'$ se refiera a una posición interior en p).

  Además, como no existen pares de pasos críticos en $\mathcal{D}'$ con respecto a R, sabemos que el subtérmino reducido en el segundo paso de la derivación no ha sido introducido por θ en $r\theta$. En otras palabras, el término demandado $((g[r]_p)\theta)|_{p.p'}$ no es otro que $r|_{p'}\theta$. De esta forma, los dos primeros pasos de $\mathcal{D}'$ tienen la siguiente forma:

  $$g \leadsto_{p,R,\theta} (g[r]_p)\theta \leadsto_{p.p',R',\theta'} g'$$

  donde si $R' = (l' \to r') \in \mathcal{R}$ entonces $g' = ((g[r]_p)\theta)[r']_{p.p'}\theta'$. Como $Dom(\theta) \not\in$

$\mathcal{V}ar(r')$, tenemos que $g' = ((g[r]_p)[r']_{p.p'})\theta\theta'$. Simplificando, nos queda que $g' = (g[r[r']_{p'}]_p)\theta\theta'$.

Ahora observemos que si el segundo paso se ha dado sobre el subtérmino $r|_{p'}\theta$ con la regla R' en la derivación considerada, entonces obviamente el término $r|_{p'}$ en r se ha desplegado en R usando R' como regla desplegante para obtener una nueva regla R'' en el programa transformado $\mathcal{R}'$. De esta forma, si observamos la Definición 3.4.1 tenemos que $R'' = l\sigma \rightarrow r[r']_{p'}\sigma$ siendo $\sigma = mgu(\{r|_{p'} = l'\})$ y p' la posición demandada que estamos considerando en r.

Por el Lema 3.2.8, tenemos que los dos primeros pasos de $\mathcal{D}$ pueden simularse en $\mathcal{R}'$ usando la regla R'' de la siguiente forma: $g \quad \leadsto_{p,R'',\sigma'} g'$ donde $\theta\theta' = \sigma'\ [\mathcal{V}ar(g)]$.

Por tanto, podemos concluir diciendo que los dos primeros pasos de $\mathcal{D}'$ dados sobre las posiciones p y $p.p'$ por las reglas $R, R' \in \mathcal{R}$, respectivamente, son perfectamente simulados por un solo paso dado sobre la misma posición p usando la regla $R'' \in \mathcal{R}'$. Es decir, los términos resultantes al aplicar los pasos en ambas derivaciones son los mismos y también lo son las respuestas computadas restringidas a las variables de g.

Por la hipótesis de inducción, el resto de la derivación $\mathcal{D}'$ también puede simularse en el programa desplegado, y por tanto, el resultado se cumple para la derivación completa.

En lo que sigue, llamaremos $\mathcal{D}''$ a la derivación que simula a $\mathcal{D}'$ en $\mathcal{R}'$. Resumiendo todo lo anterior y asumiendo que el programa desplegado $\mathcal{R}'$ es ortogonal, tenemos las siguientes conclusiones:

1. $\mathcal{D} : g \overset{\text{NP}}{\leadsto}{}^{*}_{\sigma}$ *true* es una derivación de *narrowing* perezoso para el objetivo g en el programa original $\mathcal{R}$.

2. $\mathcal{D}' : g \leadsto^*_\theta$ *true* es una derivación de *narrowing* sin restricciones (y, por tanto, no necesariamente perezosa) que carece de pares de pasos críticos (con respecto a la regla desplegada R) para el objetivo g en el programa original $\mathcal{R}$, y tal que $\theta \leq \sigma\ [\mathcal{V}ar(g)]$.

3. $\mathcal{D}'' : g \leadsto^*_\theta$ *true* es una derivación de *narrowing* sin restricciones (y, por tanto, no necesariamente perezosa) para el objetivo g en el programa desplegado $\mathcal{R}'$.

4. Si $\mathcal{R}'$ es un programa CB y ortogonal, entonces, por la completitud de la relación de *narrowing* perezoso en esta clase de programas, existe alguna derivación perezosa $\mathcal{D}''' : g \overset{\text{NP}}{\leadsto}{}^{*}_{\theta'}$ *true* para g en el programa desplegado $\mathcal{R}'$ tal que $\theta' \leq \theta\ [\mathcal{V}ar(g)]$.

Por tanto, podemos concluir que si $\mathcal{D}$ es un derivación perezosa para el objetivo g en $\mathcal{R}$ con respuesta computada σ, entonces seguro que existe otra derivación perezosa $\mathcal{D}'''$ para el mismo objetivo g en $\mathcal{R}'$ con respuesta computada igual o más general θ', o lo que es lo mismo $\theta' \leq \sigma\ [\mathcal{V}ar(g)]$, como queríamos demostrar. $\square$

Como acabamos de ver, el desplegado perezoso disfruta de resultados de corrección y completitud aunque de una forma bastante débil: la relación existente entre las respuestas asociadas a derivaciones de éxito en los programas originales y transformados no es "=" sino "$\leq$", al tiempo que la completitud de la transformación es dependiente del cáracter ortogonal de los programas transformados. Desafortunadamente, como hemos visto en el Ejemplo 21, esto último no siempre se cumple, ya que el desplegado perezoso no siempre es capaz de preservar la ortogonalidad de los programas originales, siendo ésta una condición que se revela suficiente para garantizar ya no sólo la completitud de la propia estrategia de *narrowing* perezoso sino también la completitud de la propia transformación de desplegado perezoso. En cualquier caso, pensamos que sería posible realizar un postproceso de transformación que incluyera un análisis de reglas redundantes capaz de detectar qué reglas pueden ser eliminadas del programa desplegado sin perder corrección ni completitud, de forma que sea posible restituir la ortogonalidad del programa transformado y continuar la secuencia de transformación de forma segura, siguiendo una técnica similar a los así llamados "sistemas de tipos y efectos" (*type and effect systems*) propuestos en [Hanus y Steiner, 1999].

Pese a todo, estos inconvenientes desaparecen al considerar *narrowing* necesario en la definición de desplegado. En el siguiente apartado, comprobaremos que con esta estrategia perezosa refinada, no sólo se obtiene un tipo de desplegado que goza de corrección y completitud fuertes (lo que implica la equivalencia absoluta entre los conjuntos de respuestas computadas asociadas a las derivaciones de éxito en los programas originales y transformados), sino que además preserva la clase natural de programas sobre los que la estrategia necesaria es aplicable e incluso reduce el grado de indeterminismo de algunas computaciones. Por otra parte, basándonos en la equivalencia entre el *narrowing* perezoso y el necesario sobre los así llamados "programas uniformes", en la Sección 3.4.3 derivaremos resultados de completitud más fuertes para el desplegado perezoso sobre esta clase de programas.

3.4.2 Desplegado basado en *narrowing* necesario

Al igual que ocurre con el desplegado perezoso que acabamos de estudiar, el desplegado basado en *narrowing* necesario no puede aplicarse a reglas cuya parte derecha esté encabezada por un símbolo constructor. La siguiente definición introduce nuestra regla de desplegado necesario, de una forma practicamente idéntica a la Definición 3.4.1, es decir, instanciando la definición genérica de desplegado con la estrategia (en este caso) de *narrowing* necesario y teniendo en cuenta la condición de aplicabilidad motivada en el Ejemplo 17.

Definición 3.4.8 (desplegado necesario) Sea $\mathcal{R}$ un SRT y $R = (l \to r) \ll \mathcal{R}$ una regla (renombrada) del mismo, tal que r está encabezado por un símbolo definido. El desplegado perezoso del programa $\mathcal{R}$ con respecto a la regla R es el programa $\mathcal{R}'$ definido como sigue:

$$\mathcal{R}' = (\mathcal{R} - \{R\}) \cup \{(l\theta \to r') \mid r \overset{\text{NN}}{\leadsto}_\theta r'\} \ .$$

El siguiente ejemplo ilustra como el desplegado necesario de un programa requiere que todas las posiciones demandadas de la parte derecha de una regla sean desplegadas si no se quiere perder la completitud de la transformación.

Ejemplo 22 Volvamos de nuevo al Ejemplo 21. Nótese que si el término $\mathtt{f(X,g(Y))}$ de la regla R_1 no es desplegado usando la regla R_3 entonces se pierde la regla R_7 en el programa transformado. De esta forma, el objetivo $\mathtt{test(X,Y)} \approx \mathtt{0}$ admite una derivación de éxito en $\mathcal{R}$ mientras que falla en $\mathcal{R}'$ en el caso de que no se disponga de la regla R_7. En otras palabras, el término $\mathtt{test(X,Y)}$ se reduce a $\mathtt{0}$ siempre y cuando la variable $\mathtt{X}$ se enlace a $\mathtt{s}(\square)$, mientras que su evaluación no termina si $\mathtt{X}$ se enlaza a $\mathtt{0}$ debido a la naturaleza no terminante de $\mathtt{g(0)}$.

A continuación pasamos a establecer una importante propiedad del desplegado necesario que contrasta notoriamente con todos los tipos de desplegado estudiados hasta ahora: el desplegado necesario de un programa inductivamente secuencial produce un programa que sigue perteneciendo a esta misma clase. La propiedad de preservar la estructura original de los programas durante la transformación no sólo es interesante en cualquier tipo de desplegado, sino que también resulta absolutamente imprescindible en el caso actual: el *narrowing* necesario además de ser correcto y completo sobre programas inductivamente secuenciales, se define única y exclusivamente sobre este tipo de programas. Por esta razón, si un programa $\mathcal{R}$ puede ser (correctamente) evaluado mediante *narrowing* necesario, entonces también lo podrá ser cualquiera de sus desplegados, porque cualquiera de ellos es también inductivamente secuencial, como asegura el siguiente resultado.

Teorema 3.4.9 *Sea $\mathcal{R}$ un programa inductivamente secuencial y sea $R \in \mathcal{R}$ una regla del mismo. El resultado de desplegar $\mathcal{R}$ con respecto a R es también un programa inductivamente secuencial.*

DEMOSTRACIÓN. Por la Definición 3.4.8 el programa desplegado $\mathcal{R}'$ se obtiene a partir del programa original $\mathcal{R}$ eliminando de este último una regla $R = (l \to r) \in \mathcal{R}$

y añadiendo al primero un nuevo conjunto de reglas de la forma:

$$\begin{array}{l} l\varphi_1 \to r_1 \\ \vdots \\ l\varphi_m \to r_m \end{array}$$

donde $r \leadsto_{\varphi_i} r_i$, $i = 1, \ldots, m$, son todas las derivaciones de un solo paso de *narrowing* necesario para r en $\mathcal{R}$. Asumamos que l está encabezado con la función definida f, y que f está definido en $\mathcal{R}$ por el conjunto de reglas $\{R_j = (l_j \to r_j) \mid j = 0, \ldots, n, n > 0\}$. Por la hipótesis de inducción, existe un árbol definicional $\mathcal{P}$ para f en $\mathcal{R}$. De esta forma, sabemos que la raíz de $\mathcal{P}$ es el patrón $f(\overline{x_p})$ (donde p es la aridad de f) y $S = \{l_1, \ldots, l_n\}$ es el conjunto de las hojas de $\mathcal{P}$, donde obviamente $l \in S$ y $l \in \mathcal{P}$. Para probar que $\mathcal{R}'$ es inductivamente secuencial, es suficiente con mostrar que existe un árbol definicional $\mathcal{P}'$ para el conjunto

$$S' = (S \backslash \{l\}) \cup \{l\varphi_1, \ldots, l\varphi_m\}\ .$$

Consideremos, para cada paso de *narrowing* condicional $r \leadsto_{\varphi_i} r_i$, su representación canónica asociada $(p, R, \varphi_{i1} \cdots \varphi_{ik_i}) \in \lambda(r, \mathcal{P}_r)$ (donde $\mathcal{P}_r$ es un árbol definicional para la raíz de r). Sea

$$\mathcal{P}' = \mathcal{P} \cup \{l\varphi_{i1} \cdots \varphi_{ij} \mid 1 \leq i \leq m, 1 \leq j \leq k_i\}\ .$$

A continuación probamos que $\mathcal{P}'$ es un árbol definicional para S' mostrando que cada una de las cuatro propiedades de un árbol definicional se cumplen para $\mathcal{P}'$.

Propiedad de la raíz: Los elementos mínimos de ambos árboles definicionales son idénticos, i.e., $pattern(\mathcal{P}) = pattern(\mathcal{P}')$, ya que sólo se han añadido en $\mathcal{P}'$ instancias de una hoja de $\mathcal{P}$.

Propiedad de las hojas: El conjunto de los elementos máximos de $\mathcal{P}$ es S. Como todas las sustituciones computadas por *narrowing* necesario a lo largo de diferentes derivaciones son independientes (por la proposición 2.2.14), las sustituciones $\varphi_1 \cdots \varphi_m$ son independientes entre sí. Así, el reemplazamiento del elemento l en S por el conjunto $\{l\varphi_1, \ldots, l\varphi_m\}$ nunca introduce términos comparables (con respecto al orden de subsumción). Esto implica que S' es el conjunto de los elementos maximales de $\mathcal{P}'$.

Propiedad del padre: Sea $\pi \in \mathcal{P}' \backslash \{pattern(\mathcal{P}')\}$. Consideremos dos casos para π:

1. $\pi \in \mathcal{P}$: Entonces la propiedad del padre se verifica trivialmente ya que únicamente han sido añadidas en $\mathcal{P}'$ instancias de una hoja de $\mathcal{P}$.

2. $\pi \notin \mathcal{P}$: Por definición de $\mathcal{P}'$, $\pi = l\varphi_{i1}\cdots\varphi_{ij}$ para algún $1 \leq i \leq m$ y $1 \leq j \leq k_i$. A continuación, mostramos por inducción en j que la propiedad del padre es cierta para π.
   Caso base ($j = 1$): Entonces $\pi = l\varphi_{i1}$. Por tanto, $\varphi_{i1} \neq \epsilon$ (en otro caso, $\pi = l \in \mathcal{P}$). Así, por la Proposición 2.2.13, $\varphi_{i1} = \{x \mapsto c(\overline{x_n})\}$ con $x \in \mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. Debido a la linealidad del patrón inicial l y de todos los términos sustituidos (cf. Proposición 2.2.13), l tiene una sola ocurrencia o de la variable x y, por tanto, $\pi = l[c(\overline{x_n})]_o$, i.e., l es el único padre de π.
   Paso de inducción ($j > 1$): Asumimos que la propiedad del padre se verifica para $\pi' = l\varphi_{i1}\cdots\varphi_{i,j-1}$. Sea $\varphi_{ij} \neq \epsilon$ (en otro caso, el paso de inducción es trivial). Por la Proposición 2.2.13, $\varphi_{ij} = \{x \mapsto c(\overline{x_n})\}$ con $x \in \mathcal{V}ar(l\varphi_{i1}\cdots\varphi_{i,j-1})$ (ya que $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$). Ahora procedemos como en el caso base para mostrar que π' es el único padre de π.

Propiedad de inducción: Sea $\pi \in \mathcal{P}' \backslash S'$. Consideremos dos casos para π:

1. $\pi \in \mathcal{P}\backslash\{l\}$: Entonces la propiedad de inducción se da para π ya que también se verifica en $\mathcal{P}$ y sólo se han añadido en $\mathcal{P}'$ instancias de l.

2. $\pi = l\varphi_{i1}\cdots\varphi_{ij}$ para algún $1 \leq i \leq m$ y $0 \leq j < k_i$. Asumamos que $\varphi_{i,j+1} \neq \epsilon$ (en otro caso, la demostración es idéntica con la representación $\pi = l\varphi_{i1}\cdots\varphi_{i,j+1}$). Por la Proposición 2.2.13, $\varphi_{i,j+1} = \{x \mapsto c(\overline{x_n})\}$ y π tiene una sola ocurrencia de la variable x (debido a la linealidad del patrón inicial y de todos los términos sustituidos). Por tanto, $\pi' = l\varphi_{i1}\cdots\varphi_{i,j+1}$ es un hijo de π. Consideremos otro hijo $\pi'' = l\varphi_{i'1}\cdots\varphi_{i'j'}$ de π (otros patrones en $\mathcal{P}'$ no pueden ser hijos de π debido a la propiedad de inducción de $\mathcal{P}$). Asumamos que $\varphi_{i'1}\cdots\varphi_{i'j'} \neq \varphi_{i1}\cdots\varphi_{i,j+1}$ (en otro caso, ambos hijos son idénticos). Por el Lema 2.2.14, existe algún k con $\varphi_{i'1}\cdots\varphi_{i'k} = \varphi_{i1}\cdots\varphi_{il}$, $\varphi_{i',k+1} = \{x' \mapsto c'(\cdots)\}$, y $\varphi_{i,k+1} = \{x' \mapsto c''(\cdots)\}$ con $c' \neq c''$. Ya que π'' y π' son hijos de π (i.e., sucesores inmediatos con respecto al orden de subsumción) entonces $x' = x$ (en otro caso, π' difiere de π en más de una posición) y $\varphi_{i',j'} = \cdots = \varphi_{i',k+2} = \epsilon$ (en caso contrario, π'' difiere de π en más de una posición). Así, π' y π'' difieren solo en la instanciación de la variable x que tiene exactamente una ocurrencia en su padre común π, i.e., existe una posición o de π con $\pi|_o = x$ y $\pi' = \pi[c'(\overline{x_{n'_i}})]_o$ y $\pi'' = \pi[c''(\overline{x_{n''_i}})]_o$. Ya que π'' era un hijo arbitrario de π, la propiedad de inducción se mantiene en $\mathcal{P}'$.

□

El resto de la sección está dedicada a la demostración de la corrección y completitud fuertes del desplegado necesario. Debido a que la mayor parte de los resultados que aparecen en la literatura especializada sobre reordenamiento y conmutación de pasos dentro de una derivación de *narrowing* no son aplicables a la estrategia necesaria (lo que se debe fundamentalmente al hecho de que, en este caso, no se generan necesariamente unificadores más generales), en lo que sigue abandonamos el esquema de demostración que hemos utilizado en secciones anteriores. En este sentido, en lo que sigue no intentaremos reordenar derivaciones de éxito dadas en un programa (original o desplegado) para poderlas simular en otro programa (desplegado u original, respectivamente), sino que, muy al contrario, mostraremos las simulaciones en un contexto de reescritura, aprovechando así la abundante cantidad de resultados que existen para la reescritura (necesaria) [Huet y Lévy, 1992; Middeldorp, 1997; Dershowitz y Jouannaud, 1990]. Estableceremos las conexiones entre derivaciones de *narrowing* necesario y reescritura (y viceversa) a través del Lema 3.4.13 presentado originalmente en [Alpuente *et al.*, 1999f], junto con las propiedades de corrección, completitud y minimalidad del *narrowing* necesario enunciadas por Antoy *et al.* [1993]. El elegante esquema de demostración resultante nos permite probar de forma cómoda y fácilmente comprensible las excelentes propiedades de las que disfruta el desplegado necesario de programas inductivamente secuenciales.

Lema 3.4.10 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y sean $R = (l \to r), R' \in \mathcal{R}$ dos reglas del mismo. Sea $r \overset{\text{NN}}{\leadsto}_{R',\sigma} r'$ un paso de narrowing necesario tal que el resultado de desplegar R usando R' en $\mathcal{R}$ es la regla $R'' = (\sigma(l) \to r')$. Si $t \to_{p,R''} t'$ para alguna posición $p \in \mathcal{FP}os(t)$ del término t, entonces $t \to_{p,R} t'' \to_{R'} t'$.*

Demostración. Dado el paso de *narrowing* necesario $r \overset{\text{NN}}{\leadsto}_{R',\sigma} r'$, por la corrección parcial del *narrowing* necesario (punto 1 del Teorema 2.2.15), tenemos que $r\sigma \to_{R'} r'$. Como $t \to_{p,R''} t'$, entonces existe una sustitución θ tal que $l\sigma\theta = t|_p$ y $t' = t[r'\theta]_p$. Como $r\sigma \to_{R'} r'$, por la estabilidad de la reescritura, tenemos que $r\sigma\theta \to_{R'} r'\theta$. Por tanto $t = t[l\sigma\theta]_p \to_{p,R} t[r\sigma\theta]_p \to_{R'} t[r'\theta]_q = t'$, lo que concluye la demostración. □

El siguiente resultado avanza un tipo de corrección del desplegado necesario, pero restringido a derivaciones de reescritura.

Lema 3.4.11 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Si $e \to^* true$ en $\mathcal{R}'$ entonces $e \to^* true$ en $\mathcal{R}$.*

Demostración. Demostramos este resultado por inducción sobre el número n de pasos de reescritura en la derivación $\mathcal{D} = [e \to^* true]$.

Caso base. Sea $n = 0$. Entonces tenemos que $e = true$ lo que prueba el lema trivialmente.

Caso inductivo. Si $n > 0$, tenemos que $\mathcal{D}$ no es una secuencia de reducciones vacía. Si el primer paso de reducción se da con una regla que pertenece tanto a $\mathcal{R}$ como a $\mathcal{R}'$, entonces, por la hipótesis de inducción, la secuencia de reducciones completa puede simularse en $\mathcal{R}$.

En caso contrario, la regla usada en el primer paso de reducción de $\mathcal{D}$ es el resultado de una operación de desplegado. Por tanto, existen dos reglas $R = (l \to r), R' \in \mathcal{R}$ de forma que $r \overset{\text{NN}}{\leadsto}_{R',\sigma} r'$ es un paso de *narrowing* necesario con respecto a $\mathcal{R}$ tal que el resultado de desplegar R usando R' en $\mathcal{R}$ es la regla $R'' = (l\sigma \to r') \in \mathcal{R}'$. Como R'' es la regla usada en el primer paso de $\mathcal{D}$, entonces tenemos que

$$\mathcal{D} = [e \to_{p,R''} e[r'\theta]_p \to^* true] \ .$$

Por el Teorema 3.4.9, sabemos que $\mathcal{R}'$ es inductivamente secuencial. Así, ya que se cumplen todas las condiciones requeridas en el Lema 3.4.10, tenemos que el primer paso de reducción en $\mathcal{D}$ puede simularse por medio de dos pasos de reducción dados con R y R' en $\mathcal{R}$ y, por la hipótesis de inducción, el resto de la derivación también se simula en $\mathcal{R}$. Por tanto existe una secuencia de reducciones desde e hasta *true* en $\mathcal{R}$, como queríamos demostrar.

□

El lema anterior nos permite establecer la equivalencia entre secuencias de reescritura dadas en el programa transformado con otras generadas en el original. A su vez, las relaciones entre derivaciones de *narrowing* necesario y reescritura (en uno y otro sentido) dentro de un mismo programa inductivamente secuencial vienen aseguradas por la corrección y completitud de la estrategia necesaria. Utilizando todas estas conexiones, podemos demostrar la corrección del desplegado necesario como sigue.

Teorema 3.4.12 (corrección) *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Si $e \overset{\text{NN}}{\leadsto}^*_\theta$ true es una derivación de narrowing necesario en $\mathcal{R}'$, entonces existe una derivación de narrowing necesario $e \overset{\text{NN}}{\leadsto}^*_{\theta'}$ true en $\mathcal{R}$ tal que $\theta' \leq \theta$ $[\mathcal{V}ar(e)]$.*

Demostración. Como $e \overset{\text{NN}}{\leadsto}^*_\theta$ *true* en $\mathcal{R}'$ y además $\mathcal{R}'$ es inductivamente secuencial (Teorema 3.4.9), por la corrección parcial del *narrowing* necesario (punto 1 del Teorema 2.2.15), tenemos que $e\theta \to^* true$ en $\mathcal{R}'$. Por el Lema 3.4.11, existe una secuencia de reescritura $e\theta \to^* true$ en $\mathcal{R}$. Además, por la completitud del *narrowing* necesario (punto 2 del Teorema 2.2.15), existe una derivación de *narrowing* necesario $e \overset{\text{NN}}{\leadsto}^*_{\theta'}$ *true* en $\mathcal{R}$ tal que $\theta' \leq \theta$ $[\mathcal{V}ar(e)]$, lo que completa la demostración. □

A continuación, procedemos a demostrar la completitud del desplegado necesario. En lo que sigue consideraremos secuencias de reescritura *necesaria más externa* ("outermost needed"), según definen este concepto Antoy *et al.* [1993]. Básicamente, esta noción se refiere a un tipo de reescritura necesaria donde los redexes (necesarios) explotados son siempre los más externos ("outermost needed redexes"), i.e., no contenidos por ningún otro redex necesario[1]. Es importante hacer notar el hecho de que no siempre todo redex que además de ser necesario sea más externo lo es también en el sentido de Antoy *et al.* [1993]. Por ejemplo, si consideramos un SRT con una regla $R = (\mathtt{f(X,a)} \to \mathtt{X})$ y un término $t = \mathtt{f}(\Delta, \Delta')$ donde Δ y Δ' son dos redexes cualesquiera, entonces observamos que ambos redexes son necesarios y más externos en t, pero únicamente Δ' es necesario más externo (*outermost-needed*) en el sentido de Antoy *et al.* [1993], ya que es el que inicialmente aparece "demandado" por la regla R. En lo que sigue, cuando nos refiramos a redexes necesarios más externos estaremos utilizando implícitamente la noción de Antoy *et al.* [1993].

A continuación recordamos el siguiente resultado original de Alpuente *et al.* [1999f], cuyo uso será crucial para establecer la conexión deseada entre las derivaciones de *narrowing* necesario y las secuencias de reducciones de reescritura necesaria (más externa).

Teorema 3.4.13 [Alpuente *et al.*, 1999f] *Sea $\mathcal{R}$ un SRT inductivamente secuencial. Sea σ una sustitución y V un conjunto finito de variables. Sea s un término encabezado por un símbolo de función definido y $\mathcal{V}ar(s) \subseteq V$. Sea $s\sigma \to_{p_1,R_1} \cdots \to_{p_n,R_n} t$ una secuencia de reescritura necesaria más externa tal que, para todo redex necesario $s\sigma|_p$ de $s\sigma$, $p \in \mathcal{FP}os(s)$. Entonces existe una derivación de narrowing necesario $s \overset{\text{NN}}{\leadsto}_{p_1,R_1,\sigma_1} \cdots \overset{\text{NN}}{\leadsto}_{p_n,R_n,\sigma_n} t'$ y una sustitución σ' tal que $t'\sigma' = t$ y $\sigma_1 \cdots \sigma_n \sigma' = \sigma$ $[V]$.*

El siguiente lema auxiliar es también original de Alpuente *et al.* [1999f], y utiliza la noción de *redex necesario de raíz*. Un redex dentro de un término t se dice *necesario de raíz* si él o uno de sus descendientes debe ser reducido en cualquier derivación que partiendo de t obtiene una forma normal en cabeza.

Lema 3.4.14 [Alpuente *et al.*, 1999f] *Sea $\mathcal{R}$ un SRT inductivamente secuencial y t un término. Si s es un subtérmino de t encabezado por un símbolo de función definido que contiene un redex necesario de raíz en t, entonces todo redex necesario más externo de s es necesario de raíz en t.*

Apoyándonos en el lema anterior, podemos demostrar el siguiente resultado que nos permitirá aplicar el Teorema 3.4.13 en la demostración de la completitud del desplegado necesario.

[1] De forma análoga se definen los conceptos de redex necesario más interno y reescritura necesaria más interna, aunque ahora los redexes necesarios más externos vienen determinados por árboles definicionales, lo cual es fundamental para aplicar el lema 3.4.13 usando *narrowing* necesario.

Lema 3.4.15 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y t un término. Sea $t|_p = \sigma(l)$ un redex necesario de raíz más interno en t, donde $R = (l \to r) \in \mathcal{R}$, r es un término encabezado por un símbolo de función definido y $p \in \mathcal{P}os(t)$. Dado el paso de reescritura $A = [t \to_{p,R} t[r\sigma]_p]$, para todo redex necesario más externo $r\sigma|_q$ de $r\sigma$, tenemos que $q \in \mathcal{FP}os(r)$.*

Demostración. Por contradicción. Asumamos que existe un redex necesario más externo $r\sigma|_q$ de $r\sigma$ y $q \notin \mathcal{FP}os(r)$. Como $r\sigma$ es un término encabezado por un símbolo de función definido y $t|_p$ es un redex necesario de raíz en t, entonces $r\sigma$ debe contener al menos un redex necesario de raíz en t. Por el lema 3.4.14, $r\sigma|_q$ es un redex necesario de raíz en t. Sea $p' \in \mathcal{P}os(t)$ una posición tal que $q \in p' \backslash A$, es decir, p' es un *antecedente* de q con respecto a A. Entonces el subtérmino $t|_{p'}$, $p < p'$, debe ser un redex necesario de raíz en t, lo que contradice la hipótesis de que $t|_p$ es un un redex necesario de raíz más interno en t. □

El siguiente resultado es la clave para demostrar la completitud del desplegado necesario, ya que demuestra un tipo de completitud restringido a derivaciones de reescritura. Básicamente, el lema establece que toda secuencia de reducciones para una ecuación en un programa puede simularse en el programa desplegado correspondiente.

Lema 3.4.16 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Si $e \to^* true$ en $\mathcal{R}$ entonces $e \to^* true$ en $\mathcal{R}'$.*

Demostración. Sean $B_1, \ldots, B_m$ todas las posibles secuencias de reducciones necesarias de e a $true$ en $\mathcal{R}$, y sea k_i el número de redexes reducidos en B_i, $i = 1, \ldots, m$. A continuación, probamos el lema por inducción sobre el número máximo $n = max(k_1, \ldots, k_m)$ de redexes que son necesarios para reducir e a $true$:

$n = 0$. Este caso es trivial ya que $e = true$.

$n > 0$. Como e contiene al menos un redex necesario, entonces existe una regla $R = (l \to r) \in \mathcal{R}$ tal que $e|_p = l\theta$ es un redex necesario más interno de e. Consideremos la siguiente secuencia de reducciones

$$e \to_{p,R} e[r\theta]_p \to^* true$$

en $\mathcal{R}$. Ahora, si R pertenece a ambos programas, $\mathcal{R}$ y $\mathcal{R}'$, entonces el resultado se verifica por la hipótesis de inducción.

En caso contrario, R ha sido desplegada en $\mathcal{R}$. Así, por la Definición 3.4.8, la parte derecha r de la regla R no está encabezada por un símbolo constructor y, por tanto, $r\theta$ es un término encabezado por un símbolo de función definido. Sea

$r\theta|_q$, $q \in \mathcal{P}os(r\theta)$, el redex necesario más externo de $r\theta$. Por el Lema 3.4.15, tenemos que $q \in \mathcal{P}os(r)$. Ahora, consideremos un paso de reescritura arbitrario $r\theta \to_{q,R'} r'$, $R' \in \mathcal{R}_i$, que reduce el redex necesario más externo de r. Entonces, por el Teorema 3.4.13, el paso de *narrowing* necesario $r \overset{\text{NN}}{\leadsto}_{q,R',\sigma} r''$ puede darse en $\mathcal{R}$, existiendo una sustitución σ' tal que $r''\sigma' = r'$ y $\sigma\sigma' = \theta\ [\mathcal{V}ar(r)]$. Por la Definición 3.4.8, la regla $R'' = (l\sigma \to r'')$ pertenece a $\mathcal{R}'$ (i.e., es el resultado de desplegar la regla R usando R').

Consideremos ahora la secuencia de reducciones

$$e \to_{p,R} e[r\theta]_p \to_{p.q,R'} e[r']_p \to^* true$$

en $\mathcal{R}$. Finalmente, como el siguiente paso puede darse en $\mathcal{R}'$ usando la regla desplegada R'':

$$e = e[l\theta]_p = e[l\sigma\sigma']_p \to_{p,R''} e[r''\sigma']_p = e[r']_p$$

el lema se verifica con tan sólo aplicar la hipótesis de inducción sobre la subsecuencia $e[r']_p \to^* true$ en $\mathcal{R}$ (cuyo número máximo de redexes necesarios reducidos es estrictamente menor que n).

□

Finalmente, la completitud del desplegado necesario es consecuencia directa del lema anterior, junto con la corrección y la completitud del *narrowing* necesario para programas inductivamente secuenciales.

Teorema 3.4.17 (completitud) *Sea $\mathcal{R}$ un SRT inductivamente secuencial. Sea $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Si $e \overset{\text{NN}}{\leadsto}{}^*_\theta$ true es una derivación de narrowing necesario en $\mathcal{R}$ entonces existe una derivación de narrowing necesario $e \overset{\text{NP}}{\leadsto}{}^*_{\theta'}$ true en $\mathcal{R}'$ tal que $\theta' \leq \theta\ [\mathcal{V}ar(e)]$.*

DEMOSTRACIÓN. Como $e \overset{\text{NN}}{\leadsto}{}^*_\sigma true$ en el programa inductivamente secuencial $\mathcal{R}$, por la corrección parcial del *narrowing* necesario (punto 1 del Teorema 2.2.15), tenemos que $e\theta \to^* true$ en $\mathcal{R}$. Por el Lemma 3.4.16, existe una secuencia de reescritura $e\theta \to^* true$ en $\mathcal{R}'$. Además como $\mathcal{R}'$ es un programa inductivamente secuencial (Teorema 3.4.9), por la completitud del *narrowing* necesario (punto 2 del Teorema 2.2.15), existe una derivación de *narrowing* necesario $e \overset{\text{NN}}{\leadsto}{}^*_{\theta'} true$ en $\mathcal{R}'$ tal que $\theta' \leq \theta\ [\mathcal{V}ar(e)]$.
□

Finalmente, la corrección y completitud fuertes de la transformación puede probarse fácilmente como una consecuencia directa de los Teoremas 3.4.12 y 3.4.17, junto con la independencia de las soluciones computadas por *narrowing* necesario.

Teorema 3.4.18 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un SRT inductivamente secuencial. Sea $R \in \mathcal{R}$ una regla del mismo tal que el resultado de desplegar $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Entonces, $e \overset{\texttt{NN}}{\leadsto}{}^*_{\theta}$ true es una derivación de narrowing necesario en $\mathcal{R}$ si y solo si existe una derivación de narrowing necesario $e \overset{\texttt{NP}}{\leadsto}{}^*_{\theta'}$ true en $\mathcal{R}'$ tal que $\theta' = \theta\ [\mathcal{V}ar(e)]$.*

DEMOSTRACIÓN. Consideremos separadamente las dos implicaciones del teorema:

($\Leftarrow$) Corrección fuerte. Demostramos el resultado por contradicción. Asumamos que existe una sustitución θ' calculada por *narrowing* necesario para e en $\mathcal{R}'$ tal que no existe ninguna sustitución σ computada por *narrowing* necesario para e en $\mathcal{R}$ tal que $\sigma = \theta'\ [\mathcal{V}ar(e)]$.

Por el Teorema 3.4.12 (corrección del desplegado) y la consideración anterior, concluímos que debe existir alguna sustitución θ computada por *narrowing* necesario para e en $\mathcal{R}$ tal que $\theta < \theta'\ [\mathcal{V}ar(e)]$. Entonces, por el Teorema 3.4.17, existe una sustitución σ' computada por *narrowing* necesario para e en $\mathcal{R}'$ tal que $\sigma' \leq \theta\ [\mathcal{V}ar(e)]$. Como $\sigma' \leq \theta\ [\mathcal{V}ar(e)]$ y $\theta < \theta'\ [\mathcal{V}ar(e)]$, tenemos que $\sigma' < \theta'\ [\mathcal{V}ar(e)]$, lo que contradice la independencia de las soluciones calculadas por *narrowing* necesario (punto 3 del Teorema 2.2.15).

($\Rightarrow$) Completitud fuerte. La demostración en sentido opuesto es perfectamente análoga.

$\square$

3.4.3 Comparación de los desplegados perezosos

Acabamos de ver en las secciones precedentes que los dos tipos de desplegado basados en estrategias perezosas de *narrowing* corren diferentes suertes en cuanto a resultados de corrección se refiere, lo que podemos resumir como sigue:

- Aunque el *narrowing* perezoso es completo para programas CB y ortogonales, el desplegado perezoso de un programa de la misma clase no es siempre ortogonal, y solamente cuando el programa transformado pertenece a dicha clase se obtienen resultados (débiles) de corrección y completitud.

- El *narrowing* necesario es completo sobre programas inductivamente secuenciales[2]. El desplegado necesario de un programa inductivamente secuencial también pertenece a dicho tipo de programas, al tiempo que presenta resultados óptimos de corrección y completitud fuertes.

[2]En Antoy *et al.* [1997] se propone la idea de que el cálculo de *narrowing* necesario puede extenderse a programas *quasi–ortogonales*, pero en este caso se pierden algunas propiedades de optimalidad.

A pesar de que la clase de programas ortogonales incluye a la de los programas inductivamente secuenciales, lo que en apariencia aporta *a priori* un atractivo especial al desplegado perezoso frente al necesario, son muy graves los inconvenientes del primero comparado con el segundo, como acabamos de comentar. Además, tal y como vimos en el capítulo anterior, en general el *narrowing* perezoso tiene un peor comportamiento que el *narrowing* necesario a nivel de derivaciones de éxito (en las que pueden darse pasos a veces innecesarios) y soluciones computadas (que en ocasiones se repiten o quedan subsumidas por otras). Estas deficiencias se propagan también a través de la propia transformación de desplegado, como pasamos a demostrar en el siguiente ejemplo.

Ejemplo 23 Volvamos una vez más al Ejemplo 21, donde vimos que al hacer el desplegado perezoso de la primera regla del programa ortogonal $\mathcal{R}$ obteníamos un programa transformado (que ahora denotamos por $\mathcal{R}'_{NP}$) cuyas dos primeras reglas eran:

$$\begin{array}{rcl} \mathtt{test(X,0)} & \to & \mathtt{h(f(X,g(0)))} \\ \mathtt{test(s(X),Y)} & \to & \mathtt{h(s(f(X,g(Y))))} \end{array}$$

Obsérvese que, como ya hemos avanzado, el programa desplegado no es ortogonal ya que las partes izquierdas de las cabezas de este par de reglas unifican.

Por otra parte, es fácil comprobar que $\mathcal{R}$ no solamente es ortogonal, sino también inductivamente secuencial. De esta forma, el desplegado necesario de $\mathcal{R}$ con respecto a la primera de sus reglas, es el programa transformado $\mathcal{R}'_{NN}$:

$$\begin{array}{rcl} \mathtt{test(0,0)} & \to & \mathtt{h(f(0,g(0)))} \\ \mathtt{test(s(X),Y)} & \to & \mathtt{h(s(f(X,g(Y))))} \\ \mathtt{f(0,0)} & \to & \mathtt{s(f(0,0)} \\ \mathtt{f(s(N),X)} & \to & \mathtt{s(f(N,X))0} \\ \mathtt{g(0)} & \to & \mathtt{g(0)} \\ \mathtt{h(s(X))} & \to & \mathtt{0} \end{array}$$

Como ya sabemos por el Teorema 3.4.9, $\mathcal{R}'_{NN}$ es un programa inductivamente secuencial, lo que permite que sea evaluado con *narrowing* necesario (y también perezoso), a diferencia del programa desplegado por *narrowing* perezoso $\mathcal{R}'_{NP}$ que, al no ser ortogonal, tampoco es inductivamente secuencial, y por tanto no puede ser evaluado con la estrategia necesaria (ni perezosa).

Obsérvese, por otra parte, que las dos primeras reglas de $\mathcal{R}'_{NN}$ muestran que un término de la forma $\mathtt{test(X,Y)}$ puede ser reducido a $\mathtt{0}$ si la variable $\mathtt{X}$ es enlazada a $\mathtt{s(\Box)}$, mientras que su evaluación no termina si $\mathtt{X}$ se enlaza a $\mathtt{0}$, debido a la naturaleza no terminante de $g(0)$.

En contraste con $\mathcal{R}'_{NN}$, el programa $\mathcal{R}'_{NP}$ tiene un peor comportamiento con respecto a la terminación. Por ejemplo, el término $\mathtt{test(s(0),0)}$ tiene un árbol de derivación finito con respecto a $\mathcal{R}'_{NN}$ (usando tanto *narrowing* necesario como perezoso) mientras que el árbol de derivación usando *narrowing* perezoso (ya que el necesario no es aplicable) para el mismo término con respecto a $\mathcal{R}'_{NP}$ es infinito en profundidad. La rama infinita es causada por la aplicación de las reglas $\mathtt{p(X,0)} \to \mathtt{h(f(X,g(0)))}$ y $\mathtt{g(0)} \to \mathtt{g(0)}$.

Más aún, si realizamos un desplegado perezoso posterior sobre la regla $\mathtt{test(X,0)} \to \mathtt{h(f(X,g(0)))}$ de $\mathcal{R}'_{NP}$, obtenemos la siguiente definición para la función `test`, que contiene una regla redundante:

$$\begin{array}{rcl}
\mathtt{test(X,0)} & \to & \mathtt{h(f(X,g(0)))} \\
\mathtt{test(s(X),0)} & \to & \mathtt{h(s(f(X,g(0))))} \\
\mathtt{test(s(X),Y)} & \to & \mathtt{h(s(f(X,g(Y))))}
\end{array}$$

Este último ejemplo muestra que la aplicación del desplegado basado en *narrowing* perezoso a un programa puede destruir las ventajas de las reducciones deterministas en los programas lógico-funcionales: un término que es normalizable de forma determinista con respecto a un programa puede tener una evaluación no determinista en el programa transformado. Sin embargo, esta situación no se plantea en nuestro ejemplo cuando se usa *narrowing* necesario para construir la transformación de desplegado. La siguiente proposición demuestra que esto siempre es así.

Proposición 3.4.19 *Sea $\mathcal{R}$ un programa inductivamente secuencial, y e una ecuación. Sea $\mathcal{R}'$ un programa obtenido por desplegado necesario a partir de $\mathcal{R}$. Si e es normalizable de forma determinista a true con respecto a $\mathcal{R}$, entonces e es normalizable de forma determinista a true con respecto a $\mathcal{R}'$.*

DEMOSTRACIÓN. Como e es normalizable de forma determinista a *true* con respecto a $\mathcal{R}$, existe una derivación de *narrowing* necesario $e \overset{\text{NN}}{\leadsto}{}^{*}_{\epsilon}$ *true* en $\mathcal{R}$. Por el Teorema 3.4.18, existe una derivación de *narrowing* necesario $e \overset{\text{NN}}{\leadsto}{}^{*}_{\sigma}$ *true* en $\mathcal{R}'$ con $\sigma = \epsilon\ [\mathcal{V}ar(e)]$. Esto implica que $\sigma = \epsilon$ por la definición de *narrowing* necesario. Por tanto e es normalizable de forma determinista a *true* con respecto a $\mathcal{R}'$. □

Esta propiedad de los programas transformados por desplegado basado en *narrowing* necesario, es deseable y muy importante desde un punto de vista práctico (a nivel de implementación), ya que la implementación de pasos no deterministas es una operación costosa en los lenguajes con algún tipo de componente lógica.

Además, el indeterminismo adicional en los programas transformados puede provocar derivaciones infinitas adicionales, como se ha visto en el ejemplo anterior. Este hecho puede tener el efecto de que algunas soluciones no puedan ser computables en una implementación secuencial basada en el esquema de vuelta atrás (*backtracking*).

Por tanto, esta propiedad es también deseable en la transformación de programas lógicos puros, aunque no conocemos resultados similares a los nuestros en dicho contexto.

Como el desplegado perezoso de programas ortogonales (e inductivamente secuenciales) genera programas que no tienen porqué pertenecer a dicha clase (corriéndose el riesgo de perder la completitud de la transformación), en lo que sigue pasamos a buscar una clase de programas (más restrictiva incluso que la de los programas inductivamente secuenciales) en la que se preserve la estructura de los programas a lo largo de la transformación, al tiempo que el *narrowing* perezoso sea totalmente correcto con respecto a los programas desplegados perezosamente.

El principal problema del *narrowing* perezoso cuando se usa tanto para evaluar términos como para dirigir la regla de desplegado, es que genera pasos superfluos e innecesarios. Esta es la razón por la que, a menudo, se dice que este tipo de *narrowing* es "menos perezoso" que el *narrowing* necesario. Además, el conjunto de variables instanciadas por un paso de *narrowing* perezoso es igual o menor que aquéllas enlazadas (a términos constructores) por un paso de *narrowing* necesario para un mismo objetivo (lo que implica en ocasiones la pérdida de ortogonalidad, como hemos visto en el Ejemplo 23).

Afortunadamente, existe una clase de programas donde estos problemas del *narrowing* perezoso (en comparación con *narrowing* necesario) desaparecen, ya que las estrategias perezosa y necesaria coinciden sobre dicha clase. Nos estemos refiriendo a los programas *uniformes* [Zartmann, 1997], que son una variedad de los programas inductivamente secuenciales donde aparece como máximo un único constructor en la parte izquierda de cada regla. Un programa es *uniforme* si cada función f está definida por una sola regla cuya parte izquierda es de la forma $f(\overline{x_n})$ o si el conjunto de las partes izquierdas de todas las reglas R_i que definen f tienen la forma $f(\overline{x_k}, c_i(\overline{y_{n_i}}), \overline{z_m})$, donde $\overline{x_k}, \overline{y_{n_i}}, \overline{z_m}$ son variables diferentes entre sí y los constructores c_i son distintos en reglas diferentes. En este último caso, la evaluación de una llamada a f demanda su $(k+1)$-ésimo argumento. Una definición diferente de programas uniformes puede encontrarse en [Kuchen *et al.*, 1990b].

Existe una transformación simple $\mathcal{U}$ de programas inductivamente secuenciales en programas uniformes que se puede consultar en [Zartmann, 1997] y que se basa en el aplanamiento de términos constructores anidados.

Ejemplo 24 Si $\mathcal{R}$ es el programa inductivamente secuencial que define la función "$\leq$" del Ejemplo 7, entonces $\mathcal{U}(\mathcal{R})$ consiste en el siguiente conjunto de reglas:

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

donde $\leq'$ es un símbolo de función nuevo.

El siguiente teorema establece la correspondencia entre las derivaciones de *narrowing* necesario sobre el programa original y las derivaciones de *narrowing* perezoso en el programa uniforme transformado.

Teorema 3.4.20 [Zartmann, 1997] *Sea $\mathcal{R}$ un programa inductivamente secuencial, $\mathcal{U}(\mathcal{R})$ el correspondiente programa uniforme transformado y t un término encabezado por un símbolo de función definido. Entonces existe una derivación de narrowing necesario $t \overset{\text{NN}}{\leadsto}{}^{*}_{\sigma}\, s$ con respecto a $\mathcal{R}$ a un término encabezado por un símbolo constructor s, si y solo si existe una derivación de narrowing perezoso $t \overset{\text{NP}}{\leadsto}{}^{*}_{\sigma}\, s$ con respecto a $\mathcal{U}(\mathcal{R})$.*

El lema anterior nos permite garantizar que, de forma similar a lo que ocurre con el marco de evaluación parcial presentado en [Alpuente *et al.*, 1999f], todo programa obtenido por desplegado necesario puede obtenerse también (posiblemente con más pasos de transformación) por desplegado perezoso del correspondiente programa uniforme. Este hecho muestra de alguna manera que los beneficios obtenidos por el desplegado necesario de un programa no pueden ser peores que los obtenidos por un desplegado perezoso. Por otra parte, existen casos en los que el desplegado perezoso de un programa genera un programa transformado con peor comportamiento que aquellos obtenidos por desplegado necesario (ver, por ejemplo, [Alpuente *et al.*, 1999f], donde algunos ejemplos sobre este tema para programa parcialmente evaluados podrían ser adaptados fácilmente a nuestro contexto de programas desplegados).

El siguiente ejemplo revela que el desplegado perezoso de un programa uniforme no es siempre un programa uniforme.

Ejemplo 25 Sea $\mathcal{R}$ el siguiente programa uniforme:

$$\begin{array}{rcl} \mathtt{f(X,b)} & \to & \mathtt{g(X)} \\ \mathtt{g(a)} & \to & \mathtt{a} \end{array}$$

El desplegado perezoso de $\mathcal{R}$ con respecto a su primera regla devuelve el siguiente programa no uniforme:

$$\begin{array}{rcl} \mathtt{f(a,b)} & \to & \mathtt{a} \\ \mathtt{g(a)} & \to & \mathtt{a} \end{array}$$

Afortunadamente, el programa desplegado en el ejemplo anterior sí es inductivamente secuencial. Esto plantea la cuestión de si el desplegado perezoso de un programa uniforme es siempre inductivamente secuencial. El siguiente corolario responde afirmativamente a la pregunta.

Corolario 3.4.21 *Sea $\mathcal{R}$ un programa uniforme. Si $\mathcal{R}'$ es un desplegado perezoso de $\mathcal{R}$, entonces $\mathcal{R}'$ es inductivamente secuencial.*

DEMOSTRACIÓN. Ya que un programa uniforme es inductivamente secuencial y los pasos de *narrowing* perezoso con respecto a este tipo de programas son también pasos de *narrowing* necesario (cf. como se indica en la demostración del Teorema 3.4.20), el resultado es una consecuencia directa del Teorema 3.4.9. □

De esta forma, podemos garantizar que el desplegado perezoso (y también necesario) de un programa uniforme es totalmente correcto con respecto a las respuestas calculadas por *narrowing* perezoso, ya que también lo es para la estrategia de *narrowing* necesario, que es equivalente a la perezosa para este tipo de programas. Además, a pesar de que el desplegado perezoso de un programa uniforme no siempre cae dentro de la misma clase, el hecho de que el programa transformado sí sea inductivamente secuencial, garantiza que este puede ser transformado nuevamente en uno uniforme, recuperándose así el carácter uniforme del programa original con tan sólo aplicar el método de Zartmann [1997].

Finalmente, ya que la clase de programas uniformes para los que el desplegado perezoso es totalmente correcto, es una subclase de los programas inductivamente secuenciales, podemos concluir que el desplegado necesario es aplicable de forma segura a una clase mayor de programas que el desplegado perezoso, lo que lo hace más atractivo de cara a resultados prácticos[3]. La potencia, optimalidad y amplia aplicabilidad del desplegado necesario se verá incrementada aún más cuando se combine con otras reglas de transformación para construir un sistema completo de transformaciones que iremos construyendo en los capítulos siguientes.

La transformación de desplegado se ha mostrado muy útil desde que fue concebida y descrita por primera vez tanto en contextos funcionales como lógicos puros. Sus aplicaciones se extienden sobre una amplia gama de áreas de trabajo: desde las puramente teóricas (semánticas por desplegado, composicionales, modulares, etc.) hasta otras más aplicadas (síntesis de programas, transformación y/o especialización de programas, etc.). En el resto de este capítulo abordaremos (sin ánimo de ser exhaustivos) algunas de estas aplicaciones en nuestro contexto integrado (lenguajes lógico-funcionales). Básicamente, nos centraremos en el papel fundamental que puede jugar la transformación de desplegado (definida en términos de las diferentes estrategias de *narrowing* estudiadas anteriormente) tanto a nivel de construcción de semánticas por desplegado como a la hora de producir evaluadores parciales y sistemas automáticos de transformación de programas lógico-funcionales.

[3]En [Alpuente *et al.*, 1999c] se puede encontrar un estudio detallado de las relaciones entre *narrowing* perezoso y necesario sobre distintas clases de programas uniformes

3.5 Semánticas por desplegado

A un nivel teórico, el uso de la regla de desplegado como medio para caracterizar la semántica declarativa de los programas lógicos usando técnicas puramente operacionales, aparece por primera vez descrito en [Levi y Mancarella, 1988]. En dicho texto, se evidencia el hecho de que la semántica declarativa de un programa lógico, definida de forma estándar como el modelo mínimo de Herbrand del programa, es incapaz de capturar el observable que mejor refleja la esencia operacional de una computación lógica: las respuestas computadas. A pesar de que se han descrito conceptos más potentes y ambiciosos de semánticas declarativas (la así llamada S-semántica de Falaschi [1988]; Bossi *et al.* [1994]) capaces de reflejar el comportamiento operacional asociado al observable de las respuestas computadas, es en [Levi y Mancarella, 1988] donde por primera vez se presenta la idea novedosa de concebir este tipo de semántica como un programa transformado obtenido por una secuencia (posiblemente infinita) de desplegados. Dicho programa desplegado estaría compuesto únicamente por cláusulas unitarias que se correspondería con la semántica declarativa deseada. Más aún: la semántica por desplegado así construida, puede verse como un programa donde cualquier objetivo puede ser "ejecutado" haciendo uso exclusivamente de unificación sintáctica (sin necesidad de recurrir a la resolución SLD).

En [Bossi *et al.*, 1994] se muestra que las transformaciones basadas o *guiadas* por la semántica de un programa lógico P, entre las que se incluye con especial relevancia el desplegado, suponen una vía alternativa para caracterizar la (S-)semántica asociada al observable respuestas computadas de P. De forma similar a lo que ocurre con la semántica por punto fijo, esta formalización se revela a medio camino entre la semántica declarativa y la operacional de un programa lógico, y sirve no sólo para mostrar de forma sencilla la equivalencia entre éstas, sino también para formalizar distintos tipos de semánticas composicionales que pueden aplicarse incluso a programas abiertos (donde el conjunto de cláusulas que definen un mismo predicado puede estar distribuido en módulos independientes), lo que revierte en distintos tipos de modularidad y paralelismo potencial de este tipo de programas.

En este apartado estudiaremos como construir una semántica por desplegado para programas lógico-funcionales con propiedades similares a las descritas anteriormente. Para ello, y basándonos en [Alpuente *et al.*, 1997c], daremos una definición especial de desplegado que permitirá alcanzar la semántica por desplegado de un programa de forma más rápida que usando las reglas de desplegado basadas en *narrowing* ordinario o impaciente vistas hasta ahora.

Como acabamos de decir, la semántica por desplegado de un programa lógico P supone una forma alternativa de describir el significado del mismo, al tiempo que disfruta de dos importantes propiedades (la primera de carácter semántico y la segunda de estilo más procedural) que podemos resumir así:

1. la semántica por desplegado de P es capaz de reflejar el comportamiento operacional de P asociado al observable de las respuestas computadas, y
2. la semántica por desplegado de P es otro programa (transformado) P' compuesto únicamente por hechos (cláusulas unitarias o sin cuerpo) sobre el que cualquier objetivo puede "evaluarse" usando unificación sintáctica, obteniéndose los mismos resultados que usando la resolución SLD sobre el programa original P.

Intuitivamente, al intentar adaptar estas nociones al esquema de los programas lógico-funcionales, nos encontramos primeramente con que, a nivel sintáctico, este tipo de programas no definen relaciones (predicados) a través de cláusulas sino funciones mediante reglas presentadas por ecuaciones, lo que implica que la semántica que nos interesa construir debe ser un conjunto de reglas incondicionales (sin cuerpo) en vez de un conjunto de hechos. La siguiente cuestión que surge es saber cuál es el tipo de ecuaciones que esperamos encontrar en la semántica por desplegado de un programa lógico-funcional. Esta cuestión puede ser respondida de forma razonable e intuitiva en el contexto de los programas basados en constructores: buscamos una denotación que asocie a cada símbolo de función definido sobre parámetros concretos (compuestos por constructores y variables) su valor (término constructor apropiado). Dado que las estrategias refinadas de *narrowing* que hemos estudiado hasta ahora actúan sobre programas basados en constructores, nuestro interés se centrará en alguna de estas estrategias para definir el tipo de semántica que nos interesa. Al centrarnos en programas canónicos que siguen la disciplina de constructores y son completamente definidos, tenemos que el desplegado impaciente de los mismos es correcto y completo. Además, si definimos la semántica por desplegado de un programa de este tipo como el límite de un proceso reiterado de desplegados impacientes, obtenemos la denotación deseada, compuesta por una serie de ecuaciones donde las partes derechas son términos constructores. Este tipo de construcción no puede hacerse usando *narrowing* ordinario y resulta mucho más enrevesada cuando se utilizan otros refinamientos de *narrowing*. Por ejemplo, es claro que la limitación impuesta en los tipos perezosos de desplegado (basados en *narrowing* perezoso o necesario) de no permitir desplegar términos encabezados por constructores, implica que no siempre será posible obtener reglas incondicionales cuya parte derecha sean términos constructores.

Como una aplicación relevante de la transformación de desplegado impaciente, en lo que sigue definimos una semántica basada en desplegado que es capaz de caracterizar las sustituciones de respuesta computada calculadas sintácticamente por *narrowing* impaciente. Esta semántica se mostrará equivalente a la semántica operacional (basada en la misma estrategia de *narrowing*) que pasamos a definir a continuación.

La semántica operacional de un programa es una aplicación del conjunto de programas sobre el conjunto de "denotaciones" de programas tal que, dado un programa $\mathcal{R}$, devuelve un conjunto de "resultados" de las computaciones en $\mathcal{R}$. Empezaremos formalizando una semántica operacional *no básica* para programas lógico-funcionales que se define en términos del conjunto de todos los "valores" que pueden computar las expresiones funcionales. Esta semántica caracteriza completamente las respuestas computadas por *narrowing* impaciente, al tiempo que admite una formalización equivalente en términos de desplegado impaciente.

Semántica operacional

Comenzamos dando una serie de definiciones auxiliares. Una ecuación de la forma $x = y$, $x, y \in V$ se denomina ecuación *trivial*. Una ecuación *plana* es una ecuación de la forma $f(x_1, \ldots, x_n) = x_{n+1}$ o $x_n = x_{n+1}$, donde $x_i \neq x_j$ para todo $i \neq j$. Cualquier objetivo g puede transformarse en otro equivalente (sin anidamientos funcionales), $flat(g)$, que es plano [Bosco *et al.*, 1988].

Definición 3.5.1 La semántica operacional basada en *narrowing* impaciente de un programa $\mathcal{R}$, $\mathcal{O}^{NI}(\mathcal{R})$ se define como:

$$\mathcal{O}^{NI}(\mathcal{R}) = \{(f(x_1, \ldots, x_n) = x_{n+1})\theta \mid \quad f(x_1, \ldots, x_n) = x_{n+1} \overset{\text{NI}}{\leadsto}{}^{*}_{\theta}\ true \text{ en } \mathcal{R}, \text{ y } f/n \in \Sigma \ \} .$$

Por extensión, la semántica operacional basada en *narrowing* impaciente de un objetivo g con respecto a un programa $\mathcal{R}$, $\mathcal{O}^{NI}_{\mathcal{R}}(g)$ se define como:

$$\mathcal{O}^{NI}_{\mathcal{R}}(g) = \{\theta_{\lceil \mathcal{V}ar(g)} \mid g \overset{\text{NI}}{\leadsto}{}^{*}_{\theta}\ true\} .$$

El siguiente ejemplo ilustra la definición anterior.

Ejemplo 26 Consideremos de nuevo el programa $\mathcal{R}$ del Ejemplo 11. Para obtener $\mathcal{O}^{NI}(\mathcal{R})$ lanzamos los objetivos $\mathtt{0} = \mathtt{Y}$, $\mathtt{s(X)} = \mathtt{Y}$, $\mathtt{nat(X)} = \mathtt{Y}$ e $\mathtt{inc(X)} = \mathtt{Y}$, de tal forma que al aplicarles las correspondientes sustituciones de respuesta computada asociadas a sus derivaciones de éxito en $\mathcal{R}$ tenemos que:

$$\mathcal{O}^{NI}(\mathcal{R}) = \{ \quad \mathtt{0} = \mathtt{0},\ \mathtt{s(X)} = \mathtt{s(X)},\ \mathtt{nat(0)} = \mathtt{true},\ \mathtt{nat(s(0))} = \mathtt{true}, \ldots, \mathtt{inc(0)} = \mathtt{s(0)},\ \mathtt{inc(s(0))} = \mathtt{s(s(0))}, \ldots\} .$$

Por otra parte, si consideramos el objetivo $g = (\mathtt{inc(X)} = \mathtt{s(inc(0))})$, entonces:

$$\mathcal{O}^{NI}_{\mathcal{R}}(g) = \{\{\mathtt{X} \mapsto \mathtt{s(0)}\}\} .$$

El siguiente resultado establece la correspondencia exacta entre la semántica operacional de un objetivo g y la de su forma plana equivalente $flat(g)$.

Lema 3.5.2 *Sea $\mathcal{R}$ un programa canónico y CB-CD y g un objetivo. Entonces,*

$$\mathcal{O}^{NI}_{\mathcal{R}}(g) = \mathcal{O}^{NI}_{\mathcal{R}}(flat(g))\ [\mathcal{V}ar(g)].$$

DEMOSTRACIÓN. El resultado es una consecuencia inmediata del hecho de que el aplanamiento preserva las respuestas computadas para la estrategia de *narrowing* "básico" [Nutt *et al.*, 1989], ya que las derivaciones de *narrowing* impaciente también son básicas. □

El siguiente resultado muestra una propiedad importante de la semántica operacional que estamos considerando [Julián y Moreno, 1996]. Esta propiedad, además de tener interesantes repercusiones prácticas pues indica como componer de forma "paralela" las respuestas de dos subobjetivos para obtener la solución final de un objetivo compuesto, resultará de gran ayuda a la hora de demostrar la equivalencia entre la semántica operacional definida anteriormente y la que obtendremos posteriormente por desplegado.

Teorema 3.5.3 *Sea $\mathcal{R}$ un programa canónico y CB-CD y $g = (g_1, g_2)$ un objetivo. Entonces $\mathcal{O}_{\mathcal{R}}^{NI}(g) = \mathcal{O}_{\mathcal{R}}^{NI}(g_1) \Uparrow \mathcal{O}_{\mathcal{R}}^{NI}(g_2)$.*

DEMOSTRACIÓN. La demostración es perfectamente análoga (aunque más simple) a la del Teorema 4.1 de Alpuente *et al.* [1996a]. En [Alpuente *et al.*, 1996a], la demostración se lleva a cabo considerando *narrowing* condicional básico como semántica operacional, ya que esta estrategia no permite reducir los subtérminos introducidos por la instanciación de variables del objetivo, lo que es la clave para conseguir el resultado de composicionalidad. Cuando se considera *narrowing* impaciente como semántica operacional, esta condición se satisface trivialmente, ya que las sustituciones computadas son siempre constructoras [Fribourg, 1985] y, por tanto, nunca pueden introducir téminos reducibles en los objetivos. □

El siguiente teorema establece que las sustituciones de respuesta computada de cualquier objetivo (posiblemente conjuntivo) g pueden derivarse de $\mathcal{O}^{NI}(\mathcal{R})$ (i.e., del comportamiento observable de las ecuaciones simples) por unificación de las ecuaciones del objetivo con las ecuaciones de la denotación (o semántica operacional del programa). Como ya hemos avanzado, esta propiedad es un tipo de composicionalidad que no se verifica para el *narrowing* condicional [Alpuente *et al.*, 1996a]. Asumimos que las ecuaciones en la denotación están renombradas aparte. Por otra parte, las ecuaciones del objetivo deben ser aplanadas inicialmente, es decir, los subtérminos contenidos en ellas deben ser previamente desanidados para conseguir que la estructura de los términos sea directamente accesible por la unificación. La siguiente definición es auxiliar.

Definición 3.5.4 Sea g un objetivo. Definimos la función $split(g) = (g_1, g_2)$, donde todas las ecuaciones triviales de g están en g_2, y g_1 contiene el resto de ecuaciones (las no triviales) de g.

Haciendo uso de la definición anterior y los resultados preparatorios anteriores, ya podemos enunciar y demostrar el siguiente teorema.

Teorema 3.5.5 *Sea $\mathcal{R}$ un programa canónico CB-CD y g un objetivo. Sea $split(flat(g)) = (g_1, g_2)$. Entonces θ es una sustitución de respuesta computada para g en $\mathcal{R}$ si y sólo si existe $C = (e_1, \ldots, e_m) \ll \mathcal{O}^{NI}(\mathcal{R})$ tal que $\theta' = mgu(g_1, C)$ y $\theta = (\theta' \Uparrow mgu(g_2))\ [\mathcal{V}ar(g)]$.*

DEMOSTRACIÓN. Por el Lema 3.5.2, tenemos que $\theta \in \mathcal{O}^{NI}_{\mathcal{R}}(g)$ si y solo si $\theta \in \mathcal{O}^{NI}_{\mathcal{R}}(flat(g))$. Sea $g_1 = (e_1, \ldots, e_k)$ y $g_2 = (e_{k+1}, \ldots, e_n)$. Por el Lema 3.5.3, $(flat(g) \overset{\texttt{NI}}{\leadsto}{}^*_{\theta}\ true)$ si y sólo si existen las siguientes derivaciones de éxito usando *narrowing* impaciente: $(e_i \overset{\texttt{NI}}{\leadsto}{}^*_{\theta_i}\ true)$, $i = 1, \ldots, n$, tal que $\theta_1 \Uparrow \ldots \Uparrow \theta_n = \theta$. Por la Definición 3.5.1, y el hecho de que $e_1, \ldots, e_k$ son ecuaciones no triviales aplanadas, tenemos que $e_i\theta_i \in \mathcal{O}^{NI}(\mathcal{R})$, $i = 1, \ldots, k$, y $mgu(\{e_1, \ldots, e_k\}, \{e_1\theta_1, \ldots, e_k\theta_k\}) = (\theta_1 \Uparrow \ldots \Uparrow \theta_k)$. Finalmente, como $e_{k+1}, \ldots, e_n$ son ecuaciones triviales, entonces $mgu(\{e_{k+1}, \ldots, e_n\}) = (\theta_{k+1} \Uparrow \ldots \Uparrow \theta_n)$, lo que concluye la demostración. □

El Teorema 3.5.5 muestra que $\mathcal{O}^{NI}(\mathcal{R})$ es un semántica completamente abstracta con respecto a las sustituciones de respuesta computada, i.e., dos programas $\mathcal{R}_1$ y $\mathcal{R}_2$ con $\mathcal{O}^{NI}(\mathcal{R}_1) = \mathcal{O}^{NI}(\mathcal{R}_2)$ (bajo renombramiento) no pueden producir diferentes respuestas computadas. Más aún, $\mathcal{O}^{NI}(\mathcal{R})$ puede verse como un conjunto (posiblemente infinito) de hechos (o reglas incondicionales), y las sustituciones de respuesta computada para g en $\mathcal{R}$ pueden determinarse "ejecutando" $flat(g)$ en el programa $\mathcal{O}^{NI}(\mathcal{R})$ mediante unificación sintáctica, como si el símbolo de igualdad fuese un predicado ordinario.

Puede ser interesante comentar que en Alpuente *et al.* [1996a] y Julián y Moreno [1996] se define e implementa (respectivamente) una semántica operacional similar para *narrowing* condicional básico. Ya que el *narrowing* condicional sin restricciones es composicional con respecto a las respuestas computadas normalizadas [Hamada y Middeldorp, 1997; Middeldorp *et al.*, 1996], lo que es la clave para la demostración de Alpuente *et al.* [1996a], podríamos preguntarnos si los resultados de Alpuente *et al.* [1996a] también se mantienen al considerar *narrowing* condicional sin restricciones con respuestas computadas normalizadas como observable. Desafortunadamente, las sustituciones de respuesta computada por *narrowing* sin restricciones no se preservan a través del aplanamiento, como se muestra en [Alpuente *et al.*, 1998a], lo que concluye negativamente esta cuestión.

Semántica por desplegado

A continuación pasamos a introducir una semántica por desplegado para programas lógico-funcionales, basada en la transformación de desplegado impaciente que hemos definido en la Sección 3.3. Con el objetivo de "agilizar" el proceso de desplegado reiterado de un programa (que nos permita alcanzar su semántica de forma más acelerada), primeramente damos una definición extendida de la regla de desplegado impaciente para un programa lógico-funcional como sigue.

Definición 3.5.6 (desplegado impaciente de un programa) El desplegado impaciente de un programa $\mathcal{R}$ se obtiene por desplegado impaciente de todas las reglas de $\mathcal{R}$ con respecto a $\mathcal{R}$. Formalmente:

$$Unfold^{NI}(\mathcal{R}) = \bigcup_{(l \to r \Leftarrow C) \in \mathcal{R}} \{(l\theta \to r' \Leftarrow C') \mid (C, r = y) \overset{\texttt{NI}}{\leadsto}_\theta (C', r' = y)\}.$$

En la definición anterior se asume implícitamente que toda regla que no admite un desplegado impaciente en $\mathcal{R}$ se mantiene tal cual en el programa transformado. Por otra parte, nótese que la principal diferencia entre esta definición y la Definición 3.3.1 se encuentra en la cantidad de reglas que sufren desplegado impaciente dentro del programa original (todas o sólo una, respectivamente).

Ahora, la aplicación reiterada de este tipo de desplegado conduce a una secuencia de programas equivalentes que se define inductivamente como sigue.

Definición 3.5.7 La secuencia:

$$\begin{array}{lcl} \mathcal{R}^0 & = & \mathcal{R} \\ \mathcal{R}^{i+1} & = & Unfold^{NI}(\mathcal{R}^i), \quad i \geq 0 \end{array}$$

se denomina secuencia de desplegados (impacientes) a partir de $\mathcal{R}$.

El siguiente ejemplo ilustra la definición anterior.

Ejemplo 27 Consideremos una vez más el programa $\mathcal{R}$ del Ejemplo 11. Entonces, los primeros programas de la secuencia de desplegados (impacientes) a partir de $\mathcal{R}$ son[4]:

$$\begin{array}{lrcl} \mathcal{R}^0 = \{ & \texttt{nat(0)} & \to & \texttt{true} \\ & \texttt{nat(s(X))} & \to & \texttt{nat(X)} \\ & \texttt{inc(X)} & \to & \texttt{s(X)} \ \Leftarrow\ \texttt{nat(X)} = \texttt{true}\ \ \}\ . \end{array}$$

$$\begin{array}{lrcl} \mathcal{R}^1 = \{ & \texttt{nat(0)} & \to & \texttt{true} \\ & \texttt{nat(s(0))} & \to & \texttt{true} \\ & \texttt{nat(s(s(X)))} & \to & \texttt{nat(X)} \\ & \texttt{inc(0)} & \to & \texttt{s(0)} \\ & \texttt{inc(s(X))} & \to & \texttt{s(s(X))} \ \Leftarrow\ \texttt{nat(X)} = \texttt{true}\ \ \}\ . \end{array}$$

[4] Nótese que, por simplicidad, hemos simplificado aquellas reglas con ecuaciones que unifican sintácticamente en las condiciones.

$$\begin{array}{lrcl}
\mathcal{R}^2 = \{ & \mathtt{nat(0)} & \rightarrow & \mathtt{true} \\
& \mathtt{nat(s(0))} & \rightarrow & \mathtt{true} \\
& \mathtt{nat(s(s(0)))} & \rightarrow & \mathtt{true} \\
& \mathtt{nat(s(s(s(0))))} & \rightarrow & \mathtt{true} \\
& \mathtt{nat(s(s(s(s(X)))))} & \rightarrow & \mathtt{nat(X)} \\
& \mathtt{inc(0)} & \rightarrow & \mathtt{s(0)} \\
& \mathtt{inc(s(0))} & \rightarrow & \mathtt{s(s(0))} \\
& \mathtt{inc(s(s(0)))} & \rightarrow & \mathtt{s(s(s(0)))} \\
& \mathtt{inc(s(s(s(X))))} & \rightarrow & \mathtt{s(s(s(s(X))))} \ \Leftarrow \ \mathtt{nat(X)} = \mathtt{true} \ \} .
\end{array}$$

Cabe destacar que si interpretamos las reglas incondicionales como ecuaciones y hacemos la unión de todos los programás de la secuencia anterior (lo que equivalentemente se correspondería con el último de ellos si ésta fuese finita) junto con el conjunto $\{0 = 0,\ s(X) = s(X)\}$ obtenemos exactamente el conjunto de ecuaciones $\mathcal{O}^{NI}(\mathcal{R})$.

Obsérvese que, como consecuencia inmediata del Teorema 3.3.2, para programas canónicos y CB-CD tenemos que $\mathcal{O}^{NI}(\mathcal{R}^i) = \mathcal{O}^{NI}(\mathcal{R}^{i+1})$, $i \geq 0$.

La semántica por desplegado de un programa se define como el límite del proceso de desplegado descrito en la Definición 3.5.6. A continuación, pasamos a formalizar este concepto de *semántica por desplegado*, $\mathcal{U}^{NI}(\mathcal{R})$, de un programa $\mathcal{R}$. El principal objetivo de esta definición consiste en generar una denotación compuesta por una serie de ecuaciones donde las partes derechas son términos constructores.

Sea $\Phi_{\mathcal{C}}$ el conjunto de ecuaciones idéntidad $c(x_1, \ldots, x_n) = c(x_1, \ldots, x_n)$, para todo símbolo constructor $c/n \in \mathcal{C}$.

Definición 3.5.8 Sea $\mathcal{R}$ un programa. Entonces,

$$\mathcal{U}^{NI}(\mathcal{R}) = \Phi_{\mathcal{C}} \ \cup \ \bigcup_{i \in \omega} \{(s = d) \mid (s \ \rightarrow \ d \Leftarrow) \in \mathcal{R}^i \ \text{ y } d \in \mathcal{T}(C, \mathcal{X})\}$$

donde $\mathcal{R}^0, \mathcal{R}^1, \ldots$ es la secuencia de desplegados (impacientes) a partir de $\mathcal{R}$.

Las semánticas operacional y por desplegado que acabamos de definir están fuertemente relacionadas ya que ambas se basan en la misma regla de inferencia (que se aplica a ecuaciones planas $f(x_1, \ldots, x_n) = x_{n+1}$ para recoger las respuestas computadas por *narrowing* impaciente o para construir una EP usando la misma estrategia de *narrowing*). Para poder establecer esta estrecha conexión necesitamos el siguiente resultado.

Lema 3.5.9 *Sea $\mathcal{R}$ un programa canónico y CB-CD, y $\mathcal{R}^0, \mathcal{R}^1, \mathcal{R}^2, \ldots$ la secuencia de desplegados (impacientes) a partir de $\mathcal{R}$. Sea $(l_0 \rightarrow r_0 \Leftarrow C_0) \ll \mathcal{R}^0$ una regla tal que existe la siguiente derivación de éxito con narrowing impaciente $(C_0, r_0 = y) \overset{\mathtt{NI}}{\rightsquigarrow}{}^*_\theta$ true en $\mathcal{R}^0$, donde $y \notin (\mathcal{V}ar(l_0))$. Entonces existe un $i \geq 0$ tal que la regla $(l_0 \ \rightarrow \ y)\theta$ pertenece a $\mathcal{R}^i$ y la parte derecha $y\theta \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.*

Demostración. Consideremos la siguiente derivación de éxito por *narrowing* impaciente:

$$(C_0, r_0 = y) \overset{\text{NI}}{\leadsto}_{\theta_1} \ \ldots \ \overset{\text{NI}}{\leadsto}_{\theta_n} (C_n, r_n = y) \ \overset{\text{NI}}{\leadsto}_{\theta_{n+1}} true$$

para $(C_0, r_0 = y)$ en $\mathcal{R}^0$, donde $\theta = \theta_1 \ldots \theta_{n+1}$. Nótese que, por la definición de *narrowing* impaciente, θ es una sustitución constructora [Fribourg, 1985]. A continuación, demostramos el resultado por inducción sobre el número n de pasos de *narrowing* impaciente de esta derivación (excluyendo el paso final de unificación sintáctica).

Sea $n = 0$. Consideremos dos casos: a) si la condición C_0 es $true$, entonces $\theta_1 = \{y \mapsto r_0\}$, y por tanto $(l_0 \to y\theta_1)$ pertenece a $\mathcal{R}^0$, con $y\theta_1 \in \mathcal{T}(\mathcal{C}, \mathcal{X})$; b) si C_0 no es $true$, entonces (por la Definición 3.3.1) tenemos que $(l_0 \to y)\theta_1$ pertenece a $\mathcal{R}^1$, con $y\theta_1 \in \mathcal{T}(\mathcal{C}, \mathcal{X})$.

Consideremos ahora el caso inductivo $n > 0$. Como el paso de *narrowing* impaciente $(C_0, r_0 = y) \overset{\text{NI}}{\leadsto}_{\theta_1} (C_1, r_1 = y)$ puede darse en $\mathcal{R}^0$ entonces, por la Definición 3.3.1, tenemos que $(l_0\theta_1 \to r_1 \Leftarrow C_1)$ pertenece a $\mathcal{R}^1$. Por el Teorema 3.3.2, $\mathcal{O}^{NI}(\mathcal{R}^0) = \mathcal{O}^{NI}(\mathcal{R}^1)$ y como $(C_1, r_1 = y) \overset{\text{NI}}{\leadsto}_{\theta_2} \ \ldots \ \overset{\text{NI}}{\leadsto}_{\theta_{n+1}} true$ en $\mathcal{R}^0$, entonces existe una derivación de éxito con *narrowing* impaciente $(C_1, r_1 = y) \overset{\text{NI}}{\leadsto}^*_{\theta'} true$ para $(C_1, r_1 = y)$ en $\mathcal{R}^1$, tal que $\theta' = \theta_2 \cdots \theta_{n+1}$ $[\mathcal{V}ar(C_1, r_1 = y)]$. Ya que $\mathcal{R}^1, \mathcal{R}^2, \mathcal{R}^3, \ldots$ es la secuencia de desplegados (impacientes) que parte de $\mathcal{R}^1$, por la hipótesis de inducción, existe un $i \geq 1$ tal que la regla $(l_0\theta_1 \to y)\theta'$ pertenece a $\mathcal{R}^i$, con $y\theta' \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. Finalmente, como $(C_0, r_0 = y) \overset{\text{NI}}{\leadsto}_{\theta_1} (C_1, r_1 = y)$ en $\mathcal{R}^0$, entonces tenemos que $(l_0 \to y)\theta_1\theta' = (l_0 \to y)\theta_1 \cdots \theta_{n+1}$ también pertenece a $\mathcal{R}^i$, lo que completa la demostración. $\square$

El siguiente teorema es el principal resultado de esta sección y formaliza el hecho intuitivo de que, ya que la regla de desplegado (impaciente) preserva las propiedades observables de las ejecuciones de un programa (canónico y CB-CD), podemos formular una caracterización interesante y alternativa de la semántica operacional de respuestas computadas $\mathcal{O}^{NI}(\mathcal{R})$ en términos de desplegado impaciente.

Teorema 3.5.10 *Sea $\mathcal{R}$ un programa canónico y CB-CD. Entonces, $\mathcal{U}^{NI}(\mathcal{R}) = \mathcal{O}^{NI}(\mathcal{R})$.*

Demostración. $(\subseteq)$ Sea $(s = d) \in \mathcal{U}^{NI}(\mathcal{R})$. Sea $f/n \in \Sigma$ el símbolo de función más externo de s. Ahora consideremos dos casos diferentes, dependiendo del símbolo de función f:

- Si $f/n \in \mathcal{C}$ entonces, por la Definición 3.5.8, $s = d = f(x_1, \ldots, x_n)$. Ahora, el paso de *narrowing* impaciente $f(x_1, \ldots, x_n) = x_{n+1} \overset{\text{NI}}{\leadsto}_{\sigma} true$ para el objetivo plano $f(x_1,\ldots,x_n) = x_{n+1}$ puede darse con $\sigma = \{x_{n+1} \mapsto f(x_1,\ldots,x_n)\}$. Por tanto, por la Definición 3.5.1, la ecuación $f(x_1,\ldots,x_n) = f(x_1,\ldots,x_n)$ también pertenece a $\mathcal{O}^{NI}(\mathcal{R})$.

- Si $f/n \in \mathcal{F}$, por la Definición 3.5.8, existe $m \in \omega$ tal que $(s \to d \Leftarrow) \in \mathcal{R}^m$. Entonces, podemos construir la siguiente derivación de éxito por *narrowing* impaciente para la ecuación plana $f(x_1,\ldots,x_n) = x_{n+1}$:

$$f(x_1,\ldots,x_n) = x_{n+1} \overset{\text{NI}}{\leadsto}_{R,\theta} d = x_{n+1} \overset{\text{NI}}{\leadsto}_{\sigma} \mathit{true}$$

  donde $R = (s \to d \Leftarrow) \ll \mathcal{R}^m$, $f(x_1,\ldots,x_n)\theta = s$, y $\sigma = \{x_{n+1} \mapsto d\}$. Finalmente, por la Definición 3.5.1, $(f(x_1,\ldots,x_n) = x_{n+1})\theta\sigma = (s = d) \in \mathcal{O}^{NI}(\mathcal{R}^m)$ y como $\mathcal{O}^{NI}(\mathcal{R}) = \mathcal{O}^{NI}(\mathcal{R}^m)$ (por el Teorema 3.3.2), tenemos que $(s = d) \in \mathcal{O}^{NI}(\mathcal{R})$.

($\supseteq$) Sea $(s = d) \in \mathcal{O}^{NI}(\mathcal{R})$. Por la Definición 3.5.1, existe $f/n \in \Sigma$ tal que se verifica la siguiente derivación de éxito por *narrowing* impaciente para la ecuación plana $f(x_1,\ldots,x_n) = x_{n+1}$ en $\mathcal{R}$:

$$f(x_1,\ldots,x_n) = x_{n+1} \overset{\text{NI}}{\leadsto}_{\theta_0} C_0, r_0 = x_{n+1} \overset{\text{NI}}{\leadsto}_{\theta_1} C_1, r_1 = x_{n+1} \overset{\text{NI}}{\leadsto}_{\theta_2} \ldots \overset{\text{NI}}{\leadsto}_{\theta_m} \mathit{true}$$

donde $\theta = \theta_0 \cdots \theta_m$ y $(f(x_1,\ldots,x_n) = x_{n+1})\theta = (s = d)$. Ahora distinguimos dos casos separados:

- Si $m = 0$, entonces $f \in \mathcal{C}$ y por tanto $s = d = f(x_1,\ldots,x_n)$. Por la Definición 3.5.8, la ecuación $f(x_1,\ldots,x_n) = f(x_1,\ldots,x_n)$ pertenece a $\mathcal{U}^{NI}(\mathcal{R})$.

- Si $m > 0$, entonces $f/n \in \mathcal{F}$. Consideremos el primer paso de la derivación:

$$f(x_1,\ldots,x_n) = x_{n+1} \overset{\text{NI}}{\leadsto}_{R_0,\theta_0} C_0, r_0 = x_{n+1}$$

  donde $R_0 = (l_0 \to r_0 \Leftarrow C_0) \ll \mathcal{R}$ y $l_0 = f(x_1,\ldots,x_n)\theta_0$. Por la aplicación del Lema 3.5.9 sobre la subderivación $(C_0, r_0 = x_{n+1}) \overset{\text{NI}}{\leadsto}_{\theta_1} \ldots \overset{\text{NI}}{\leadsto}_{\theta_m} \mathit{true}$, tenemos que $(l_0 \to x_{n+1})\theta_1 \ldots \theta_m$ pertenece a algún $\mathcal{R}^i$, $i \geq 0$, con $x_{n+1}\theta_1 \ldots \theta_m \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. Finalmente, como $l_0 = f(x_1,\ldots,x_n)\theta_0$ y $x_{n+1} \notin \mathit{Dom}(\theta_0)$, por la Definición 3.5.8, tenemos que $(f(x_1,\ldots,x_n) = x_{n+1})\theta_0 \cdots \theta_m = (s = d) \in \mathcal{U}^{NI}(\mathcal{R})$.

$\square$

3.6 Desplegado y evaluación parcial

El objetivo general de las técnicas de transformación de programas consiste en derivar programas "mejores" pero semánticamente equivalentes al programa original. En

[Pettorossi y Proietti, 1996a,b] se indica que, en general, las diferentes técnicas conocidas de transformación de programas declarativos (evaluación parcial, especialización, "reglas + estrategias", deforestación, fusión, formación de tuplas, etc.) presentan puntos comunes (en mayor o menor medida) que se pueden resumir como sigue:

1. *computación simbólica*, expresada fundamentalmente a través de la regla de desplegado,

2. *búsqueda de regularidades* vía desplegado, para posibilitar pasos de plegado posteriores y

3. *extracción del programa final*, lo que puede involucrar procesos de renombramiento, eliminación de reglas redundantes, etc.

La evaluación parcial (EP) es una técnica de transformación de programas que, siguiendo el esquema anterior, consiste en la especialización de un programa respecto a ciertos datos de entrada, conocidos en tiempo de compilación [Futamura, 1971]. El programa resultante puede ser ejecutado más eficientemente que el programa original ya que, usando el conjunto de datos conocidos (parcialmente), es posible evitar algunas computaciones en tiempo de ejecución que se realizarán, una única vez, durante el proceso de compilación. Por otro lado, las técnicas de EP han demostrado también su utilidad para la optimización del proceso de compilación, así como en el campo de la generación automática de compiladores [Jones *et al.*, 1993], permitiendo la obtención de un compilador para el lenguaje a partir de su intérprete (por evaluación parcial del propio evaluador respecto a éste).

Las técnicas convencionales de EP para programas funcionales se basan en la reducción de expresiones y en la propagación de constantes, mientras que las técnicas empleadas para la deducción parcial de programas lógicos explotan la propagación de parámetros basada en unificación. Han habido, hasta la fecha, pocos intentos de estudiar las relaciones entre las técnicas usadas en programación funcional y programación lógica [Glück y Sørensen, 1994]. Un tratamiento unificado del problema que establece las bases para una comparación precisa lo encontramos en [Alpuente *et al.*, 1998a,b] y diferentes refinamientos y extensiones de la EP de programas lógico-funcionales en [Alpuente *et al.*, 1996b, 1997a; Albert *et al.*, 1998a,b; Alpuente *et al.*, 1999f].

Por otra parte, las transformaciones de evaluación parcial y las que se basan en plegado/desplegado se han desarrollado de forma bastante independiente. Recientemente, su relación ha sido objeto de numerosas discusiones [Lloyd, 1987; Levi y Mancarella, 1988; Bossi *et al.*, 1990; Leuschel *et al.*, 1996; Pettorossi y Proietti, 1994; Proietti y Pettorossi, 1993; Seki, 1993]. En esencia, la evaluación parcial de un programa es un subconjunto estricto de las transformaciones basadas en plegado/desplegado, donde el desplegado es la regla de transformación básica que se usa para implementar el núcleo de un evaluador parcial, mientras que el plegado suele emplearse para realizar un postproceso de renombramiento sobre el programa especializado (como veremos en la Sección 4.5). Durante el proceso de EP en sí, sólo se tiene una forma limitada de plegado implícito al imponer la llamada condición de cierre [Pettorossi y Proietti, 1994]. De esta forma, si el desplegado puede considerarse como la base sobre la que se implementa el método de evaluación parcial, en el sentido contrario y siguiendo los argumentos de Lloyd [1987], Levi y Mancarella [1988] y Alpuente *et al.* [1997c], tenemos que el desplegado de un programa se corresponde con un una evaluación parcial del mismo donde no hay especialización. En contrapartida, la evaluación parcial de un programa posee una menor complejidad y una mayor comprensión del control del proceso, lo que la hace más fácilmente implementable y automatizable.

En la literatura sobre el desplegado de programas lógicos, esta forma de transformación sigue un esquema que podríamos llamar "paso a paso", al que nosotros nos hemos adherido a lo largo de todo el capítulo. Como se cita en [Amtoft, 1992; Gardner y Shepherdson, 1991], este método pierde cierto potencial en el sentido de que la transformación parece acelerarse si se permite dar varios pasos de evaluación (en vez de uno solo) sobre el cuerpo de la cláusula a desplegar, mientras se desarrolla la transformación. De hecho, en la literatura se ha propuesto una definición más clara y potente del desplegado basado en evaluación parcial, la cual permite derivar programas equivalentes siguiendo una estrategia (más rápida) que la operación de desplegado "paso a paso" no puede producir [Amtoft, 1992; Gardner y Shepherdson, 1991], salvo que toda la secuencia de programas desplegados intermedios sea almacenada para poder considerar pasos subsiguientes de desplegado [Pettorossi y Proietti, 1994]. En términos sencillos, una operación de desplegado que persiga estos objetivos se basaría en la idea de evaluar parcialmente un programa con respecto a las cabezas de todas sus cláusulas. De acuerdo con este punto de vista y recordando lo que mencionábamos anteriormente, este tipo de desplegado puede verse como una evaluación parcial sin especialización [Levi y Mancarella, 1988; Lloyd y Shepherdson, 1991], es decir, sin restricción del dominio de aplicación de los programas transformados con respecto a una serie de datos de entrada. Este clase de transformación permitiría, por ejemplo, que la cláusula $\mathtt{p(x)} \Leftarrow \mathtt{p(f(x))}$ pudiese ser desplegada contra sí misma, obteniéndose la nueva cláusula $\mathtt{p(x)} \Leftarrow \mathtt{p(f(f(f(x))))}$, lo que no puede conseguirse paso a paso, ya que el primer paso reemplazaría la regla original por $\mathtt{p(x)} \Leftarrow \mathtt{p(f(f(x)))}$ y nunca se tendría después la posibilidad de obtener un número par de símbolos f en el cuerpo de ninguna cláusula desplegada.

Antes de pasar a explotar la relación entre las técnicas de desplegado y evaluación parcial en el contexto integrado de los programas lógico-funcionales, haremos una breve revisión de la técnica de EP en el contexto de los lenguajes lógico-funcionales integrados, tomando como referencia fundamental los textos [Alpuente *et al.*, 1998a,b, 1999g]. Una breve panorámica y recorrido histórico sobre los métodos de EP en lenguajes lógicos y funcionales puros puede encontrarse en [Vidal, 1996].

Evaluación parcial de programas lógico-funcionales

Como ya hemos comentado en el capítulo anterior, el mecanismo de ejecución de los programas lógico-funcionales (*narrowing*) utiliza generalmente la unificación para el paso de parámetros en las llamadas a función, consiguiéndose así de forma automática el efecto de la regla de instanciación, lo que tiene fuertes repercusiones no solo en la operación de desplegado definida en términos de (las diferentes estrategias) de *narrowing*, sino también por extensión en el proceso mismo de la evaluación parcial. La especialización de programas basada en el mecanismo operacional de *narrowing* aporta una visión unificada de los mecanismos de ejecución y transformación, que permite desarrollar un marco simple y efectivo para la EP de programas lógico-funcionales. Siendo *narrowing* un mecanismo operacional bien estudiado, podemos hacer un uso inmediato de sus propiedades, sin tener que partir de cero para caracterizar el comportamiento de los métodos desarrollados (como sucede con los métodos convencionales de EP de los lenguajes funcionales, generalmente definidos de forma *ad-hoc* para realizar la transformación). Más aún, los trabajos de Alpuente *et al.* [1998a,b] demuestran que es posible definir varias optimizaciones del esquema que se deben exclusivamente al mecanismo de computación de los programas lógico-funcionales (como, por ejemplo, la inclusión de un proceso de simplificación determinista durante el proceso de EP). Debido a la posibilidad de explotar su componente funcional, la EP de un programa lógico-funcional puede resultar más efectiva que la EP de un programa lógico equivalente.

En lo que sigue pasamos a detallar nuestra noción de EP para el caso más general de los programas lógico-funcionales condicionales. Para especializar un programa $\mathcal{R}$ con respecto a un término s, obtenemos un árbol (finito) de *narrowing* para el objetivo "artificial" $s = y$, donde y es una variable nueva (i.e., $y \notin \mathcal{V}ar(s)$). La razón para usar este tipo de objetivos es computar los distintos términos a los que se puede reducir por *narrowing* el término s usando el programa $\mathcal{R}$. Así, consideramos derivaciones de la forma:

$$(s = y) \leadsto^*_\theta (e_1, \ldots, e_n, t = y)$$

donde $(t = y)$ es la forma reducida (por *narrowing*) de la ecuación inicial $(s = y)$, y $(e_1, \ldots, e_n)$ son las ecuaciones introducidas por las condiciones de las reglas aplicadas durante la computación. Entonces, definimos el resultante asociado a dicha derivación mediante la regla:

$$s\theta \rightarrow t \Leftarrow e_1, \ldots, e_n.$$

Definición 3.6.1 [Alpuente *et al.*, 1998a] Sea $\mathcal{R}$ un programa y s un término. Sea $\mathcal{D} = [(s = y) \leadsto_\theta^* g]$ una derivación de narrowing condicional para el objetivo ecuacional $(s = y)$, $y \notin \mathcal{V}ar(s)$, con respecto al programa $\mathcal{R}$. Entonces el resultante de la derivación $\mathcal{D}$ es:

1. $(s\theta \ \rightarrow \ t \Leftarrow C)$, si $g = (C, t = y)$; o
2. $(s \ \rightarrow \ y)\theta$, si $g = true$.

Como puede verse, la definición anterior de resultante parece algo más compleja que su contrapartida para programas lógicos. Veamos algunos ejemplos que ilustran esta definición.

Ejemplo 28 Dada la regla $(\mathtt{f(X)} \ \rightarrow \ \mathtt{X} \ \Leftarrow \ \mathtt{a} = \mathtt{X}, \mathtt{b} = \mathtt{X})$ en $\mathcal{R}$, el resultante de la derivación:

$$\begin{array}{ll} \underline{\mathtt{f(f(Z))}} = \mathtt{Y} & \leadsto_{\{\mathtt{X} \mapsto \mathtt{f(Z)}\}} (\mathtt{a} = \mathtt{f(Z)}, \mathtt{b} = \mathtt{f(Z)}, \underline{\mathtt{f(Z)} = \mathtt{Y}}) \\ & \leadsto_{\{\mathtt{Y} \mapsto \mathtt{f(Z)}\}} (\mathtt{a} = \mathtt{f(Z)}, \mathtt{b} = \mathtt{f(Z)}, \mathtt{true}) \end{array}$$

es la regla: $(\mathtt{f(f(Z))} \ \rightarrow \ \mathtt{f(Z)} \ \Leftarrow \ \mathtt{a} = \mathtt{f(Z)}, \mathtt{b} = \mathtt{f(Z)})$.
El resultante asociado a la derivación:

$$\underline{\mathtt{f(Z)}} = \mathtt{Y} \quad \leadsto_{\{\mathtt{Z} \mapsto \mathtt{X}\}} (\mathtt{a} = \mathtt{X}, \mathtt{b} = \mathtt{X}, \mathtt{X} = \mathtt{Y})$$

es la regla: $(\mathtt{f(Y)} \ \rightarrow \ \mathtt{Y} \ \Leftarrow \ \mathtt{a} = \mathtt{Y}, \mathtt{b} = \mathtt{Y})$.
Por último, el resultante asociado a la derivación:

$$\begin{array}{ll} \underline{\mathtt{f(Z)}} = \mathtt{Y} & \leadsto_{\{\mathtt{Z} \mapsto \mathtt{X}\}} (\underline{\mathtt{a} = \mathtt{X}}, \mathtt{b} = \mathtt{X}, \mathtt{X} = \mathtt{Y}) \\ & \leadsto_{\{\mathtt{X} \mapsto \mathtt{a}\}} (\mathtt{true}, \mathtt{b} = \mathtt{a}, \mathtt{a} = \mathtt{Y}) \end{array}$$

es la regla: $(\mathtt{f(a)} \ \rightarrow \ \mathtt{a} \ \Leftarrow \ \mathtt{true}, \mathtt{b} = \mathtt{a})$.

Una vez que se dispone de una definición de resultante adecuada, la formalización del concepto de EP resulta muy simple. La EP de un término s en un programa $\mathcal{R}$ se obtiene construyendo un árbol de búsqueda (posiblemente incompleto)[5] para el objetivo $(s = y)$, y extrayendo posteriormente la definición especializada –los resultantes– asociados a las hojas del árbol. Decimos que un resultante es *trivial* si se ha obtenido aplicando la regla de unificación sintáctica directamente sobre el objetivo inicial. Nótese que, cuando se usa *narrowing* condicional sin restricciones, siempre aparece un resultante trivial en el proceso de especialización ya que, dada la forma del objetivo inicial, la siguiente derivación siempre es posible: $(s = y) \leadsto_{\{y \mapsto s\}} true$, y da lugar al resultante trivial $s \ \rightarrow \ s$, que no debe ser considerado al construir la EP de un objetivo ecuacional, tal y como aparece en la siguiente definición de evaluación parcial.

[5]De forma similar a Lloyd y Shepherdson [1991], consideramos que las derivaciones pueden ser potencialmente incompletas. Una rama del árbol puede ser entonces de fallo, incompleta, de éxito o infinita.

Definición 3.6.2 [Alpuente *et al.*, 1998a] Sea $\mathcal{R}$ un programa, s un término e $y \notin \mathcal{V}ar(s)$ una variable nueva. Sea τ un árbol finito de *narrowing* (posiblemente incompleto) para el objetivo $(s = y)$ en $\mathcal{R}$ conteniendo al menos un nodo distinto de la raíz. Sea $\{h_i \mid i = 1, \ldots, k\}$ el conjunto de hojas no falladas de τ y $\mathcal{R}'=\{r_i\}_{i=1}^m$, $m \leq k$, el conjunto de resultantes no triviales asociados con las derivaciones $\{(s = y) \leadsto^+_{\theta_i} h_i \mid i = 1, \ldots, k\}$. Entonces, el conjunto $\mathcal{R}'$ es una *evaluación parcial de s en $\mathcal{R}$ (usando τ).*

Esta definición se puede extender de manera inmediata a conjuntos de términos y a objetivos ecuacionales arbitrarios. Así, una evaluación parcial de un conjunto finito de términos (módulo variantes) S en un programa $\mathcal{R}$ (o EP de $\mathcal{R}$ con respecto a S) se define como la unión de las evaluaciones parciales de los elementos de S en $\mathcal{R}$. A su vez, la evaluación parcial de un objetivo ecuacional $(s_1 = t_1, \ldots, s_n = t_n)$ en $\mathcal{R}$ consiste en la EP del conjunto de términos $\{s_1, \ldots, s_n, t_1, \ldots, t_n\}$ en $\mathcal{R}$. A continuación recogemos los resultados de corrección y completitud de la evaluación parcial (basada en *narrowing* condicional sin restricciones) que se establecen en Alpuente *et al.* [1998a]. Recordamos primeramente que un término t es *lineal* si ninguna variable aparece repetida en t. Abusando de esta notación, podemos extender este concepto a un conjunto de términos S diciendo que S es *lineal* si todos los términos que lo componen son lineales.

Teorema 3.6.3 (corrección y completitud de la EP)
Sea $\mathcal{R}$ un programa canónico, g un objetivo, S un conjunto finito de términos y $\mathcal{R}'$ una evaluación parcial de $\mathcal{R}$ con respecto a S. Entonces,

1. (CORRECCIÓN) *Si $g \leadsto^*_\theta$ true es una derivación de narrowing condicional en $\mathcal{R}'$ entonces existe una derivación de narrowing condicional $g \leadsto^*_{\theta'}$ true en $\mathcal{R}$ tal que $\theta' \leq \theta$ $[\mathcal{V}ar(g)]$.*

2. (COMPLETITUD) *Si S es lineal y $\mathcal{R}' \cup \{g\}$ es S-cerrado entonces, si $g \leadsto^*_\sigma$ true es una derivación de narrowing condicional en $\mathcal{R}$ entonces existe una derivación de narrowing condicional $g \leadsto^*_{\sigma'}$ true en $\mathcal{R}'$ tal que $\sigma' \leq \sigma$ $[\mathcal{V}ar(g)]$.*

El resultado anterior puede reforzarse añadiendo el prerrequisito de *independencia* basado en el de *solapamiento*) que pasamos a definir a continuación.

Definición 3.6.4 (solapamiento) Un término s solapa a un término t si existe un subtérmino no variable $s|_p$ de s tal que $s|_p$ y t unifican. Si $s = t$, requerimos que t sea unificable con un subtérmino propio no variable de s.

Definición 3.6.5 (independencia) Un conjunto de términos S es independiente, si no contiene términos s y t en S tal que s solapa a t.

Teorema 3.6.6 (corrección fuerte de la EP [Alpuente *et al.*, 1998a]**)**
Sea $\mathcal{R}$ un programa confluente, g un objetivo, y S un conjunto finito e independiente de términos. Sea $\mathcal{R}'$ una evaluación parcial de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}' \cup \{g\}$ es S-cerrado. Si $g \leadsto^*_\theta$ true es una derivación de narrowing condicional en $\mathcal{R}'$ entonces existe una derivación de narrowing condicional $g \leadsto^*_{\theta'}$ true en $\mathcal{R}$ tal que $\theta' = \theta\ [\mathcal{V}ar(g)]$.

Por su parte, la completitud fuerte de la evaluación parcial se formula como sigue.

Definición 3.6.7 (programa ultra-lineal [Vidal, 1996]**)**
Una regla de reescritura $l \to r \Leftarrow s_1 = t_1, \ldots, s_n = t_n$ es ultra-lineal, si la secuencia $(r, s_1, t_1, \ldots, s_n, t_n)$ no contiene ocurrencias múltiples de una misma variable. Un SRTC es ultra-lineal si sólo contiene reglas ultra-lineales.

Teorema 3.6.8 (completitud fuerte de la EP [Vidal, 1996]**)**
Sea $\mathcal{R}$ un programa confluente y ultra-lineal, g un objetivo, y S un conjunto finito y lineal de términos. Sea $\mathcal{R}'$ una evaluación parcial de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}' \cup \{g\}$ es S-cerrado. Si $g \leadsto^*_\theta$ true es una derivación de narrowing condicional en $\mathcal{R}$ entonces existe una derivación de narrowing condicional $g \leadsto^*_{\theta'}$ true en $\mathcal{R}'$ tal que $\theta' = \theta\ [\mathcal{V}ar(g)]$.

Una vez conocido el concepto y las propiedades de la evaluación parcial de programas lógico-funcionales, en el siguiente apartado pasamos a establecer las diferentes conexiones existentes entre este tipo de transformación y la operación de desplegado. Básicamente, la idea consiste en comprobar que todo programa desplegado $\mathcal{R}'$ es sintácticamente idéntico (bajo renombramiento) a otro obtenido por una evaluación parcial del programa original correspondiente $\mathcal{R}$ con respecto a un conjunto de términos S, donde $S = \{f(x_1, \ldots, x_n) \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in \mathcal{R}$, y $x_i \neq x_j$, para todo $i \neq j\}$. La evaluación parcial concreta a la que nos estamos refiriendo se construye como sigue. En primer lugar, para todo término $f(x_1, \ldots, x_n) \in S$ cuyo símbolo de función más externo no coincide con el símbolo de función más externo de la parte izquierda de la cabeza de R, la evaluación parcial se obtiene construyendo un árbol de *narrowing* en el que todas las ramas se cortan al final del primer nivel (obteniéndose como resultantes reglas de $\mathcal{R}$). En otro caso, la evaluación parcial se obtiene cortando las ramas del árbol de *narrowing* al final del primer nivel para todas las reglas excepto R, y continuando las ramas que usan R un nivel más.

En general, aunque no siempre, una evaluación parcial definida en los términos anteriores es equivalente a una serie de operaciones de desplegado, siempre y cuando ambas técnicas se basen en la misma estrategia refinada de *narrowing* y se apliquen sobre la misma clase de programas. Esta misma relación será explotada de forma inmediata en la demostración del Teorema 3.3.2, para el caso del desplegado basado en *narrowing* impaciente. El resultado también se verifica de forma sencilla para las estrategias perezosa y necesaria (ver [Alpuente *et al.*, 1997a; Albert *et al.*, 1998a,b; Alpuente *et al.*, 1999f] para un estudio detallado de especializaciones basadas en ambos tipos de estrategias perezosas de *narrowing*).

Los principales problemas se presentan cuando se utiliza la relación de *narrowing* (condicional) sin restricciones para definir tanto el desplegado como la evaluación parcial de un programa. En este caso, no siempre se satisface que el primero pueda expresarse en función de la segunda técnica, tal y como muestra el siguiente ejemplo inmediato.

Ejemplo 29 Sea $\mathcal{R}$ el siguiente programa compuesto por una sola regla:

$$\texttt{f(X)} \quad \rightarrow \quad \texttt{c(f(X))}$$

El desplegado basado en *narrowing* de $\mathcal{R}$ devuelve el siguiente programa desplegado $\mathcal{R}'$:

$$\texttt{f(X)} \quad \rightarrow \quad \texttt{c(c(f(X)))}$$

mientras que ninguna evaluación parcial de $\mathcal{R}$ genera un programa sintácticamente equivalente, ya que (al menos) la única regla de $\mathcal{R}$ siempre se reproduce en cualquier programa especializado.

Únicamente cuando se da una definición generalizada de desplegado basado en *narrowing* condicional volvemos a recuperar la relación deseada con la técnica de evaluación parcial, como pasamos a estudiar en el siguiente apartado.

Desplegado generalizado con *narrowing* condicional

A continuación pasamos a extender la noción de desplegado basado en *narrowing* condicional que vimos en la Sección 3.2, para obtener una definición más general que nos permita establecer la relación existente entre este nuevo tipo de desplegado generalizado y la evaluación parcial de un programa.

Definición 3.6.9 (desplegado generalizado de una regla) Sea $\mathcal{R}$ un SRTC y $R = (l \rightarrow r \Leftarrow C) \ll \mathcal{R}$ una regla del mismo. Definimos el desplegado generalizado de la regla R dentro del programa $\mathcal{R}$ como:
$\mathit{Gen\text{-}Unf}_{\mathcal{R}}(R) = \mathit{Unf}_{\mathcal{R}}(R) \cup \{(l \rightarrow y)\theta \mid \theta = mgu(C \cup \{r = y\}) \neq fail\}$.

Nótese que el desplegado basado en *narrowing* condicional y el desplegado generalizado de una regla dentro de un programa, coinciden en el caso de que $mgu(C \cup \{r = y\}) = fail$. Intuitivamente, la regla que se añade (cuando es posible) al desplegado normal para obtener el generalizado, puede verse como una versión “comprimida” de la regla original que se despliega: en realidad la nueva regla es incondicional, pero resume (gracias a la aplicación de la sustitución θ) en la parte derecha de su cabeza, toda la información que aparece en el cuerpo de la regla original. Además, para el caso de reglas incondicionales, $mgu(\{true, r = y\})$ nunca es $fail$, lo que implica que las reglas incondicionales desplegadas siempre se reproducen en los programas transformados. Finalmente, si para una regla $R = (l \rightarrow r \Leftarrow C)$ tenemos que $Gen\text{-}Unf_{\mathcal{R}}(R) = \varnothing$ entonces no existen derivaciones de éxito partiendo de $(C, r = y)$ en $\mathcal{R}$.

El siguiente ejemplo ilustra la definición anterior y señala la principal diferencia con respecto a la Definición 3.2.2.

Ejemplo 30 Consideremos de nuevo el programa $\mathcal{R}$ del Ejemplo 12. El desplegado generalizado de la primera regla de $\mathcal{R}$ es:

$$\begin{array}{lll} \mathtt{f(X)} & \rightarrow & \mathtt{g(X,X)} \\ \mathtt{f(X)} & \rightarrow & \mathtt{c} \qquad \Leftarrow \mathtt{f(a)} = \mathtt{c} \end{array}$$

que contiene la regla original desplegada. Gracias a este hecho, el objetivo $(\mathtt{f(a)} = \mathtt{c})$ calcula la respuesta (normalizada) ϵ tanto en el programa original $\mathcal{R}$ como en el que se obtiene por desplegado generalizado, a diferencia de lo que ocurría con el desplegado ordinario, donde la computación entraba en un bucle infinito.

Una transformación de desplegado que siempre reprodujera la regla original desplegada en el programa transformado sería trivialmente completa, pero de poca utilidad a efectos prácticos. Sin embargo, eso no ocurre en general con nuestro desplegado generalizado, ya que con la nueva definición únicamente se reproduce (una versión “comprimida” de) la regla original en el programa transformado cuando es estrictamente necesario para preservar la completitud de la transformación. A partir de ahora consideramos que la relación de *narrowing* condicional aplica de forma prioritaria la regla de unificación sintáctica sobre un objetivo cuando el conjunto completo de ecuaciones que lo componen es unificable [Middeldorp y Hamoen, 1994].

Ahora estamos en condiciones de definir el desplegado generalizado de un programa con respecto a una de sus reglas como sigue.

Definición 3.6.10 (desplegado generalizado de un programa)
El desplegado generalizado de un programa $\mathcal{R}$ con respecto a una de sus reglas R se define como:

$$Gen\text{-}Unfold(\mathcal{R}, R) = (\mathcal{R} - \{R\}) \cup Gen\text{-}Unf_{\mathcal{R}}(R).$$

El siguiente ejemplo ilustra la definición anterior.

Ejemplo 31 Consideremos el siguiente programa $\mathcal{R}$:

$$\begin{array}{rcll} \mathtt{f(0)} & \rightarrow & \mathtt{0} & & (R_1) \\ \mathtt{g(X,Y)} & \rightarrow & \mathtt{Y} & \Leftarrow \mathtt{f(Y)} = \mathtt{X} & (R_2) \end{array}$$

Entonces

$$Gen\text{-}Unfold(\mathcal{R}, R_1) = \mathcal{R}$$

mientras que $Gen\text{-}Unfold(\mathcal{R}, R_2)$ es:

$$\begin{array}{rcll} \mathtt{f(0)} & \rightarrow & \mathtt{0} & & (R_1) \\ \mathtt{g(X,0)} & \rightarrow & \mathtt{0} & \Leftarrow \mathtt{0} = \mathtt{X} & (R_3) \\ \mathtt{g(f(Y),Y)} & \rightarrow & \mathtt{Y} & & (R_4) \end{array}$$

Nótese que la regla R_4 puede verse como una versión comprimida de R_2 que, por otra parte, nunca aparecería en el desplegado ordinario de $\mathcal{R}$.

El desplegado ordinario y el generalizado de un programa $\mathcal{R}$ con respecto a una de sus reglas R no coincide en general para todas las estrategias de *narrowing*. Para el caso de *narrowing* condicional, si $Unfold(\mathcal{R}, R)$ y $Gen\text{-}Unfold(\mathcal{R}, R)$ se refieren respectivamente al desplegado normal y generalizado de un programa $\mathcal{R}$ con respecto a una de sus reglas R, tenemos que $Unfold(\mathcal{R}, R) \subseteq Gen\text{-}Unfold(\mathcal{R}, R)$. De esta forma, $Gen\text{-}Unfold(\mathcal{R}, R)$ es completo siempre que lo sea $Unfold(\mathcal{R}, R)$.

En lo que sigue, pasamos a probar la equivalencia entre la evaluación parcial y el desplegado generalizado basado en *narrowing* condicional. Esto nos permitirá demostrar diferentes tipos tipos de corrección y completitud para el desplegado generalizado bajo condiciones sencillas y más débiles que las requeridas para el desplegado ordinario visto en la Sección 3.2, apoyándonos en las propiedades de la evaluación parcial basada en *narrowing* condicional que se estudia en [Alpuente *et al.*, 1998a]. Comenzamos formalizando una caracterización alternativa de $Gen\text{-}Unfold(\mathcal{R}, R)$ en términos de evaluación parcial. Intuitivamente, la definición basada en evaluación parcial del desplegado generalizado se fundamenta en la idea de evaluar parcialmente un programa con respecto a las partes izquierdas de las cabezas de las reglas del mismo [Lloyd y Shepherdson, 1991].

Definición 3.6.11 (desplegado generalizado de una regla usando EP) Sea $\mathcal{R}$ un programa y $R = (l \rightarrow r \Leftarrow C) \ll \mathcal{R}$ una regla del mismo. Sea f/n el símbolo de función más externo de l. Sea $s = f(x_1, \ldots, x_n)$, con $x_i \neq x_j$, para todo $i \neq j$. El desplegado generalizado basado en evaluación parcial de R en $\mathcal{R}$, $EP\text{-}Unf_{\mathcal{R}}(R)$ es una evaluación parcial del término s en $\mathcal{R}$ obtenida a partir de un árbol de *narrowing* $\mathcal{T}$ para s donde todas las ramas de $\mathcal{T}$ tienen un solo nivel para todas las reglas de $\mathcal{R}$ excepto las ramas que comienzan con la regla R que tienen dos niveles de profundidad.

La siguiente proposición es la clave para demostrar la corrección del desplegado generalizado, ya que establece la correspondencia precisa entre la evaluación parcial y este tipo de desplegado, que explotaremos posteriormente.

Proposición 3.6.12 *Sea $\mathcal{R}$ un SRTC, y $R = (f(t_1, \ldots, t_n) \rightarrow r \Leftarrow C) \ll \mathcal{R}$ una regla del mismo. Entonces, Gen-$Unf_{\mathcal{R}}(R) \cup (\mathcal{R}_R - \{R\})=$ EP-$Unf_{\mathcal{R}}(R)$, donde $\mathcal{R}_R$ denota al subconjunto de reglas $(f(t'_1, \ldots, t'_n) \rightarrow r' \Leftarrow C') \ll \mathcal{R}$. Además, Gen-$Unfold(\mathcal{R}, R) = (\mathcal{R} - \{R\}) \cup$ Gen-$Unf_{\mathcal{R}}(R) = (\mathcal{R} - \{R\}) \cup$ EP-$Unf_{\mathcal{R}}(R)$[6].*

Demostración. Procedemos por separado con las dos implicaciones de la proposición.

($\subseteq$) Sea $R = (l \rightarrow r \Leftarrow C) \ll \mathcal{R}$ y sea $R' \in$ Gen-$Unf_{\mathcal{R}}(R)$. Por la Definición 3.6.9, existe un paso de *narrowing*:

$$\mathcal{D}_{unf} = [(C, r = y) \rightsquigarrow_{\theta} g]$$

tal que : 1) $R' = (l\theta \rightarrow r' \Leftarrow C')$, si $g = (C', r' = y)$; o 2) $R' = (l \rightarrow y)\theta$, si $g = true$. Ahora podemos construir la siguiente derivación de *narrowing* condicional:

$$\mathcal{D}_{EP} = [f(x_1, \ldots, x_n) = y \rightsquigarrow_{1.1,r,\sigma} (C, r = y) \rightsquigarrow_{\theta} g]$$

donde $f/n) \in \Sigma$ es el símbolo de función más externo de l y $f(x_1, \ldots, x_n)\sigma = l$.

Es inmediato comprobar que la regla desplegada asociada a $\mathcal{D}_{unf}$ y el resultante asociado a $\mathcal{D}_{EP}$ son equivalentes (bajo renombramiento) y, por tanto, $R' \in$ EP-$Unf_{\mathcal{R}}(R)$. Finalmente, como la EP de $f(x_1, \ldots, x_n)$ se construye de tal forma que todas las ramas del árbol de *narrowing* se cortan en el primer nivel para todas las reglas excepto R, entonces el conjunto de reglas $(\mathcal{R}_R - \{R\})$ también está incluido en EP-$Unf_{\mathcal{R}}(R)$, lo que completa la demostración.

($\subseteq$) Es perfectamente análogo al caso anterior, utilizando de nuevo la equivalencia entre los resultantes asociados a $\mathcal{D}_{EP}$ y las reglas desplegadas asociadas a $\mathcal{D}_{unf}$. □

Los resultados de corrección del desplegado generalizado se siguen directamente de los resultados sobre EP que hemos descrito anteriormente, ya que las condiciones de cierre, linealidad e independencia requeridas en ellos (ver de nuevo [Alpuente *et al.*, 1998a]) para la corrección de la EP se satisfacen automáticamente sobre el conjunto de términos parcialmente evaluados cuando se genera el desplegado generalizado de un programa $\mathcal{R}$.

[6]Esta última equivalencia se verifica porque, a pesar de que el conjunto $(\mathcal{R}_R - \{R\})$ no está incluido en Gen-$Unf_{\mathcal{R}}(R)$ y sí en EP-$Unf_{\mathcal{R}}(R)$, dicho conjunto se une a los dos anteriores en la componente $(\mathcal{R} - \{R\})$ para generar, en ambos casos, Gen-$Unfold(\mathcal{R}, R)$.

Teorema 3.6.13 (completitud) *Sea $\mathcal{R}$ un SRTC canónico y sea $R \in \mathcal{R}$ una regla del mismo tal que el desplegado generalizado de $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}' = Gen\text{-}Unfold(\mathcal{R}, R)$. Sea g un objetivo. Si $g \leadsto_{\theta}^{*}$ true es una derivación de narrowing condicional en $\mathcal{R}$ entonces existe una derivación de narrowing condicional $g \leadsto_{\theta'}^{*}$ true en $\mathcal{R}'$ tal que $\theta' \leq \theta$ $[\mathcal{V}ar(g)]$.*

Demostración. Por la Proposición 3.6.12, $\mathcal{R}'$ puede obtenerse también como una evaluación parcial de $\mathcal{R}$ con respecto a S, donde $S = \{f(x_1, \ldots, x_n) \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in \mathcal{R},\ \text{y } x_i \neq x_j, \text{ para todo } i \neq j\}$. La evaluación parcial se construye como sigue. Primero, para todo término $f(x_1, \ldots, x_n) \in S$ cuyo símbolo de función más externo no coincida con el símbolo de función más externo de la parte izquierda de la cabeza de R, la evaluación parcial se obtiene construyendo un árbol de *narrowing* en el que todas las ramas se cortan al final del primer nivel (obteniéndose así como resultantes reglas de $\mathcal{R}$). En otro caso, la evaluación parcial se obtiene podando las ramas del árbol de *narrowing* en el primer nivel para todas las reglas excepto R, y continuando las ramas que usan R un solo nivel más. Por la construcción de S, tenemos que S es lineal y $\mathcal{R} \cup \{g\}$ es S-cerrado, para todo objetivo g. De esta forma el resultado se verifica con tan solo aplicar el Teorema 3.6.3. $\square$

Como una consecuencia del Teorema 3.6.13, el desplegado generalizado aplicado sobre programas canónicos es completo para la semántica del $\mathcal{E}$-modelo mínimo de Herbrand. El siguiente ejemplo muestra que el requerimiento de noetherianidad no puede eliminarse de las condiciones del Teorema 3.6.13.

Ejemplo 32 Sea $\mathcal{R}$ el siguiente programa confluente que no cumple la condición de noetherianidad:

$$\begin{array}{rcl} \mathtt{f(X)} & \to & \mathtt{g(X,X)} \Leftarrow \mathtt{g(X,X)} = \mathtt{c} \\ \mathtt{a} & \to & \mathtt{b} \\ \mathtt{b} & \to & \mathtt{a} \\ \mathtt{g(a,b)} & \to & \mathtt{c} \end{array}$$

El desplegado generalizado de $\mathcal{R}$ con respecto a su primera regla es el programa transformado $\mathcal{R}'$:

$$\begin{array}{rcl} \mathtt{a} & \to & \mathtt{b} \\ \mathtt{b} & \to & \mathtt{a} \\ \mathtt{g(a,b)} & \to & \mathtt{c} \end{array}$$

Ahora, el objetivo $\mathtt{f(a)} = \mathtt{c}$ es evaluado con éxito en $\mathcal{R}$ devolviendo la respuesta normalizada ϵ, mientras que falla en $\mathcal{R}'$.

La corrección y completitud fuertes del desplegado generalizado se formula usando la condición de ultra-linealidad.

Teorema 3.6.14 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un programa confluente y sea $R \in \mathcal{R}$ una regla del mismo tal que el desplegado generalizado de $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}' = Gen\text{-}Unfold(\mathcal{R}, R)$. Sea g un objetivo. Entonces,*

1. (CORRECCIÓN FUERTE) *Si $g \leadsto^*_\theta$ true es una derivación de narrowing condicional en $\mathcal{R}'$ entonces existe una derivación de narrowing condicional $g \leadsto^*_{\theta'}$ true en $\mathcal{R}$ tal que $\theta' = \theta\ [\mathcal{V}ar(g)]$.*

2. (COMPLETITUD FUERTE) *Si $\mathcal{R}$ es ultralineal entonces, si $g \leadsto^*_\sigma$ true es una derivación de narrowing condicional en $\mathcal{R}$ entonces existe una derivación de narrowing condicional $g \leadsto^*_{\sigma'}$ true en $\mathcal{R}'$ tal que $\sigma' = \sigma\ [\mathcal{V}ar(g)]$.*

DEMOSTRACIÓN. La demostración es similar a la del Teorema 3.6.13, reemplazando el uso de del Teorema 3.6.3 por el uso de los Teoremas 3.6.8 y 3.6.6, obteniéndose así la misma respuesta computada en lugar de una más general. □

El siguiente ejemplo ilustra la necesidad de la restricción de linealidad. Intuitivamente, esta condición evita la posible duplicación de términos a través de la instanciación de variables repetidas en una reducción de *narrowing*.

Ejemplo 33 Consideremos el siguiente programa $\mathcal{R}$ confluente y decreciente pero no ultra-lineal:

$$\begin{array}{rcll} \mathtt{f(X,Z)} & \to & \mathtt{g(X,X,Z)} & \Leftarrow\ \mathtt{g(X,X,Z) = c} \\ \mathtt{a} & \to & \mathtt{b} & \\ \mathtt{g(a,b,c)} & \to & \mathtt{c} & \\ \mathtt{g(b,b,W)} & \to & \mathtt{c} & \end{array}$$

El desplegado generalizado de $\mathcal{R}$ con respecto a su primera regla es el programa transformado $\mathcal{R}'$:

$$\begin{array}{rcll} \mathtt{f(b,Z)} & \to & \mathtt{g(b,b,Z)} & \Leftarrow\ \mathtt{c = c} \\ \mathtt{f(b,Z)} & \to & \mathtt{c} & \Leftarrow\ \mathtt{g(b,b,Z) = c} \\ \mathtt{a} & \to & \mathtt{b} & \\ \mathtt{g(a,b,c)} & \to & \mathtt{c} & \\ \mathtt{g(b,b,Z)} & \to & \mathtt{c} & \end{array}$$

Ahora, el objetivo $\mathtt{f(a,Z) = c}$ obtiene como respuesta computada (normalizada) la sustitución $\{\mathtt{Z} \mapsto \mathtt{c}\}$ en $\mathcal{R}$, mientras que en $\mathcal{R}'$ solo puede computarse la respuesta más general $\{\mathtt{Z} \mapsto \mathtt{W}\}$.

Desplegado impaciente y EP

Cuando en la definición de evaluación parcial que hemos dado anteriormente se reemplaza la relación *narrowing* sin restricciones por la de *narrowing* impaciente para construir los árboles de desplegado, es posible establecer la relación precisa entre esta forma de EP y el tipo de desplegado estudiado en la sección 3.3, lo que es la clave para la demostración de la corrección y completitud fuertes de esta última transformación. En este caso, la conexión existente entre ambas técnicas de transformación es más satisfactoria que para el caso de *narrowing* condicional sin restricciones. Comenzamos recordando el resultado de corrección total fuerte para la evaluación parcial (EP) de programas impacientes (y que luego extenderemos de forma segura para la transformación de desplegado) que se establece en [Alpuente *et al.*, 1996b]. Para establecer la corrección de fuerte de la transformación es necesario exigir la condición de independencia (Definición 3.6.5).

Teorema 3.6.15 [Alpuente *et al.*, 1996b] *Sea $\mathcal{R}$ un programa canónico y CB-CD, S un conjunto finito y lineal de patrones y g un objetivo. Sea $\mathcal{R}'$ una evaluación parcial de $\mathcal{R}$ con respecto a S tal que $\mathcal{R}' \cup \{g\}$ es S-cerrado. Entonces,*

1. (Corrección fuerte). *Si S es independiente y $g \overset{\text{NI}}{\leadsto}{}^*_\theta$ true es una derivación de narrowing impaciente en $\mathcal{R}'$ entonces $g \overset{\text{NI}}{\leadsto}{}^*_{\theta'}$ true es una derivación de narrowing impaciente en $\mathcal{R}$, donde $\theta' = \theta\ [\mathcal{V}ar(g)]$.*
2. (Completitud fuerte). *Si $g \overset{\text{NI}}{\leadsto}{}^*_\theta$ true es una derivación de narrowing impaciente en $\mathcal{R}$ entonces $g \overset{\text{NI}}{\leadsto}{}^*_{\theta'}$ true es una derivación de narrowing impaciente en $\mathcal{R}'$, donde $\theta' = \theta\ [\mathcal{V}ar(g)]$.*

La corrección y completitud fuertes del desplegado impaciente estudiado en la sección 3.3 puede demostrarse fácilmente gracias al teorema anterior.

Teorema 3.6.16 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un SRTC canónico y CB-CD. Sea $R \in \mathcal{R}$ una regla del mismo y sea $p \in \mathcal{FP}os(R)$ la posición de un subtérmino más interno de R, tal que el resultado de desplegar $\mathcal{R}$ con respecto a la posición p de R es el programa transformado $\mathcal{R}'$. Sea g un objetivo. Entonces, $g \overset{\text{NI}}{\leadsto}{}^*_\theta$ true es una derivación de narrowing impaciente en $\mathcal{R}$, sii $g \overset{\text{NI}}{\leadsto}{}^*_{\theta'}$ true es una derivación de narrowing impaciente en $\mathcal{R}'$, donde $\theta' = \theta\ [\mathcal{V}ar(g)]$.*

Demostración. A continuación probamos por separado las dos implicaciones del teorema:

($\Leftarrow$) Corrección fuerte. Este caso se demuestra de forma perfectamente análoga al Teorema 3.2.9, mostrando simplemente que cada paso de *narrowing* impaciente en $\mathcal{R}'$ también puede realizarse usando una o dos reglas de $\mathcal{R}$. En efecto, sea $g_0 \overset{\text{NI}}{\leadsto}_{R'',\theta''} g''$ un paso de *narrowing* impaciente en $\mathcal{R}'$. Si $R'' \in \mathcal{R}$, entonces el resultado es cierto trivialmente. En otro caso, por la Definición 3.3.1 existen dos reglas $R, R' \in \mathcal{R}$ y una posición más interna $p \in \mathcal{FP}os(R)$ tal que R'' es el resultado de desplegar la posición p de R usando R'. A su vez, como el Lema 3.2.8 se cumple también para *narrowing* impaciente (ya que toda computación impaciente es a su vez un caso particular de *narrowing* condicional), entonces sabemos que existe la derivación de *narrowing* impaciente $g_0 \overset{\text{NI}}{\leadsto}_{R,\theta} g \overset{\text{NI}}{\leadsto}_{[R',\theta']} g'$ en $\mathcal{R}$, donde $g'' = g'$ y $\theta'' = \theta\theta'$ $[\mathcal{V}ar(g_0)]$, como queríamos demostrar.

($\Rightarrow$) Completitud fuerte. Procedamos ahora con la completitud del desplegado impaciente. Sea $R = (l \rightarrow r \Leftarrow C)$ la regla desplegada en $\mathcal{R}$, donde $(f/n) \in \Sigma$ es el símbolo de función más externo de l. En primer lugar, y de acuerdo con la Definición 3.3.1, puede verse fácilmente que el conjunto de reglas obtenidas al desplegar la ocurrencia más interna p de R con respecto al programa $\mathcal{R}$, coincide con una evaluación parcial de $\{f(x_1, \ldots, x_n)\}$ en $\mathcal{R}$ obtenida a partir de las siguientes derivaciones de *narrowing* impaciente:

$$f(x_1, \ldots, x_n) = y \;\overset{\text{NI}}{\leadsto}_{1.1,R,\sigma}\; (C, r = y) \;\overset{\text{NI}}{\leadsto}_{p,R',\theta}\; g$$

donde $f(x_1, \ldots, x_n)\sigma = l$ y $R' \ll \mathcal{R}$.

Por otra parte, como hemos indicado anteriormente, el programa desplegado $\mathcal{R}'$ puede obtenerse también a través de una evaluación parcial de $\mathcal{R}$ con respecto a S, donde $S = \{f(x_1, \ldots, x_n) \mid (f(t_1, \ldots, t_n) \rightarrow r \Leftarrow C) \in \mathcal{R},\ \text{y } x_i \neq x_j, \text{ para todo } i \neq j\}$. La evaluación parcial concreta a la que nos estamos refiriendo se construye como sigue. En primer lugar, para todo término $f(x_1, \ldots, x_n) \in S$ cuyo símbolo de función más externo no coincide con el símbolo de función más externo de la parte izquierda de la cabeza de R, la evaluación parcial se obtiene construyendo un árbol de *narrowing* impaciente en el que todas las ramas se cortan al final del primer nivel (obteniéndose como resultantes las propias reglas de $\mathcal{R}$). En otro caso, la evaluación parcial se obtiene cortando las ramas del árbol de *narrowing* impaciente al final del primer nivel para todas las reglas excepto R, y continuando las ramas que usan R un nivel más. Por construcción de S, tenemos que S es lineal y $\mathcal{R} \cup \{g\}$ es S-cerrado, para cualquier objetivo g. Por tanto, el resultado que queremos demostrar se verifica con tan sólo aplicar el Teorema 3.6.15. □

3.7 Conclusiones

En esta sección resumimos los principales resultados presentados en este capítulo y mostramos el importante papel que juega la regla de desplegado en el sistema de transformación de programas lógico-funcionales que completaremos más adelante.

El desplegado de un programa declarativo se entiende, en general, como el proceso en virtud del cual una regla (o cláusula) del mismo es sustituida por un nuevo conjunto de reglas obtenidas a partir de la original realizándose sobre ésta uno (o varios) pasos de ejecución simbólica. Para el caso de programs lógico-funcionales, esta clase de evaluación simbólica se realiza usando *narrowing*, lo que aporta una visión unificada de los mecanismos de ejecución y transformación de este tipo de programas, al tiempo que, debido a la posibilidad de explotar su componente funcional, el desplegado de un programa lógico-funcional resulta más efectivo que el de un programa lógico equivalente. A su vez, el desplegado de programas lógico-funcionales basado en (alguna variante de) *narrowing* también subsume (y mejora) los procesos de instanciación y desplegado por reescritura de los programas funcionales puros.

Debido a que el mecanismo de *narrowing* admite diferentes refinamientos, es posible definir tantas instancias de la regla de desplegado como variantes de *narrowing* queramos utilizar para dirigir el proceso. Cada refinamiento del desplegado exige diferentes condiciones de aplicabilidad y disfruta de distintas propiedades que han sido investigadas exhaustivamente en este capítulo y que pasamos a resumir a continuación.

El desplegado basado en *narrowing* condicional sin restricciones no preserva, en general, el significado computacional de los programas transformados (con respecto a los originales), incluso si nos restringimos a requerimientos tan razonables como son observar únicamente las respuestas normalizadas en programas confluentes. En general, la transformación es siempre parcialmente correcta pero es necesario exigir *confluencia, linealidad por la izquierda, decrecimiento* y una condición de *cierre* para asegurar la completitud de la transformación (referida a respuestas normalizadas). En particular, la combinación de la linealidad por la izquierda y el L-cierre garantiza que es posible realizar posteriores desplegados sobre programas previamente transformados sin pérdida de corrección.

Por otra parte, la estrategia de *narrowing* impaciente es correcta y completa con respecto al conjunto de soluciones constructoras básicas sobre programas canónicos, basados en constructores y completamente definidos (CB-CD). Por extensión, el desplegado impaciente de un programa que cumpla estas condiciones, disfruta de resultados fuertes de corrección y completitud con respecto al mismo tipo de semántica, lo que puede demostrarse fácilmente gracias a la relación existente entre este tipo de transformación y la evaluación parcial basada en la misma estrategia sobre la misma clase de programas.

A diferencia de los tipos de desplegado anteriores, marcados por el carácter terminante de los programas, el uso de estrategias perezosas de *narrowing* como núcleo de la regla de desplegado permite trabajar con programas no terminantes. Este tipo de desplegado exige que la parte derecha de la cabeza de toda regla a desplegar no sea un término encabezado por un constructor. Por una parte, cuando el desplegado se define en términos de un *narrowing* perezoso (sin refinar) sobre el tipo de programas para los cuales dicho cálculo es correcto y completo (programas ortogonales), la transformación genera programas que no están dentro de la misma clase, con lo que se corre el riesgo de que las ejecuciones perezosas sobre los mismos no sean completas. Aún cuando se asume que el programa desplegado es ortogonal, los resultados de corrección no son tan fuertes como se hubiese deseado.

Por otro lado, al dirigir el desplegado con una estrategia refinada de *narrowing* perezoso, como es el *narrowing* necesario, se obtienen resultados de corrección y completitud fuertes al tiempo que la transformación preserva la clase natural de programas sobre los que se define la propia estrategia de *narrowing* necesario (inductivamente secuenciales). Además, el desplegado necesario de un programa genera menos reglas redundantes que el perezoso e incluso es capaz de incrementar el grado de determinismo de los programas transformados.

Existe una clase de programas donde las estrategias perezosa y necesaria coinciden: los programas uniformes [Julián, 2000; Zartmann, 1997]. Para esta esta subclase de los programas inductivamente secuenciales, ambos tipos de desplegado presentan resultados de corrección similares. A pesar de que el desplegado perezoso de un programa uniforme no cae siempre dentro de la misma clase, el hecho de que el programa transformado sí sea inductivamente secuencial garantiza que éste puede ser transformado nuevamente en uniforme, recuperándose así el carácter uniforme del programa original.

La transformación de desplegado se ha mostrado muy útil desde que fue concebida y descrita por primera vez tanto en contextos funcionales como lógicos puros. En este capítulo hemos mostrado el importante papel que juega esta regla de transformación (definida en términos de las diferentes estrategias de *narrowing* estudiadas anteriormente) tanto a nivel de generación de semánticas como a la hora de producir evaluadores parciales y sistemas automáticos de transformación de programas lógico-funcionales.

Como aplicación de la transformación de desplegado impaciente, en este capítulo hemos definido una semántica basada en desplegado que es capaz de caracterizar las sustituciones de respuesta computada calculadas sintácticamente por *narrowing* impaciente. La semántica por desplegado de un programa se define como el límite del proceso de desplegado impaciente y su principal objetivo consiste en generar una denotación compuesta por un conjunto de ecuaciones donde las partes derechas son términos constructores. Esta caracterización disfruta de dos propiedades importantes: a) la semántica por desplegado de un programa $\mathcal{R}$ es capaz de reflejar el comportamiento operacional de $\mathcal{R}$ asociado al observable de las respuestas computadas, y b) la semántica por desplegado de $\mathcal{R}$ es otro programa (transformado) $\mathcal{R}'$ compuesto únicamente por reglas incondicionales sobre el que cualquier objetivo puede "evaluarse" usando unificación sintáctica, obteniéndose los mismos resultados que con *narrowing* impaciente sobre el programa original $\mathcal{R}$.

Por su parte, la transformación de evaluación parcial y las que se basan en plegado/desplegado se han desarrollado de forma bastante independiente, aunque su relación ha sido objeto recientemente de numerosas discusiones. Si el desplegado puede considerarse como la base sobre la que se implementa el método de evaluación parcial, en el sentido contrario y siguiendo los argumentos de Lloyd [1987], Levi y Mancarella [1988] y Alpuente *et al.* [1997c], tenemos que el desplegado (generalizado) de un programa se corresponde con una evaluación parcial del mismo donde no hay especialización, lo que redunda en su mejor comprensión y aplicabilidad (sobre todo cuando se combina con la operación de plegado). En contrapartida, la evaluación parcial de un programa posee una menor complejidad y un mayor control del proceso, lo que la hace más fácilmente implementable y automatizable.

En general (aunque no siempre), una evaluación parcial definida en los términos anteriores es equivalente a una secuencia de operaciones de desplegado, siempre y cuando ambas técnicas se basen en la misma estrategia refinada de *narrowing* y se apliquen sobre la misma clase de programas. Este resultado se verifica de forma sencilla para las estrategias impaciente, perezosa y necesaria, pero no cuando se utiliza la relación de *narrowing* (condicional) sin restricciones. Únicamente cuando se da una definición generalizada de desplegado basado en *narrowing* condicional volvemos a recuperar la relación deseada con la técnica de evaluación parcial. Esta nueva noción de desplegado tiene la particularidad de poder reproducir en el programa transformado (una versión comprimida de) la regla desplegada si resulta estrictamente necesario para preservar la completitud de la transformación. Concretamente, exigiendo confluencia y ultralinealidad, es posible establecer una equivalencia total entre este tipo de desplegado y la técnica de evaluación parcial.

A diferencia de la evaluación parcial de programas, las técnicas de transformación de programas basadas en la aproximación "reglas + estrategias" de Burstall y Darlington [1977] no persiguen la especialización de un programa ni restringen el dominio de aplicación de los programas transformados (con respecto a los originales) aunque sí comporten el objetivo común de conseguir programas transformados semánticamente equivalentes y más eficientes que los originales. Los sistemas de tranformación que se obtienen con esta aproximación gozan de una mayor potencia en este sentido pudiendo conseguir optimizaciones no alcanzables mediante técnicas de EP, aunque en general adolecen de poseer un menor grado de automatización. La regla de desplegado juega un papel fundamental en este tipo de sistemas de transformación que pasaremos a discutir a continuación. Los principales resultados que recogeremos aquí, adaptados al campo de los lenguajes lógico-funcionales, son originales de [Alpuente *et al.*, 1997c, 1999e].

Siguiendo la aproximación basada en *reglas + estrategias* de Burstall y Darlington [1977] (ver también [Pettorossi y Proietti, 1996b] para un estudio actual y detallado de esta técnica de transformación tanto en programas lógicos como funcionales puros), consideramos un programa inicial $\mathcal{R}_0$ que es transformado "paso a paso" en un nuevo programa final $\mathcal{R}_n$. Hablamos entonces de una *secuencia de transformaciones* $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$ en la que cada programa $\mathcal{R}_{i+1}$ se obtiene a partir del programa inmediatamente precedente $\mathcal{R}_i$ mediante la aplicación de una regla de transformación elemental. A lo largo de todo el proceso, la semántica del programa inicial debe preservarse, lo que implica la necesidad de que cada regla aplicada disfrute "per se" de propiedades de corrección parcial y completitud (bajo una serie de condiciones de aplicabilidad). Por supuesto, también estamos interesados en que el programa final sea más eficiente que el original, aunque este apartado no concierne intrínsecamente a la definición de las propias reglas, sino más bien al orden "cronológico" en que son aplicadas, es decir a la heurística o "estrategia" utilizada para conducir el proceso de transformación. Este último proceso exige un cierto tipo de control externo que no siempre es automatizable totalmente y que, en general, precisará de cierto grado de interacción entre el sistema y el usuario.

Al asumir esta noción genérica de sistema de transformación (basado en dos componentes: reglas y estrategias), encontramos al menos tres diferencias fundamentales con otros modelos comunes de transformación, como son, por ejemplo, los evaluadores parciales de programas:

1. la necesidad de describir el proceso de transformación paso a paso en base a una secuencia de transformaciones elementales (reglas)

2. un menor grado de automatización a la hora de implementar las estrategias que guían el proceso (estrategias) y

3. un mayor potencial de optimización, ya que es posible mejorar programas siguiendo estrategias (como la formación de tuplas) que no son implementables por evaluación parcial.

Por otra parte, las nociones de computación simbólica y búsqueda de regularidades presentes en prácticamente todas las técnicas de transformación existentes [Pettorossi y Proietti, 1996a,b], aquí se evidencian con mayor claridad, ya que las reglas de plegado y desplegado implementan fielmente estos conceptos, al tiempo que constituyen el núcleo de todo sistema de transformación basado en "reglas + estrategias". De hecho a estos sistemas también se les conoce comúnmente como sistemas basados en plegado y desplegado [Tamaki y Sato, 1984; Bossi y Cocco, 1993; Pettorossi y Proietti, 1996a,b; Sands, 1996], por el papel crucial que juegan ambas reglas en el sistema general. La presencia auxiliar de otras reglas de menor peso en estos sistemas (que, no obstante, a veces resultan imprescindibles) normalmente se justifica por sus implicaciones a la hora de hacer más sencilla la tarea de enunciar las condiciones de aplicabilidad del plegado y desplegado (introducción/eliminación de definiciones), facilitar la aplicación de los mismos (reemplazamiento algebraico, leyes) o permitir la implementación de estrategias sofisticadas como la formación de tuplas (abstracciones).

Centrándonos en el importantísimo papel que juega el desplegado dentro de este tipo de sistemas, diremos que esta regla describe e implementa de forma fidedigna el concepto de computación simbólica. Esto es así porque, en esencia, la aplicación de la regla de desplegado a un programa supone "ejecutar" las (partes derechas) de las reglas del mismo, "adelantando y agilizando" así ejecuciones futuras sobre objetivos concretos, lo que revierte a veces en una mayor eficiencia de los programas transformados. Además, si somos capaces de demostrar la corrección de la propia regla de desplegado (como hemos hecho anteriormente), tendremos también garantizada la corrección de los programas transformados mediante ésta. El binomio "eficicencia + corrección" parece pues más que asequible en aquellos sistemas de transformación equipados con reglas de desplegado definidas en esta línea.

Por otra parte, la aplicación reiterada de la regla de desplegado siguiendo una estrategia apropiada dentro de una secuencia de transformaciones, se intercala muy frecuentemente entre la búsqueda de regularidades (con la ayuda de otras reglas de menor peso). Este hecho debe entenderse como la necesidad de encontrar en las partes derechas de las reglas transformadas expresiones "similares" a (instancias de) las partes derechas de otras reglas, posibilitando de esta manera un proceso posterior de plegado que permita obtener definiciones recursivas y eficientes de las funciones de los programas trasformados.

Ejemplo 34 Consideremos el siguiente programa $\mathcal{R}$ que define las clásicas funciones `append` y `doubleappend` para concatenar dos y tres listas, respectivamente:

$$\begin{array}{rcll}
\mathtt{append(nil,L)} & \rightarrow & \mathtt{L} & (R_1) \\
\mathtt{append(H:T,L)} & \rightarrow & \mathtt{H:append(T,L)} & (R_2) \\
\mathtt{doubleappend(L1,L2,L3)} & \rightarrow & \mathtt{append(append(L1,L2),L3)} & (R_3)
\end{array}$$

Con la pretensión de obtener una versión recursiva de `doubleappend`, procedemos a desplegar la regla R_3 en $\mathcal{R}$, obteniendo el nuevo par de reglas:

$$\begin{array}{rcll}
\mathtt{doubleappend(nil,L1,L2)} & \rightarrow & \mathtt{append(L1,L2)} & (R_4) \\
\mathtt{doubleappend(H:T,L1,L2)} & \rightarrow & \mathtt{append(H:append(T,L1),L2)} & (R_5)
\end{array}$$

Un nuevo desplegado sobre R_5 genera la regla:

$$\mathtt{doubleappend(H:T,L1,L2)} \quad \rightarrow \quad \mathtt{H:}\underline{\mathtt{(append(append(T,L1),L2)}} \quad (R_6)$$

Obsérvese que, tras esta serie de computaciones simbólicas o desplegados, hemos obtenido una regla (la regla R_6) que presenta una regularidad en su parte derecha: el término subrayado en la misma es una instancia de la parte derecha de la regla original R_3. En este momento, y tras un paso de plegado, obtenemos la regla:

$$\mathtt{doubleappend(H:T,L1,L2)} \quad \rightarrow \quad \mathtt{H:doubleappend(T,L1),L2)} \quad (R_7)$$

que, junto con R_4, constituye una definición recursiva y eficiente de la función `doubleappend`.

Este ejemplo ilustra el papel importante del desplegado (y también del plegado) en el seno de un sistema de transformación. Más adelante veremos que, con algunas modificaciones, este ejemplo cae dentro de la estrategia conocida como composición (ver Sección 5.3.1).

Como se apuntó en el primer capítulo de esta tesis, uno de los principales objetivos de este trabajo consiste en la definición de un sistema de transformación para programas lógico-funcionales. Ya que la regla de desplegado muestra su mejores propiedades cuando se define en términos de *narrowing* necesario (lo que viene avalado por sus excelentes propiedades de corrección bajo condiciones de aplicabilidad poco restrictivas y los resultados de optimalidad que presenta: generación de reglas no redundantes, derivaciones de longitud mínima, etc.), en lo que sigue invertiremos el mayor esfuerzo en construir un sistema de transformación (que se basa y extiende al original de Tamaki y Sato [1984] para programas lógicos) sobre sistemas de reescritura inductivamente secuenciales que hace uso de la regla de desplegado necesario.

Capítulo 4

Plegado

La regla de plegado junto con la de desplegado constituyen el núcleo de todo sistema de transformación basado en la contracción o expansión, respectivamente, de subexpresiones dentro de un programa usando las definiciones del mismo (o de otro programa precedente en la secuencia de transformación).

Aunque con algunas matizaciones, en general, la transformación de plegado de un programa declarativo se considera la operación inversa de la transformación de desplegado, es decir, se espera que un paso de desplegado seguido del paso de plegado correspondiente (y viceversa) devuelva el programa inicial (o practicamente el mismo). Sin embargo, es necesario hacer algunas puntualizaciones respecto a este hecho. Intuitivamente, para conseguir el efecto comentado de inversibilidad o reversibilidad total, es necesario exigir como mínimo que todas las reglas involucradas en una transformación de plegado pertenezcan al mismo programa, al tiempo que se necesita poder plegar un conjunto de reglas simultáneamente para obtener una única regla transformada. Cabe destacar que esta última condición es justo la contraria a la que se verifica en el desplegado (donde una sola regla se despliega originando un conjunto de reglas desplegadas), ya que solamente de esta forma puede alcanzarse el resultado inverso esperado. Estas condiciones no son siempre requeridas a la hora de definir algunos tipos de plegado, lo que da lugar a distintas variantes de esta operación que poseen una capacidad desigual con respecto a su potencial para la optimización de programas.

En términos generales, si el desplegado de una regla (cláusula) en un programa funcional (lógico) reemplaza una llamada a función (átomo) por su respectiva definición (aplicando la correspondiente sustitución), el plegado reemplaza un cierto fragmento de código por la llamada a función (átomo) apropiada. Más exactamente, la operación de plegado consiste en la sustitución, en la parte derecha de una regla, de una expresión o un conjunto de llamadas a función (junto con un conjunto de condiciones ecuacionales, si estamos hablando de programas condicionales), por una expresión o llamada a función semánticamente equivalente. Esta operación se utiliza generalmente en las técnicas de transformación de programas para "empaquetar" las reglas desplegadas y detectar definiciones recursivas implícitas. También es común su uso en las técnicas de evaluación parcial cuando éstas se formulan en términos de transformaciones de plegado/desplegado [Pettorossi y Proietti, 1994].

En general, y a diferencia del desplegado, el proceso de plegado es menos dependiente no ya sólo del paradigma declarativo que estemos considerando (lógico puro, funcional puro o lógico-funcional) sino también del mecanismo operacional asociado a las computaciones (resolución SLD, reescritura o *narrowing*, respectivamente). En particular, en nuestro contexto integrado, las diferentes variantes de *narrowing* no afectan sustancialmente a la definición o propiedades del plegado. Es más, resultan insignificantes las diferencias derivadas de utilizar distintas variantes del *narrowing* en las formulaciones del plegado (salvo, claro está, en las reversibles), siendo ésta la razón principal por la que en este capítulo distinguiremos diferentes versiones del plegado basándonos prioritariamente en los criterios comentados en los párrafos anteriores. Comenzamos este capítulo haciendo un breve recorrido histórico sobre los antecedentes del plegado en contextos funcionales y lógicos puros, y motivando el problema de la formulación de esta regla en un contexto integrado.

4.1 Motivación y antecedentes

La primera formulación a la regla de plegado (análogamente al caso del desplegado) es original de Burstall y Darlington [1977]. El plegado aquí se refiere a programas funcionales puros, entendidos como conjuntos de ecuaciones recursivas, y se define textualmente como sigue:

> *Si $E = E'$ y $F = F'$ son ecuaciones y existe alguna ocurrencia de F' que es instancia de E', entonces se reemplaza por la instancia correspondiente de E, obteniendo F'' y añadiendo la ecuación $F = F''$.*

Obsérvese la similitud de esta definición con su homóloga del desplegado vista en el capítulo anterior. La principal diferencia está en el hecho de que ahora el proceso de transformación puede verse como un paso de reescritura dado sobre la parte derecha de la ecuación que se pliega, pero utilizando la ecuación plegante en sentido inverso. De esta forma, la operación de plegado procede en dirección contraria a los pasos usuales de reducción, es decir, la reescritura se realiza aplicando la regla previamente invertida.

En esta primera aproximación no se hace explícito el hecho de que es conveniente retirar la regla plegada del conjunto resultante de ecuaciones para, de esta forma, intercambiarse por la regla transformada correspondiente. Obsérvese que tampoco se indica directamente de qué programas se extraen las reglas involucradas en la transformación. En este sentido, tal y como muy acertadamente señala Sands [1996], existe un punto importante que no está detallado, pero que es esencial en la práctica, en la definición estándar de plegado: un paso de plegado que siempre reemplaza una instancia del cuerpo de una definición de una función f por la correspondiente instancia de una llamada a función, pero que no es capaz de plegarla haciendo uso de una versión anterior de la función f (como ocurre por ejemplo en [Courcelle, 1990]) nunca puede generar una definición recursiva directa y eficiente de la misma.

En el trabajo de Burstall y Darlington [1977] no se dan resultados de corrección ni completitud para esta regla de transformación (ni para ninguna otra), lo cual no es sorprendente (máxime al tratarse de un trabajo pionero), ya que en general la operación de plegado es mucho más compleja que el desplegado, si se pretende que sea conservativa, es decir, correcta. Es bien conocido el hecho de que el plegado sin restricciones de un programa funcional (o lógico) puro no es, en general, conservativo o correcto [Amtoft, 1992; Bossi y Cocco, 1993; Pettorossi y Proietti, 1994; Seki, 1993; Tamaki y Sato, 1984], incluso cuando se considera la semántica más simple posible. Mientras que la corrección de la transformación es fácilmente demostrable, no ocurre lo mismo con la completitud, ya que el plegado puede introducir un tipo de recursión no terminante en los programas transformados. Un ejemplo clásico es el plegado de la segunda regla del programa funcional $\mathcal{R} = \{\mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(N+1)} \to \mathtt{f(N)}\}$ con respecto a sí misma. Esta operación genera el programa transformado $\mathcal{R}' = \{\mathtt{f(0)} \to \mathtt{0},\ \mathtt{f(N+1)} \to \mathtt{f(N+1)}\}$, que deja indefinido $\mathtt{f(N)}$ para todo $\mathtt{N} > \mathtt{0}$.

Han habido algunos otros desarrollos en este mismo contexto funcional puro (ver por ejemplo los trabajos de Kott [1985]; Scherlis [1981]; Courcelle [1990]; Zhu [1994]), aunque quizás el texto más interesante y moderno sea el de Sands [1996], donde se presenta un tipo de plegado (basado en *marcas*) que es correcto y completo con respecto a la semántica de valores constructores básicos, bajo condiciones que no resultan excesivamente restrictivas, en programas funcionales perezosos con orden superior.

Por otra parte, en la literatura relacionada con las transformaciones de programas lógicos, vuelven a manifestarse todas las dificultades que estaban presentes en el caso funcional. En particular, se repite el problema relacionado con la terminación de los programas plegados. Sirva como ejemplo el clásico programa lógico $\mathcal{P} = \{\mathtt{p(X)} \Leftarrow \mathtt{q(X).},\ \mathtt{q(a).}\}$, donde si plegamos la primera cláusula de $\mathcal{P}$ con respecto a sí misma, obtenemos el programa transformado $\mathcal{P}'$ que contiene la regla $\mathtt{p(X)} \Leftarrow \mathtt{p(X)}$, de forma que el objetivo $\Leftarrow \mathtt{p(X)}$ entra en un bucle en $\mathcal{P}'$ mientras que tiene éxito en $\mathcal{P}$ con respuesta computada $\{\mathtt{X} \mapsto \mathtt{a}\}$.

La primera adaptación de la regla de plegado al contexto lógico puro se debe a Tamaki y Sato [1984], donde se extienden las ideas originales de Burstall y Darlington [1977], perfilándose así las condiciones de aplicabilidad que permiten probar el carácter correcto y completo de la transformación para la semántica básica (modelo mínimo de Herbrand) de los programas lógicos. Más aún, Kawamura y Kanamori [1990], demuestran que la misma definición de plegado (con ligerísimas variaciones) preserva también la semántica más general de las respuestas computadas.

A partir del trabajo inicial de [Tamaki y Sato, 1984] se han descrito diferentes formulaciones de la operación de plegado en el contexto lógico puro. Las diferencias dependen principalmente de la forma de derivar la equivalencia entre la conjunción de átomos que van a ser plegados y las llamadas correspondientes.

El caso más simple se presenta cuando tal equivalencia se deriva directamente de una cláusula que pertenece al mismo programa sobre el que se realiza la operación de plegado [Gardner y Shepherdson, 1991; Maher, 1987a; Pettorossi y Proietti, 1994]. Por ejemplo, en [Gardner y Shepherdson, 1991] se presenta un tipo de plegado que, bajo ciertas condiciones, puede deshacerse mediante un paso de desplegado apropiado (la inversibilidad se verifica cuando este paso de desplegado devuelve una sola regla). Este hecho simplifica enormemente las demostraciones de corrección de la transformación (que se obtienen directamente a partir de la corrección del desplegado), pero se requieren unas condiciones de aplicabilidad que son muy restrictivas, al tiempo que el poder de optimización del sistema de transformación queda reducido drásticamente.

Por otro lado, el *plegado reversible* que se define en [Pettorossi y Proietti, 1994] requiere que los conjuntos formados por las reglas a plegar y las reglas usadas para realizar el plegado sean disjuntos, además de pertenecer al mismo programa. Los pasos de plegado dados de esta forma son totalmente *reversibles*, ya que después de un paso de plegado, siempre existe otro de desplegado que restituye el programa inicial. Las condiciones que se requieren para garantizar la corrección del proceso son sintácticas y muy fáciles de comprobar pero, a menudo, son muy restrictivas (ya que, en un sistema de transformación de programas, las reglas que deben ser plegadas pueden haber sido modificadas o eliminadas del programa durante las transformaciones previas).

Por todo ello, y al igual que ya ocurre en la primera aproximación de Tamaki y Sato [1984], en un buen número de propuestas para la transformación de programas lógicos (ver, por ejemplo, [Bossi y Cocco, 1993; Seki, 1993; Pettorossi y Proietti, 1994, 1996b]), los dos conjuntos de cláusulas utilizados en una operación de plegado pueden pertenecer a programas distintos. O bien, en el caso más restrictivo, las cláusulas plegantes deben pertenecer al primer programa de una secuencia obtenida por sucesivas transformaciones, al estilo de Kawamura y Kanamori [1990]. En ambas situaciones, el carácter no reversible del plegado definido en estos términos se revela crucial para conseguir optimizaciones efectivas. Curiosamente, en las diferentes formulaciones de esta clase de plegados potentes pero no reversibles, es suficiente con exigir que la secuencia de átomos a plegar dentro de una cláusula sea una mera instancia del cuerpo de la cláusula plegante. De esta forma este tipo de plegado utiliza emparejamiento de patrones en vez de unificación para implementar y llevar a cabo la transformación, simplificándola de forma similar a lo que ya ocurre en programación funcional y (sorprendentemente) sin perder por ello ningún poder de optimización.

En el contexto integrado de la programación lógico-funcional, para garantizar que se preservan las propiedades de terminación y corrección del plegado, su aplicación debe ser restringida también por condiciones apropiadas, lo que en ocasiones depende de la semántica que en cada caso se pretenda preservar. Tales restricciones pueden afectar a la propia secuencia de transformaciones que sufre un programa (como ocurría en los trabajos de [Kawamura y Kanamori, 1988; Kott, 1985; Tamaki y Sato, 1984] para el caso lógico puro) y/o pueden expresarse sólo en términos de las propiedades que cumplen los programas, independientemente a veces de su historia de transformación (como ocurría también en el caso de [Bossi y Cocco, 1993; Bossi *et al.*, 1992]).

Ejemplo 35 Consideremos el siguiente programa lógico-funcional $\mathcal{R}$:

$$\begin{array}{lcll} \mathtt{f(0)} & \rightarrow & \mathtt{0} & (R_1) \\ \mathtt{g(0)} & \rightarrow & \mathtt{0} & (R_2) \end{array}$$

La regla R_2 puede ser plegada usando la regla R_1, obteniéndose el programa transformado $\mathcal{R}'$:

$$\begin{array}{lcll} \mathtt{f(0)} & \rightarrow & \mathtt{0} & (R_1) \\ \mathtt{g(0)} & \rightarrow & \mathtt{f(0)} & (R_2') \end{array}$$

Ahora, el objetivo $\mathtt{g(X)} = \mathtt{Y}$ tiene éxito con respuesta computada $\{\mathtt{Y} \mapsto \mathtt{f(0)},\ \mathtt{X} \mapsto \mathtt{0}\}$ en el programa $\mathcal{R}'$, mientras que en el programa original $\mathcal{R}$ nunca se computa una respuesta igual (o más general) para el mismo objetivo.

Dada la gran cantidad de variantes del plegado que se han presentado en los contextos lógicos y funcionales puros, en lo que sigue pretendemos adaptar a nuestro contexto integrado sólo las aproximaciones más significativas. Básicamente los criterios de diferenciación más importantes son dos[1]:

- la posibilidad de plegar varias cláusulas (o reglas) simultaneamente –lo que se conoce como plegado *disyuntivo*–, o sólo una de ellas de una vez –plegado *conjuntivo*–, y

[1] Otros criterios adicionales que no vamos a considerar aquí permitirían tomar en cuenta el carácter recursivo de las reglas involucradas en el proceso de plegado [Kanamori y Fujita, 1987; Tamaki y Sato, 1986; Roychoudhury *et al.*, 1999], la transformación de programas con negación [Aravindan y Dung, 1995; Seki, 1991, 1993], el tratamiento y aplicación de otros problemas prácticos [Bossi *et al.*, 1990; Boulanger y Bruynooghe, 1993; Pettorossi *et al.*, 1997].

- la condición de que todas las reglas involucradas en el plegado pertenezcan o no a un mismo programa.

En función de estos criterios, las tres próximas secciones de este capítulo se organizan como sigue:

1. En la Sección 4.2 presentamos una versión del plegado reversible inspirada en el plegado *in-situ* descrito en [Pettorossi y Proietti, 1998], que es disyuntivo y utiliza reglas de un único programa. Dado que la noción de reversibilidad siempre se entiende con respecto a un tipo de desplegado concreto, en nuestro caso lo hacemos con arreglo al desplegado impaciente descrito en la Sección 3.3 del capítulo anterior. La razón para adoptar esta postura estriba en el hecho de que solamente en este caso se obtiene una formulación del plegado que es practicable bajo condiciones de aplicabilidad razonables.

2. La Sección 4.3 introduce una variante conjuntiva del plegado, eliminando la posibilidad de que sean varias las reglas plegadas simultáneamente. En general, se obtiene de esta forma un plegado (similar al definido por Gardner y Shepherdson [1991] para programas lógicos) que no siempre es reversible en general, pero que sigue considerando reglas de un único programa, lo que todavía no es suficiente para dotar a la operación del poder de optimización deseado.

3. Finalmente, en la Sección 4.4 se formula una extensión y adaptación del plegado clásico de Tamaki y Sato [1984] a un contexto declarativo integrado. El tipo de plegado así obtenido permite el uso de diferentes programas para extraer las reglas plegantes y plegadas, al tiempo que es conjuntivo. Esta variedad de plegado es obviamente irreversible y goza de todo el potencial suficiente para optimizar programas inductivamente secuenciales al combinarse junto con el desplegado necesario descrito en la Sección 3.4.2 del capítulo anterior (junto con otras reglas de transformación que introduciremos posteriormente).

Cerraremos el capítulo comentando algunas aplicaciones de esta transformación (en particular, estudiaremos el rol del plegado en el seno de un sistema de transformación y de un sistema de especialización basado en evaluación parcial en la línea de Alpuente *et al.* [2000b]) y resumiendo los principales resultados obtenidos.

4.2 Plegado reversible

Como ya vimos en capítulos anteriores, una característica sustancial de la estrategia de *narrowing* impaciente es que la función de selección para elegir la posición más interna explotada en cada paso hace uso de un indeterminismo de tipo *don't care*.

Ello implica que en el desplegado impaciente de una regla del programa, es suficiente con considerar una única posición más interna para desplegar, independientemente de que existan varias ocurrencias seleccionables. Ésta es una característica positiva del desplegado impaciente que, además, también resulta de gran utilidad a la hora de definir un tipo de plegado reversible cuyos efectos puedan ser anulados por un eventual desplegado impaciente posterior. De esta forma, se puede dar una definición simplificada de este tipo de plegado en la que se considera un conjunto de reglas a plegar, pero sólo una posición más interna ("innermost") plegable.

Es importante destacar que esta propiedad no se verificaría si persiguiéramos la inversibilidad usando una estrategia de *narrowing* con indeterminismo *don't know* (i.e., que explota varias posiciones en cada paso, como es el caso del desplegado sin restricciones, el perezoso y el necesario), ya que se arrastraría este nuevo grado de indeterminismo (además del asociado al hecho de usar varias reglas plegantes) sobre la definición de plegado. En otras palabras, no sólo sería necesario considerar un conjunto de reglas a plegar, sino también un conjunto (no necesariamente unitario) de posiciones plegables. En aras de simplificar la definición de plegado reversible (que, por lo demás, nunca es del todo sencilla), nos limitamos pues a hacerlo tomando como base el *narrowing* impaciente.

A continuación, introducimos nuestra definición de plegado reversible, que puede verse como una extensión al caso lógico-funcional del plegado reversible de Pettorossi y Proietti [1998] definido para programas lógicos. Como ya hemos avanzado, hemos elegido una forma de plegado impaciente que se beneficia de las ventajas que ya disfruta la propia regla de desplegado impaciente, al tiempo que se consigue igualmente preservar las respuestas computadas.

Cabe destacar que la definición todavía presenta dos fuentes de indeterminismo. La primera surge en la elección de las llamadas plegables, mientras que la segunda viene asociada a la selección de una generalización (*llamada plegante*) de las cabezas de las definiciones de función que se usan para sustituir las llamadas plegables. En la siguiente definición usamos la notación $\mathcal{P}os(R)$ para referenciar el conjunto de posiciones de la parte derecha o de la condición de una regla R, mientras que con $R[t]_p$ nos referimos a una regla que se obtiene por reemplazamiento del término asociado a la posición $p \in \mathcal{P}os(R)$ por el término t.

Definición 4.2.1 (plegado reversible) Sea $\mathcal{R}$ un programa. Sean $\{R_1, \ldots, R_n\} \ll \mathcal{R}$ (las "reglas plegables") y $R_{def} = \{R'_1, \ldots, R'_n\} \ll \mathcal{R}$ (las "reglas plegantes") dos conjuntos disjuntos de reglas de programa (renombradas), con $R'_i = (l'_i \to r'_i \Leftarrow C'_i)$, $i = 1, \ldots, n$. Sea R una regla[2], $p \in \mathcal{P}os(R)$ una posición de la regla R y t un patrón

[2]En términos sencillos, R viene a ser el "esqueleto común" de las reglas empleadas en el paso de plegado. La posición p de R actúa como un apuntador al "hueco" donde se incorpora la llamada plegante.

tales que, para todo $i = 1, \ldots, n$:

1. $\theta_i = mgu(\{l'_i = t\}) \neq fail$,
2. $R_i = (l \to r_i \Leftarrow C'_i, C_i)\theta_i$ y $R[r'_i]_p = (l \to r_i \Leftarrow C_i)$, y
3. para cada regla $R' = (l' \to r' \Leftarrow C') \ll \mathcal{R}$ no perteneciente al conjunto R_{def}, $mgu(\{l' = t\}) = fail$.

El plegado reversible del programa $\mathcal{R}$ con respecto al conjunto de reglas plegables $\{R_1, \ldots, R_n\}$ usando R_{def} como conjunto de reglas plegantes, es el programa transformado $\mathcal{R}'$ definido como sigue:

$$\mathcal{R}' = (\mathcal{R} - \{R_1, \ldots, R_n\}) \cup \{R_{fold}\},$$

donde $R_{fold} = R[t]_p$.

Intuitivamente, nuestro plegado reversible actúa en la dirección contraria a los pasos de *narrowing* que se aplican en una operación de desplegado. En tales pasos de *narrowing*, para un unificador concreto del redex y la parte izquierda de la regla explotada, se da un paso de reescritura sobre el redex instanciado, a continuación se añaden a la regla desplegada las condiciones de la regla desplegante y, finalmente, se aplica la sustitución computada por *narrowing*. Aquí, simplemente se procede de forma simétrica: primero se "desinstancian" (generalizan) las reglas plegables, a continuación se eliminan las condiciones de las reglas plegantes aplicadas y, finalmente, se realiza un paso de reducción contra las cabezas invertidas de las reglas plegantes. El siguiente ejemplo ilustra la definición de plegado reversible.

Ejemplo 36 Consideremos el siguiente programa canónico y CB-CD $\mathcal{R}$:

$$\begin{array}{rcll}
\texttt{f(X)} & \to & \texttt{s(X)} & \Leftarrow \texttt{h(s(X))} = \texttt{0} \quad (R_1) \\
\texttt{f(s(Z))} & \to & \texttt{s(s(0)))} & \Leftarrow \texttt{Z} = \texttt{0} \quad (R_2) \\
\texttt{num(Y)} & \to & \texttt{Y} & \Leftarrow \texttt{h(Y)} = \texttt{0} \quad (R_3) \\
\texttt{num(s(s(Z)))} & \to & \texttt{s(s(0))} & \Leftarrow \texttt{Z} = \texttt{0} \quad (R_4)
\end{array}$$

Si plegamos las reglas $\{R_1, R_2\}$ de $\mathcal{R}$ con respecto a $R_{def} = \{R_3, R_4\}$ usando $R = (\texttt{f(X)} \to \Box)$ y $t = \texttt{num(s(X))}$, obtenemos el siguiente programa transformado $\mathcal{R}'$:

$$\begin{array}{rcll}
\texttt{f(X)} & \to & \texttt{num(s(X))} & \quad (R_{fold}) \\
\texttt{num(Y)} & \to & \texttt{Y} & \Leftarrow \texttt{h(Y)} = \texttt{0} \quad (R_3) \\
\texttt{num(s(s(Z)))} & \to & \texttt{s(s(0))} & \Leftarrow \texttt{Z} = \texttt{0} \quad (R_4)
\end{array}$$

La Definición 4.2.1 se refiere a un tipo de plegado que posee las siguientes características:

1. es disyuntivo, ya que el conjunto de reglas plegantes R_{def} no tiene por qué ser unitario,

2. los conjuntos de reglas plegables y plegantes pertenecen al mismo programa, y

3. además, dichos conjuntos son disjuntos.

Es fácil ver que las dos primeras condiciones son precondiciones necesarias, aunque no suficientes, para obtener una versión reversible del plegado. Al añadir la tercera condición, obtenemos lo que en [Pettorossi y Proietti, 1998] se conoce como plegado "in-situ", que a su vez ya había sido denotado como reversible en [Pettorossi y Proietti, 1994], pues efectivamente lo es en programación lógica (aunque no es cierto en un contexto integrado).

Para conseguir el objetivo deseado de reversibilidad en el contexto lógico-funcional, aparte de los requerimientos anteriores, en nuestra definición de plegado impaciente hemos añadido una nueva condición de aplicabilidad para asegurar el carácter reversible de la transformación: el término t que reemplaza a las llamadas plegables ha de ser un patrón.

Obsérvese que este último requerimiento es nuevo en el contexto lógico-funcional con respecto a cualquier tipo de plegado conocido y ya adelanta la correspondencia de esta operación con su contrapartida del desplegado impaciente, aún cuando en la definición no se ha hecho una referencia explícita a la propia estrategia de *narrowing* impaciente. Esta condición es clave para demostrar la propiedad de inversibilidad que asegura que un programa plegado puede ser desplegado nuevamente por un paso impaciente volviendo así al programa original.

El siguiente lema formaliza la condición de reversibilidad, mostrando que los pasos de plegado pueden deshacerse mediante pasos de desplegado impaciente apropiados. Esto nos permitirá demostrar, posteriormente, la corrección y completitud de la transformación.

Lema 4.2.2 *Sea $\mathcal{R}$ un programa canónico y CB-CD. Sea $\mathcal{R}'$ es el resultado de plegar el conjunto de reglas plegables $\{R_1, \ldots, R_n\}$ usando R_{def} como conjunto de reglas plegantes. Sea R_{fold} la nueva regla introducida en $\mathcal{R}'$ por el paso de plegado. Entonces existe una posición $p \in \mathcal{FP}os(R_{fold})$ asociada a un patrón tal que $\mathcal{R}$ es el programa obtenido por el desplegado impaciente de la regla $R_{fold} \in \mathcal{R}'$ con respecto a su posición p en $\mathcal{R}'$.*

Demostración. Sea $R_{def} = \{R'_1, \ldots, R'_n\} \ll \mathcal{R}$, con $R'_i = (l'_i \to r'_i \Leftarrow C'_i)$, $i = 1, \ldots, n$. Por la Definición 4.2.1, R_{fold} es de la forma $R[t]_p$, donde $R = (l \to r \Leftarrow C)$ es una regla, $p \in \mathcal{P}os(R)$ es una posición de R[3] y t es un patrón tal que, para todo $i = 1, \ldots, n$:

[3]Por simplicidad, consideramos que $p \in \mathcal{P}os(C)$ y, por tanto, que $C|_p = t$ (el caso en el que t es un subtérmino de r es perfectamente análogo).

1. $\theta_i = mgu(\{l'_i = t\}) \neq fail$,
2. $R_i = (l \to r \Leftarrow C'_i, C[r'_i]_p)\theta_i$, y
3. para cada regla $R' = (l' \to r' \Leftarrow C') \ll \mathcal{R}$ no perteneciente al conjunto R_{def}, $mgu(\{l' = t\}) = fail$.

Ahora demostramos que $\{R_1, \ldots, R_n\}$ es precisamente el conjunto de nuevas reglas que se obtienen al desplegar la regla R_{fold} a la posición p en $\mathcal{R}'$, ya que el lema se sigue directamente de este hecho. Primeramente, ya que $\{R_1, \ldots, R_n\}$ y R_{def} son conjuntos disjuntos, tenemos que las reglas de R_{def} siguen perteneciendo a $\mathcal{R}'$. Además, como $R[t]_p = (l \to r \Leftarrow C[t]_p)$ y t es un patrón, entonces puede darse el siguiente conjunto de pasos de *narrowing* impaciente:

$$(C[t]_p, r = y) \overset{\text{NI}}{\leadsto}_{p, R'_i, \theta_i} (C'_i, C[r'_i]_p, r = y)\theta_i$$

para $i = 1, \ldots, n$. Además, como no existe ninguna regla en $(\mathcal{R} - R_{def})$ cuya parte izquierda unifique con t, entonces no existen más pasos de *narrowing* impaciente posibles para $(C[t]_p, r = y)$ en $\mathcal{R}'$ a la posición p. Finalmente, por la Definición 3.3.1 de desplegado impaciente, tenemos que el desplegado de la regla R_{fold} a la posición p en $\mathcal{R}'$ genera el conjunto de reglas $\{(l \to r \Leftarrow C'_i, C[r'_i]_p)\theta_i \mid i = 1, \ldots, n\} = \{R_1, \ldots, R_n\}$, lo que completa la demostración. □

Ejemplo 37 Consideremos de nuevo el programa plegado $\mathcal{R}'$ del Ejemplo 36. Si ahora desplegamos la regla R_{fold} de $\mathcal{R}'$ con respecto a la llamada a función `num(s(x))`, entonces obtenemos de nuevo el programa inicial $\mathcal{R}$.

Como consecuencia del Lema 4.2.2 anterior, tenemos que los efectos de un desplegado impaciente pueden a su vez ser deshechos por un paso de plegado impaciente reversible apropiado, tal y como se enuncia en el siguiente corolario:

Corolario 4.2.3 *Sea $\mathcal{R}$ un programa canónico y CB-CD. Si $\mathcal{R}'$ es el resultado de desplegar la regla R a la posición más interna $p \in \mathcal{P}os(R)$ usando R_{def} como conjunto de reglas desplegantes, entonces $\mathcal{R}$ es el programa obtenido por el plegado impaciente del conjunto de reglas plegables $\{R_1, \ldots, R_n\}$ usando R_{def} como conjunto de reglas plegantes en $\mathcal{R}'$, donde $\{R_1, \ldots, R_n\}$ es el nuevo conjunto de reglas introducidas en $\mathcal{R}'$ por el paso de desplegado impaciente.*

Apoyándonos en los resultados anteriores, estamos en condiciones de probar de forma sencilla la corrección y completitud fuertes de la transformación de plegado reversible.

Teorema 4.2.4 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un SRTC canónico y CB-CD, $\mathcal{R}'$ un programa obtenido por plegado reversible de $\mathcal{R}$ y g un objetivo. Entonces, $g \overset{\text{NI}}{\leadsto}^*_\theta true$ es una derivación de narrowing impaciente en $\mathcal{R}$, si y sólo si $g \overset{\text{NI}}{\leadsto}^*_{\theta'} true$ es una derivación de narrowing impaciente en $\mathcal{R}'$, donde $\theta' = \theta\ [\mathcal{V}ar(g)]$.*

DEMOSTRACIÓN. Este resultado es consecuencia directa del Teorema 3.3.2 y el Lema 4.2.2. En primer lugar, por el Lema 4.2.2, siempre existe una regla $R_{fold} \in \mathcal{R}'$ y una posición p de dicha regla tales que $\mathcal{R}$ se obtiene al desplegar $\mathcal{R}'$ con respecto a la regla R_{fold} sobre la posición p. En segundo lugar, por la corrección y completitud fuerte del desplegado impaciente, expresada en el Teorema 3.3.2, obtenemos el resultado que queremos demostrar. □

Los Teoremas 4.2.4 y 3.3.2 muestran que es factible la construcción de un sistema de transformación de programas canónicos y CB-CD basado en las reglas de plegado impaciente reversible y desplegado impaciente que es capaz de preservar la semántica de respuestas computadas para este tipo de programas. Si bien de esta forma quedan resueltos los problemas de corrección del sistema, no ocurre lo mismo con la capacidad "optimizadora" del mismo. Es más, como es bien conocido, todo sistema de transformación equipado con un solo tipo de plegado, que además es reversible, tiene un poder optimizador muy limitado ya que, entre otras dificultades, las reglas que interesaría usar como plegantes no es frecuente que se encuentren en el mismo programa que las plegables. En los apartados siguientes nos proponemos investigar nuevas versiones del plegado que, a cambio de perder el carácter reversible, superan estas deficiencias.

4.3 Plegado no reversible

En esta sección describimos un tipo de plegado que, a diferencia del anterior, ya no es siempre reversible (debido a que no es disyuntivo). Aunque la potencia de optimización de la transformación todavía es muy limitada, el nuevo tipo de plegado puede verse como un primer paso para la consecución de la versión más potente y flexible que veremos en la Sección 4.4. La inspiración en esta ocasión proviene del texto de Gardner y Shepherdson [1991], donde se describe un tipo de plegado conjuntivo en el que las reglas plegables y plegantes se extraen de un mismo programa lógico. Nuestra adaptación al caso de los programas inductivamente secuenciales con una semántica operacional basada en *narrowing* necesario se plantea como sigue.

Definición 4.3.1 (plegado no reversible) Sea $\mathcal{R}$ un programa inductivamente secuencial. Sea $R = (l \to r) \in \mathcal{R}$ una regla (la "regla plegable") donde $\mathcal{V}ar(l) = \mathcal{V}ar(r)$ y sea $R' = (l' \to r') \in \mathcal{R}$ otra regla (la "regla plegante") tal que R' no es una variante de R y $r|_p = r'\theta$ para alguna posición $p \in \mathcal{FP}os(r)$. El plegado no reversible del programa $\mathcal{R}$ con respecto a la regla plegable R usando R' como regla plegante, es el programa transformado $\mathcal{R}'$ definido como sigue:

$$\mathcal{R}' = (\mathcal{R} - \{R\}) \cup \{l \to r[l'\theta]_p\} .$$

A diferencia de la definición original de Gardner y Shepherdson [1991], en nuestro caso no es necesario exigir que R' sea la única regla en $\mathcal{R}$ cuya parte izquierda unifique con $l\theta$. Por otra parte, el hecho de que R' no pueda ser una variante de R garantiza que no está permitido el "autoplegado". De no ser así, se generarían reglas con recursión infinita al coincidir sus partes izquierda y derecha.

Como probaremos en la siguiente sección, los tipos de plegado conjuntivo que presentamos para el contexto integrado se benefician de los siguientes hechos:

- Las sustituciones utilizadas en las definiciones de plegado son simples emparejadores de términos, de forma similar a los plegados funcionales y algunas propuestas de plegado para programas lógicos puros.
- No es necesario exigir que el subtérmino a plegar sea un redex desde el punto de vista del *narrowing* necesario, lo que flexibiliza la definición al tiempo que, de alguna manera, independiza la regla de transformación de la estrategia de evaluación considerada.
- Se simplifica ligeramente el control sobre las variables de las reglas involucradas en el plegado con respecto a los casos funcional y lógico puro, debido al hecho de que la clase de programas considerados no contienen reglas con variables extra.

Antes de presentar los resultados de corrección y completitud para el plegado necesario no reversible, pasamos a demostrar una propiedad importante del mismo, que se revelará crucial para demostrar la corrección y completitud fuertes.

Teorema 4.3.2 *Sea $\mathcal{R}$ un programa inductivamente secuencial y sea $R \in \mathcal{R}$ una regla del mismo. El plegado no reversible de $\mathcal{R}$ con respecto a R es también un programa inductivamente secuencial.*

DEMOSTRACIÓN. El carácter inductivamente secuencial de un programa únicamente se caracteriza por el formato de las partes izquierdas de sus reglas. Por la Definición 4.3.1, el programa transformado se obtiene a partir del original al eliminar una regla $R = (l \to r) \in \mathcal{R}$ y añadir en su lugar una nueva regla de la forma $R' = (l \to r[l'\theta]_p)$ (que, además, no contiene variables extra debido a la restricción impuesta en la Definición 4.3.1). Como las partes izquierdas de las reglas presentes en ambos programas coinciden y R es inductivamente secuencial, entonces $\mathcal{R}'$ también lo es. □

El siguiente resultado avanza un tipo de corrección del plegado no reversible, restringido a derivaciones de reescritura.

Lema 4.3.3 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $R \in \mathcal{R}$ una regla del mismo tal que el plegado no reversible de $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Si $e \to^* true$ en $\mathcal{R}'$ entonces $e \to^* true$ en $\mathcal{R}$.*

DEMOSTRACIÓN. Demostramos este resultado por inducción sobre el número n de pasos de reescritura en la derivación $\mathcal{D} = [e \to^* true]$.

Caso base. Sea $n = 0$. Entonces tenemos que $e = true$ y el resultado se sigue trivialmente.

Caso inductivo. Si $n > 0$ tenemos que $\mathcal{D}$ no es una secuencia de reducciones vacía. Si el primer paso de reducción se da con una regla R^* que pertenece tanto a $\mathcal{R}$ como a $\mathcal{R}'$, entonces, por la hipótesis de inducción, la secuencia de reducciones completa puede simularse en $\mathcal{R}$.

En caso contrario, la regla usada en el primer paso de reducción de $\mathcal{D}$ es el resultado de una operación de plegado no reversible, existiendo dos reglas $R = (l \to r) \in \mathcal{R}$ y $R' = (l' \to r') \in \mathcal{R}$ tal que R^* es el resultado de plegar R usando R'. Entonces $r|_p = r'\theta$ para alguna posición $p \in Pos(r)$ y sustitución θ, con $R^* = (l \to r[l'\theta]_p)$. Por tanto, tenemos que

$$\mathcal{D} = [e \to_{q,R^*} e[(r[l'\theta]_p)\sigma]_q \to^* true]$$

donde $e|_q = l\sigma$. Como la regla plegante R' pertenece a ambos programas, el siguiente paso de reescritura puede darse en $\mathcal{R}'$:

$$e[(r[l'\theta]_p)\sigma]_q \to_{q.p,R'} e[(r[r'\theta]_p)\sigma]_q = e[(r[r_{|p}]_p)\sigma]_q = e[r\sigma]_q$$

Al ser $\mathcal{R}'$ inductivamente secuencial (por el Teorema 4.3.2), entonces necesariamente es confluente y, por tanto, una posible derivación de e a $true$ en $\mathcal{R}'$ puede construirse como sigue:

$$\mathcal{D}' = [e \to_{q,R^*} e[(r[l'\theta]_p)\sigma]_q \to_{q.p,R'} e[r\sigma]_q \to^* true].$$

Por la hipótesis de inducción, tenemos que $e[r\sigma]_q \to^* true$ en $\mathcal{R}$. Finalmente, ya que $R = (l \to r) \in \mathcal{R}$ y $e|_q = l\sigma$, tenemos que los dos primeros pasos de $\mathcal{D}'$ se pueden simular en $\mathcal{R}$ dando un solo paso con R y por tanto:

$$e \to_{q,R} e[r\sigma]_q \to^* true$$

lo que completa la demostración.

□

La contrapartida al lema anterior es el siguiente resultado, que enuncia un tipo de completitud del plegado necesario no reversible referido de nuevo a derivaciones de reescritura.

Lema 4.3.4 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $R \in \mathcal{R}$ una regla del mismo tal que el plegado no reversible de $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Si $e \to^* true$ en $\mathcal{R}$ entonces $e \to^* true$ en $\mathcal{R}'$.*

DEMOSTRACIÓN. Sea $\mathcal{D} = [e \to^* true]$ una secuencia de reducciones en $\mathcal{R}$. Basándonos en $\mathcal{D}$, a continuación construimos una secuencia de reducciones de e a $true$ en $\mathcal{R}'$ por inducción sobre la longitud de $\mathcal{D}$.

El caso base obviamente se verifica porque $e = true$. En otro caso, $\mathcal{D}$ es de la forma $\mathcal{D} = [e \to_R e' \to^* true]$ en $\mathcal{R}$. Si $R \in \mathcal{R}'$, entonces el resultado se sigue por la hipótesis de inducción. En otro caso, $R = (l \to r)$ ha sido plegada en $\mathcal{R}$ usando como regla plegante $R' = (l' \to r') \in \mathcal{R}$, obteniéndose así una nueva regla $R^* = (l \to r[l'\theta]_q) \in \mathcal{R}'$ (donde $r|_q = r'\theta$ para alguna posición $q \in Pos(r)$).

Como el primer paso de $\mathcal{D}$ se da con R, entonces $e' = e[r\sigma]_p$, donde $e|_p = l\sigma$ para alguna posición $p \in Pos(e)$. Como

$$e' = s[r\sigma]_p = e[(r[r_{|q}]_q)\sigma]_p = e[(r[r'\theta]_q)\sigma]_p$$

y tanto R^* como R' pertenecen a $\mathcal{R}'$, entonces es posible dar el siguiente par de pasos de reescritura en $\mathcal{R}'$:

$$e \to_{p,R^*} e[(r[l'\theta]_q)\sigma]_p \to_{p.q,R'} e[(r[r'\theta]_q)\sigma]_p = e'$$

Por la hipótesis de inducción, $e' \to^* true$ en $\mathcal{R}'$ lo que, junto con el hecho anterior, implica que

$$e \to_{p,R^*} e'' \to_{p.q,R'} e' \to^* true$$

en $\mathcal{R}'$, como queríamos demostrar. □

La corrección y completitud fuertes del plegado no reversible (al nivel del *narrowing*) se deriva de los dos resultados anteriores (corrección y completitud en un contexto de reescritura) junto con las propiedades de corrección, completitud y optimalidad del *narrowing* necesario (ver de nuevo el Teorema 2.2.15).

Teorema 4.3.5 (corrección y completitud fuertes) *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $R \in \mathcal{R}$ una regla del mismo tal que el plegado no reversible de $\mathcal{R}$ con respecto a R es el programa transformado $\mathcal{R}'$. Sea e una ecuación. Entonces, $e \overset{\text{NN}}{\leadsto}{}^*_\theta$ true es una derivación de narrowing necesario en $\mathcal{R}$ si y sólo si existe una derivación de narrowing necesario $e \overset{\text{NP}}{\leadsto}{}^*_{\theta'}$ true en $\mathcal{R}'$ tal que $\theta' = \theta$ $[\mathcal{V}ar(e)]$.*

DEMOSTRACIÓN. Distinguimos dos casos:

Corrección y completitud. Primero probamos la corrección de la transformación. Como $e \overset{\text{NN}}{\leadsto}{}^*_\sigma$ *true* en $\mathcal{R}'$, por la corrección del *narrowing* necesario (punto 1 del Teorema 2.2.15), tenemos que $e\sigma \to^* true$ en $\mathcal{R}'$. Por el Lema 4.3.3 (corrección bajo reescritura) existe una secuencia de reescritura $e\sigma \to^* \mathit{true}$ en $\mathcal{R}$. Por tanto, por la completitud del *narrowing* necesario (punto 2 del Teorema 2.2.15), existe una derivación de *narrowing* necesario $e \stackrel{\text{NN}}{\leadsto}^*_{\sigma'} \mathit{true}$ en $\mathcal{R}$ tal que $\sigma' \leq \sigma$ $[\mathcal{V}ar(e)]$, lo que prueba la corrección de la transformación.

La demostración de completitud es perfectamente análoga.

Corrección y completitud fuertes. Comenzamos también con la demostración de la corrección fuerte y la hacemos por contradicción. Asumamos que existe una sustitución σ' computada por la estrategia de *narrowing* necesario para e en $\mathcal{R}'$ tal que no existe ninguna sustitución θ computada por *narrowing* necesario para e en $\mathcal{R}$ con $\theta = \sigma'$ $[\mathcal{V}ar(e)]$ (bajo renombramiento).

Por la corrección del plegado no reversible y la asunción anterior, concluimos que debe existir alguna sustitución σ computada por *narrowing* necesario para e en $\mathcal{R}$ tal que $\sigma < \sigma'$ $[\mathcal{V}ar(e)]$. Entonces, por la completitud del plegado no reversible existe una sustitución θ' computada por la estrategia de *narrowing* necesario para e en $\mathcal{R}'$ tal que $\theta' \leq \sigma$ $[\mathcal{V}ar(e)]$. Como $\theta' \leq \sigma$ $[\mathcal{V}ar(e)]$ y $\sigma < \sigma'$ $[\mathcal{V}ar(e)]$, tenemos que $\theta' < \sigma'$ $[\mathcal{V}ar(e)]$, lo que contradice la independencia de las soluciones computadas por *narrowing* necesario (punto 3 del Teorema 2.2.15).

La demostración de la completitud fuerte se construye de forma similar.

□

En la siguiente sección relajamos una de las condiciones del plegado necesario no reversible que acabamos de describir para flexibilizar su uso y dotarlo de un mayor poder de optimización. En contrapartida, se observará que los esquemas de demostración se complican enormemente, pero sin perderse por ello las mismas propiedades de corrección y completitud fuertes que acabamos de analizar en esta sección.

4.4 Plegado T&S

En esta sección describimos un tipo de plegado no reversible más potente que extiende la noción original de plegado de los programas lógicos expresada inicialmente por Tamaki y Sato [1984] (de ahí su nombre) al caso de programas inductivamente secuenciales con una semántica operacional basada en *narrowing* necesario. En este caso, hemos adoptado la misma nomenclatura que Pettorossi y Proietti [1994, 1998][4].

Este tipo de plegado se caracteriza porque es conjuntivo (al igual que el presentado en la sección anterior) y, además, porque en su definición se permite extraer las reglas

[4]En [Pettorossi y Proietti, 1998], a esta variante del plegado se la denomina *single-folding*.

plegables y plegantes de distintos programas de la secuencia de transformación (a diferencia de las versiones anteriores). Esta última característica es la clave para dotar de toda la potencia de optimización a este último tipo de plegado.

4.4.1 Secuencias de transformación virtuales

Para poder definir el plegado T&S, debemos asumir la existencia de diferentes programas, distribuidos a lo largo de una secuencia de transformación, de tal forma que, para plegar alguna regla del último programa de la secuencia, la regla plegante pueda extraerse de cualquier programa de la misma (no necesariamente del último programa). En la versión original de Tamaki y Sato [1984], la secuencia de transformación se construye aplicando, en cada paso, una regla de transformación escogida de entre un buen número de operaciones (introducción de una definición, plegado, desplegado y distintas variedades del reemplazamiento algebraico). Sin embargo, es en [Kawamura y Kanamori, 1990] donde las demostraciones de corrección del plegado/desplegado descritas en [Tamaki y Sato, 1984] se formulan sobre una clase especial de secuencias en las que sólo puede aplicarse el par básico de transformaciones. El esquema así generado se simplifica enormemente y, sin embargo, no pierde generalidad ya que la adaptación es puramente sintáctica, como veremos más tarde.

Siguiendo este mismo esquema (que extenderemos en el capítulo siguiente), y antes de introducir la noción de plegado T&S, en lo que sigue consideramos la siguiente noción de secuencia de transformación, que llamaremos *virtual* por compatibilidad con la nomenclatura que usaremos en las secciones posteriores:

Definición 4.4.1 (secuencia de transformación virtual) Una secuencia ordenada de programas $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq 0$, es una secuencia de transformación virtual si cumple las siguientes condiciones:

1. $\mathcal{R}_0$ es un SRT inductivamente secuencial dividido en dos conjuntos disjuntos de reglas, $\mathcal{R}_{new}$ y $\mathcal{R}_{old}$, donde los símbolos de función definidos en $\mathcal{R}_{new}$ se llaman símbolos de función *nuevos*, mientras que aquellos símbolos de función definidos en $\mathcal{R}_{old}$ se llaman símbolos de función *viejos*.

2. Los símbolos de función nuevos nunca aparecen en $\mathcal{R}_{old}$ ni tampoco en las partes derechas de las reglas de $\mathcal{R}_{new}$.

3. Todo símbolo de función *nuevo* f está definido en $\mathcal{R}_{new}$ por una sola regla (llamada *regla de definición*) de la forma $f(\overline{t_n}) \to r$, donde $\mathcal{V}ar(f(\overline{t_n})) = \mathcal{V}ar(r)$.

4. Todo programa de la secuencia distinto del inicial se obtiene por la aplicación de una operación de desplegado necesario o plegado T&S sobre el programa inmediatamente precedente.

Obsérvese que, por la segunda condición de la definición anterior, las reglas de $\mathcal{R}_{new}$ nunca son recursivas, lo que facilitará posteriormente las demostraciones de corrección. Por otra parte, la tercera condición de la definición es nueva con respecto a otras aproximaciones, y resulta necesaria para garantizar que la aplicación del plegado nunca puede generar reglas con variables extra en sus partes derechas (lo que, en cierta manera, se relaciona con otras condiciones similares impuestas sobre las variables de las cláusulas plegables y plegantes en el contexto lógico puro). Esta última cuestión es muy importante, en tanto que nuestra noción básica de programa se refiere implítamente a sistemas de reescritura de términos donde no aparecen variables extra en las partes derechas.

Finalmente, queremos hacer notar dos hechos importantes que tienen que ver con la terminología utilizada hasta ahora y que aquí únicamente adelantaremos por mantener una nomenclatura coherente y compatible con lo que veremos en el siguiente capítulo. Por una parte, el nombre que damos a las reglas de $\mathcal{R}_{new}$ (reglas de definición) tiene que ver con otras operaciones de transformación que describiremos más adelante y, más concretamente, con la introducción y eliminación de definiciones. Por otro lado, en [Tamaki y Sato, 1984] se denomina secuencia de transformación virtual a la mayor porción de una secuencia (reordenada) de transformación donde no cambia la signatura de los programas y tan sólo se aplican operaciones de plegado/desplegado. Como nuestro interés actual se centra en secuencias de transformación que cumplen estas condiciones (usando el acertado criterio de Kawamura y Kanamori [1990]), utilizamos aquí el mismo adjetivo para designar a este tipo de secuencias de transformación aunque, por comodidad, en lo que sigue podremos obviarlo cuando quede claro por el contexto (nótese que todavía no hemos introducido ninguna otra regla de transformación distinta de las de plegado/desplegado). De la misma forma, en todo lo que sigue, cuando usemos los términos de desplegado y plegado, nos estaremos refiriendo abreviadamente al desplegado necesario descrito en la Sección 3.4.2 y al plegado T&S que pasamos a definir a continuación.

Definición 4.4.2 (plegado T&S) Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq 0$, una secuencia de transformaciones virtual . Sea $R = (l \to r) \in \mathcal{R}_k$ una regla (la "regla plegable") y sea $R' = (l' \to r') \in \mathcal{R}_j$, $0 \leq j \leq k$, otra regla (la "regla plegante") tal que $r|_p = r'\theta$ para alguna posición $p \in \mathcal{FP}os(r)$, que verifican las siguientes condiciones:

1. $r|_p$ no es un término constructor,
2. o bien l (la parte izquierda de la regla plegable) esta encabezada con un símbolo de función viejo, o bien la regla plegable se obtuvo como resultado de, al menos, un desplegado dentro de la secuencia $\mathcal{R}_0, \ldots, \mathcal{R}_k$, y
3. la regla plegante R' es una regla de definición[5].

[5]Es importante destacar que una regla de definición mantiene su estatus siempre y cuando per-

Entonces, podemos obtener el programa $\mathcal{R}_{k+1}$ a partir del programa $\mathcal{R}_k$ como sigue:

$$\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{l \to r[l'\theta]_p\}.$$

Obsérvese que las condiciones de aplicabilidad 2 y 3 garantizan que no está permitido el "autoplegado", es decir, la posibilidad de que una regla sea incorrectamente plegada con respecto a sí misma [Pettorossi y Proietti, 1994]. De hecho, implícitamente se está afirmando que la regla plegable R nunca puede ser una regla de definición, mientras que la regla plegante R' lo ha de ser siempre.

Existen varios puntos que resulta interesante comentar sobre nuestra Definición 4.4.2 de plegado T&S:

- A diferencia de la regla de desplegado necesario, el subérmino seleccionado para ser plegado en una regla plegable no necesariamente ha de ser un redex desde el punto de vista de *narrowing* necesario, lo que en cierta manera puede verse como un grado de independencia de la regla de plegado con respecto a la estrategia de *narrowing* que se usa como semántica operacional de los programas considerados. Esta característica no sólo flexibiliza de forma segura la propia definición de plegado, sino que en ocasiones se revela absolutamente imprescindible de cara a obtener optimizaciones efectivas en los programas transformados, tal y como veremos en el capítulo dedicado a las estrategias de transformación. Allí mostraremos algunos ejemplos donde no se consigue ninguna optimización si sólo se consideran como plegables aquellos redexes que sean necesarios. Nótese, por otra parte, que esta condición no se podría relajar si pretendiésemos dotar a la operación de plegado de un carácter reversible con respecto al desplegado necesario.

- En contraste con el plegado reversible de la Sección 4.2, ahora no es necesaria ninguna generalización de las reglas plegables. De hecho, la sustitución θ de la Definición 4.4.2 no es un unificador en el sentido amplio de la palabra, sino un simple emparejador. Como ya hemos adelantado, este hecho se repite de forma similar en otras reglas de plegado para programas lógicos, que sorprendentemente se definen también en este mismo "estilo funcional" (ver, por ejemplo, [Bossi y Cocco, 1993; Kawamura y Kanamori, 1990; Pettorossi y Proietti, 1996b; Tamaki y Sato, 1984]). Lo importante de esta simplicación es que, con ella, todavía es posible producir optimizaciones potentes con un coste computacional muy bajo.

---

manezca sin cambios, i.e., una vez que la regla sufre una transformación (por ser usada como regla desplegable en algún paso de desplegado, ya que nunca puede ser plegada) deja de ser considerada como regla de definición en el resto de la secuencia.

- Finalmente, ahora podemos clarificar la condición que exigimos en la Definición 4.4.1 sobre la coincidencia de los conjuntos de variables en las partes izquierdas y derechas de las reglas de definición (obsérvese la estrecha conexión entre esta condición y las homólogas definidas en [Tamaki y Sato, 1984; Kawamura y Kanamori, 1990; Pettorossi y Proietti, 1994, 1998], etc.). Esencialmente, esta restricción evita la posibilidad de considerar una regla de la forma $l \to r$, con $\mathcal{V}ar(r) \subset \mathcal{V}ar(l)$[6], como regla plegante, ya que en este caso se podrían introducir variables extra en la parte derecha de la regla resultante, lo que está explícitamente prohibido. Para ilustrar este hecho, basta con considerar el plegado de la regla $\mathtt{f(X)} \to \mathtt{h(X)}$ con respecto a la regla $\mathtt{g(X,Y)} \to \mathtt{h(X)}$, que produce la regla plegada no permitida $\mathtt{f(X)} \to \mathtt{g(X,Y)}$.

Comparando nuestra definición con la original de Tamaki y Sato [1984], encontramos que las condiciones de aplicabilidad son bastante similares, aunque en nuestro caso las partes izquierdas de la reglas plegantes son de la forma $f(t_1, \ldots, t_n)$ donde $t_1, \ldots, t_n$ son términos constructores (lineales) y no necesariamente variables. Por otra parte, los resultados de corrección asociados al observable de las respuestas computadas que nosotros obtenemos, son similares a los que Kawamura y Kanamori [1990] demuestran también para el caso lógico. Pero, además, nuestro plegado preserva también los valores (constructores básicos) obtenidos en derivaciones de éxito, de forma similar a lo que ocurre en las computaciones funcionales donde, en contrapartida, no existe la noción de respuesta computada. En particular, nuestro plegado T&S es similar al plegado con marcas de Sands [1996] para programas funcionales perezosos. Sin embargo nuestras demostraciones de corrección y completitud (con resultados más fuertes) son bastante más complejas, debido al hecho de tener que preservar dos tipos de semánticas: la de los valores constructores básicos y la de respuestas computadas constructoras, asociadas respectivamente a las componentes funcional y lógica de un lenguaje integrado.

A continuación pasamos a presentar los principales resultados teóricos que acompañan a la nueva regla de plegado T&S. Encontraremos aquí una dificultad nueva con respecto a todas las otras reglas de transformación (tanto las diferentes variedades de plegado como las de desplegado) vistas hasta ahora: en el plegado que estamos estudiando aparecen por primera vez involucrados tres programas diferentes y no sólo dos (el original y el transformado), que venía siendo lo habitual. Este hecho tiene repercusiones importantes a la hora de desarrollar las demostraciones de corrección y completitud, ya que, por razones técnicas, las pruebas de inducción se complican considerablemente (al tener que considerar una secuencia completa de transformaciones virtuales).

[6]Nótese que no consideramos el caso $\mathcal{V}ar(l) \subset \mathcal{V}ar(r)$ puesto que no admitimos como programas aquellos sistemas de reescritura de términos con variables extra en las partes derechas de sus reglas.

4.4.2 Propiedades

Antes de presentar nuestros resultados de corrección y completitud para el plegado T&S, pasamos a demostrar una propiedad esperada de las secuencias de transformaciones virtuales, que se revelará crucial para demostrar la corrección y completitud fuerte del sistema de transformación en general y de la propia regla de plegado T&S en particular.

Teorema 4.4.3 *Todo programa perteneciente a una secuencia de transformación virtual,* $(\mathcal{R}_0, \ldots, \mathcal{R}_n), n > 0$*, es inductivamente secuencial.*

DEMOSTRACIÓN. Probamos el resultado por inducción sobre n. Como el caso $n = 0$ es obvio (ya que, por la Definición 4.4.1, el programa inicial de toda secuencia de transformación virtual pertenece a esta clase de programas), procedemos con el caso inductivo. Por la hipótesis de inducción, tenemos que $\mathcal{R}_{n-1}$ es inductivamente secuencial. Entonces, para todo símbolo de función definido f/n en $\mathcal{R}_{n-1}$, existe un árbol definicional para f en $\mathcal{R}_{n-1}$. Ahora demostramos que este hecho también se cumple para $\mathcal{R}_n$, y lo hacemos distinguiendo dos casos, dependiendo de cuál ha sido la regla transformación aplicada sobre $\mathcal{R}_{n-1}$:

Desplegado necesario. Este caso se cumple trivialmente por el Teorema 3.4.9 del capítulo anterior, que demuestra esta misma propiedad cuando se realiza una transformación de desplegado necesario sobre un programa inductivamente secuencial.

Plegado T&S. El carácter inductivamente secuencial de un programa únicamente se caracteriza por el formato de las partes izquierdas de sus reglas. Por la Definición 4.4.2, el programa $\mathcal{R}_n$ se obtiene a a partir de $\mathcal{R}_{n-1}$ al eliminar una regla $R = (l \to r) \in \mathcal{R}_{n-1}$ y añadir en su lugar una nueva regla de la forma $R' = (l \to r[l'\theta]_p)$ (que, además, no contiene variables extra debido a la restricción impuesta en la Definición 4.4.1). Como las partes izquierdas de las reglas presentes en ambos programas coinciden, entonces el resultado se sigue por la hipótesis de inducción.

□

En lo que sigue, nuestras técnicas de demostración se inspiran en los esquemas originales de Tamaki y Sato [1984] y también en la subsiguiente extensión de Kawamura y Kanamori [1990] para la semántica de respuestas computadas. Mantenemos por ello su notación (al igual que ya hemos hecho con el concepto de secuencia de transformación virtual) introduciendo una noción apropiada de "consistencia de rangos" cuyo significado pasamos a describir.

Intuitivamente, una secuencia de transformaciones de plegado/desplegado es correcta si se dan “como mucho tantos pasos de plegado como de desplegado” o, equivalentemente, si “los pasos hacia atrás en las computaciones (como hace el plegado) no prevalecen sobre los pasos hacia adelante en las mismas (como hace el desplegado)” [Pettorossi y Proietti, 1994; Sands, 1996]. Esto significa, esencialmente, que debe existir algún tipo de medida de “coste computacional” que se vaya reduciendo (o que al menos no se incremente) a lo largo de la secuencia de transformación. En la literatura encontramos diferentes formulaciones para esta clase de medida: *rango de un objetivo* en [Tamaki y Sato, 1984], *peso de un árbol de prueba* en [Kawamura y Kanamori, 1990] o la noción de *mejora (improvement)* en [Sands, 1996]. En nuestro contexto, introducimos la noción de *rango de un término* para medir el coste computacional de reducir un término dado hasta su forma normal (constructora). Nuestra definición es ligeramente más compleja que otras anteriores (ver [Tamaki y Sato, 1984]) ya que debemos tener en cuenta las particularidades de la estrategia de *narrowing* necesario que utilizamos para evaluar los términos. En lo que sigue, denotamos con $length(\mathcal{D})$ el número de pasos de reescritura dados en la secuencia $\mathcal{D}$.

Definición 4.4.4 (rango de un término) Sea $\mathcal{R}_0$ el programa inicial de una secuencia de transformación virtual. Sea s un término tal que $s \to^* t$ en $\mathcal{R}_0$, donde t es constructor. Sea $\mathcal{D} = [s \to^* s']$ una secuencia de reducciones en $\mathcal{R}_0$ que sólo usa reglas de definición, de tal forma que no son aplicables más reglas de definición sobre s'. Sean $(\mathcal{D}_0, \ldots, \mathcal{D}_m)$, $m \geq 0$, todas las posibles secuencias de reducción necesarias (más externas) de s' a t en $\mathcal{R}_0$ con $n = max(length(\mathcal{D}_0), \ldots, length(\mathcal{D}_m))$. Entonces, $rank(s) = n$.

Nótese que en la definición anterior exigimos que $(\mathcal{D}_0, \ldots, \mathcal{D}_m)$, $m \geq 0$, sean secuencias de reducciones necesarias, es decir, donde sólo se reducen redexes necesarios. Esta restricción es necesaria para asegurar que el rango de un término nunca puede ser infinito y que, por tanto, está bien definido, ya que es suficiente con reducir un número finito de redexes necesarios para obtener la forma normal constructora (si existe) de un término cualquiera [Huet y Lévy, 1992].

Ejemplo 38 Consideremos el siguiente programa inicial $\mathcal{R}_0$

$$\begin{array}{lcll} \mathtt{f(X)} & \to & \mathtt{0} & (R_1) \\ \mathtt{g(X)} & \to & \mathtt{g(X)} & (R_2) \end{array}$$

La única reducción necesaria de $\mathtt{f(g(X))}$ a 0 es:

$$\underline{\mathtt{f(g(X))}} \to_{\Lambda,\mathtt{R_1}} \mathtt{0}$$

Por tanto el rango de $rank(\mathtt{f(g(X))}) = \mathtt{1}$, mientras que sería infinito si considerásemos la mayor de las secuencias de reducciones arbitrarias (por la continua aplicación de la regla R_2).

En nuestra definición de rango hemos considerado además la secuencia de reducción necesaria *más larga*, ya que el desplegado necesario de un programa puede suponer, en general, un incremento de la longitud de las secuencias de reducciones necesarias arbitrarias en el programa transformado, corriéndose así el riesgo de que el rango de los términos no se preserve a través del desplegado (como veremos posteriormente al enunciar el concepto de invariantes). Pero, además, también los pasos de plegado pueden introducir pasos de reducción adicionales en las posibles reducciones necesarias para un término dado. En este caso, resolvemos el problema no teniendo en cuenta el número (finito) de pasos dados con reglas de definición. De esta forma, tanto el plegado como el desplegado preservan el rango de los términos como demostraremos seguidamente.

La corrección del plegado T&S no puede demostrarse de forma aislada al estilo de las reglas anteriores, sino que debe desarrollarse en el seno de una secuencia de transformación virtual, lo que involucra también de forma directa a la regla de desplegado necesario. Con nuestra adaptación de los esquemas de demostración de Tamaki y Sato [1984] y Kawamura y Kanamori [1990], probamos los resultados de corrección y completitud mostrando que los siguientes invariantes se mantienen en todos los programas de una secuencia de transformación (virtual).

Definición 4.4.5 (invariantes) Los invariantes de un programa $\mathcal{R}_i$ perteneciente a una secuencia de transformación virtual $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $0 \leq i \leq k$, se definen como sigue:

I1. Dada una ecuación e, $e \to^* true$ en $\mathcal{R}_i$ si y sólo si $e \to^* true$ en $\mathcal{R}_0$.

I2. Sea $\mathcal{D} = [e_0 \to^* e_n]$, $n \geq 0$, una secuencia de reducciones necesarias en $\mathcal{R}_i$ tal que e_0 es una ecuación y $e_n = true$. Para todo paso de reducción $e_j \to_R e_{j+1}$ en $\mathcal{D}$, $0 \leq j \leq n-1$, tenemos que $rank(e_j) = rank(e_{j+1})$ si R es una regla de definición, o $rank(e_j) > rank(e_{j+1})$ en caso contrario.

Antes de probar que los invariantes se preservan a lo largo de toda una secuencia de transformación, es preciso formalizar una propiedad del plegado T&S que ya hemos mencionado de forma intuitiva anteriormente.

Proposición 4.4.6 *Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_k, \mathcal{R}_{k+1})$, $k \geq 0$, una secuencia de transformación virtual. Si $\mathcal{R}_{k+1}$ se obtiene por plegado T&S usando la regla plegable $R \in \mathcal{R}_k$, entonces R no es una regla de definición.*

Demostración. Por la Definición 4.4.2, tenemos que la regla plegable $R = (l \to r) \in \mathcal{R}_k$ verifica que o bien l está encabezado por un símbolo de función viejo o bien R es el resultado de algún desplegado en la secuencia $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$. Ambas condiciones son incompatibles con el concepto de regla de definición, ya que por la Definición 4.4.1

sabemos que toda regla de definición define necesariamente a un símbolo de función nuevo, al tiempo que adoptamos el convenio de que toda transformación que afecte a una regla de definición (como es el caso del desplegado), le hace perder su estatus. Por tanto, los conjuntos de reglas plegables y de definición son disjuntos, como queríamos demostrar. □

Ahora ya estamos en condiciones de demostrar la preservación de algunos invariantes en las secuencias de transformación virtuales, lo que será la clave para demostrar la corrección del sistema completo en un contexto de reescritura y, posteriormente, elevarlo al caso del *narrowing*. Comenzamos probando dicho resultado en el programa inicial de una secuencia.

Lema 4.4.7 *Sea $\mathcal{R}_0$ el programa inicial de una secuencia de transformación virtual. Entonces, los invariantes (I1) e (I2) se verifican en $\mathcal{R}_0$.*

Demostración. Como el invariante (I1) se da trivialmente en $\mathcal{R}_0$, procedemos directamente con la demostración del invariante (I2) en $\mathcal{R}_0$. Sea $\mathcal{D} = [e_0 \to^* e_n]$, $n \geq 0$, una secuencia de reducciones necesarias en $\mathcal{R}_0$ tal que e_0 es una ecuación y $e_n = true$. Debemos probar que para cada paso de reducción $e_j \to_R e_{j+1}$ en $\mathcal{D}$, $0 \leq j \leq n-1$, se verifica que $rank(e_j) = rank(e_{j+1})$ si R es una regla de definición o $rank(e_j) > rank(e_{j+1})$ en otro caso. Demostramos el resultado por inducción sobre $n = length(\mathcal{D})$.

$n = 0$. Este caso es inmediato ya que $e_0 = true$.

$n > 0$. Entonces, tenemos que $\mathcal{D} = [e_0 \to_{q,R} e_1 \to^* true]$. Si R es una regla de definición, por la definición de rango tenemos que $rank(e_0) = rank(e_1)$. De esta forma, el resultado se sigue por la hipótesis de inducción.

Si R no es una regla de definición, entonces tenemos el siguiente par de secuencias de reducción en $\mathcal{R}_0$:

$$\mathcal{D}_0 = [e_0 \to^* e'_0] \quad \text{y} \quad \mathcal{D}_1 = [e_1 \to^* e'_1]$$

donde solamente se han usado reglas de definición en $\mathcal{D}_0$ y $\mathcal{D}_1$, y no son aplicables más reglas de definición sobre e'_0 ni e'_1. Ahora, obsérvese que existe es un redex necesario $e_0|_q$ encabezado por un símbolo de función viejo, ya que e_0 puede ser reducido por una regla R que no es de definición. Como solamente se usan reglas de definición en $\mathcal{D}_0$ y éstas cumplen $\mathcal{V}ar(l) = \mathcal{V}ar(r)$ para toda regla de definición $l \to r$, sabemos que existe al menos un descendiente de $e_0|_q$ en e'_0. Además, como $e_0|_q$ es un redex necesario en e_0, entonces tenemos que (algunos de) sus descendientes también son redexes necesarios en e'_0. Asumamos que tales redexes necesarios son $e'_0|_{p_1}, \ldots, e'_0|_{p_k}$, $\{p_1, \ldots, p_k\} \subseteq \mathcal{P}os(e'_0)$, $k > 0$. Por tanto, la siguiente secuencia de reducciones necesarias $e'_0 \to_{p_1,R} \ldots \to_{p_k,R} e'_1$ puede darse en $\mathcal{R}_0$ con simplemente reducir repetidamente todos los redexes necesarios usando la regla R (que no es de definición).

Por la Definición 4.4.4, sabemos que $rank(e_0) = rank(e'_0)$ y $rank(e_1) = rank(e'_1)$. Asumamos que la mayor secuencia de reducciones necesarias para e'_1 a *true* es $\mathcal{D}'_1 = [e'_1 \to^m true]$. Entonces $rank(e_1) = rank(e'_1) = m$. Por tanto, como e'_0 admite una secuencia de reducciones necesarias a *true* de la forma $e'_0 \to_{p,R} e'_1 \to^m true$, entonces el rango de e'_0 es como mínimo $m + 1$. Esto implica que $rank(e_0) = rank(e'_0) \geq m + 1 > m = rank(e'_1) = rank(e_1)$. Finalmente, como $rank(e_0) > rank(e_1)$, el teorema queda demostrado por la hipótesis de inducción.

□

Los siguientes lemas demuestran que los invariantes se preservan también en el resto de programas transformados aplicando pasos de plegado necesario T&S y de desplegado necesario. El primero de ellos refleja un tipo de corrección (sujeta al invariante (I1)) en un contexto de reescritura.

Lema 4.4.8 *Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$, una secuencia de transformación virtual y e una ecuación. Si el invariante (I1) se verifica en $\mathcal{R}_i$ entonces, si $e \to^* true$ en $\mathcal{R}_{i+1}$ entonces $e \to^* true$ en $\mathcal{R}_i$.*

DEMOSTRACIÓN. Sea e una ecuación tal que $\mathcal{D} = [e \to^* true]$ en $\mathcal{R}_{i+1}$. Nuestra demostración es por inducción sobre la longitud de $\mathcal{D}$.

$n = 0$. Este caso es inmediato ya que $e = true$.

$n > 0$. Sea $\mathcal{D} = [e \to_{q,R^*} e' \to^* true]$. Si $R^* \in \mathcal{R}_i$, entonces el resultado se sigue por la hipótesis de inducción. En otro caso, R^* se ha obtenido a partir de $\mathcal{R}_i$ por un paso de desplegado necesario o plegado T&S. En el primer caso, el resultado se verifica por el Lema 3.4.11 que establece la correción del desplegado necesario. En el segundo caso, existen dos reglas $R = (l \to r) \in \mathcal{R}_i$ y $R' = (l' \to r') \in \mathcal{R}_0$ tales que R^* es el resultado de plegar R usando R'. Entonces $r|_p = r'\theta$ para alguna posición $p \in Pos(r)$ y sustitución θ, con $R^* = (l \to r[l'\theta]_p)$. Por tanto, tenemos que

$$\mathcal{D} = [e \to_{q,R^*} e[(r[l'\theta]_p)\sigma]_q = e' \to^* true]$$

donde $e|_q = l\sigma$. Por la hipótesis de inducción, sabemos que $e[(r[l'\theta]_p)\sigma]_q \to^*$ *true* en $\mathcal{R}_i$. Como el invariante (I1) se mantiene en $\mathcal{R}_i$, tenemos también en $\mathcal{R}_0$ que $e[(r[l'\theta]_p)\sigma]_q \to^* true$. Por la Definición 4.4.2, la regla plegante $R' = (l' \to$

r') es una regla de definición y, por tanto, $R' \in \mathcal{R}_0$. De esta forma, el siguiente paso de reescritura puede darse en $\mathcal{R}_0$:

$$e[(r[l'\theta]_p)\sigma]_q \to_{q.p,R'} e[(r[r'\theta]_p)\sigma]_q = e[(r[r_{|p}]_p)\sigma]_q = e[r\sigma]_q$$

Como $\mathcal{R}_0$ es inductivamente secuencial, entonces es confluente y, por tanto, tenemos:

$$e[(r[l'\theta]_p)\sigma]_q \to_{q.p,R'} e[r\sigma]_q \to^* true$$

Aplicando el invariante (I1) de nuevo, tenemos que $e[r\sigma]_q \to^* true$ en $\mathcal{R}_i$. Finalmente, ya que $R = (l \to r) \in \mathcal{R}_i$ y $e|_q = l\sigma$, se cumple

$$e \to_{q,R} e[r\sigma]_q \to^* true$$

en $\mathcal{R}_i$, lo que completa la demostración.

$\square$

Antes de continuar con la demostración de los invariantes en una secuencia de transformación virtual, pasamos a definir el orden bien fundado $>>$ sobre ecuaciones (e, e') que pueden reducirse a *true* en $\mathcal{R}_0$ (y, obviamente, también en $\mathcal{R}_i$, debido al invariante (I1)) como sigue.

Definición 4.4.9 Sea $\mathcal{R}_0$ el programa inicial de una secuencia de transformación virtual y sean e y e' dos ecuaciones reducibles a *true* en $\mathcal{R}_0$. Decimos que $e >> e'$ si y sólo si

1. $rank(e) > rank(e')$ o bien

2. $rank(e) = rank(e')$ y existe alguna regla de definición en $R \in \mathcal{R}_0$ tal que $e \to_R e'$.

Consideramos que *true* es el elemento minimal bajo el orden bien fundado $>>$.

El siguiente lema puede verse como un tipo de completitud (sujeta a los invariantes (I1) e (I2)) para las transformaciones involucradas en una secuencia virtual, de nuevo restringida aún a un contexto de reescritura.

Lema 4.4.10 *Sea* $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$*,* $i \geq 0$*, una secuencia de transformación virtual y* e *una ecuación. Si los invariantes (I1) e (I2) se verifican en* $\mathcal{R}_i$ *entonces, si* $e \to^* true$ *en* $\mathcal{R}_i$ *se cumple que* $e \to^* true$ *en* $\mathcal{R}_{i+1}$.

Demostración. Sea $\mathcal{D} = [e \to^* true]$ una secuencia de reducciones necesarias en $\mathcal{R}_i$. Como el invariante (I2) se da en $\mathcal{R}_i$ tenemos que, para todo paso de reducción $e_j \to_R e_{j+1}$ en $\mathcal{D}$, $rank(e_j) = rank(e_{j+1})$ si R es una regla de definición o $rank(e_j) > rank(e_{j+1})$ en otro caso. Esto implica que $e_j >> e_{j+1}$. Ahora construimos una secuencia de reducciones de e a $true$ en $\mathcal{R}_{i+1}$ por inducción sobre el orden bien fundado $>>$.

El caso base obviamente se verifica porque $e = true$. En otro caso, $\mathcal{D}$ es de la forma $\mathcal{D} = [e \to_R e' \to^* true]$ en $\mathcal{R}_i$, donde $e >> e'$ por el invariante (I2). Si $R \in \mathcal{R}_{i+1}$, entonces el resultado se sigue por la hipótesis de inducción. En otro caso, distinguimos dos posibilidades:

1. Si R es desplegada en $\mathcal{R}_i$, tenemos que existe la siguiente secuencia de reducciones necesarias en $\mathcal{R}_i$ (con un argumento similar al utilizado en la demostración del Lema 3.4.16 que establece la completitud del desplegado necesario en un contexto de reescritura):

   $$\mathcal{D} = [e \to_{p,R} e' \to_{p.q,R'} e'' \to^* true]$$

   en $\mathcal{R}_i$, tal que el resultado de desplegar $R = (l \to r) \in \mathcal{R}_i$ usando $R' \in \mathcal{R}_i$ a la posición $q \in Pos(r)$ es la nueva regla $R^* \in R_{i+1}$, y $e \to_{p,R^*} e''$. Obsérvese que R y R' no son reglas de definición al mismo tiempo, ya que una regla de definición siempre define un símbolo de función nuevo por medio de símbolos de función viejos. Además, como el invariante (I2) se mantiene en $\mathcal{R}_i$, tenemos que $e >> e' >> e''$. Por la hipótesis de inducción, $e'' \to^* true$ en $\mathcal{R}_{i+1}$ y, por tanto, $e \to_{p,R^*} e'' \to^* true$ en $\mathcal{R}_{i+1}$.

2. Si $R = (l \to r)$ es plegada en $\mathcal{R}_i$, por la Proposición 4.4.6, sabemos que R no es una regla de definición. Como $e >> e'$ y R no es una regla de definición, por (I2) concluimos necesariamente que $rank(e) > rank(e')$.

   Además, existe una regla plegante (que es una regla de definición) $R' = (l' \to r') \in \mathcal{R}_0$ tal que el resultado de plegar R usando R' es la regla $R^* = (l \to r[l'\theta]_q) \in \mathcal{R}_{i+1}$ (donde $r|_q = r'\theta$ para alguna posición $q \in Pos(r)$).

   Como el primer paso de $\mathcal{D}$ se da con R, entonces $e' = e[r\sigma]_p$, donde $e|_p = l\sigma$ para alguna posición $p \in Pos(e)$. Por (I1), $e' \to^* true$ en $\mathcal{R}_0$. Como $R' \in \mathcal{R}_0$ y

   $$e' = s[r\sigma]_p = e[(r[r_{|q}]_q)\sigma]_p = e[(r[r'\theta]_q)\sigma]_p$$

   entonces el siguiente paso de reducción puede darse en $\mathcal{R}_0$:

   $$e'' = e[(r[l'\theta]_q)\sigma]_p \to_{p.q,R'} e[(r[r'\theta]_q)\sigma]_p = e'$$

Así, $e'' \to_{p.q,R'} e' \to^* true$ en $\mathcal{R}_0$ y, por el invariante (I1), $e'' \to^* true$ en $\mathcal{R}_i$ también.

Como R' es una regla de definición y $e'' \to_{p.q,R'} e'$, tenemos que $rank(e'') = rank(e')$. Además, como hemos visto previamente que $rank(e) > rank(e')$, concluimos que $rank(e) > rank(e'')$ y, de esta forma, $e >> e''$. Por la hipótesis de inducción, tenemos que $e'' \to^* true$ en $\mathcal{R}_{i+1}$. Finalmente, como $e \to_{p,R^*} e[(r[l'\theta]_q)\sigma]_p = e''$, tenemos que $e \to_{p,R^*} e'' \to^* true$ en $\mathcal{R}_{i+1}$, lo que completa la demostración.

□

La siguiente proposición es necesaria para demostrar el Lema 4.4.12.

Proposición 4.4.11 *Sea* $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$, *una secuencia de transformación virtual. Sea* e *una ecuación tal que* $e \to^* true$ *en* $\mathcal{R}_{i+1}$. *Si* $e|_p$ *es un redex necesario en* e *con respecto a* $\mathcal{R}_{i+1}$, *entonces* $e|_p$ *es un redex necesario en* e *con respecto a* $\mathcal{R}_i$.

Demostración. (esquema) El resultado se sigue por inducción estructural sobre e, ya que el desplegado únicamente puede introducir nuevos redexes necesarios en e (debido a la instanciación de algunas partes izquierdas de las reglas de un programa) y el plegado no cambia los redexes necesarios de e ya que las partes izquierdas permanecen inalteradas y el termino que se pliega, además de no ser constructor (condición 1 de la Definición 4.4.2), se sustituye por otro con exactamente el mismo conjunto de variables (las reglas plegantes son conservativas). □

Lema 4.4.12 *Sea* $(\mathcal{R}_0, \ldots, \mathcal{R}_i, \mathcal{R}_{i+1})$, $i \geq 0$ *una secuencia de transformación virtual. Si los invariantes (I1) e (I2) se verifican en* $\mathcal{R}_i$, *entonces (I2) se verifica en* $\mathcal{R}_{i+1}$.

Demostración. Probamos que (I2) se preserva en toda secuencia de reducciones necesaria $\mathcal{D} = [e \to^* true]$ en $\mathcal{R}_{i+1}$. La demostración se plantea por inducción sobre el orden bien fundado $>>$ introducido en la Definición 4.4.9.

El caso base es inmediato ya que $e = true$. En otro caso, tenemos que $\mathcal{D} = [e \to_{p,R^*} e' \to^* true]$ es una secuencia de reducciones necesarias en $\mathcal{R}_{i+1}$. Ahora distinguimos tres casos:

1. Si $\mathcal{R}^* \in \mathcal{R}_i$, por (I1), tenemos que $e \to_{p,R^*} e' \to^* true$ en $\mathcal{R}_i$. Por la Proposición 4.4.11, $e \to_{p,R^*} e'$ es un paso de reescritura necesaria en $\mathcal{R}_i$. Combinando esto con (I2), tenemos que $e \to_{p,R^*} e' \to^* true$ es una secuencia de reducciones necesarias en $\mathcal{R}_i$ y $e >> e'$. Entonces el resultado se sigue por la hipótesis de inducción.

2. En otro caso, si $R^* \in \mathcal{R}_{i+1}$ es el resultado de desplegar $R = (l \to r) \in \mathcal{R}_i$ usando $R' \in \mathcal{R}_i$ sobre alguna posición $q \in Pos(r)$, entonces existe la siguiente secuencia de reducciones necesarias (que preserva (I2)) en $\mathcal{R}_i$ (por un argumento similar al de la demostración del Lema 3.4.16):

$$e \to_{p,R} e'' \to_{p.q,R'} e' \to^* t$$

Obsérvese que R y R' no son simultáneamente reglas de definición, ya que una regla de definición siempre define un símbolo de función nuevo por medio de símbolos de función viejos. Además, como el invariante (I2) se preserva en $\mathcal{R}_i$, tenemos que $e >> e'' >> e'$. Combinando esto con el hecho de que R y R' no son reglas de definición al mismo tiempo, podemos concluir con que $rank(e) > rank(e')$. Uniendo todas las piezas tenemos:

   (a) $e \to_{p,R^*} e'$ es un paso de reescritura necesaria dado con la regla $R^* \in \mathcal{R}_{i+1}$ (que no es de definición) verificándose que $rank(e) > rank(e')$, lo que preserva (I2), y

   (b) por la hipótesis de inducción, ya que $e >> e'$, sabemos que $e' \to^* true$ es una secuencia de reducciones necesarias que preservan (I2) en $\mathcal{R}_{i+1}$.

3. Finalmente, si no se da ninguno de los casos anteriores, tenemos que $R^* \in \mathcal{R}_{i+1}$ se ha obtenido por un paso de plegado, i.e., existe una regla plegable $R = (l \to r) \in \mathcal{R}_i$ y una regla plegante $R' = (l \to r) \in \mathcal{R}_0$ tal que $R^* = (l \to r[l'\theta]_q) \in \mathcal{R}_{i+1}$ (donde $r|_q = r'\theta$ para alguna posición $q \in Pos(r)$). Por las condiciones que evitan el autoplegado en la Definición 4.4.2 y la Proposición 4.4.6, sabemos que R no es una regla de definición mientras que R' sí lo es.

   Por (I1), tenemos que $e \to_{p,R} e'' \to^* true$ en $\mathcal{R}_i$, donde (para alguna sustitución σ tal que $e|_p = l\sigma$)

$$e'' = e[r\sigma]_p = e[(r[r_{|q}]_q)\sigma]_p = e[(r[r'\theta]_q)\sigma]_p$$

   Por la Proposición 4.4.11, $e \to_{p,R} e''$ es un paso de reescritura necesaria en $\mathcal{R}_i$. Entonces, como (I2) se preserva en $\mathcal{R}_i$, concluimos que $e >> e'$. Ya que $e >> e''$ y R no es una regla de definición, tenemos que $rank(e) > rank(e'')$.

   Por (I1) tenemos que e'' puede ser reducido a $true$ en $\mathcal{R}_0$. Además, existe también la siguiente secuencia de rescritura en $\mathcal{R}_0$:

$$e' = e[(r[l'\theta]_q)\sigma]_p \to_{p.q,R'} e[(r[r'\theta]_q)\sigma]_p = e'' \to^* true$$

   Ya que $e' \to_{p.q,R'} e''$ es un paso de reducción dado con una regla de definición en $\mathcal{R}_0$, tenemos que $rank(e'') = rank(e')$. Combinando esto con el hecho de que

$rank(e) > rank(e'')$, podemos garantizar que $rank(e) > rank(e')$. Por tanto, $e >> e'$ y, por la hipótesis de inducción, $e \to_{p,R^*} e' \to^* true$ es una secuencia de reducciones necesarias que preserva (I2) en $\mathcal{R}_{i+1}$.

□

La siguiente proposición resume y conecta todos los resultados sobre invariantes probados hasta ahora.

Proposición 4.4.13 *Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, una secuencia de transformación virtual. Entonces, los invariantes (I1) e (I2) se verifican en $\mathcal{R}_i$, para todo $i = 0, \ldots, n$.*

DEMOSTRACIÓN. El resultado es una consecuencia directa de los Lemas 4.4.7, 4.4.8, 4.4.10, y 4.4.12. □

Cabe destacar que todas las demostraciones realizadas en esta sección se han planteado en un contexto de reescritura, de forma similar a las realizadas para otras reglas de transformación relacionadas con computaciones necesarias. Esta forma de simplificar los esquemas de demostración es factible gracias a los abundantes resultados sobre reescritura necesaria disponibles en la literatura, así como a la posibilidad de ser extrapolados (bajo ciertas condiciones) al caso general de estrategias de *narrowing* correctas y completas. De hecho, la corrección y completitud fuerte del sistema de transformación se deriva de la preservación de los invariantes (I1) e (I2) a lo largo de toda secuencia de transformación virtual (demostrado en la Proposición 4.4.13), junto con los resultados de corrección, completitud y optimalidad del *narrowing* necesario (recogidos en el Teorema 2.2.15).

Teorema 4.4.14 (corrección y completitud fuertes) *Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, una secuencia de transformación virtual y sea e una ecuación. Entonces, $e \overset{\text{NN}}{\leadsto}{}^*_\theta$ true es una derivación de narrowing necesario en $\mathcal{R}_0$ si y solo si existe una derivación de narrowing necesario $e \overset{\text{NN}}{\leadsto}{}^*_{\theta'}$ true en $\mathcal{R}_n$ tal que $\theta' = \theta\ [\mathcal{V}ar(e)]$.*

DEMOSTRACIÓN. La demostración es perfectamente análoga a la del Teorema 4.3.5 (corrección y completitud fuertes del plegado no reversible), sustituyendo el uso de los Lemas 4.3.3 y 4.3.4 (corrección y completitud bajo reescritura del plegado no reversible) por la preservación en toda secuencia de transformación virtual del invariante (I1) enunciado en la Proposición 4.4.13. □

Obsérvese que en este resultado de corrección y completitud fuertes se expresa de forma implícita que no sólo el desplegado necesario es fuertemente correcto y completo (aspecto que ya conocíamos del capítulo anterior) sino también el plegado T&S (y la combinación de ambas reglas), lo que era nuestro principal objetivo en esta sección.

4.5 Plegado y evaluación parcial

Si, como ya se ha comentado, la operación de desplegado juega un papel fundamental en el contexto de la especialización de programas declarativos, también en ocasiones se usan versiones (posiblemente restringidas) de plegado en el proceso de la evaluación parcial. Así ocurre, por ejemplo, con las técnicas de supercompilación de Turchin [1986b] para especializar programas funcionales, o con la propia condición de cierre exigida en los contextos lógicos puros. De hecho, existen aproximaciones que describen el propio proceso general de evaluación parcial estrictamente en términos de plegado/desplegado. En la transformación de evaluación pascial, el plegado generalmente se ve como una fase final de renombramiento que se aplica al finalizar el proceso fundamental de generación de los árboles de desplegado y extracción de resultantes. Más aún, los beneficios de efectuar esta clase de renombramiento (similar a un paso de plegado) después de especializar (por desplegado) un conjunto de términos, resultan especialmente relevantes cuando dichos términos se corresponden con conjunciones de átomos (lo que se conoce como *evaluación parcial conjuntiva*). En este contexto, el proceso de plegado puede realizarse de forma automática, al igual que la fase previa de desplegado. En [Albert *et al.*, 1998a] se detalla el papel que juega la regla de plegado en el contexto de la evaluación parcial de programas lógico-funcionales. Como ya se ha comentado, las técnicas de evaluación parcial son un subconjunto estricto de las transformaciones basadas en plegado/desplegado, donde el desplegado es la regla de transformación básica que se usa para implementar el núcleo de un evaluador parcial (ver Sección 3.6 del capítulo anterior), mientras que el plegado puede simular un post-proceso de renombramiento sobre el programa especializado [Alpuente *et al.*, 2000b]. De esta forma, muchas de las técnicas de EP (que originalmente no se han descrito explícitamente en términos de la aproximación "reglas + estrategias" de Burstall y Darlington [1977]) pueden ser reformuladas en términos de transformaciones de plegado/desplegado donde el desplegado supone el núcleo de la transformación, mientras que sólo se obtiene una forma limitada de plegado implícito al imponer la condición de cierre [Pettorossi y Proietti, 1996a]. Este requerimiento es similar a la condición conocida como *need for folding* o búsqueda de regularidades que aparecen en el contexto de los sistemas de transformación basados en plegado/desplegado.

Intuitivamente, la condición de cierre requiere que el proceso de deducción parcial al estilo original de Lloyd y Shepherdson [1991] no se realice en base a un solo objetivo atómico, sino sobre un conjunto de átomos S, de tal forma que cada átomo que aparece en el cuerpo de una cláusula en el programa transformado y que tenga un predicado que aparezca en S, sea una concreción (instancia) de algún átomo de S. Si esta condición se mantiene, cada átomo en el cuerpo de una cláusula transformada se refiere a una de las cabezas de las cláusulas transformadas, obteniéndose así una forma limitada de plegado.

Además, la mayoría de los métodos de deducción parcial hacen uso de una transformación de renombramiento. De nuevo, este postproceso se relaciona con el marco original de Lloyd y Shepherdson [1991], en este caso con la condición de independencia. Los átomos del conjunto S, junto con sus correspondientes ocurrencias en las cláusulas transformadas, son renombrados para evitar la duplicación de código para aquellos pares de átomos con instancias comunes. Esto evita la generación de soluciones redundantes e incluso, en presencia de negación, de soluciones incorrectas. De nuevo, este renombramiento se relaciona estrechamente con la transformación de plegado. Concretamente, si se considera que existe una regla de definición al estilo de las que aparecen en nuestras secuencias de transformación virtuales (Definición 4.4.1), donde cada predicado nuevo se define de forma que tenga el mismo valor de verdad que el predicado viejo asociado, entonces pueden realizarse una serie de pasos de plegado para reemplazar las ocurrencias del predicado viejo por las correspondientes del nuevo.

Siguiendo la idea anterior, en [Fujita, 1987; Bossi *et al.*, 1990; Sahlin, 1991; Bossi y Cocco, 1993; Prestwich, 1993; Proietti y Pettorossi, 1993; Gallagher, 1993] se estudian diversas técnicas para la especialización de programas lógicos basadas en las reglas de plegado/desplegado. Siguiendo estas formulaciones, dado un programa P y un objetivo $\Leftarrow G$, se trata de introducir un nuevo predicado *newp* definido por la cláusula D siguiente:

$$D : newp(X_1, \ldots, X_n) \Leftarrow G$$

donde $X_1, \ldots, X_n$ son las variables de G. Obviamente, $newp(X_1, \ldots, X_n)$ y G son objetivos equivalentes con respecto a la semántica de $P \cup \{D\}$. Más concretamente, es posible relacionar una evaluación parcial de $newp(X_1, \ldots, X_n)$ con respecto a $P \cup \{D\}$ con otra de G con respecto a P donde, para conseguir la equivalencia, es suficiente con realizar un proceso de renombramiento (o plegado) en el segundo caso, gracias al cual todas las instancias de G se sustituyen por las instancias apropiadas de *newp* en el programa especializado (nótese que esta última fase siempre es posible gracias a la condición de cierre).

Una forma alternativa de conseguir este mismo resultado es mediante una secuencia de aplicaciones de las reglas de plegado/desplegado. En este caso, se parte del programa extendido $P \cup \{D\}$, donde D se considera una regla de definición (al estilo de la Definición 4.4.1 sobre secuencias de transformación virtuales) que define al predicado nuevo *newp*. A partir de ahí, se aplican una serie de pasos de desplegado sobre D hasta encontrar regularidades y proceder a un paso de plegado utilizando D como regla plegante. De esta forma, se obtiene una definición recursiva y eficiente de *newp* que puede verse una vez más como una especialización renombrada del objetivo original G.

Hasta ahora hemos visto la estrecha conexión existente entre la regla de plegado y las condiciones de cierre e independencia que se requieren en la evaluación parcial con renombramiento de programas lógicos. Sin embargo, todavía existen otras relaciones más interesantes que considerar.

El principal defecto a la hora de describir la especialización de un programa como un proceso de transformación basado en plegado/desplegado es que, como es bien conocido, estas técnicas tienen en general un carácter menos automatizable que los métodos de evaluación parcial "ad hoc". Sin embargo, con esta aproximación es posible conseguir (a un coste más elevado) una clase mayor de transformaciones, que no son factibles mediante las técnicas estándar de evaluación parcial como, por ejemplo, la eliminación de variables redundantes, la deforestación o la formación de tuplas.

Este problema reside fundamentalmente en el hecho de que las técnicas clásicas de deducción parcial al estilo de Lloyd y Shepherdson [1991] realizan evaluaciones parciales para átomos independientes. Recientemente, en [Glück *et al.*, 1996; Leuschel *et al.*, 1996] se ha introducido una técnica para la deducción parcial de conjunciones de átomos. A diferencia de las aproximaciones clásicas, la evaluación parcial conjuntiva permite considerar conjunciones de átomos en S incluso en el proceso de renombramiento.

En el contexto de la programación lógico-funcional, el método de evaluación parcial dirigido por *narrowing* de Alpuente *et al.* [1997a, 1998a] utiliza las nociones de resultante, cierre e independencia que hemos descrito en el capítulo anterior. Además, en este marco genérico para la especialización de programas lógico-funcionales, el proceso de renombramiento efectuado tras la evaluación parcial propiamente dicha actúa como una forma de plegado similar al caso lógico descrito anteriormente, con algunas peculiaridades que veremos a continuación.

Ejemplo 39 Consideremos la definición bien conocida para la función `append` que concatena dos listas en el siguiente programa $\mathcal{R}$[7]:

$$\mathtt{append(nil, Y_s)} \to \mathtt{Y_s}$$
$$\mathtt{append(X : X_s, Y_s)} \to \mathtt{X : append(X_s, Y_s)}$$

Una especialización del programa anterior con respecto al conjunto:

$$S = \{\mathtt{append(append(X_s, Y_s), Z_s)},\ \mathtt{append(X_s, Y_s)}\}$$

devuelve el siguiente conjunto de resultantes como programa transformado $\mathcal{R}'$:

$$\mathtt{append(append(nil, Y_s), Z_s)} \to \mathtt{append(Y_s, Z_s)}$$
$$\mathtt{append(append(X : X_s, Y_s), Z_s)} \to \mathtt{X : append(append(X_s, Y_s), Z_s)}$$
$$\mathtt{append(nil, Z_s)} \to \mathtt{Z_s}$$
$$\mathtt{append(Y : Y_s, Z_s)} \to \mathtt{Y : append(Y_s, Z_s)}$$

[7]Usamos `nil` y `:` como constructores de las listas.

donde $\mathcal{R}'$ es S-cerrado.

A continuación podemos realizar un renombramiento de las reglas de $\mathcal{R}'$ y de los términos de S de tal forma que, para toda llamada t que sea S-cerrada, las respuestas computadas para t en $\mathcal{R}$ y las respuestas computadas para la llamada renombrada en el programa $\mathcal{R}'$ renombrado coincidan. Además, este proceso puede resultar necesario en aquellas ocasiones en las que, tras la evaluación parcial, sea necesario recuperar la clase original de los programas que especializamos. En particular, si en el ejemplo anterior consideramos programas CB (como ocurre en los marcos de evaluación parcial para programas perezosos de Albert *et al.* [1998a] y Alpuente *et al.* [1999f]) como es el caso del programa original $\mathcal{R}$, entonces el conjunto de resultantes calculado anteriormente no puede considerarse una especialización correcta, ya que $\mathcal{R}'$ no es CB. Una forma de corregir este defecto es renombrar los términos especializados de tal forma que, tras el renombramiento, S contenga únicamente patrones lineales encabezados por diferentes símbolos de función. En particular, interesa que las partes izquierdas de las reglas que componen el programa especializado (que son instancias constructoras de de los términos especializados) sean reemplazadas por los correspondientes patrones lineales nuevos a través del renombramiento. La siguiente definición de Alpuente *et al.* [1999f] captura la idea del tipo de renombramiento que nos interesa.

Definición 4.5.1 (renombramiento independiente) Un renombramiento ρ para un conjunto de términos S es una aplicación de términos en términos definida como sigue: para todo $s \in S$, $\rho(s) = f_s(\overline{x_n})$, donde $\overline{x_n}$ son las variables distintas de s en su orden de aparición y f_s es un nuevo símbolo de función que no aparece en $\mathcal{R}$ o S y es diferente del símbolo raíz de cualquier otro término $\rho(s')$, con $s' \in S$ y $s' \neq s$. Por abuso, $\rho(S)$ denota el conjunto $S' = \{\rho(s) \mid s \in S\}$.

La aplicación de un renombramiento de este tipo a un conjunto de términos S se hace de forma obvia, mientras que para renombrar un conjunto de resultantes el renombramiento procede como sigue. Primero se aplica el renombramiento independiente ρ a las partes izquierdas de los resultantes, lo que es posible gracias al hecho de que estas son instancias constructoras de las llamadas especializadas en S. Por su parte, las partes derechas de los resultantes son renombradas por medio de una función auxiliar ren_ρ, que reemplaza recursivamente cada llamada en la expresión dada por una llamada a la correspondiente función renombrada (de acuerdo con ρ). Esta inspección recursiva es similar al propio test de closedness, cuyos detalles pueden consultarse en [Alpuente *et al.*, 1999f].

Definición 4.5.2 Sea t un término y ρ un renombramiento independiente. Entonces

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

Ejemplo 40 Considerando de nuevo el programa `append` y el conjunto S del Ejemplo 39, tenemos que la siguiente aplicación ρ es un renombramiento independiente:

$$\{ \quad \mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}) \mapsto \mathtt{app}(\mathtt{X_s}, \mathtt{Y_s}),$$
$$\mathtt{append}(\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{Z_s}) \mapsto \mathtt{dapp}(\mathtt{X_s}, \mathtt{Y_s}, \mathtt{Z_s}) \ \}.$$

Si renombramos el programa $\mathcal{R}'$ usando ρ obtenemos el siguiente programa especializado:

$$\begin{array}{rcl} \mathtt{dapp}(\mathtt{nil}, \mathtt{Y_s}, \mathtt{Z_s}) & \rightarrow & \mathtt{app}(\mathtt{Y_s}, \mathtt{Z_s}) \\ \mathtt{dapp}(\mathtt{X} : \mathtt{X_s}, \mathtt{Y_s}, \mathtt{Z_s}) & \rightarrow & \mathtt{X} : \mathtt{dapp}(\mathtt{X_s}, \mathtt{Y_s}, \mathtt{Z_s}) \\ \mathtt{app}(\mathtt{nil}, \mathtt{Y_s}) & \rightarrow & \mathtt{Y_s} \\ \mathtt{app}(\mathtt{X} : \mathtt{X_s}, \mathtt{Y_s}) & \rightarrow & \mathtt{X} : \mathtt{app}(\mathtt{X_s}, \mathtt{Y_s}) \end{array}$$

Una forma alternativa de conseguir este mismo programa vía plegado/desplegado consiste en construir una secuencia de transformaciones similar a la vista en el Ejemplo 34 que estudiamos en el capítulo anterior (haciendo uso de las reglas del desplegado necesario de la Sección 3.4.2 y el plegado T&S definido en la Sección 4.4). En este caso, el programa original coincide con $\mathcal{R}$ añadiendo además las siguientes reglas de definición para los símbolos nuevos `dapp` y `app` (obsérvese la relación entre estas reglas y ρ):

$$\begin{array}{rcll} \mathtt{app}(\mathtt{X_s}, \mathtt{Y_s}) & \rightarrow & \mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}) & (R_1) \\ \mathtt{dapp}(\mathtt{X_s}, \mathtt{Y_s}, \mathtt{Z_s}) & \rightarrow & \mathtt{append}(\mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{Z_s}) & (R_2) \end{array}$$

A continuación procedemos a desplegar la primera regla de definición R_1 y obtenemos:

$$\begin{array}{rcll} \mathtt{app}(\mathtt{nil}, \mathtt{Y_s}) & \rightarrow & \mathtt{Y_s} & (R_3) \\ \mathtt{app}(\mathtt{X} : \mathtt{X_s}, \mathtt{Y_s}) & \rightarrow & \mathtt{X} : \mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}) & (R_4) \end{array}$$

Y plegando la regla R_4 con la regla de definición original R_1 queda:

$$\mathtt{app}(\mathtt{X} : \mathtt{X_s}, \mathtt{Y_s}) \ \rightarrow \ \mathtt{X} : \mathtt{app}(\mathtt{X_s}, \mathtt{Y_s}) \quad (R_5)$$

Por otro lado, con la idea de obtener una versión recursiva de `dapp`, procedemos a desplegar la regla de definición R_2, obteniendo el nuevo par de reglas:

$$\begin{array}{rcll} \mathtt{dapp}(\mathtt{nil}, \mathtt{Y_s}, \mathtt{Z_s}) & \rightarrow & \mathtt{append}(\mathtt{Y_s}, \mathtt{Z_s}) & (R_6) \\ \mathtt{dapp}(\mathtt{X} : \mathtt{X_s}, \mathtt{Y_s}, \mathtt{Z_s}) & \rightarrow & \mathtt{append}(\mathtt{X} : \mathtt{append}(\mathtt{X_s}, \mathtt{Y_s}), \mathtt{Z_s}) & (R_7) \end{array}$$

Al plegar R_6 usando R_1 nos queda:

$$\mathtt{dapp(nil, Y_s, Z_s)} \quad \to \quad \mathtt{app(Y_s, Z_s)} \quad (R_8)$$

Un nuevo desplegado de R_7 genera:

$$\mathtt{dapp(X : X_s, Y_s, Z_s)} \quad \to \quad \mathtt{X : append(append(X_s, Y_s), Z_s)} \quad (R_9)$$

Finalmente, por plegado de R_9 con respecto a la definición original de `dapp` (regla R_2) obtenemos la regla:

$$\mathtt{dapp(X : X_s, Y_s, Z_s)} \quad \to \quad \mathtt{X : dapp(X_s, Y_s, Z_s)} \quad (R_{10})$$

De esta forma, el programa transformado final, además de todas las reglas de $\mathcal{R}$, contiene el siguiente conjunto:

$$\begin{array}{rcll}
\mathtt{app(nil, Y_s)} & \to & \mathtt{Y_s} & (R_3) \\
\mathtt{app(X : X_s, Y_s)} & \to & \mathtt{X : app(X_s, Y_s)} & (R_5) \\
\mathtt{dapp(nil, Y_s, Z_s)} & \to & \mathtt{app(Y_s, Z_s)} & (R_8) \\
\mathtt{dapp(X : X_s, Y_s, Z_s)} & \to & \mathtt{X : dapp(X_s, Y_s, Z_s)} & (R_{10})
\end{array}$$

Nótese que, como esperábamos, este conjunto de reglas coincide con el programa anterior obtenido por evaluación parcial tras el renombramiento posterior.

Es importante destacar la estrecha conexión existente entre los pasos de plegado dados en la secuencia de transformación anterior y los de renombramiento que se aplican normalmente al final de una evaluación parcial. Por ejemplo, el paso de plegado dado sobre el subtérmino $\mathtt{append(append(X_s, Y_s, Z_s)}$ puede verse como un renombramiento de dicho término (donde se anidan dos símbolos "viejos") por el término ("nuevo") $\mathtt{dapp(X_s, Y_s, Z_s)}$.

En este punto, ya hemos comprobado que en el contexto de los lenguajes lógico-funcionales, la relación existente entre la transformación de plegado y la fase de renombramiento tras una EP se mantiene, de forma similar a lo que ocurre en el contexto de la programación lógica. Más aún, en el caso integrado (a diferencia del lógico puro) este renombramiento (o un plegado equivalente) puede resultar imprescindible para poder especializar programas preservando la estructura original de los mismos.

Por otro lado, si partimos de una versión lógica del ejemplo anterior, al aplanar la llamada a especializar obtenemos un conjunto de átomos de la forma $\mathtt{append(X_s, Y_s, Z_s)}$, $\mathtt{append(Z_s, U_s, V_s)}$. En este caso no es posible sintetizar por especialización un predicado capaz de concatenar tres listas utilizando el esquema clásico de evaluación parcial original de Lloyd y Shepherdson [1991]. Esta situación también es cierta en el caso integrado si en vez de intentar especializarlo como una llamada anidada

$\mathtt{append(append(X_s, Y_s), Z_s)}$ lo hacemos partiendo de su versión aplanada correspondiente $\mathtt{append(X_s, Y_s)} = \mathtt{Z_s}$, $\mathtt{append(Z_s, U_s)} = \mathtt{V_s}$. Al utilizar el método de Alpuente *et al.* [1996b, 1997a], la especialización del ejemplo anterior tiene éxito si se lleva a cabo sobre una expresión anidada de la forma $\mathtt{append(append(X_s, Y_s), Z_s)}$, ya que se puede explotar la potencia de la sintaxis funcional. Esto permite transformar el problema original (de formación de tuplas) en otro más simple (de deforestación), que es fácilmente resuelto por el método de Alpuente *et al.* [1996b, 1997a].

Ahora bien, en un marco de deducción parcial conjuntivo, tenemos la posibilidad de considerar la especialización de conjunciones de átomos sobre programas lógicos que sí producen especializaciones efectivas [Glück *et al.*, 1996; Leuschel *et al.*, 1996].

Ejemplo 41 Siguiendo con nuestro ejemplo, una evaluación parcial conjuntiva del conjunto de llamadas $\{\mathtt{append(X_s, Y_s, Z_s)} \wedge \mathtt{append(Z_s, U_s, V_s)}, \mathtt{append(X_s, Y_s, Z_s)}\}$ con respecto al programa lógico clásico que define el predicado `append`, utilizando el renombramiento (conjuntivo) ρ:

$$\begin{array}{ll} \{ & \mathtt{append(X_s, Y_s, Z_s)} \mapsto \mathtt{app(X_s, Y_s, Z_s)}, \\ & \mathtt{append(X_s, Y_s, Z_s)} \wedge \mathtt{append(Z_s, U_s, V_s)} \mapsto \mathtt{dapp(X_s, Y_s, U_s, V_s)} \ \}. \end{array}$$

generaría el programa pretendido (ver [Glück *et al.*, 1996; Leuschel *et al.*, 1996] para más detalles):

$$\begin{array}{rcl} \mathtt{dapp(nil, Y_s, Z_s, U_s)} & \Leftarrow & \mathtt{app(Y_s, Z_s, U_s)} \\ \mathtt{dapp(X : X_s, Y_s, Z_s, X : U_s)} & \Leftarrow & \mathtt{dapp(X_s, Y_s, Z_s, U_s)} \\ \mathtt{app(nil, Y_s, Y_s)} & \Leftarrow & \\ \mathtt{app(X : X_s, Y_s, X : Z_s)} & \Leftarrow & \mathtt{app(X_s, Y_s, Z_s)} \end{array}$$

El marco de Alpuente *et al.* [1996b] tiene el mismo potencial de especialización que la evaluación parcial conjuntiva (y la supercompilación positiva de la programación funcional) como se demuestra en [Alpuente *et al.*, 1998a]). Esto es así porque el método genérico de Alpuente *et al.* [1996b] permite trabajar con ecuaciones y conjunciones durante la especialización con sólo considerar los operadores de igualdad y conjunción como símbolos primitivos del lenguaje. Desafortunadamente, el uso de tales símbolos primitivos puede afectar al proceso de especialización si no se hace un tratamiento especial de los mismos que permita alcanzar la condición de cierre de forma razonable. En [Albert *et al.*, 1998a] se formula una extensión del método de evaluación parcial dirigida por *narrowing* que maneja de forma efectiva los símbolos primitivos de igualdad (estricta) y conjunción durante la especialización de programas lógico-funcionales perezosos.

Ejemplo 42 Considerando ahora la evaluación parcial conjuntiva del programa presentado en el Ejemplo 39 a partir del término $\mathtt{append(X_s, Y_s)} \approx \mathtt{Z_s} \wedge \mathtt{append(Z_s, U_s)} \approx$

$\mathtt{V_s}$, se obtiene el siguiente conjunto de resultantes:

$$
\begin{array}{ll}
\mathtt{append(nil,Y_s)} \approx \mathtt{U_s} \wedge \mathtt{append(U_s,Z_s)} \approx \mathtt{V_s} \rightarrow & \mathtt{append(Y_s,Z_s)} \approx \mathtt{V_s} \\
\mathtt{append(X:X_s,Y_s)} \approx \mathtt{U_s} \wedge \mathtt{append(U_s,Z_s)} \approx \mathtt{V_s} \rightarrow & \mathtt{append(X_s,Y_s)} \approx \mathtt{W_s} \wedge \\
 & \mathtt{append(U_s,Z_s)} \approx \mathtt{W_s} \wedge \\
 & \mathtt{X:W_s} \approx \mathtt{U_s} \\
\mathtt{append(nil,Z_s)} \rightarrow \mathtt{Z_s} & \\
\mathtt{append(Y:Y_s,Z_s)} \rightarrow \mathtt{Y:append(Y_s,Z_s)} &
\end{array}
$$

que, tras un renombramiento adecuado para reemplazar un término de la forma $\mathtt{append(X_s,Y_s)} \approx \mathtt{Z_s} \wedge \mathtt{append(Z_s,U_s)} \approx \mathtt{V_s}$ por otro de la forma $\mathtt{dapp(X_s,Y_s,Z_s,U_s,V_s)}$, derivaría una regla R con el siguiente aspecto:

$$
\begin{array}{ll}
\mathtt{dapp(X:X_s,Y_s,Z:Z_s,U_s,V:V_s)} \rightarrow & \mathtt{X} \approx \mathtt{Z} \wedge \\
 & \mathtt{Z} \approx \mathtt{V} \wedge \\
 & \mathtt{dapp(X_s,Y_s,Z_s,U_s,V_s)}
\end{array}
$$

obteniéndose un programa similar en eficiencia al del Ejemplo 40, aún cuando ahora aparecen un par de parámetros redundantes en la definición de `dapp`.

Sin embargo, lo que más nos interesa observar en este ejemplo es el hecho de que el renombramiento actúa sobre una conjunción de términos, lo cual tiene sus repercusiones en el paso de plegado equivalente. Si la especialización se hace siguiendo una secuencia de transformación, a partir de la regla de definición siguiente:

$$
\begin{array}{ll}
\mathtt{dapp(X_s,Y_s,Z_s,U_s,V_s)} \rightarrow & \mathtt{append(X_s,Y_s)} \approx \mathtt{Z_s} \wedge \\
 & \mathtt{append(Z_s,U_s)} \approx \mathtt{V_s}
\end{array}
$$

obtenemos por sucesivos desplegados:

$$
\begin{array}{ll}
\mathtt{dapp(X:X_s,Y_s,Z:Z_s,U_s,V:V_s)} \rightarrow & \mathtt{X} \approx \mathtt{Z} \wedge \\
 & \mathtt{Z} \approx \mathtt{V} \wedge \\
 & \mathtt{append(X_s,Y_s)} \approx \mathtt{Z_s} \wedge \\
 & \mathtt{append(Z_s,U_s)} \approx \mathtt{V_s}
\end{array}
$$

que, tras un paso de plegado, se transforma exactamente en R.

Lo importante de este ejemplo es que el renombramiento considerado anteriormente se puede simular con el paso de plegado T&S anterior de una forma más natural que en el Ejemplo 40 ya que, en general, un paso de plegado no siempre se restringe a un solo término, sino que lo habitual es que se refiera a una conjunción de ellos, como es el caso que acabamos de ver.

4.6 Conclusiones

La regla de plegado junto con la de desplegado constituye el núcleo de todo sistema de transformación basado en la aproximación "reglas + estrategias". En la Sección 4.2

hemos presentado una versión del plegado reversible inspirada en el plegado *in-situ* descrito en [Pettorossi y Proietti, 1998], que es disyuntivo, utiliza reglas de un único programa y es reversible con respecto al desplegado impaciente descrito en la Sección 3.3 del capítulo anterior. La condición de reversibilidad demostrada muestra que los pasos de plegado pueden deshacerse por pasos de desplegado impaciente apropiados, lo que permite demostrar de forma inmediata la corrección y completitud fuerte de la transformación.

En la Sección 4.3 hemos introducido una variante conjuntiva del plegado al estilo de Gardner y Shepherdson [1991], que sigue considerando reglas de un único programa. Aunque la potencia de optimización de la transformación es todavía muy limitada, el nuevo tipo de plegado puede verse como un primer paso en la consecución de la versión más potente y flexible del plegado presentada en la Sección 4.4.

La extensión se ha hecho al caso de programas inductivamente secuenciales con una semántica operacional basada en *narrowing* necesario. A diferencia de la definición original de Gardner y Shepherdson [1991], en este caso no es necesario exigir que la regla plegable sea instancia de una sola regla plegante. El autoplegado se evita considerando que ambas reglas no se solapen, al tiempo que las sustituciones utilizadas son simples emparejadores de términos. Las principales propiedades de este tipo de plegado son su corrección y completitud fuertes, al tiempo que se preserva el carácter inductivamente secuencial de los programas transformados.

Finalmente, en la Sección 4.4, hemos formulado una extensión del plegado clásico de Tamaki y Sato [1984] para un contexto declarativo integrado, obteniéndose una versión conjuntiva que permite extraer las reglas plegantes y plegables de programas distintos. Esta última característica es la que dota de toda la potencia de optimización al nuevo tipo de plegado.

Para definir el plegado T&S, se introduce el concepto de secuencia de transformación virtual en la cual se permite plegar redexes que no tienen porqué ser necesarios, lo que simplifica la propia definición de plegado. Tampoco es necesario generalizar el término a plegar, ya que es suficiente que éste sea una mera instancia del termino plegante y, además, se simplifica el control de las variables de las reglas involucradas en el plegado.

Gracias al hecho de que esta transformación preserva también la secuencialidad inductiva de los programas, las demostraciones se realizan básicamente en un contexto de reescritura, donde se pueden reutilizar los abundantes resultados sobre reescritura necesaria disponibles en la literatura (y que se extrapolan de forma simple al caso de *narrowing* necesario). La corrección y completitud fuertes del sistema de transformación se deriva del mantenimiento de un par de invariantes (I1) e (I2) a lo largo de toda secuencia de transformación virtual, junto con los resultados estándar de corrección, completitud y optimalidad del *narrowing* necesario.

Entre las aplicaciones de la transformación de plegado, está la posibilidad de describir el proceso de evaluación parcial en términos de pasos de plegado/desplegado. En estas aproximaciones, el plegado puede verse como una fase final de renombramiento, aplicada tras el proceso de generación de árboles de desplegado y extracción de resultantes. En el contexto de evaluación parcial conjuntiva, el postproceso de renombramiento puede verse como una forma especial de plegado, que resulta imprescindible para preservar la clase de programas que se considera en la transformación.

Aparte de la su aplicabilidad en las técnicas de EP, la transformación de plegado (junto con la de desplegado) muestra su mayor grado de aplicación en todo sistema de transformación de programas declarativos basado en la aproximación “reglas+estrategias”. Básicamente, su poder consiste en su capacidad para sintetizar definiciones recursivas y eficientes de funciones cuyas reglas originales hayan sido previamente desplegadas (y posiblemente también tratadas por otras reglas de transformación). En particular, siguiendo a Pettorossi y Proietti [1996a,b], para derivar programas “mejores” pero semánticamente equivalentes al programa original por plegado/desplegado, es usual dirigir el proceso de transformación por una estrategia consistente en la realización de una serie de pasos de desplegado que permiten encontrar regularidades y posibilitar así pasos de plegado posteriores. Sin esta última posibilidad, en general no se puede derivar programas eficientes, incluso aunque se disponga de otras reglas de transformación. En [Alpuente *et al.*, 1999e,d] describimos un sistema de transformación para programas lógico-funcionales que usa el plegado T&S descrito en la Sección 4.4 y que detallaremos a continuación.

En un sistema de transformación basado en la aproximación *reglas + estrategias*, una posibilidad para conseguir el objetivo de obtener un programa más eficiente que el de partida, consiste en intentar asegurar la “ganancia en eficiencia” mientras se avanza en la secuencia de programas, como se hace por ejemplo con el concepto de *improvement* del sistema propuesto en [Sands, 1996] para la transformación de programas funcionales.

Esta propiedad está asegurada por la regla de desplegado que vimos en el capítulo anterior, ya que, en términos sencillos, su aplicación supone avanzar en los pasos de computación (de las expresiones desplegadas). Sin embargo, al ser el plegado una operación que, en cierta manera, actúa en sentido contrario al desplegado (“atrasando” pasos en la computación), puede parecer a *priori* que su papel sea justo el contrario al esperado: que en vez de mejorar la eficiencia del programa transformado, produzca una versión más ineficiente del mismo. Sin embargo, esto no es cierto cuando se considera el plegado no reversible en la línea del propuesto por Tamaki y Sato [1984]. Al menos, no sucede así cuando se imponen las condiciones de aplicabilidad apropiadas. Estas condiciones pueden implicar un chequeo (sintáctico) de distintos elementos asociados a varios programas, teniendo en cuenta la historia de la secuencia de transformación seguida. El objetivo fundamental que se persigue en principio al imponerlas, es el de permitir demostrar la corrección de la transformación aunque, colateralmente, con ellas también se consigue asegurar un cierto grado de optimización. Esto se debe a que se trata de restricciones sintácticas que, de alguna forma, sirven para identificar y preservar algún tipo de medida de "coste computacional" (rango de un objetivo [Tamaki y Sato, 1984], peso de un árbol de prueba [Kawamura y Kanamori, 1990], *improvement* [Sands, 1996]) que se vaya reduciendo (o que al menos no se incremente) al aplicar pasos tanto de desplegado como de plegado. Si observamos nuestra definición de plegado T&S de la Sección 4.4, encontramos que tales restricciones impiden un tipo de plegado reiterado sobre una misma regla, a pesar de que ésta haya sido desplegada sucesivamente. Además se evita el autoplegado que introduciría bucles infinitos en evaluaciones posteriores. De esta manera se consigue preservar el rango de los términos, tal y como indican los invariantes de todos los programas que aparecen en una misma secuencia de transformación. Así, mientras el invariante (I1) se relaciona con la corrección de la regla de plegado, el invariante (I2) garantiza que nunca haya pérdidas de eficiencia[8].

El Ejemplo 40 que acabamos de ver en la sección anterior resulta clarificador en este sentido. Obsérvese que, por una parte, el número de pasos de desplegado aplicados es superior al de plegados, obteniéndose una versión eficiente de la función `dapp`. El mismo ejemplo también muestra que no se puede sintetizar la versión deseada de `dapp` si únicamente se dispone del tipo de plegado no reversible visto en la Sección 4.3. La razón estriba en que la regla plegante que hemos utilizado en el paso de plegado T&S no se encuentra en el último programa de la secuencia (que es el que contiene la regla plegable), siendo esto un requisito necesario para poder aplicar el plegado no reversible. Además, esta situación se repite en la mayoría de aplicaciones prácticas: una versión eficiente de una función se obtiene plegando una de su reglas desplegadas con respecto a la original, que precisamente por haber sido desplegada nunca se encuentra en el último programa de la secuencia de transformación.

Por todas estas razones, la regla de plegado T&S que hemos introducido en la Sección 4.4 se revela como la única formulación apropiada de plegado para ser usada en un sistema de transformación para programas inductivamente secuenciales. Su uso combinado con la regla de desplegado necesario (y las que veremos en el capítulo siguiente) permite obtener optimizaciones efectivas en los programas transformados.

La mayoría de las estrategias de transformación que se usan para conducir la secuencia de aplicación de las reglas de transformación, suelen seguir un patrón genérico en el que, partiendo de una definición de función que se pretende optimizar, primero se realiza sobre ella todas las transformaciones de desplegado (acompañadas de otras

[8]En el caso lógico-funcional, el invariante (I2) asegura que las derivaciones no crecen en longitud, aunque puede darse el caso de que, al aparecer más reglas en los programas transformados, también crezca el número de derivaciones posibles para un mismo objetivo.

auxiliares que sean necesarias) hasta conseguir encontrar alguna regularidad que permita realizar un paso final de plegado que devuelva una versión recursiva y eficiente de la regla original. Es decir, un solo paso final de plegado, por varios pasos intermedios de desplegado y de aplicación de otras reglas auxiliares.

Sin embargo, es frecuente que en una secuencia de transformación existan pasos de plegado extra a los que acabamos de comentar. Esto ocurre cuando se pretende usar la definición eficiente y transformada de una función f en la definición de otra función que originalmente utiliza la versión inicial de f. Esta situación se repetirá muy frecuentemente cuando apliquemos la regla de introducción de definiciones que veremos en el capítulo siguiente. Para dar una idea intuitiva de este segundo uso del plegado, veamos el siguiente ejemplo.

Ejemplo 43 Consideremos la siguiente extensión del programa original del Ejemplo 39

$$\begin{array}{rcl}
\mathtt{append(nil, Y_s)} & \to & \mathtt{Y_s} \\
\mathtt{append(X : X_s, Y_s)} & \to & \mathtt{X : append(X_s, Y_s)} \\
\mathtt{doubleappend(X_s, Y_s, Z_s)} & \to & \mathtt{append(append(X_s, Y_s), Z_s)} \\
\mathtt{test(X_s, Y_s, Z_s)} & \to & \mathtt{C}[\ldots \mathtt{append(append(X_s, Y_s), Z_s)} \ldots]
\end{array}$$

donde $\mathtt{C}[\ldots \mathtt{append(append(X_s, Y_s), Z_s)} \ldots]$ se refiere a dos llamadas anidadas de `append` dentro de un contexto cualquiera. Siguiendo una secuencia de transformaciones similar a la vista en el Ejemplo 40, podemos derivar un programa que define de forma eficiente la función `doubleappend`. Sin embargo, en el programa final seguiría existiendo la función `test` que, en su parte derecha, no utiliza las ventajas que representa la nueva forma de concatenar tres listas asociada a la nueva definición de `doubleappend`. Esta deficiencia puede eliminarse con un paso extra de plegado sobre la regla original que define a *test* con respecto a la regla original que define a `doubleappend`, obteniéndose así el programa final optimizado:

$$\begin{array}{rcl}
\mathtt{append(nil, Y_s)} & \to & \mathtt{Y_s} \\
\mathtt{append(X : X_s, Y_s)} & \to & \mathtt{X : append(X_s, Y_s)} \\
\mathtt{doubleappend(nil, Y_s, Z_s)} & \to & \mathtt{append(Y_s, Z_s)} \\
\mathtt{doubleappend(X : X_s, Y_s, Z_s)} & \to & \mathtt{X : doubleappend(X_s, Y_s, Z_s)} \\
\mathtt{test(X_s, Y_s, Z_s)} & \to & \mathtt{C}[\ldots \mathtt{doubleappend(X_s, Y_s, Z_s)} \ldots]
\end{array}$$

En el capítulo siguiente completaremos el conjunto de reglas de transformación consideradas en este capítulo, para conformar un sistema robusto y potente de transformación de programas lógico-funcionales perezosos.

Capítulo 5

Un sistema completo de transformación

En este capítulo presentamos el conjunto completo de reglas que constituyen un sistema de transformación para programas inductivamente secuenciales. Aparte de las reglas básicas de desplegado necesario y plegado T&S estudiadas anteriormente (Secciones 3.4.2 y 4.4), en la Sección 5.2 completamos el sistema con otras reglas auxiliares para la introducción y eliminación de definiciones, operaciones de abstracción y, finalmente, algunas leyes para las funciones primitivas (reglas de reemplazamiento algebraico). Nuestro objetivo es definir un conjunto amplio de transformaciones para programas lógico-funcionales que goce de las propiedades de corrección y completitud fuertes con respecto a la semántica de valores y respuestas computadas. Este conjunto de reglas extiende el sistema original definido por Burstall y Darlington [1977] para la transformación de programas funcionales, incorporando muchas de las optimizaciones que se han propuesto posteriormente en el área y generalizándolas para el caso de los lenguajes integrados con una semántica operacional basada en *narrowing* necesario.

En la Sección 5.3 demostramos que este conjunto de reglas es suficiente para implementar en nuestro contexto la mayoría de las estrategias de transformación más potentes que se conocen, como son las de composición y formación de tuplas (aparte de otras estrategias propuestas en la literatura, que también serán objeto de discusión). Los resultados experimentales se muestran en la Sección 5.4, donde introducimos el sistema de transformación SYNTH, con el cual realizamos una serie de pruebas usando ejemplos clásicos y otros más novedosos que ponen de manifiesto las ventajas del sistema de transformación a dos niveles: a) ganancia de eficiencia en los programas transformados y b) buen nivel de automatización del proceso de transformación.

Comenzamos este capítulo haciendo un breve recuento de las diversas propuestas para definir reglas y estrategias de transformación que se han hecho en los contextos lógico y funcional puros, planteando a la vez los problemas y ventajas de la extensión de estas ideas al contexto integrado.

5.1 Motivación y antecedentes

Según la aproximación de "reglas + estrategias", para optimizar un programa es necesario definir un conjunto de reglas de transformación elementales (incluyendo las operaciones básicas de plegado y desplegado), que se aplican sobre un programa siguiendo una determinada estrategia. Por estrategia entendemos un conjunto de metareglas que dirigen de forma automática, guiada por una heurística o asistida por el usuario, el proceso de aplicación de las reglas elementales para construir la secuencia de transformaciones [Pettorossi y Proietti, 1994, 1996b]. Esta técnica puede conseguir resultados tales como: la eliminación de variables innecesarias [Proietti y Pettorossi, 1993], evitar la construcción de estructuras de datos intermedias o el recorrido múltiple de una misma estructura de datos [Burstall y Darlington, 1977], convertir especificaciones (ineficientes) en programas (eficientes) [Komorowski, 1991], realizar la compilación de un programa fuente (es el caso del compilador del lenguaje funcional *Haskell* [Peyton-Jones, 1996]), así como efectuar la especialización de programas respecto a parte de sus datos de entrada.

Respecto a este último punto, la principal ventaja de un marco general de transformación de programas estriba en que su potencia es mucho más alta que la de cualquier especializador parcial (de hecho, ya hemos visto que las técnicas de especialización pueden verse como como un caso particular de las técnicas de transformación por plegado/desplegado [Pettorossi y Proietti, 1996b]) y, aunque su grado de automatización puede presentar ciertas limitaciones, el uso de heurísticas apropiadas y la posibilidad de interacción con el usuario consigue paliar de forma efectiva estas deficiencias.

Aparte de las reglas básicas de plegado y desplegado, existe en la literatura una extensa gama de definiciones de reglas de transformación elementales de menor rango que las anteriores. Estas reglas accesorias son necesarias en un marco genérico de transformación de programas si realmente se pretende derivar programas eficientes. Aunque existe una gran cantidad de variantes con diferentes nomenclaturas (*replacement, instantiation, abstraction, pruning, thinning, fattening*, etc.)y significados [Pettorossi y Proietti, 1994; Bossi *et al.*, 1990], la idea básica subyacente a todas ellas consiste en añadir, modificar o eliminar porciones del programa o del objetivo original, como vamos a describir en este capítulo.

En la literatura especializada en el tema, tanto en el campo lógico ([Bossi y Cocco, 1993; Bossi *et al.*, 1990; Gardner y Shepherdson, 1991; Pettorossi y Proietti, 1994, 1996a; Tamaki y Sato, 1984]) como en el funcional ([Burstall y Darlington, 1977; Kott, 1985; Nielson y Nielson, 1990; Sands, 1996; Scherlis, 1981; Zhu, 1994]), se da buena cuenta de las ventajas que aporta el uso sistemático de conjuntos robustos de reglas y la aplicación de buenas estrategias de transformación. Un hecho llamativo de los contextos funcionales puros es que todos los sistemas de transformación precisan de una regla especial de *instanciación* que se revela necesaria para posibilitar algunos pasos de desplegado (por reescritura) posteriores, tal y como comentamos ya en el Capítulo 3.

En [Burstall y Darlington, 1977] se desarrolla el primer sistema de transformación para programas escritos en un lenguaje funcional basado en ecuaciones recursivas con patrones y en el cual se asume un sistema de evaluación perezoso. Además de las operaciones de plegado y desplegado, se añaden cuatro nuevas reglas: *definición* (introduce una nueva ecuación recursiva cuya parte izquierda no es una instancia de la parte izquierda de una definición previa), *instanciación* (introduce una instancia de una ecuación existente), *abstracción* (de alguna forma, la inversa de la anterior, vincula una expresión a una variable, y reemplaza en una ecuación ocurrencias de esa expresión por la variable) y *leyes para las primitivas* (para permitir la reescritura usando funciones primitivas). Todas las reglas de transformación propuestas preservan ciertos resultados de corrección parcial (a pesar de que los pasos locales de plegado/desplegado preservan el significado del programa transformado, no siempre se puede garantizar que una secuencia de transformaciones sea capaz de producir cualquier programa equivalente) y el sistema global goza del potencial necesario para la síntesis y optimización de programas.

La aproximación basada en las transformaciones de plegado/desplegado fue adaptada inicialmente a la programación lógica por Tamaki y Sato [1984], reemplazando emparejamiento por unificación en las reglas de transformación. Además de esta adaptación de las reglas básicas de plegado/desplegado, el sistema se amplía con una regla que permite introducir nuevas definiciones y otra más general que permite realizar reemplazamiento de objetivos.

En [Bossi *et al.*, 1990] se elabora un marco de transformación de programas lógicos que incorpora operaciones definidas en trabajos previos y que, básicamente, son adaptaciones para el caso lógico de las leyes de primitivas de los programas funcionales (definición, inserción, borrado y mezcla de objetivos, mezcla de funciones, inserción y borrado de cláusulas y leyes de primitivas [Tamaki y Sato, 1984; Sato y Tamaki, 1984]). Además, se incluyen de forma novedosa otras operaciones nuevas como *thinning* (que elimina un átomo en el cuerpo de una cláusula), *fattening* (inversamente, añade un átomo al cuerpo de una cláusula), *pruning* (elimina una cláusula redundante en un programa) y *constraining* (que restringe un programa al subprograma dado por el cierre transitivo de un átomo). Todas las reglas son conservativas y se usan con el objeto de especializar programas dentro de un dominio restringido y prefijado. Se trata de un método de especialización de programas lógicos que permite restringir un programa general a una serie de casos particulares por medio de predicados de restricción (*constraint predicates*). El método usa las operaciones anteriores para restringir el dominio de un programa y propagar la información de las restricciones a través del mismo con el objetivo de simplificarlo siempre que sea posible. La eficiencia se obtiene porque algunas partes de la computación general se convierten en redundantes en la versión especializada y pueden omitirse, o bien pueden precomputarse otras partes. El sistema de transformación es interactivo y guiado por el usuario y usa ciertas heurísticas para dirigir el proceso.

En [Bossi y Cocco, 1993; Bossi *et al.*, 1992] se considera la corrección con respecto a las así llamadas *s-semánticas* (o semánticas de respuestas computadas) y se enuncian condiciones de aplicabilidad para las operaciones de plegado, reemplazamiento, *thinning* y *fattening* que aseguran la preservación de tales semánticas a lo largo de la transformación. Los mismos autores también han estudiado en [Bossi y Etalle, 1994; Bossi *et al.*, 1995] la correción del reemplazamiento de objetivos y sus operaciones derivadas (básicamente todas ellas consisten en la sustitución de una conjunción de átomos por otra en el cuerpo de una cláusula), con respecto a las principales semánticas sobre programas con negación, tales como la semántica de Fitting y la semántica de Kunen.

Finalmente, Proietti y Pettorossi han señalado en numerosos textos ([Proietti y Pettorossi, 1990, 1993; Pettorossi y Proietti, 1994, 1996a,b, 1998]) la necesidad de introducir heurísticas o estrategias que dirijan el proceso de transformación, al tiempo que adaptan ciertas estrategias desarrolladas en programación funcional para identificar una definición (predicado eureka) mediante la inspección de los árboles de desplegado. En [Pettorossi y Proietti, 1994], por ejemplo, se introducen algunas reglas de transformación (introducción y eliminación de definiciones y reemplazamiento de objetivos) que, junto con las operaciones de plegado y desplegado, permiten construir un sistema automático de transformación de programas lógicos definidos y normales, al tiempo que se discuten las condiciones de aplicabilidad de tales operaciones, así como las semánticas que se preservan en cada caso. Los autores revisan también los fundamentos teóricos de la aproximación "reglas +estrategias" a la transformación de programas lógicos. En [Pettorossi y Proietti, 1994], establecen un marco unificado para formular y comparar las diferentes reglas propuestas en la literatura. El marco es paramétrico con respecto a las semánticas que se preservan durante la transformación. Se presentan varios conjuntos de reglas de transformación y sus correspondientes resultados de corrección con respecto a las siguientes semánticas: modelo mínimo de Herbrand, semántica de respuestas computadas, fallo finito y semántica de Prolog. También se considera el caso de los programas normales y, usando el mismo marco, se presentan las reglas que preservan las semánticas de fallo finito, conjunto de éxitos y complección de Clark. Con este marco unificado es posible describir algunos de los métodos de transformación de programas lógicos más significativos propuestos en la literatura. Aquí se tratan algunas estrategias como la formación de tuplas, absorción de bucles y generalización, y se muestra que algunas técnicas básicas relacionadas con el control de la compilación, la composición de programas, el cambio de la representación de los datos, la evaluación parcial y la especialización de programas pueden verse como aplicaciones concretas de estas estrategias. Básicamente, algunas de las estrategias citadas se formulan en forma de algoritmos sucesivamente refinados que, de forma automática, guían el proceso de generación de los predicados eureka. Estos predicados (que una vez transformados se definen de forma recursiva) no aparecen en los programas originales y son los que realmente consiguen a la postre evitar la evaluación repetida de subtérminos comunes, visitas múltiples a las mismas estructuras de datos, construcción de enlaces intermedios, etc.

La transformación de programas lógico-funcionales es un área de investigación relativamente nueva que no ha recibido todavía una atención particular en la literatura especializada. En nuestros primeros estudios de [Alpuente *et al.*, 1997c,d] presentamos una primera aproximación al problema, considerando inicialmente una extensión directa al caso lógico-funcional de las transformaciones de plegado/desplegado, haciendo uso de la relación de *narrowing condicional* para definir el paso de computación elemental. Por otra parte, el primer sistema de transformación que extiende al contexto integrado todas las reglas del marco original de Burstall y Darlington [1977] se introduce en [Alpuente *et al.*, 1999e,d, 2000a]. En este capítulo describiremos en detalle esta propuesta, que permite transformar programas inductivamente secuenciales aplicando de forma automática la estrategia de composición y (una forma semi-automática de) la de formación de tuplas.

La estrategia de composición (y algunas de sus variantes, como la especialización interna y la deforestación) se introdujo originalmente para la optimización de programas funcionales puros. Esta técnica permite evitar la construcción de estructuras de datos intermedias producidas por una cierta función `g` y consumidas por otra función `f` (e.g., cuando aparece una expresión `f(g(_))` en el programa). En algunos casos, muchos de las mejoras en eficiencia que pueden obtenerse con la estrategia de composición, también puede conseguirse simplemente usando evaluación perezosa sobre los programas originales. Sin embargo, la estrategia de composición a menudo permite derivar programas con rendimiento mejorado incluso en contextos de evaluación perezosa.

Por su parte, la estrategia de formación de tuplas también fue introducida originalmente para la optimización de programas funcionales. Esta estrategia resulta muy efectiva cuando aparecen en el programa varias llamadas a función que requieren la computación de una misma expresión, en cuyo caso es posible obtener una única función nueva que devuelve (en una *tupla*) el resultado de las distintas llamadas, evitando así el acceso múltiple a una misma estructura de datos y computaciones redundantes. De esta forma, es posible producir programas lineales (i.e., programas cuyas partes derechas sólo contienen como mucho una llamada recursiva) a partir de programas no lineales. Esta clase de optimización no siempre puede conseguirse por métodos de transformación puramente automáticos como la evaluación parcial o la estrategia de composición.

5.2 El conjunto de reglas de transformación

En el capítulo anterior, al definir el plegado T&S, hemos asumido la existencia de una secuencia de programas transformados por plegado/desplegado que llamábamos secuencia de transformación virtual (ver la Definición 4.4.1), que sigue los esquemas Tamaki y Sato [1984] y Kawamura y Kanamori [1990]). En estas secuencias caracterizadas porque la signatura de los programas no varía en todo el proceso, es posible extraer reglas plegantes y plegables de programas distintos obtenidos únicamente por aplicación de las reglas de plegado/desplegado a partir de un programa inicial dividido en dos conjuntos disjuntos de reglas, $\mathcal{R}_{new}$ y $\mathcal{R}_{old}$. Estas restricciones (signatura invariable, programa inicial particionado y sólo las reglas de plegado y desplegado) resultan convenientes de cara a simplificar las demostraciones de la corrección y completitud de las transformaciones consideradas, ya que se pueden abstraer y soslayar detalles accesorios como, por ejemplo, el cambio de la signatura de los programas a lo largo de la secuencia de transformación.

Sin embargo, esta formulación resulta ciertamente artificiosa en la práctica, y en su lugar se suele considerar que el conjunto de reglas de definición que componen $\mathcal{R}_{new}$ no está incluido originalmente en $\mathcal{R}_0$, sino que estas reglas se introducen dinámicamente durante el proceso de construcción de la secuencia de transformación. Esta aproximación más relajada lleva a la necesidad de considerar una nueva regla de transformación que permita introducir reglas de definición para símbolos nuevos. Y, por simetría, también es conveniente incluir en el conjunto de reglas otra operación que sea capaz de eliminar definiciones bajo una serie de restricciones apropiadas.

Pero si bien el sistema de transformación gana en naturalidad expresiva con este nuevo esquema, la contrapartida es que aparecen problemas laterales relacionados con el cambio de la signatura de los programas pertenecientes a la secuencia de transformación. Así, mientras la regla de introducción de definiciones incrementa el conjunto de símbolos de función definidos, la regla de eliminación lo reduce. Este hecho afecta directamente a la corrección y completitud del sistema de transformación, ya que en función de cual sea la signatura asociada al programa, el conjunto de objetivos que se puede evaluar es distinto (y, por tanto, no pueden compararse las semánticas de los distintos programas).

Para resolver este problema, se suele asumir que la signatura original de $\mathcal{R}_0$ está incluida en la de todos los programas transformados, lo cual implica que durante la transformación, de forma interactiva, sólo se pueden introducir o eliminar reglas que definan funciones nuevas. De esta forma se consigue el efecto de preservar la semántica de valores y respuestas computadas para aquellos objetivos que consideren la misma signatura que el programa inicial, lo cual parece razonable si pensamos que el principal objetivo es optimizar (no especializar ni extender) programas sin que cambie su semántica.

De esta manera generalizamos el concepto de secuencia de transformación respecto a las secuencias virtuales de la Definición 4.4.1 en 3 aspectos:

1. Ya no es necesario considerar que $\mathcal{R}_0$ está dividido en dos conjuntos disjuntos, sino que directamente se considera un único conjunto de reglas que definen todos los símbolos *viejos* del programa (i.e., siguiendo la notación de la Definición 4.4.1 tenemos que $\mathcal{R}_0 = \mathcal{R}_{old}$).

2. Por su parte, la componente asociada a todas las reglas de definición para los símbolos de función *nuevos* (que llamábamos $\mathcal{R}_{new}$), se va incorporando gradualmente a los programas transformados a medida que avanza la secuencia de transformación. Además, existe la posibilidad de que en algún momento se apliquen reglas de eliminación que reduzcan esta componente.

3. Todo programa de la secuencia distinto del inicial se obtiene por la aplicación de una regla de transformación cualquiera, y no exclusivamente aplicando el desplegado necesario o el plegado T&S sobre el inmediatamente precedente.

Obviamente, para que este nuevo concepto de secuencia de transformación se adecúe al virtual, con las matizaciones hechas anteriormente, es necesario garantizar que la regla de introducción de una definición respeta los criterios exigidos sobre las reglas de definición de símbolos de función nuevos (i.e., que tengan las mismas variables en sus partes izquierda y derecha, que no sean recursivas y que definan símbolos nuevos en función de símbolos viejos). Ahora queda claro el nombre que dábamos a las reglas de $\mathcal{R}_{new}$ (reglas de definición) en la Definición 4.4.1, ya que ésta es la forma habitual de nombrar a las reglas que introducen una definición.

Finalmente, para incrementar el potencial de optimización del sistema, es necesario dotarlo de reglas de abstracción y reemplazamiento algebraico que permitan la implementación de estrategias de transformación tan potentes como la formación de tuplas. La principal aportación de las abstracciones es que permiten reducir el número de subexpresiones a evaluar dentro de un objetivo. Por su parte, el reemplazamiento algebraico se describe clásicamente de una forma muy relajada, como una serie de propiedades naturales que cumplen algunos símbolos de función primitivos.

A continuación pasamos a describir cada una de las reglas auxiliares que, junto con el plegado necesario y el plegado T&S, completan el sistema de transformación propuesto.

5.2.1 Introducción y eliminación de definiciones

La regla de introducción de una definición se inspira en la original de Tamaki y Sato [1984], aunque extendida para tratar con programas inductivamente secuenciales. La aplicación de esta regla supone una partición del conjunto de definiciones en dos conjuntos disjuntos que definen los símbolos viejos y nuevos, similares a los conjuntos $\mathcal{R}_{old}$ y $\mathcal{R}_{new}$ que vimos en la Definición 4.4.1.

Definición 5.2.1 (introducción de una definición) Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq 0$, una secuencia de transformación. Podemos obtener el programa $\mathcal{R}_{k+1}$ añadiendo al programa $\mathcal{R}_k$ una regla (llamada *regla de definición*) de la forma $f(\overline{t_n}) \to r$, tal que:

1. $f(\overline{t_n})$ es un patrón lineal con $\mathcal{V}ar(f(\overline{t_n})) = \mathcal{V}ar(r)$,
2. f no aparece en la secuencia $\mathcal{R}_0, \ldots, \mathcal{R}_k$ (f es *nuevo*), y
3. cada símbolo de función que aparece en r pertenece a $\mathcal{R}_0$.

Decimos que f es un símbolo de función *nuevo*, mientras que aquellos símbolos de función definidos en $\mathcal{R}_0$ se llaman símbolos de función *viejos*.

Se debe notar en esta definición que el concepto de *regla de definición* aquí introducido es idéntico a su homólogo de la definición 4.4.1, de tal forma que toda regla de definición perteneciente a $\mathcal{R}_{new}$ en una secuencia de transformación virtual es equivalente a otra introducida por la regla propuesta en la Definición 5.2.1. En lo que sigue, seguirá siendo válida la asunción de que toda regla de definición pierde su status una vez que se aplica sobre ella cualquier regla de transformación.

La introducción de una nueva definición se considera normalmente el primer paso de una secuencia de transformación. El determinar qué definiciones deben introducirse es una tarea que cae en el terreno de las estrategias de transformación, como discutiremos en la Sección 5.3 (ver [Pettorossi y Proietti, 1994] para más detalles). Esta identificación de la definiciones a introducir requiere un cierto nivel de ingenio que es muy difícil de mecanizar, y ésta es la razón por la que en la literatura muchas veces se suele hablar de definiciones *eureka* (del griego *euriskes*, encontrar). En cualquier caso, su uso es crucial para realizar un renombramiento de (alguna parte de) una expresión dada, de tal forma que sea posible derivar una versión más eficiente de la nueva función mediante el posterior plegado/desplegado.

En la literatura relacionada encontramos definiciones similares a la nuestra. Esta regla ya aparece formulada en la primera propuesta de Burstall y Darlington [1977], donde se llama directamente "definición" y consiste en la introducccción de una nueva ecuación cuya parte izquierda no sea una instancia de la parte izquierda de ninguna otra ecuación previa. Sin embargo, como ya hemos avanzado, en nuestra formulación estamos mucho más cercanos a la primera adaptación que hacen Tamaki y Sato [1984] al caso lógico, con las siguientes diferencias:

1. En nuestro caso, la parte izquierda de una regla de definición debe ser un patrón líneal de la forma $f(\overline{t_n})$ donde es suficiente que los términos $\overline{t_n}$ no compartan variables entre sí (por la restricción de linealidad), pero no es necesario que todos ellos sean variables sino simplemente términos constructores, a diferencia de la definición de Tamaki y Sato [1984] donde se exige que la cabeza de la cláusula nueva sea un átomo de la forma $q(X_1, \ldots, X_n)$ donde $X_1, \ldots, X_n$ han de ser variables (distintas). Tal relajamiento en nuestra definición nos permitirá despues simplificar nuestra noción de regla de abstracción, al tiempo que no altera el carácter inductivamente secuencial del programa transformado.

2. En la definición 5.2.1, exigimos que los conjuntos de variables de las partes izquierda y derecha de una regla de definición sean idénticos, mientras que en el caso lógico no es necesaria ninguna asumción de este tipo. La razón estriba en que en el contexto integrado únicamente se consideran programas aquellos sistemas de reescritura de términos que no contienen variables extra en las partes derechas de sus reglas. La condición $\mathcal{V}ar(f(\overline{t_n})) = \mathcal{V}ar(r)$ que exigimos sobre las reglas de definición justamente impide que en pasos posteriores de plegado puedan aparecer reglas no válidas. Por ejemplo, el plegado de la regla $\mathtt{old(X)} \to \mathtt{h(X)}$ con respecto a la regla de definición $\mathtt{new(X,Y)} \to \mathtt{h(X)}$, produciría la regla plegada no permitida $\mathtt{old(X)} \to \mathtt{new(X,Y)}$.

Nótese que la regla de introducción de una definición propuesta no precisa de requerimientos de aplicabilidad adicionales, ya que sólo nos interesa preservar la semántica asociada a objetivos compuestos por símbolos pertenecientes a la signatura original del primer programa de la secuencia de transformación.

En otras aproximaciones [Maher, 1987b; Pettorossi y Proietti, 1998] se permite introducir de una vez un conjunto de reglas (o cláusulas) -incluso posiblemente recursivas- para definir un mismo símbolo de función (o predicado) nuevo. En nuestro caso, este aspecto no se ha considerado ya que no aporta ventajas significativas en la práctica y sólo complica la definición. De hecho, mientras una regla única encabezada por un patrón mantiene la secuencialidad inductiva de un programa (como se demuestra en [Alpuente *et al.*, 1999f]), no ocurre lo mismo cuando son varias las reglas de definición, a menos que se explicite el hecho de que el conjunto de patrones que constituyen las partes izquierdas de todas ellas se pueda organizar en forma de árbol definicional.

A continuación introducimos nuestra noción de eliminación de una definición.

Definición 5.2.2 (eliminación de una definición) Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq 0$, una secuencia de transformación. Podemos obtener el programa $\mathcal{R}_{k+1}$ eliminando del programa $\mathcal{R}_k$ el conjunto de reglas R^f que definen a un símbolo de función nuevo f, tal que f no aparece en $\mathcal{R}_0$ ni en $\mathcal{R}_k - R^f$.

A diferencia de las reglas anteriores, la eliminación de una definición no aparece en las propuestas originales de Burstall y Darlington [1977] ni Tamaki y Sato [1984]. Su primera aparición la encontramos en el contexto lógico puro con el nombre de "borrado" (*deletion*) en [Maher, 1993] y también en [Bossi y Cocco, 1993], donde se llama "restricción" (*restriction*).

Un detalle importante a tener en cuenta con esta regla es que la eliminación de las reglas que definen una función f implica que posteriormente no se permite ninguna llamada a f. Este hecho nuevamente se relaciona con la idea de que solamente estamos interesados en preservar la semántica de aquellos objetivos construidos usando los símbolos del programa inicial, los cuales nunca desaparecen en una secuencia de transformación. Con todo, existe el riesgo de que los subsiguientes pasos de transformación reintroduzcan los símbolos eliminados en las partes derechas de algunas reglas transformadas, produciendo inconsistencias en los programas resultantes. En particular, la operación de plegado es (la única) capaz de corromper el proceso de transformación en este sentido.

Ejemplo 44 Consideremos el siguiente programa inductivamente secuencial $\mathcal{R}_i$:

$$\begin{array}{lcll} \texttt{f(X)} & \rightarrow & \texttt{h(X)} & (R_1) \\ \texttt{g(X)} & \rightarrow & \texttt{h(X)} & (R_2) \\ \texttt{h(X)} & \rightarrow & \texttt{0} & (R_3) \end{array}$$

Asumamos que R_2 es una regla de definición. Si realizamos una eliminación de la definición de `g`, obtenemos el programa $\mathcal{R}_{i+1} = \mathcal{R}_i - \{R_2\}$. Ahora, si plegamos $R_1 \in \mathcal{R}_{i+1}$ usando la regla plegante $R_2 \in \mathcal{R}_i$ obtenemos el programa transformado $\mathcal{R}_{i+2}$ siguiente:

$$\begin{array}{lcll} \texttt{f(X)} & \rightarrow & \texttt{g(X)} & (R'_1) \\ \texttt{h(X)} & \rightarrow & \texttt{0} & (R_3) \end{array}$$

Ahora observamos que el término `f(X)` puede reducirse a 0 en $\mathcal{R}_i$ y $\mathcal{R}_{i+1}$ pero no en $\mathcal{R}_{i+2}$.

La forma más común de evitar este inconveniente consiste en imponer el clásico requerimiento [Pettorossi y Proietti, 1994] de no permitir pasos de plegado después de haber aplicado la regla de eliminación de una definición. Para respetar esta condición, la forma más razonable de aplicar las reglas de transformación dentro de una secuencia completa consiste en adelantar todas los pasos de introducción de definiciones al principio de la misma, y postergar todos los pasos de eliminación de definiciones al final de ella. Con esta metodología se consigue que la franja central de una secuencia de transformación consista sólo de aplicaciones de las reglas de plegado/desplegado, con lo que en este tramo no cambia la signatura de los programas y obtenemos una situación idéntica a la planteada en las secuencias de transformación virtuales.

De forma alternativa, una secuencia de transformaciones en la que no se dan pasos de plegado después de haber eliminado definiciones, siempre se puede reordenar sin cambiar los programas inicial y final de tal forma que aparezcan tres tramos (alguno de ellos posiblemente vacío) con las siguientes características:

1. En el primer tramo únicamente se aplica la regla de introducción de definiciones. Con esto se consigue preservar la secuencialidad inductiva de los programas y, a pesar de que la signatura de los mismos se incrementa en cada paso de transformación, la semántica de todos ellos se preserva con respecto a los objetivos construidos sólo con símbolos del programa inicial, ya que en ningún momento se usan las reglas que definen los símbolos nuevos.

2. El tramo central no es más que una secuencia de transformación virtual, para la cual ya hemos demostrado los resultados de corrección y completitud fuerte y el mantenimiento de la secuencialidad inductiva. Estos resultados son más generales incluso que los que nosotros requerimos actualmente, ya que se refieren a objetivos sobre una signatura extendida. Por tanto, en particular, tenemos garantizada también la corrección y completitud fuerte para la clase particular de objetivos que consideramos ahora.

3. En el tercer tramo sólo se permite eliminar definiciones. Esto obviamente no altera la secuencialidad inductiva de los programas y, de forma similar al primer tramo, la semántica asociada a los objetivos con símbolos del programa inicial no se altera, ya que en las derivaciones no se hace uso de las reglas eliminadas (que, por definición, se refieren a símbolos nuevos).

Como el proceso de reordenación de secuencias de transformación que acabamos de describir siempre es posible mientras no se efectúen pasos de plegado después de otros de eliminación, podemos garantizar la corrección y la completitud fuerte (con respecto a los objetivos sin símbolos nuevos) del sistema de transformación descrito en esta sección. Además, partiendo de un programa inductivamente secuencial inicial, todos los transformados por las reglas de introducción/eliminación de definiciones y plegado/desplegado, pertenecen también a esta misma clase.

5.2.2 Reglas de abstracción

El conjunto de reglas presentado hasta ahora es suficiente para automatizar la estrategia de composición. Sin embargo, aún es posible incrementar más la potencia del sistema de transformación hasta conseguir implementar la estrategia de formación de tuplas. Para ello es imprescindible extender el sistema con algún tipo de regla de abstracción (también conocida como *where–abstraction* [Pettorossi y Proietti, 1996b]) al estilo de Burstall y Darlington [1977].

Esta regla consiste esencialmente en el reemplazamiento de ocurrencias de una misma expresión e en la parte derecha de una regla R por una variable nueva X, añadiendo la "declaración local" $X = e$ dentro de una expresión *where* en R.

Ejemplo 45 Siguiendo este método, podemos transformar la regla

$$\mathtt{double\_sum(X,Y)} \ \rightarrow \ \mathtt{sum(sum(X,Y),sum(X,Y))}$$

en la nueva regla

$$\mathtt{double\_sum(X,Y)} \ \rightarrow \ \mathtt{sum(Z,Z)\ where\ Z = sum(X,Y).}$$

En el caso más general, se pueden hacer varios reemplazamientos de una vez, de tal forma que un conjunto de expresiones $\{e_1, \ldots, e_j\}$ en la parte derecha de una regla R, son sustituidas por un conjunto de variables nuevas $\{X_1, \ldots, X_j\}$, añadiendo dentro de una expresión *where* en R el conjunto de declaraciones locales $(X_1 = e_1, \ldots, X_j = e_j)$ o, equivalentemente, haciendo uso del constructor distinguido $\langle\ \rangle$ para representar tuplas, la declaración $\langle X_1, \ldots, X_j\rangle = \langle e_1, \ldots, e_j\rangle$.

Como indican Pettorossi y Proietti [1996b], el uso de la regla de abstracción tiene la ventaja de que, en un modo de ejecución de impaciente, la evaluación de una expresión e se realiza una sola vez ya que, tras una única evaluación de e, su valor asociado queda enlazado a una variable que propaga esta información a todos los contextos que la requieran, sin necesidad de tener que realizar nuevas reevaluaciones. Este hecho también se verifica en contextos perezosos o de llamada por nombre siempre y cuando se disponga de una implementación basada en *sharing* (o compartición de variables) ya que, en este caso, la aparición de una misma variable en diferentes posiciones siempre se refiere a una misma representación interna de la misma.

El principal problema que presenta la adaptación de esta regla al contexto integrado es que, por definición, las variables locales declaradas en una construcción *where* suponen la aparición de variables extra en las partes derechas de las reglas transformadas. Sin embargo, como se indica en [Sands, 1996], este problema puede ser solucionado fácilmente usando las técnicas estándar de eliminación de expresiones "lambda" (*lambda lifting*) de la programación funcional [Johnsson, 1985]. Estas técnicas permiten transformar programas funcionales con declaraciones de funciones locales (a través de construcciones del tipo *where* o *let*), posiblemente con variables libres en las definiciones de las funciones, en programas consistentes sólo en conjuntos de definiciones globales de funciones que se usan como reglas de reescritura. Se ha demostrado que estas técnicas son correctas en un compilador para ML perezoso considerando una máquina de reducción de grafos [Johnsson, 1985]. Recientemente, esta técnica se ha introducido también en el contexto lógico-funcional integrado (ver [Hanus (ed.), 1999]). En nuestro entorno, utilizamos una versión restringida de esta técnica basada en la eliminación de expresiones lambda, que goza de la propiedad de poder ser planteada en términos de reglas de transformación como una introducción de definición seguida de un paso de plegado apropiado [Sands, 1996].

Ejemplo 46 Si consideramos de nuevo la regla

$$\mathtt{double\_sum(X,Y)} \ \rightarrow \ \mathtt{sum(Z,Z)}\ \mathtt{where}\ \mathtt{Z} = \mathtt{sum(X,Y)},$$

vemos que es posible transformarla por eliminación de expresiones lambda [Johnsson, 1985] en el nuevo par de reglas

$$\begin{array}{rcl} \mathtt{double\_sum(X,Y)} & \rightarrow & \mathtt{ds\_aux(sum(X,Y))} \\ \mathtt{ds\_aux(Z)} & \rightarrow & \mathtt{sum(Z,Z)} \end{array}$$

Nótese que este par de reglas se puede generar de forma alternativa a partir de la definición original introduciéndo la nueva regla de definición ($\mathtt{ds\_aux(Z)} \rightarrow \mathtt{sum(Z,Z)}$) y, posteriormente, plegando la regla original a la expresión $\mathtt{sum(sum(X,Y),sum(X,Y))}$ usando como regla plegante la definición recién generada para `ds_aux/1`.

La inclusión de la regla de abstracción es tradicional en los marcos de plegado/desplegado para programas funcionales [Burstall y Darlington, 1977; Pettorossi y Proietti, 1996b; Sands, 1996; Scherlis, 1981]. En el caso de los programas lógicos, este tipo de transformación es simulable mediante la así llamada estrategia de *generalización* [Pettorossi y Proietti, 1996b]. Básicamente esta técnica actúa generalizando algunas llamadas (lo que es asimilable al aplanamiento de expresiones anidadas) para eliminar las barreras que impidan un paso de plegado posterior.

A continuación pasamos a formalizar nuestra primera aproximación a la regla de abstracción haciendo uso de una técnica similar a la eliminación de expresiones lambda. Para una secuencia de posiciones disjuntas $P = \overline{p_n}$, denotamos con $t[\overline{s_n}]_P$ el término obtenido como $(((t[s_1]_{p_1})[s_2]_{p_2})\ldots[s_n]_{p_n})$. Por abuso, denotamos $t[\overline{s_n}]_P$ como $t[s]_P$ cuando $s_1 = \ldots = s_n = s$, así como $((t[s_1]_{P_1})\ldots[s_n]_{P_n})$ como $t[\overline{s_n}]_{\overline{P_n}}$.

Definición 5.2.3 (abstracción simple) Sea $(\mathcal{R}_0,\ldots,\mathcal{R}_k)$, $k \geq 0$, una secuencia de transformación. Sea $R = (f(\overline{t_n}) \rightarrow r) \in \mathcal{R}_k$ una regla y P una secuencia de posiciones disjuntas en $\mathcal{FP}os(r)$ tal que $r|_p = e$ para todo p en P, i.e., $r = r[e]_P$. Podemos obtener el programa $\mathcal{R}_{k+1}$ a partir del programa $\mathcal{R}_k$ como sigue:

$$\mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) \cup \{f(\overline{t_n}) \to f\_aux(\overline{y_m}, e), \quad f\_aux(\overline{y_m}, z) \to r[z]_P\}$$

donde z es una variable nueva que no aparece en $\overline{t_n}$, f_aux es un símbolo de función nuevo que no aparece en $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, y $\mathcal{V}ar(r[z]_P) = \{\overline{y_m}, z\}$.

El siguiente ejemplo ilustra esta definición de abstracción simple.

Ejemplo 47 El máximo elemento de una lista de números naturales puede calcularse mediante el siguiente programa $\mathcal{R}$:

$$\begin{array}{rcl} \texttt{max([ ])} & \to & \texttt{0} \\ \texttt{max([H|T])} & \to & \texttt{if H > max(T) then H else max(T)} \end{array}$$

Obsérvese que este programa es ineficiente debido a la doble evaluación de `max(T)` en la segunda regla, lo que conlleva una complejidad algorítmica de $O(2^n)$, donde n representa la longitud de la lista. Por aplicación de la regla de abstracción simple, obtenemos el siguiente programa transformado (con un comportamiento mucho más eficiente de $O(n)$):

$$\begin{array}{rcl} \texttt{max([ ])} & \to & \texttt{0} \\ \texttt{max([H|T])} & \to & \texttt{max\_aux(H,max(T))} \\ \texttt{max\_aux(H,Z)} & \to & \texttt{if H > Z then H else Z} \end{array}$$

Nótese que este programa también se puede obtener de forma alternativa introduciendo la última regla por un paso de introducción de definición y, posteriormente, plegando la segunda regla de $\mathcal{R}$ usando esta última.

A continuación pasamos a generalizar y extender la anterior regla de abstracción simple con el objetivo de poder abstraer varias expresiones de una sola vez. Esta será la noción de abstracción que utilizaremos en la práctica, ya que la abstracción simple no es más que un caso particular de ella. Para facilitar la definición hacemos uso del constructor de tuplas $\langle\ \rangle$.

Definición 5.2.4 (abstracción) Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, $k \geq 0$, una secuencia de transformación. Sea $R = (f(\overline{t_n}) \to r) \in \mathcal{R}_k$ una regla y sean $\overline{P_j}$ secuencias de posiciones disjuntas en $\mathcal{FP}os(r)$ tal que $r|_p = e_i$ para todo p en P_i, $i = 1, \ldots, j$, i.e., $r = r[\overline{e_j}]_{\overline{P_j}}$. Podemos obtener el programa $\mathcal{R}_{k+1}$ a partir del programa $\mathcal{R}_k$ como sigue:

$$\begin{array}{rcl} \mathcal{R}_{k+1} = (\mathcal{R}_k - \{R\}) & \cup & \{\ f(\overline{t_n}) \to f\_aux(\overline{y_m}, \langle e_1, \ldots, e_j\rangle) \\ & & \quad f\_aux(\overline{y_m}, \langle z_1, \ldots, z_j\rangle) \to r[\overline{z_j}]_{\overline{P_j}}\ \} \end{array}$$

donde $\overline{z_j}$ son variables nuevas que no aparecen en $\overline{t_n}$, f_aux es un símbolo de función nuevo que no aparece en $(\mathcal{R}_0, \ldots, \mathcal{R}_k)$, y $\mathcal{V}ar(r[\overline{z_j}]_{\overline{P_j}}) = \{\overline{y_m}, \overline{z_j}\}$.

Informalmente, el par de reglas generado por la regla de abstracción puede entenderse como una variante sintáctica de la siguiente regla:

$$f(\overline{t_n}) \ \to\ r[\overline{z_j}]_{\overline{P_j}} \ \ where\ \langle z_1, \ldots, z_j\rangle = \langle e_1, \ldots, e_j\rangle$$

que usa la notación convencional de la declaraciones locales con `where`. En lo que sigue utilizaremos esta notación más convencional por razones de legilibilidad aunque siempre teniendo en cuenta que, en la representación interna, las reglas abstraidas no contienen variables extra.

Por otra parte, cabe destacar que la regla de abstracción general presentada en la Definición 5.2.4 puede expresarse también en términos de introducción de una definición y posterior plegado. Gracias a esta interesante propiedad, los dos tipos de abstracción definidos heredan directamente todos los resultados de corrección y completitud fuerte de que disfrutan las reglas de introducción de una definición y plegado (incluida también la propiedad de preservar la estructura inductivamente secuencial de los programas).

5.2.3 Reemplazamiento algebraico

Nuestra última regla de transformación permite reordenar ciertas porciones de la parte derecha de una regla con el objetivo de permitir posteriormente la aplicación de otros pasos transformación. La regla se inspira en las así llamadas "leyes para primitivas" del sistema de transformación original de Burstall y Darlington [1977] que, básicamente permiten la transformación de una ecuación en otra aplicando propiedades tales como la commutatividad, asociatividad o distributividad de ciertos operadores considerados como primitivos (e.g., $+$, $\times$, `if_then_else`).

En un contexto lógico, esta regla se suele denominar genéricamente como "reemplazamiento del objetivo" (*goal replacement*). En [Tamaki y Sato, 1984], esta regla genérica de transformación se describe mediante una caracterización semántica, cuya verificación no siempre es factible por medios sintácticos (en particular, las dificultades surgen al intentar demostrar que el invariante (I2) de las secuencias de transformación virtuales se mantiene tras la aplicación de esta regla). Tan sólo en una serie de casos concretos y particulares se pueden dar versiones restringidas del reemplazamiento algebraico que pueden probarse correctas y completas. En particular, algunas de estas caracterizaciones más populares que han ido surgiendo en la literatura especializada son la eliminación, mezcla y adición de objetivos y cláusulas [Tamaki y Sato, 1984; Bossi y Cocco, 1993; Bossi *et al.*, 1992; Pettorossi y Proietti, 1998].

Ya que la sintaxis de los programas lógico-funcionales es muy similar a la de los funcionales puros, a continuación formulamos nuestra regla de reemplazamiento algebraico de forma similar a como se hace en [Pettorossi y Proietti, 1996b]:

Definición 5.2.5 (reemplazamiento algebraico) Derivamos una nueva regla de programa usando las igualdades que se dan cuando se interpretan las variables de función como los menores puntos fijos de las correspondientes ecuaciones, e interpretamos los símbolos de función básicos, como $+$, $\times$ y otros, como operadores definidos en un álgebra dada.

Por ejemplo, la ecuación $\mathtt{f(n+1)} = \mathtt{f(n)} \times \mathtt{(n+1)}$ puede ser derivada a partir de la ecuación $\mathtt{f(n+1)} = \mathtt{(n+1)} \times \mathtt{f(n)}$, si interpretamos la operación usual de multiplicación sobre números naturales, usando la igualdad $\mathtt{f(n)} \times \mathtt{(n+1)} = \mathtt{(n+1)} \times \mathtt{f(n)}$.

Aunque en general la aplicación de la regla de reemplazamiento algebraico es difícil de automatizar, su uso puede ser transcendental para poder posibilitar pasos subsiguientes de transformación. En particular, es muy frecuente la aplicación de leyes de commutatividad, asociatividad o distributividad para conseguir reordenar o anidar expresiones de tal forma que puedan ser posteriormente plegadas [Pettorossi y Proietti, 1996b].

Un detalle técnico a tener en cuenta al aplicar nuestra noción de reemplazamiento algebraico es que, contrariamente a lo que venimos asumiendo hasta ahora, esta transformación no debe alterar el estatus de las reglas de definición. En particular, este hecho debe verificarse si queremos preservar el invariante (I2) en una secuencia de transformación, siendo éste un aspecto fundamental para garantizar la corrección de la transformación, como ilustra el siguiente ejemplo.

Ejemplo 48 Supongamos que en un programa $\mathcal{R}_i$ disponemos de la siguiente regla de definición R:

$$\mathtt{new(X,Y)} \quad \rightarrow \quad \mathtt{X} + \mathtt{Y}$$

Al transformar R por reemplazamiento algebraico aplicando la commutatividad de la suma, obtenemos la nueva regla R':

$$\mathtt{new(X,Y)} \quad \rightarrow \quad \mathtt{Y} + \mathtt{X}$$

Ahora, si consideramos (erróneamente) que R' ha perdido su carácter de regla de definición, entonces puede ser plegada con respecto a R generándose la nueva regla R'':

$$\mathtt{new(X,Y)} \quad \rightarrow \quad \mathtt{new(Y,X)}$$

Obviamente el programa resultante ya no preserva la semántica del original debido al hecho de que el invariante (I2) deja de verificarse sobre el mismo pues la regla R'' introduce derivaciones infinitas.

Teniendo en cuenta este hecho y todo lo comentado anteriormente, ya estamos en condiciones de enunciar las propiedades del sistema de transformación propuesto en el siguiente teorema, cuya demostración se sigue directamente de todos los resultados previos demostrados hasta ahora.

Teorema 5.2.6 (propiedades del sistema de transformación)
Sea $(\mathcal{R}_0, \ldots, \mathcal{R}_n)$, $n > 0$, *una secuencia de transformación donde el programa inicial es inductivamente secuencial y los restantes se obtienen por aplicación de las reglas de desplegado necesario, plegado T&S, introducción y eliminación de definiciones, abstracción y reemplazamiento algebraico, tal que:*

1. *no se efectúan pasos de plegado después de la eliminación de una definición y*

2. *toda regla de definición pierde su estatus al aplicarse sobre ella cualquier regla transformación diferente al reemplazamiento algebraico.*

Entonces:

1. $\mathcal{R}_i$ *es inductivamente secuencial para todo* $i = 0, \ldots, n$ *y*

2. $e \overset{\text{NN}}{\leadsto}{}^*_\theta$ *true es una derivación de narrowing necesario en* $\mathcal{R}_0$ *si y solo si existe una derivación de narrowing necesario* $e \overset{\text{NN}}{\leadsto}{}^*_{\theta'}$ *true en* $\mathcal{R}_n$ *tal que* $\theta' = \theta\ [\mathcal{V}ar(e)]$, *para toda ecuación* e *sin símbolos de función nuevos.*

Obsérvese que este resultado de corrección y completitud fuerte implica que la semántica de respuestas computadas no sólo se preserva en el último programa de la secuencia, sino en cualquiera de ellos. Aparte de estos resultados de corrección, el segundo aspecto a tener en cuenta en todo sistema de transformación es su capacidad para producir programas realmente eficientes. Este aspecto depende fundamentalmente de la heurística que se usa para conducir el proceso de aplicación de las reglas de transformación, lo que se conoce genéricamente como estrategias de transformación y que pasamos a estudiar en la siguiente sección.

5.3 Estrategias de transformación

Las estrategias de transformación pueden verse como un conjunto de meta–reglas que se usan para conducir el proceso de aplicación de reglas de transformación con el objetivo de optimizar programas. Estos métodos se suelen formular de forma algorítmica como una serie de heurísticas que describen la secuencia de cambios que sufre un programa mientras es transformado. En [Feather, 1987; Proietti y Pettorossi, 1993; Pettorossi y Proietti, 1994, 1996b] podemos encontrar diferentes tratamientos de estrategias de transformación tanto para programas lógicos como funcionales puros, cuya extrapolación al caso integrado es fácilmente realizable. En la definición de una estrategia de transformación, la naturaleza de los cambios se puede describir de forma abstracta, es decir, independientemente de las particularidades de los programas que van a ser transformados.

Como se afirma en [Pettorossi y Proietti, 1996b], para guiar el proceso de transformación se requiere algún tipo de estrategia si se desea alcanzar una mejora efectiva sobre los programas transformados ya que, mediante un uso arbitrario de las reglas de transformación es posible obtener programas similares a otros ya calculados previamente en la secuencia, debido fundamentalmente a la presencia de transformaciones reversibles.

Uno de los retos más importantes al aplicar una estrategia de transformación consiste en la introducción de reglas nuevas o definiciones *eureka* apropiadas. En los primeros tiempos de la teoría de transformación basada en "reglas + estrategias", estas funciones se generaban de forma inteligente y externa al sistema de transformación y no a través de estrategias. Posteriormente se han hecho algunos avances en el área, como son por ejemplo los trabajos de Wadler [1990] y Chin [1993], donde se proponen algunos análisis que permiten extraer (bajo ciertas condiciones) definiciones eureka especialmente concebidas para implementar las estrategias de deforestación y formación de tuplas, respectivamente.

Aunque en la actualidad no existe todavía una teoría sobre estrategias que asegure que los programas transformados son siempre más eficientes que los originales, sí existen algunos resultados parciales. Recordemos por ejemplo que, en el marco de Sands [1996] para la transformación de programas funcionales perezosos de orden superior, se propone una teoría de *improvement* que, mediante métodos sintácticos, permite guiar y restringir el proceso de plegado/desplegado de tal forma que siempre queda garantizada la corrección total y la mejora en rendimiento de los programas. La idea básica mostrada en [Sands, 1996] es que mientras cada paso de plegado reduce la eficiencia de un programa (en un paso), cada paso de desplegado la incrementa (en un paso). Con esta idea en mente, se trata de establecer un equilibrio entre los pasos de plegado y desplegado, de forma que el número de desplegados sea igual o mayor que el de plegados. Con un planteamiento similar, Kanamori y Fujita [1987] hacen un uso explícito de contadores para contabilizar el número de pasos de plegado/desplegado realizados durante la transformación de un programa lógico. Estas ideas están implícitas en nuestro marco por el matenimiento del rango de los términos como asegura el invariante (I2) (de forma similar a [Tamaki y Sato, 1984]), lo que se relaciona con la condición de mejora de Sands.

En lo que sigue, pasamos a discutir e ilustrar el poder del sistema de transformación propuesto con respecto a las estrategias clásicas de composición [Wadler, 1990] y formación de tuplas [Burstall y Darlington, 1977; Darlington, 1982].

5.3.1 Composición

La estrategia de composición se introdujo originalmente en [Burstall y Darlington, 1977; Darlington, 1982] para la optimización de programas funcionales puros. Los métodos de *especialización interna* de Scherlis [1981] y la *deforestación* de Wadler [1990] son también variantes de esta misma estrategia. La técnica de composición permite evitar la construcción de estructuras de datos intermedias producidas por una cierta función `g` y consumidas por otra función `f` (e.g., cuando aparece una expresión `f(g(_))` en el programa). Esencialmente, la estrategia de composición procede como indica el siguiente algoritmo.

Algoritmo de composición

1. Sea $R = (l = r)$, una regla del programa inicial $\mathcal{R}_0$ tal que:
   (a) $r|_p = f(t_1, \ldots, t_n)$ para alguna posición p,
   (b) $t_i|_q = g(s_1, \ldots, s_m)$, para algún término t_i, $1 \leq i \leq n$, y alguna de sus posiciones q y
   (c) f y g son funciones definidas (i.e., $f, g \in \mathcal{F}$).

   Estas condiciones aseguran la presencia de, al menos, un anidamiento entre las funciones definidas f y g.
2. Usando la regla de introducción de una definición, se introduce una nueva regla de definición: $R' = (new(x_1, \ldots, x_k) \;\; = \;\; r|_p)$, donde $\mathcal{V}ar(r|_p) = \{x_1, \ldots, x_k\}$.
3. Aplicando la regla de desplegado, se despliega reiteradamente la parte derecha de la nueva definición (lo que posiblemente puede implicar algún paso intermedio de reordenamiento por reemplazamiento algebraico para permitir más pasos de desplegado) hasta obtener un conjunto de reglas cuyas partes derechas cumplan una de las siguientes condiciones:
   (a) contienen una instancia de $r|_p$ (i.e., caso general de la definición)
   (b) o bien no poseen llamadas a alguna de las funciones f o g que permitan que éstas se puedan volver a anidar (i.e., los casos base de la definición desplegada).
4. Si no es posible alcanzar la condición de parada anterior (o se realiza un número de pasos superior a una cota máxima prefijada), entonces el proceso termina con fallo. En caso contrario, continuar con el paso siguiente.
5. Mediante la regla plegado, plegar las partes derechas de las reglas desplegadas usando la regla original R'. De esta forma, obtenemos una definición recursiva para la nueva función new.

6. Aplicando de nuevo la regla de plegado, plegamos también $r|_p$ en la regla R usando R' para sustitutir $r|_p$ por una llamada a la función *new*, obteniéndose la nueva regla $R' = (l = r[new(x_1, \ldots, x_k)]_p)$.

Nótese que, en el Ejemplo 40 hemos optimizado siguiendo esta técnica el programa que define la función `doubleappend`. Este es un ejemplo clásico de deforestación.

En algunos casos, muchas de las mejoras en eficiencia que pueden obtenerse con la estrategia de composición también puede conseguirse mediante la simple evaluación perezosa de los programas originales [Feather, 1987]. Sin embargo, la estrategia de composición a menudo permite derivar programas con mejor rendimiento incluso en contextos de evaluación perezosa [Wadler, 1985]. El uso de técnicas de evaluación perezosas es decisivo cuando, en una llamada a función anidada $\mathtt{f(g(X))}$, la estructura de datos intermedia producida por `g` es infinita pero la función `f` puede producir su resultado conociendo únicamente una porción finita de la salida de `g`. El siguiente ejemplo ilustra las ventajas de las reglas de transformación del sistema propuesto frente a las que hemos descartado en capítulos anteriores (fundamentalmente las que se refieren al plegado reversible y las variantes de desplegado basadas en estrategias no perezosas de *narrowing*), ya que con ellas no es posible realizar las optimizaciones perseguidas.

Ejemplo 49 La función $\mathtt{sum\_prefix(X,Y)}$ definida en el siguiente programa $\mathcal{R}_0$ calcula la suma de los `Y` números naturales consecutivos a partir del `X`. La función genera primeramente una lista infinita de numeros consecutivos a partir de `X`, y entonces suma los primeros `Y` números de la lista:

$$
\begin{array}{rcll}
\mathtt{sum\_prefix(X,Y)} & \rightarrow & \mathtt{sum\_sublist(from(X),Y)} & (R_1)\\
\mathtt{sum\_sublist(L,0)} & \rightarrow & \mathtt{0} & (R_2)\\
\mathtt{sum\_sublist([H|T],s(X))} & \rightarrow & \mathtt{H+sum\_sublist(T,X)} & (R_3)\\
\mathtt{from(X)} & \rightarrow & \mathtt{[X|from(s(X))]} & (R_4)\\
\mathtt{0+X} & \rightarrow & \mathtt{X} & (R_5)\\
\mathtt{s(X)+Y} & \rightarrow & \mathtt{s(X+Y)} & (R_6)
\end{array}
$$

Nótese que la función `from` no es terminante (lo que no afecta a la corrección de la transformación). Ahora, es posible incrementar la eficiencia de $\mathcal{R}_0$ evitando la creación y subsecuente uso de la lista parcial intermedia (e infinita) generada por la llamada a la función `from`:

1. Introducción de una definición:

$$\mathtt{new(X,Y)} \rightarrow \mathtt{sum\_sublist(from(X),Y)} \quad (R_7)$$

2. Desplegado de la regla R_7 (nótese que la instanciación es automática) reduciendo

el redex necesario $\mathtt{from(X)}$):

$$\begin{array}{rcll} \mathtt{new(X,0)} & \rightarrow & \mathtt{0} & (R_8) \\ \mathtt{new(X,s(Y))} & \rightarrow & \mathtt{sum\_sublist([X|from(s(X))],s(Y))} & (R_9) \end{array}$$

3. Desplegado de la regla R_9 (cabe destacar que este paso necesario para llegar al programa final no es posible darlo con un tipo de desplegado impaciente) reduciendo el redex necesario encabezado por `sum_sublist`:

$$\mathtt{new(X,s(Y))} \quad \rightarrow \quad \mathtt{X + sum\_sublist(from(s(X)),Y)} \quad (R_{10})$$

4. Plegado del subtérmino $\mathtt{sum\_sublist(from(s(X)),Y)}$ en la regla R_{10} usando la regla de definición R_7 (obsérvese que este paso no puede darse con ningún tipo de plegado distinto al plegado T&S, ya que la regla plegante no está en el programa anterior):

$$\mathtt{new(X,s(Y))} \quad \rightarrow \quad \mathtt{X + new(s(X),Y))} \quad (R_{11})$$

5. Plegado de la parte derecha de la regla R_1 usando R_7:

$$\mathtt{sum\_prefix(X,Y)} \quad \rightarrow \quad \mathtt{new(X,Y)} \quad (R_{12})$$

Ahora, el programa transformado $\mathcal{R}_5$ está formado por las siguientes reglas:

$$\begin{array}{rcll} \mathtt{sum\_prefix(X,Y)} & \rightarrow & \mathtt{new(X,Y)} & (R_{12}) \\ \mathtt{new(X,0)} & \rightarrow & \mathtt{0} & (R_8) \\ \mathtt{new(X,s(Y))} & \rightarrow & \mathtt{X + new(s(X),Y))} & (R_{11}) \end{array}$$

(junto con las definiciones originales para $+$, `from`, y `sum_sublist`).

El uso del desplegado necesario es esencial en el ejemplo anterior, ya que asegura que no se producen reglas redundantes (como sí hace el desplegado condicional), permite la transformación incluso ante la presencia de funciones no terminantes (a diferencia del desplegado impaciente) y preserva la misma clase de programas a lo largo de toda la secuencia de transformación (a diferencia del desplegado perezoso).

5.3.2 Formación de tuplas

La estrategia de formación de tuplas fue introducida originalmente en [Burstall y Darlington, 1977; Darlington, 1982] para la optimización de programas funcionales. Esta estrategia resulta muy efectiva cuando aparecen en el programa varias llamadas a función que requieren la computación de una misma expresión, en cuyo caso es posible obtener una única función nueva que devuelve (en una *tupla*) el resultado de las distintas llamadas, evitando así el acceso múltiple a una misma estructura de datos y computaciones redundantes. De esta forma, es posible producir programas lineales (i.e., programas cuyas partes derechas contienen como mucho una llamada recursiva) a partir de programas no lineales [Pettorossi y Proietti, 1996b]. Esta clase de optimización no siempre puede conseguirse por métodos de transformación puramente automáticos como la evaluación parcial o la estrategia de composición.

El algoritmo de formación de tuplas es similar al de composición pero con la particularidad de que ahora los pasos de plegado van siempre precedidos por abstracciones que reordenan las reglas a plegar para posibilitar la transformación. Intuitivamente, la estrategia de composición pretende fundir dos llamadas anidadas en una sola llamada a una función nueva. El hecho de que originalmente estas llamadas aparezcan anidadas implica normalmente que, tras la búsqueda de regularidades (tras varios pasos de desplegado), los mismos anidamientos se repiten en las reglas desplegadas, forzando así un paso de plegado que es directamente aplicable sobre las llamadas anidadas (lo que se conoce como *need for folding*).

Por otra parte, cuando las llamadas a computar de forma conjunta no están anidadas, es necesario usar el constructor de tuplas "$\langle\rangle$" y la regla de abstracción para conseguir fundir las mismas y obtener un efecto similar al del anidamiento[1]. Este es el caso de la formación de tuplas, donde las llamadas que se pretende fusionar no están anidadas, sino que se distribuyen arbitrariamente sobre las reglas originales y desplegadas. Para conseguir fundirlas en una misma estructura que pueda ser plegada posteriormente, es preciso "recolectar" dichas llamadas (tanto en la regla original como en las desplegadas) y "empaquetarlas" en una tupla mediante la aplicación de la regla de abstracción. Solamente así es posible plegar las tuplas de expresiones y sustituirlas por una llamada a función nueva. De esta forma, queda clara la necesidad de usar tuplas y reglas de abstracción para implementar la estrategia de formación de tuplas.

Otro detalle importante conectado con lo que acabamos de decir, es que si bien las reglas de definición que se introducen en la estrategia de composición se caracterizan por tener llamadas (a optimizar) anidadas, en la estrategia de formación de tuplas estas reglas de definición se usan para definir tuplas de expresiones cuyas evaluaciones se pretende realizar en un solo cálculo.

Algoritmo de formación de tuplas

1. Sea $R = (f(x_1, \ldots, x_n) = r$ una regla del programa original tal que $r = r[\overline{e_i}]_{\overline{P_i}}$, donde P_i es el conjunto de posiciones asociadas a la expresión e_i para todo $i = 1, \ldots, k$ y, además, $e_1, \ldots, e_k$ son expresiones encabezadas por símbolos de función definidos y que comparten al menos una misma variable x.

[1] En general, cualquier constructor puede ser válido para cumplir este cometido, pero de forma estándar se suelen utilizar tuplas para empaquetar un número variable de expresiones.

2. Mediante la regla de introducción de una definición, introducimos la nueva definición: $R' = (new(x, y_1, \ldots, y_m) = \langle \overline{e_i} \rangle$, donde $x, y_1, \ldots, y_m$ son las variables libres de $e_1, \ldots, e_k$.
3. De forma similar al paso (3) del algoritmo de composición, desplegamos reiteradamente la parte derecha de la nueva definición (con posibles pasos intermedios de reemplazamiento algebraico) hasta obtener un conjunto de reglas cuyas partes derechas cumplan una de las siguientes condiciones:
   (a) contienen al menos alguna instancia de todas y cada una de las expresiones $e_1, \ldots, e_k$ (caso general de la definición), es decir, las reglas de este tipo tienen el formato:
   $$new(t_0, t_1, \ldots, t_m) = (C[\overline{e_i}]_{\overline{Q_i}})\theta,$$
   donde $t_0, t_1 \ldots, t_m$, C, Q_i y θ son, respectivamente, una secuencia de términos, un contexto, un conjunto de posiciones y una sustitución cualquiera,
   (b) o bien no poseen llamadas a alguna de las expresiones e_i (i.e., los casos base de la definición desplegada).
4. Si no es posible alcanzar la condición de parada anterior (o se realiza un número de pasos superior a una cota máxima prefijada), entonces el proceso termina con fallo. En caso contrario, continuar con el paso siguiente.
5. Mediante la aplicación de la regla de abstracción, transformamos las reglas de la forma:
   $$new(t_0, t_1, \ldots, t_m) = (C[\overline{e_i}]_{\overline{Q_i}})\theta,$$
   en otras con el aspecto:
   $$new(t_0, t_1, \ldots, t_m) = (C[\overline{z_i}]_{\overline{Q_i}} \ \ where \ \ \langle \overline{z_i} \rangle = \langle \overline{e_i} \rangle)\theta,$$
   y, por aplicación de plegado usando R', tenemos:
   $$new(t_0, t_1, \ldots, t_m) = (C[\overline{z_i}]_{\overline{Q_i}} \ \ where \ \ \langle \overline{z_i} \rangle = new(x, y_1, \ldots, y_m))\theta,$$
   donde $z_1, \ldots, z_k$ son variables nuevas.
6. Finalmente, y de forma similar a los pasos de abstracción y plegado dados en el paso anterior, transformamos la regla original R en la nueva regla
   $$f(x_1, \ldots, x_n) = r[\overline{z_i}]_{\overline{P_i}} \ \ where \ \ \langle \overline{z_i} \rangle = new(x, y_1, \ldots, y_m)$$
   donde $z_1, \ldots, z_k$ son variables nuevas.

El siguiente ejemplo ilustra la estrategia de formación de tuplas.

Ejemplo 50 Los números de fibonacci pueden calcularse (de forma natural pero ineficiente) con el siguiente programa $\mathcal{R}_0$:

$$\begin{array}{rcll} \mathtt{fib(0)} & \to & \mathtt{s(0)} & (R_1) \\ \mathtt{fib(s(0))} & \to & \mathtt{s(0)} & (R_2) \\ \mathtt{fib(s(s(X)))} & \to & \mathtt{fib(s(X)) + fib(X)} & (R_3) \end{array}$$

(junto con las reglas para la suma $+$). Cabe destacar que este programa tiene una complejidad exponencial, que puede reducirse a lineal aplicando la estrategia de formación de tuplas como sigue:

1. Introducción de una definición:

   $$\mathtt{new(X)} \;\to\; \langle \mathtt{fib(s(X)), fib(X)} \rangle \quad (R_4)$$

2. Desplegado de la regla R_4 reduciendo el redex $\mathtt{fib(s(X))}$:

   $$\begin{array}{rcll} \mathtt{new(0)} & \to & \langle \mathtt{s(0), fib(s(0))} \rangle & (R_5) \\ \mathtt{new(s(X))} & \to & \langle \mathtt{fib(s(X)) + fib(X), fib(s(X))} \rangle & (R_6) \end{array}$$

3. Desplegado de la regla R_5 reduciendo $\mathtt{fib(s(0))}$:

   $$\mathtt{new(0)} \;\to\; \langle \mathtt{s(0), s(0)} \rangle \quad (R_7)$$

4. Abstracción de R_6:

   $$\begin{array}{rcll} \mathtt{new(s(X))} & \to & \mathtt{new\_aux(\langle fib(s(X)), fib(X) \rangle)} & (R_8) \\ \mathtt{new\_aux(\langle Z_1, Z_2 \rangle)} & \to & \langle \mathtt{Z_1 + Z_2, Z_1} \rangle & (R_9) \end{array}$$

5. Plegado de $\langle \mathtt{fib(s(X)), fib(X)} \rangle$ en la regla R_8 usando R_4:

   $$\mathtt{new(s(X))} \;\to\; \mathtt{new\_aux(new(X))} \quad (R_{10})$$

6. Abstracción de la regla R_3:

   $$\begin{array}{rcll} \mathtt{fib(s(s(X)))} & \to & \mathtt{fib\_aux(\langle fib(s(X)), fib(X) \rangle)} & (R_{11}) \\ \mathtt{fib\_aux(\langle Z_1, Z_2 \rangle)} & \to & \mathtt{Z_1 + Z_2} & (R_{12}) \end{array}$$

7. Plegado de $\langle \mathtt{fib(s(X)), fib(X)} \rangle$ en la regla R_{11} usando de nuevo la regla R_4:

   $$\mathtt{fib(s(s(X)))} \;\to\; \mathtt{fib\_aux(new(X))} \quad (R_{13})$$

Ahora, el programa transformado (y más eficiente) $\mathcal{R}_7$, que tiene una complejidad lineal gracias al uso de la función recursiva `new`, es el siguiente:

$$\begin{array}{rcll} \mathtt{fib(0)} & \to & \mathtt{s(0)} & (R_1) \\ \mathtt{fib(s(0))} & \to & \mathtt{s(0)} & (R_2) \\ \mathtt{fib(s(s(X)))} & \to & \mathtt{Z_1 + Z_2 \ where\ \langle Z_1, Z_2 \rangle = new(X)} & (R_{12}, R_{13}) \\ \mathtt{new(0)} & \to & \langle \mathtt{s(0), s(0)} \rangle & (R_7) \\ \mathtt{new(s(X))} & \to & \mathtt{\langle Z_1 + Z_2, Z_1 \rangle \ where\ \langle Z_1, Z_2 \rangle = new(X)} & (R_9, R_{10}) \end{array}$$

donde por legibilidad las reglas (R_{12}, R_{13}) y (R_9, R_{10}) se han expresado usando declaraciones locales.

El siguiente ejemplo muestra que, a diferencia del método de composición, la estrategia de formación de tuplas no es siempre fácil de aplicar de forma completamente automática.

Ejemplo 51 Consideremos el problema clásico de las Torres de Hanoi, definido de forma natural (pero ineficiente) por el siguiente programa $\mathcal{R}_0$:

$$\begin{array}{rcll} \mathtt{h(0,A,B,C)} & \rightarrow & \mathtt{nil} & (R_1) \\ \mathtt{h(s(N),A,B,C)} & \rightarrow & \mathtt{app(h(N,A,C,B),mov(A,B):h(N,C,B,A))} & (R_2) \end{array}$$

(junto con las reglas para app). De forma similar al Ejemplo 50, este programa tiene una complejidad exponencial debido a las dos llamadas recursivas en la parte derecha de la regla R_2. A diferencia de los números de fibonacci, la aplicación directa de nuestro algoritmo para la formación de tuplas es incapaz de reducir su complejidad algorítmica, como mostramos a continuación.

1. Introducción de una definición:

$$\mathtt{new(N,A,B,C)} \quad \rightarrow \quad \langle \mathtt{h(N,A,B,C),h(N,B,C,A)} \rangle \quad (R_3)$$

2. Desplegado de la regla R_3 reduciendo el redex $\mathtt{h(N,A,B,C)}$:

$$\begin{array}{rcll} \mathtt{new(0,A,B,C)} & \rightarrow & \langle \mathtt{nil,h(0,B,C,A)} \rangle & (R_4) \\ \mathtt{new(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(h(N,A,C,B),mov(A,B):h(N,C,B,A))}, & \\ & & \quad \mathtt{h(s(N),B,C,A)} \rangle & (R_5) \end{array}$$

3. Desplegado de la regla R_4 reduciendo $\mathtt{h(0,B,C,A)}$:

$$\mathtt{new(0,A,B,C)} \quad \rightarrow \quad \langle \mathtt{nil,nil} \rangle \quad (R_6)$$

4. Desplegado de la regla R_5 reduciendo el redex $\mathtt{h(s(N),B,C,A)}$:

$$\begin{array}{rcll} \mathtt{new(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(h(N,A,C,B),mov(A,B):h(N,C,B,A))}, & \\ & & \quad \mathtt{app(h(N,B,A,C),mov(B,C):h(N,A,C,B))} \rangle & (R_7) \end{array}$$

   En este momento, la regla R_7 está lo suficientemente desplegada como para que, mediante abstracción y plegado, se pueda obtener una versión recursiva de new. Sin embargo, debido a que en la parte derecha de esta regla aparecen cuatro llamadas a la función original h, y la regla de definición inicial para new únicamente puede computar dos de esas llamadas de una sola vez, el resultado final de la estrategia de formación de tuplas generaría una regla con dos llamadas recursivas a new, lo que obviamente no supondría ninguna mejora con respecto a la versión original.

   Por otra parte, siguiendo el análisis de Chin [1993] y sabiendo que el *eureka* más apropiado para resolver este problema es aquel que encapsula en una misma tupla las llamadas $\mathtt{h(N,A,B,C)}$, $\mathtt{h(N,B,C,A)}$ y $\mathtt{h(N,C,A,B)}$, podemos reintentar la aplicación de la estrategia de formación de tuplas, pero partiendo ahora de la regla R_7.

5. Introducción de una definición:

$$\mathtt{new2(N,A,B,C)} \quad \rightarrow \quad \langle \mathtt{h(N,A,B,C),h(N,B,C,A),h(N,C,A,B)} \rangle \quad (R_8)$$

6. Desplegado de la regla R_8 reduciendo el redex $\mathtt{h(N,A,B,C)}$:

$$\begin{array}{rcll} \mathtt{new2(0,A,B,C)} & \rightarrow & \langle \mathtt{nil}, \mathtt{h(0,B,C,A)}, \mathtt{h(0,C,A,B)} \rangle & (R_9) \\ \mathtt{new2(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(h(N,A,C,B), mov(A,B):h(N,C,B,A))}, & \\ & & \ \mathtt{h(s(N),B,C,A)}, \mathtt{h(s(N),C,A,B)} \rangle & (R_{10}) \end{array}$$

7. Desplegado de la regla R_9 reduciendo $\mathtt{h(0,B,C,A)}$:

$$\mathtt{new2(0,A,B,C)} \rightarrow \langle \mathtt{nil}, \mathtt{nil}, \mathtt{h(0,C,A,B)} \rangle \quad (R_{11})$$

8. Desplegado de la regla R_{11} reduciendo $\mathtt{h(0,C,A,B)}$:

$$\mathtt{new2(0,A,B,C)} \rightarrow \langle \mathtt{nil}, \mathtt{nil}, \mathtt{nil} \rangle \quad (R_{12})$$

9. Desplegado de la regla R_{10} reduciendo el redex $\mathtt{h(s(N),B,C,A)}$:

$$\begin{array}{rcll} \mathtt{new2(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(h(N,A,C,B), mov(A,B):h(N,C,B,A))}, & \\ & & \ \mathtt{app(h(N,B,A,C), mov(B,C):h(N,A,C,B))}, & \\ & & \ \mathtt{h(s(N),C,A,B)} \rangle & (R_{13}) \end{array}$$

10. Desplegado de la regla R_{13} reduciendo el redex $\mathtt{h(s(N),C,A,B)}$:

$$\begin{array}{rcll} \mathtt{new2(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(h(N,A,C,B), mov(A,B):h(N,C,B,A))}, & \\ & & \ \mathtt{app(h(N,B,A,C), mov(B,C):h(N,A,C,B))}, & \\ & & \ \mathtt{app(h(N,C,B,A), mov(C,A):h(N,B,A,C))} \rangle & (R_{14}) \end{array}$$

11. Abstracción de R_{14}:

$$\begin{array}{rcll} \mathtt{new2(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(U, mov(A,B):V)}, & \\ & & \ \mathtt{app(W, mov(B,C):U)}, & \\ & & \ \mathtt{app(V, mov(C,A):W)} \rangle & \\ & & \qquad \mathtt{where} & \\ & & \langle \mathtt{U,V,W} \rangle = \langle \mathtt{h(N,A,C,B)}, \mathtt{h(N,C,B,A)}, \mathtt{h(N,B,A,C)} \rangle & (R_{15}) \end{array}$$

12. Plegado de $\langle \mathtt{h(N,A,C,B)}, \mathtt{h(N,C,B,A)}, \mathtt{h(N,B,A,C)} \rangle$ en la regla R_{15} usando R_8:

$$\begin{array}{rcll} \mathtt{new2(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(U, mov(A,B):V)}, & \\ & & \ \mathtt{app(W, mov(B,C):U)}, & \\ & & \ \mathtt{app(V, mov(C,A):W)} \rangle & \\ & & \qquad \mathtt{where} & \\ & & \langle \mathtt{U,V,W} \rangle = \mathtt{new2(N,A,C,B)} & (R_{16}) \end{array}$$

13. Abstracción de la regla R_7:

$$\begin{array}{rcll} \mathtt{new(s(N),A,B,C)} & \rightarrow & \langle \mathtt{app(U, mov(A,B):V)}, & \\ & & \ \mathtt{app(W, mov(B,C):U)} \rangle & \\ & & \qquad \mathtt{where} & \\ & & \langle \mathtt{U,V,W} \rangle = \langle \mathtt{h(N,A,C,B)}, \mathtt{h(N,C,B,A)}, \mathtt{h(N,B,A,C)} \rangle & (R_{17}) \end{array}$$

14. Plegado de $\langle \mathtt{h(N,A,C,B)}, \mathtt{h(N,C,B,A)}, \mathtt{h(N,B,A,C)} \rangle$ en la regla R_{17} usando de nuevo la regla R_8:

$$\begin{array}{rcl}\mathtt{new(s(N),A,B,C)} & \to & \langle\mathtt{app(U,mov(A,B):V)},\\ & & \ \mathtt{app(W,mov(B,C):U)}\rangle\\ & & \qquad\quad \mathtt{where}\\ & & \langle\mathtt{U,V,W}\rangle = \mathtt{new2(N,A,C,B)} \quad (R_{18})\end{array}$$

Y, finalmente, pasamos a realizar los últimos pasos de abstracción y plegado en la formación de tuplas de partida.

15. Abstracción de la regla R_2:

$$\begin{array}{rcl}\mathtt{h(s(N),A,B,C)} & \to & \mathtt{app(U,mov(A,B):V)}\\ & & \qquad\quad \mathtt{where}\\ & & \langle\mathtt{U,V}\rangle = \langle\mathtt{h(N,A,C,B),h(N,C,B,A)}\rangle \quad (R_{19})\end{array}$$

16. Plegado de $\langle\mathtt{h(N,A,C,B),h(N,C,B,A))}\rangle$ en la regla R_{19} usando la regla de definición R_3:

$$\begin{array}{rcl}\mathtt{h(s(N),A,B,C)} & \to & \mathtt{app(U,mov(A,B):V)}\\ & & \qquad\quad \mathtt{where}\\ & & \langle\mathtt{U,V}\rangle = \mathtt{new(N,A,C,B)} \quad (R_{20})\end{array}$$

Ahora, el programa transformado (y más eficiente) $\mathcal{R}_{16}$, que tiene complejidad lineal, ya que nunca hay más de una llamada recursiva gracias al uso de las funciones `new` y `new2`, está compuesto por el siguiente conjunto de reglas (junto con las que definen `app`):

$$\begin{array}{rcl}\mathtt{h(0,A,B,C)} & \to & \mathtt{nil} \qquad\qquad\qquad\qquad (R_1)\\ \mathtt{h(s(N),A,B,C)} & \to & \mathtt{app(U,mov(A,B):V)}\\ & & \qquad\quad \mathtt{where}\\ & & \langle\mathtt{U,V}\rangle = \mathtt{new(N,A,C,B)} \quad (R_{20})\\ \mathtt{new(0,A,B,C)} & \to & \langle\mathtt{nil,nil}\rangle \qquad\qquad\qquad (R_6)\\ \mathtt{new(s(N),A,B,C)} & \to & \langle\mathtt{app(U,mov(A,B):V)},\\ & & \ \mathtt{app(W,mov(B,C):U)}\rangle\\ & & \qquad\quad \mathtt{where}\\ & & \langle\mathtt{U,V,W}\rangle = \mathtt{new2(N,A,C,B)} \quad (R_{18})\\ \mathtt{new2(0,A,B,C)} & \to & \langle\mathtt{nil,nil,nil}\rangle \qquad\qquad (R_{12})\\ \mathtt{new2(s(N),A,B,C)} & \to & \langle\mathtt{app(U,mov(A,B):V)},\\ & & \ \mathtt{app(W,mov(B,C):U)},\\ & & \ \mathtt{app(V,mov(C,A):W)}\rangle\\ & & \qquad\quad \mathtt{where}\\ & & \langle\mathtt{U,V,W}\rangle = \mathtt{new2(N,A,C,B)} \quad (R_{16})\end{array}$$

Las estrategias de composición y formación de tuplas que acabamos de estudiar son, probablemente, las que con mayor éxito han sido explotadas para la transformación de programas funcionales y lógicos puros dentro de la aproximación "reglas + estrategias" (ver [Pettorossi y Proietti, 1996b] para un resumen detallado).

En el contexto funcional puro, la estrategia de composición se define generalmente de forma equivalente a como lo hemos hecho en la Sección 5.3.1. Sin embargo, no existe una noción equiparable de composición en el contexto lógico puro, debido a que la sintaxis de los programas lógicos no permite anidamientos de predicados.

Por su parte, la estrategia de formación de tuplas que hemos descrito en la sección 5.3.2 ya es conocida desde la primera propuesta para la transformación de programas funcionales [Burstall y Darlington, 1977], existiendo también adaptaciones al caso lógico, donde se denomina "formación de tuplas de predicados" [Debray, 1988; Bruynooghe *et al.*, 1989].

En este último contexto, la formación de tuplas de predicados tiene el poder de conseguir optimizaciones similares (evitar la construcción y visita múltiple de estructuras de datos intermedias, o la evaluación de subexpresiones comunes y otras computaciones redundantes) a las que se obtienen por composición y/o formación de tuplas en los programas funcionales [Pettorossi y Proietti, 1996b]. En los programas lógicos, la ausencia de reglas de abstracción y algunas facilidades sintácticas de los lenguajes funcionales (como las construcciones `let` o `where`) no impide la adaptación de la estrategia en este contexto, ya que esta carencia se compensa sobradamente con reglas de reemplazamiento de objetivos, e incluso, bajo ciertas condiciones, con el propio mecanismo de unificación (que, en ocasiones, puede simular los efectos de una abstracción). Es más, mediante la formación de tuplas de predicados, es posible alterar la regla de computación de izquierda a derecha usada por Prolog e incluso reducir el grado de indeterminismo de los programas lógicos, como ocurre con la técnica automática de *control de compilación* que se basa en esta estrategia [Bruynooghe *et al.*, 1989].

El resto de estrategias usadas en la transformación de programas declarativos se utilizan muy a menudo para identificar definiciones eureka apropiadas que permitan automatizar las comentadas anteriormente. Este es el caso de las estrategias de *absorción de bucles* o *las definiciones implícitas* de Proietti y Pettorossi [1990, 1993] (en programación lógica) o el método de análisis de Chin [1993] (en programación funcional).

5.3.3 Otras estrategias

Bajo el nombre de *estrategias de generalización* se aglutinan normalmente un conjunto de métodos que, tanto en el caso lógico como en el funcional, pueden verse como particularizaciones o variantes de la composición y la formación de tuplas. Siguiendo [Pettorossi y Proietti, 1996b] comentamos brevemente tres variantes de la estrategia de generalización para el caso de los programas funcionales.

- **Generalización de expresiones a variables.** Dada una ecuación recursiva de la forma $f(x_1, \ldots, x_n) = e$, tal que la subexpresión s aparece en e, entonces:

1. introducimos la ecuación recursiva generalizada $g(x, y_1, \ldots, y_m) = e[x/s]$, donde x es una variable nueva, $e[x/s]$ denota a la expresión e donde se han sustituido todas las apariciones de la expresión s por la variable x, y $x, y_1, \ldots, y_m$ son las variables libres de $e[x/s]$. Obsérvese la estrecha conexión entre esta transformación y nuestra regla de abstracción basada en *lambda lifting*;
2. mediante pasos de desplegado y plegado buscamos una definición recursiva para $g(x, y_1, \ldots, y_m)$; y
3. finalmente expresamos la función $f(x_1, \ldots, x_n)$ en términos de $g(x, y_1, \ldots, y_m)$ plegando una o más instancias de $e[x/s]$.

- **Generalización de funciones a funciones por definición implícita.** Dada una ecuación recursiva de la forma $f(x_1, \ldots, x_n) = C[expr]$ en un programa P:

  1. introducimos la función g con aridad k, tal que existen una serie de expresiones $e_1, \ldots, e_k$, posiblemente con variables $x_1, \ldots, x_n$, tal que para todos los valores de las variables $x_1, \ldots, x_n$ tenemos que $C[g(e_1, \ldots, e_k)]$ es equivalente a $C[expr]$ con respecto al programa P;
  2. mediante pasos de desplegado y plegado buscamos una definición recursiva para $g(x_1, \ldots, x_k)$ y
  3. finalmente reemplazamos la ecuación $f(x_1, \ldots, x_n) = C[expr]$ por la nueva ecuación $f(x_1, \ldots, x_n) = C[g(e_1, \ldots, e_k)]$.

- **Abstracción *lambda*.** Consiste en reemplazar una expresión dada e_1 por $e_1[x/e_2]$ *where* $x = e_2$ o, equivalentemente, $(\lambda x.e_1[x/e_2])e_2$, donde x es una variable nueva y e_2 es una subexpresión de e_1. De nuevo observamos una gran similitud entre esta transformación y nuestras definiciones de la Sección 5.2.2, aunque ahora usamos una sintaxis totalmente funcional. La abstracción *lambda* es una forma de generalización que permite facilitar pasos de plegados en situaciones complicadas y puede ser adaptada para transformar programas funcionales de orden superior [Pettorossi y Proietti, 1996b].

Aparte de las técnicas de transformación basadas en reglas y estrategias, existen otras muchas técnicas que caen fuera del ámbito de este trabajo y entre las que podemos citar los métodos de especialización por evaluación parcial (que hemos comentado en las Secciones 3.6 y 4.5), la síntesis de programas o la transformación mediante esquemas (ver [Pettorossi y Proietti, 1996b] para un estudio detallado).

5.4 El sistema SYNTH

El sistema SYNTH es un sistema de transformación de programas que sigue la aproximación “reglas + estrategias” para optimizar programas escritos en *Curry*, un lenguaje multiparadigma moderno basado en *narrowing* necesario [Hanus *et al.*, 1995; Hanus (ed.), 1999].

El sistema SYNTH incorpora las reglas de transformación básicas que hemos visto en las secciones 3.4.2 (desplegado necesario), 4.4 (plegado T&S) y 5.2 (introducción/eliminación de definiciones, abstracción y reemplazamiento algebraico). El sistema incluye las estrategias de composición (en versiones parcial y completamente automática) y de formación de tuplas (de forma semi-automática). La aplicación puede obtenerse desde la dirección URL: `http://www.dsic.upv.es/users/elp/soft.html`.

5.4.1 Características de SYNTH

El sistema está escrito en SICStus Prolog v3.6 e incluye un analizador sintáctico para (un subconjunto de) *Curry*. La implementación completa consta de unas 680 cláusulas (2450 líneas de código). El transformador se expresa en 330 cláusulas (incluyendo el interfaz de usuario y el código necesario para manejar las *representaciones básicas*), el analizador sintáctico y otras utilidades se definen usando 190 cláusulas, el *narrowing* necesario se implementa con 90 cláusulas y los árboles definicionales se construyen por medio de 70 cláusulas. La implementación considera programas incondicionales mientras que las reglas condicionales son tratadas usando las funciones predefinidas `and`, `if_then_else`, y `case_of`, que se reducen usando reglas estándar (ver, e.g., [Moreno-Navarro y Rodríguez-Artalejo, 1992]). El sistema SYNTH no necesita una instalación previa. Se ejecuta bajo SICStus Prolog v3.x introduciendo:

```
?- consult(synth).
```

Una vez que el sistema está cargado, se muestra el siguiente menú de comandos que refleja todas las opciones disponibles en la aplicación.

```
************************************************************
*** load   --> Read R0 from a file                       ***
*** clean  --> Clean the system                          ***
*** intr   --> Definition introduction                   ***
*** elim   --> Definition elimination                    ***
*** list   --> List the rules of a program               ***
*** show   --> Show a definitional tree                  ***
*** unfold --> Lazy unfolding                            ***
*** fold   --> Fold two rules                            ***
*** abs    --> Abstraction Rule                          ***
*** acomp  --> Automatic Composition Strategy            ***
*** comp   --> Composition Strategy                      ***
*** tup    --> Tupling Strategy                          ***
*** exit   --> Leave the program                         ***
************************************************************
```

Recientemente hemos incorporado al sistema un interfaz gráfico escrito en Java (5000 líneas de código) que mejora notablemente su presentación e interacción con el usuario, como veremos a continuación. Para ejecutar la aplicación con este interfaz es necesario tener instalada la herramienta jdk1.1.7 e invocarla con la orden:

```
C:\SYNTH>java synth.
```

La pantalla general de SYNTH está dividida en las 4 zonas que muestra la Figura 5.1:

1. La *barra de herramientas* se presenta en la parte superior de la pantalla y permite seleccionar los menús de:
   (a) `File`, con opciones para cargar/grabar un programa, generar uno nuevo, limpiar la aplicación o salir de la misma;
   (b) `Edit`, que permite seleccionar texto para cortar, copiar o pegar en la ventana de edición, al tiempo que ofrece la posibilidad de realizar operaciones de análisis sintáctico (*parse*) y mostrar/editar cualquier programa de la secuencia de transformación o cualquier árbol definicional de un símbolo de función definido;
   (c) `Rules`, desde donde se pueden aplicar diversas reglas de transformación, como son la introducción de una definición, plegado y desplegado;
   (d) `Strategies`, para aplicar las estrategias de composición (de forma parcial o completamente automática) y de formación de tuplas;
   (e) `Options`, que permite modificar algunas características de visualización, como son el tipo de letra o el color; y finalmente
   (f) `Help`, donde se da información adicional sobre las el sistema y sus posibilidades.
2. La *ventana de edición* conforma la parte central de la pantalla y en ella es posible editar simultáneamente diferentes programas pertenecientes a una misma secuencia de transformación, como se aprecia en la Figura 5.5.
3. La *ventana de mensajes* aparece en la parte inferior de la pantalla y es utilizada por el sistema para mostrar información relativa a la operación que se realiza en cada momento de una sesión de trabajo, como puede ser el proceso de análisis sintáctico, la generación de árboles definicionales o la aplicación de una regla o estrategia de transformación.
4. Finalmente, la *relación de opciones disponibles* se muestra en la columna derecha de la pantalla. El conjunto de iconos que aparece en ella cambia dinámicamente conforme se desarrolla la sesión de trabajo y se corresponde con aquellas opciones que, de forma menos directa, también son seleccionables a través de los diferentes submenús de la barra de herramientas. Por ejemplo, el icono referente a la regla de desplegado únicamente se muestra cuando existe un programa cargado en el sistema y una regla marcada dentro del mismo.

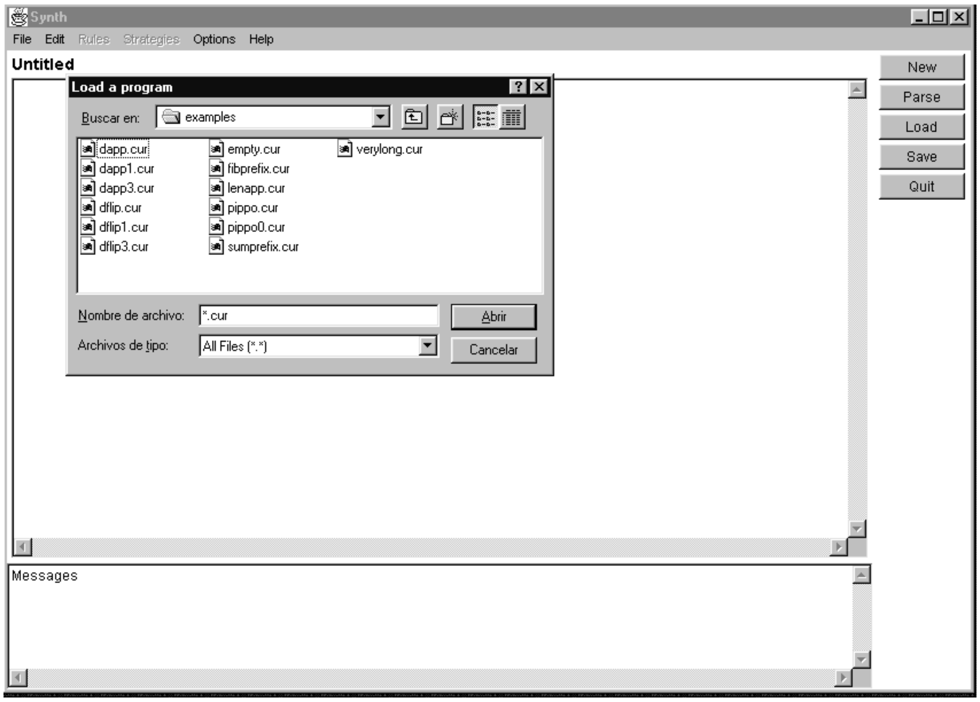


Figura 5.1: Menú de carga de un programa en el entorno SYNTH

En la figura 5.1 se muestra desplegado el menú de carga de un programa fuente que permite leer un fichero con un programa fuente inicial y grabarlo en la base de datos interna. Si la carga se hace con el comando `load` directamente desde el intérprete de Prolog, entonces los nombres de fichero deben ajustarse a la notación para términos de este lenguaje, i.e., se requieren comas simples cuando aparecen caracteres especiales en el nombre del fichero (e.g., debe teclearse `'dapp.cur'` en vez de `dapp.cur`).

La sintaxis del lenguaje se ajusta a la del lenguaje *Curry*. Cada programa consiste en un conjunto de declaraciones de tipos y funciones. Por ejemplo, la declaración de tipo

```
data treeInt = leaf Int | tree treeInt Int treeInt
```

introduce el tipo `treeInt` de árboles binarios cuyos nodos están etiquetados con enteros.

Una función se define mediante una lista de ecuaciones o reglas de definición. Por ejemplo, la declaración de función

```
append [] X = X
append (H:T) X = H:(append T X)
```

define la operación para la concatenación de dos listas.

Las principales diferencias con respecto a la sintaxis de *Curry*, tal y como se define en [Hanus (ed.), 1999], son:

- no se permiten ecuaciones condicionales;
- siguiendo la notación Prolog, las variables comienzan con una letra mayúscula y las funciones (constructoras y definidas) comienzan con una letra minúscula;
- las declaraciones de tipo para las funciones definidas no son tratadas por SYNTH, ya que el sistema no incluye un comprobador de tipos;
- no se permite la construcción `let`, aunque es posible realizar declaraciones locales por medio de la construcción `where`.

El sistema de transformación actual todavía no contempla aspectos avanzados del lenguaje *Curry*, como son el tratamiento del orden superior, la concurrencia o la residuación, aspectos todos ellos que se proponen como trabajo futuro.

Una vez que en el sistema tenemos cargados uno o más programas pertenecientes a una misma secuencia de transformación, es posible referirse a cualquiera de ellos escogiendo entre las opciones `First`, `Previous`, `Next`, `Last` o `Go to` que aparecen entre las ventanas de edición y de mensajes, como muestra la figura 5.5. Para un programa seleccionado, es posible pulsar cualquier icono de la columna derecha de opciones disponibles con el objetivo de editarlo, salvarlo, limpiarlo, transformarlo, analizarlo sintácticamente, mostrar alguno de sus árboles definicionales, etc. El sistema se cierra escogiendo la opción `quit`, que siempre está disponible.

En el siguiente apartado detallamos el uso de las opciones de transformación de los menús `Rules` y `Strategies` que nos permitirán optimizar un programa concreto.

5.4.2 Un ejemplo de transformación

A continuación realizamos un seguimiento paso a paso de una sesión de trabajo en la que transformamos por composición el ejemplo clásico y bien conocido de la función que concatena tres listas, `doubleAppend`, que nos ha servido para ilustrar numerosos aspectos de esta tesis (ver los Ejemplos 34, 39, 40, 41, 42 y 43).

Una vez que el programa `dapp.cur` ha sido cargado con la opción `load`, todas sus reglas aparecen numeradas en la ventana de edición para poder referirse a cada una de ellas de forma inequívoca. También al propio programa se le asigna el número 0 para identificarlo como el programa inicial en la secuencia de transformación, lo que queda reflejado en la parte superior de la ventana de edición como se ve en la figura 5.2. A partir de ese momento es posible aplicarle cualquier regla básica de transformación y construir una secuencia de transformación de forma interactiva en la que cada nuevo programa aparece etiquetado con un número creciente que identifica su posición en la secuencia de transformación (como se aprecia en la Figura 5.3 y sucesivas). También

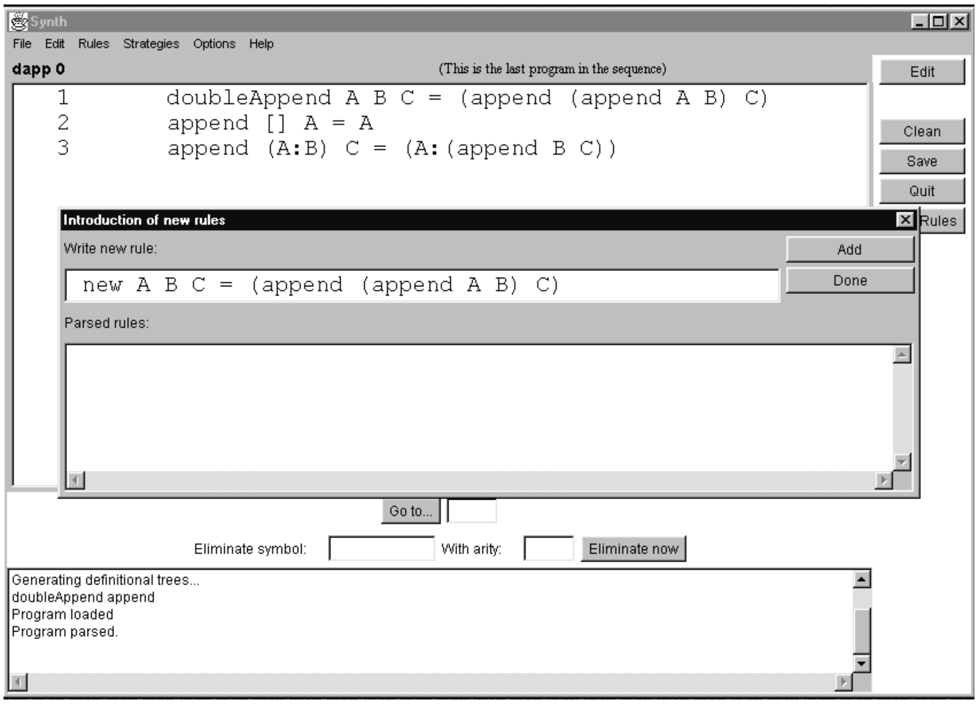


Figura 5.2: Introducción de una definición en el entorno SYNTH

es posible seleccionar la opción `acomp` en el menú de estrategias para desarrollar una heurística automática de composición, o las opciones `composition` y `tupling`, si se quiere aplicar las estrategias de composición y formación de tuplas, respectivamente, de forma semi-automática (con cierto grado de interacción por parte del usuario en el proceso de transformación). Actualmente estamos intentando extender el sistema para que pueda desarrollar la estrategia de formación de tuplas de forma completamente automática (partiendo de una extensión de los métodos de análisis de Chin [1993]).

A continuación pasamos a describir un proceso de transformación interactivo que comienza introduciendo una definición mediante la selección de la opción `New Rules`, como se ve en la Figura 5.2. Es posible introducir de una vez una serie de nuevas definiciones pulsando `Add` y terminar el proceso al seleccionar `Done` en la ventana abierta que solapa a la edición. Una vez que la nueva regla ha sido introducida, ésta sufre un proceso automático de análisis sintáctico que, si es superado con éxito, termina con la generación del correspondiente árbol definicional asociado al nuevo símbolo recién introducido.

A continuación procedemos a desplegar la nueva regla cuyo número de identificación es el 4. Para ello basta con marcar la regla que acabamos de introducir en

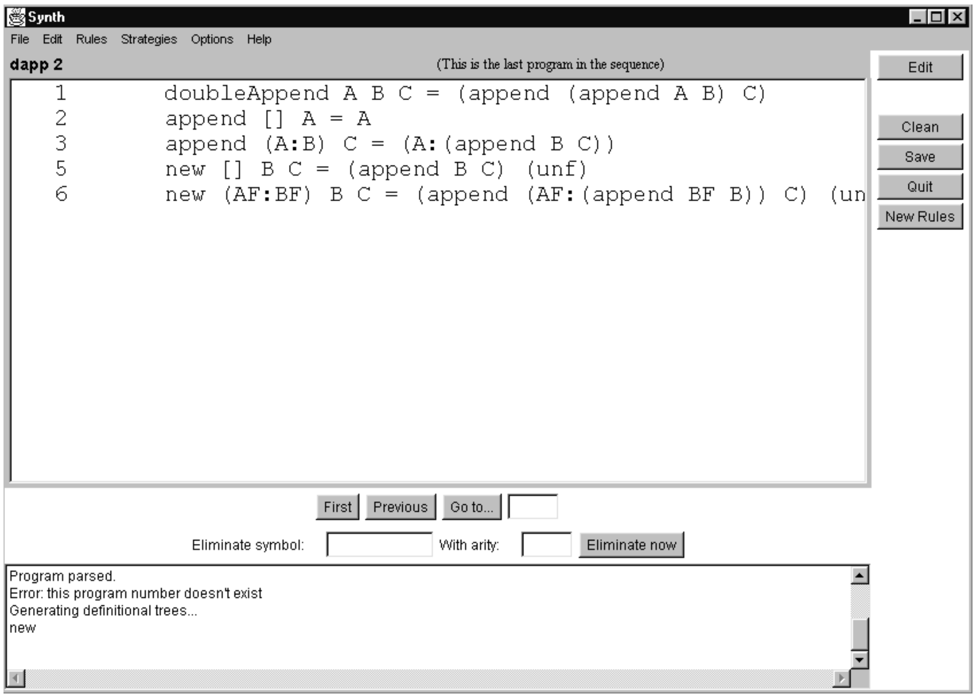


Figura 5.3: Desplegado de una regla en el entorno SYNTH

la ventana de edición y seguidamente seleccionar la opción `Unfold` en la columna de opciones disponibles o, alternativamente, en el menú de `Rules`. El sistema responde mostrando el nuevo programa transformado con número de identificación 2, tal y como se aprecia en la Figura 5.3. Obsérvese que las nuevas reglas desplegadas (numeradas con 5 y 6 respectivamente) aparecen también marcadas en su parte derecha con la etiqueta `(unf)`, lo que ayuda a detectar en todo momento cual es la transformación que las ha originado a partir del programa inmediatamente precedente en la secuencia de transformación. De esta forma, al movernos por los diferentes programas que componen dicha secuencia de transformación (mediante la selección de cualquiera de las opciones que aparecen entre las ventanas de adición y mensajes), podemos reconocer y reconstruir el proceso de transformación paso a paso, identificando cual es la transformación concreta que se ha aplicado en cada tramo. Cabe destacar que en todo momento están disponibles las opciones de editar y grabar el programa transformado antes de proceder a una nueva transformación.

Si a continuación realizamos dos nuevas operaciones de desplegado sobre las reglas 5 y 6, obtenemos el programa **dapp 4** que se muestra en la Figura 5.4. En este momento se aprecia la presencia de una regularidad (o instancia del cuerpo de la

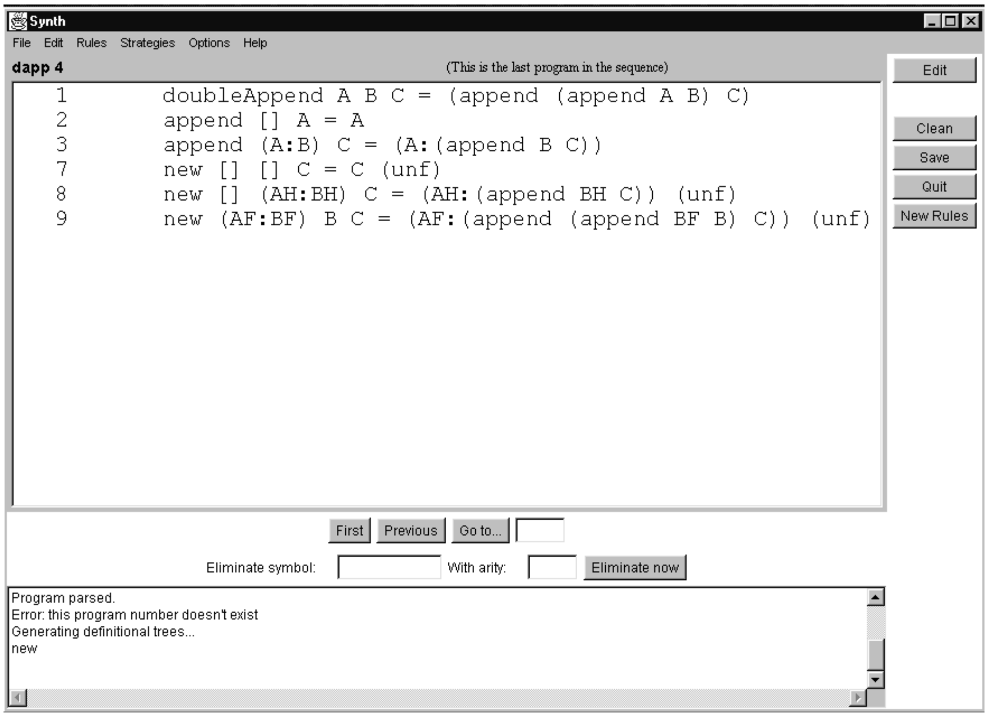


Figura 5.4: Programa obtenido tras múltiples desplegados en el entorno Synth

regla de definición 4) en la parte derecha de la regla 9, lo que abilita un paso de plegado posterior.

En la Figura 5.5 se muestra el aspecto que presenta el sistema en el momento que se intenta aplicar la opción `Fold Rules` sobre las reglas 9 y 4. Ya que las reglas plegable (9) y plegante (4) no pertenecen al mismo programa, es necesario editar los programas **dapp 4** y **dapp 1** simultáneamente para marcar ambas reglas y proceder a la transformación de plegado. La posibilidad de dividir la ventana de edición en dos áreas independientes está contemplada en el menú de `Options`. Obsérvese como, en la ventana inferior de la Figura 5.5, el sistema muestra el carácter plegable o plegante de cada una de las reglas involucradas en la transformación. De esta forma, el usuario puede intercambiarlas si así lo desea, cancelar la operación o finalmente llevarla a cabo. Una vez que se ha realizado la operación de plegado, es posible reintentar su aplicación para plegar la regla 1 con respecto a la regla 4 con lo que se obtiene un programa final con un aspecto similar al que se muestra en la Figura 5.6 (con identificador **dapp 6**). Es interesante destacar la etiqueta `(fold)` con que aparecen marcadas las últimas reglas plegadas en el programa transformado.

Nótese que la secuencia de transformación (seguida de forma totalmente interac-

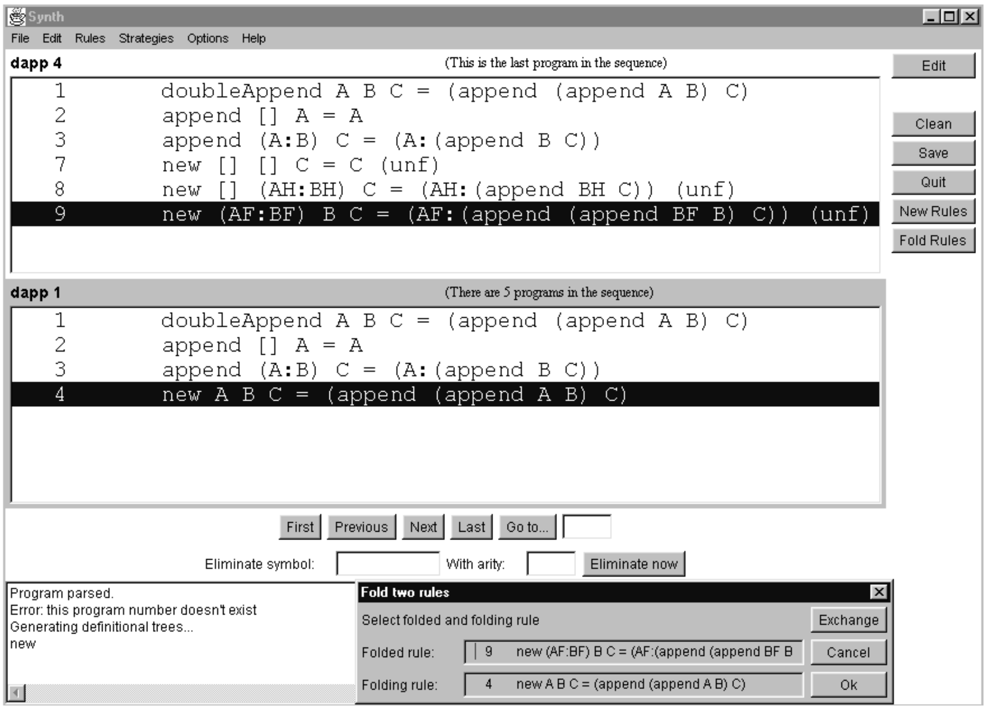


Figura 5.5: Operación de plegado en el entorno SYNTH

tiva) que acabamos de ver, devuelve la definición recursiva deseada para `new` (reglas R_6, R_7 y R_8). De esta forma, el programa transformado supone una mejora (tanto en tiempo como en espacio) con respecto al programa original **dapp 0**.

Aunque en este ejemplo no ha sido necesario aplicar las reglas de eliminación de una definición, abstracción y reemplazamiento algebraico, todas estas transformaciones pueden llevarse a cabo en el entorno SYNTH de la siguiente manera:

- Para eliminar una definición es suficiente con dar al sistema el nombre y la aridad del símbolo a eliminar en la barra que aparece justo encima de la ventana de mensajes. La operación se aplicará satisfactoriamente solamente cuando se detecte que el símbolo a eliminar no aparece en ninguna regla diferente a las que lo definen, tal y como hemos indicado en la Sección 5.2.1. También es importante destacar el hecho comentado anteriormente en este capítulo de que, una vez que se ha eliminado un símbolo en la secuencia de transformación, la opción de plegado deja de ser accesible en la columna derecha de opciones disponibles.
- Para realizar una operación de abstracción sobre una regla previamente marcada por el usuario, el sistema actúa en dos fases (ver de nuevo la Sección 5.2.2):
  1. en primer lugar, se da un un paso de introducción de una definición en el que la

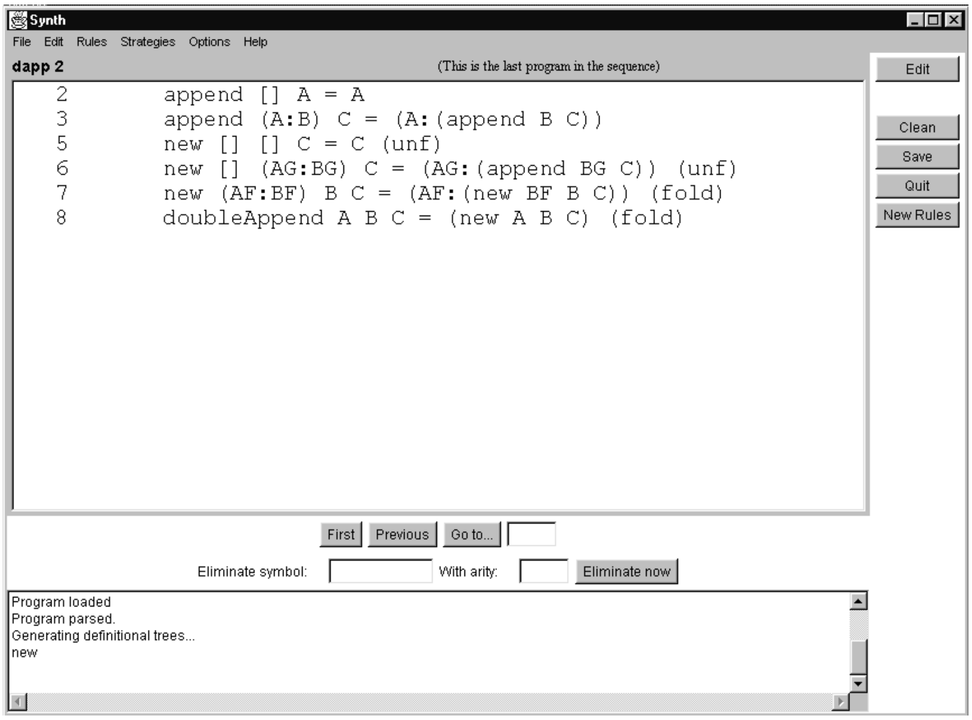


Figura 5.6: Programa final transformado en el entorno SYNTH

parte derecha de la regla de definición coincide con la parte derecha de la regla a abstraer, pero con la particularidad de que las expresiones que se pretenden abstraer son sustituidas por variables nuevas que aparecen empaquetadas en una tupla en la parte izquierda de la regla recién introducida;

2. a continuación el sistema aplica de forma automática un paso de plegado donde la regla plegante coincide con la introducida en el paso anterior, mientras que la regla plegable no es otra que la regla a abstraer.

De esta forma se consigue el efecto de abstracción perseguido en la Definición 5.2.4, donde una regla es sustituida por un par de nuevas reglas equivalentes.

- El reemplazamiento algebraico viene incorporado al sistema de forma completamente interactiva: es el usuario quien directamente realiza (en la ventana de edición) las modificaciones oportunas sobre las partes derechas de las reglas a reemplazar, seleccionando la opción `Edit`.

Para terminar este apartado, incluimos unos breves comentarios acerca de la implementación de las estrategias de transformación en SYNTH. Las diferencias entre los métodos semi-automático y completamente automático que incluye el sistema para realizar, por ejemplo, la estrategia de composición, consisten básicamente en lo si-

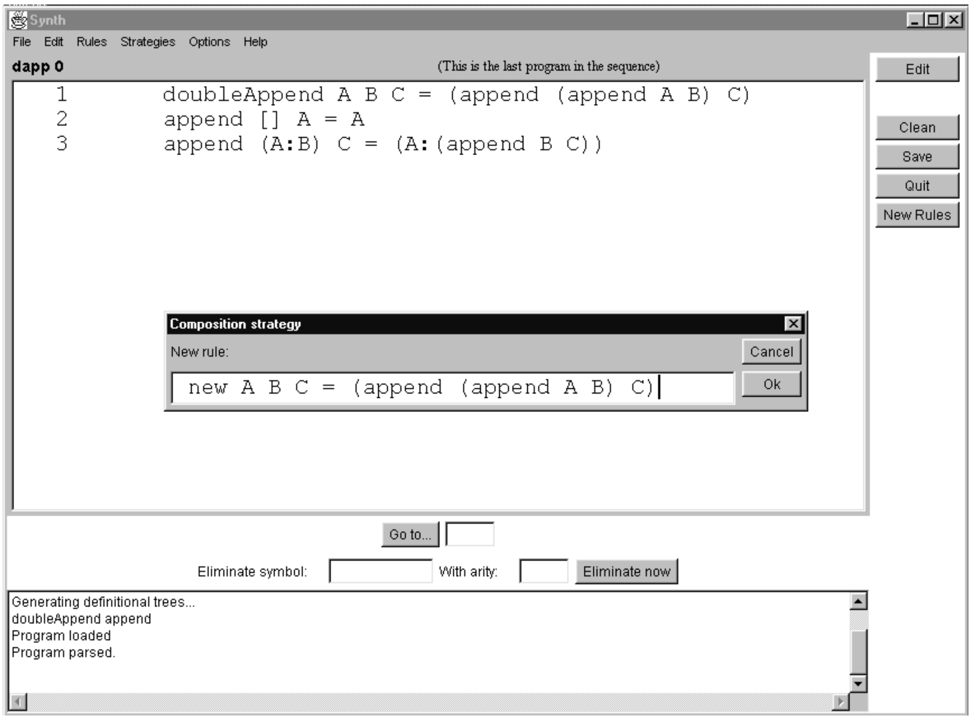


Figura 5.7: Aplicación de la estrategia de composición en el entorno SYNTH

guiente. En el primer caso (opción `composition`), el usuario debe indicar cuál es la regla en la que aparecen llamadas anidadas e introducir la nueva regla de definición apropiada a partir de la cual se pone en marcha la estrategia. La segunda alternativa (opción `acomp`) es completamente automática: es el propio transformador quien localiza las llamadas anidadas y se introduce una nueva definición para que el proceso se active. A partir de aquí, el proceso es completamente automático e idéntico en ambos casos. SYNTH construye, mediante pasos de desplegado, un árbol de búsqueda incompleto y busca una rama con regularidades donde pueda obtenerse una definición recursiva de la nueva función tras aplicar un paso final de plegado. Si se encuentra dicha rama en una profundidad finita (limitando la profundidad del árbol al imponer una cota máxima de desplegados o bien aplicando algún tipo de test de *embedding* para garantizar la terminación del proceso), entonces la regla original se reemplaza por la correspondiente a dicha rama; en otro caso, se devuelve el conjunto de reglas desplegadas asociadas a todas las ramas no falladas.

En la Figura 5.7 se muestra la ventana que presenta el sistema para introducir la regla de definición inicial a partir de la cual es posible aplicar la estrategia de composición (opción de `composition`) de forma semi-automática sobre el programa

Programas	Rw_1	RT_1	Rw_2	RT_2	%
`doubleappend`	3	1.77	6	1.63	8%
`sumprefix`	4	3.59	6	3.48	3%
`lengthapp`	5	1.61	7	1.51	6%
`doubleflip`	3	0.95	5	0.7	26%
`fibonacci`	3	2.21	5	0.18	92%
`factlist`	4	1.84	6	0.75	59%
`average`	5	1.52	7	0.87	43%
`hanoi`	4	2.08	8	0.12	94%

Tabla 5.1: Evaluación del sistema

inicial **dapp 0**. Por defecto, se asume que el máximo número de desplegados (*depth*) aplicados a la nueva regla de definición es 2, aunque este parámetro puede modificarse apropiadamente. Como puede verse en la Figura 5.6, el resultado final del proceso (semi-automático) de composición coincide con el programa obtenido tras la aplicación de una serie de transformaciones básicas guiadas manualmente. Una tercera forma alternativa de conseguir el mismo resultado final consiste en escoger la opción `acomp` para aplicar la estrategia de composición de forma completamente automática, con la ventaja añadida en esta ocasión de que no es necesario siquiera especificar la regla de definición inicial.

5.4.3 Evaluación del sistema

En [Alpuente *et al.*, 1999a,b] hemos realizado una serie de experimentos con el sistema SYNTH que corroboran el interés práctico de nuestra propuesta. Concretamente, hemos evaluado las estrategias de composición y formación de tuplas sobre diferentes programas. Todos los programas han sido ejecutados usando TasteCurry, un intérprete de (un subconjunto de) *Curry* de libre disposición [Hanus *et al.*, 1995].

En la Tabla 5.1 se pueden ver algunos de los resultados obtenidos. Las primeras dos columnas muestran el número de reglas (Rw_1) y el tiempo de ejecución (RT_1) de los programas originales. Las siguientes dos columnas muestran el número de reglas (Rw_2) y el tiempo de ejecución (RT_2) para los programas transformados, mientras que en la última columna (%) se puede ver el porcentaje en el que se ha reducido el tiempo de ejecución respecto al programa original. Los tiempos están expresados en segundos y representan la media de 10 ejecuciones. Podemos observar que nuestra estrategia de composición automática es capaz de realizar optimizaciones apreciables en los cuatro primeros programas (`doubleappend`, `sumprefix`, `lengthapp` y `doubleflip`, cuyos códigos pueden verse en los Ejemplos 40 y 49 y en la Tabla 5.2, respectivamente). Estos programas son ejemplos típicos usados en la literatura para ilustrar las ventajas de las técnicas de composición [Wadler, 1990]. Por otro lado, usando la estrategia de

PROGRAMA ORIGINAL	PROGRAMA FINAL
lengthapp	**Por composición**

`lengthapp(L1,L2) → len(app(L1,L2))`
`len(nil) → 0`
`len(H:T) → s(len(T))`
`app(nil,L) → L`
`app(H:T,L) → H:app(T,L)`
	`lengthapp(L1,L2) → new(L1,L2)`
	`new(nil,L) → len(L)`
	`new(H:T,L) → s(new(T,L))`

doubleflip	**Por composición**

`doubleflip(T) → f(f(T))`
`f(leaf(N)) → leaf(N)`
`f(tree(L,N,R) → tree(f(R),N,f(L))`
	`doubleflip(T) → n(T)`
	`n(leaf(N)) → leaf(N)`
	`n(tree(L,N,R) → tree(n(L),N,n(R))`

factlist	**Por formación de tuplas**

`flist(0) → nil`
`flist(s(N)) → fact(s(N)):flist(N)`
`fact(0) → 1`
`fact(s(N)) → s(N)*fact(N)`
	`flist(0) → nil`
	`flist(s(N)) → U:V where`
	`⟨U,V⟩ = n(N)`
	`n(0) → ⟨s(0),nil⟩`
	`n(s(N)) → ⟨s(s(N))*U,U:V⟩ where`
	`⟨U,V⟩ = n(N)`

average	**Por formación de tuplas**

`average(L) → sum(L)/length(L)`
`sum(nil) → 0`
`sum(H:T) → H+sum(T)`
`length(nil) → 0`
`length(H:T) → s(length(T))`
	`average(L) → U/V where`
	`⟨U,V⟩ = new(L)`
	`new(nil) → ⟨0,0⟩`
	`new(H:T) → ⟨H+U,s(V)⟩ where`
	`⟨U,V⟩ = new(T)`

Tabla 5.2: Ejemplos de aplicación de estrategias de transformación

formación de tuplas es posible obtener mejoras aún más notables, como las obtenidas en los ejemplos `fibonacci` (Ejemplo 50), `hanoi` (Ejemplo 51), `factlist` y `average` (Tabla 5.2), aunque, en este caso, el proceso de optimización en SYNTH debe ser guiado por el usuario.

5.5 Conclusiones

En este capítulo hemos completado el conjunto de reglas de transformación para conformar un sistema de transformación efectivo para programas inductivamente secuenciales. Además de las básicas de desplegado necesario (Sección 3.4.2) y plegado T&S (Sección 4.4) estudiadas en capítulos anteriores, el sistema incorpora un conjunto estándar de reglas de transformación en el estilo de la propuesta inicial de Burstall y Darlington [1977], excepto la regla de instanciación que, en nuestro marco, queda subsumida de forma natural por la regla de desplegado, al tiempo que se preserva la semántica de los programas lógico-funcionales con una semántica operacional basada en *narrowing* necesario. Las nuevas reglas de transformación introducidas en este capítulo son:

1. **Introducción/eliminación de una definición**. Como su nombre indica, estas reglas permiten introducir o eliminar reglas que definen nuevos símbolos de función (llamados también eurekas) bajo una serie de condiciones. La introducción de una nueva definición normalmente se considera como el primer paso de una secuencia de transformación, mientras que la eliminación de un símbolo suele hacerse al final de la secuencia, cuando la nueva función deje de ser necesaria. El uso de eurekas es crucial para realizar un renombramiento de (alguna parte) de una expresión dada, de tal forma que sea posible derivar una versión más eficiente de la nueva función mediante la aplicación de otras reglas de transformación. La aplicación de la reglas de introducción/eliminación de definiciones supone una alteración de la signatura de los programas en una secuencia de transformación ya que se incrementa/reduce el conjunto de símbolos de función definidos. Por esta razón, los resultados de corrección y completitud del sistema global se dan con respecto a aquellos objetivos que consideran la misma signatura que el programa inicial, al tiempo que no se permiten pasos de plegado después de un paso de eliminación de una definición.

2. **Reglas de abstracción**. Con estas reglas es posible incrementar la potencia del sistema de transformación, permitiendo realizar la estrategia de formación de tuplas. Una abstracción simple consiste esencialmente en el reemplazamiento de ocurrencias de una misma expresión e en la parte derecha de una regla R por una variable nueva X, añadiendo la "declaración local" $X = e$ dentro de una expresión $where$ en R. De esta forma es suficiente con evaluar la expresión e una sola vez, ya que su valor asociado queda vinculado a una variable que propaga esta información a todos los contextos que la requieran, sin necesidad de tener que realizar nuevas reevaluaciones. Las variables locales declaradas en una construcción $where$ suponen la aparición de variables extra en las partes derechas de las reglas transformadas, lo que está prohibido en los lenguajes integrados. Sin embargo, hemos resuelto este problema usando las técnicas estándar de eliminación de expresiones "lambda " (*lambda lifting*) de la programación funcional. Nuestra regla de abstracción generaliza y extiende la regla de abstracción simple al permitir abstraer varias expresiones de una sola vez usando el operador tupla $\langle\ \rangle$.

3. **Reemplazamiento algebraico**. Esta última regla de transformación permite reordenar ciertas porciones de la parte derecha de una regla con el objetivo de

permitir la posterior aplicación de otros pasos transformación. La regla se inspira en las así llamadas "leyes para primitivas" de Burstall y Darlington [1977] que, básicamente, permiten la transformación de una ecuación en otra aplicando propiedades tales como la commutatividad, asociatividad o distributividad de ciertos operadores considerados como primitivos (e.g., $+$, $\times$, `if_then_else`).

Las principales propiedades del conjunto de reglas anterior, tal y como se recogen en el Teorema 5.2.6, pueden resumirse como sigue. Toda secuencia de transformación que parte de un programa inductivamente secuencial (donde no se efectúan pasos de plegado después de la eliminación de una definición) está compuesta por programas que, además de pertenecer a la misma clase, devuelven exactamente los mismos valores y respuestas computadas para todo objetivo sin símbolos de función nuevos.

Aparte de las cuestiones de corrección, este conjunto de reglas es suficiente para implementar en nuestro contexto la mayoría de las estrategias de transformación más potentes que se conocen, como son las de composición y formación de tuplas. Estas estrategias de transformación pueden verse como un conjunto de metareglas que se usan para conducir el proceso de aplicación de reglas de transformación con el objetivo de optimizar programas. Estos métodos se suelen formular de forma algorítmica como una serie de heurísticas que describen la secuencia de cambios que sufre un programa mientras es transformado, consiguiéndose resultados como la eliminación de variables innecesarias , evitar la construcción de estructuras de datos intermedias o el recorrido múltiple de una misma estructura de datos, etc.

El algoritmo de formación de tuplas es similar al de composición pero con la particularidad de que los pasos de plegado van siempre precedidos por abstracciones que reordenan las reglas a plegar para posibilitar la transformación. La razón está en el hecho de que las llamadas a plegar no aparecen anidadas ni en las reglas originales ni en las transformadas, lo que implica la necesidad de fundirlas y "empaquetarlas" en una tupla mediante la aplicación de la regla de abstracción. Solamente así es posible plegar las tuplas de expresiones y sustituirlas por la llamada a función nueva.

El sistema SYNTH es una aplicación implementada en Prolog que permite optimizar programas escritos en (un subconjunto de) *Curry*. El sistema implementa las reglas de transformación básicas que hemos introducido en esta tesis e incluye las estrategias de composición (en versiones parcial y completamente automática) y de formación de tuplas (de forma semi-automática). Las diferencias entre los métodos semi-automático y completamente automático de aplicación de una estrategia consisten básicamente en la posibilidad de introducir interactiva o automáticamente el eureka apropiado para activar el proceso de transformación. A partir de aquí, el proceso es completamente automático e idéntico en ambos casos.

Para finalizar, hemos realizado una serie de experimentos con el sistema SYNTH que corroboran el interés práctico de nuestra propuesta. Concretamente, hemos aplicado las estrategias de composición y formación de tuplas sobre diferentes programas ejecutados usando el sistema TasteCurry.

Capítulo 6

Conclusiones y trabajo futuro

En esta tesis investigamos las bases de un sistema de transformación de programas lógico-funcionales basado en las técnicas de plegado/desplegado que permite obtener algunas de las optimizaciones más potentes que se conocen sobre programas declarativos. Las principales aportaciones se pueden clasificar tal y como sigue.

A nivel teórico, hemos estudiado el comportamiento de las transformaciones elementales de plegado/desplegado sobre diferentes clases de programas que se ejecutan mediante distintas variantes del *narrowing*. En particular, hemos presentado cuatro versiones (y dos refinamientos más en las Secciones 3.5 y 3.6) de la regla de desplegado y hemos estudiado su potencia transformacional y las condiciones para su corrección. Por otra parte, hemos presentado y comparado tres formulaciones de la regla de plegado en cuanto a nivel de reversibilidad y potencial de optimización. Para todas ellas, hemos analizado las condiciones de aplicabilidad que garantizan las propiedades de corrección y completitud, así como el preservar la estructura original del programa transformado. También hemos mostrado algunas de las aplicaciones más relevantes, como son la generación de semánticas por desplegado y la especialización de programas de forma alternativa a las técnicas clásicas de evaluación parcial con renombramiento. Finalmente, hemos introducido y adaptado a la plataforma integrada las reglas para la introducción/eliminación de definiciones, abstracción y reemplazamiento algebraico. Con ello hemos cubierto el objetivo de construir un sistema completo de transformación de programas inductivamente secuenciales con una semántica operacional basada en *narrowing* necesario.

A nivel práctico, hemos demostrado que el conjunto de reglas anterior es capaz de producir las optimizaciones de composición y formación de tuplas en el marco integrado. Finalmente hemos implementado las reglas y estrategias propuestas en la herramienta experimental SYNTH. En este entorno, el usuario puede intervenir en el proceso de transformación o puede optar por la aplicación de las heurísticas presentadas de forma completamente automática (composición) o semi-automática (formación de tuplas). La utilidad de esta propuesta ha sido demostrada mediante distintos ejemplos de transformación que culminan en las optimizaciones perseguidas.

Son varias las líneas de trabajo abiertas por esta tesis. La formalización de técnicas basadas en la semántica para la transformación de programas lógico-funcionales es un campo de investigación incipiente y atractivo. Tenemos previsto extender el trabajo desarrollado en esta tesis teniendo en cuenta las siguientes ideas, que son argumento de varias tesis doctorales estrechamente relacionadas con la nuestra y actualmente en desarrollo en el grupo de investigación ELP:

- Ampliación del sistema de transformación para poder tratar con programas escritos en *Curry* en los que se contemplen características avanzadas del lenguaje, como son el tratamiento de tipos, clases, orden superior y residuación. Para ello es necesario extender las reglas de transformación en una línea similar a como se hace en [Sands, 1996], donde se consideran programas funcionales de orden superior, y [Albert *et al.*, 1999], donde se considera un marco de evaluación parcial de programas lógico-funcionales basado en *narrowing* necesario y residuación.
- Extensión del sistema de transformación para poder trabajar con programas ortogonales. Con el objetivo de ampliar la gama de programas a optimizar, es preciso retomar la estrategia de *narrowing* perezoso como base para la implementación de la regla de desplegado. A pesar de que esta regla no preserva la estructura ortogonal de los programas, pensamos que sería posible realizar un postproceso de transformación que incluyera un análisis de reglas redundantes capaz de detectar qué reglas pueden ser eliminadas del programa desplegado sin perder corrección ni completitud, de forma que sea posible en algunos casos restituir la ortogonalidad del programa transformado y continuar la secuencia de transformación de forma segura. Una fuente de inspiración para conseguir este objetivo la encontramos en [Hanus y Steiner, 1999], donde los autores usan una extension de los sistemas de inferencia de tipos llamados "sistemas de tipos y efectos" (*type and effect systems*) que consisten en añadir a los tipos asociados a las funciones de un programa una información adicional (los efectos) para permitir la inferencia de determinadas propiedades (como es el caso del carácter indeterminista de las funciones en [Hanus y Steiner, 1999]). En nuestro contexto, pensamos que a través de una adecuada definición de la noción de efectos en relación con el uso de variables lógicas, es posible identificar aquellas reglas desplegadas redundantes que perjudican la ortogonalidad de los programas transformados.
- La línea de trabajo más interesante consiste en intentar automatizar totalmente la estrategia de formación de tuplas. La idea es partir de los análisis de [Chin, 1993] que permiten derivar eurekas en un contexto funcional puro y que se basan esencialmente en un método similar a las técnicas de evaluación parcial: se cons-

truyen árboles de desplegado y se identifican regularidades que de alguna manera se corresponden con los eurekas deseados. Pensamos que en el marco integrado se puede explotar de forma eficaz la regla de desplegado basada en *narrowing* (que incorpora automáticamente la regla de instanciación) tanto para identificar eurekas como para desarrollar la propia estrategia de formación de tuplas de forma totalmente automática.

Bibliografía

Aït-Kaci H. y Nasr R., 1989. Integrating Logic and Functional Programming. *Lisp and Symbolic Computation*, 2(1):51–89.

Albert E., Alpuente M., Falaschi M., Julián P. y Vidal G., 1998a. Improving Control in Functional Logic Program Specialization. En G. Levi, ed., *Proc. of Static Analysis Symposium, SAS'98*, págs. 262–277. Springer LNCS 1503.

Albert E., Alpuente M., Falaschi M., Julián P. y Vidal G., 1998b. Improving Control in Functional Logic Program Specialization. Informe Técnico DSIC-II/37/97, UPV. Accesible en URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Albert E., Alpuente M., Hanus M. y Vidal G., 1999. A Partial Evaluation Framework for Curry Programs. En *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning, LPAR'99*, págs. 376–395. Springer LNAI 1705.

Alpuente M., Falaschi M., Ferri C., Moreno G. y Vidal G., 1999a. Un Sistema de Transformación para Programas Multiparadigma. *Revista Iberoamericana de Inteligencia Artificial*, X/99(8):27–35.

Alpuente M., Falaschi M., Ferri C., Moreno G., Vidal G. y Ziliotto I., 1999b. The Transformation System SYNTH. Informe Técnico DSIC-II/16/99, UPV. Accesible en URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Alpuente M., Falaschi M., Julián P. y Vidal G., 1997a. Specialization of Lazy Functional Logic Programs. En *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, Sigplan Notices 32(12), págs. 151–162. ACM Press, New York.

Alpuente M., Falaschi M., Julián P. y Vidal G., 1999c. Técnicas de Evaluación Parcial Perezosa en Programas Uniformes. Informe Técnico DSIC-II/18/99, UPV.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 1997b. Plegado/Desplegado usando Narrowing Condicional. En *Proc. of the 1997 Joint Conf. on Declarative Programming, Appia-Gulp-Prode'97, Grado (Italy)*. G.R.U. Logic Programming, pp. 347-358.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 1997c. Safe Folding/Unfolding with Conditional Narrowing. En H.H. M. Hanus y K. Meinke, eds., *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, págs. 1–15. Springer LNCS 1298.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 1997d. Safe Folding/Unfolding with Conditional Narrowing. Informe Técnico DSIC-II/3/97, UPV.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 1999d. Transformation-based Strategies for Lazy Functional Logic Programs. Informe Técnico DSIC-II/23/99, UPV.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 1999e. A Transformation System for Lazy Functional Logic Programs. En A. Middeldorp y T. Sato, eds., *Proc. of the 4th Fuji International Symposyum on Functional and Logic Programming, FLOPS'99, Tsukuba (Japan)*, págs. 147–162. Springer LNCS 1722.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 2000a. "Rules + Strategies" for Transforming Lazy Functional Logic Programs. Informe Técnico DSIC-II/2000, UPV. Sometido para publicación en revista.

Alpuente M., Falaschi M., Moreno G. y Vidal G., 2000b. Using Partial Evaluation to Automate Program Composition. Informe Técnico DSIC-II/2000, UPV. Sometido al 26th Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM'2000.

Alpuente M., Falaschi M. y Vidal G., 1996a. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. *Theoretical Computer Science*, 165(1):97–131.

Alpuente M., Falaschi M. y Vidal G., 1996b. Narrowing-driven Partial Evaluation of Functional Logic Programs. En H.R. Nielson, ed., *Proc. of the 6th European Symp. on Programming, ESOP'96*, págs. 45–61. Springer LNCS 1058.

Alpuente M., Falaschi M. y Vidal G., 1998a. Partial Evaluation of Functional Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844.

Alpuente M., Falaschi M. y Vidal G., 1998b. A Unifying View of Functional and Logic Program Specialization. *ACM Computing Surveys*, 30(3es):9es.

Alpuente M., Hanus M., Lucas S. y Vidal G., 1999f. Specialization of Functional Logic Programs Based on Needed Narrowing. Informe Técnico 99-4, RWTH Aachen. Accesible en URL: `http://www.dsic.upv.es/users/elp/papers.html`.

Alpuente M., Hanus M., Lucas S. y Vidal G., 1999g. Specialization of Inductively Sequential Functional Logic Programs. En P. Lee, ed., *Proc. of 1999 International Conference on Functional Programming, ICFP'99, Paris (France)*. ACM, New York.

Amtoft T., 1992. Unfold/fold Transformations Preserving Computed Answers. En M. Bruynooghe y M. Wirsing, eds., *Proc. of PLILP'92, Leuven (Belgium)*, págs. 187–201. Springer LNCS 631.

Antoy S., 1992. Definitional Trees. En *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, págs. 143–157. Springer LNCS 632.

Antoy S., 1997. Optimal Non-Deterministic Functional Logic Computations. En *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, págs. 16–30. Springer LNCS 1298.

Antoy S., Echahed R. y Hanus M., 1993. A Needed Narrowing Strategy. Technical report MPI-I-93-243, Max-Planck-Institut für Informatik, Saarbrücken.

Antoy S., Echahed R. y Hanus M., 1994. A Needed Narrowing Strategy. En *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, págs. 268–279. ACM Press, New York.

Antoy S., Echahed R. y Hanus M., 1997. Parallel Evaluation Strategies for Functional Logic Languages. En *Proc. of the Fourteenth International Conference on Logic Programming, ICLP'97*, págs. 138–152. MIT Press, Cambridge, Mass.

Aravindan C. y Dung P., 1995. On the correctness of unfold/fold transformations of normal and extended logic programs. *Journal of Logic Programming*, págs. 24(3):201–217.

Baader F. y Nipkow T., 1998. *Term Rewriting and All That*. Cambridge University Press.

Bellia M. y Levi G., 1986. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236.

Bert D. y Echahed R., 1986. Design and implementation of a generic, logic and functional programming language. En *Proc. of First European Symp. on Programming, ESOP'86*, págs. 119–132. Springer LNCS 213.

Bert D. y Echahed R., 1994. Integrating Disequations in the Algebraic and Logic Programming Language LPG. En *Proc. of the ICLP'94 Post-Conference Workshop on Integration of Declarative Paradigms*. MPI-I-94-224, Saarbrucken.

Bockmayr A. y Werner A., 1995. LSE Narrowing for Decreasing Conditional Term Rewrite Systems. En *Conditional Term Rewriting Systems, CTRS'94, Jerusalem*. Springer LNCS 968.

Bosco P., Giovannetti E. y Moiso C., 1988. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23.

Bossi A. y Cocco N., 1993. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16:47–87.

Bossi A., Cocco N. y Dulli S., 1990. A Method for Specializing Logic Programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302.

Bossi A., Cocco N. y Etalle S., 1992. On Safe Folding. En M. Bruynooghe y M. Wirsing, eds., *Proc. of PLILP'92, Leuven (Belgium)*, págs. 172–186. Springer LNCS 631.

Bossi A., Cocco N. y Etalle S., 1995. Simultaneous replacement in normal programs. Technical Report CS-R9357, CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands.

Bossi A. y Etalle S., 1994. Transforming Acyclic Programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096.

Bossi A., Gabbrielli M., Levi G. y Martelli M., 1994. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19-20:149–197.

Bossi (ed.) A., 1999. Special Issue of Synthesis, Transformation and Analysis of Logic Programs. *Journal of Logic Programming*, 41(2-3).

Boulanger D. y Bruynooghe M., 1993. Deriving unfold/fold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, págs. 495–521.

Bruynooghe M., De Schreye D. y Krekels B., 1989. Compiling Control. *Journal of Logic Programming*, 6(1&2):135–162.

Burstall R. y Darlington J., 1977. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67.

Caballero-Roldán R., López-Fraguas F. y Sánchez-Hernández J., 1997. User's manual for Toy. Informe Técnico SIP-5797, UCM, Madrid (Spain).

Cheong P. y Fribourg L., 1992. A survey of the implementations of narrowing. En J. Darlington y R. Dietrich, eds., *Declarative Programming. Workshops in Computing*, págs. 177–187. Springer-Verlag and BCS.

Chin W., 1993. Towards an Automated Tupling Strategy. En *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, 1993*, págs. 119–132. ACM, New York.

Clark K. y Sickel S., 1977. Predicate Logic: A Calculus for Deriving Programs. En *Proc. of 5th International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, USA*, págs. 419–420.

Courcelle B., 1990. Recursive Applicative Program Schemes. En J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, volumen B: Formal Models and Semantics, págs. 459–492. Elsevier, Amsterdam.

Darlington J., 1972. *A Semantic Approach to Automatic Program Improvement*. Tesis Doctoral, Department of Artificial Intelligence, Edinburgh University,Edinburg, UK.

Darlington J., 1982. Program Transformation. En J. Darlington, P. Henderson y D.A. Turner, eds., *Functional Programming and its Applications*, págs. 193–215. Cambridge University Press.

Darlington J. y Guo Y., 1989. Narrowing and Unification in Functional Programming: an evaluation mechanism for absolute set abstraction. En *Proc. of the Conf. on Rewrite Techniques and Applications, RTA'89*, págs. 92–108. Springer LNCS 355.

Darlington J. y Pull H., 1988. A Program Development Methodology Based on a Unified Approach to Execution and Transformation. En D. Bjørner, A. Ershov y N. Jones, eds., *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, págs. 117–131. North-Holland, Amsterdam.

Debray S.K., 1988. Unfold/fold transformations and loop optimization of logic programs. En *Proc. of SIGPLAN'88 Conf. on Programming Language Design and Implementation*.

DeGroot D. y Lindstrom G., eds., 1986. *Logic Programming, Functions, Relations and Equations*. Prentice Hall, Englewood Cliffs, NJ.

Dershowitz N., 1987. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115.

Dershowitz N. y Jouannaud J.P., 1990. Rewrite Systems. En J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, volumen B: Formal Models and Semantics, págs. 243–320. Elsevier, Amsterdam.

Dershowitz N. y Plaisted A., 1988. Equational Programming. *Machine Intelligence*, 11:21–56.

Dershowitz N. y Reddy U., 1993. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494.

Durand I., 1994. Bounded, Strongly Sequential and Forward Branching Term Rewriting Systems. *Journal of Symbolic Computation*, págs. 18:319–352.

Echahed R., 1988. On completeness of narrowing strategies. En *Proc. of CAAP'88*, págs. 89–101. Springer LNCS 299.

Falaschi M., 1988. *Semantica del non determinismo nei linguaggi logici concorrenti*. Servizio Editoriale Universitario, Università di Pisa. (En italiano).

Falaschi M., Levi G., Martelli M. y Palamidessi C., 1989. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318.

Fay M., 1979. First Order Unification in an Equational Theory. En *Proc of 4th Int'l Conf. on Automated Deduction*, págs. 161–167.

Feather M.S., 1987. A Survey and Classification of some Program Transformation Approaches and Techniques. En *IFIP 87*, págs. 165–195.

Fribourg L., 1985. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. En *Proc. of Second IEEE Int'l Symp. on Logic Programming*, págs. 172–185. IEEE, New York.

Fujita H., 1987. An Algorithm for Partial Evaluation with Constraints. Technical Report TM-367, ICOT, Tokyo, Japan.

Futamura Y., 1971. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50.

Gallagher J., 1993. Tutorial on Specialisation of Logic Programs. En *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, págs. 88–98. ACM, New York.

Gardner P.A. y Shepherdson J.C., 1991. Unfold/fold Transformation of Logic Programs. En J. Lassez y G. Plotkin, eds., *Computational Logic, Essays in Honor of Alan Robinson*, págs. 565–583. The MIT Press, Cambridge, MA.

Glück R., Jørgensen J., Martens B. y Sørensen M., 1996. Controlling Conjunctive Partial Deduction of Definite Logic Programs. En *Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'96*, págs. 152–166. Springer LNCS 1140.

Glück R. y Sørensen M., 1994. Partial Deduction and Driving are Equivalent. En *Proc. Int'l Symp. on Programming Language Implementation and Logic Programming, PLILP'94*, págs. 165–181. Springer LNCS 844.

González-Moreno J., Hortalá-González M. y Rodríguez-Artalejo M., 1992. Denotational versus Declarative Semantics for Functional Programming. En *Proc. of CSL'91*, págs. 134–148. Springer LNCS 626.

Hamada M. y Middeldorp A., 1997. Strong Completeness of a Lazy Conditional Narrowing Calculus. En *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming*, págs. 14–32. World Scientific, Shonan Village.

Hanus M., 1990. Compiling Logic Programs with Equality. En *Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming*, págs. 387–401. Springer LNCS 456.

Hanus M., 1994a. Combining Lazy Narrowing with Simplification. En *Proc. of 6th Int'l Symp. on Programming Language Implementation and Logic Programming*, págs. 370–384. Springer LNCS 844.

Hanus M., 1994b. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628.

Hanus M., 1997. Lazy Narrowing with Simplification. *Computer Languages*, 23(2–4):61–85.

Hanus M., Kuchen H. y Moreno-Navarro J., 1995. Curry: A Truly Functional Logic Language. En *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, págs. 95–107.

Hanus M., Lucas S. y Middeldorp A., 1998. Strongly sequential and inductively sequential term rewriting systems. *Information Processing Letters*, 67(1):1–8.

Hanus M. y Steiner F., 1999. A Type-based Nondeterminism Analysis for Functional Logic Languages. En *8th Int'l Workshop on Functional and Logic Programming, WFLP'98, Grenoble (France).*

Hanus (ed.) M., 1999. Curry: An Integrated Functional Logic Language. Accesible en `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`.

Hogger C., 1981. Derivation of Logic Programs. *ACM*, 28(2):372–392.

Hölldobler S., 1989. *Foundations of Equational Logic Programming*. Springer LNAI 353.

Huet G. y Lévy J., 1992. Computations in Orthogonal Rewriting Systems, Part I + II. En J. Lassez y G. Plotkin, eds., *Computational Logic – Essays in Honor of Alan Robinson*, págs. 395–443. The MIT Press, Cambridge, MA.

Hullot J., 1980. Canonical Forms and Unification. En *Proc of 5th Int'l Conf. on Automated Deduction*, págs. 318–334. Springer LNCS 87.

Hussman H., 1985. Unification in Conditional-Equational Theories. En *Proc. European Conf. on Computer Algebra EUROCAL'85*, págs. 543–553. Springer LNCS 204.

Johnsson T., 1985. Lambda Lifting: Transforming Programs to Recursive Equations. En *Functional Programming Languages and Computer Architecture (Nancy,France)*, págs. 190–203. Springer LNCS 201.

Jones N., 1994. The Essence of Program Transformation by Partial Evaluation and Driving. En N. Jones, M. Hagiya y M. Sato, eds., *Logic, Language and Computation*, págs. 206–224. Springer LNCS 792.

Jones N., Gomard C. y Sestoft P., 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ.

Julián P., 2000. *Especialización de Programas Lógico–Funcionales Perezosos*. Tesis Doctoral, DSIC-UPV. En español.

Julián P. y Moreno G., 1996. Nuevas Semánticas para Programas Lógico–Ecuacionales: Teoría e Implementación. Informe Técnico DSIC-II/13/96, UPV.

Kanamori K. y Fujita H., 1987. Unfold/fold Transformation of Logic Programs with Counters. Informe técnico, En USA-Japan Seminar on Logics of Programs, Accesible en http://www.cs.sunysb.edu/ abhik/transform/papers.html.

Kaplan S., 1987. Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence. *Journal of Symbolic Computation*, 4:295–334.

Kawamura T. y Kanamori T., 1988. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. En *Proc. Int'l Conf. on Fifth Generation Computer Systems*, págs. 413–422. Institute for New Generation Computer Technology, Tokyo.

Kawamura T. y Kanamori T., 1990. Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation. *Theoretical Computer Science*, 75:139–156.

Klop J., 1992. Term Rewriting Systems. En S. Abramsky, D. Gabbay y T. Maibaum, eds., *Handbook of Logic in Computer Science*, volumen I, págs. 1–112. Oxford University Press.

Klop J. y Middeldorp A., 1991. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, págs. 161–195.

Komorowski J., 1991. Synthesis of Programs in the Partial Deduction Framework. En M. Lowry y R. McCartney, eds., *Automating Software Design*, capítulo 15, págs. 377–403. The MIT Press, Cambridge, MA.

Kott L., 1985. Unfold/fold program transformation. En M.Nivat y J. Reynolds, eds., *Algebraic methods in semantics*, capítulo 12, págs. 411–434. Cambridge University Press.

Kuchen H., Loogen R., Moreno-Navarro J. y Rodríguez-Artalejo M., 1990a. Graph-based Implementation of a Functional Logic Language. En *Proc. of ESOP'90*, págs. 279–290. Springer LNCS 432.

Kuchen H., Loogen R., Moreno-Navarro J. y Rodríguez-Artalejo M., 1990b. Lazy Narrowing in a Graph Machine. En *Proc. of the Int'l Conf. on Algebraic and Logic Programming*, págs. 298–317. Springer LNCS 463.

Kurtz S., 1992. Narrowing and Basic Forward Closures. Technical Reports 5, Bielefeld University, Germany.

Lankford D., 1975. Canonical Inference. Informe Técnico ATP-32, University of Texas at Austin.

Lassez J.L., Maher M.J. y Marriott K., 1988. Unification Revisited. En J. Minker, ed., *Foundations of Deductive Databases and Logic Programming*, págs. 587–625. Morgan Kaufmann, Los Altos, Ca.

Leuschel M., De Schreye D. y de Waal A., 1996. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. En M. Maher, ed., *Proc. of the Joint International Conference and Symposium on Logic Programming, JICSLP'96*, págs. 319–332. The MIT Press, Cambridge, MA.

Levi G. y Mancarella P., 1988. The Unfolding Semantics of Logic Programs. Informe Técnico TR-13/88, Dipartimento di Informatica, Università di Pisa.

Levi G. y Sirovich F., 1975. Proving Program Properties, Symbolic Evaluation and Logical Procedural Semantics. En *Proc. of MFCS'75*, págs. 294–301. Springer LNCS 32.

Lloyd J., 1987. *Foundations of Logic Programming*. Springer-Verlag, Berlin. Second edition.

Lloyd J., 1995. Declarative Programming in Escher. Informe Técnico CSTR-95-013, Computer Science Department, University of Bristol.

Lloyd J. y Shepherdson J., 1991. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242.

Loogen R., López-Fraguas F. y Rodríguez-Artalejo M., 1993. A Demand Driven Computation Strategy for Lazy Narrowing. En J. Penjam y M. Bruynooghe, eds., *Proc. of PLILP'93, Tallinn (Estonia)*, págs. 184–200. Springer LNCS 714.

Maher M., 1987a. Correctness of a Logic Transformation System. Informe Técnico RC 13496, IBM - T.J. Watson Research Center, Yorktown Heights, NY.

Maher M., 1987b. Logic semantics for a class of committed-choice programs. En J.L. Lassez, ed., *Proc. of Fourth Int'l Conf. on Logic Programming*, págs. 858–876. The MIT Press, Cambridge, MA.

Maher M., 1993. A Transformation System for Deductive Database Modules with Perfect Model Semantics. *Theoretical Computer Science*, 110(2):377–403.

Middeldorp A., 1997. Call by Need Computations to Root-Stable Form. En *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, págs. 94–105. ACM, New York.

Middeldorp A. y Hamoen E., 1994. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253.

Middeldorp A., Okui S. y Ida T., 1996. Lazy Narrowing: Strong Completeness and Eager Variable Elimination. *Theoretical Computer Science*, 167(1,2):95–130.

Moreno-Navarro J., 1989. *Babel: diseño, semántica e implementación de un lenguaje que integra la programación funcional y lógica*. Tesis Doctoral, Facultad de Informática, Universidad Complutense de Madrid.

Moreno-Navarro J. y Rodríguez-Artalejo M., 1992. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224.

Nielson H. y Nielson F., 1990. Eureka Definition for Free! En N. Jones, ed., *Proc. of 3rd European Symp. on Programming, ESOP'90*, págs. 291–305. Springer LNCS 432.

Nutt W., Réty P. y Smolka G., 1989. Basic narrowing revisited. *Journal of Symbolic Computation*, 7:295–317.

O'Donnell M., 1977. *Computing in Systems Described by Equations*. Springer LNCS 58.

Padawitz P., 1988. *Computing in Horn Clause Theories*, volumen 16 de *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin.

Palamidessi C., 1990. Algebraic Properties of Idempotent Substitutions. En M. Paterson, ed., *Proc. of 17th Int'l Colloquium on Automata, Languages and Programming*, págs. 386–399. Springer LNCS 443.

Pettorossi A. y Proietti M., 1994. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320.

Pettorossi A. y Proietti M., 1996a. A Comparative Revisitation of Some Program Transformation Techniques. En O. Danvy, R. Glück y P. Thiemann, eds., *Partial Evaluation, Int'l Seminar, Dagstuhl Castle, Germany*, págs. 355–385. Springer LNCS 1110.

Pettorossi A. y Proietti M., 1996b. Rules and Strategies for Transforming Functional and Logic Programs. *ACM Computing Surveys*, 28(2):360–414.

Pettorossi A. y Proietti M., 1998. Transformation of Logic Programs. En *Handbook of Logic in Artificial Intelligence*, volumen 5, págs. 697–787. Oxford University Press.

Pettorossi A., Proietti M. y Renault S., 1997. Reducing nondeterminism while specializing logic programs. En *Proc. of 24th Annual ACM Symp. on Principles of Programming Languages, POPL'24*, págs. 414–427. ACM Press, New York.

Peyton-Jones S., 1996. Compiling Haskell by Program Transformation: a Report from the Trenches. En H.R. Nielson, ed., *Proc. of the 6th European Symp. on Programming, ESOP'96*, págs. 18–44. Springer LNCS 1058.

Prestwich S., 1993. Online Partial Deduction of Large Programs. En *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, págs. 111–118. ACM, New York.

Proietti M. y Pettorossi A., 1990. Synthesis of eureka predicates for developing logic programs. En N. Jones, ed., *Proc. of 3rd European Symp. on Programming, ESOP'90*, págs. 306–325. Springer LNCS 432.

Proietti M. y Pettorossi A., 1993. 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.

Reddy U., 1985. Narrowing as the Operational Semantics of Functional Languages. En *Proc. of Second IEEE Int'l Symp. on Logic Programming*, págs. 138–151. IEEE, New York.

Réty P., 1987. Improving basic narrowing techniques. En *Proc. of the Conf. on Rewriting Techniques and Applications*, págs. 228–241. Springer LNCS 256.

Romanenko A., 1991. Inversion and Metacomputation. En *Partial Evaluation and Semantics-Based Program Manipulation*, págs. 12–22. Sigplan Notices, 26(9), ACM, New York.

Roychoudhury A., Kumar K.N., Ramakrishnan C. y Ramakrishnan I., 1999. A Parameterized Unfold/Fold Transformation Framework for Definite Logic Programs. En *Proc. of the International Conference on Principles and Practice of Declarative Programming, PPDP'99, Paris, France*, págs. 141–157. Springer LNCS 1702.

Sahlin D., 1991. *An Automatic Partial Evaluator for Full Prolog*. Tesis Doctoral, Kungliga Tekniska Högskolan, Stockholm, Sweden. Report TRITA-TCS-9101.

Sands D., 1995. Higher Order Expression Procedures. En *Proc. of ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, págs. 178–189. ACM Press, New York.

Sands D., 1996. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Transactions on Programming Languages and Systems*, 18(2):175–234.

Sato T., 1992. Equivalence Preserving First Order Unfold/Fold Transformation System. *Theorical Computer Science*, 105(1):57–84.

Sato T. y Tamaki H., 1984. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240.

Scherlis W., 1981. Program Improvement by Internal Specialization. En *Proc. of 8th Annual ACM Symp. on Principles of Programming Languages*, págs. 41–49. ACM Press, New York.

Seki H., 1990. A Comparative Study of the Well-Founded and Stable Model Semantics: Transformation's viewpoint. En *Proc. of Whorkshop of Logic Programming and Non- Monotonic Logic, Austin, Texas*, págs. 115–123. Association for Logic Programming and Mathematical Sciences Institute, Cornell University.

Seki H., 1991. Unfold/fold Transformation of Stratified Programs. *Theorical Computer Science*, 86(1):107–139.

Seki H., 1993. Unfold/fold Transformation of General Logic Programs for the Well-Founded Semantics. *Journal of Logic Programming*, 16(1&2):5–23.

Slagle J., 1974. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642.

Sørensen M., 1994. Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation. Informe Técnico 94/7, Master's Thesis, DIKU, University of Copenhagen, Denmark.

Sørensen M. y Glück R., 1995. An Algorithm of Generalization in Positive Supercompilation. En J. Lloyd, ed., *Proc. of ILPS'95*, págs. 465–479. The MIT Press, Cambridge, MA.

Sørensen M., Glück R. y Jones N., 1994. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. En D. Sannella, ed., *Proc. of the 5th European Symp. on Programming, ESOP'94*, págs. 485–500. Springer LNCS 788.

Strandh R., 1989. Classes of Equational Programas that Compile into Efficient Machine Code. En *Proc. of RTA'89*, Lecture Notes in Computer Science, págs. 449–461. Springer-Verlag, Berlin.

Tamaki H. y Sato T., 1984. Unfold/Fold Transformations of Logic Programs. En S. Tärnlund, ed., *Proc. of Second Int'l Conf. on Logic Programming, Uppsala, Sweden*, págs. 127–139.

Tamaki H. y Sato T., 1986. A Generalized correctness proof of the unfold/fold logic program transformation. Informe Técnico 86-4, Ibaraki University, Japan, Accesible en http://www.cs.sunysb.edu/ abhik/transform/papers.html.

Turchin V., 1986a. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325.

Turchin V., 1986b. Program Transformation by Supercompilation. En H. Ganzinger y N. Jones, eds., *Programs as Data Objects, 1985*, págs. 257–281. Springer LNCS 217.

Turchin V., 1988. The Algorithm of Generalization in the Supercompiler. En D. Bjørner, A. Ershov y N. Jones, eds., *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, págs. 531–549. North-Holland, Amsterdam.

Turchin V., 1989. *Refal-5, Programming Guide & Reference Manual*. New England Publishing Co., Holyoke, MA.

Vidal G., 1996. *Semantics-Based Analysis and Transformation of Functional Logic Programs*. Tesis Doctoral, DSIC-UPV. En español.

Wadler P., 1985. *Listlessness is better than Laziness*. Computer Science Department, CMU-CS-85-171, Carnegie Mellon Univertsity, Pittsburgh, PA. Ph.D. Thesis.

Wadler P., 1988. Deforestation: transforming programs to eliminate trees. En *Proc of the European Symp. on Programming, ESOP'88*, págs. 344–358. Springer LNCS 300.

Wadler P., 1990. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248.

Walker S. y Strong H., 1972. Characterization of Flowchartable Recursions. En *Proc. of the 4th Annual ACM Symp. on Theory Computing, Denver*, págs. 18–34. ACM Press, New York.

You Y., 1989. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7:319–343.

Zartmann F., 1997. Denotational Abstract Interpretation of Functional Logic Programs. En P.V. Hentenryck, ed., *Proc. of the 4th Int'l Static Analysis Symposium, SAS'97*, págs. 141–159. Springer LNCS 1302.

Zhu H., 1994. How powerful are folding/unfolding transformations? *Journal of Functional Programming*, 4(1):89–112.

Índice de Materias