Evaluación Parcial *Offline* Dirigida por *Narrowing*:
Técnicas de Optimización y Aplicaciones

Gustavo Arroyo Delgado
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Memoria presentada para optar al título de:

Doctor en Informática

Directores:

Dr. Germán F. Vidal Oriola
Dr. J. Guadalupe Ramos Díaz

Tribunal de Lectura:

Presidente:	Dr. Javier Oliver Villarroya	U.P. Valencia
Vocales:	Dr. Jesús Almendros Jiménez	U. de Almería
	Dr. Pascual Julián Iranzo	U. Castilla-La Mancha
	Dr. Ginés Moreno Valverde	U. Castilla-La Mancha
Secretario:	Dr. César Ferri Ramírez	U.P. Valencia

Valencia, Octubre de 2012

Dedicada a:
Angélica, mi esposa.
A mis hijas Zanya e Itzayana.
Juntos, los cuatro, hemos culminado esta aventura.

Resumen

La evaluación parcial dirigida por *narrowing* (NPE) [Vid96, AFV98, AV02] es una técnica para la especialización de programas funcionales y lógico funcionales. La técnica original es de tipo *online*, es decir, realiza la propagación de valores estáticos y los controles de terminación durante el propio proceso de especialización. Los esquemas *offline*, por el contrario, tienen dos fases bien diferenciadas: el análisis de tiempo de enlace, en el que se propagan los parámetros estáticos y se determina la estrategia de control mediante anotaciones, y la fase de especialización propiamente dicha (la cual sólo debe seguir las anotaciones de la primera fase). En general, se dice que la evaluación parcial *offline* es más eficiente—pero menos precisa—que la evaluación parcial *online*.

El objetivo principal de esta tesis es mejorar la aproximación *offline* a la evaluación parcial dirigida por *narrowing* de [RSV05a]. En particular, se mejora el procedimiento de anotación y se implementa un evaluador parcial más efectivo con el fin de obtener programas mejor especializados y más rápidos. Como aplicaciones, empleamos el evaluador parcial para llevar a cabo la especialización de un intérprete, lo que de acuerdo a la *primera proyección de Futamura* [Fut71] se considera como un tipo de compilación. Esta aplicación representa la primera aproximación a la compilación por evaluación parcial de programas lógico funcionales. Además, consideramos también la especialización de un lenguaje de dominio específico que tiene como propósito generar programas para máquinas de control numérico computarizado (CNC).

Desde un punto de vista más teórico, se extiende la técnica *offline* básica para programas funcionales de orden superior mediante la aproximación conocida como *desfuncionalización*. Además, se aplica una *transformación polivariante* que permite mejorar la precisión del proceso de

especialización sin necesidad de implementar un esquema polivariante real.

El esquema resultante tiene un buen equilibrio entre eficiencia y precisión, mejorando así las propuestas anteriores y constituyendo un buen punto de partida para el desarrollo de herramientas de especialización de programas lógico funcionales eficaces.

Abstract

The narrowing driven partial evaluation (NPE) [Vid96, AFV98, AV02] is a technique for the specialization of functional and functional logic programs. The original technique follows the online approach of partial evaluators, i.e., it performs the propagation of static values and the termination controls during the proper specialization process. On the other hand, the offline schemes usually proceed in two distinct phases: the binding time analysis, which propagates the static parameter values and determines the control strategy to be employed by adding annotations into the subject program, then the proper specialization phase, just obeys the introduced annotations. Offline partial evaluators are usually more efficient—but less precise—than online partial evaluators.

The aim of this thesis is to improve the approach to offline narrowing driven partial evaluation of [Ram07]. In particular, the annotation procedure is enhanced and a more effective partial evaluator is developed in order to get better specialized and faster programs. As applications, we carry out the specialization of interpreters which, in agreement with the *first Futamura projection* [Fut71], is considered like a kind of program compilation. This application represents the first approach to the compilation by partial evaluation of functional logic programs. Additionally, we consider the specialization of a domain specific language whose goal is to generate programs for computerized numerical control of machines (CNC).

From a theoretical point of view, the basic offline technique for higher order functional programs is extended through an approach which is known as *defunctionalization*. Furthermore, a *polivariant transformation* that improves the accuracy of specialization process is applied without to implement a real polivariant scheme.

The resulting scheme has a good balance between efficiency and precision, thus improving previous proposals and providing a good starting point for the development of effective specialization tools for functional logical programs.

Resum

L'avaluació parcial dirigida per *narrowing* (NPE) [Vid96, AFV98, AV02], és una tècnica per a l'especialització de programes funcionals i lògic-funcionals. La tècnica original és de tipus *online*, és a dir, realitza la propagació de valors estàtics i els controls de terminació durant el propi procés d'especialització. Els esquemes *offline*, per contra, tenen dues fases ben diferenciades: l'anàlisi de temps d'enllaç, en el qual es propaguen els paràmetres estàtics i es determina l'estratègia de control mitjançant anotacions, i la fase d'especialització pròpiament dita (la qual només ha de seguir les anotacions de la primera fase). En general, es diu que l'avaluació parcial *offline* és més eficient—però menys precisa—que l'avaluació parcial *online*.

L'objectiu principal d'aquesta tesi és millorar l'aproximació *offline* a l'avaluació parcial dirigida per narrowing de [Ram07]. En particular, es millora el procediment d'anotació i s'implementa un avaluador parcial més efectiu per tal d'obtenir programes millor especialitzats i més ràpids. Com aplicacions, emprem l'avaluador parcial per dur a terme l'especialització d'un intèrpret, el que d'acord a la *primera projecció de Futamura* [Fut71] es considera com un tipus de compilació. Aquesta aplicació representa la primera aproximació a la compilació per avaluació parcial de programes lògic-funcionals. A més, considerem també l'especialització d'un llenguatge de domini específic que té com a propòsit generar programes per a màquines de control numèric computat (CNC).

Des d'un punt de vista més teòric, s'estén la tècnica *offline* bàsica per a programes funcionals d'ordre superior mitjançant l'aproximació coneguda com *desfuncionalització*. A més, s'aplica una *transformació polivariant* que permet millorar la precisió del procés d'especialització sense necessitat d'implementar un esquema polivariant real.

L'esquema resultant té un bon equilibri entre eficiència i precisió, millorant així les propostes anteriors i constituint un bon punt de partida per al desenvolupament d'eines d'especialització de programes lògic-funcionals eficaços.

Agradecimientos

Esta tesis doctoral, como producto de un posgrado de alta calidad, es el resultado de numerosos esfuerzos y bastantes horas de trabajo arduo no sólo míos, sino de muchas personas organismos e Instituciones. Quiero agradecer a todos esperando no excluir a ninguno.

A las autoridades de la DGIT quienes conjuntamente con las correspondientes del CENIDET, en su momento, iniciaron en Cuernavaca este proyecto de doctorado.

En especial quiero agradecer a mis directores de tesis:

- A Germán Vidal, desde un principio supe que debía trabajar con él. Todos los del grupo de la etapa de docencia sabíamos que su área era la más abstracta y difícil pero también, para mí, la más afín, gracias a él hemos descifrado una pequeña parte de la evaluación parcial en el paradigma lógico funcional. Gracias también a su rigor científico y a la colaboración de J. Guadalupe Ramos, Josep Silva, Salvador Tamarit y Claudio Ochoa; he podido salvar la difícil marca que significa el proyecto de tesis y el desarrollo de la misma. En todos los sentidos gracias Germán.

- A J. Guadalupe Ramos, como compañero de cursos, como colaborador en publicaciones y luego como mi propio director de tesis. Gracias por su invaluable ayuda.

A los profesores de la etapa de docencia del doctorado: Isidro Ramos, Matilde Celma, Ma. José Ramírez, Oscar Pastor, María Alpuente, Salvador Lucas y Germán Vidal. Todos ellos investigadores y científicos de una calidad indiscutible.

A los compañeros que iniciaron conmigo esta aventura, particularmente por su amistad a Rogelio Limón y Ricardo Blanco, mención especial merece J. Guadalupe Ramos, además de su amistad por su apoyo académico y consejos técnicos.

Mi más sincero agradecimiento a los compañeros del Departamento de Sistemas informáticos y Computación (DSIC) de la UPV, particularmente a Vicent Estruch, Beatriz Alarcón, Antonio Bella y Salvador Tamarit compañeros del laboratorio, siempre echaré de menos su desinteresada camaradería. A Cesar Ferri, Santiago Escobar y José Hernández; quienes además de brindarnos su amistad y confianza, nos han compartido la cultura de la Comunidad Valenciana.

A las autoridades de la DGEST, Carlos Alfonso García Ibarra y del Centro Interdisciplinario de Investigación y Docencia en Educación Técnica (CIIDET), Maricela Castillo, José Carlos Paz, Roberto de la Torre, Juan Manuel Ricaño y Adrián Pesina. Por su apoyo de gestión sin el cual no se hubiera logrado este resultado.

No puedo dejar de agradecer a las instituciones que han financiado en gran parte este proyecto:

- En México, al convenio SES-ANUIES quien nos apoyó en las estancias de Cuernavaca y España. A la DGEST-SEP y al CIIDET quienes en todo momento han estado pendientes de este proyecto.

- En España, al grupo de investigación MIST por su apoyo para asistir a congresos nacionales y escuelas de verano en otros países europeos. Al personal del DSIC y al Vicerrectorado de Relaciones Internacionales y Cooperación de la UPV.

A mis hermanos que de algún modo me han apoyado: Oscar, Angelina, en especial a Ma. Abraham. A Felisa y Florentino, mis padres; a quienes les debo mucho más que mi existencia. A mi hijas Zanya e Itzayana y a mi esposa Angélica; quienes con sacrificios, me han acompañado los casi tres años de estancia en España. Estoy seguro que sin ellas no hubiera logrado culminar la tesis.

Gracias a la fe que profeso, porque me ha dado fuerza en todo momento y gracias a la sociedad que me la ha dado. De todo corazón, gracias al principio de esta fe.

Índice general

Capítulo 1

Introducción

En este capítulo presentamos el conjunto de ideas fundamentales que componen el marco contextual de la tesis. Primero, presentamos el concepto general de evaluación parcial de programas, sus definiciones, antecedentes, los tipos *online* y *offline* de evaluación parcial. Posteriormente presentamos el marco de evaluación parcial dirigida por *narrowing*. A continuación mostramos algunas consideraciones de los lenguajes lógico funcionales y una ligera introducción al lenguaje lógico funcional Curry, ya que éste será el lenguaje de los programas a manipular. Más adelante discutimos la relación entre evaluación parcial y compilación, después damos una introducción a los lenguajes de dominio específico. Estos dos últimos temas serán motivo de aplicación y experimentación de la evaluación parcial. Antes de finalizar el capítulo presentamos el objetivo general y las contribuciones de la tesis. Concluimos con una breve explicación del contenido total del documento.

1.1. Evaluación parcial de programas

De acuerdo al esquema original de evaluación parcial de Jones, Gomard y Sestoft [JGS93], dados un programa `p` y sólo una parte de sus datos de entrada, un evaluador parcial intentará ejecutar `p` tanto como sea posible produciendo como resultado un *programa residual* (o especializado) el cual ejecutará el resto de los cómputos cuando se le suministre el resto de los datos de entrada y producirá el mismo resultado que `p`

habría computado con ambos datos.

1.1.1. Definiciones y antecedentes de la EP

La evaluación parcial (EP) de programas es una técnica formal para la especialización y optimización de programas. Un evaluador parcial toma un programa y sólo una parte de sus datos de entrada (los llamados datos estáticos) e intenta llevar a cabo todas las computaciones que sean posibles a partir de tales datos. El evaluador parcial devuelve un programa nuevo, denominado programa *residual* el cual se ejecuta generalmente de manera más eficiente que el programa original, ya que las computaciones que dependen de los datos estáticos se han realizado en la fase de evaluación parcial de una vez y para siempre [JGS93]. La evaluación parcial es una técnica de optimización de programas basada en semántica la cual ha sido investigada dentro de diferentes paradigmas de programación y aplicada a una amplia variedad de lenguajes. También es conocida como una técnica de transformación de programas fuente-a-fuente para especializar programas con respecto a una parte de sus datos de entrada (por ello también es conocida como especialización de programas). La evaluación parcial ha sido intensamente aplicada en el área de la programación funcional [CD93, JGS93, Tur86] y en programación lógica [Gal93, Kom82b, LS91, PP94], donde ésta es normalmente conocida como *deducción parcial*. También en lenguajes imperativos como C en [TBC+98], o aplicada a un subconjunto importante de C en [And92] donde reportan la primera implementación autoaplicable de evaluación parcial para un lenguaje imperativo. Y en lenguajes formales como Scheme en [Jør92a, Jør92b] donde generan compiladores a partir de intérpretes.

Cuando tenemos un programa sólo con algunos de sus datos de entrada conocidos no podemos ejecutar el programa, sin embargo podemos optimizar el programa computando respuestas tanto como sea posible. La evaluación parcial es una técnica que permite la ejecución parcial de un programa [MS97]. En el entorno de los lenguajes funcionales las técnicas de evaluación parcial dependen generalmente de la propagación de constantes y de la reducción de expresiones, mientras que en los lenguajes lógicos tales técnicas usan la propagación de información basada en

unificación [GS94]. La aproximación de las transformaciones de plegado y desplegado, introducidas por [BD77] para optimizar programas funcionales y posteriormente usadas en programación lógica [AFMV04], son ampliamente usadas por las diferentes técnicas de EP. El desplegado (*unfolding*) es esencialmente el reemplazo de una llamada por su definición, con sustituciones apropiadas, esto es, una expansión. El plegado (*folding*) es la transformación inversa, es decir, la sustitución de una expresión por una llamada a función donde el cuerpo de esta expresión es equivalente a su definición, esto es, una contracción. En trabajos relacionados se han establecido algunas correspondencias entre las diferentes técnicas en particular los casos [GS94, PP96, SGJ96a].

La evaluación parcial ofrece un paradigma abierto, unificador y en constante evolución en la optimización de programas, interpretación y compilación, en diferentes formas de generación de programas e incluso en la generación de generadores automáticos de programas [JGS93]. Es una técnica de optimización de programas, quizá mejor llamada *especialización de programas*. Gran parte del trabajo de evaluación parcial tiene relación con la generación automática de compiladores a partir de la implementación de una aproximación a una semántica de un lenguaje de programación a través de un intérprete, pero además, la evaluación parcial tiene importantes aplicaciones en la computación científica, programación lógica, metaprogramación y sistemas expertos [Jon96].

1.1.2. Evaluación parcial *online* y *offline*

Existen dos tendencias en el campo de la evaluación parcial: *offline*, evaluación parcial en etapas, y *online*, evaluadores parciales monolíticos [CD93]. En el estilo *offline* la evaluación parcial es considerada como un proceso de dos fases: una fase de pre-procesamiento usualmente basada en un *análisis de tiempo de enlace* [JSS89] (BTA[1]) y la propia fase de especialización. A su vez, los evaluadores parciales *online* son esencialmente intérpretes no-estándar [CD93]. Los cuales evalúan expresiones mientras tengan suficiente información disponible y por otra parte producen código residual. El tratamiento para cada expresión es determinado al vuelo,

[1]Siglas que provienen de la expresión en inglés: *binding-time analysis*.

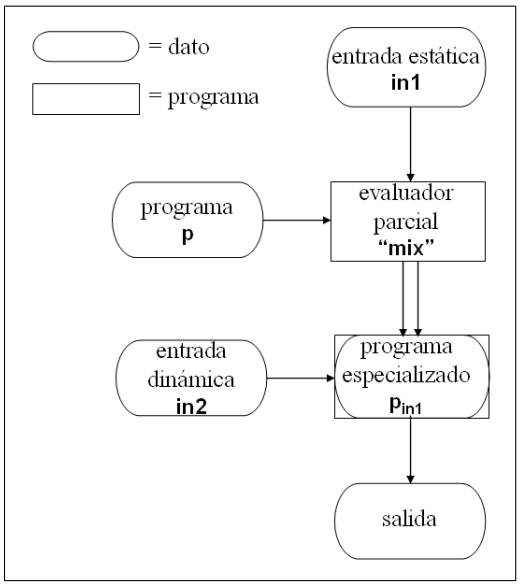


Figura 1.1: Esquema original de evaluación parcial de [JGS93].

lo cual es muy contrastante con la evaluación parcial *offline* [AV02].

Un evaluador parcial *online* ejecuta un sólo y gran proceso el cual combina evaluación simbólica, propagación de valores parciales, y algún análisis dinámico para garantizar la terminación del proceso y producir así un programa especializado. Un evaluador parcial realiza una mezcla de ejecución con acciones de generación de código —por ésta razón Ershov utilizó el término "mixed computation" en el mismo sentido que el proceso de evaluación parcial realiza [MS97], y de ahí el nombre de **mix** asociado al concepto de evaluación parcial (véase la Figura 1.1).

Por otro lado, los evaluadores parciales *offline* tienen dos procesos claramente separados. El objetivo del primer proceso es agregar algunas anotaciones en el programa para guiar el proceso de especialización. Usualmente incluye algún análisis estático para propagar valores conocidos a través del programa completo, así como un análisis de terminación, y de esta forma producir un nuevo programa anotado. Después, en el segundo paso, la propia especialización, toma el programa anotado, conjuntamente con los valores de entrada conocidos y produce el programa residual asociado obedeciendo principalmente las anotaciones del progra-

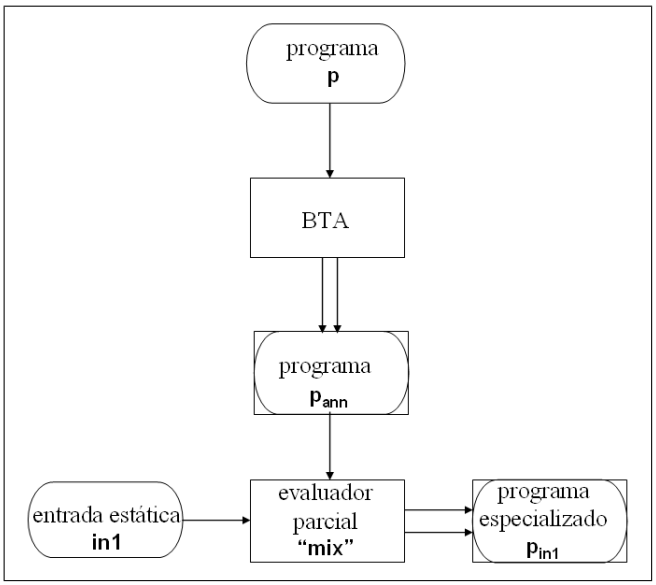


Figura 1.2: Esquema general de la evaluación parcial *offline*.

ma (véase la Figura 1.2).

El BTA, necesario en caso de evaluación parcial *offline*, también tiene como objetivo determinar cuando está disponible el valor de una variable: si el valor es conocido en tiempo de compilación se determina como *estático*, si el valor es conocido hasta el momento de la ejecución se dice que es *dinámico*. El análisis de tiempo de enlace de una expresión puede ser resumido como sigue: una *constante* tiene un valor de *tiempo de enlace estático*, el valor de *tiempo de enlace* de una *variable* está definido por un *entorno del análisis de tiempo de enlace*[2], una *expresión condicional* es *estática* sí cada subcomponente, de la misma, es *estática* de otra forma es *dinámica*, una *llamada primitiva* es *estática* si cada argumento es *estático* de otra forma es *dinámica*, finalmente una *llamada a función* es *estática* sí el valor de *tiempo de enlace* del tipo resultante es *estático* de otra forma es *dinámico* [Con90] (véase la secc.5.3).

Un BTA generalmente realiza una análisis estático para propagar los valores abstractos conocidos, es decir, los valores estático y dinámico a

[2]Frase derivada de la expresión: *binding-time environment*, propiamente es una sustitución que correlaciona variables con valores de *tiempo de enlace*.

través del programa completo. En nuestro caso, el BTA calcula un punto fijo sobre los argumentos de las funciones del programa en cuestión; a partir de los valores iniciales de una llamada a función. El resultado de nuestro BTA es una división *monovariante*, es decir, que se asocia una sóla secuencia de valores de *tiempo de enlace* a los argumentos de cada función del programa.

Particularmente nuestro procedimiento de anotación está basado en un análisis de cuasi-terminación, procedimiento que se discute en el capítulo 4, el cual a su vez está basado en el principio de terminación de los grafos de cambio de tamaño (o grafos *size change*), éste análisis es utilizado para rastrear cambios de tamaño entre los argumentos de función cuando van de una llamada a otra, así como para determinar los bucles de un programa que en última instancia sirven para analizar y garantizar su terminación. Para afinar un poco más la precisión de la especialización, consideramos la salida de un BTA monovariante que en términos generales, propaga una llamada inicial con argumentos en el dominio estático/dinámico. El BTA retorna como resultado una lista de funciones del programa con sus argumentos evaluados en este mismo dominio, es decir, una *división* monovariante (véa [JGS93]).

1.2. Evaluación parcial dirigida por *narrowing* (NPE)

En [AV02] se examinan los fundamentos de la evaluación parcial *online* dirigida por *narrowing* . Introducido originalmente por Slagle [Sla74] como un mecanismo de demostración de teoremas, *narrowing* es un método completo y correcto para resolver ecuaciones con respecto a un conjunto de reglas *confluente* y *terminante* [Hul80]. Esta es una razón suficiente para usar *narrowing* como un principio básico para definir la semántica de ejecución de los programas lógico funcionales [Han94].

La evaluación parcial dirigida por *narrowing* (NPE: *Narrowing-driven partial evaluation*) [AV02] es una poderosa técnica de especialización para el componente de primer orden de diversos lenguajes funcionales como Haskell [PJ03] y lógico funcionales como Curry [Han06]. En NPE, se usa un perfeccionamiento del *narrowing* [Sla74] para ejecutar computación

simbólica, siendo *narrowing* necesario (*needed narrowing*) [AEH00] la estrategia que presenta mejores propiedades. En [AFV98] afirman que NPE esta formalizado dentro del marco teórico establecido en [LS91] y por Martens y Gallagher en [MG95] para la deducción parcial de programas lógicos, aunque varios conceptos han sido generalizados para ocuparse de las características funcionales tales como: funciones definidas por el usuario, llamadas a función anidadas, estrategias de evaluación ansiosa y perezosa, y la reducción de pasos determinísticos. En general, el espacio de términos de *narrowing* puede ser infinito, sin embargo, incluso en ese caso, NPE puede terminar cuando el programa original es *cuasi-terminante* con respecto a la estrategia de *narrowing* considerada, es decir, cuando se computan definitivamente solo términos diferentes —módulo renombramiento de variables. La razón es que la evaluación (parcial) de múltiples ocurrencias del mismo término (módulo renombramiento de variables) en un computo pueden ser evitadas insertando una llamada a alguna variante previamente encontrada (una técnica conocida como *inserción de puntos de especialización* en la literatura de evaluación parcial[GJ05]).

En [Vid96] Se especifica la primera aproximación del marco general para la evaluación parcial de programas lógico funcionales, el cual consiste en la formalización de un algoritmo genérico de EP para alguna estrategia de *narrowing*. El método es paramétrico con respecto a la relación de *narrowing*, a la regla de desplegado y al operador de abstracción. En el documento ya se consideran los dos niveles de especialización introducidos en [MG95], también se demuestra la corrección, completitud y terminación del algoritmo.

En [AFJV97, Jul00] se formula una instancia del algoritmo genérico de NPE [AFV98] basada en el empleo de *narrowing* perezoso (Moreno-Navarro y M. Rodríguez-Artalejo [MR92]). El proceso de evaluación parcial lo formalizan en dos fases. En la primera se aplica una instancia concreta del método NPE y, a continuación, se introduce una fase de renombramiento que es necesaria para conseguir recuperar la disciplina de constructores (reglas sin funciones anidadas y sin variables repetidas en la parte izquierda). El postproceso de renombramiento también es necesario para lograr la llamada condición de independencia del con-

junto de términos evaluados parcialmente una condición indispensable para garantizar que el programa transformado no produzca respuestas adicionales y por lo tanto indeseadas. En [Jul00] se introducen mejoras a los mecanismos de control mediante una regla de desplegado dinámica y se introducen técnicas de partición de términos evaluados parcialmente para evitar una acción de generalización excesiva a nivel global similares a las de [GJMS96, LDdW96].

Como hemos introducido en [AFV98] presentan un algoritmo NPE para programas lógico funcionales el cual sigue una estructura similar al marco conceptual desarrollado por Martens y Gallagher [1995] para deducción parcial donde se hace una clara distinción entre control *local* y *global*. A grosso modo indican que el control local involucra la construcción de árboles parciales de *narrowing* para términos individuales, mientras que el control global esta dedicado a garantizar la cerradura del programa evaluado parcialmente sin riesgo de no terminación. Tal algoritmo inicia evaluando parcialmente el conjunto de llamadas que aparecen en la llamada inicial, y después recursivamente especializa los términos introducidos dinámicamente en el proceso. Introduce apropiadamente operadores de desplegado y abstracción (generalización) los cuales aseguran terminación (tanto local como global). De esta forma, dicho marco conceptual define un método de especialización, que es independiente de la estrategia *narrowing* y que siempre termina y garantiza la cerradura del programa resultante. El concepto recursivo de cerradura incrementa sustancialmente el poder de la especialización del método, ya que la generalización necesita ser aplicada en menos casos que cuando se aplica un concepto estándar de cerradura. Usando la terminología de Glück y Sørensen [1996], su estrategia de control global les permite producir tanto especializaciones *polivariantes* como *poligenéticas*, esto es, la evaluación parcial dirigida por *narrowing* puede producir diferentes especializaciones para la misma función, y puede combinar distintas funciones originales dentro de una función integral especializada.

Alpuente et al. [AHLV99] introducen una instancia del método NPE para programas inductivamente secuenciales basada en la estrategia de *narrowing* necesario (Antoy et al. [AEH00]). El método traslada al esquema de evaluación parcial la idea de evaluar código sólo cuando es

necesario. Además, esta instancia preserva la estructura del programa original, i.e., el programa residual es también inductivamente secuencial, propiedad que no se cumple en general para otras instancias del marco NPE (ver [AHLV99]).

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

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

El desarrollo de un esquema de evaluación parcial para programas lógico funcionales realistas requiere el tratamiento de características avanzadas, tales como: orden superior, restricciones, llamadas a funciones externas, etc. Para tratar con tales características se requeriría un cálculo operacional muy complejo. En [HP99] se introduce una representación abstracta para programas en la que los árboles definicionales [Ant92] (usados para guiar la estrategia de *narrowing* necesario) se hacen explícitos por medio de construcciones *case*. También, se formula una semántica operacional para esos programas: el cálculo LNT[3]. En [AHV00b, AHV00a] se introduce un marco de trabajo en el que los programas de alto nivel son traducidos a la representación abstracta de programas y se incluye una extensión residualizante del cálculo LNT, es decir, el cálculo RLNT (Residualizing LNT). A partir del nuevo cálculo es posible implementar un evaluador parcial realista para programas en representación abstracta. En [AHV03] se demuestra la equivalencia entre el cálculo LNT y RLNT. En [AHV01] y [AHV02] se describe el esquema práctico de evaluación parcial (*online*) dirigido por *narrowing* para programas abstractos (traducidos a partir de programas Curry) y la forma en que se resuelven las características extendidas del lenguaje: guardas, restricciones,

[3]Del inglés: *Lazy Narrowing with definitional Trees.*

funciones externas, orden superior, etc.

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

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

En [RSV05a] introdujeron la primera aproximación al esquema *offline* para NPE, además identificaron una clase sistemas de reescritura cuasi-terminantes (con respecto a *narrowing* necesario) a los que llamaron *no-crecientes*. Esta caracterización es puramente sintáctica y muy fácil de verificar, aunque muy restringida para ser útil en la práctica. Algunas de las consideraciones que hacen [RSV05a] en la primera etapa de este novel esquema de EP son:

1. anotan las expresiones del programa que *violan la propiedad no-creciente* y,

2. consideran una ligera extensión del *narrowing* necesario para ejecutar los cómputos parciales tal que los subtérminos anotados son

generalizados al momento de la especialización (lo que asegura la terminación del proceso).

1.3. Lenguajes declarativos lógico funcionales

De acuerdo a [BL86] algunos de los principales argumentos que han sido considerados para la integración de los paradigmas de programación lógico y funcional son los siguientes:

1. *Notación.* Puesto que los lenguajes lógicos (basados en la definición de claúsulas de Horn) permiten la definición de *relaciones*, y ya que algunos problemas pueden ser descritos de manera más natural en términos de *funciones*. Entonces un lenguaje sintácticamente poderoso debe permitir la definición y composición de *relaciones* y *funciones*.

2. *Control.* Un programa funcional contiene mucho más información de control que el correspondiente programa lógico. Tal información puede ser aprovechada por una implementación determinista muy eficiente, un lenguaje funcional es entonces muy adecuado para definir algoritmos. El mismo objetivo puede ser alcanzado en programas lógicos a través de una compleja combinación de información y control declarativos.

3. *Características del lenguaje.* Los lenguajes funcionales ofrecen una variedad de conceptos de programación de gran alcance (*funciones de orden superior, evaluación perezosa, tipos y polimorfismo*), que son difíciles de fundir en el marco conceptual de la programación lógica y que podrían ser fáciles de entender en un lenguaje lógico funcional.

4. *Proceso de reescritura.* En los lenguajes lógicos existe un procedimiento de refutación completa llamada resolución SLD; que es esencialmente un *proceso de reescritura indeterminista* y que calcula soluciones por composición de los unificadores más generales.

La *reescritura* es la típica semántica operacional de los lenguajes de programación funcionales. La propiedad de la correspondiente semántica declarativa es la existencia de un modelo estándar (el mínimo modelo de Herbrand), que también puede ser obtenido como un mínimo punto fijo.

Considerando algunos aspectos similares entre programas lógicos y funcionales tenemos los siguientes:

(1) *Notación*. Relaciones vs. Funciones.

(2) *Estilo*. Definiciones por claúsulas separadas vs. expresiones condicionales.

(3) *Determinismo*. Búsqueda basada en el cálculo vs. cálculo determinista.

(4) *Variables lógicas*. Unificación vs. paso de parámetros y valor de retorno.

Según [BL86] los aspectos (1) y (2) son esencialmente sintácticos, los aspectos (3) y (4) son aspectos semánticos realmente relevantes cuyas soluciones técnicas están basadas en alguna de las siguientes ideas.

- Agregar variables lógicas (y a veces de búsqueda) a un lenguaje funcional para obtener un lenguaje lógico con notación funcional.
- Restringir el comportamiento de las variables lógicas (y a veces de la regla de búsqueda) de un lenguaje lógico para obtener un lenguaje "más funcional".

Se han construido varios sistemas con el objetivo de disponer en un sólo entorno un lenguaje funcional y un lenguaje lógico. La mayoría de éstos sistemas están basados en LISP o en dialectos de LISP, por ejemplo LOGLISP [RS82a, RS82b] y QLOG [Kom82a]. La semántica operacional de los lenguajes ecuacionales, tal como la semántica de cualquier lenguaje ecuacional, puede ser definida en términos de reducciones (o *reescrituras*). Si se permiten variables cuantificadas existencialmente y unificación, tendríamos una semántica operacional basada en *narrowing*.

Aplicar *narrowing* a una expresión funcional es aplicarle la sustitución mínima tal que la expresión resultante pueda ser reducida. La sustitución es encontrada unificando la expresión (funcional) con las partes izquierdas de las ecuaciones. En general habrá algunas reducciones por *narrowing* de la expresión, una para cada ecuación cuya parte izquierda sea unificable con la expresión.

La evaluación perezosa ha mostrado ser muy útil en lenguajes funcionales para definir una regla de evaluación externa eficiente y para manejar estructuras de datos infinitas. Es muy difícil extender las técnicas de evaluación perezosa a programación lógica. Implementar esto requeriría la existencia de una relación de orden parcial sobre los átomos, similar a una que es implícita en el anidamiento de un programa funcional. Algunos ejemplos de lenguajes lógicos caracterizando la evaluación perezosa son:

El lenguaje propuesto por Reddy [Red85], el cual define una versión especial de *narrowing* (*narrowing* perezoso) y del cual demuestra su completitud.

LEAF [BBLM84, BBLM86], que también considera evaluación perezosa de *relaciones*. El orden parcial está definido anotando variables así como átomos relacionales.

Las funciones de *orden superior* constituyen una de las características más atractivas de la programación funcional la cual debe ser preservada en alguna extensión de la programación funcional con características de programación lógica. Para realizar esto se requiere la implementación de un algoritmo de *unificación de orden superior*. Al respecto Hanus [HKMN95] menciona que la *unificación de orden superior* es necesaria para calcular todas las soluciones de todas las consultas [NM88, Pre94]. Si las variables lógicas ubicadas en las funciones se han cuantificado sobre todas las (aplicaciones parciales de las) funciones definidas, en lugar de todas las expresiones lambda, la *unificación de orden superior* puede ser evitada y reemplazada por una enumeración de todos los símbolos de función (acorde a un tipo) [GHGRA92, War82]. Curry soporta éstas dos formas de tratar el orden superior, además provee una forma restringida de *unificación de orden superior* (ya que las partes izquierdas de las definiciones de función son necesarias para formar patrones, discrepando con λProlog [NM88]) y de una anotación para variables de función es-

pecificando que éstas variables están cuantificadas exclusivamente sobre todos los símbolos de función que ocurren en el programa. Curry es una integración real de los lenguajes lógicos y funcionales ya que cubre la mayoría de los aspectos de ambos paradigmas. Para el caso de la programación funcional, Curry provee funciones de orden superior, evaluación perezosa y evaluación determinista de expresiones básicas[4]. Las características de programación lógica están soportadas por variables lógicas, estructuras de datos parciales y servicios de búsqueda.

Debido a la disponibilidad de algunas nuevas características en relación a programación lógica pura, Curry evita las siguientes construcciones impuras de Prolog:

- El operador *cut* y algunos otros operadores de poda similares son reemplazados por la evaluación determinista de funciones.
- El predicado *call* es reemplazado por las características de *orden superior* de Curry.
- Las operaciones de *entrada/salida* de Prolog son reemplazadas por el concepto declarativo monádico de *entrada/salida* de la programación funcional.

La integración de la programación lógica y funcional ha sido abordada en dos formas. Desde el punto de vista funcional, los aspectos de la programación lógica pueden ser integrados dentro de los lenguajes funcionales permitiendo variables lógicas en expresiones y reemplazando la operación de emparejamiento (en un paso de reducción) por unificación [Red85]. Desde el punto de vista de la programación lógica, las funciones pueden ser integradas dentro de los lenguajes lógicos combinando el principio de resolución con alguna clase de evaluación funcional [Han94]. Aunque, no hay una forma obvia de combinar los servicios de búsqueda de soluciones de la programación lógica con los eficientes principios de evaluación de la programación funcional. Las aproximaciones funcionales de evaluación requieren un flujo de datos dirigido y no permite manejar estructuras de datos de instancias parciales. Tales aproximaciones las

[4]Mejor conocidas como: *ground expressions*.

cuales permiten un flujo de datos arbitrarios guardan cierto equilibrio entre la completitud y la eficiencia. Como consecuencia han sido propuestos diferentes métodos para integrar los lenguajes lógicos y funcionales. Los principios operacionales más prometedores son *narrowing* y *residuación*.

El principio de *residuación*[5] se basa en la idea de retardar las llamadas a función hasta que estén listas para su evaluación determinista. El principio de *residuación* representa una alternativa de integración razonable de los paradigmas lógico y funcional; en la medida que éste combine la reducción determinista de funciones con estructuras de datos parciales (variables lógicas). Desafortunadamente es incompleto, ya que es incapaz de calcular soluciones si no se encuentran instancias suficientes de los argumentos de funciones durante la computación.

Por su parte *narrowing* es una combinación de unificación, por paso de parámetros, y reducción como mecanismo de evaluación. Es completo tanto en el sentido de la programación funcional como en el respectivo de la programación lógica. Para obtener una implementación eficiente se requieren sofisticadas estrategias de *narrowing*. La estrategia de *narrowing necesario* [AEH94] intercala la evaluación de los argumentos demandados con un mecanismo de indexación para seleccionar las reglas aplicables. Es óptima con respecto a la longitud de las derivaciones y al número de soluciones calculadas. Esto muestra claramente la ventaja de integrar funciones en programas lógicos: transfiriendo resultados desde la programación funcional a la programación lógica se obtienen mejores y optimas estrategias de evaluación sin perder los servicios de búsqueda [HK96].

La integración de funciones dentro de la programación lógica es muy simple desde un punto de vista sintáctico. Por lo cual se tiene que extender el lenguaje lógico por medio de:

1. Un método para definir nuevas funciones.

2. La posibilidad de utilizar estas funciones dentro de las cláusulas del programa.

[5]El desarrollo de un lenguaje lógico funcional estándar se ha complicado debido a que no hay un acuerdo sobre la semántica operacional. Aunque hay dos principales aproximaciones a considerar: *residuation y narrowing [HK96]*.

Para llevar a cabo el primer punto se podría permitir la construcción de funciones en un lenguaje funcional externo. La alternativa más interesante es la integración directa de definiciones de función dentro de un lenguaje lógico. Para ello se tendría que permitir claúsulas de programa definiendo el predicado de igualdad. La igualdad "=" es un predicado predefinido en sistemas Prolog la que se satisface sí y solo sí ambos argumentos son sintácticamente iguales (es decir, unificación sintáctica de ambos argumentos). En [Han94] se puede consultar un estudio completo de la integración de funciones en programación lógica. Como muestra de la segunda aproximación a continuación se muestran algunas características sintácticas de Curry:

El sistema de tipos de Curry permite crear fácilmente nuevos tipos a partir de aquellos básicos. Por ejemplo, el siguiente código:

```
data Boolean = True | False
type Name  = [Char]
```

define un nuevo tipo de datos `Boolean` con aridad 0, esto es, True y False son términos constructores (*constantes*) y sólo dos valores `True` y `False` (Curry tiene un cierto número de tipos básicos predefinidos, tales como `Bool e Int`). La segunda es una *declaración de tipo sinónimo* `Name`, la cual representa una lista de caracteres. Esta última declaración tiene la siguiente forma general:

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

la cual introduce un nuevo tipo constructor T de aridad n, donde $\alpha_1 \ldots \alpha_n$ son variables de tipo distinto y τ es una expresión contruida a partir de los constructores y las variables $\alpha_1 \ldots \alpha_n$. El tipo $(T\ \tau_1 \ldots \tau_n)$ es equivalente al tipo $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}(\tau)$, o sea, a la expresión τ donde cada α_i es reemplazada por τ_i. Así pues, un tipo sinónimo y su definición son simpre intercambiables y no influyen sobre los tipos de un programa.

En Curry la forma general para la declaración de tipos es:

$$\textbf{data}\ T\ \alpha_1 \ldots \alpha_n\ =\ C_1\ \tau_{11} \ldots \tau_{1n_1}\ |\ \ldots\ |\ C_k\ \tau_{k1} \ldots \tau_{kn_k}$$

donde ésta introduce un nuevo *constructor de tipo* T de aridad n y k nuevos *constructores de datos* $C_1 \ldots C_k$, donde cada C_i tiene el tipo:

τ_{i1} `->` $\ldots$ `->` τ_{in_i}`->` $T\ \alpha_1 \ldots \alpha_n$
`con` $i\ =\ 1, \ldots\ ,\ k.$

Cada τ_{ij} es una *expresión de tipo* construida a partir de las *variables de tipo* $\alpha_1 \ldots \alpha_n$ y algunos constructores de tipo. Puesto que Curry es un lenguaje de orden superior, los tipos de las funciones (esto es, constructores y operaciones) están escritas en su forma currificada τ_1 `->` τ_2 $\ldots$ `->` τ_n `->` τ donde τ no es un tipo funcional. En este caso, n es la llamada *aridad* de la función. Por ejemplo, las siguientes declaraciones de función:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
inc x = x + 1
```

definen la típica función `map` que aplica la función `f` al primer elemento de una lista, después aplica recursivamente `f` al resto de la lista. De esta forma la llamada a función `map inc [1,2]` produce `[2,3]` como resultado.

La evaluación perezosa nos permite definir estructuras de datos infinitas las cuales pueden ser ejecutadas en un tiempo finito. Por ejemplo,

```
from :: Int -> [Int]
from x =  x: from (x + 1)
```

es una función que toma un entero y devuelve una lista de enteros infinita. Sin embargo, la llamada `take 5 (from 3)` produce la lista finita `[3,4,5,6,7]`. `take` está definida como sigue:

```
take :: Int -> [a] -> [a]
take n l = if n<=0 then []
                   else takep n l
  where takep _ [] = []
        takep n (x:xs) = x : take (n-1) xs
```

La siguiente definición de tipo de datos:

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

define el tipo de datos polimórfico `List a`. Aquí, "`:`" representa un operador infijo, o sea que, "`a:List a`" es otra notación para "`(:) a (List a)`".

Las listas están predefinidas en Curry, donde la notación "`[a]`" se usa para denotar tipos de listas (en lugar de "`List a`"). Las notaciones más usuales para indicar listas están soportadas, esto es, `[0,1,2]` es una abreviación de `0:(1:(2:[]))` (véa [Han06] para mayor detalle).

1.4. Evaluación parcial y compilación

En esta sección introducimos los conceptos de compilador e intérprete y su relación con la evaluación parcial. En la literatura [Hud98, JGS93] se mencionó la posibilidad de que, mediante la evaluación parcial se puede llevar a cabo la compilación de programas y con ello mejorar su eficiencia. Sin embargo lograr dicha mejora exige aplicar principios, métodos y técnicas de evaluación parcial más avanzadas, lo que constituye una motivación fundamental para el desarrollo de la presente tesis.

Tal como un intérprete de los idiomas inglés al español le da una semántica del inglés al español en una traducción simultánea; así, dar una semántica a un **lenguaje** de programación es asignar sistemáticamente un significado a cada programa escrito en ese **lenguaje**. En este documento, tal como en [JGS93], asignamos el significado de un programa escrito en un lenguaje S mediante la construcción de un *intérprete* para S-programas, es decir, un programa para ejecutar S-programas. Esto provee de una semántica muy concreta orientada al ordenador. Por muchos años los intérpretes han sido usados para definición de lenguajes, e.g., Lisp y el cálculo lambda [JGS93].

Si definimos que L-programas indica el conjunto de programas sintácticamente correctos en un lenguaje L, si `p` representa un programa escrito en L y D representa un conjunto fijo de datos para L-programas, podemos definir que el significado de un programa $\mathtt{p} \in$ L-programas es:

$$[[\mathtt{p}]]_{\mathrm{L}} : \mathtt{entrada} \rightarrow \mathtt{salida}$$

Donde $[[\_]]_{\mathrm{L}}$ es la función del significado de un programa y el subíndice L indica cómo debe ser interpretado `p`. Entonces para $n \geq 0$, tenemos que:

$$\mathtt{salida} = [[\mathtt{p}]]_{\mathrm{L}}[\mathtt{in}_1, \mathtt{in}_2, \ldots, \mathtt{in}_n]$$

donde `salida` resulta de ejecutar `p` con los valores de entrada $\mathtt{in}_1, \mathtt{in}_2, \ldots, \mathtt{in}_n$ y es de tipo D, esto es, $D \to D^* \to D$. Por lo tanto podemos decir que el significado del programa `p` queda establecido por la función de entrada-salida que computa.

Sea L un lenguaje (de implementación) y S un lenguaje (fuente). Un intérprete `int` $\in$ L-programas para un lenguaje S el cual tiene dos argumentos de entrada: un *programa fuente* `p` $\in$ S-programas a ser ejecutado, y los datos de entrada `d` del programa fuente de tipo D, es decir que `d` $\in D$. Ejecutando el intérprete con entradas `p` y `d` sobre una L-máquina debe producir el mismo resultado que ejecutar `p` con entrada `d` sobre una S-máquina. De manera más precisa, el L-programa `int` es un *intérprete* para S si para cada programa `p` $\in$ S-programas y cada dato `d` $\in D$, se cumple que,

$$[[\mathtt{p}]]_{\mathrm{S}}\ \mathtt{d} = [[\mathtt{int}]]_{\mathrm{L}}\ [\mathtt{p}, \mathtt{d}]$$

Podemos usar el simbolo $\mathtt{I}_{\mathrm{L}}^{\mathrm{S}}$

$$\mathtt{I}_{\mathrm{L}}^{\mathrm{S}} = \{\mathtt{int} \mid \forall \mathtt{p}, \mathtt{d}.\ [[\mathtt{p}]]_{\mathrm{S}}\ \mathtt{d} = [[\mathtt{int}]]_{\mathrm{L}}\ [\mathtt{p}, \mathtt{d}]\ \}$$

para denotar el conjunto de todos los intérpretes para S escritos en L.

Por otro lado, si tenemos que T es un lenguaje objeto. Un compilador `comp` $\in$ L-programas traduce de S a T y tiene como entrada: un *programa fuente* `p` $\in$ S-programas a ser compilado. Entonces, ejecutar el compilador con entrada `p` (sobre una L-máquina) debe producir un `pobjeto`, tal que ejecutar `pobjeto` en una T-máquina tiene el mismo efecto que ejecutar `p` sobre una S-máquina. Más formalmente, el L-programa `comp` es un *compilador* de S a T si para cada programa `p` $\in$ S-programas y cada `d` $\in D$, se cumple que:

$$[[\mathtt{p}]]_{\mathrm{S}}\ \mathtt{d} = [[\ [[\mathtt{comp}]]_{\mathrm{L}}\ \mathtt{p}]]_{\mathrm{T}}\ \mathtt{d}$$

Podemos usar el siguiente símbolo $\mathtt{C}_{\mathrm{L}}^{\mathrm{S}\to\mathrm{T}}$

$$\mathtt{C}_{\mathrm{L}}^{\mathrm{S}\to\mathrm{T}} = \{\mathtt{comp} \mid \forall \mathtt{p}, \mathtt{d}.\ [[\mathtt{p}]]_{\mathrm{S}}\ \mathtt{d} = [[\ [[\mathtt{comp}]]_{\mathrm{L}}\ \mathtt{p}]]_{\mathrm{T}}\ \mathtt{d}\ \}$$

para denotar el conjunto de compiladores escritos en L y que traducen de S a T.

Es posible compilar por medio de la especialización de un intérprete que ejecute un sólo un programa fijo, produciendo un programa objeto en el lenguaje de salida del evaluador parcial tal que:

$$\texttt{pobjeto} = [[\texttt{mix}]][\texttt{int, pfuente}]$$

Donde el programa `pobjeto` se espera sea más rápido que interpretar `pfuente` ya que muchas acciones del interprete dependen sólo de `pfuente` y por lo tanto pueden ser precalculadas. En general, el programa `pobjeto` será una mezcla de `int` y `pfuente` conteniendo partes derivadas de ambos [JGS93]. La compilación por evaluación parcial siempre produce programas objeto correctos lo que se verifica como sigue:

$$\begin{array}{lcll}
\texttt{salida} & = & [[\texttt{pfuente}]]_S\ \texttt{entrada} & \\
& = & [[\texttt{int}]]_L\ [\texttt{pfuente}, \texttt{entrada}] & \text{definición de un intérprete} \\
& = & [[\ \underbrace{[[\texttt{mix}]]_L\ [\texttt{int, pfuente}]}_{\texttt{pobjeto}}\ ]]_L\ \texttt{entrada} & \text{ecuación del } \textbf{mix} \\
& = & [[\texttt{pobjeto}]]_T\ \texttt{entrada} & \text{nombrando el programa residual como: } \texttt{pobjeto}
\end{array}$$

Las ultimas tres igualdades concuerdan respectivamente con las definiciones de un intérprete, `mix`[6], y `pobjeto`. Por lo tanto el efecto neto ha sido trasladar de lenguaje S a T. La ecuación;

$$\texttt{pobjeto} = [[\texttt{mix}]]_L\ [\texttt{int, pfuente}]$$

es comúnmente llamada *primera proyección de Futamura* reportada originalmente en [Fut71].

Realizar la *compilación por evaluación parcial* a partir de un intérprete obedece a las siguientes razones:

- En la práctica los intérpretes son pequeños, más fáciles de entender y más fácil de depurar que los compiladores.

[6]Un evaluador parcial también es conocido con el nombre de `mix`, véa la sección 1.1.2.

- Un intérprete representa una semántica operacional (una forma de bajo-nivel), y puede servir como una definición de un lenguaje de programación.

- La pregunta de compilador correcto es completamente eludible, ya que el compilador siempre será confiable respecto del intérprete a partir del cual fue generado.

Ahora mostramos que el `mix` (evaluador parcial) también puede generar compiladores independientes:

$$\texttt{compilador} = [[\texttt{mix}]]\ [\texttt{mix}, \texttt{int}]$$

se trata de un programa escrito en L que cuando se aplica a `pfuente`, produce un `pobjeto`, y de esta forma es un compilador de lenguaje S a L escrito en L. La verificación se ve directa a partir de la ecuación del `mix`:

$$\begin{aligned} \texttt{pobjeto} &= [[\texttt{mix}]]\ [\texttt{int},\ \texttt{pfuente}] \\ &= [[\ \underbrace{[[\texttt{mix}]]\ [\texttt{mix}, \texttt{int}]}_{\texttt{compilador}}\ ]]\ \texttt{pfuente} \\ &= [[\texttt{compilador}]]\ \texttt{pfuente} \end{aligned}$$

La ecuación $\texttt{compilador} = [[\texttt{mix}]]\ [\texttt{mix}, \texttt{int}]$ es conocida como la *segunda proyección de Futamura*. El compilador genera versiones especializadas del intérprete `int`. Operacionalmente, la construcción de un compilador de esta forma es difícil de entender porque involucra la auto aplicación —utilizando el `mix` para especializarse a si mismo.

Es conocido desde 1971 [Fut71] que el principio de transformación de programas: *evaluación parcial* o *especialización de programas* está estrechamente conectado con la compilación. En particular un especializador de programas puede ser usado para compilar, dada una definición interpretativa de un lenguaje de programación como dato de entrada. Adicionalmente un programa especializador puede generar un compilador e incluso un generador de compiladores, siempre que éste sea autoaplicable.

Futamura fue el primer investigador que comprendió que la autoaplicación de un evaluador parcial puede en principio lograr la generación

de compiladores. Consecuentemente las ecuaciones que describen compilación, generación de un copilador y generación de compiladores, son ahora llamadas *proyecciones de Futamura.*

$$\begin{aligned} \texttt{pobjeto} &= [[\texttt{mix}]]_{\mathrm{L}}\ [\texttt{int},\ \texttt{programa fuente}] \\ \texttt{compilador} &= [[\texttt{mix}]]_{\mathrm{L}}\ [\texttt{mix},\ \texttt{int}] \\ \texttt{cogen} &= [[\texttt{mix}]]_{\mathrm{L}}\ [\texttt{mix},\ \texttt{mix}] \end{aligned}$$

Aunque es fácil de verificar, se debe admitir que la tracendencia intuitiva de estas ecuaciones es difícil de ver [JGS93].

1.5. Lenguajes de dominio específico (DSLs)

Un lenguaje de dominio específico (DSL) es un lenguaje de programación diseñado para un dominio de aplicación en particular.

Las características de un DSL efectivo son: la capacidad de desarrollar programas de aplicación completos de manera rápida y efectiva. Ejemplos comunes de DSLs incluyen LEXX y YACC para generar analizadores léxicos y sintácticos, PERL para manipulación de Texto, VHDL para descripción de hardware, TEX y LATEX para preparación de documentos, HTML y SGML para especificar reglas de etiquetado de documentos. Tcl/TK para elaborar interfases gráficas GUIs, VRML y OpenGL para gráficos en 3D, Mathematica y Maple para realizar computación simbólica, AutoLisp y AutoCAD para diseño asistido por computadora [Hud98].

Una de las principales razones por la cual los proyectos de desarrollo de software fallan; es la falta de comunicación entre los usuarios, quienes conocen realmente el domino del problema, y los desarrolladores quienes diseñan e implementan la áplicación. Los usuarios entienden la terminología del dominio y hablan un lenguaje que puede ser muy raro a los desarrolladores; no es de extrañar que la comunicación se puede romper en el inicio del ciclo de vida del proyecto. Un lenguaje de dominio específico conecta la brecha semántica entre los usuarios y los desarrolladores de la aplicación fomentando una mejor comunicación a través de un vocabulario compartido. Las abstracciones que el DSL ofrece equiparan la sintaxis y la semántica del dominio del problema. Como resultado, los

usuarios pueden involucrarse en la verificación de las reglas de la aplicación a través del ciclo de vida completo del proyecto [Gho11].

Un método simple para elevar el nivel de abstracción de los programas, es produciendo interfaces que componen las llamadas librerías de un lenguaje. Un método más sofisticado consiste en desarrollar un nuevo lenguaje de programación que incluya principios y abstracciones de una forma estable; que estén inspirados en objetivo del dominio de la aplicación. Estos lenguajes de programación son conocidos como *Domain Specific Languages* (DSLs) [Hud98] y proporcionan una poderosa solución para realizar abstracciones de alto nivel.

Hemos desarrollado un lenguaje de dominio específico para máquinas de Control Numérico Computarizado (CNC). Los programas CNC son series de código que consisten de instrucciones semejantes al lenguaje ensamblador, consecuentemente, son programas de bajo nivel que requieren programadores especializados [ARS11]. Programar en CNC no es una tarea fácil ya que su codificación representa un lenguaje de bajo nivel que carece de sentencias de control, procedimientos, y muchas otras ventajas de modernos lenguajes de alto nivel [AOSV04]. En la actualidad, las máquinas CNC han llegado a ser la base de muchos procesos industriales. Las máquinas CNC incluyen robots, lineas de producción y todas aquellas máquinas que son controladas por dispositivos digitales. Típicamente, las máquinas CNC tienen una *Unidad de Control de Máquina* (MCU) el cual ingresa un programa CNC y controla el comportamiento y los movimientos de todos los componentes de la máquina. Consideramos que elevar el nivel de abstracción del lenguaje CNC nos permitirá

1. Poder desarrollar programas CNC más amigablemente ya que utilizamos abstracciones de alto nivel.

2. Ser más productivos puesto que proponemos una librería de funciones, cada una de las cuales encapsulará muchas instrucciones CNC simples.

Presentamos nuestro lenguaje como un conjunto de funciones que encapsulan instrucciones CNC, que a su vez pueden generar código CNC con el fin de producir aplicaciones del mundo real.

Uno de los principales incovenientes de construir una librería modular es la sobrecarga de interpretación que representa cada módulo del DSL, dicha desventaja podría acarrear como resultado un intérprete aparentemente impráctico para cualquier DSL realista. Hudak [Hud98] propone dos niveles de especialización para resolver éste problema, uno de ellos es especializar un intérprete concreto con respecto a un programa fuente y producir un *programa instrumentado*, esto es, aplicar la primera proyección de Futamura (Compilación). En nuestro caso aplicamos la EP directamente a los módulos del DSL para CNC, es decir, definimos una llamada a alguna de las funciones del DSL desarrollado y nuestro evaluador parcial se encarga de especializar el DSL de acuerdo a dicha llamada. Puesto que el DSL maneja números de punto flotante hemos dado algunas características básicas al evaluador parcial para especializar dichas cifras.

La sección actual y la anterior son muestras claras de aplicaciones de la evaluación parcial de programas, sin embargo tenemos que hacer patente los principios y técnicas de optimización implementadas, y que se han especificado principalmente en el capítulo 4, para lograr realizar tales aplicaciones.

1.6. Objetivos de la tesis

La relación de la evaluación parcial con la compilación de programas, así como la posibilidad de mejorar el desempeño de los DSLs materializaron una inspiración y un reto para llevar a cabo la presente tesis, cuya idea principal será mejorar el esquema de evaluación parcial *offline* dirigida por *narrowing*. Toda vez que la primera aproximación carecía de las implementaciones tanto de un proceso de análisis de tiempo de enlace (BTA) como de un análisis de terminación más refinado, ambas técnicas persiguen un perfeccionamiento del procedimiento de anotación y que culmina con una especialización más precisa.

1.6.1. Objetivo general

El principal objetivo de la tesis es mejorar el desempeño de la primera aproximación de evaluación parcial *offline* dirigida por *narrowing* [Ram07], para lograr esto resulta natural perfeccionar el procedimiento de anotación con el propósito de obtener programas mejor especializados y por ende más rápidos. Así pues debemos desarrollar técnicas de mejora como el análisis de los grafos *size-change* (SCA) y el BTA, así como extender el especializador para procesar *built-ins* y *constraints*. Al aplicar tales técnicas a esta aproximación es posible especializar de forma más precisa, así como evaluar programas más realistas tales como intérpretes y DSLs.

1.6.2. Contribuciones

En esta investigación hemos perfeccionado la caracterización de los sistemas de reescritura *no-crecientes* de [RSV05a] adoptando el uso del principio de terminación de los grafos *size-change* [LJBA01], el cual aproxima los cambios de tamaño entre los argumentos de función cuando van de una llamada a otra. En particular, usamos la información del resultado del análisis *size-change* para identificar una forma específica de cuasi-terminación, i.e., que sólo pueden ser producidas en una computación un número finito de *llamadas a función* diferentes (módulo renombramiento de variables). Para este mismo propósito, utilizamos el resultado de un análisis de tiempo de enlace estándar (*binding-time analysis* —BTA) para tener disponible la información sobre qué argumentos de función son *estáticos* y (por lo tanto *ground*) o *dinámicos*. Cuando la información recabada de la combinación de los grafos *size-change* y del BTA no nos permiten inferir que los sistemas de reescritura cuasi-terminan, procedemos como en [RSV05a] y anotamos los subtérminos problemáticos para ser generalizados en tiempo de evaluación parcial [ARSV06a].

Un BTA monovariante y adaptación del principio *size-change*. En la Figura 1.3 se muestra el esquema completo de nuestra implementación, donde vemos que el proceso encerrado en la caja punteada incluye tanto el BTA simple (monovariante) como el análisis *size-change*,

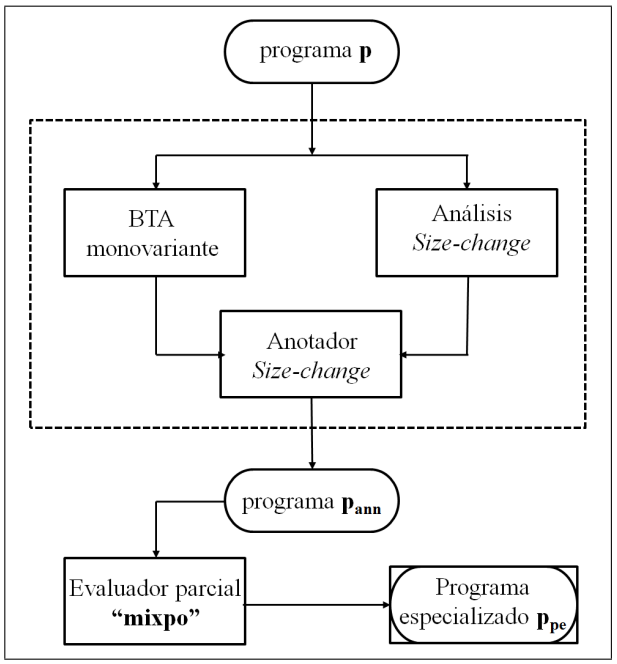


Figura 1.3: Esquema del evaluador parcial *offline*.

ambos procesos toman como entrada el programa (**p**) a especializar y entregan sus respectivos resultados al propio proceso de anotación, el cual produce el programa anotado ($\mathbf{p_{ann}}$) y lo entrega al especializador (**mixpo**). Las derivaciones que se computan en el proceso de especialización siguen una ligera extensión del cálculo RLNT puesto que, dicha semántica, originalmente sólo especializaba de forma *online*, ahora al agregar anotaciones a las funciones y a sus argumentos, el calculo RLNT debe extenderse para soportar dichas anotaciones.

Aproximaciones de anotación. Hemos implementado dos aproximaciones de anotación a las que hemos denominado *híbrida*, por realizar una prueba simple de igualdad módulo renombramiento de variables al aplicar el operador de desplegado en el procedimiento de especialización para evitar la no terminación del desplegado. Este inconveniente se supero con la implementación de la segunda aproximación de anotación denominada *offline pura* o *100 % offline* en la cual el esquema de anotación

es más intuitivo al usuario, contrastando con la primera aproximación donde se maneja un sólo tipo de anotación, i.e., generalización. En la segunda aproximación se manejan tres tipos: despeglar sin restriciones, no desplegar y generalizar, en este caso el especializador esta dirigido completamente por las anotaciones al momento de aplicar el operador de desplegado (ver de la subsección 4.7.1 hasta la sección 4.9). Aunque propiamente no se ha logrado una mejora sustancial en el desempeño general de la especialización con repecto al evaluador parcial con anotación híbrida [ARTV07].

Aproximación de NPE con programas de orden superior. También en este trabajo hemos introducido una aproximación al primer evaluador parcial *offline* dirigido por *narrowing* que trata con programas de orden superior y produce especializaciones polivariantes, tal aproximación está sustentada en dos diferentes transformaciones: un algoritmo de desfuncionalización que hace explícita la información de orden-superior del programa tanto como sea posible, después al programa desfuncionalizado se le aplica una transformación polivariante para obtener un programa doblemente transformado, luego aplicamos nuestro BTA monovariante sobre el programa doblemente transformado con el fín de obtener la misma información (de tiempo de enlace) que aquella obtenida a partir de un BTA polivariante sobre el programa original (de orden superior). Una vez calculada la información del BTA polivariante aproximado se realiza el análisis *size-change* (SCA) sobre el mismo programa doblemente transformado y con ambos resultados tanto del BTA como del SCA se realiza la anotación. Ya con el programa anotado se lleva a cabo el proceso de especialización [ARTV09].

Aproximación a la compilación por EP de programas LF. También se tiene como objetivo extender la implementación del evaluador parcial *pure offline* o *100 % offline* para que acepte especializar en buena medida el lenguaje Curry [Han06] incluyendo *built-ins* y *constraints*, y enseguida realizar un metaintérprete de un segmento de Curry para llevar a cabo una aproximación a la autoaplicación, es decir, la evaluación parcial del metaintérprete dando como segundo parámetro un programa

Curry de primer orden, con lo que se obtendría la primera aproximación a la compilación por evaluación parcial de programas lógico funcionales.

Desarrollo de un DSL para generar Programas CNC y su Especialización. Otra aplicación que se presenta es el desarrollo de un DSL para generar programas de Control Numérico Computarizado. Desde el punto de vista de evaluación parcial hemos considerado varios *built-ins* en nuestro evaluador parcial para que soporte especializar programas generados por el DSL, puesto que la aplicación trata principalmente con números de punto flotante y Curry maneja nombres de funciones aritméticas diferentes para este tipo cifras, así pues se han dado algunas capacidades al evaluador parcial para que procese algunos casos de punto flotante [ARS11].

1.7. Organización de la tesis

La presente tesis consta de 8 capítulos. El presente capítulo trata, como su nombre indica, de introducir una serie de conceptos relacionados todos con el tema de la evaluación parcial de programas, específicamente con la evaluación parcial *offline* dirigida por *narrowing* sobre un lenguaje multiparadigma lógico funcional, trata los objetivos de la tesis y relaciona asimismo aplicaciones de EP como la compilación y especialización de un DSL. El capítulo 2 resume los conceptos técnicos básicos y la notación que se utilizará en el documento que soportan la teoría del tema de evaluación parcial. En el capítulo 3 se define la primera aproximación a la evaluación parcial *offline* dirigida por *narrowing*, especifica más detalladamente el concepto de evaluación parcial e identifica sobre qué tipo de programas trabaja y el alcance de dicha aproximación, garantiza terminación sobre un tipo específico de programas, conecta el intervalo semántico entre las evaluaciones parciales *online* y *offline*. Finalmente define a detalle el método de evaluación parcial *offline* dirigido por *narrowing*.

En el capítulo 4 se presenta una mejora a la primera aproximación de evaluación parcial *offline* descrita en el capítulo 3. Se engloba dentro de la primera fase del evaluador parcial *offline*, un análisis de tiempo de enlace estándar monovariante y se especifica e implementa un método

de terminación adaptado más eficiente. El resultado de ambos procedimientos se usan para establecer un método de anotación más eficiente, lo que redunda en lograr una mejor especialización de programas. Además se muestra el resultado que compara sendas implementaciones de EP *offline* llamadas híbrida y *offline* pura con un EP *Online*.

El capítulo 5 trata sobre la especialización de programas de orden superior a los que se aplica una doble transformación, primero aplicamos una desfuncionalización al programa de orden superior, luego al programa desfuncionalizado se aplica una transformación polivariante, es decir, generamos una copia de cada definición de función del programa que posea valores de tiempo de enlace diferentes. Después el programa doblemente transformado se evalúa parcialmente. Aunque el tamaño del código de los programas utilizados crece, la ejecución de los programas politransformados y especializados resulta más rápida que los correspondientes programas especializados sin transformación polivariante.

En el capítulo 6, describimos los primeros experimentos en la especialización de intérpretes, es decir, obtenemos la compilación utilizando evaluación parcial *offline* dirigida por *narrowing*. Dicho en otras palabras aplicamos la primera proyección de Futamura. En general, escribimos un intérprete en Curry con funciones de primer orden y le damos de manera estática un programa a especializar. Dicho interprete es especializado con el evaluador parcial *offline* puro, extendido además con la capacidad de procesar varios *built-ins* y *constraints*.

En el capítulo 7 presentamos el desarrollo de un DSL que genera programas para máquinas Control Numérico Computarizado (CNC) por medio de una librería de funciones desarrollada en Curry. Como aplicación hemos extendido el evaluador parcial *offline* puro con algunos *built-in*s para que sea capaz de especializar algunos casos de punto flotante. Lo anterior con el objetivo de especializar la librería a partir de una llamada a las funciones del DSL desarrollado.

Finalmente en el capítulo 8, mostramos las conclusiones y una descripción de las aportaciones de esta investigación, asimismo se mencionan algunas líneas de trabajo futuro que pueden dar continuidad al presente trabajo.

Capítulo 2

Preliminares

Los sistemas de reescritura de términos (del inglés *term rewriting systems* [BN98]) ofrecen un marco apropiado para modelar el componente de primer orden de muchos lenguajes de programación funcionales y lógico funcionales[1]. Puesto que nosotros trabajamos con conceptos fundamentales de la programación lógico funcional, en el resto de este documento seguimos los conceptos estándar de la reescritura de términos para desarrollar nuestros resultados. Una gran parte de conceptos y definiciones se tomaron de [Ant92, Han94, AEH00, AFJV03].

2.1. Signaturas y términos

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

[1] No obstante, las características de orden superior (*higher-order*) pueden ser modeladas utilizando un operador de aplicación explícito, esto es, por desfuncionalización [Rey98].

Para denotar el dominio de *términos* y *términos constructores* usamos $\mathcal{T}(\mathcal{F},\mathcal{V})$ y $\mathcal{T}(\mathcal{C},\mathcal{V})$ (ambos posiblemente contienen variables de $\mathcal{V}$), respectivamente. Con $\mathcal{V}ar(t)$ denotamos al conjunto de variables que aparecen en un término t. Se denomina variable *fresca* a una variable nueva que no ha sido empleada con anterioridad. Un término t es *básico* (o *ground*) si $\mathcal{V}ar(t) = \varnothing$. Un término t es una variante de t' si ambos son iguales módulo renombramiento de variables. Un término es *lineal* si no contiene ocurrencias repetidas de ninguna variable. Una lista finita de objetos $o_1, \ldots, o_n$ se representa con $\overline{o_n}$.

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

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

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

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

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

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

como sigue:

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

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

2.2. Sustituciones

Una *sustitución* σ es una correlación de variables a términos indicada por $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ donde $\sigma(x_i) = t_i$ para $i = 1, \ldots, n$ y tal que su dominio $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ es finito. El codominio de σ esta determinado por el conjunto $\mathcal{R}an(\sigma) = \{\sigma(x) \mid x \in \mathcal{D}om(\sigma)\}$. El rango de σ esta formado por las variables que ocurren en el codominio, i.e. el conjunto $\{y \in \mathcal{V} \mid y \in \mathcal{V}ar(\mathcal{R}an(\sigma))\}$. La sustitución identidad esta indicada por id. Una sustitución σ es un *constructor*, si $\sigma(x)$ es un término constructor para todo $x \in \mathcal{D}om(\sigma)$. Un término t' es una *instancia* del término t si hay una sustitución σ con $t' = \sigma(t)$. Un *unificador* de dos términos s y t es una sustitución σ con $\sigma(s) = \sigma(t)$. La *composición* de dos sustituciones σ y τ se define por $(\sigma \circ \tau)(x) = \sigma(\tau(x))$ para toda $x \in \mathcal{V}$.

Dada una sustitución θ y un conjunto de variables $V \subseteq \mathcal{V}$, denotamos con $\theta_{\restriction V}$ la sustitución obtenida a partir de θ restringiendo su dominio a V. Escribimos $\theta = \sigma[V]$ si $\theta_{\restriction V} = \sigma_{\restriction V}$ y $\theta \leq \sigma[V]$ denota la existencia de una sustitución γ tal que $\gamma \circ \theta = \sigma[V]$. Un unificador σ es el *unificador más general*: mgu (del inglés *most general unifier*), si $\sigma \leq \sigma'[\mathcal{V}]$ para cualquier otro unificador σ'.

2.3. Sistemas de reescritura de términos

Se llama *Sistema de Reescritura de Términos* (SRT)[2] al conjunto de reglas de reescritura (o ecuaciones orientadas) de la forma $l \to r$ tal que $l \notin \mathcal{V}$ y r es un término cuyas variables aparecen en l, i.e.,

[2]De acuerdo a su traducción del concepto en inglés: *Term Rewriting System* (TRS).

$\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$); los términos l y r son llamados parte izquierda y parte derecha de la regla, respectivamente. Dado un SRT $\mathcal{R}$ determinado sobre una signatura Σ, los símbolos $\mathcal{D}$ *definidos* son los símbolos raíz de las partes izquierdas de las reglas, i.e., $\mathcal{D} = \{root(l) |\ l \to r \in \mathcal{R}\}$ son los símbolos de función del SRT, y los *constructores* son $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Nos limitamos a signaturas y SRTs finitos.

Un SRT $\mathcal{R}$ está *basado en constructores* si la parte izquierda de sus reglas tienen la forma $f(s_1, \ldots, s_n)$ donde s_i son términos constructores, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, para todo $i = 1, \ldots, n$. El conjunto de variables que aparecen en un término t están indicadas por $\mathcal{V}ar(t)$. Un término t es *lineal* si cada variable de $\mathcal{V}ar(t)$ ocurre sólo una vez en t. Un SRT $\mathcal{R}$ es *lineal por la izquierda* (respectivamente *lineal por la derecha*) si l (respectivamente r) es lineal para todas las reglas $l \to r \in \mathcal{R}$. La *definición* de f en $\mathcal{R}$ es el conjunto de reglas en $\mathcal{R}$ cuyo símbolo raíz en la parte izquierda es f. Una función $f \in \mathcal{D}$ es lineal por la izquierda (respectivamente lineal por la derecha) si las reglas de su definición son lineales por la izquierda (respectivamente lineales por la derecha).

El símbolo en cabeza (raíz) de un término t esta indicado por $root(t)$. Un término t esta *encabezado por operación* (*operation-rooted*) (respectivamente *encabezado por constructor*(*constructor-rooted*)) si $root(t) \in \mathcal{D}$ (respectivamente si $root(t) \in \mathcal{C}$).

Los SRTs inductivamente secuenciales [Ant92] son una subclase de SRTs lineales por la izquierda basados en constructores. Esencialmente, un SRT es *inductivamente secuencial* cuando todas sus operaciones están definidas por reglas de reescritura que, recursivamente, hacen una distinción *case* sobre sus argumentos de manera análoga a la inducción (de la estructura) de un tipo de datos. La clase inductivamente secuencial no es una restricción para la programación. De hecho, el componente de primer orden (*first-order*) de varios programas (lógico) funcionales escritos en, e.g., Haskell, ML o Curry, es inductivamente secuencial. Además, la clase de programas inductivamente secuenciales provee una forma de cómputo inmejorable tanto para programación funcional como lógico funcional [Ant92, AEH00].

Ejemplo 2.1 *Consideremos las siguientes reglas las cuales definen la función "$\leqslant$" (menor o igual) sobre los números naturales (En este caso*

los números naturales se construyen a partir de los símbolos constructores ***zero*** *y* ***succ****):*

$$\begin{array}{rcll} zero & \leqslant & y & \rightarrow \ True \\ succ(x) & \leqslant & zero & \rightarrow \ False \\ succ(x) & \leqslant & succ(y) & \rightarrow \ x \leqslant y \end{array}$$

Esta función es inductivamente secuencial tal que sus partes izquierdas pueden organizarse jerárquicamente en una estructura case como sigue:

$$\boxed{n} \leqslant m \Longrightarrow \begin{cases} zero \leqslant m \\ succ(x) \leqslant \boxed{m} \Longrightarrow \begin{cases} succ(x) \leqslant zero \\ succ(x) \leqslant succ(y) \end{cases} \end{cases}$$

donde los argumentos en las cajas indican una distinción case (esto es similar al concepto de árbol definicional[3] *en [Ant92]).*

2.4. Semántica de los SRTs

La evaluación de términos con respecto a un SRT está formalizada con el concepto de *reescritura*. Un *paso de reescritura* es la aplicación de una regla de reescritura a un término, i.e., $t \rightarrow_{p,R} s$ si existe una posición $p \in \mathcal{P}os(t)$, una regla de reescritura $R = (l \rightarrow r)$ y una sustitución σ tal que $t|_p = \sigma(l)$ y $s = t[\sigma(r)]_p$ (frecuentemente p y R se omiten en la notación cuando están claros en el contexto). La parte izquierda instanciada $\sigma(l)$ de una regla $l \rightarrow r$ se le llama *redex*[4]. Un redex $t|_p$ de un término t es un *redex más externo* si no existe otro redex $t|_q$ de t con $q < p$. El paso de reescritura aplicado sobre un redex más externo se denomina *paso de reescritura más externo*. La reescritura más externa es la base operacional de los lenguajes funcionales perezosos. A la acción de emparejar un subtérmino $t|_p$ para que ajuste con la parte izquierda instanciada $\sigma(l)$ de alguna regla tal que $t|_p = \sigma(l)$ se le llama *emparejamiento de patrones* (de la expresión en inglés *pattern matching*).

Un término t es llamado *irreducible* o en *forma normal* si no hay término s tal que $t \rightarrow s$. Denotamos con $t\!\downarrow$ la forma normal de t. Un

[3] Véase el concepto de árboles definicionales en la sección 2.4.2.

[4] Del inglés *red*ucible *exp*ression, esto es, una expresión reducible [Ram07, Vid04].

término t está en *forma normal en cabeza* si no se puede reducir a un redex.

Indicamos por medio de $\rightarrow^+$ la cerradura transitiva de la relación binaria $\rightarrow$, y con $\rightarrow^*$ su cerradura reflexiva y transitiva. Dado un SRT $\mathcal{R}$ y un término t, decimos que t *se evalúa* a s sii[5] $t \rightarrow^* s$ y s está en forma normal. Un SRT $\mathcal{R}$ se dice *noetheriano* (o terminante) si no hay una secuencia infinita de la forma $t_0 \rightarrow_{\mathcal{R}} t_1 \rightarrow_{\mathcal{R}} \ldots$; $\mathcal{R}$ es *confluente* si para todo término t tal que $t \rightarrow^*_{\mathcal{R}} s_1$ y $t \rightarrow^*_{\mathcal{R}} s_2$, entonces existe un término s tal que $s_1 \rightarrow^*_{\mathcal{R}} s$ y $s_2 \rightarrow^*_{\mathcal{R}} s$. Un SRT que es noetheriano y confluente se llama *canónico* o *completo*.

2.4.1. *Narrowing*

Los programas *lógico* funcionales varían de los programas puramente funcionales principalmente en que las llamadas a función pueden contener variables *libres*. Para evaluar términos con variables libres, generalmente es necesario sustituir dichas variables por instancias, esto es, por términos apropiados con el fín de que sea posible aplicar un paso de reescritura. Esto se logra utilizando unificación en lugar de emparejamiento en el paso de reescritura lo cual es conocido como *narrowing*. Por lo tanto en un paso de *narrowing* unificamos un subtérmino (no variable) del objetivo[6] solicitado con la parte izquierda de una regla y después, una vez realizada la unificación de la regla instanciada, reemplazamos por la parte derecha de la regla instanciada y continuamos con la reducción [Han94]. Para ser más precisos decimos que un término t es *reducido por narrowing* a un término t' si:

1. p representa una posición en t tal que $t|_p$ es un subtérmino no variable (es decir, $t|_p \notin \mathcal{V}$),
2. $l \rightarrow r$ es una nueva variante de una regla del programa,
3. la sustitución σ es el mgu[7] de $t|_p$ y l,
4. $t' = \sigma(t[r]_p)$.

[5] En adelante abreviamos *si y sólo si* con sii.
[6] Del inglés: *goal*.
[7] De las siglas de la expresión en inglés: *most general unifier*.

Formalmente,

Definición 1 (paso de *narrowing*) $t \leadsto_{p,R,\sigma} t'$ *es un* paso de narrowing *sii* p *es una posición no variable de* t *y* $\sigma(t) \rightarrow_{p,R} t'$

Frecuentemente omitimos p, R y/o σ cuando la posición y la regla son claros en el contexto. La sustitución σ es comúnmente *el unificador mas general*[8] de $t|_p$ y de la parte izquierda de (una variante de) R, restringiendo su dominio a $\mathcal{V}ar(t)$. Tal como en los procedimientos de demostración de programación lógica, asumimos que las reglas del SRT siempre contienen variables frescas si son utilizadas en un paso de *narrowing*. Denotamos por medio de $t_0 \leadsto^*_\sigma t_n$ a una secuencia de pasos *narrowing* $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ con $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (si $n = 0$ entonces $\sigma = id$).

A causa de la presencia de variables libres, un término puede ser reducido a diferentes valores después de sustituir las instancias de dichas variables por diferentes términos. Dada una derivación de *narrowing* $t_0 \leadsto^*_\sigma t_n$, decimos que t_n es un *valor* computado y σ es un resultado computado para t_0.

Ejemplo 2.2 *Considérese la siguiente definición de función "+":*

$$\begin{array}{rcll} zero + y & \rightarrow & y & (R_1) \\ succ(x) + y & \rightarrow & succ(x + y) & (R_2) \end{array}$$

Dado el término $x + succ(zero)$*, como objetivo, entonces* narrowing *de*

[8] Algunas estrategias de *narrowing* (e.g., *narrowing* necesario) computan unificadores los cuales no son los más generales, véase más adelante.

forma indeterminista ejecuta las siguientes derivaciones:

$$
\begin{array}{ll}
x + succ(zero) & \\
\quad \leadsto_{\epsilon,R_1,\{x \mapsto zero\}} & succ(zero) \\
x + succ(zero) & \\
\quad \leadsto_{\epsilon,R_2,\{x \mapsto succ(y_1)\}} & succ(y_1 + succ(zero)) \\
\quad \leadsto_{1,R_1,\{y_1 \mapsto zero\}} & succ(succ(zero)) \\
x + succ(zero) & \\
\quad \leadsto_{\epsilon,R_2,\{x \mapsto succ(y_1)\}} & succ(y_1 + succ(zero)) \\
\quad \leadsto_{1,R_2,\{y_1 \mapsto succ(y_2)\}} & succ(succ(y_2 + succ(zero))) \\
\quad \leadsto_{1.1,R_1,\{y_2 \mapsto zero\}} & succ(succ(succ(zero))) \\
\ldots &
\end{array}
$$

Por lo tanto, $x+succ(zero)$ *computa de forma indeterminista los siguientes valores (aquí, usamos* $succ^n$ *como una abreviación para* n *aplicaciones de la función succ):*

- $succ(zero)$ *con sustitución* $\{x \mapsto zero\}$,
- $succ^2(zero)$ *con sustitución* $\{x \mapsto succ(zero)\}$,
- $succ^3(zero)$ *con sustitución* $\{x \mapsto succ^2(zero)\}$, *etc.*

Tal como en programación lógica, las derivaciones de *narrowing* pueden ser representadas por un *árbol* de ramificación finita (posiblemente infinita). Formalmente, dado un SRT $\mathcal{R}$ y un término t encabezado por operación, un *árbol narrowing* para t en $\mathcal{R}$ es un árbol que satisface las siguientes condiciones:

1. cada nodo del árbol es un término,
2. el nodo raíz es t,
3. si s es un nodo del árbol entonces, para cada paso de *narrowing* $s \leadsto_{p,R,\sigma} s'$, el nodo tiene un hijo s' y el arco correspondiente esta etiquetado con (p, R, σ), y
4. los nodos que son términos constructores no tienen hijos.

Para evitar tratar con estructuras de datos infinitas y cómputos innecesarios, varias estrategias de *narrowing perezoso* han adoptado la creación del espacio de búsqueda dirigido–por–demanda [GLMP91, LLR93, MR92]. Una estrategia de *narrowing perezoso* reduce las expresiones ubicadas en las posiciones de reducción (por *narrowing*) más externas. La reducción por *narrowing* en las posiciones más internas son ejecutadas sólo si es demandada por la parte izquierda de alguna regla [AFJV03]. A causa de estas propiedades de optimización respecto a la longitud de las derivaciones y al número de soluciones computadas, *narrowing necesario* [AEH00] es actualmente la mejor estrategia de *narrowing* perezoso.

2.4.2. *Narrowing* necesario

Decimos que $s \leadsto_{p,R,\sigma} t$ es un *paso de narrowing necesario* sii $\sigma(s) \to_{p,R} t$ es un paso de *reescritura necesario* en el sentido de Huet y Lévy [HL92], es decir, en cada cómputo que va desde $\sigma(s)$ a una forma normal, tiene que reducirse ya sea $\sigma(s)|_p$ o uno de sus *descendientes*. Aquí, estamos interesados en una estrategia particular de *narrowing necesario*, denotada por λ en [AEH00, Def. 13], la cual se basa en el concepto de *árbol definicional* [Ant92] (una estructura jerárquica que contiene las reglas de una definición de función, que se usa para guiar los pasos de *narrowing necesario*). Esta estrategia es básicamente equivalente a *narrowing perezoso* [MR92] donde los pasos de *narrowing* se aplican a la función más externa, si es posible, y las funciones interiores se reducen por *narrowing* sólo si su evaluación se *demanda* por un símbolo constructor en la parte izquierda de alguna regla (esto es, una típica estrategia de evaluación *call-by-name*).

Definición 2 (*narrowing* necesario [AEH00]) *Un paso de narrowing* $t \leadsto_{p,R,\sigma} t'$ *es* necesario *(o necesario más externo) sii, para cada* $\eta \geq \sigma$, p *es la posición de un redex necesario (o necesario más externo) de* $\eta(t)$. *Una derivación de* narrowing *es llamada necesaria (o necesaria más externa) sii cada paso de la derivación es necesario (o necesario más externo).*

La principal diferencia es que *narrowing* necesario no calcula el *unificador más general* entre el redex seleccionado y la parte izquierda de la

regla sino sólo un unificador. Los enlaces adicionales se requieren para asegurar que sólo se ejecutan cómputos "*needed*" (véase, e.g., [AEH00]), de este modo, *narrowing* necesario generalmente computa un espacio de búsqueda menor.

Ejemplo 2.3 *Considérese nuevamente las reglas de la definición de función "$\leqslant$" del Ejemplo 2.1. En un término como $t_1 \leqslant t_2$, narrowing necesario procede como sigue: Primero, t_1 debe ser evaluado a alguna* forma normal en cabeza *(o sea, una variable libre o un término encabezado-por-constructor) dado que todas las reglas que definen "$\leqslant$" tienen un primer argumento no-variable. Entonces,*

1. *Si t_1 se evalúa a zero entonces se aplica la primera regla.*

2. *si t_1 se evalúa a $succ(t'_1)$ entonces t_2 es evaluado a forma normal en cabeza:*

   (a) *Si t_2 se evalúa a zero entonces se aplica la segunda regla.*

   (b) *Si t_2 se evalúa a $succ(t'_2)$ entonces se aplica la tercera regla.*

   (c) *Si t_2 se evalúa a una variable libre, entonces ésta es instanciada a un término encabezado-por-constructor, aquí zero o $succ(x)$ y, dependiendo de ésta instanciación, procedemos como en los casos (a) or (b) de arriba.*

3. *Finalmente, si t_1 se evalúa a una variable libre, narrowing necesario crea una instancia de ésta a un término encabezado-por-constructor (zero o succ(x)). Dependiendo de ésta instanciación, procedemos como en los casos (1) o (2) de arriba.*

Observemos que *narrowing* necesario está únicamente definido sobre términos encabezados-por-operación, es decir, una derivación de *narrowing* necesario se suspende cuando se obtiene una forma normal en cabeza (un *valor* en nuestro contexto). Esto no es una limitación dado que la evaluación a una forma normal puede reducirse a una secuencia de cómputos de forma normal en cabeza (véase [HP99]).

Árboles definicionales

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

Definición 3 (árbol definicional [AEH00]) *$\mathcal{T}$ es un árbol definicional parcial o pdt (de la expresión en inglés "partial definitional tree") con patrón π sii se cumple uno de los siguientes casos:*

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

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

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

Para facilitar la comprensión del concepto de árbol definicional, a menudo es conveniente dar una representación gráfica en la que cada nodo se etiqueta con un patrón, la posición inductiva en las ramas se enmarca dentro de una caja y las hojas contienen las reglas correspondientes. En el ejemplo 2.4 mostramos el concepto de árbol definicional.

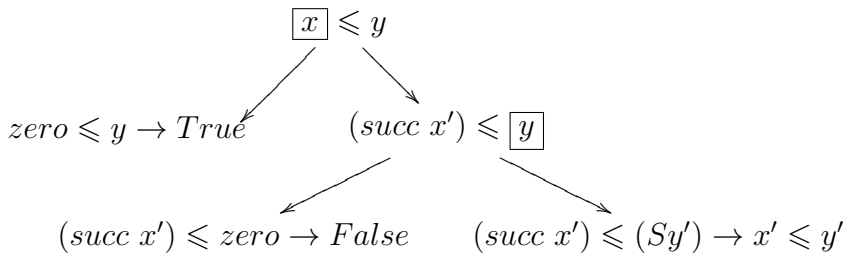


Figura 2.1: Árbol definicional para la función "$\leqslant$".

Ejemplo 2.4 *Considérese nuevamente la función "$\leqslant$" sobre los números naturales del ejemplo 2.1*

$$\begin{array}{rcll} zero & \leqslant & y & \to\ True \\ succ(x) & \leqslant & zero & \to\ False \\ succ(x) & \leqslant & succ(y) & \to\ x \leqslant y \end{array}$$

La Figura 2.1 muestra el árbol definicional para la función "$\leqslant$". Cada nodo que representa una rama contiene el patrón del nodo correspondiente del árbol definicional. Cada nodo de tipo hoja representa una regla. Los argumentos inductivos de cada rama se enmarcan con un recuadro.

Para la exposición de la estrategia de *narrowing* necesario es importante distinguir si un nodo tipo hoja de un árbol definicional de una función f representa o no una regla de la definición de f. Para señalar que $leaf(\pi)$ es un *pdt* que no representa a ninguna regla se va a usar $exempt(\pi)$ en lugar de $leaf(\pi)$. De la misma manera, se abrevia con $rule(\pi, \sigma(l) \to \sigma(r))$ el hecho de que $leaf(\pi)$ sea un *pdt* que representa alguna regla $l \to r$ del sistema de reescritura considerado, donde σ es una sustitución de renombramiento tal que $\sigma(l) = \pi$. Los patrones de un árbol definicional son un conjunto finito parcialmente ordenado por el preorden de subsumción y es completo en el sentido de [HH82].

Para obtener una definición precisa de concepto de la estrategia del *narrowing* necesario, la cual esta basada en el concepto de *árbol definicional*, el lector interesado puede encontrar definiciones detalladas en [Ant92, AEH00]). En el resto del documento, usamos *narrowing necesario* para referirnos a la estrategia particular λ_{needed} definida en [AEH00, Def. 13].

Capítulo 3

Evaluación parcial *offline*

En este capítulo evocamos el marco conceptual de evaluación parcial dirigida por *narrowing* (NPE [AV02]), comenzando por sus antecedentes y con los conceptos de especialización *online* de programas funcionales y lógico funcionales de primer orden. Así como la introducción del primer esquema de evaluación parcial *offline* donde se indentifica una clase de sistemas de reescritura cuasi-terminantes. Los conceptos y definiciones presentadas en este capítulo están publicados en [RSV05a, RSV05b].

3.1. Introducción

Dado un programa y una llamada inicial a función (conteniendo algún dato conocido), el objetivo de un evaluador parcial es la construcción de un nuevo programa residual especializado para esta llamada. El componente esencial de muchos evaluadores parciales es una técnica para calcular una representación *finita* del espacio de cómputo —generalmente *infinito*— para la llamada inicial, tal que un programa residual (previsiblemente más eficiente) se pueda extraer de esta representación. Por ejemplo, dado un programa $\mathcal{P}$ y una llamada a función inicial, $f(t, x)$, donde t es un dato de entrada conocido y x es una variable libre, un evaluador parcial trivial puede devolver un programa residual $\mathcal{P}' = \mathcal{P} \cup \{f_t(x) = f(t, x)\}$ que incluye una versión especializada f_t de la función f. En tanto que la validez de este evaluador parcial trivial es obvia, también es claro que podría o no alcanzarse una mejora en la eficiencia. Un reto en la evalua-

ción parcial es la definición de técnicas para construir representaciones finitas del espacio de cómputo de un programa a partir de las cuales sea posible extraer programas residuales eficientes.

La evaluación parcial dirigida por *Narrowing* (NPE) es una técnica poderosa de especialización para sistemas de reescritura [AV02], esto es, para el componente de primer orden de muchos lenguajes (lógico) funcionales como Haskell [PJ03] o Curry [Han06]. Las características de orden-superior pueden ser modeladas usando un operador de aplicación explícito, esto es, por desfuncionalización [Rey98]. Ésta estrategia se usa en algunas implementaciones de lenguajes lógico funcionales perezosos, tales como el Münster Curry Compiler (MCC [Lux03]) y el sistema Portland-Aachen-Kiel Curry System (PAKCS [HeAE+04]). Aunque la NPE puede ser vista como un esquema tradicional de evaluación parcial para la especialización de programas, puede alcanzar optimizaciones más poderosas, tales como: la deforestación [Wad90], la eliminación de funciones de orden-superior (representadas en un entorno de primer orden por desfuncionalización), etc. Un evaluador parcial dirigido por *narrowing* está actualmente integrado en el entorno de PAKCS para Curry (una evaluación experimental se encuentra en [AHV02]).

En el núcleo del esquema NPE subyace un método para construir una representación finita a partir de un espacio de computación (normalmente) infinito. Para ser precisos, dado un sistema de reescritura $\mathcal{R}$ y un término t, NPE contruye una representación *finita* de todas las posibles derivaciones de t —y cualquiera de sus instancias si éste contiene variables— en $\mathcal{R}$, y extrae un nuevo sistema de reescritura, frecuentemente más simple y eficiente. Dado que t puede contener variables, se requiere alguna forma de *computación simbólica*. En NPE, se usa un refinamiento del *narrowing* [Sla74] para ejecutar computaciones simbólicas, resultando ser *narrowing* necesario (*needed narrowing*) [AEH00] la estrategia que presenta mejores propiedades (como se muestra en [AHLV05]). En general, el espacio de *narrowing* de un término puede ser infinito. Sin embargo, inclusive en este caso, NPE puede terminar cuando el programa original es *cuasi-terminante* [Der87] con respecto a la estrategia de *narrowing* considerada, es decir, cuando se computa un número finito de términos diferentes —módulo renombramiento de

variables. La razón es que la evaluación (parcial) de múltiples ocurrencias del mismo término (módulo renombramiento de variables) en una computación se puede evitar, insertando una llamada a una variante previamente encontrada.

Los evaluadores parciales se clasifican en dos grandes categorías, *online* y *offline*, de acuerdo al momento temporal en que se consideran los aspectos de terminación. Los evaluadores parciales *online* son usualmente más precisos ya que disponen de mayor información. Por ejemplo, el esquema original NPE (el cual sigue la aproximación *online*) considera una variante del teorema de Kruskal (*Kruskal's Tree Theorem*) llamada subsumción homeomórfica (*"homeomorphic embedding"* [Leu02]) para asegurar la terminación del proceso [AFV98]: si un término subsume algún término previo en la misma computación de *narrowing*, se aplica alguna forma de generalización —usualmente el operador de *generalización más especifica*— y la evaluación parcial se reinicia con los términos generalizados. Sin embargo, esta precisión adicional implica un coste: las comprobaciones de la subsumción homeomórfica, conjuntamente con las generalizaciones asociadas, hacen que el esquema NPE *online* sea muy costoso; debido a esto, no se adapta adecuadamente a problemas realistas tales como la especialización de intérpretes [Jon04] o la generación de compiladores por auto-aplicación [Fut99].

Los evaluadores parciales *offline* normalmente se ejecutan en dos etapas: la primera etapa devuelve un programa que incluye anotaciones para guiar los cómputos parciales (e.g., para identificar aquellas llamadas a función que pueden ser desplegadas con toda seguridad, es decir, sin riesgo de no terminación); después, la segunda etapa —la propia especialización— sólo debe obedecer las anotaciones y, por tanto, este tipo de evaluador parcial generalmente es mucho más rápido que los evaluadores parciales *online*. En un escenario lógico funcional, indudablemente, se requiere la evaluación (indeterminista) de términos que incluyen variables libres al momento de la ejecución. Por lo tanto, la primera etapa del esquema de evaluación parcial *offline* garantiza la terminación de la especialización aún cuando todos los argumentos de una llamada a función inicial sean dinámicos (o sea, desconocidos).

Las principales contribuciones de la primera aproximación *offline* a

la evaluación parcial de programas lógico funcionales de [RSV05a] son las siguientes. Primero, se identifica una clase de SRTs, llamados *no-crecientes*, mediante la especificación de condiciones suficientes. Esto es un resultado interesante por si mismo ya que en la literatura no se aprecia una caracterización previa. Desafortunadamente, esta clase es muy restrictiva y, por lo tanto, se introduce también un algoritmo que toma un programa *inductivamente secuencial* —una clase mucho más amplia— y generan un programa *anotado* cuyas anotaciones indican los términos que violan las condiciones suficientes. A continuación, se define una relación extendida del *narrowing* necesario, *el narrowing necesario generalizante*, en la que se generalizan los sub-términos anotados. Se demuestra que los cómputos con esta relación son cuasi-terminantes para programas inductivamente secuenciales anotados y, de esta forma, se establece una base apropiada para garantizar la terminación de NPE *offline*.

3.2. Evaluación parcial

En esta sección se presenta una introducción general e informal de la aproximación a la evaluación parcial de programas (lógico) funcionales. En este escenario, los datos de entrada para el evaluador parcial son: un sistema de reescritura —un programa funcional de primer orden típico— y una llamada a función inicial, la cual suele contener algunos datos conocidos (los llamados datos *estáticos*). Por ejemplo, considérese el siguiente sistema de reescritura:

$$\begin{array}{rcl} inc(x) & \rightarrow & add(succ(zero), x) \\ add(zero, y) & \rightarrow & y \\ add(succ(x), y) & \rightarrow & succ(add(x, y)) \end{array}$$

donde los números naturales son construidos a partir de *zero* y *succ*. Podemos evaluar parcialmente este programa con respecto al término inicial $inc(x)$ para obtener una definición directa de la función *inc* (es decir, especializando la función *add* con el primer argumento estático $succ(zero)$).

Ambos evaluadores parciales tanto *online* como *offline* deben construir alguna forma de *árbol de ejecución simbólica*. Se dice *simbólica*

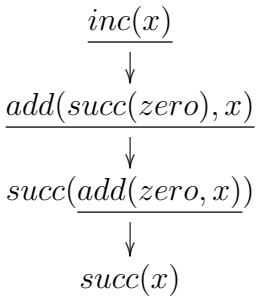
porque los términos pueden contener variables libres y, por lo tanto, a menudo se requiere un mecanismo de ejecución simbólica no estándar. Además, se obtiene una estructura de *árbol* ya que la evaluación de las llamadas a función incluyendo variables libres generalmente requieren derivaciones indeterministas.

La construcción de tal árbol de ejecución simbólica es explícito en algunas técnicas de evaluación parcial (tales como, e.g., la supercompilación positiva [SGJ96a] o la evaluación parcial dirigida por *narrowing* [AV02]). En algunas otras técnicas, dicha construcción es implícita. Por ejemplo, muchos evaluadores parciales para programas funcionales (véase, e.g., [JGS93]) incluyen un algoritmo que iterativamente (1) toma una llamada a función, (2) ejecuta algunas evaluaciones simbólicas, y (3) de la expresión evaluada parcialmente extrae el conjunto de llamadas a función pendientes por resolver —los denominados *sucesores* de la llamada a función inicial— en la siguiente iteración del algoritmo. Obsérvese que, si agregamos una flecha desde cada término a su conjunto de *sucesores*, podríamos obtener una clase de árbol de ejecución simbólica.

Para ejecutar computaciones simbólicas y evaluar términos con variables libres en un contexto funcional, se requiere una extensión de la semántica estándar. Aquí, surge de manera natural la alternativa del *narrowing* [Sla74] como mecanismo de computación simbólica ya que éste combina reducciones funcionales con la instanciación de las variables libres (para una definición formal véase el capítulo 2). Además, en el escenario de la programación lógico funcional, puede utilizarse el mismo principio operacional para ejecutar tanto computaciones simbólicas como estándar [AV02] (de manera análoga a la evaluación parcial de programas lógicos, donde la resolución-SLD es usada tanto para cómputos simbólicos como estándar [LS91]).

Por ejemplo, de acuerdo al programa anterior tenemos el siguiente árbol de ejecución simbólica que obedece a la llamada inicial $inc(x)$ (la

llamada a función seleccionada aparece subrayada):



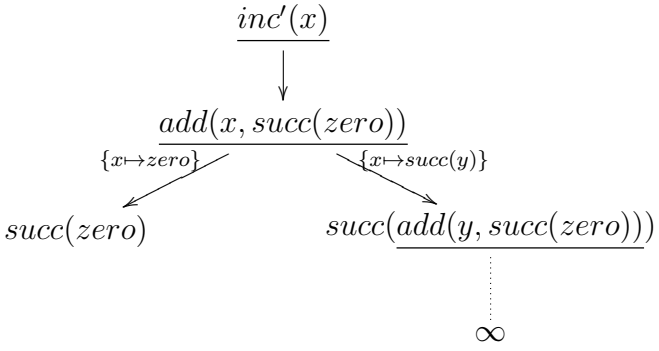
Aquí, no fue necesario calcular posibles instancias de las variables libres; por lo tanto, tenemos una evaluación determinista. El programa residual asociado puede extraerse fácilmente a partir de las computaciones que van de la raíz a las hojas en el árbol de ejecución simbólica. Del ejemplo anterior obtenemos la siguiente regla:

$$inc(x) \;\rightarrow\; succ(x)$$

En la práctica, los evaluadores parciales incluyen alguna clase de técnica de memorización para evitar la evaluación repetida del mismo término (módulo renombramiento de variables). Considérese la siguiente definición de función:

$$inc'(x) \;\rightarrow\; add(x, succ(zero))$$

Aunque el árbol de ejecución simbólica para $inc'(x)$ es infinito:



un evaluador parcial terminaría de especializar exitosamente este ejemplo puesto que la llamada a función $add(y, succ(zero))$ es una variante de $add(x,\ succ(zero))$. En el árbol anterior, las flechas que surgen de $add(x,$

$succ(zero)$) están etiquetadas con las sustituciones calculadas por *narrowing*, esto es, sustituciones tales que, cuando se aplican a $add(x, succ(zero))$ con la semántica estándar, permiten un paso de reducción. El programa residual asociado es el siguiente:

$$\begin{array}{rcl} inc'(x) & \rightarrow & add(x, succ(zero)) \\ add(zero, succ(zero)) & \rightarrow & succ(zero) \\ add(succ(y), succ(zero)) & \rightarrow & succ(add(y, succ(zero))) \end{array}$$

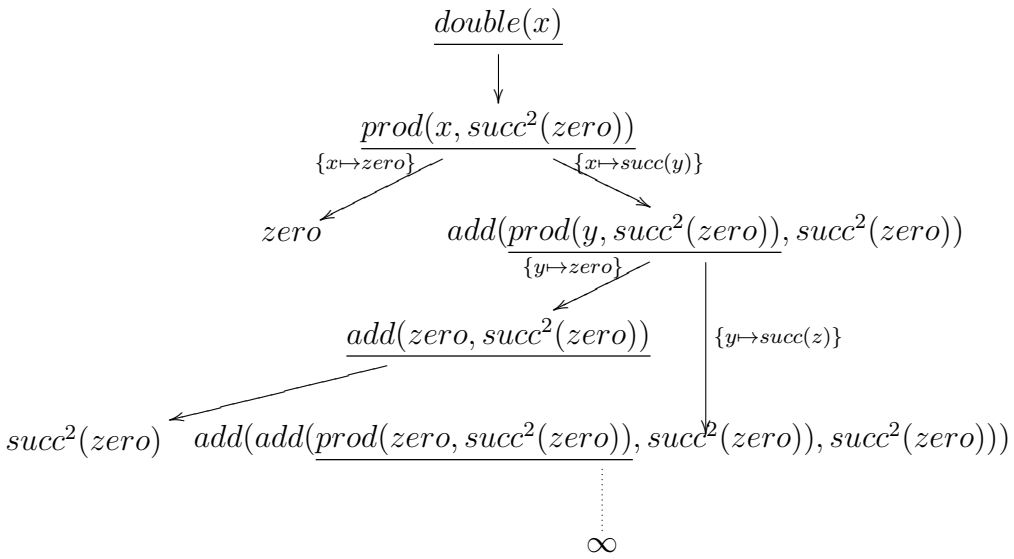
En este caso, tenemos una regla residual asociada al primer paso de evaluación y dos reglas residuales asociadas, una para cada paso indeterminista (aquí, los enlaces computados se aplican a las partes izquierdas de las reglas).

La terminación del árbol de ejecución simbólica puede asegurarse cuando las computaciones simbólicas son *cuasi-terminantes*, es decir, cuando se obtiene un número finito de términos diferentes —módulo renombramiento de variables. Nótese que, aún si el programa considerado es terminante con respecto a la semántica estándar, el mecanismo de ejecución simbólica puede originar tanto computaciones no-terminantes como no-cuasi-terminantes. Considérese la siguiente definición de función:

$$\begin{array}{rcl} double(x) & \rightarrow & prod(x, succ(succ(zero))) \\ prod(zero, y) & \rightarrow & zero \\ prod(succ(x), y) & \rightarrow & add(prod(x, y), y) \end{array}$$

Dada la llamada inicial $double(x)$, el árbol simbólico asociado es infinito

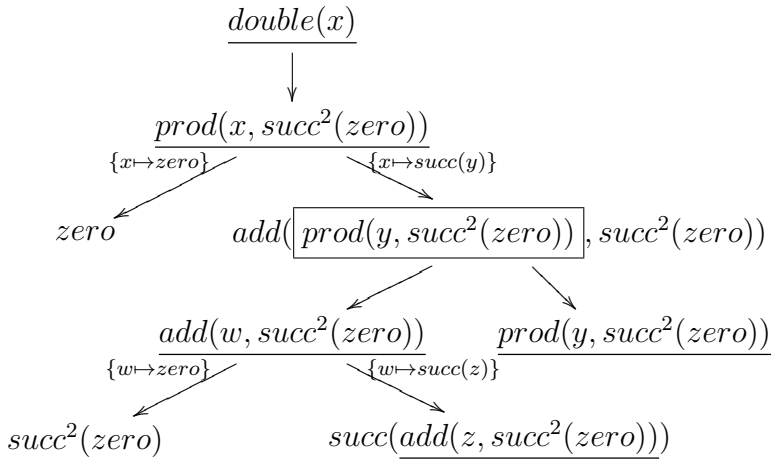
(se usa $succ^2(zero)$ como una abreviación para $succ(succ(zero))$):



Para asegurar siempre la terminación de los árboles de ejecución simbólica, se debe considerar una operación de *generalización* sobre términos. La decisión sobre cuáles términos deben ser generalizados puede ser tomada en una etapa de pre-procesamiento (el caso de la evaluación parcial *offline*) o durante el propio proceso de evaluación parcial (como en la evaluación parcial *online*). Los evaluadores parciales *online* son usualmente más precisos puesto que tienen más información disponible para decidir si es necesario generalizar o no. En contraste, los evaluadores parciales *offline* son menos precisos pero son generalmente mucho más rápidos ya que en la etapa de especialización sólo deben seguir el procedimiento indicado por las anotaciones agregadas durante el pre-procesamiento (el llamado análisis *binding-time*).

En el ejemplo anterior, la terminación puede garantizarse generali-

zando la segunda llamada a la función *prod* como sigue:



Ahora, el árbol de ejecución simbólica se mantiene finito ya que todas las hojas son valores tales como $zero$ y $succ^2(zero)$ (o sea, no contienen llamadas a función) o contienen una llamada a función que es una variante de una llamada a función previa dentro del árbol (como son los casos $prod(y, succ^2(zero))$ y $add(z, succ^2(zero))$, los cuales son variantes de $prod(x, succ^2(zero))$ y $add(w,\ succ^2(zero))$, respectivamente).

A partir de este árbol de ejecución simbólica, se puede extraer el siguiente programa residual:

$$\begin{aligned} double(x) &\rightarrow prod(x, succ^2(zero)) \\ prod(zero, succ^2(zero)) &\rightarrow zero \\ prod(succ(y), succ^2(zero)) &\rightarrow add(prod(y, succ^2(zero))) \\ add(zero, succ^2(zero)) &\rightarrow succ^2(zero) \\ add(succ(z), succ^2(zero)) &\rightarrow succ(add(z, succ^2(zero))) \end{aligned}$$

En el resto de este capítulo, se presenta un enfoque sistemático para la evaluación parcial *offline* de sistemas inductivamente secuenciales.

3.3. Garantizando cuasi-terminación con respecto a *narrowing* necesario

En esta sección se resume la aproximación de evaluación parcial *offline* de [RSV05a]. En el esquema NPE, *narrowing* se usa como mecanismo

de computación simbólica para ejecutar cómputos parciales [AV02]. A grandes rasgos, dado un programa $\mathcal{R}$ y un término inicial t, el proceso de evaluación parcial procede construyendo un árbol de *narrowing* para t en $\mathcal{R}$ con la restricción adicional de no evaluar términos que sean variantes de otros ya evaluados previamente en ese árbol. Por lo tanto, la terminación del proceso NPE puede ser garantizada cuando todos los cómputos de *narrowing* son cuasi-terminantes. De manera análoga a Holst [Hol91], decimos que una computación es *cuasi-terminante* cuando ésta contiene un número finito de términos diferentes (módulo renombramiento de variables).

La instancia mas reciente del esquema NPE esta basado en *narrowing necesario* [AHLV05]. Sin embargo, mientras el esquema NPE original garantiza que los cómputos son cuasi-terminantes de manera *online* (aplicando operadores de generalización y chequeos de terminación apropiados), aquí se introduce una condición suficiente para SRTs tal que las computaciones de *narrowing* necesario siempre son cuasi-terminantes. Para ello se introducen las siguientes definiciones:

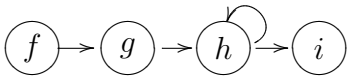
Definición 4 (grafo de dependencias funcionales) *Dado un SRT $\mathcal{R}$, su grafo de dependencias funcionales, en símbolos $\mathcal{G}(\mathcal{R})$, contiene nodos etiquetados con los símbolos de función en $\mathcal{D}$ y existe un arco del nodo f al nodo g sii hay una llamada a la función g desde la parte derecha de alguna regla en la definición de función f.*

Definición 5 (función cíclica, función no cíclica) *Dado un SRT $\mathcal{R}$. Una función $f \in \mathcal{D}$ es cíclica si el nodo f forma parte de un ciclo en $\mathcal{G}(\mathcal{R})$ y en caso contrario es no cíclica.*

Ejemplo 3.1 Considérese el siguiente SRT $\mathcal{R}$:

$$\begin{array}{rcl} f(s(x), y) & \to & g(x, y) \\ g(x, s(y)) & \to & h(x, y) \\ h(0, y) & \to & y \\ h(s(x), y) & \to & c(i(x), h(x, y)) \\ i(x) & \to & x \end{array}$$

donde $f, g, h, i \in \mathcal{D}$ son funciones definidas y $0, s, c \in \mathcal{C}$ son símbolos constructores. El grafo de dependencias funcionales $\mathcal{G}(\mathcal{R})$ asociado, es como sigue:



Las funciones f, g, e i son no cíclicas, mientras que h es cíclica.

Claramente, las funciones no-cíclicas no pueden introducir computaciones no terminantes (tampoco no-cuasi-terminantes) siempre y cuando las funciones cíclicas no las introduzcan. De este modo, la atención se enfocará en las funciones cíclicas. De acuerdo a [CK96], se dice que la *profundidad de una variable x en un término constructor t*, en símbolos $dv(t, x)$, está definida como sigue:

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

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

La siguiente definición introduce el concepto de función *no-creciente*, esto es, una función que siempre *consume* sus parámetros o los deja inalterados:

Definición 6 (función no-creciente) *Sea $\mathcal{R}$ un SRT constructor lineal por la izquierda. Una función $f \in \mathcal{D}$ es no-creciente sii cada regla $f(\overline{s_n}) \to r$ incluida en la definición de f satisface las siguientes condiciones:*

1. *la parte derecha no contiene símbolos de función anidados (esto es, símbolos de función que ocurran dentro de otros símbolos de función definidos), y*
2. *$dv(s_i, x) \geqslant dv(t_j, x)$ para todos los sub-términos encabezados por operación $g(\overline{t_m})$ en r, donde $i \in \{1, \ldots, n\}$, $x \in \mathcal{V}ar(s_i)$, y $j \in \{1, \ldots, m\}$.*

Ejemplo 3.2 Una función definida por una sola regla

$f(x, y, s(z)) \to c(g(x)\ , h(z))$, con $s, c \in \mathcal{C}$ y $f, g, h \in \mathcal{D}$, es no-creciente dado que se cumplen las siguientes relaciones:

$$\begin{aligned} dv(x,x) = 0 &\geqslant 0 = dv(x,x) \\ dv(x,x) = 0 &\geqslant -1 = dv(z,x) \\ dv(y,y) = 0 &\geqslant -1 = dv(x,y) \\ dv(y,y) = 0 &\geqslant -1 = dv(z,y) \\ dv(s(z),z) = 1 &\geqslant -1 = dv(x,z) \\ dv(s(z),z) = 1 &\geqslant 0 = dv(z,z) \end{aligned}$$

es decir, la variable x únicamente es copiada, la variable y se desvanece, y (la profundidad de) la variable z decrece.

De forma análoga a [Der87], se dice que un SRT es *cuasi-terminante para un conjunto de términos T con respecto a narrowing necesario* sii son cuasi-terminantes todas las derivaciones de *narrowing* necesario que se computan a partir de los términos en T. Ahora, se ofrece una condición suficiente para la cuasi-terminación de SRTs:

Definición 7 (SRT no-creciente) *Sea $\mathcal{R}$ un SRT inductivamente secuencial. $\mathcal{R}$ es no-creciente sii todas las funciones $f \in \mathcal{D}$ son lineales por la derecha y no cíclicas o no-crecientes.*

La restricción a SRTs inductivamente secuenciales no es realmente necesaria (es decir, los SRTs constructores lineales por la izquierda serían suficientes) pero se impone esta condición porque *narrowing necesario* está definido únicamente para esta clase de SRTs. Por otro lado, la linealidad por la derecha es necesaria no sólo para garantizar cuasi-terminación sino para asegurar que el desplegado de funciones no introduzca computaciones repetidas.

Teorema 8 (SRT cuasi-terminante [RSV05a]) *Si $\mathcal{R}$ es un SRT no-creciente, entonces $\mathcal{R}$ es cuasi-terminante para cualquier término lineal con respecto a narrowing necesario.*

Se advierte que no hay una relación clara entre cuasi-terminación con respecto a *narrowing necesario* y las condiciones relacionadas en reescritura de términos. Por ejemplo, considérese el siguiente SRT:

$$\begin{aligned} f(0,y) &\to y \\ f(s(x),y) &\to f(x,s(y)) \end{aligned}$$

el cual no cumple las condiciones para los SRTs no-crecientes, y donde $0, s \in \mathcal{C}$ y $f \in \mathcal{D}$. Este SRT es trivialmente terminante con respecto a reescritura ya que el primer parámetro de la función f decrece estrictamente en cada llamada recursiva. Sin embargo, no es cuasi-terminante con respecto a *narrowing necesario*, como se muestra en la siguiente computación (infinita):

$$\begin{array}{lll} f(x,y) & \leadsto_{\{x \mapsto s(x')\}} & f(x', s(y)) \\ & \leadsto_{\{x' \mapsto s(x'')\}} & f(x'', s(s(y))) \\ & \leadsto & \ldots \end{array}$$

Otros conceptos relacionados con la terminación son igualmente poco efectivos para garantizar cuasi-terminación con respecto a *narrowing necesario*, como la terminación por cambio de tamaño (*size-change analysis* [LJBA01]) adaptada a SRTs en [TG03]), ya que sólo aseguran que *algunos* parámetros decrezcan (pero no todos ellos), lo cual no es suficiente en el contexto lógico funcional donde todos los parámetros pueden ser desconocidos (esto es, variables libres). La linealidad por la derecha es un requerimiento esencial inclusive para funciones muy simples. Por ejemplo, considérese las siguientes funciones no-crecientes:

$$\begin{aligned} f(0, y) &\to y \\ f(s(x), y) &\to f(x, y) \\ g(x) &\to f(x, x) \end{aligned}$$

donde $0, s \in \mathcal{C}$ y $f, g \in \mathcal{D}$. Éste no es un SRT no-creciente ya que la función g no es lineal por la derecha. Por lo tanto no asegura cuasi-terminación con respecto a *narrowing necesario*:

$$\begin{array}{llllll} g(x) & \leadsto_{id} & f(x,x) & \leadsto_{\{x \mapsto s(x')\}} & f(x', s(x')) \\ & & & \leadsto_{\{x' \mapsto s(x'')\}} & f(x'', s(s(x''))) \\ & & & \leadsto & \ldots \end{array}$$

Evidentemente, el uso de la estrategia de *narrowing necesario* es crucial, es decir, no se garantiza la cuasi-terminación para otras estrategias de *narrowing* (e.g., *narrowing innermost*). Por ejemplo, dado el siguiente

SRT cuasi-terminante:

$$\begin{array}{rcl} f(x) & \to & g(h(x)) \\ h(0) & \to & 0 \\ h(s(x)) & \to & s(h(x)) \\ g(x) & \to & x \end{array}$$

de acuerdo a *narrowing necesario* el sistema es cuasi-terminante mientras que para la estrategia *innermost* podría producir la siguiente derivación no cuasi-terminante:

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

La especificación más cercana a la caracterización no-creciente ha sido presentada en [Wad90] y [CK96]. Wadler introdujo el concepto de funciones *treeless* para asegurar la terminación del proceso de *deforestación* [Wad90]. Las funciones *treeless* son una subclase de las funciones no-crecientes donde, además, todas las llamadas a función en las partes derechas de las reglas sólo pueden tener variables en sus argumentos. Chin y Khoo [CK96] introdujeron la clase de funciones denominadas *consumidores no-crecientes* y demostraron que cualquier conjunto de funciones mutuamente recursivas que sean *consumidores no-crecientes* pueden ser transformadas en un conjunto equivalente de funciones *treeless*, de tal forma que pueda ser aplicado el proceso de deforestación. Esta caracterización difiere de la no-creciente en dos puntos principalmente. Primero, Chin y Khoo sólo requieren *llamadas a función* lineales en las partes derechas de las reglas (en lugar de exigir linealidad en toda la parte derecha de las reglas, como se requiere en la caracterización no-creciente). Esta definición menos restrictiva no es correcta en el contexto de *narrowing*. Por ejemplo considérese las siguientes funciones de *consumidores no-crecientes* de acuerdo a Chin y Khoo [CK96]:

$$\begin{array}{rcl} f(x) & \to & c(g(x), x) \\ g(s(x)) & \to & g(x) \\ h(c(s(x), y)) & \to & x \end{array}$$

donde $c, s \in \mathcal{C}$ y $f, g, h \in \mathcal{D}$. Tenemos que dado el término inicial $h(f(x))$, *narrowing necesario* produce una derivación infinita la cual no es cuasi-terminante:

$$\begin{array}{lll} h(f(x)) & \leadsto_{id} & h(c(g(x), x)) \\ & \leadsto_{\{x \mapsto s(x')\}} & h(c(g(x'), s(x'))) \\ & \leadsto_{\{x' \mapsto s(x'')\}} & h(c(g(x''), s(s(x'')))) \\ & \leadsto & \ldots \end{array}$$

En segundo lugar, Chin y Khoo no aceptan llamadas a función anidadas en la parte derecha de las reglas de programa. Por el contrario, la caracterización de SRTs no-crecientes acepta términos (lineales) arbitrarios en las partes derechas de las funciones no cíclicas, lo que permite cubrir un rango más amplio de funciones.

3.4. De NPE *online* a NPE *offline*

La formulación original del esquema NPE asegura terminación de modo *online* (véase, e.g., [AHV02, AV02, AFV98]), esto se debe a que, durante el proceso de evaluación parcial se utilizan operadores de generalización y pruebas de terminación apropiadas para garantizar que sólo se compute un número finito de términos diferentes (módulo renombramiento de variables). Como se mencionó anteriormente, este esquema logra potentes optimizaciones pero también son muy costosas (en términos de consumo tanto de tiempo como de espacio); por lo tanto, no es muy adecuado para especializar problemas realistas. Para remediar esta situación, en [RSV05a] se introduce un método NPE más rápido que asegura terminación de manera *offline* incluyendo una etapa de preprocesamiento basada en el concepto de SRT no-creciente.

En principio, un método NPE simple podría restringir los programas fuente a SRTs no-crecientes; entonces, ya no sería necesario aplicar controles de terminación ni operaciones de generalización durante la evaluación parcial, puesto que las derivaciones de *narrowing necesario* serían cuasi-terminantes (cf. Teorema 8). Este esquema produciría una herramienta NPE muy rápida —puesto que sólo requeriría verificaciones de igualdad módulo renombramiento de variables— aunque desafortunada-

mente sería muy restrictiva con respecto a la clase de programas susceptibles de ser especializados.

Así pues, se considera una clase mucho más amplia de SRTs, esto es, los programas inductivamente secuenciales, la clase de programas para la cual NPE fue originalmente definido, y se especifica un algoritmo que *anote* las expresiones que puedan causar la no cuasi-terminación de las derivaciones por *narrowing necesario*. El programa anotado es procesado más tarde por una relación extendida del *narrowing necesario* para *generalizar* los subtérminos problemáticos. Sea $\mathcal{F}_\bullet = \mathcal{F} \cup \{\bullet\}$, donde $\bullet \notin \mathcal{F}$ es un símbolo nuevo. Dado un SRT $\mathcal{R}$, un término t se anota reemplazando t por $\bullet(t)$. Las siguientes funciones auxiliares son útiles para manipular términos anotados:

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

es decir, dado un término anotado $t \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$, la expresión $gen(t) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ retorna una generalización de t reemplazando los subtérminos anotados por variables nuevas.

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

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

A continuación se ilustra el uso de las funciones *gen* y *aterms* con algunos ejemplos:

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

La siguiente definición introduce el concepto de transformación para anotar programas inductivamente secuenciales. Intuitivamente, se revisarán las partes derechas de las reglas y se anotarán aquellos argumentos correspondientes a subtérminos encabezados por función definida, es decir, los argumentos de función que incluyan símbolos de función (evitando así, funciones anidadas). Además se anotarán aquellos que violen la propiedad no-creciente; después, cada subtérmino anotado será tratado de manera similar a la anotación de la parte derecha de la regla (de esta forma, es posible obtener anotaciones anidadas). Finalmente, serán anotadas todas las ocurrencias repetidas de la misma variable excepto una. Formalmente se tiene:

Definición 9 ($ann(\mathcal{R})$) *Sea* $\mathcal{R} = \{l_i \to r_i \mid i = 1, \ldots, k\}$ *un SRT inductivamente secuencial sobre* $\mathcal{F}$. *El SRT anotado,* $ann(\mathcal{R})$, *sobre* $\mathcal{F}_\bullet$ *esta dado por el conjunto de reglas* $\{l_i \to r'_i \mid i = 1, \ldots, k\}$ *donde* r'_i, $i = 1, \ldots, k$, *es calculado como sigue:*

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

2. *Si* $root(l_i)$ *es cíclica, entonces* r'_i *se obtiene a partir de* $qs(l_i, r_i)$ *anotando el menor número de variables tal que* $gen(t)$ *sea lineal para todo* $t \in \{qs(l_i, r_i)\} \cup aterms(qs(l_i, r_i))$. *La definición de la función auxiliar qs se muestra en la Figura 3.1.*

De primera mano, la función auxiliar qs ignora símbolos constructores hasta que se encuentra un subtérmino encabezado por un símbolo de función $f(t_1, \ldots, t_n)$. Entonces, por cada argumento t_i, se procede (llamando a qs') como sigue:

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

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

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

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

Figura 3.1: Funciones Auxiliares qs y qs'.

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

Nótese que la Definición 9 es indeterminista ya que no precisa qué variable no debe anotarse cuando hay ocurrencias repetidas de la misma variable. En algunos casos, esta decisión puede afectar significativamente al resultado de la evaluación parcial (véase Sect 3.5.2). Esta situación podría ser optimizada en algunos casos permitiendo al programador elegir la variable (estática) que se dejará sin anotación.

Ejemplo 3.3 Considérese el siguiente programa inductivamente secuencial $\mathcal{R}$:

$$\begin{array}{rcl} f(0,y) & \to & y \\ f(s(x),y) & \to & g(x,f(x,s(y))) \\ g(x,y) & \to & g(y,x) \end{array}$$

donde $f,g \in \mathcal{D}$ y $0,s \in \mathcal{C}$. El SRT $ann(\mathcal{R})$ se anota como sigue:

$$\begin{array}{rcl} f(0,y) & \to y & \\ f(s(x),y) & \to & g(x,\bullet(f(x,\bullet(s(y))))) \\ g(x,y) & \to & g(y,x) \end{array}$$

Obsérvese que las ocurrencias repetidas de x en la segunda regla no se deben de anotar puesto que

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

y, por lo tanto, $gen(t)$ es lineal para todo

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

es decir, $g(x, w_1)$, $f(x, w_2)$, y $s(y)$ son términos lineales, donde w_1 y w_2 son variables (frescas) nuevas.

Puesto que los cómputos parciales realizados en el esquema NPE son calculados por medio de *narrowing necesario*, ahora se extiende esta relación para generalizar subtérminos anotados (y así asegurar la terminación del proceso de evaluación parcial).

Definición 10 (*narrowing* necesario generalizante) *sea $\mathcal{R}$ un SRT inductivamente secuencial anotado sobre $\mathcal{F}_\bullet$. La relación de narrowing necesario generalizante, en símbolos $\rightsquigarrow$, está definida como la menor relación que satisface*

(narrowing necesario)

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

(generalización)

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

(descomposición)

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

Una derivación de *narrowing* necesario generalizante $s \rightsquigarrow_\sigma^* t$ esta compuesta de pasos de *narrowing necesario* (para términos no anotados y encabezados por un símbolo de función), generalizaciones (para términos anotados), y descomposición de constructores (para términos no anotados encabezados por constructor), donde σ representa la composición de las sustituciones que etiquetan los pasos propios de *narrowing necesario*. Nótese que, *narrowing necesario* únicamente computa una *forma normal en cabeza* (es decir, una variable o un término encabezado por constructor), la regla de descomposición se requiere para asegurar que todas las

posibles funciones internas sean, a la larga, evaluadas parcialmente. En la siguiente sección se examinan algunos ejemplos del cálculo de *narrowing* necesario generalizante

Considérese también que un paso de generalización, es de algún modo equivalente a la operación de *splitting* de la *deducción parcial conjuntiva* (*conjunctive partial deduction* CPD) de programas lógicos [SGJ+99]. En tanto que CPD considera conjunciones de átomos, aquí tratamos con términos que posiblemente contienen símbolos de función anidados. Por lo tanto, aplanar la llamada de una función anidada es básicamente equivalente a una operación de *splitting* de una conjunción (en ambos casos se pierde algo de información).

El siguiente resultado muestra la validez del algoritmo de anotación.

Teorema 11 (SRT anotado cuasi-terminante [RSV05b]) *Sea $\mathcal{R}$ un SRT inductivamente secuencial y t un término lineal. Cada derivación de narrowing necesario generalizante para t en $ann(\mathcal{R})$ es cuasi-terminante.*

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

En esta sección, primeramente, se describe el método *offline* NPE completo, que está basado en la anotación de programas procesados con *narrowing* necesario generalizante. Luego, se ilustra el nuevo esquema por medio de algunos ejemplos seleccionados. Finalmente, [RSV05a] presentan un resumen de experimentos realizados con un prototipo implementado cuyos resultados muestran las ventajas de este enfoque comparado con el método *online* NPE original.

3.5.1. Descripción del método NPE *offline*

En esta aproximación *offline* del NPE, dado un SRT inductivamente secuencial $\mathcal{R}$, la *primera etapa* consiste en calcular el SRT anotado: $ann(\mathcal{R})$. Después, la *segunda etapa* —la evaluación parcial propiamente dicha— toma el SRT anotado, $ann(\mathcal{R})$, conjuntamente con un término

(lineal) inicial, t, construye el árbol (finito) de *narrowing* necesario generalizante para t en $ann(\mathcal{R})$, y extrae el programa residual —evaluado parcialmente.

Esencialmente, los programas residuales se extraen a partir de las llamadas *resultantes*, $\sigma(s) \to rann(t)$, por cada paso respectivo de *narrowing necesario* $s \leadsto_\sigma t$ en el árbol de *narrowing* necesario generalizante considerado, donde la función *rann* simplemente elimina las ocurrencias del símbolo "•" en un término. En general, las partes izquierdas de las *resultantes* no están necesariamente en la forma $f(s_1, \ldots, s_n)$, donde s_i representa los términos constructores, por lo que pueden contener símbolos de función definida anidados. Por consiguiente, se precisa un renombramiento de términos para restaurar el formato legal del programa. Las siguientes definiciones originales de [AHLV05] formalizan el concepto de renombramiento:

Definición 12 (renombramiento independiente) *Un renombramiento independiente ρ para un conjunto de términos T esta representado por una función de términos a términos definida como sigue: para todo término $t \in T$,*

- $\rho(t) = t$ *if* $t = f(\overline{x_n})$, donde $f \in \mathcal{D}$ y $\overline{x_n}$ son variables diferentes, y
- $\rho(t) = f_t(\overline{x_n})$, de otra forma, donde $\overline{x_n}$ son variables distintas de t de acuerdo al orden de su primera ocurrencia y $f_t \notin \mathcal{D}$ es un símbolo de función nuevo.

Obsérvese que las llamadas a función cuyos argumentos son variables diferentes no son renombrados ya que esto no es necesario.

Ejemplo 3.4 Considérese el siguiente conjunto de términos

$$T = \{f(x, y),\ g(h(x), y),\ s(c(x), x)\}$$

donde $f, g, h, s \in \mathcal{D}$ son funciones definidas y $c \in \mathcal{C}$ es un símbolo constructor. Entonces, la siguiente distribución ρ es un renombramiento independiente para T:

$$\rho = \{ \begin{array}{rcll} f(x,y) & \mapsto & f(x,y), & \\ g(h(x),y) & \mapsto & g'(x,y), & \\ s(c(x),x) & \mapsto & s'(x) & \} \end{array}$$

Básicamente, dado un programa anotado $ann(\mathcal{R})$ y un término lineal t, la etapa de evaluación parcial procede construyendo un árbol de *narrowing* necesario generalizante para t en $ann(\mathcal{R})$, donde se incluye una comprobación adicional para verificar si ya se ha procesado una variante del término actual y, en caso afirmativo, se detiene la derivación. La cuasi-terminación de los cómputos de *narrowing* necesario generalizante (Teorema 11) garantiza que el árbol generado de esta forma es finito. Una vez que el árbol es construido, se computa un renombramiento independiente ρ para el conjunto de términos $\{s \mid s \leadsto_\sigma t\}$, es decir, para los términos a los que se aplica su respectivo paso de *narrowing necesario*. En tanto que la función ρ es suficiente para al renombramiento de las partes izquierdas de las resultantes, las partes derechas requieren una relación más elaborada, ren_ρ, la cual reemplaza *recursivamente* cada llamada en el término por una llamada a la correspondiente función renombrada. Formalmente,

Definición 13 (función de renombramiento [AHLV05]) *Sea T un conjunto finito de términos y ρ un renombramiento independiente de T. Dado un término $\mathtt{s}$, la función (indeterminista) ren_ρ se define como sigue:*

$$ren_\rho(\mathtt{s}) = \begin{cases} \mathtt{s} & \text{si } \mathtt{s} \in \mathcal{V} \\ c(\overline{ren_\rho(t_n)}) & \text{si } \mathtt{s} = c(\overline{t_n}),\ c \in \mathcal{C},\ \text{y } n \geqslant 0 \\ \theta'(\rho(t)) & \text{si existe un término } t \in T \\ & \text{tal que } \mathtt{s} = \theta(t) \ \text{ y} \\ & \theta' = \{x \mapsto ren_\rho(\theta(x)) \mid x \in \mathcal{D}om(\theta)\} \end{cases}$$

Ejemplo 3.5 Considérese el conjunto de términos T y el renombramiento independiente del ejemplo 3.5.1. Dado el término $g(h(x), f(a, s(c(b), b)))$, donde $a, b \in \mathcal{C}$ son símbolos constructores, la función ren_ρ devuelve el término renombrado $g'(x, f(a, s'(b)))$.

Ahora, el método NPE *offline* puede ser formalizado como sigue:

Definición 14 (NPE *offline*) *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $f(\overline{x_n})$ un término lineal*[1] *con $f \in \mathcal{D}$. La NPE offline de $\mathcal{R}$ con respecto a $f(\overline{x_n})$ se obtiene como sigue:*

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

2. *Después, se construye un árbol (finito) de narrowing necesario generalizante, τ, para $f(\overline{x_n})$ en $ann(\mathcal{R})$, donde cada derivación se detiene cuando ésta alcance un término constructor o un término encabezado por función que sea un renombramiento de variables de algún término precedente en la misma derivación (o en una anterior).*

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

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

Por cada paso de narrowing necesario $s \rightsquigarrow_\sigma s'$ en τ. Aquí, ρ es un renombramiento independiente de $\{s \mid s \rightsquigarrow_\sigma s' \in \tau\}$.

Para hacer más sencilla la definición anterior, a partir de cada paso de *narrowing* necesario individual se extrae una resultante. Evidentemente, es posible definir algoritmos más refinados para extraer las resultantes a partir de un árbol de *narrowing* necesario generalizante, e.g., en muchos casos, se puede extraer una sola resultante asociada a una *secuencia* de pasos *narrowing* en lugar de una resultante de un sólo paso de *narrowing*. De hecho, el prototipo implementado sigue este refinamiento.

Ahora se establece la corrección y terminación de este método de evaluación parcial.

Teorema 15 *Sea $\mathcal{R}$ un SRT inductivamente secuencial y $f(\overline{x_n})$ un término lineal con $f \in \mathcal{D}$. El algoritmo de la Definición 14 siempre termina computando un SRT inductivamente secuencial $\mathcal{R}'$ tal que narrowing necesario calcula los mismos resultados para $f(\overline{x_n})$ en $\mathcal{R}$ y en $\mathcal{R}'$.*

[1] Esta no es una restricción ya que se puede considerar un término arbitrario t agregando simplemente una nueva definición de función $f(\overline{x_n}) \to t$ a $\mathcal{R}$, donde $\overline{x_n}$ son las distintas variables de t.

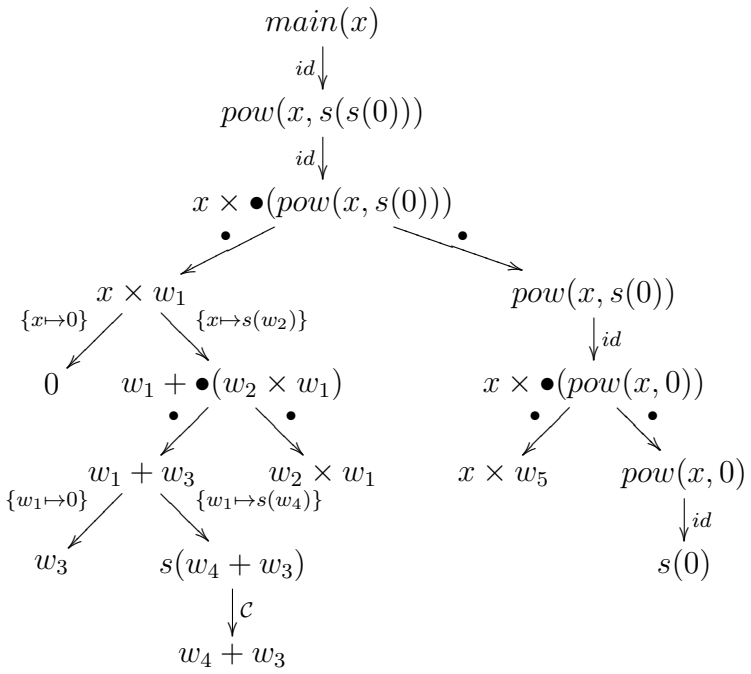


Figura 3.2: Árbol de *narrowing* necesario generalizante para $main(x)$.

3.5.2. Ejemplos seleccionados

En esta sección se muestra una serie de ejemplos seleccionados con los cuales se ilustra el método NPE *offline* diseñado hasta ahora.

Especialización de programas. El primer ejemplo ilustra el uso del método NPE *offline* para especialización de programas. Considérese el siguiente SRT que ha sido anotado de acuerdo a la Definición 9:

$$\begin{array}{rcl} main(x) & \to & pow(x, s(s(0))) \\ pow(x, 0) & \to & s(0) \\ pow(x, s(n)) & \to & x \times \bullet(pow(x, n)) \\ 0 \times m & \to & 0 \\ s(n) \times m & \to & m + \bullet(n \times m) \\ 0 + m & \to & m \\ s(n) + m & \to & s(n + m) \end{array}$$

Dado el término inicial $main(x)$, se construye el árbol de *narrowing* necesario generalizante representado en la Figura 3.2. Así pues, el SRT resi-

dual asociado contiene las siguientes reglas:

$$\begin{array}{rcl} main(x) & \to & pow_2(x) \\ pow_2(x) & \to & x \times pow_1(x) \\ pow_1(x) & \to & x \times pow_0(x) \\ pow_0(x) & \to & s(0) \end{array}$$

junto con las definiciones originales de "$\times$" y "$+$". El renombramiento independiente considerado es como sigue:

$$\begin{array}{rrcll} \rho = \{ & main(x) & \mapsto & main(x), & \\ & pow(x, s(s(0))) & \mapsto & pow_2(x), & \\ & pow(x, s(0)) & \mapsto & pow_1(x), & \\ & pow(x, 0) & \mapsto & pow_0(x), & \\ & x \times y & \mapsto & x \times y, & \\ & x + y & \mapsto & x + y & \} \end{array}$$

Adicionalmente, estas cuatro reglas se pueden simplificar fácilmente realizando un post-proceso estándar de desplegado, llamado *transformación por compresión*[2], como sigue:

$$main(x) \to x \times (x \times s(0))$$

ya que las funciones pow_2, pow_1, y pow_0 son únicamente funciones intermedias (esto es, sólo hay una llamada a cualquiera de ellas y no son recursivas). Este sencillo ejemplo muestra que, a pesar de las anotaciones de algunos subtérminos, la capacidad de especialización del esquema NPE (*online*) original no se pierde con la aproximación *offline*.

Deforestación. El segundo ejemplo está relacionado con el proceso de *deforestación* de Wadler que permite eliminar estructuras de datos intermedias [Wad90]. Considérese el siguiente SRT $\mathcal{R}$:

$$\begin{array}{rcl} lenapp(x, y) & \to & len(app(x, y)) \\ len([\,]) & \to & 0 \\ len(x : xs) & \to & s(len(xs)) \\ app([\,], y) & \to & y \\ app(x : xs, y) & \to & x : app(xs, y) \end{array}$$

[2]De la expresión en inglés: *transition compression* [JGS93].

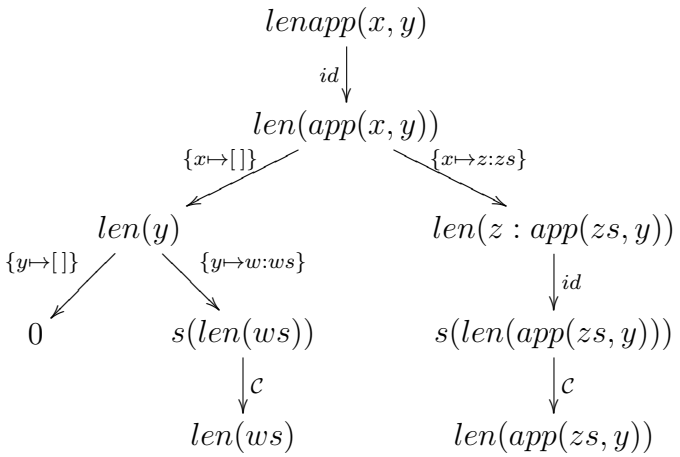


Figura 3.3: Árbol de *narrowing* necesario generalizante para $lenapp(x, y)$.

donde $lenapp(x, y)$ calcula la longitud de la concatenación de dos listas x e y. Esta función no es del todo eficiente ya que se construye una estructura de datos intermedia (la concatenación de x e y). Dado que $\mathcal{R}$ es no-creciente, se tiene que $ann(\mathcal{R}) = \mathcal{R}$. Entonces, dado el término inicial $lenapp(x, y)$, se construye el árbol de *narrowing* necesario generalizante representado en la Figura 3.3. Ahora, usando el siguiente renombramiento independiente:

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

el SRT asociado es como sigue:

$$\begin{aligned} lenapp(x,y) &\to la(x,y) \\ la([\,],y) &\to len(y) \\ la(z : zs,y) &\to la2(z,zs,y) \\ la2(z,zs,y) &\to s(la(zs,y)) \end{aligned}$$

junto con la definición de la función original len. Y, tal como en el ejemplo anterior, una transformación simple de post-desplegado podría eliminar la función intermedia $la2$. Nótese que el SRT residual esta completamente deforestado (esto es, no se construye una lista intermedia).

Eliminación del orden superior. El último ejemplo consiste en la eliminación de funciones de orden superior. En algunos lenguajes de programación, las características del orden superior son *desfuncionalizadas* [Rey98, War82], es decir, son expresadas por medio de programas de primer orden con un operador de aplicación explícito[3]. Por ejemplo, el siguiente SRT, el cual ya ha sido anotado de acuerdo a la Definición 9, incluye la definición de la conocida función de orden superior *map*:

$$\begin{array}{rcl} minc(x) & \to & map(inc_0, x) \\ map(f, [\,]) & \to & [\,] \\ map(f, x : xs) & \to & apply(\bullet(f), x) : map(f, xs) \\ inc(x) & \to & s(x) \\ apply(inc_0, x) & \to & inc(x) \end{array}$$

aquí, se utiliza un operador de aplicación explícito *apply* junto con la aplicación de función parcial inc_0 (un símbolo constructor).

Obsérvese que, en este ejemplo, se ha anotado la ocurrencia de la variable f que está más a la izquierda, en la tercera regla del programa. Esto es esencial para obtener una definición de primer orden $map(inc_0, x)$. Por otro lado, anotando la segunda ocurrencia de la variable f, el evaluador parcial devuelve básicamente el programa original.

Dado el término inicial $minc(x)$, se construye el árbol de *narrowing* necesario generalizante mostrado en la Figura 3.4. Nótese que $apply(w, y)$ no se reduce más puesto que, como se mencionó antes, esta llamada de orden superior contiene una variable funcional libre y, de este modo, su evaluación se suspende (lo cual significa que la definición original de *apply* debería ser incluida en el programa residual).

Dado el siguiente renombramiento independiente:

$$\begin{array}{rcl} \rho = \{ \quad minc(x) & \mapsto & minc(x), \\ map(inc_0, ys) & \mapsto & mapinc(ys), \\ inc(y) & \mapsto & inc(y), \\ apply(w, y) & \mapsto & apply(w, y) \quad \} \end{array}$$

[3]Tal como en el lenguaje Curry, no se permite la evaluación de llamadas de orden superior incluyendo variables libres que actúen como funciones (es decir, tales llamadas son *suspendidas* para evitar la aplicación de unificación de orden superior). En [AT99] se encuentra una estrategia más flexible.

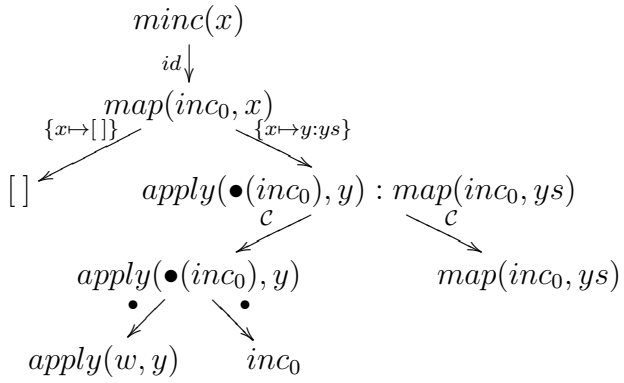


Figura 3.4: Árbol de *narrowing* necesario generalizante para $minc(x)$.

el SRT residual calculado por el NPE offline es como sigue:

$$\begin{array}{rcl} minc(x) & \to & mapinc(x) \\ mapinc([\,]) & \to & [\,] \\ mapinc(y : ys) & \to & apply(inc_0, y) : mapinc(ys) \\ inc(y) & \to & s(y) \\ apply(inc_0, y) & \to & inc(y) \end{array}$$

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

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

donde el operador de aplicación explícito *apply* ya no es necesario. Nótese que esta transformación, con frecuencia, logra mejoras significativas en el tiempo de ejecución de los programas (véase, e.g., [AHV02]).

3.5.3. Evaluación experimental

El método NPE offline esbozado en la Sección 3.5.1 ha sido implementado en el lenguaje declarativo multi-paradigma Curry [Han06]. Los programas fuente del evaluador parcial y la explicación detallada de los resultados considerados a continuación, están disponibles públicamente en:

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

La herramienta NPE es puramente declarativa y acepta programas Curry incluyendo funciones de orden superior, algunas funciones propias del sistema *(built-ins)*, etc.

El cuadro 3.1 muestra los resultados en cuanto a mejora en los tiempos de ejecución y que considera los siguientes ejemplos:

`ackermann`: Esta es la conocida función de Ackermann especializada para un argumento de entrada cuyo valor es mayor o igual a 10.

`allones`: El objetivo de este ejemplo es producir automáticamente una función nueva que transforme cada elemento de una lista a "1" calculando primero la longitud de la lista original y, después, construyendo una nueva lista de la misma longitud cuyos elementos sean "1". Este es un típico ejemplo de deforestación [Wad90].

`fliptree`: Otro típico ejemplo de deforestación. Aquí, el objetivo es invertir dos veces una estructura de árbol de tal forma que se obtiene la estructura de árbol original. No se proporcionan valores estáticos de entrada.

`foldr.allones`: El objetivo de este ejemplo es obtener la especialización de una función que concatene cierto número de listas y, después, transformar todos los elementos a "1". La función original se define por medio del combinador de orden superior *foldr*. El proceso de especialización considera que una de las listas es conocida.

`foldr.sum`: En este ejemplo, se produce una función especializada para sumar los elementos de una lista (con un prefijo dado) usando la función de orden superior *foldr*.

`fun_inter`: Este ejemplo consiste en la especialización de un intérprete funcional simple para un programa dado.

`gauss`: El objetivo de este ejemplo es la especialización de la conocida función de Gauss para números naturales mayores o iguales a 5.

`kmp_matcher`: Es un comparador de patrones especializado para un patrón dado. El ejemplo es conocido como el test "KMP" [CD89].

Cuadro 3.1: Resultados de estándares de comparación

benchmark	codesize (bytes)	onlineNPE (ms.)	speedup1 (*online*)	offlineNPE		speedup2 (offline)
				ann (ms.)	mix (ms.)	
ackermann	1496	20290	1.006	100	590	4.750
allones	1191	180	1.065	50	200	1.050
fliptree	1861	1940	0.985	100	240	0.977
foldr.allones	2910	3633	1.024	120	430	2.034
foldr.sum	3734	6797	1.311	170	3340	1.293
fun_inter	4266	28955	—	160	5190	—
gauss	1241	11090	1.040	100	757	1.013
kmp_matcher	3222	11670	5.346	157	9410	1.219
power	1693	160	3.087	110	280	1.012
Average	**2402**	**9413**	**1.858**	**119**	**2271**	**1.668**

`power`: Considera el ejemplo de especialización mostrado en la Sección 3.5.2 para un exponente constante de 6.

Para cada ejemplo, se muestra el tamaño (en bytes) del programa (codesize), el tiempo de ejecución de la herramienta NPE *online* (`onlineNPE`), el tiempo de ejecución de la nueva herramienta NPE *offline* descrita aquí (`offlineNPE`), donde se muestra tanto los tiempos para analizar y anotar el programa original (`ann`) como para ejecutar los cómputos parciales y extraer el programa residual (`mix`), así como la relación de mejora alcanzada por los programas especializados con cada esquema de especialización (`speedup1` y `speedup2`). Las relaciones de mejora están dadas por *orig*/*spec*, donde *orig* y *spec* son tiempos de ejecución absolutos de los programas originales y especializados, respectivamente. Los tiempos son expresados en milisegundos y son el promedio de 10 ejecuciones sobre una PC-Linux a 2.4 GHz (Intel Pentium IV con 512 KB cache). Los datos de entrada fueron seleccionados para producir tiempos de ejecución razonablemente grandes. Los programas fueron ejecutados con el compilador de Curry a Prolog (curry2prolog) de PAKCS [HeAE+04].

Como puede verse en el Cuadro 3.1, se han reducido los tiempos de evaluación parcial a un 20 % aproximadamente respecto a la herramienta NPE original, lo que significa que el principal objetivo se ha alcanzado. Respecto a la relación de mejora, se aprecia que la mayoría son problemas de *especialización* (en lugar de considerar problemas de *optimización*), lo cual explica los buenos resultados alcanzados por la herramienta NPE offline. Sin embargo, el nuevo método no es capaz de pasar el conocido

test "KMP" [CD89] (véase el ejemplo `kmp_matcher`). Existen dos requerimientos esenciales para pasar el test KMP: una buena propagación de información y un análisis de terminación más potente que evite un alto grado de generalización. En tanto que la aproximación *offline* propaga la información lo mismo que el esquema *online* (que sí pasa el test KMP), el análisis de terminación (implícito) del esquema *offline* es mucho más simple. Sería interesante verificar si una aproximación combinada *online*/*offline* es más útil. El actual evaluador parcial *offline* trata adecuadamente con funciones aritméticas (`ackermann`), con la simplificación de llamadas de orden superior (ejemplos `foldr.allones` y `foldr.sum`), y con un intérprete funcional simple (ejemplo `fun_inter`), donde las relaciones de mejora no son mostradas puesto que el tiempo de ejecución de los programas especializados es cero (es decir que, el programa de entrada para el intérprete ha sido completamente evaluado).

3.6. Trabajo relacionado y discusión

A pesar de la relevancia de *narrowing* como mecanismo de computación simbólica, se ha encontrado poco trabajo dedicado a analizar su terminación. Por ejemplo, Dershowitz and Sivakumar [DS88] definieron un procedimiento de *narrowing* que incorpora una poda de algunos objetivos no satisfacibles. Otras aproximaciones similares han sido presentadas por Chabin and Réty [CR91], donde *narrowing* está dirigido por un grafo de términos, y por Alpuente et al. [AFRV93], donde se introdujo el concepto de *loop-check*. También Antoy y Ariola [AA97] introdujeron un tipo de técnica de memorización[4] para lenguajes lógico funcionales tal que, en algunos casos, se logra obtener una representación finita de un espacio de *narrowing* infinito. Todas estas técnicas son *online*, ya que usan información acerca del término que está siendo especializado por *narrowing*. Por otro lado, Christian [Chr92] introdujo una caracterización de SRTs para los cuales *narrowing* termina. Básicamente éste requiere que las partes izquierdas sean *flat*, es decir, que todos los argumentos sean

[4]De la definición en inglés de *memoization* como técnica de optimización usada para acelerar la ejecución de programas almacenando los resultados de las llamadas a función para su uso posterior en lugar de recalcularlos en cada petición de la función.

variables o términos básicos (*ground*). Ninguno de éstos trabajos consideró la *cuasi-termination* ni presentó un método para anotar SRTs con el fin de forzar su terminación.

Otros trabajos relacionados vienen de la extensa literatura sobre evaluación parcial. Dentro del paradigma de programación lógica, Decorte et al. [DDL+97] investigaron la cuasi-terminación de programas lógicos *con memorización* para trasladar las técnicas de especialización de programas lógicos "estándar" a programas lógicos *con memorización*. Ellos introdujeron la caracterización de programas *cuasi-aceptable* y demostraron que esta clase de programas garantiza cuasi-terminación. Sin embargo, no es sencillo determinar si un programa es *cuasi-aceptable* (los autores bosquejaron cómo podría ser extendido el análisis de terminación estándar).

Dentro del escenario funcional, Holst [Hol91] introdujo una condición suficiente de cuasi-terminación para asegurar la terminación de un proceso de evaluación parcial (el cual fue usado, en su momento, por Glenstrup y Jones [GJ96] para definir el algoritmo de un BTA garantizando la terminación de su proceso de evaluación parcial *offline*). Holst presentó adicionalmente un análisis estático basado en interpretación abstracta para verificar la condición de suficiencia por cuasi-terminación. De modo semejante a [DDL+97], las condiciones presentadas están basadas en semántica, de tal forma que son difíciles de analizar.

En contraste, la aproximación no-creciente se apoya en una sencilla caracterización *sintáctica* la cual es generalmente menos precisa pero muy fácil de verificar. De hecho, las aproximaciones más cercanas a este trabajo son las caracterizaciones sintácticas dadas por Wadler [Wad90] y por Chin y Khoo [CK96], las cuales ya han sido discutidas en la Sección 3.3.

En resumen, [RSV05a] presentaron una nueva caracterización para SRTs que asegura la cuasi-terminación de computaciones de *narrowing* necesario. Este es un problema, difícil de interés particular, que no había sido atacado anteriormente. Dado que la clase de SRTs considerada es muy restrictiva, consideraron entonces los programas inductivamente secuenciales —una clase mucho más amplia— e introdujeron un algoritmo que anota aquellos subtérminos que pueden causar la no cuasi-terminación del *narrowing* necesario. También establecieron una extensión generalizante del *narrowing* necesario la cual esta dirigida por ano-

taciones agregadas al programa. Finalmente, describen cómo se usan los nuevos desarrollos para definir un esquema NPE preciso que garantice terminación en el proceso *offline*. Los experimentos preliminares orientados sobre una amplia variedad de programas son alentadores y demuestran la utilidad de la presente aproximación.

Aunque se están considerando sistemas inductivamente secuenciales como programas y *narrowing necesario* [AEH00] como semántica operacional, los desarrollos de [RSV05a] podrían ser fácilmente extendidos a sistemas *narrowing* inductivamente secuenciales e inductivamente secuenciales *solapantes* [Ant97]. La principal diferencia es que los sistemas solapantes permiten el uso de un operador de disyunción explícito el cual introduce indeterminismo de tipo "*don't-know*". En este contexto, introducir una función con una disyunción en la parte derecha, e.g., $f(x) \rightarrow t_1$ *or* t_2, es básicamente equivalente a escribir las siguientes dos reglas individuales:

$$f(x) \rightarrow t_1$$
$$f(x) \rightarrow t_2$$

Puesto que la terminación de la caracterización no-creciente depende principalmente sobre cómo cambian los parámetros de función desde la parte izquierda a la parte derecha de la regla, el tratamiento de disyunciones en los sistemas solapantes no presenta problemas adicionales; básicamente, un operador de disyunción se puede considerar como un símbolo constructor.

La supercompilación positiva [SGJ96b] comparte muchas similitudes con NPE ya que *driving*, el mecanismo de computación simbólica de la supercompilación positiva, es equivalente a *narrowing* necesario en programas equiparables. Por lo tanto, los resultados podrían ser transferidos fácilmente al escenario de la supercompilación positiva.

Uno de los enfoques más recientes para garantizar la cuasi-terminación de programas funcionales está basado en los grafos de cambio de tamaño (los grafos *size-change* [LJBA01] ya han sido usados en el contexto de la evaluación parcial en [JG05]). De este modo una cuestión interesante es el uso de grafos de cambio de tamaño para definir un algoritmo de anotación más preciso —aunque computacionalmente más costoso. En el siguiente capítulo se muestra una extensión del evaluador parcial *offline*,

donde se reemplazan los SRTs no-crecientes por los SRTs EP-terminantes [ARSV06a] que se caracterizan así a partir de la extensión del esquema NPE *offline*, la cual adopta los grafos *size-change*, implementados originalmente sólo para programas funcionales, a nuestro método de anotación que considera programas lógico funcionales. Por otro lado, el algoritmo de anotación para SRTs de este capítulo es independiente del término considerado para la evaluación parcial. Esto significa que un SRT necesita ser anotado sólo una vez, y después puede ser evaluado parcialmente, con respecto a diferentes términos, sin calcular nuevas anotaciones. Sin embargo, esto también significa que no se está explotando la estructura conocida del término considerado para la evaluación parcial, es decir, la cualidad estática de los términos (para mayor claridad véase el siguiente capítulo). Por lo tanto, sería interesante estudiar la combinación de la primera etapa, descrita en este capítulo, con un proceso tradicional de análisis de tiempo de enlace[5] (BTA) en el que se propagan los valores de los datos estáticos. Aquí, el contexto *lógico* funcional presenta nuevas demandas para el BTA debido al uso de variables lógicas y funciones indeterministas. Con este propósito se planea investigar técnicas para el análisis de tiempo de enlace de programs lógicos dentro de la estrategia seguida en la *deducción parcial* (tales como, e.g., [CGLH05, LJVB04]).

[5] Véase el capítulo 4.

Capítulo 4

Análisis de tiempo de enlace

Cuando algunos de los datos de entrada de un programa son conocidos en tiempo de compilación, ciertas expresiones, que dependen de ellos, pueden ser detectadas y evaluadas en una fase previa a la ejecución del programa; esta es la base del concepto de evaluación parcial. La identificación anticipada de estos cómputos, independientemente de los valores actuales de los datos de entrada, pueden ser determinados por un análisis estático llamado `análisis de tiempo de enlace`, proceso conocido en inglés como: *Binding Time Analysis* (BTA). En este capítulo documentaremos el desarrollo de un BTA con el propósito de mejorar la eficiencia de un evaluador parcial dirigido por *narrowing*. Prácticamente, nuestro análisis (BTA) incluye la implementación de un principio de terminación y una clasificación de variables conocidas y no conocidas del programa a especializar. La información del BTA nos permite hacer anotaciones sobre un programa fuente. Después, para evaluar parcialmente un programa, simplemente seguimos las anotaciones, es decir, los datos de tiempo de enlace: evaluando algunas expresiones en tiempo de compilación y aplazando otras a tiempo de ejecución [Con90]. Los desarrollos y conceptos que presentamos en este capítulo están publicados en [ARSV06a, ARSV07, ARTV07]

4.1. Introducción

En este capítulo, presentamos un perfeccionamiento en la caracterización de los sistemas de reescritura no crecientes empleando un formalismo denominado *grafos size-change* [LJBA01], los cuales rastrean los cambios en la talla de los parámetros en las llamadas a función. Más detalladamente, usamos la información de los grafos *size-change* para identificar una forma particular de cuasi-terminación, es decir, en la que sólo pueden producirse un número finito de *llamadas a función* diferentes (módulo renombramiento de variables) en una computación. Con este objetivo, se utiliza también el resultado de un análisis BTA para contar con la información sobre cuáles argumentos de función son *estáticos* (y por lo tanto conocidos) o *dinámicos*. Cuando la información recolectada de los grafos *size-change* en conjunto con la que nos devuelve el BTA, no permite inferir que el sistema de reescritura cuasi-termina, entonces se procede como en [RSV05b] y se anotan los subtérminos problemáticos para ser generalizados en tiempo de evaluación parcial. Más adelante, en la sección 4.4 se presenta una tabla de resultados obtenidos a partir de procesar un conjunto de programas de prueba en el nuevo esquema.

Trabajo relacionado

Respecto al concepto de cuasi-terminación, encontramos relativamente pocos trabajos dedicados al análisis de cuasi-terminación de programas lógicos o funcionales (y ningún trabajo previo en relación a la cuasi-terminación de programas lógico funcionales). Dershowitz [Der87] introdujo originalmente el concepto de cuasi-terminación, donde una derivación de reescritura es llamada cuasi-terminante cuando ésta contiene una cantidad finita de términos diferentes. En el ámbito de la programación lógica, una de las primeras aproximaciones se encuentra en [DDL+98], donde los autores introducen el concepto de *cuasi-aceptabilidad*, una condición suficiente y necesaria de cuasi-terminación. Este trabajo ha sido extendido en [VSD01].

Por lo que respecta al análisis *size-change*, tal aproximación se introdujo originalmente en [LJBA01] dentro del contexto de la programación funcional. Este esquema fue adaptado más tarde a la reescritura de tér-

minos en [TG05].

Finalmente, si consideramos el uso del análisis de cuasi-terminación para asegurar la terminación de la evaluación parcial *offline*, existen pocas aproximaciones. No obstante, desde hace mucho tiempo se reconoció a la cuasi-terminación como una propiedad esencial para garantizar la terminación de la evaluación parcial (véase, e.g., el estudio precursor de Holst [Hol91]). En particular, nosotros compartimos muchas semejanzas con la aproximación introducida por Glenstrup y Jones [GJ05], donde se usa un análisis de cuasi-terminación basado en los grafos *size-change* para asegurar la terminación de un evaluador parcial *offline* para programas funcionales de primer orden. Sin embargo, trasladar el esquema de Jones y Glenstrup a programación lógico funcional no es un proceso inmediato debido a que los cómputos de *narrowing* propagan hacia adelante los enlaces dentro de los propios cómputos (tal como se hace en programación lógica). Como consecuencia, es necesario introducir algunas condiciones adicionales con el fin de preservar la terminación de la evaluación parcial. Además, consideramos grafos *size-change* más simples (esto es, la relación *"may-increase"* de [GJ05] no se usa en este trabajo). Esto puede debilitar de algún modo el alcance de nuestro analisis *size-change*, pero podría extenderse directamente de la misma forma que [GJ05].

Toda vez que hemos visto la aproximación original de evaluación parcial *offline* dirigida por *narrowing* en el capítulo 3, el presente capítulo esta estructurado como sigue: Primeramente, en la sección 4.2 introducimos un análisis de cuasi-terminación basado en los grafos *size-change* y establecemos el principal resultado de este capítulo. La sección 4.3 presenta el procedimiento de anotación y se ilustra con un ejemplo. La sección 4.4 describe una evaluación experimental de nuestra aproximación utilizando un prototipo de evaluador parcial *offline*, donde se muestra una significativa relación de mejora del nuevo EP *offline* con respecto EP *offline* introducido por [RSV05b]. A partir de la sección 4.5 hasta la sección 4.9 describimos más detalladamente la implementación del evaluador parcial *offline* para el lenguaje *FlatCurry* (la representación intermedia del lenguaje lógico funcional Curry [Han06]), incluyendo sendas definiciones formales del procedimiento de anotación, las que culminan en dos estrategias de especialización que hemos denominado *offline pura e híbri-*

da. Adicionalmente introducimos, la especificación formal de la extensión del cálculo RLNT de ambas estrategias. Aquí mismo mostramos los resultados de la puesta en práctica de ambas estrategias *offline pura e híbrida* con la implementación de dos especializadores que a su vez se comparan con un EP *online*. Finalmente, la sección 4.10 concluye y apunta algunas líneas de trabajo que recientemente hemos explorado. Mayores detalles de las demostraciones no provistas pueden ser consultadas en [ARSV06b].

4.2. Garantizando cuasi-terminación con grafos *size-change*

En esta sección, retomaremos algunos conceptos básicos de [TG05] sobre los grafos *size-change*, donde se adapta el esquema de [LJBA01] a reescritura de términos, después, introduciremos nuestra nueva aproximación para garantizar cuasi-terminación.

Decimos que una relación binaria transitiva y antisimétrica $\succ$ representa un *orden* y una relación binaria transitiva y reflexiva $\succsim$ representa un *cuasi-orden*. Una relación binaria $\succ$ está bien fundada *si y sólo si* (*sii*[1]) no existe una secuencia decreciente infinita $t_0 \succ t_1 \succ t_2 \succ \ldots$ En adelante, diremos que dado un orden "$\succ$" *es cerrado bajo sustituciones* (o *estable*) si $s \succ t$ implica que $\sigma(s) \succ \sigma(t)$ para todo $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ y toda sustitución σ.

Los grafos *size-change* se parametrizan por un llamado par de reducción:

Definición 16 (par de reducción) *Decimos que $(\succsim, \succ)$ es un par de reducción si $\succsim$ es un cuasi-orden y $\succ$ es un orden bien fundado sobre términos donde ambos $\succsim$ y $\succ$ son cerrados bajo sustituciones y compatibles (esto es, $\succsim \circ \succ \subseteq \succ$ y $\succ \circ \succsim \subseteq \succ$ pero $\succsim \subseteq \succ$ no es necesario, donde "$\circ$" esta definido sobre las relaciones binarias R y R' como sigue: $R \circ R' = \{(a, c) \mid (a, b) \in R \text{ y } (b, c) \in R'\}$). También requerimos que $s\,R\,t$ implica que $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ para todo $R \in \{\succsim, \succ\}$ y todo término s y t.*

[1] En adelante abreviamos *si y sólo si* como *sii*.

Hablando informalmente, la restricción $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ anterior es necesaria con el fin de propagar correctamente información *groundness*[2] a través de las derivaciones *narrowing*.

Definición 17 (grafo *size-change*) *Sea $(\succsim, \succ)$ un par de reducción. Para cada regla $f(\overline{s_n}) \to r$ de un SRT $\mathcal{R}$ y cada subtérmino $g(\overline{t_m})$ de r donde $g \in \mathcal{D}$, entonces definimos un grafo size-change como sigue:*

- *El grafo tiene n nodos de salida especificados con $\{1_f, \ldots, n_f\}$ y m nodos de entrada especificados con $\{1_g, \ldots, m_g\}$.*
- *Si $s_i \succ t_j$, entonces hay un arco denotado con $\succ$ desde i_f a j_g. De otra forma, si $s_i \succsim t_j$, entonces hay un arco denotado con $\succsim$ desde i_f a j_g.*

De este modo un grafo size-change *es un grafo etiquetada bipartito $G = (V, W, E)$ donde $V = \{1_f, \ldots, n_f\}$ y $W = \{1_g, \ldots, m_g\}$ son las etiquetas de los nodos de salida y entrada, respectivamente, y tenemos que los arcos se obtienen a partir de $E \subseteq V \times W \times \{\succsim, \succ\}$.*

Los grafos *size-change* se usan para representar la forma en que cambia cada parámetro de función de la parte izquierda de la regla con respecto al mismo parámetro presente en alguna llamada a función en la parte derecha de la misma regla, denotada de acuerdo al par de reducción dado. Ahora, con el fin de analizar la terminación (o cuasi-terminación) de un programa, basta con centrar nuestra atención en sus bucles. Para cumplir con este propósito, calculamos la cerradura transitiva de sus grafos *size-change* como sigue:

Definición 18 (multigrafo, concatenación) *Cada gafo size-change de $\mathcal{R}$ es un multigrafo de $\mathcal{R}$ y si*

$$G = (\{1_f, \ldots, n_f\}, \{1_g, \ldots, m_g\}, E_1)$$

y

$$H = (\{1_g, \ldots, m_g\}, \{1_h, \ldots, p_h\}, E_2)$$

[2]En programación lógica *groundness information* es aquella que no contiene variables.

son multigrafos de $\mathcal{R}$ con respecto al mismo par reducción $(\gtrsim, \succ)$, entonces la concatenación

$$G \cdot H = (\{1_f, \ldots, n_f\}, \{1_h, \ldots, p_h\}, E)$$

también es un multigrafo de $\mathcal{R}$. Para $1 \leq i \leq n$ y $1 \leq k \leq p$, E contiene un arco de i_f a k_h sii E_1 contiene un arco de i_f a algún j_g y E_2 contiene un arco de j_g a k_h. Además, si alguno de los arcos está etiquetado con "$\succ$", entonces el arco en E también es etiquetado con "$\succ$". De otro modo, es etiquetado con "$\gtrsim$".

Un multigrafo G es idempotente si $G = G \cdot G$ (lo que implica que tanto sus nodos de entrada como de salida, ambos estén etiquetados con $\{1_f, \ldots, n_f\}$ para alguna f). En adelante, nos enfocaremos únicamente en los multigrafos idempotentes de un programa, ya que estos representan sus bucles (potenciales).

Ejemplo 4.1 *Considérese el siguiente ejemplo que devuelve los elementos de una lista en posición invertida respecto a una lista dada:*

$$\begin{array}{lcl} rev([\,]) & \to & [\,] \\ rev(x : xs) & \to & app(rev(xs), x : [\,]) \\ app([\,], y) & \to & y \\ app(x : xs, y) & \to & x : app(xs, y) \end{array}$$

donde "[]" y ":" son constructores de lista. En este ejemplo, consideramos un par de reducción en particular $(\gtrsim, \succ)$ que se define a continuación:

- $s \gtrsim t$ sii $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ y para todo $x \in \mathcal{V}ar(t)$, $dv(t,x) \leqslant dv(s,x)$;
- $s \succ t$ sii $\mathcal{V}ar(t) \subseteq \mathcal{V}ar(s)$ y para todo $x \in \mathcal{V}ar(t)$, $dv(t,x) < dv(s,x)$.

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

$$\begin{array}{rcll} dv(c(\overline{t_n}), x) & = & 1 + max(\overline{dv(t_n, x)}) & \text{if } x \in \mathcal{V}ar(c(\overline{t_n})) \\ dv(c(\overline{t_n}), x) & = & -1 & \text{if } x \notin \mathcal{V}ar(c(\overline{t_n})) \\ dv(y, x) & = & 0 & \text{if } x = y \\ dv(y, x) & = & -1 & \text{if } x \neq y \\ dv(t, x) & = & -1 & \text{if } t \text{ no es un término constructor} \end{array}$$

donde $c \in \mathcal{C}$ es un símbolo constructor con aridad $n \geqslant 0$. Entonces, los grafos size-change correspondientes al programa del ejemplo 4.1 son los siguientes:

$$G_1: \quad 1_{rev} \xrightarrow{\succ} 1_{rev} \qquad G_2: \quad 1_{rev} \overset{\succsim}{\searrow} \begin{matrix} 1_{app} \\ 2_{app} \end{matrix} \qquad G_3: \quad \begin{matrix} 1_{app} \xrightarrow{\succ} 1_{app} \\ 2_{app} \xrightarrow{\succsim} 2_{app} \end{matrix}$$

donde G_1 y G_3 son también los multigrafos idempotentes del programa.

Definición 19 (EP-terminación, SRT EP-terminante) *Un cómputo por narrowing necesario es EP-terminante si el desplegado de funciones genera sólo un número finito de llamadas a función (es decir, de redexes) diferentes módulo renombramiento de variables. Un SRT es EP-terminante si cada posible cómputo por narrowing necesario es EP-terminante.*

Obsérvese que un SRT EP-terminante no asegura la cuasi-terminación de sus cómputos. Dado el SRT del ejemplo 4.1 y la llamada a función inicial *rev(xs)*, tenemos la siguiente derivación por *narrowing* necesario:

$$\begin{array}{lll} \underline{rev(xs)} & \leadsto_{\{xs \mapsto y:ys\}} & app(\underline{rev(ys)}, y : [\,]) \\ & \leadsto_{\{ys \mapsto z:zs\}} & app(app(\underline{rev(zs)}, z : [\,]), y : [\,]) \\ & \leadsto_{\{zs \mapsto w:ws\}} & \ldots \end{array}$$

Aunque esta derivación consta de un número infinito de términos diferentes, existe sólo un número finito de llamadas a función no variantes. Afortunadamente, este hecho es suficiente para asegurar la terminación en muchos esquemas de evaluación parcial ya que tales esquemas incluyen regularmente alguna forma de memorización[3].

Los métodos de evaluación parcial *online* usualmente consideran una distinción entre los llamados niveles de control *local* y *global*. El control *local* debe asegurar que las llamadas a función (o procedimiento) no sean desplegadas de manera infinita, mientras que el control *global* debe encargarse de no desplegar infinitamente varias llamadas a función (o procedimiento). De hecho, esta característica puede aplicarse tanto a

[3]Una técnica simple de *memoization* que memoriza los términos computados para evitar reevaluar una expresión.

evaluadores parciales *online* como *offline*. En algunos casos, esta distinción, se hace explicita (e.g., en el esquema de evaluación parcial para programas lógicos de [Gal93]). En algunos otros casos, la distinción se deja implícita.

En este capítulo, consideramos un procedimiento de evaluación parcial como se explica a continuación:

- *Control local* : aquí, suspendemos las derivaciones de *narrowing* necesario generalizante (es decir, aquellas derivaciones de *narrowing* necesario donde los subtérminos anotados son reemplazados por variables frescas) cuando la llamada a función seleccionada es una variante de una llamada a función previamente reducida dentro de la misma derivación. Obsérvese que nuestro control local examina las llamadas a función previas para determinar si una llamada a función debe ser desplegada o no. Esto no debe ser considerado una estrategia *online* sino una simple técnica de memorización. Adicionalmente, podríamos considerar estrategias de menor coste (aunque menos precisas) tales como, e.g., una estrategia de desplegado *depth-k* donde los cómputos de *narrowing* se suspenden después de k desplegados de función y no es necesario un chequeo de variantes.

- *Control global*: una vez que el desplegado de una llamada a función se detiene, los términos ubicados en las hojas del árbol de *narrowing* necesario generalizante, y que no son constructores, son aplanados completamente antes de agregarlos al conjunto de llamadas (a ser) evaluadas parcialmente. Por ejemplo, dado el término $f(g(x), h(y))$, las llamadas a función $f(w_1, w_2)$, $g(x)$ and $h(y)$ se agregan al conjunto actual de llamadas a función (a ser) evaluadas parcialmente, y donde w_1, w_2 son variables frescas. Este paso de aplanamiento se requiere para lograr la EP-terminación lo cual implica la terminación del proceso de evaluación parcial.

Ahora, consideramos que tenemos disponible el resultado de un análisis de tiempo de enlace (BTA) monovariante simple. Informalmente, dado un SRT y la información de cuáles parámetros de la llamada a función

inicial son estáticos y cuáles dinámicos, un BTA calcula los valores estático/dinámico para cada uno de los argumentos de cada función de programa. Aquí consideramos que un parámetro estático es absolutamente conocido en tiempo de especialización (por lo tanto es información *ground*), mientras que un parámetro dinámico es posiblemente desconocido en tiempo de especialización. La salida de un BTA debe ser *congruente* [JGS93]: el valor de cada parámetro estático esta determinado por los valores de otros parámetros estáticos (a la postre queda definido por la información de entrada disponible).

En lo sucesivo, requeriremos también que el componente $\succsim$ del par de reducción $(\succsim, \succ)$ sea de *particionamiento finito*[4], esto es, que el conjunto $\{s \mid t \succsim s\}$ debe contener un número finito de términos no variantes para cualquier término t. Otros conceptos estrechamente relacionados son *rigidez*[5] [BCF91] y *suficientemente instanciado*[6] [LS97], ambos definidos con respecto a la concepción de *norma simbólica*. Tales conceptos son usados en muchos análisis de terminación de programas lógicos (e.g., [CT99, GCGL02, LS97]).

El siguiente teorema establece las condiciones suficientes para asegurar la propiedad de EP-terminación de SRTs. La validez de la prueba está basada en el teorema de Ramsey [ARSV06b, Ram30].

Teorema 20 *Sea $\mathcal{R}$ un SRT y $(\succsim, \succ)$ un par de reducción. $\mathcal{R}$ es $EP-$ terminante con respecto a cualquier término lineal si cada multigrafo idempotente asociado a alguna función f/n, contiene, ya sea*

(i) al menos un arco $i_f \xrightarrow{\succ} i_f$ para algún $i \in \{1, \ldots, n\}$ tal que i_f es estático, o bien

(ii) un arco $i_f \xrightarrow{R} i_f$, $R \in \{\succsim, \succ\}$, para todo $i = 1, \ldots, n$, tal que i_f es estático y $\succsim$ es de particionamiento finito.

Demostración. Esta afirmación se demuestra por contradicción. Supongamos que el cálculo de un término (lineal) inicial dado t_0 produce una secuencia infinita de términos: $t_0 \rightsquigarrow t_1 \rightsquigarrow t_2 \rightsquigarrow \ldots$ con respecto a un

[4] De la definición en inglés *finitely partitioning* que es similar a la noción de *bounded*.
[5] Del concepto *rigidity* de [BCF91].
[6] Del concepto *instantiated enough* de [LS97].

programa $\mathcal{R}$ EP-terminante. Ahora, considerando un reordenamiento de esta derivación en la cual los *redexes* son instanciados en el orden más-izquierdo-más-interno (*leftmost-innermost*). Suponiendo además, que se calculan sólo unificadores más generales en este reordenamiento; restringidos a las variables de $\mathcal{V}ar(s_{i-1})$ (en lugar de calcular unificadores simples, es decir, sin anticipar enlaces tal como en λ [AEH00, Def. 13]). Esta derivación *más-interna* está denotada por $s_0 \leadsto_{p_1,R_1,\sigma_1} s_1 \leadsto_{p_2,R_2,\sigma_2} s_2 \leadsto_{p_3,R_3,\sigma_3} \cdots$

Tal como en [TG05], dados dos grafos *size-change* G and H donde los nodos de entrada de G tienen las mismas etiquetas que los nodos de salida de H, denotamos con $G \circ H$ al grafo que resulta a partir de identificar los nodos de entrada de G y los nodos de salida de H, esto es, $G \circ H$ únicamente difiere de $G \cdot H$ en que estos nodos no son eliminados.

Por cada paso, $s_j \leadsto s_{j+1}$, en la secuencia infinita, hay un grafo *size-change* G_j asociado. También, por cada j, tenemos $s_{j+1} = s_j[\theta_j(r_j)]_{p_j}$, donde $\delta_j = \sigma_j \cup \theta_j$ es el unificador más general de $s_j|_{p_j}$ y l_j, con $R_j = (l_j \to r_j)$ y $\sigma_j(s_j|_{p_j}) = \theta_j(l_j)$, es decir, los enlaces en σ_j no necesitan ser aplicados a $s_j[\theta_j(r_j)]_{p_j}$.

Puesto que $\mathcal{R}$ es EP-terminante, cada multigrafo maximal de una función $f \in \mathcal{R}$ con aridad n tiene, ya sea:

(i) un arco $i_f \xrightarrow{\succ} i_f$ para algún $i \in \{1, \ldots, n\}$, o bien

(ii) un arco $i_f \xrightarrow{R} i_f$ para todo $i = 1, \ldots, n$, con $R \in \{\succ, \succsim\}$.

En el primer caso, de manera similar a [TG05, Lema 6], el grafo formado por $G_0 \circ G_1 \circ G_2 \circ \ldots$ incluye una derivación infinita; donde aparecen un número infinito de arcos etiquetados con "$\succ$". Sin pérdida de generalidad, suponemos que ésta derivación comienza en G_0 y $s_0 = f(\ldots)$. Por cada j, sea a_j el nodo de salida en G_j el cual está en esta derivación.

Para $j = 0$, tenemos que $s_0 \leadsto s_1$, donde $l_0 \to r_0$ es una regla, $\theta = mgu(s_0|_{p_0}, l_0)$, $\theta_0(s_0|_{p_0}) = \theta_0(l_0)$, y $s_1|_{p_1} = \theta_0(r_0)|_q$ para alguna posición q. Supongamos que $(s_0|_{p_0})|_{a_0} \succ (s_1|_{p_1})|_{a_1}$ se mantiene (el caso $s_0|_{p_0} \succsim s_1|_{p_1}$ es análogo) en el grafo *size-change* asociado a la regla $l_0 \to r_0$. Por la estabilidad de $\succ$, tenemos que $\theta_0((s_0|_{p_0})|_{a_0}) \succ \theta_0((s_1|_{p_1})|_{a_1})$. Dado que $\theta_0((s_0|_{p_0})|_{a_0})$ es *ground* (estática); $\theta_0((s_1|_{p_1})|_{a_1})$ (por Definición 16) también lo es.

Aplicando este razonamiento una y otra vez tenemos que $s_j|_{a_j} \succ s_{j+1}|_{a_{j+1}}$ para todo $j \in J$ y $s_j|_{a_j} \succsim s_{j+1}|_{a_{j+1}}$ para todo $j \in I\!N \setminus J$, lo cual contradice lo bien fundamentado de "$\succ$".

La prueba del segundo caso es análoga considerando el hecho que $\succsim$ es de particionamiento finito.

□

La propiedad de particionamiento finito de "$\succsim$" en el caso (ii) del Teorema 20 es necesaria para asegurar que no se permiten secuencias infinitas de llamadas a función no variantes con argumentos de la misma "talla" de acuerdo a $\succsim$. Considérese, por ejemplo, un orden $\succsim$ el cual está basado en la longitud de una lista, es decir, $t_1 \succsim t_2$ si t_1 y t_2 son listas y el número de elementos de t_2 es menor o igual que el número de elementos de t_1. En este caso, $\succsim$ no es de particionamiento finito: considérese, e.g., el término $[x]$ tal que el conjunto $\{s \mid [x] \succsim s\}$ contiene un número infinito de términos no variantes. Por lo tanto, podemos tener secuencias infinitas de llamadas a función donde cada argumento es menor o igual al argumento de un elemento anterior en dicha secuencia:

$$f([x]) \rightsquigarrow f([succ(x)]) \rightsquigarrow f([succ(succ(x))]) \rightsquigarrow \ldots$$

con $[x] \succsim [succ(x)] \succsim [succ(succ(x))] \succsim \ldots$.

4.3. Procedimiento de anotación

En esta sección, introducimos nuestro procedimiento de anotación para el evaluador parcial dirigido por *narrowing*. De manera análoga a [RSV05b], en lugar de requerir programas fuente que satisfagan las condiciones del Teorema 20, utilizamos este resultado para determinar qué subtérminos (si hay alguno) violan las condiciones de este teorema.

El procedimiento de anotación procede como sigue: considera cada símbolo de función f/n relativo a un programa tal que f tenga un multigrafo idempotente asociado (es decir que, exista un bucle potencial que involucre a la función f), y ejecuta una de las siguientes acciones:

1. si las condiciones del Teorema 20 se cumplen, no se agregan anotaciones al programa;

2. de otra forma, cada argumento t_j de toda llamada a función $f(t_1, \ldots, t_j, \ldots, t_n)$ sin arco $j_f \xrightarrow{R} j_f$, $R \in \{\succsim, \succ\}$, se anota como sigue: $f(t_1, \ldots, \bullet(t_j), \ldots, t_n)$;[7]

3. finalmente, cada argumento dinámico se anota también.

Informalmente, la validez del procedimiento de anotación parte de los siguientes hechos:

- Consideremos una llamada a función f/n con un multigrafo idempotente asociado (nótese que, por el Teorema 20, la terminación puede garantizarse concentrándonos únicamente en aquellas funciones de programa que tengan un multigrafo idempotente asociado).

- Si las condiciones del Teorema 20 se cumplen, tenemos que partiendo de cada llamada $f(t_1, \ldots, t_n)$ a la siguiente llamada $f(s_1, \ldots, s_n)$ en un cómputo, se mantienen las siguientes condiciones:

  - existe algún $i \in \{1, \ldots, n\}$ tal que $t_i \succ s_i$ y el i-esimo argumento de f es estático (es decir, ambos t_i y s_i son *ground*; no contienen variables), lo cual significa que sólo pueden generarse un número finito de llamadas diferentes a f;[8]
  - de otra forma, tenemos que $t_i \succsim s_i$ o bien se anota s_i (y así el *narrowing* necesario generalizante reemplaza este argumento por una variable fresca) para todo $i = 1, \ldots, n$, lo cual significa que únicamente pueden ser generadas un número finito de llamadas no variantes a la función f ya que $\succsim$ es de particionamiento finito.

Ilustremos el proceso completo con un ejemplo.

Ejemplo 4.2 *Consideremos la bien conocida función Ackermann:*

$$\begin{array}{lcl} ack(zero, n) & \to & succ(n) \\ ack(succ(m), zero) & \to & ack(m, succ(zero)) \\ ack(succ(m), succ(n)) & \to & ack(m, ack(succ(m), n)) \end{array}$$

[7] Análogamente a [RSV05b], utilizamos un nuevo símbolo, denotado por $\bullet$, para anotar los subtérminos problemáticos los cuales deben ser generalizados en la fase de especialización (esto es, en el momento de la propia evaluación parcial).

[8] Este caso es similar al principio *bounded anchoring* de [GJ05].

Primero, calculamos los grafos size-change de este programa (aquí, consideramos el mismo par reducción del ejemplo 4.1):

$$G_1: \quad 1_{ack} \xrightarrow{\succ} 1_{ack} \qquad 2_{ack} \quad 2_{ack} \qquad G_2: \quad 1_{ack} \xrightarrow{\succ} 1_{ack} \qquad 2_{ack} \quad 2_{ack} \qquad G_3: \quad 1_{ack} \xrightarrow{\succeq} 1_{ack} \qquad 2_{ack} \xrightarrow{\succ} 2_{ack}$$

donde el grafo G_1 está asociado a la segunda regla y los grafos G_2 y G_3 están asociados a la tercera regla. En este ejemplo, G_2 y G_3 coinciden con los multigrafos idempotentes del programa.

Supongamos que deseamos especializar este programa con respecto a la llamada inicial $ack(succ(succ(succ(zero))), y)$, esto es, el primer argumento es estático (ground). Claramente el BTA devuelve la división $\{ack \mapsto [S, D]\}$, lo que significa que el primer argumento de cada llamada a ack es estático y el segundo argumento es dinámico. En este caso, tenemos que:

- *la primera condición del Teorema 20 se cumple para G_1 y G_2 ya que el primer argumento de ack es estático y existe un arco $1_{ack} \xrightarrow{\succ} 1_{ack}$, y*
- *la segunda condición del Teorema 20 no se cumple para G_3 ya que existe un arco asociado al argumento 2_{ack} el cual no es estático.*

4.4. Evaluación experimental

Hemos emprendido la implementación del procedimiento de anotación mejorado. En particular hemos incluido el nuevo procedimiento de anotación dentro de un evaluador parcial *offline* para programas Curry [RSV05b]. Este evaluador parcial ha sido implementado asimismo en lenguaje Curry [Han06]. En dicha implementación sólo se considero un subconjunto de Curry. La extensión de algunas otras características de Curry (e.g., *constraints*, funciones de orden superior, *builtin's*, etc.) se han implementado en el primer prototipo para realizar la compilación por evaluación parcial de programas lógico funcionales (esto es, la primera aproximación de compilación por evaluación parcial de lenguaje Curry) los detalles serán explicados en el capítulo 6. Los programas fuente del

evaluador parcial correspondiente a los avances especificados en este capitulo, así como la explicación detallada de los ejemplos considerados, están disponibles públicamente en:

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

El cuadro 4.1 muestra los resultados de algunos ejemplos. Para cada ejemplo, mostramos su tamaño en bytes (`codesize`), el tiempo de ejecución del programa original, (`original`) el tiempo de ejecución del programa residual especializado con el evaluador parcial *offline*; el cual usa el procedimiento de anotación simple (`simple peval`), el tiempo de ejecución del programa residual producido con el evaluador parcial; el cual incluye el nuevo procedimiento de anotación (`improved peval`) y la relación de mejora en la rapidez de ejecución alcanzada por cada evaluador parcial. Las relaciones de mejora están dadas por *orig*/*spec*, donde *orig* y *spec* son los tiempos de ejecución absolutos de los programas originales y especializados, respectivamente. Los tiempos se expresan en milisegundos y son el promedio de 10 ejecuciones realizadas sobre una PC-Linux a 2.4 GHz (Intel Pentum IV con 512 KB de memoria cache). Todos los programas (incluyendo los evaluadores parciales) fueron ejecutados con el compilador de Curry a Prolog de PAKCS [HeAE+04]. Como puede verse en el cuadro 4.1, los programas residuales obtenidos con la ejecución del evaluador parcial mejorado son (en promedio) 7 % más rápidos que los programas residuales obtenidos con el evaluador parcial *offline* anterior, sin embargo, esto equivale a una reducción cercana al mitad del tiempo de ejecución original. Esta no es una mejora espectacular pero demuestra que el nuevo procedimiento de anotación es capaz de producir programas especializados más rápidos. Notamos que el evaluador parcial es relativamente simple (esto es, siguiendo la estrategia mencionada en la Sección 4.2). Esperamos producir programas residuales aún más rápidos mejorando los procedimientos de control involucrados en la fase de especialización.

benchmark	codesize	original	simple peval	speedup1	improved peval	speedup2
ackermann	739	3363	1077	3.12	688	4.89
allones	662	1522	1444	1.05	1452	1.05
dec_list	825	589	587	1.00	525	1.12
gauss	2904	308	320	0.96	252	1.22
inc_list	817	937	834	1.12	730	1.28
insert_sort	1005	1953	1280	1.53	1322	1.48
kmpA*B	30580	428	298	1.44	227	1.89
kmpB*A	30582	86	80	1.08	72	1.21
power	794	591	602	0.98	571	1.03
Average	**7656**	**1086**	**725**	**1.36**	**649**	**1.68**

Cuadro 4.1: Tiempos de ejecución de los programas de prueba.

4.5. Implementación del evaluador parcial *off-line*

A partir de esta sección presentamos un procedimiento de anotación basado en el análisis de cuasi-terminación de la sección 4.2 que es más intuitivo y apegado a la práctica ya que mostramos las estrategias de control que pueden ser usadas para diseñar un poderoso evaluador parcial dirigido por *narrowing*, para lo cual en la siguiente sección presentamos el lenguaje *flat* y posteriormente en la sección 4.7 adaptamos los resultados del análisis de la sección 4.2 al lenguaje *flat*. Después introducimos un algoritmo de especialización que distingue dos niveles de control diferentes. El nivel *global* garantiza que el número de las diferentes funciones especializadas se mantiene finito. El nivel *local* toma una llamada a función y construye una evaluación finita (posiblemente parcial) de ésta llamada. El método resultante es *offline* puro y de esta manera muy eficiente [ARTV07]. Finalmente, también discutimos un algoritmo híbrido que incluye algunos tests simples durante la evaluación parcial, de forma que la calidad de los programas residuales pueda ser mejorada.

4.6. El lenguaje

En esta sección, presentamos la sintaxis de los programas *flat* [HP99], una representación estándar para programas lógico funcionales la cual hace explícita la estrategia de emparejamiento de patrones por medio de

expresiones *case*. Esta representación *flat* constituye el núcleo de lenguajes lógico funcionales modernos como Curry [Han06] o Toy [LS97]. Representaciones similares son consideradas en [HP99, HGU01, LK99]. A diferencia de éstas, consideramos dos clases de expresiones *case flexible/rigid* para representar anotaciones de evaluación de programas fuentes. Dado que los programas inductivamente secuenciales [Ant92] (con anotaciones de evaluación) pueden ser automáticamente traducidos a su representación *flat*, nuestro enfoque cubre propuestas recientes para la programación multi-paradigma lógico funcional. La sintaxis para programas en la representación *flat* es como sigue:

$$
\begin{array}{lcll}
P & ::= & \mathcal{E}_1 \ldots \mathcal{E}_m & \text{(programa)} \\
\mathcal{E} & ::= & f(\overline{x_n}) = e & \text{(definición de función)} \\
e & ::= & x & \text{(variable)} \\
 & | & c(\overline{e_n}) & \text{(llamada a constructor)} \\
 & | & f(\overline{e_n}) & \text{(llamada a función)} \\
 & | & \mathit{case}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} & \text{(case rigido)} \\
 & | & \mathit{fcase}\ e\ \mathit{of}\ \{\overline{p_n \to e_n}\} & \text{(case flexible)} \\
p & ::= & c(\overline{x_n}) & \text{(patrón)}
\end{array}
$$

De esta forma, un programa P consiste de una secuencia de definiciones de función $\mathcal{E}$ tales que la parte izquierda es lineal y tiene únicamente variables como argumentos, esto es, el emparejamiento de patrones se compila a expresiones *case*. La parte derecha de cada definición de función es una expresión e compuesta por variables ($\mathcal{V}$), constructores ($\mathcal{C}$), llamadas a función ($\mathcal{D}$), y expresiones *case* para emparejamiento de patrones. Las variables son denotadas por $x, y, z, \cdots$, los términos constructores por $a, b, c, \cdots$, y la definición de funciones por $f, g, h, \cdots$. La forma general de una expresión case es:

$$
\begin{array}{rl}
(f)\mathit{case}\ e\ \mathit{of}\ \{ & c_1(\overline{x_{n_1}}) \to e_1 \quad ; \\
 & \cdots \quad ; \\
 & c_k(\overline{x_{n_k}}) \to e_k \quad \}
\end{array}
$$

donde e es una expresión, $c_1, \ldots, c_k$ son constructores diferentes del tipo de e, y $e_1, \ldots, e_k$ son expresiones (conteniendo posiblemente estructuras *case*). Las variables $\overline{x_{n_i}}$ son variables locales cuya ocurrencia se presenta

únicamente en la expresión correspondiente e_i. La diferencia entre *case* y *fcase* se hace patente cuando el argumento e es una variable libre: mientras la ejecución de *case* suspende (lo cual corresponde a residuación), *fcase* enlaza de forma indeterminista dicha variable al patrón en una rama del *fcase* y procede con la evaluación de dicha rama (lo cual corresponde al *narrowing*). Las funciones definidas por expresiones *fcase* o *case* son llamadas *flexible* o *rigid*, respectivamente.

Una expresión es *operation-rooted* si está encabezada por un símbolo de definición de función. Y es *constructor-rooted* si el símbolo que la encabeza es un símbolo constructor.

Por ejemplo, la función (flexible) "app" para concatenar dos listas puede ser escrita en la representación *flat* mediante la siguiente regla:

$$\begin{aligned} \texttt{app}\ (\texttt{x},\texttt{y}) = \texttt{fcase x of}\ \{ & \\ [\,] &\to \texttt{y}; \\ (\texttt{z} : \texttt{zs}) &\to \texttt{z} : \texttt{app}\ (\texttt{zs},\texttt{y});\ \} \end{aligned}$$

La semántica operacional de los programas *flat* está basada en el cálculo LNT (*Lazy Narrowing with definitional Trees*) [HP99]. En la Sección 4.8.2 representamos una ligera extensión de esta semántica para ejecutar computaciones en tiempo de evaluación parcial.

4.7. Análisis de cuasi-terminación y anotación de programas *flat*

Primero adaptamos el análisis de cuasi-terminación de la sección 4.2, introducido originalmente para sistemas de reescritura de términos, al lenguaje *flat*. Posteriormente presentamos un procedimiento de anotación para programas *flat* que está basado en éste análisis de cuasi-terminación.

De manera similar al análisis de cuasi-terminación de SRTs; una relación binaria transitiva y antisimétrica $\succ$ representa un *orden* y una relación binaria transitiva y reflexiva $\succsim$ representa un *cuasi-orden*. Una relación binaria $\succ$ está bien fundada *sii* no existe una secuencia decreciente infinita $t_0 \succ t_1 \succ t_2 \succ \cdots$. Y diremos que un orden "$\succ$" *es cerrado bajo sustituciones* (o *estable*) si $s \succ t$ implica que $\sigma(s) \succ \sigma(t)$ para todos los términos s, t y toda sustitución σ.

$$pairs(l,e) = \begin{cases} \bigcup_{i=1}^{k} pairs(\{x \mapsto p_i\}(l), e_i) & \text{if } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\}, \\ \{(l,r) \mid r \text{ es un subtérmino de } e \\ \quad \text{encabezado por operación } \} & \text{en caso contrario} \end{cases}$$

Figura 4.1: Función auxiliar *pairs*.

El análisis de cuasi-terminación para programas *flat* está basado en la definición 17 de grafo *size-change*, para adaptar el concepto original de grafo *size-change* a programas *flat*; tomamos la definición 16 de **par de reducción**, y necesitamos la siguiente definición auxiliar:

Definición 21 (pares de llamada) *Dada una definición de función* $f(\overline{x_k}) = e$*, tenemos que los pares* $pairs(f(\overline{x_k}), e)$ *conforman el conjunto asociado de* pares de llamada*, el cual se define inductivamente como se muestra en Figura 4.1.*[9]

Entonces los grafos *size-change* para programas *flat* se pueden especificar como sigue:

Para cada definición $f(\overline{x_n}) = e$ de un programa *flat* P y cada *par de llamada* $(f(\overline{s_n}), g(\overline{t_m})) \in pairs(f(\overline{x_k}), e)$ donde e es un subtérmino encabezado por operación, esto es, una *llamada a función*, tenemos un grafo *flat size-change* prácticamente igual al grafo de la definición 17. Con sus nodos de salida que dependen de la aridad de $f(\overline{s_n})$, esto es n y sus nodos de entrada comprendidos en la aridad de $g(\overline{t_m})$, esto es m.

Así pues un grafo *size-change* para un programa *flat* P se representa de la misma manera que para un SRT, esto es, con una representación gráfica bipartita etiquetada $G = (V, W, E)$ donde V y W representan los conjuntos de las etiquetas de los nodos de entrada y salida respectivamente y E el conjunto de los arcos formados por $V \times W \times \{\succsim, \succ\}$.

Con el fin de analizar la cuasi-terminación de un programa *flat*, bastará con centrarse en sus bucles. Para lograr este propósito, calculamos la cerradura transitiva de sus grafos *size-change*, de la misma forma como

[9] Aquí, asumimos que las expresiones *case* sólo ocurren en posiciones más externas (*outermost*). Esta es una suposición razonable ya que los programas *flat* obtenidos por traducción de programas fuente siempre la satisfacen [HP99].

$$
\begin{array}{rl}
d_1 \equiv \texttt{applast(xs,x)} & = \texttt{last(append(xs,[x]))} \\
d_2 \equiv \texttt{last(xs)} & = \texttt{fcase xs of \{ (y : ys)} \rightarrow \texttt{last}'\texttt{(ys,y) \}} \\
d_3 \equiv \texttt{last}'\texttt{(ys,y)} & = \texttt{fcase ys of \{ [ ]} \rightarrow \texttt{[y]; (w : ws)} \rightarrow \texttt{last(w : ws) \}} \\
d_4 \equiv \texttt{append(xs,ys)} & = \texttt{fcase xs of \{[ ]} \rightarrow \texttt{ys; (w : ws)} \rightarrow \texttt{w : append(ws,ys)\}}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{pairs}(d_1) & = & \{(\texttt{applast(xs,x)}, \texttt{append(xs,[x])}), \\
 & & \ (\texttt{applast(xs,x)}, \texttt{last(append(xs,[x]))})\} \\
\mathit{pairs}(d_2) & = & \{(\texttt{last(y : ys)}, \texttt{last}'\texttt{(ys,y)})\} \\
\mathit{pairs}(d_3) & = & \{(\texttt{last}'\texttt{(w : ws,y)}, \texttt{last(w : ws)})\} \\
\mathit{pairs}(d_4) & = & \{(\texttt{append(w : ws,ys)}, \texttt{append(ws,ys)})\}
\end{array}
$$

Figura 4.2: Ejemplo de deforestación `applast` y sus **pares de llamada**.

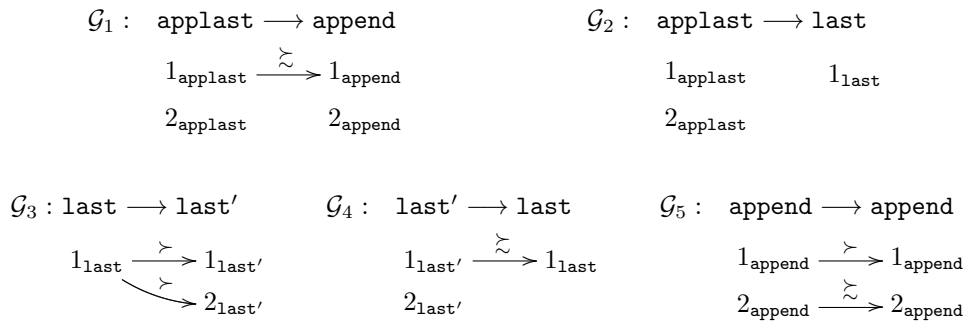


Figura 4.3: Grafos *size-change* de `applast`.

se indica en la Definición 18 considerando que se trata de programas *flat* en lugar de SRTs. Tal como en el análisis de cuasi-terminación de los SRTs, aquí también nos enfocaremos en los multigrafos idempotentes de un programa *flat*, puesto que éstos representan sus bucles potenciales.

Ejemplo 4.3 *Consideremos el ejemplo de deforestación mostrado en la Figura 4.2 y su conjunto de pares de llamada asociado* $\mathit{pairs}(l, e)$*. Este ejemplo es una ligera modificación del programa* `applast` *de la librería DPPD (Docena de Problemas para Deducción Parcial [Leu07]) para mejorar la ilustración del concepto de los grafos size-change de programas flat.*

Sea $(\succsim, \succ)$ *un par de reducción tal que* $s \succsim t$ *si* s *y* t *son iguales módulo renombramiento de variables y* $s \succ t$ *si* s *es un subtérmino estricto de* t *módulo renombramiento de variables. Entonces, tenemos cinco grafos*

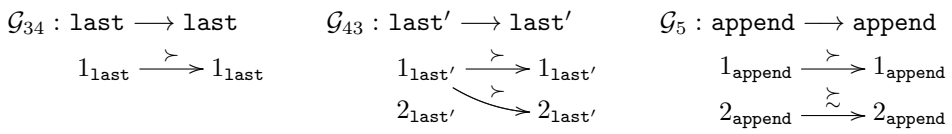


Figura 4.4: *Multigrafos* idempotentes de `applast`.

size-change mostrados en la Figura 4.3 asociados a las cinco pares de llamada. Finalmente, al igual que en sección 4.2, calculamos el cierre transitivo de los grafos size-change bajo el operador de concatenación y obtenemos los tres multigrafos idempotentes mostrados en la Figura 4.4.

Centraremos ahora nuestra atención en el concepto de terminación denominado *EP-terminación* especificado en la Definición 19 (un caso particular de cuasi-terminación), adaptado al contexto de los programas *flat*.

Definición 22 (Pflat EP-terminante) *Un programa flat es EP-terminante si cada posible computación es EP-terminante.*

Básicamente, una computación EP-terminante (posiblemente infinita) es una computación en la cual sólo un número finito de llamadas (módulo renombramiento de variables) a función son desplegadas.

4.7.1. Anotación de programas

En esta sección, mostramos un procedimiento de anotación basado en el análisis de cuasi-terminación presentado en la sección anterior.

La siguiente definición introduce nuestro procedimiento de anotación. Aquí, consideramos el programa P, el par reducción $(\succsim, \succ)$,[10] los multigrafos idempotentes de P, y el resultado del BTA como parámetros globales.

Definición 23 (anotación de programa) *Un programa se anota reemplazando cada regla $f(\overline{x_n}) = e$ en P por una nueva regla definida como*

[10]Existen varias técnicas para determinar automáticamente los pares de reducción (*lexicographic path order* (*LPO*), interpretaciones polinomiales, etc., véase, e.g., [Der87]).

$$ann^u(e) = \begin{cases} x & \text{si } e \equiv x \in \mathcal{X} \\ c(\overline{ann^u(e_n)}) & \text{si } e \equiv c(\overline{e_n}),\ c \in \mathcal{C} \\ (f)case\ x\ of\ \{\overline{p_k \to ann^u(e_k)}\} & \text{si } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\} \\ f^u(\overline{ann^u(e_n)}) & \text{si } e \equiv f(\overline{e_n}),\ f \in \mathcal{D}, \text{ y cada multigrafo idempotente asociado a } f/n \text{ contiene al menos un arco } i_f \xrightarrow{\succ} i_f \text{ para algún } i \in \{1,\ldots,n\} \text{ tal que } i_f \text{ es estático.} \\ f^m(\overline{ann^u(e_n)}) & \text{de otra forma, donde } e \equiv f(\overline{e_n}) \end{cases}$$

Figura 4.5: Función de anotación ann^u.

$f(\overline{x_n}) = ann^g(ann^u(e))$[11]. *La función* ann^u *se usa para agregar anotaciones de* unfolding *como se muestra en la Figura 4.5: una función* f *es anotada como* f^u *si ésta puede ser desplegada con toda seguridad, de otra forma se anota como* f^m *(donde* m *representa una anotación* memo*).*[12]

La función ann^g *se usa para agregar anotaciones de* generalización *(su uso será explicado en la siguiente sección). Su definición se muestra en la Figura 4.6, donde* $\mathcal{D}^{an}$ *indica el dominio de las funciones anotadas.*

Ejemplo 4.4 *Considérese el programa* `applast` *mostrado en la Figura 4.2. Consideremos su especialización con respecto al valor conocido del primer argumento de* `applast`*. En este caso, el BTA podría devolver la siguiente división:*

$$\{\texttt{applast} \mapsto (\texttt{S},\texttt{D}),\ \ \texttt{append} \mapsto (\texttt{S},\texttt{D}),\ \ \texttt{last} \mapsto (\texttt{D}),\ \ \texttt{last}' \mapsto (\texttt{D},\texttt{D})\}$$

donde `S` *indica que un argumento es estático (ground) y* `D` *que éste es*

[11] Usamos dos funciones independientes para mayor claridad. La implementación requiere un solo paso para agregar todas las anotaciones.

[12] Como se mencionó antes, sólo consideramos variables como argumentos de expresiones *case*.

$$ann^g(e) = \begin{cases} x & \text{si } e \equiv x \in \mathcal{X} \\ c(\overline{ann^g(e_n)}) & \text{si } e \equiv c(\overline{e_n}),\ c \in \mathcal{C} \\ (f)case\ x\ of\ \{\overline{p_k \to ann^g(e_k)}\} & \text{si } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\} \\ f^{an}(\overline{e'_n}) & \text{si } e \equiv f^{an}(\overline{e_n}),\ f^{an} \in \mathcal{D}^{an}, \text{ and} \\ & \quad e'_i = \begin{cases} ann^g(e_i) & \text{si cada multigrafo idempotente asociado a } f/n \text{ contiene un arco } i_f \xrightarrow{R} i_f,\ R \in \{\succsim, \succ\}, \text{ con } i_f \text{ estático para } i \in \{1 \ldots n\} \\ e'_i = gen(ann^g(e_i)) & \text{de otra forma} \end{cases} \end{cases}$$

Figura 4.6: Función de anotación ann^g.

dinámico. El procedimiento de anotación devuelve el siguiente programa:

```
applast(xs,x)=last^m(append^u(xs,[x]))
last(xs)      =fcase xs of
                 { (y:ys) → last'^m(ys,y) }
last'(ys,y)   =fcase ys of
                 { [] → [y];
                   (w:ws) → last^m(w:ws) }
append(xs,ys)=fcase xs of
                 { [] → ys;
                   (w:ws) → w:append^u(ws,ys) }
```

4.8. Aspectos de control

En esta sección, recordaremos primeramente el procedimiento genérico para evaluación parcial dirigido por *narrowing* y, enseguida, presentamos las nuevas estrategias para los niveles de control global y local. El procedimiento genérico se muestra en la Figura 4.7. De forma similar al procedimiento de evaluación parcial de Gallagher para programas lógicos [Gal93], nuestro algoritmo distingue claramente dos niveles diferentes:

Nivel local. Dado un conjunto de términos encabezados por operación (esto es, llamadas a función), el nivel de control local aplica un operador

Input: un programa P y un conjunto T de llamadas a función
Output: un conjunto de llamadas S
Initialization: $i := 0;\ T_0 := T$
Repeat
$P' := \mathit{unfold}(T_i, P)$;
$T_{i+1} := \mathit{abstract}(T_i, P'_{\mathit{calls}})$;
$i := i + 1$;
Until $T_i = T_{i-1}$ (módulo renombramiento de variables)
Return: $S := T_i$

Figura 4.7: Procedimiento genérico para NPE.

de desplegado *unfold* de tal forma que devuelve un conjunto de reglas residuales como resultado (ver Sección 4.8.2). El operador de desplegado debe asegurar que el proceso de desplegado sea finito, i.e., que los cómputos parciales no se ejecuten infinitamente.

Nivel global. Este nivel debe asegurar que el número de funciones especializadas, además de ser diferentes, debe mantenerse finito. Para este propósito, se usa un operador de abstracción *abstract*. El operador de abstracción toma un conjunto finito de términos encabezados por operación T_i y después agrega apropiadamente el conjunto de subtérminos encabezados por operación, ya desplegados, en las partes derechas de las llamadas. Tal conjunto está denotado por P'_{calls}. Quizá sea necesario evaluar adicionalmente el nuevo conjunto T_{i+1}. De esta forma, el proceso se repite iterativamente mientras sean introducidos nuevos términos.

Obsérvese que este procedimiento no retorna un programa evaluado parcialmente pero sí un conjunto finito de términos encabezados por operación. El programa residual, sin embargo, puede ser fácilmente construido aplicando el operador de desplegado al conjunto de términos retornado y, después, renombrar la reglas usando una fase estándar de post-desplegado (ver Sección 4.8.2)

4.8.1. Control global

Nuestro operador de abstracción esta basado en la siguiente propiedad: consideremos una computación (posiblemente infinita) en un programa

anotado de acuerdo a la Definición 23 y sea $t_1, t_2, t_3, \ldots$ cualquier secuencia de términos encabezados por operación dentro de esta computación. Sea $abs(t)$ una función que reemplace cada subtérmino anotado $gen(t')$ en t (si hay alguno) por una variable fresca. Entonces, la secuencia $abs(t_1), abs(t_2), abs(t_3), \ldots$ es cuasi-terminate.

Por lo tanto, nuestro operador de abstracción esta basado en reemplazar subtérminos anotados por variables frescas. En adelante, denotamos por $t \in \overline{T}$ el hecho que hay un término $t' \in T$ tal que t y t' son iguales módulo renombramiento de variables.

Definición 24 (operador de abstracción) *Sean T_1 y T_2 conjuntos finitos de términos. Entonces, $abstract(T_1, T_2)$ esta definido como sigue:*

$$abstract(T_1, T_2) = \begin{cases} T_1 & \texttt{if } T_2 = \{\,\} \\ abstract(T_1, T_2') & \texttt{if } T_2 = \{t\} \cup T_2' \\ & \texttt{y } \; gen(t) \in \overline{T}_1 \\ abstract(T_1 \cup \{t'\}, T_2') & \texttt{if } T_2 = \{t\} \cup T_2' \\ & \texttt{y } \; t' = gen(t) \notin \overline{T}_1 \end{cases}$$

4.8.2. Control local

Ahora, introducimos nuestro operador de desplegado. Éste está dirigido por las anotaciones de desplegado, así que las funciones de la forma f^u deben ser desplegadas mientras que las funciones f^m no. Claramente, cada cómputo en el que sólo se despliegan funciones f^u debe ser finito.

Los cómputos son ejecutados con una ligera extensión del cálculo RLNT [AHV03] como se muestra en la Figura 4.8. Primero, nótese que los símbolos "$[\![$" y "$]\!]$" en una expresión como $[\![e]\!]$ son puramente sintácticos (es decir, éstos no denotan "el valor de e"). De hecho, éstos sólo son usados para señalar las subexpresiones donde las reglas de inferencia pueden ser aplicadas. Expliquemos brevemente las reglas del cálculo:

Las primeras tres reglas tratan con llamadas a función. Si la función está anotada con u, entonces la regla Unfold ejecuta una operación de desplegado. Si está anotada con m, la regla Memo suspende la evaluación de la llamada. Finalmente, la regla Gen se usa simplemente para identificar y no desplegar la expresión encabezada por dicha anotación en éste nivel.

Unfold

$$[\![f^u(\overline{e_n})]\!] \Rightarrow [\![\sigma(e')]\!] \quad \text{si} \quad f(\overline{x_n}) = e' \in P \quad \text{y} \quad \sigma = \{\overline{x_n \mapsto e_n}\}$$

Memo

$$[\![f^m(\overline{e_n})]\!] \Rightarrow f(\overline{e_n})$$

Gen

$$[\![gen(e)]\!] \Rightarrow gen([\![e]\!])$$

Select

$$[\![(f)case\ c(\overline{e_n})\ of\ \{\overline{p_k \to e'_k}\}]\!] \Rightarrow [\![\sigma(e'_i)]\!] \ \text{si} \ p_i = c(\overline{x_n}) \ \text{y} \ \sigma = \{\overline{x_n \mapsto e_n}\}, \ i \in \{1, \ldots, k\}$$

Guess

$$[\![(f)case\ x\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow (f)case\ x\ of\ \{\overline{p_k \to [\![\sigma_k(e_k)]\!]}\} \ \text{si} \ \sigma_i = \{x \mapsto p_i\},\ i = 1, \ldots, k$$

Eval

$$[\![(f)case\ e\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow [\![(f)case\ e'\ of\ \{\overline{p_k \to e_k}\}]\!] \ \text{si} \ [\![e]\!] \Rightarrow [\![e']\!],\ e \notin \mathcal{X},\ root(e) \notin \mathcal{C}\ ,\ \text{y}\ e \neq (f)case\ x\ of\ \{\ldots\}$$

Case-of-Case

$$[\![(f)case\ ((f)case\ x\ of\ \{\overline{p_k \to e_k}\})\ of\ \{\overline{p'_j \to e'_j}\}]\!] \Rightarrow [\![(f)case\ x\ of\ \{p_k \to (f)case\ e_k\ of\ \{\overline{p'_j \to e'_j}\}\}]\!]$$

Figura 4.8: El cálculo RLNT *offline*.

Pero las expresiones anotadas por Gen indican al control local que dicha expresión debe ser generalizada en el control global. Observemos que la expresión evaluada nunca contiene anotaciones u ni m, ya que éstas no son necesarias en el nivel global.

Las ultimas cuatro reglas tratan con expresiones *case*. La regla Select es utilizada para seleccionar la bifurcación de emparejamiento de una expresión *case* cuando su argumento es un término encabezado por constructor. La regla Guess se aplica cuando el argumento es una variable libre. Aquí, residualizamos la estructura *case* y continuamos con la evaluación de las diferentes bifurcaciones (aplicando la correspondiente sustitución para propagar los valores implicados en el cómputo). La regla Eval es usada para evaluar expresiones *case* con un llamado a función o bien con otra expresión *case* dada en la posición del argumento. Aquí, $root(e)$ indica el símbolo más externo de e. Finalmente, la regla Case-of-Case mueve el *case* externo dentro de las bifurcaciones del más interno y, de este modo, la evaluación de las ramificaciones puede proseguir (reglas similares pueden ser encontradas en el Compilador de Glasgow Haskell

así como en la deforestación de Wadler [Wad90]).

Obsérvese que los cómputos RLNT con un programa anotado son siempre finitos como una simple consecuencia del Teorema 20 y las consideraciones realizadas para los programas *flat*.

Nuestro operador de desplegado puede ser definido como sigue:

Definición 25 (operador de desplegado) *Dado un programa flat P y un conjunto de términos T, tenemos que $unfold(T, P) =$*

$$\{\ f(\overline{e_n}) = [\![\sigma(e)]\!] \mid f(\overline{e_n}) \in T,\ f(\overline{x_n}) = e \in P,\ \mathtt{y}\ \ \sigma = \{\overline{x_n \mapsto e_n}\}\ \}$$

Nótese que el operador de desplegado no devuelve un programa *flat* válido. Esto no es relevante durante el proceso de especialización. Una vez que el proceso iterativo termina, se puede agregar un post-proceso de renombramiento estándar que reemplace cada parte izquierda de la forma $f(\overline{e_n})$ por $f(\overline{x_m})$ donde $\overline{x_m}$ son las variables diferentes de $\overline{e_n}$ en el mismo orden en el que ocurren y, entonces, se renombran las correspondientes expresiones en las partes derechas.

Ejemplo 4.5 *Consideremos nuevamente el programa anotado del Ejemplo 4.4. Dado el conjunto inicial de llamadas $T_0 = \{\mathtt{applast}([1], \mathtt{x})\}$ la evaluación parcial produce la siguiente secuencia de llamadas (de acuerdo al algoritmo de la Figura 4.7):*

$$\begin{aligned}
T_1 &= T_0 \cup \{\mathtt{last}(\mathtt{append}([1], [\mathtt{x}]))\} \\
T_2 &= T_1 \cup \{\mathtt{last}'(\mathtt{append}([\,], [\mathtt{x}]), 1)\} \\
T_3 &= T_2 \cup \{\mathtt{last}([\mathtt{x}])\} \\
T_4 &= T_3 \cup \{\mathtt{last}'([\,], \mathtt{x})\}
\end{aligned}$$

el algoritmo se detiene ya que T_5 sería igual a T_4 módulo renombramiento de variables.

Usando nuestra implementación de evaluación parcial, obtenemos los siguientes resultados:

$$\begin{aligned}
\mathtt{applast}([1], \mathtt{x}) &= \mathtt{last}(\mathtt{append}([1], [\mathtt{x}])) \\
\mathtt{last}(\mathtt{append}([1], [\mathtt{x}])) &= \mathtt{last}'(\mathtt{append}([\,], [\mathtt{x}]), 1) \\
\mathtt{last}'(\mathtt{append}([\,], [\mathtt{x}]), 1) &= \mathtt{last}([\mathtt{x}]) \\
\mathtt{last}([\mathtt{x}]) &= \mathtt{last}'([\,], \mathtt{x}) \\
\mathtt{last}'([\,], \mathtt{x}) &= [\mathtt{x}]
\end{aligned}$$

Unfold

$[\![f(\overline{e_n})]\!]^T \;\Rightarrow\; [\![\sigma(e')]\!]^{T\cup\{gen(f(\overline{e_n}))\}}$ si $gen(f(\overline{e_n})) \notin \overline{T}$, $f(\overline{x_n}) = e' \in P$ y $\sigma = \{\overline{x_n \mapsto e_n}\}$

Memo

$[\![f(\overline{e_n})]\!]^T \;\Rightarrow\; f(\overline{[\![e_n]\!]^T})$ si $gen(f(\overline{e_n})) \in \overline{T}$

Figura 4.9: El cálculo RLNT híbrido.

mismos que, después de un proceso simple de renombramiento y simplificación, el resultado final es únicamente la siguiente regla (la especialización óptima):

$$\texttt{applast}_1(\texttt{x}) \;=\; [\texttt{x}]$$

4.8.3. Refinamiento del control local

Finalmente, presentamos un sencillo refinamiento del operador de desplegado presentado en la sección anterior.

La idea básica es la siguiente: consideramos que el proceso de anotación no incluye las anotaciones u y m. Así pues, el control local aplica una prueba de terminación similar a la aplicada en el control global. Con este propósito, se modifica el cálculo RLNT *offline* como sigue:

Durante la evaluación, tenemos expresiones de la forma $[\![e]\!]^T$ donde T registra las llamadas ya evaluadas, siendo la expresión inicial de la forma $[\![e]\!]^{\{\}}$.

Las primeras dos reglas son redefinidas como se muestra en la Figura 4.9. Básicamente, desplegamos aquellas funciones que, después de reemplazar subtérminos por variables frescas, son iguales módulo renombramiento de variables a alguna llamada previamente desplegada. En este caso, la llamada generalizada se agrega al conjunto actual de llamadas memorizadas. De otra forma, la llamada no es desplegada y procedemos a evaluar sus argumentos.

La reglas restantes solo propagan el conjunto actual de llamadas memorizadas.

La terminación de la nueva estrategia local es, no obstante, una simple consecuencia del Teorema 20. La principal diferencia con la estrategia del

benchmark	original runtime	Híbrido			Offline			Online		
		spec. time	runtime spec.	speedup	spec. time	runtime spec.	speedup	spec. time	runtime spec.	speedup
ackermann	1953	860	533	3,66	730	543	3,60	290	342	5,71
allones	1477	170	1418	1,04	170	1442	1,02	170	1428	1,03
applast	1145	190	1121	1,02	220	1112	1,03	310	1133	1,01
dapp	338	330	377	0,90	260	360	0,94	220	332	1,02
flip	806	220	796	1,01	270	796	1,01	1870	790	1,02
gauss	235	640	237	0,99	700	239	0,98	10480	228	1,03
interSB	161	1610	164	0,98	1780	170	0,95	4720	168	0,96
kmp3B*A	97	18650	78	1,24	19330	52	1,87	24510	8	12,13
lengthapp	2769	590	2876	0,96	550	2637	1,05	1020	2581	1,07
power	25	640	27	0,93	800	30	0,83	8940	63	0,40
Average	**901**	**2390**	**763**	**1,27**	**2481**	**738**	**1,33**	**5253**	**707**	**2,54**

Cuadro 4.2: Resultados de los programas de prueba.

control local anterior consiste en que, ahora, no es una estrategia *offline* pura, ya que se ejecutan algunas pruebas (*online* simples) en el nivel local, por esto lo llamamos *híbrido*.

El cuadro 4.2 muestra los resultados de una evaluación experimental de ambas estrategias. En general, logran mejoras similares y son igualmente eficientes.

Ejemplo 4.6 *Consideremos nuevamente el ejemplo 4.4 pero tomando en cuenta el refinamiento del control local. Entonces, tenemos el siguiente computo.*

$$\begin{aligned}\texttt{applast}([\texttt{1}],\texttt{x})^{\{\}} &\\ \Rightarrow &\texttt{last}(\texttt{app}([\texttt{1}],[\texttt{x}]))^{\{\}\,\cup\,\{\texttt{applast}([\texttt{1}],\texttt{x})\}}\\ \Rightarrow \ldots \Rightarrow\ & [\texttt{x}]\end{aligned}$$

así pues el resultado renombrado es:

$$\texttt{applast}_1(\texttt{x}) \;=\; [\texttt{x}]$$

es decir que, la llamada fue completamente desplegada.

4.9. Implementación

Hemos integrado en una aplicación de Curry dos módulos de anotación de programas debido a que se tienen propiamente dos procedimientos de anotación, uno para la evaluación parcial *offline pura*, mencionada

en la sección 4.8.2 y otro para la especialización *híbrida* expuesta en la sección 4.8.3. Asimismo hemos implementado las dos aproximaciones de especialización asociadas.

Los dos módulos de anotación usan el resultado de un mismo análisis BTA para saber qué argumentos son estáticos y cuáles dinámicos. A continuación, se entregan como parámetros el resultado del BTA y el programa a anotar al respectivo módulo de anotación, obteniendo como resultado en ambos casos el programa Curry anotado. Los criterios utilizados para anotar los términos son los siguientes:

Procedimiento de anotación para la estrategia *híbrida*:

Para cada función con un multigrafo idempotente asociado que no cumpla con las condiciones del Teorema 20:

- anotar con GEN los parámetros estáticos que no tengan asociado un arco $\succ$.
- anotar con GEN los parámetros dinámicos.

Procedimiento de anotación para la estrategia *offline pura*:

Para cada función f de aridad n con un multigrafo idempotente asociado que contenga al menos un arco $i_f \xrightarrow{\succ} i_f$ para algún $i \in \{1, \ldots, n\}$ tal que i_f sea estático:

- la función debe ser anotada con UNF

De lo contrario,

- la función debe ser anotada con MEM

Adicionalmente,

- se debe considerar el procedimiento de anotación de la estrategia híbrida

Una vez que se tiene el programa anotado, al invocar el respectivo comando para especializar, las acciones se pueden resumir de acuerdo a la tabla 4.3. En el caso de la aproximación *offline pura* [ARSV07], el control LOCAL está dirigido por las anotaciones UNF y MEM donde únicamente son desplegadas las funciones con anotación UNF cuya terminación está garantizada. Para el caso de las funciones con anotación

Cuadro 4.3: Acciones de la especialización.

Control	***Offline* puro**	**Híbrido**
LOCAL	Obedece las anotaciones UNF/MEM	test de terminación
GLOBAL	Generaliza los subtérminos anotados con GEN	

MEM, el control LOCAL suspende el desplegado porque no podemos asegurar su terminación. Para el caso de especialización *híbrida*, a este mismo nivel, ya que éste no incluye anotaciones UNF o MEM, se aplica una prueba de terminación muy similar a la del nivel global, es decir, un test de igualdad módulo renombramiento de variables para no desplegar indefinidamente. Debido a la aplicación de esta prueba en el nivel de control local le hemos llamado aproximación híbrida. En el caso del control GLOBAL, tanto para la aproximación *offline pura* como para la *híbrida*, se generalizan los sub-términos anotados con GEN, es decir, se reemplaza cada término anotado con GEN por una variable fresca.

Para comparar las estrategias *offline* con el evaluador parcial *online*, hemos hecho una ligera modificación al evaluador parcial *online* de [AHV02] y se adaptaron los ejemplos para ambas clases de especializadores. En el cuadro 4.2 se analizan tiempos de especialización, tiempos de ejecución de los programas especializados, y la relación de mejora con respecto al tiempo de ejecución del programa original. Consideramos las aproximaciones *híbrida*, *offline pura* y *online* [AHV02]. Puede verse cómo la mejora del especializador *online* es superior a las dos especializaciones *offline*. Asimismo, el tiempo de especialización tarda, en promedio, más del doble con la especialización *online* que con las dos aproximaciones *offline*. Comparando ahora *híbrida* y *offline pura*, esperábamos una mejora significativa en la *offline pura*; sin embargo, en promedio, no hay grandes diferencias, es decir, que se han desempeñado de forma similar como ya se había comentado. Los evaluadores parciales *híbrido* y *offline puro* así como el *online* están disponibles públicamente en: `http://www.dsic.upv.es/~garroyo/bench.htm`

4.10. Conclusiones

En este capítulo hemos introducido un nuevo procedimiento de anotación para la evaluación parcial de programas lógico funcionales. Este procedimiento combina la información recolectada de un análisis de tiempo de enlace monovariante y un análisis de grafos *size-change* [LJBA01]. En contraste a previas aproximaciones tales como [GJ05], algunas extensiones han sido necesarias para afrontar la componente lógica del lenguaje lógico funcional considerado. (e.g., las condiciones de particionamiento finito no fueron necesarias en [GJ05]). Los experimentos de la ejecución de un evaluador parcial que incluye el nuevo procedimiento de anotación muestran la mejora en el desempeño con respecto al evaluador parcial anterior de [RSV05b].

Para mejorar aún más la precisión del evaluador parcial, hemos implementando una versión *polivariante* de la fase de anotación, véase el capítulo 5. En este caso, cada llamada a función es procesada de manera separada de acuerdo a la información comprendida en su multigrafo asociado. El algoritmo resultante es más costoso pero también es más preciso.

También hemos introducido las estrategias de anotación y de control apropiadas para diseñar un evaluador parcial *offline* dirigido por *narrowing*. Además se realizaron las implementaciones de evaluación parcial siguiendo las ideas presentadas hasta el momento, siendo lo suficientemente alentadores los resultados alcanzados.

Capítulo 5

Transformación polivariante de funciones de orden superior

En este capítulo presentamos una aproximación transformacional a un BTA polivariante de orden superior para programas funcionales. Técnica que consiste, de manera intuitiva, en una desfuncionalización del programa de orden superior. Posteriormente se aplica una segunda transformación, al programa desfuncionalizado, generando copias de sus correspondientes definiciones de función por cada llamada que tenga diferentes valores de tiempo de enlace. Con lo anterior se consigue aliviar algunas limitaciones del análisis de tiempo de enlace mostrado en el capítulo 4. Gran parte de este capítulo ha sido publicado en [ARTV09].

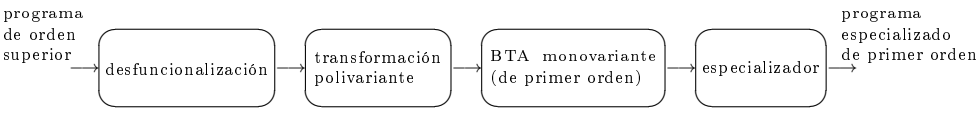
5.1. Introducción

El análisis *size-change* de [ARSV07] y su BTA asociado sufren de algunas limitaciones. Primeramente, el análisis *size-change* está únicamente definido para programas funcionales de *primer orden*, lo cual limita su aplicabilidad. En segundo lugar, el análisis de tiempo de enlace es *monovariante*, es decir, se asocia una sola secuencia de datos de tiempo de enlace[1] a los argumentos de una función dada y, todas las llamadas a

[1] Consideramos los valores convencionales de los datos arrojados por el *análisis de tiempo de enlace* como: *estático* (absolutamente conocido en tiempo de evaluación parcial) y *dinámico* (probablemente conocido en tiempo de evaluación parcial).

la misma función son procesadas de la misma forma, lo cual implica una considerable pérdida de precisión.

En este capítulo, presentamos una aproximación transformacional para superar las desventajas mencionadas anteriormente. Básicamente, primero transformamos por *desfuncionalización* [Rey98] el programa original de *orden superior*. En concreto, introducimos una extensión de técnicas de desfuncionalización (similares a [AT99, GM93]) especialmente diseñadas para mejorar la precisión del análisis *size-change*. Después con la finalidad de mejorar la precisión tanto del análisis *size-change* como del BTA asociado, implementamos una transformación de código fuente-a-fuente haciendo explícito en el programa, cada valor que arroja el análisis de tiempo de enlace para cada argumento de una función. De esta forma, cada llamada a función que posea diferentes valores de tiempo de enlace, podrá ser tratada de un modo distinto. Gracias a esta transformación, podemos obtener la misma precisión usando un BTA monovariante sobre el programa transformado, que utilizando un BTA polivariante (donde de manera natural, sin transformaciones, pueden ser asociados a una función dada, diferentes valores de tiempo de enlace) sobre el programa original.



5.2. Desfuncionalización

En esta sección, introducimos paso a paso una transformación que toma un programa de orden superior y devuelve uno de primer orden (un sistema de reescritura de términos). En contraste a un algoritmo de desfuncionalización estándar, nosotros realizamos una forma restringida de instanciación de variables y de desplegado (ver paso dos) con el fin de hacer explícita una mayor información de orden superior. Aunque puede incrementarse el tamaño del código, los subsiguientes pasos del BTA — análisis *size-change* y propagación de valores del análisis de tiempo de enlace— lo pueden aprovechar para producir resultados más precisos; con la esperanza de que el incremento de código será eliminado durante la fase de especialización (véase los tiempos de mejora de ejecución en la sección 5.4).

Presentamos paso a paso nuestra técnica de desfuncionalización optimizada:

1. El primer paso hace explícitas tanto aplicaciones como llamadas parciales a función.

2. Luego, en el segundo paso se crean instancias de variables funcionales con todas las posibles llamadas parciales a función.

3. Finalmente, si después de reducir las aplicaciones[2], tanto como sea posible, el programa aún contiene algunas aplicaciones, el tercer paso agrega una definición apropiada para la aplicación de la función.

5.2.1. Haciendo explícitas las llamadas parciales y aplicaciones

Primero, hacemos explícita cada aplicación del programa de orden superior utilizando la función apply. También, distinguimos aplicaciones parciales a partir de funciones totales. En particular, las aplicaciones parciales se representan por medio del símbolo constructor partcall. Las llamadas totales se indican en la forma usual, e.g., $f(\overline{t_n})$ para algún símbolo de función definido f/n. Una llamada parcial se denota por $\mathsf{partcall}(f, k, \overline{t_m})$ donde f/n es un símbolo de función definida, $0 < k \leq n$, y $m + k = n$, es decir, $\overline{t_m}$ son los primeros m argumentos de f/n pero aún faltan k argumentos (si $k = n$, entonces la aplicación parcial no tiene argumentos, dicho de otro modo, tenemos $\mathsf{partcall}(f, n)$)[3]. Para mayor claridad, en lo que sigue consideramos que partcall es una función "variádica"[4]; sin embargo, se puede formalizar usando una función con tres argumentos tal que el tercer argumento es una lista (posiblemente vacía) con argumentos ya aplicados.

[2]Básicamente, cada expresión de la forma apply(partcall(...),...) puede ser reducida, donde apply es la aplicación de la función y partcall denota una llamada parcial a función.

[3]Una representación similar se usa en *FlatCurry*, la representación intermedia del lenguaje de programación lógico funcional Curry [Han06].

[4]Esto es, una función de aridad variable.

Una vez que todas las aplicaciones se han hecho explícitas con apply y partcall, aplicamos la siguiente transformación, tanto como sea posible, a las partes derechas de las funciones del programa:

$$\mathsf{apply}(\mathsf{partcall}(f, k, \overline{t_m}), t_{m+1}) = \begin{cases} f(\overline{t_{m+1}}) & \text{si } k = 1 \\ \mathsf{partcall}(f, k-1, \overline{t_{m+1}}) & \text{si } k > 1 \end{cases} \quad (*)$$

Esto es útil para evitar aplicaciones innecesarias en el programa desfuncionalizado cuando hay suficiente información disponible para reducirlas de manera estática.

A continuación, asumimos que cada programa contiene una función, llamada main, la cual está definida por una sola regla de la forma $(\mathsf{main}\ x_1\ \ldots\ x_n\ =\ r)$, con $x_1, \ldots, x_n \in \mathcal{V}$ variables diferentes, y que el programa no contiene llamadas a esta función particular. Adicionalmente, consideramos que la llamada a main tiene únicamente términos constructores (esto es, valores) como argumentos. Esto se requiere para evitar la introducción de expresiones de orden superior las cuales no deben existir en el programa en tiempo de ejecución.

Ejemplo 5.1 *Considérese el siguiente programa de orden superior $\mathcal{R}_1$ (tal como se usa en la práctica, utilizamos la notación currificada para programas de orden superior):*

$$\begin{array}{lcllcl} \mathsf{main}\ xs & = & map\ inc\ xs & map\ f\ [\,] & = & [\,] \\ inc\ x & = & Succ\ x & map\ f\ (x : xs) & = & f\ x : map\ f\ xs \end{array}$$

donde los números naturales son construidos a partir de Z y $Succ$, las listas a partir de $[\,]$ y ":". Primero, hacemos explícitas todas las aplicaciones y llamadas parciales:

$$\begin{array}{lcl} \mathsf{main}(xs) & = & \mathsf{apply}(\mathsf{apply}(\mathsf{partcall}(map, 2), \mathsf{partcall}(inc, 1)), xs) \\ map(f, [\,]) & = & [\,] \\ map(f, x : xs) & = & \mathsf{apply}(f, x) : \mathsf{apply}(\mathsf{apply}(\mathsf{partcall}(map, 2), f), xs) \\ inc(x) & = & Succ(x) \end{array}$$

Después, reducimos todas las llamadas a apply *que contengan una llamada parcial como primer argumento utilizando la transformación* $(*)$

indicada anteriormente a fin de obtener el programa transformado $\mathcal{R}_2$*:*

$$\begin{array}{lcl} \mathsf{main}(xs) & = & map(\mathsf{partcall}(inc, 1), xs) \\ map(f, [\,]) & = & [\,] \\ map(f, x : xs) & = & \mathsf{apply}(f, x) : map(f, xs) \\ inc(x) & = & Succ(x) \end{array}$$

5.2.2. Instanciación de variables funcionales

En el siguiente paso, sustituimos las variables funcionales del programa por instancias con el fin de que algunas aplicaciones puedan ser reducidas. Esta es la principal diferencia con respecto a algoritmos previos de desfuncionalización. En lo subsiguiente, decimos que una variable es una variable *funcional* si ésta puede ser enlazada (en tiempo de ejecución) a una llamada parcial. Ahora, ya podemos reemplazar cada variable funcional por todas las aplicaciones parciales posibles.

Sea $\textsc{pcalls}_{\mathcal{R}}$ el conjunto de símbolos de función que aparecen en los partcalls de $\mathcal{R}$, esto es,

$$\textsc{pcalls}_{\mathcal{R}} = \{f/n \mid \mathsf{partcall}(f, k, t_1, \ldots, t_m) \text{ dadas en } \mathcal{R}, \text{ con } k + m = n\}$$

Después, para cada función $f/n \in \textsc{pcalls}_{\mathcal{R}}$ con el tipo[5]

$$\tau_1 \mapsto \ldots \mapsto \tau_n \mapsto \ldots \mapsto \tau_k \mapsto \tau_{k+1} \qquad (n \leq k)$$

reemplazamos cada regla $l[x]_p = r$ donde x es una variable funcional de tipo

$$\tau'_j \mapsto \ldots \mapsto \tau'_k \mapsto \tau'_{k+1}$$

y $1 \leq j \leq n$ (tal que aún faltan algunos argumentos), por las instancias

$$\sigma(l[x]_p = r) \qquad \text{donde } \sigma = \{x \mapsto \mathsf{partcall}(f, n - j + 1, x_1, \ldots, x_{j-1})\}$$

con $x_1, \ldots, x_{j-1}$ variables diferentes (si $j = 1$, no se agregan argumentos a la llamada parcial). Claramente nos referimos a los tipos inferidos en el programa de orden superior original. No obstante, si no hay información de tipos disponible simplemente instanciamos las reglas[6] con todas sus

[5] Obsérvese que $n < k$ implica que la función f retorna un valor funcional.

[6] Vamos a llamar *instanciación de reglas* a la *sustitución de reglas por instancias.*

llamadas parciales posibles. Esto quizá podría introducir algunas reglas inútiles pero serían seguras. Por ejemplo, consideremos la regla

$$f(x, xs) = map(x, xs)$$

donde x es una variable funcional de tipo $\mathbb{N} \mapsto \mathbb{N}$. Dada la función $sum/2 \in \text{PCALLS}_{\mathcal{R}}$ con tipo $\mathbb{N} \mapsto \mathbb{N} \mapsto \mathbb{N}$, reemplazamos la regla anterior por la siguiente instancia:

$$f(\mathsf{partcall}(sum, 1, n), xs) = map(\mathsf{partcall}(sum, 1, n), xs)$$

La instanciación de reglas se aplica repetidamente hasta que no aparezcan más reglas con una variable funcional en el programa.[7] Después, tal como en el paso anterior, aplicamos la transformación $(*)$ citada; tanto como sea posible a las partes derechas del programa. El siguiente ejemplo ilustra este proceso de instanciación.

Ejemplo 5.2 *Consideremos nuevamente el programa transformado $\mathcal{R}_2$ del Ejemplo 5.1. Tenemos* $\text{PCALLS}_{\mathcal{R}_2} = \{inc/1\}$. *Hay sólo una variable funcional f (con tipo $\tau_1 \mapsto \tau_2$) en las reglas que definen a map, por lo tanto después de la instanciación obtenemos las siguientes reglas:*

$$\begin{array}{lcl} map(\mathsf{partcall}(inc, 1), [\,]) & = & [\,] \\ map(\mathsf{partcall}(inc, 1), x : xs) & = & \mathsf{apply}(\mathsf{partcall}(inc, 1), x) : \\ & & \quad map(\mathsf{partcall}(inc, 1), xs) \end{array}$$

Ahora, reduciendo todas las llamadas a apply *con un* partcall *como primer argumento, obtenemos el programa transformado $\mathcal{R}_3$:*

$$\begin{array}{lcl} \mathsf{main}(xs) & = & map(\mathsf{partcall}(inc, 1), xs) \\ map(\mathsf{partcall}(inc, 1), [\,]) & = & [\,] \\ map(\mathsf{partcall}(inc, 1), x : xs) & = & inc(x) : map(\mathsf{partcall}(inc, 1), xs) \\ inc(x) & = & Succ(x) \end{array}$$

Nótese que $map(\mathsf{partcall}(inc, 1), \ldots)$ debe concebirse como una función fresca, e.g., podríamos reescribir el programa como sigue:

$$\begin{array}{lcllcl} \mathsf{main}(xs) & = & map_{inc}(xs) & map_{inc}([\,]) & = & [\,] \\ inc(x) & = & Succ(x) & map_{inc}(x : xs) & = & inc(x) : map_{inc}(xs) \end{array}$$

[7] Nótese que una función puede tener varios argumentos funcionales y, de este modo, podríamos aplicar el proceso de instanciación a las propias instancias de una regla previamente considerada.

Obsérvese que no ocurren llamadas a apply *en el programa final y, de esta forma, no hay necesidad de agregar una definición para* apply *(es decir, que podría no ser necesario aplicar el siguiente paso para este ejemplo). Un tema interesante para trabajo futuro es determinar la clase de programas para la cual podemos garantizar la no ocurrencia de* apply*s en el programa transformado.*

Nótese que este paso de la transformación es correcto ya que main puede ser llamado únicamente con términos constructores como argumentos. Por otra parte, las variables funcionales deberían ser instanciadas con todas las aplicaciones parciales posibles y no sólo por aquellas en $\text{PCALLS}_{\mathcal{R}}$. Pero por otro lado, no todas las instancias de variables funcionales serán asequibles desde main. El uso de un *análisis de cerradura*[8] puede mejorar la precisión del programa transformado (pero esto agregará también un gasto significativo de tiempo en el proceso de desfuncionalización).

5.2.3. Incorporando una definición explícita de apply

En contraste con algunas técnicas de desfuncionalización estándar (tales como [AT99, GM93]), el proceso de transformación, hasta ahora, puede producir un programa de primer orden sin ocurrencias de la función apply en muchos casos (como en el ejemplo anterior).

En otros casos, sin embargo, algunas ocurrencias de apply permanecen en el programa transformado, y entonces debe ser añadida una definición apropiada de apply. Este es el caso, e.g., cuando hay una llamada a apply con una llamada a función como primer argumento. En este caso, el valor de la llamada parcial no será conocido hasta el momento de la ejecución, de este modo, incorporamos la siguiente secuencia de reglas:

$$\begin{aligned}
&\mathsf{apply}(\mathsf{partcall}(f, n), x_1) = \mathsf{partcall}(f, n-1, x_1)\\
&\mathsf{apply}(\mathsf{partcall}(f, n-1, x_1), x_2) = \mathsf{partcall}(f, n-2, x_1, x_2)\\
&\ldots\\
&\mathsf{apply}(\mathsf{partcall}(f, 1, x_1, \ldots, x_{n-1}), x_n) = f(x_1, \ldots, x_n)
\end{aligned}$$

para cada símbolo de función $f/n \in \text{PCALLS}_{\mathcal{R}}$.

[8] Del término en inglés: *closure analysis*.

Nuestro proceso de desfuncionalización puede ser aplicado efectivamente no sólo a programas que usan construcciones simples tales como $(map\ f\ \ldots)$ sino también a programas que hacen uso esencial de características de orden superior, como se ilustra en el siguiente ejemplo.

Ejemplo 5.3 *Considérese el siguiente programa de orden superior tomado de [Ser07]:*

$$\begin{array}{ll} \mathsf{main}\ x\ y = f\ x\ y & \\ g\ r\ a = r\ (r\ a) & f\ Z = inc \\ inc\ n = Succ\ n & f\ (Succ\ n) = g\ (f\ n) \end{array}$$

donde los números naturales se construyen a partir de Z y $Succ$. El primer paso del proceso de desfuncionalización devuelve:

$$\begin{array}{ll} \mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & \\ g(r,a) = \mathsf{apply}(r,\mathsf{apply}(r,a)) & f(Z) = \mathsf{partcall}(inc,1) \\ inc(n) = Succ(n) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n)) \end{array}$$

Aquí, $\textsc{pcalls}_{\mathcal{R}} = \{inc/1, g/2\}$. *Tenemos sólo una variable funcional r en la regla que define la función g (con tipo asociado $\mathbb{N} \mapsto \mathbb{N}$), consecuentemente, se agregan las siguientes instancias de las reglas que definen la función g:*

$$\begin{array}{l} g(\mathsf{partcall}(inc,1),a) = \mathsf{apply}(\mathsf{partcall}(inc,1),\mathsf{apply}(\mathsf{partcall}(inc,1),a)) \\ g(\mathsf{partcall}(g,1,x),a) = \mathsf{apply}(\mathsf{partcall}(g,1,x),\mathsf{apply}(\mathsf{partcall}(g,1,x),a)) \end{array}$$

Reduciendo todas las llamadas a apply *con un* partcall *como primer argumento, obtenemos:*

$$\begin{array}{ll} \mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & f(Z) = \mathsf{partcall}(inc,1) \\ g(\mathsf{partcall}(inc,1),a) = inc(inc(a)) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n)) \\ g(\mathsf{partcall}(g,1,x),a) = g(x,g(x,a)) & inc(n) = Succ(n) \end{array}$$

Finalmente, ya que aún permanece una ocurrencia de la función apply *en el programa, incorporamos las siguientes reglas:*

$$\begin{array}{l} \mathsf{apply}(\mathsf{partcall}(inc,1),x) = inc(x) \\ \mathsf{apply}(\mathsf{partcall}(g,2),x) = \mathsf{partcall}(g,1,x) \\ \mathsf{apply}(\mathsf{partcall}(g,1,x),y) = g(x,y) \end{array}$$

La validez formal de nuestra transformación de desfuncionalización puede verse como una extensión de aquella mostrada en [AT99, GM93] considerando que la función apply es estricta en su primer argumento, de esta forma, nuestra principal extensión, la instanciación de variables funcionales, es segura.

Tomado en cuenta que la evaluación de llamadas de orden superior que comprenden variables libres como funciones; no se permite en las implementaciones actuales de *narrowing* (tales llamadas son *suspendidas* para evitar el uso de unificación de orden superior [HP99]). Entonces nuestra aproximación también es segura en tiempo de evaluación parcial donde la información faltante (en forma de variables lógicas) podría surgir durante el proceso.

Respecto al incremento de tamaño del código debido a nuestro algoritmo de desfuncionalización, el hecho de hacer lo más explícita la información de orden superior origina un costo: en el peor caso, el programa fuente puede crecer exponencialmente en el número de funciones (e.g., cuando el programa contiene llamadas parciales a todas las funciones definidas). Sin embargo, este caso sucederá raramente y de esta manera el incremento del tamaño de código es generalmente razonable. Además, la fase de especialización subsecuente es usualmente capaz de reducir el código (véase la Secc. 5.4).

5.3. Transformación polivariante

En esta sección, presentamos una transformación fuente-a-fuente que, dado un programa $\mathcal{R}$, devuelve un nuevo programa $\mathcal{R}'$ que es semánticamente equivalente a $\mathcal{R}$ pero que puede ser analizado de forma más precisa. Básicamente, nuestro objetivo es obtener la misma información a partir de un BTA monovariante sobre el programa transformado $\mathcal{R}'$ que a partir de un BTA polivariante sobre el programa original $\mathcal{R}$.

Hablando de manera intuitiva, hacemos una copia de cada definición de función por cada llamada, al BTA, que devuelva diferentes valores de tiempo de enlace correspondientes a los argumentos de dicha función. Simplemente, consideramos los valores de *tiempo de enlace* S (estático, valor conocido) y D (dinámico, valor posiblemente desconocido). Defini-

mos la relación *menor cota superior*[9] ($\sqcup$) sobre los valores de *tiempo de enlace* como sigue:

$$\mathrm{S} \sqcup \mathrm{S} = \mathrm{S} \qquad \mathrm{S} \sqcup \mathrm{D} = \mathrm{D} \qquad \mathrm{D} \sqcup \mathrm{S} = \mathrm{D} \qquad \mathrm{D} \sqcup \mathrm{D} = \mathrm{D}$$

La operación de la menor cota superior puede ser extendida a secuencias de valores de *tiempo de enlace* en forma natural, e.g.,

$$\mathrm{SDS} \sqcup \mathrm{SSD} = \mathrm{SDD} \qquad \mathrm{SDS} \sqcup \mathrm{DSD} = \mathrm{DDD} \qquad \mathrm{SDS} \sqcup \mathrm{DSS} = \mathrm{DDS}$$

Un *entorno de tiempo de enlace*[10] es una función de sustitución de variables a valores de *tiempo de enlace*. Usaremos la siguiente función auxiliar B_e (adaptada de [JGS93]) para calcular un valor de *tiempo de enlace* de una expresión:

$$\begin{array}{lcll} B_e[[x]]\rho & = & \rho(x) & (\text{si } x \in \mathcal{V}) \\ B_e[[\mathsf{h}(t_1, \ldots, t_n)]]\rho & = & B_e[[t_1]]\rho \sqcup \ldots \sqcup B_e[[t_n]]\rho & (\text{si } \mathsf{h} \in \mathcal{C} \cup \mathcal{D}) \end{array}$$

donde ρ denota un *entorno de tiempo de enlace*. Hablando a grandes rasgos, una expresión ($B_e[[t]]\rho$) retorna S si t no contiene una variable que sea enlazada a D en ρ, y D en caso contrario.

Dado un término lineal $f(\overline{t_n})$ (comúnmente la parte izquierda de una regla), y una secuencia de valores de *tiempo de enlace* $\overline{b_n}$ para f, el entorno de *tiempo de enlace* asociado, $bte(f(\overline{t_n}), \overline{b_n})$, esta definido como sigue:

$$bte(f(\overline{t_n}), \overline{b_n}) = \{x \mapsto b_1 \mid x \in \mathcal{V}ar(t_1)\} \cup \ldots \cup \{x \mapsto b_n \mid x \in \mathcal{V}ar(t_n)\}$$

Definición 26 (transformación polivariante [ARTV09]) *Sea $\mathcal{R}$ un programa y $\overline{b_n}$ una secuencia de valores de* tiempo de enlace *para* main/n. *La transformación polivariante de $\mathcal{R}$ con respecto a $\overline{b_n}$, denotada por $\mathcal{R}^{\overline{b_n}}_{poly}$, es calculada como sigue:*

$$\begin{array}{rcl} \mathcal{R}^{\overline{b_n}}_{poly} & = & \{\ \mathsf{main}(\overline{x_n}) = pt(r, bte(\mathsf{main}(\overline{x_n}), \overline{b_n})) \mid \mathsf{main}(\overline{x_n}) = r \in \mathcal{R}\ \} \\ & & \cup\ poly\_trans(pc(\mathcal{R} \setminus \{\mathsf{main}(\overline{x_n}) = r\})) \end{array}$$

donde las funciones auxiliares poly_trans, pc y pt están definidas en la Figura 5.1. Aquí, denotamos por BT^n el conjunto de todas las secuencias posibles de n valores de tiempo de enlace.

[9] *lub* por sus siglas en inglés (*least upper bound*), también conocida como *supremum*.
[10] Del término en inglés: *binding-time environment*.

$$
\begin{array}{l}
poly\_trans(\{\,\}) = \{\,\} \\
poly\_trans(\{R\} \cup \mathcal{R}) = poly\_trans(\mathcal{R}) \\
\qquad \cup \left\{ \begin{array}{l}
\{f_{\overline{b_n}}(\overline{t_n}) = pt(r, bte(f(\overline{t_n}), \overline{b_n})) \mid \overline{b_n} \in BT^n\} \text{ si } R = (f(\overline{t_n}) = r) \\
\\
\{\mathsf{apply}_{b_n}(\mathsf{partcall}(f_{\overline{b_{n-1}}}, k, \overline{x_{n-1}}), x_n) = \mathsf{partcall}(f_{\overline{b_n}}, k-1, \overline{x_n}) \mid \\
\ \overline{b_n} \in BT^n\} \\
\ \text{si } R = (\mathsf{apply}(\mathsf{partcall}(f, k, \overline{x_{n-1}}), x_n) = \mathsf{partcall}(f, k-1, \overline{x_n})) \\
\\
\{\mathsf{apply}_{b_n}(\mathsf{partcall}(f_{\overline{b_{n-1}}}, k, \overline{x_{n-1}}), x_n) = f_{\overline{b_n}}(\overline{x_n}) \mid \overline{b_n} \in BT^n\} \\
\qquad\qquad \text{si } R = (\mathsf{apply}(\mathsf{partcall}(f, k, \overline{x_{n-1}}), x_n) = f(\overline{x_n}))
\end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
pc(\{\,\}) = \{\,\} \\
pc(\{R\} \cup \mathcal{R}) = \\
\quad \left\{ \begin{array}{ll}
pc(\{f(t_1, \ldots, \mathsf{partcall}(g_{\overline{b_m}}, k, \overline{t_m}), \ldots, t_n) = r \mid \overline{b_m} \in BT^m\} \cup \mathcal{R}) & \\
\qquad \text{si } R = (f(\overline{t_n}) = r),\ t_i = \mathsf{partcall}(g, k, \overline{t_m}),\ i \in \{1, \ldots, n\} & \\
\\
\{R\} \cup pc(\mathcal{R}) & \text{de otra forma}
\end{array} \right.
\end{array}
$$

$$
pt(t, \rho) = \left\{ \begin{array}{ll}
t & \text{si } t \in \mathcal{V} \\
c(\overline{pt(t_n, \rho)}) & \text{si } t = c(\overline{t_n}),\ c \in \mathcal{C} \\
f_{\overline{b_n}}(\overline{pt(t_n, \rho)}) & \text{si } t = f(\overline{t_n}),\ f \in \mathcal{D},\ B_e[\![t_i]\!]\rho = b_i, \\
 & \qquad i = 1, \ldots, n \\
\mathsf{partcall}(f_{\overline{b_n}}, k, \overline{pt(t_n, \rho)}) & \text{si } t = \mathsf{partcall}(f, k, \overline{t_n}),\ B_e[\![t_i]\!]\rho = b_i, \\
 & \qquad i = 1, \ldots, n \\
\mathsf{apply}_{b_2}(pt(t_1, \rho), pt(t_2, \rho)) & \text{si } t = \mathsf{apply}(t_1, t_2),\ B_e[\![t_i]\!]\rho = b_i, \\
 & \qquad i = 1, 2
\end{array} \right.
$$

Figura 5.1: Transformación Polivariante: $poly\_trans$ y pt.

Intuitivamente, la transformación polivariante procede como sigue:

- Primero, la parte izquierda de la función main no esta etiquetada ya que no hay ninguna llamada a main en el programa. La parte derecha es transformada como cualquier otra función definida por el usuario utilizando pt (véase el ejemplo 5.4).

- Para las reglas del programa (esto es, las reglas que definen funciones diferentes de apply), primero etiquetamos las ocurrencias de

partcall en las partes izquierdas[11] utilizando la función auxiliar pc.[12] Nótese que el primer caso de la definición de pc incluye la regla transformada en la siguiente llamada recursiva puesto que la parte izquierda puede contener varios argumentos partcall. En el segundo caso, cuando no permanecen ocurrencias de partcall, la regla es eliminada de la llamada recursiva.

- Después, reemplazamos las reglas restantes por un número de copias etiquetadas con todas las secuencias posibles de valores de *tiempo de enlace*, cuyas partes derechas son transformadas usando la función pt. Aquí, podríamos restringir los posibles valores de *tiempo de enlace* correspondientes a los argumentos $\overline{t_m}$ de la llamada parcial; para aquellos valores de *tiempo de enlace* que sean aproximaciones seguras. Sin embargo por facilidad mantenemos la actual formulación.

- Se transforman las reglas que definen a apply con el fin de hacer explícitos tanto los valores de *tiempo de enlace* como los nuevos argumentos. Obsérvese que etiquetamos el símbolo de función dentro de una llamada parcial pero no la propia llamada parcial. También, apply es etiquetada con el valor de *tiempo de enlace* de su segundo argumento; el valor de *tiempo de enlace* del primer argumento no es necesario ya que los valores de *tiempo de enlace* que etiquetan la función dentro la correspondiente llamada parcial ya contienen información más precisa.

- La función pt toma un término y un entorno de *tiempo de enlace* y procede como sigue:

  - Las variables y símbolos constructores se dejan intactos.
  - Las llamadas a función son etiquetadas con los valores de *tiempo de enlace* de sus argumentos de acuerdo al actual *entorno de tiempo de enlace*. Los símbolos de función incluidos en las

[11] De manera análoga a los argumentos de apply.

[12] Para mayor transparencia, asumimos que todas las ocurrencias de partcall aparecen como cabeza de los argumentos, esto es, ya sea que cualquiera de los argumentos t_i tenga la forma partcall(...) o que no contenga ocurrencias de partcall.

llamadas parciales también son etiquetadas de la misma forma.

- Las aplicaciones y llamadas parciales son etiquetadas como en la función *poly_trans*.

Nótese que etiquetar las funciones con todas las secuencias posibles los valores de *tiempo de enlace* produce normalmente un significativo incremento de tamaño de código. Claramente, se podría ejecutar un preprocesamiento para determinar los *patrones de llamadas* $f(\overline{b'_m})$ que pueden ocurrir desde la llamada inicial a $\mathsf{main}(\overline{b_n})$. No obstante, esta aproximación, implicaría una complejidad similar a construir el grafo de la llamada de orden superior de [Ser07]. Aquí, preferimos cambiar la complejidad de un algoritmo tal como el grafo de [Ser07] por la complejidad de copiar funciones etiquetadas en el dominio $\{\mathrm{S}, \mathrm{D}\}$. Además, muchas de estas copias son código muerto que será fácilmente eliminado en la fase de evaluación parcial (véase la Sect. 5.4).

Ejemplo 5.4 *Considérese el programa desfuncionalizado* $\mathcal{R}$ *del Ejemplo 5.3:*

$$\begin{array}{ll}
\mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & \\
f(Z) = \mathsf{partcall}(inc,1) & inc(n) = Succ(n) \\
f(Succ(n)) = \mathsf{partcall}(g,1,f(n)) & \\
g(\mathsf{partcall}(inc,1),a) = inc(inc(a)) & \\
g(\mathsf{partcall}(g,1,x),a) = g(x,g(x,a)) & \\
\mathsf{apply}(\mathsf{partcall}(inc,1),x) = inc(x) & \\
\mathsf{apply}(\mathsf{partcall}(g,2),x) = \mathsf{partcall}(g,1,x) & \\
\mathsf{apply}(\mathsf{partcall}(g,1,x),y) = g(x,y) &
\end{array}$$

Dada la secuencia inicial de los valores de tiempo de enlace SD, *nuestra*

transformación polivariante genera el siguiente programa $\mathcal{R}^{\text{SD}}_{poly}$:[13]

$$\mathsf{main}(x, y) = \mathsf{apply}_{\text{D}}(f_{\text{S}}(x), y)$$

$$f_{\text{S}}(Z) = \mathsf{partcall}(inc, 1) \qquad inc_{\text{S}}(n) = Succ(n)$$
$$f_{\text{S}}(Succ(n)) = \mathsf{partcall}(g_{\text{S}}, 1, f_{\text{S}}(n)) \qquad inc_{\text{D}}(n) = Succ(n)$$

$$g_{\text{SD}}(\mathsf{partcall}(inc, 1), a) = inc_{\text{D}}(inc_{\text{D}}(a))$$
$$g_{\text{SD}}(\mathsf{partcall}(g_{\text{S}}, 1, x), a) = g_{\text{SD}}(x, g_{\text{SD}}(x, a))$$

$$\mathsf{apply}_{\text{D}}(\mathsf{partcall}(inc, 1), x) = inc_{\text{D}}(x)$$
$$\mathsf{apply}_{\text{D}}(\mathsf{partcall}(g_{\text{S}}, 1, x), y) = g_{\text{SD}}(x, y)$$

La siguiente sección presenta un resumen de una evaluación experimental lograda con nuestro prototipo de implementación de evaluación parcial.

5.4. La transformación llevada a la práctica

En esta sección, presentamos un resumen de nuestro avance en el desarrollo de un evaluador parcial que sigue las ideas presentadas hasta el momento. La implementación emprendida sigue estas líneas directrices:

- El sistema acepta programas de orden superior los cuales son primeramente transformados usando las técnicas de la Secc. 5.2 (desfuncionalización) y Secc. 5.3 (transformación polivariante).
- Después se aplica el análisis *size-change* estándar de [ARSV07] (para programas de primer orden) a los programas transformados.
- Finalmente, anotamos el programa utilizando el resultado del análisis *size-change* y aplicamos la fase de especialización del actual evaluador parcial *offline* [RSV05a, ARSV07]. Nótese que no se requiere propagación de valores de *tiempo de enlace* en este caso[14] puesto

[13] Para tener una mejor claridad no mostramos algunas otras reglas (inútiles) que la transformación produce actualmente. Nótese también que, de acuerdo a nuestra técnica, la ocurrencia de *inc* en la expresión $\mathsf{partcall}(inc, 1)$ debe ser etiquetada con una secuencia vacía de valores de *tiempo de enlace*. Sin embargo para mayor simplicidad, especificamos solo *inc*.

[14] En el esquema original, el valor de *tiempo de enlace* de cada argumento de función es requerido para identificar bucles estáticos que pueden ser desplegados con toda seguridad.

Cuadro 5.1: Tiempos de ejecución de programas de evaluación.

programa de ejemplo	tiempo original	especializados		poli especializados	
		tiempo	incremento	tiempo	incremento
ack	1526	507	3.01	522	2.92
bulyonkov	559	727	0.77	402	1.39
combinatorial	991	887	1.12	612	1.62
changeargs	772	1157	0.67	478	1.62
dfib (HO)	326	294	1.11	95	3.43
dmap (HO)	905	427	2.12	885	1.02
Promedio	**760**	**602**	**1.26**	**416**	**1.83**

que dicha información se encuentra ya explícita en cada llamada de función gracias a la transformación polivariante.

El cuadro 5.1[15] muestra la eficacia de nuestra transformación sobre los siguientes ejemplos: `ack`, la bien conocida función de Ackermann, la cual es especializada para un primer argumento dado; `bulyonkov`, es una ligera extensión de la ejecución de un ejemplo incluido en [Bul93]; `combinatorial`, un programa sencillo que incluye el cómputo de combinatorias; changeargs, otra variante de la ejecución de un ejemplo incluido en [Bul93]; `dfib`, un ejemplo de orden superior que usa la conocida función de Fibonacci; `dmap`, un ejemplo de orden superior con una función para correlacionar dos funciones a cada elemento de una lista.

Para los ejemplos de primer orden, consideramos el anterior evaluador parcial de [RSV05a, ARSV07], la única diferencia es que en las dos últimas columnas el programa respectivo es procesado primeramente con la transformación polivariante. Como puede verse, la transformación polivariante mejora la rapidez de ejecución en tres de cuatro ejemplos.

Para los ejemplos de orden superior, puesto que el evaluador parcial anterior no acepta programas de orden superior, comparamos el nuevo evaluador parcial *offline* con un evaluador parcial *online* para Curry que acepta funciones de orden superior [AHV02]. En este caso, obtenemos una mejora en uno de los ejemplos (`dfib`) y una reducción de la rapidez de ejecución en otro (`dmap`). Este resultado no es extraordinario, ya que un evaluador parcial *online* es normalmente capaz de propagar mucho más

[15]Los tiempos de ejecución están en milisegundos.

Cuadro 5.2: Comparación de tamaños de código (bytes).

programa de ejemplo	tamaño original	especializados		poly especializados	
		tamaño	variación	tamaño	variación
ack	951	3052	3.21	4168	4.38
bulyonkov	2250	3670	1.63	2440	1.08
combinatorial	2486	3546	1.43	6340	2.55
changeargs	3908	5335	1.37	5599	1.43
dfib (HO)	2911	4585	1.58	6204	2.12
dmap (HO)	2588	5236	2.02	3279	1.27
Average	**2321**	**4147**	**1.79**	**4408**	**1.90**

información que un evaluador parcial *offline*. No obstante, el comentario importante aquí es que ahora somos capaces de tratar con programas que no podrían ser tratados con la version antigua del evaluador parcial *offline*.

Los promedios son obtenidos a partir de la media geométrica de los resultados de mejora en la rapidez de ejecución.

Un problema crítico de nuestra transformación es que puede producir un incremento significativo en el tamaño de código. Esto se considera en el cuadro 5.2. Aquí, aunque el tamaño del programa residual generado con nuestro enfoque es ligeramente mayor que el tamaño del programa residual obtenido con aproximaciones anteriores, este aún es razonable. Actualmente, las pruebas de evaluación realizadas confirman que la mayoría del código añadido en la transformación polivariante ha sido eliminado en tiempo de especialización.

5.5. Trabajo relacionado y conclusiones

La desfuncionalización fue originalmente introducida por Reynolds [Rey98] (véase también [DN01], donde se presentan varias aplicaciones). La desfuncionalización ya ha sido usada en el contexto de evaluación parcial (véase, e.g., [Bon90]) así como en la aproximación *online* de evaluación parcial dirigida por *narrowing* [AHV02]. La principal novedad con respecto a estas aproximaciones es que hemos introducido una desfuncionalización más agresiva por generación de instancias de variables funcionales con todas las llamadas parciales posibles. Aunque esto puede

incrementar la extensión del código, el programa transformado tiene más información explícita y en conjunto con el análisis *size-change* y el proceso de especialización puede producir mejores resultados.

El análisis *size-change* ha sido extendido recientemente en [Ser07] a programas funcionales de orden superior. En contraste a nuestro enfoque, Sereni propone una aproximación directa sobre programas de orden superior que requiere la construcción de un complejo grafo de llamadas que podría producir un análisis de *tiempo de enlace* menos eficiente. Hemos aplicado nuestra técnica al ejemplo mostrado en [Ser07] y obtuvimos la misma precisión (a pesar del uso de la desfuncionalización). Una comparación más profunda es tema de trabajo en curso.

Respecto a la definición de la aproximación transformaciónal al BTA polivariante, solo encontramos el trabajo de [Bul93]. En contraste a nuestro enfoque, Bulyonkov duplica los argumentos de función tal que, para cada argumento de la función original, hay otro argumento con su valor de *tiempo de enlace*. Además, se agrega algo de código adicional para calcular los valores de *tiempo de enlace* de las llamadas en las partes derechas de las definiciones de función. Después, se ejecuta una primera etapa de la evaluación parcial con algunos valores concretos de *tiempo de enlace* correspondientes a los argumentos de algunas funciones. Como resultado, el programa especializado puede incluir diferentes versiones de la misma función (para las diferentes combinaciones de valores de *tiempo de enlace*). Después la evaluación parcial se aplica nuevamente, usando los valores actuales de los argumentos estáticos. Nuestro enfoque reemplaza la primera etapa de transformación y evaluación parcial por una simple transformación basada en duplicación de código y etiquetado de los símbolos de función. No se hizo comparación experimental ya que no tenemos conocimiento de alguna implementación de la aproximación de Bulyonkov.

Otras aproximaciones al BTA polivariante de programas de orden superior incluyen el trabajo de [Mog89] para programas funcionales y la aproximación modular de Vanhoof [Van00] para programas Mercury. En contraste a nuestro enfoque, Mogensen presenta una aproximación *directa* (es decir, no está basada en desfuncionalización) para BTA polivariante

de orden superior de programas funcionales[16]. El enfoque de Vanhoof es también una aproximación directa a un BTA polivariante de programas Mercury de orden superior. Un aspecto interesante de [Van00] es que no requiere análisis de cerradura, puesto que las cerraduras están encapsuladas en el concepto de valor de *tiempo de enlace*. La integración de algunas ideas de [Van00] en nuestro contexto podrían mejorar la precisión del método y reducir el incremento de la talla del código.

Otro enfoque relacionado para mejorar la precisión del análisis de terminación por etiquetado de funciones se encuentra en [SKGST08], el cual está basado en una técnica estándar a partir de programación lógica [Apt97]. Aquí, algunas cláusulas de programa están duplicadas y etiquetadas con diferentes *modos* —el modo de un argumento puede ser de *entrada*, si éste es conocido al momento de la llamada, o de *salida*, si es desconocido— a fin de tener un programa con modos bien definidos donde cada llamada al mismo predicado tenga los mismos modos. Esta técnica puede ser vista como una versión simple de nuestra transformación polivariante.

Para resumir, en este capítulo hemos introducido una aproximación de transformación hacia un BTA polivariante de orden superior para programas funcionales. Nuestra estrategia está basada en dos transformaciones diferentes: un algoritmo de desfuncionalización mejorado que hace explícita la información de orden superior tanto como sea posible, conjuntamente con una transformación polivariante que mejora la precisión de propagación de los valores de *tiempo de enlace*. Hemos desarrollado una implementación prototipo de evaluación parcial completa, lo que constituye el primer evaluador parcial *offline* dirigido por *narrowing* que trata con programas de orden superior y produce especializaciones polivariantes [ARTV09]. Nuestros resultados experimentales son alentadores y establecen que el nuevo BTA es eficiente y suficientemente preciso.

En cuanto a trabajo futuro, hay varios temas interesantes que planeamos investigar posteriormente. Como se menciona antes, [Van00] presenta algunas ideas que podrían ser adaptadas a nuestro contexto con

[16]Actualmente, la aproximación de Mogensen incluye pasos semejantes a un proceso de desfuncionalización pero nunca agregan una definición para una determinada aplicación de función.

el fin de mejorar la precisión del BTA y evitar la explosión de código debida a la carencia de un análisis de cerradura en nuestra transformación. Además, el uso de dominios más refinados de análisis de *tiempo de enlace* (incluyendo información parcialmente estática como en, e.g. [Mog89, Van00]) pueden mejorar aún más la precisión de la especialización a un costo razonable.

Capítulo 6

Especialización de intérpretes aplicando NPE *offline*

El proceso de la compilación por especialización de intérpretes consiste en una transformación de programas fuente a fuente que ha inspirado el trabajo de investigadores en evaluación parcial por muchos años. Las últimas investigaciones sobre NPE *offline* nos permiten especializar programas más grandes, condición necesaria para realizar la especialización de intérpretes. En este capítulo introduciremos las etapas de un nuevo evaluador parcial *offline* puro desarrollado en el lenguaje lógico funcional Curry el cual es capaz de especializar programas en el lenguaje de metaprogramación denominado *FlatCurry*. En particular describiremos los primeros experimentos en la especialización de intérpretes, considerando que nuestro evaluador parcial especializa programas más realistas que versiones anteriores ya que es posible procesar programas que incluyen *built-ins* y *constraints*. Dado que la especialización de intérpretes es una aplicación de la evaluación parcial, en este capítulo hemos utilizado varios conceptos, desarrollos y definiciones publicados en [ARSV07, ARTV07] y [RSV05a] inclusive.

6.1. Introducción

Es bien conocida la existencia de dos métodos para describir formalmente la semántica de un lenguaje de programación. El primero consiste

en describir el proceso de traducción por medio de un lenguaje fuente a otro lenguaje objeto cuya semántica es conocida, es decir, la descripción de un traductor. El segundo consiste en describir un procedimiento para evaluar las declaraciones que pertenecen al lenguaje a ser definido, es decir, la descripción de un intérprete [Fut99]. Aunque los intérpretes son fáciles de escribir y mantener, éstos son ineficientes. Por otro lado el programa ejecutable de una compilación es más eficiente, pero el compilador es más costoso de implementar. Una forma de obtener los mejor de ambos enfoques es realizar la especialización de un intérprete, con el propósito de generar automáticamente una implementación eficiente [TBC+98].

La evaluación parcial de programas es una técnica formal para la especialización y optimización de programas que se basa en la semántica del lenguaje, técnica que ha sido investigada en diferentes paradigmas de programación y aplicada a una amplia variedad de lenguajes. También es conocida como una técnica de transformación de programas fuente a fuente para especializar un programa con respecto a una parte de sus datos de entrada (por lo tanto también es llamada especialización de programas).

Las virtudes de la escritura de intérpretes y el mejor rendimiento de los compiladores pueden obtenerse por medio de la evaluación parcial. Se obtiene, la facilidad de crear prototipos de intérpretes en general y la eficiencia de los compiladores. Partiendo del hecho que la especialización de un intérprete (de un programa) es la compilación [And92].

En este capítulo, presentamos la especialización de intérpretes escritos en el lenguaje funcional Curry con definición de funciones de primer orden. Para ello utilizamos la aproximación *offline* mejorada del evaluador parcial dirigido por *narrowing* presentada en el capítulo 4, es decir, la versión *offline* pura de EP. Hemos incluido una extensión considerando varios *built-ins* y *constraints*. Este capítulo está organizado como sigue: la Sección 6.2 introduce la estructura general de nuestro evaluador parcial llamado **mixpo**. Después, la sección 6.3 presenta una explicación de la implementación de intérpretes. La sección 6.4 muestra la propia especialización de intérpretes incluyendo *built-ins* y *constraints*. En la sección 6.5 se incluye una discusión de algunos trabajos relacionados y finalmente se concluye en la sección 6.6.

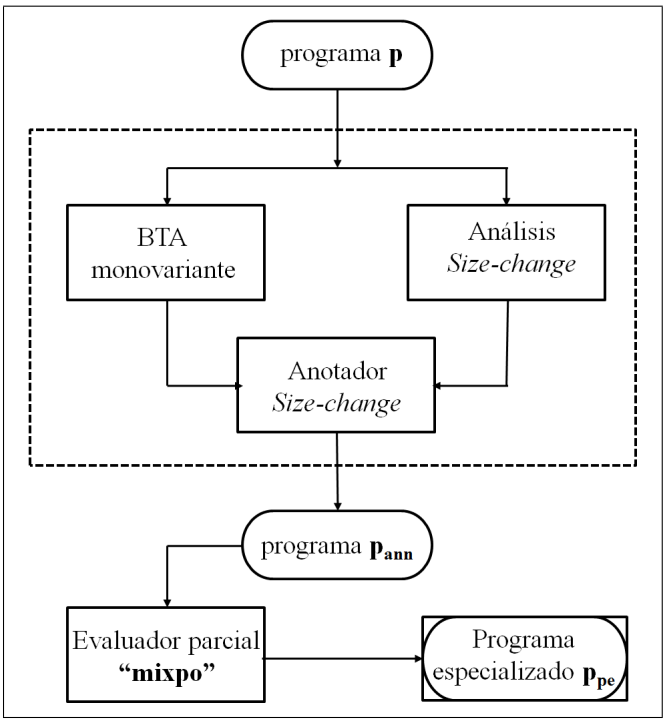


Figura 6.1: Esquema del evaluador parcial *offline*.

6.2. La estructura de *mixpo*

La Figura 6.1 muestra el esquema completo de nuestra implementación, podemos ver que los procesos contenidos en caja punteada incluyen tanto el BTA como el SCA, ambos procesos toman como dato de entrada el programa (**p**) a ser especializado y entregan sus resultados al propio proceso de anotación; que produce el programa anotado ($\mathbf{p_{ann}}$). Después el proceso de evaluación parcial (**mixpo**) recibe $\mathbf{p_{ann}}$ y ejecuta la propia evaluación parcial para producir el programa especializado $\mathbf{p_{pe}}$ final.

Presentamos un procedimiento de anotación que está basado originalmente en el análisis de cuasi-terminación de [ARSV07] (véase la sección 4.2), y que fue mejorado en [ARTV07]. Además, puesto que dicho análisis de cuasi-terminación fue establecido originalmente para SRTs, éste también fue adaptado al lenguaje *flat* (véase la sección 4.5).

Para realizar la especialización de intérpretes usamos las mismas defini-

```
data Nat = Z | S Nat
main x = prod (x) (fib (S (S (S (S (S Z))))))

fib x = case x of
          Z -> Z
          (S Z) -> (S Z)
          (S (S n)) -> sum (fib (S n)) (fib n)

prod x y = case x of
             Z -> Z
             (S w) -> sum y (prod w y)

sum x y = case x of
            Z -> y
            (S w) -> S (sum w y)
```

Figura 6.2: Programa de la sucesión de `Fibonacci` para números naturales.

ciones del análisis de cuasi-terminación de [ARTV07], pero hacemos un ligero cambio en el procedimiento de anotación de programas.

Ejemplo 6.1 *Consideremos el programa de la sucesión de* `fibonacci` *para números naturales mostrado el la Figura 6.2. En general hemos calculado ocho grafos* size-change*, pero el SCA produce tres multigrafos idempotentes que se muestran en la Figura 6.3.*

Para la anotación de programas nos basamos prácticamente en el procedimiento de la sección 4.7.1. Hemos cambiado el criterio de anotación para la generalización, esto es, ya no agregamos anotaciones para generalización. Ahora creamos un vector de generalización en el programa anotado, el cual indica al procedimiento de especialización los términos que deben ser generalizados, esto es, para indicar qué argumentos de función deben generalizarse (véase la definición `GEN2`[1] en la Figura 6.4, que corresponde

[1]Podemos observar que no existe la definición del constructor BtD en el programa anotado, la razón obedece a que la generación de este programa es un paso intermedio entre la anotación y la especialización, i.e., no se ejecuta el programa anotado.

$$\mathcal{G}_4: \quad \texttt{fib} \longrightarrow \texttt{fib} \qquad\qquad \mathcal{G}_7: \quad \texttt{prod} \longrightarrow \texttt{prod}$$
$$1_{\texttt{fib}} \xrightarrow{\succ} 1_{\texttt{fib}} \qquad\qquad 1_{\texttt{prod}} \xrightarrow{\succ} 1_{\texttt{prod}}$$
$$2_{\texttt{prod}} \xrightarrow{\succsim} 2_{\texttt{prod}}$$

$$\mathcal{G}_8: \texttt{sum} \longrightarrow \texttt{sum}$$
$$1_{\texttt{sum}} \xrightarrow{\succ} 1_{\texttt{sum}}$$
$$2_{\texttt{sum}} \xrightarrow{\succsim} 2_{\texttt{sum}}$$

Figura 6.3: Multigrafos idempotentes del programa de la sucesión de `fibonacci`.

a este vector). Con el vector de generalización, ahora el control global del proceso de especialización se encargará de generalizar; puesto que ahora no existen anotaciones GEN que pueda procesar el control local, con lo que se espera una ejecución del nivel de control local más rápida.

Ejemplo 6.2 *Consideremos nuevamente el programa del ejemplo 6.1 de la sucesión de* `fibonacci` *para números naturales. Vamos a considerar una llamada a la etapa de anotación incluyendo este programa y el valor abstracto* [D] *como argumento de la definición* `main`*. Nuestro BTA devuelve como resultado la siguiente división:*

$$\{\texttt{main} \mapsto (\texttt{D}),\ \ \texttt{fib} \mapsto (\texttt{S}),\ \ \texttt{prod} \mapsto (\texttt{D},\texttt{S}),\ \ \texttt{sum} \mapsto (\texttt{S},\texttt{D})\}$$

donde S *denota que el argumento es estático y* D *que éste es dinámico. Aquí, nuestro procedimiento de anotación devuelve el programa anotado de la Figura 6.4. De acuerdo a la función* ann^u *de la Figura 4.5, podemos anotar las llamadas a función* `fib` *y* `sum` *con* `UNF`*; porque ambas tienen al menos un arco* $i_f \xrightarrow{\succ} i_f$ *en sus respectivos multigrafos idempotentes, tales que el argumento* i_f *es estático. También de acuerdo con la función* ann^u `prod` *se anota con* `MEM`*, es decir, no cumple con las primeras tres alternativas de* ann^u*(véase la función de anotación* ann^u *en la Figura 4.5); al menos el primer argumento de* `prod` *debe ser anotado con* `GEN`*. Sin embargo ahora para cumplir con este requerimiento; agregamos una tupla al vector de generalización. Esta tupla debe contener: nombre de función, aridad, y los valores* [S] *o* [D] *para los argumentos. Donde únicamente* [D] *indica que el correspondiente término debe ser generalizado en la etapa de especialización.*

```
data Nat = Z | S Nat

GEN2 = BtD ("prod", 2, [D, S])

main v1 = MEM (prod v1 (UNF (fib (S (S (S (S (S Z))))))))

fib eval rigid
fib Z = Z
fib (S Z) = (S Z)
fib (S (S v3)) = UNF (sum (UNF (fib (S v3))) (UNF (fib v3)))

prod eval rigid
prod Z v2 = Z
prod (S v3) v2 = UNF (sum v2 (MEM (prod v3 v2)))

sum eval rigid
sum Z v2 = v2
sum (S v3) v2 = S (UNF (sum v3 v2))

UNF v1 = v1
MEM v1 = v1
```

Figura 6.4: Programa anotado de la sucesión de `Fibonacci`.

Hasta ahora, hemos descrito únicamente la etapa de anotación. Evocando la etapa de especialización, primero recordaremos el procedimiento genérico de NPE [AV02], que se muestra en la Figura 4.7, el cual es similar al procedimiento de Gallager para evaluación parcial de programas lógicos [Gal93]. Dado un conjunto de términos encabezados por operación[2], esto es, llamadas a función, el procedimiento aplica un operador de desplegado *unfold* de tal forma que devuelve un conjunto de reglas residuales. Este operador *unfold* está dirigido por las anotaciones `UNF`, de modo que todas las funciones anotadas con `UNF` deben ser desplegadas ya que de acuerdo con los multigrafos idempotentes que originaron la ano-

[2]Una expresión está encabezada por operación si está encabezada por un símbolo de función.

tación; existe al menos un argumento con un orden estricto de reducción ($\succ$) y estático, lo que significa que su valor es conocido al momento de la especialización. Esto ciertamente asegura que la evaluación (parcial) de la función termine. Mientras que las funciones anotadas con `MEM` deben suspender de evaluación de la llamada y retornar la evaluación al control global.

Las derivaciones que se calculan durante el proceso de especialización en el control local están basadas en una ligera extensión del cálculo RLNT [AHV03]. El cálculo RLNT es una semántica no-estándar usada para evaluar parcialmente las llamadas a función y describe las operaciones básicas de cómputo de los programas *flat*. Originalmente los programas especializados con la semántica RLNT fueron procesados sólo de manera *online*, en la configuración *offline*, requerimos agregar anotaciones a las funciones y a sus argumentos; el cálculo RLNT fue extendido para soportar esas anotaciones. Esta semántica extendida está especificada en [ARTV07] y la podemos ver en la Figura 4.8 de esta tesis. Cada cómputo realizado con el cálculo RLNT genera *reglas residuales* las cuales componen el programa residual.

Por otro lado, el operador *abstract* toma un conjunto finito de términos encabezados por operación T_i y luego agrega correctamente el conjunto de subtérminos en las respectivas partes derechas de las llamadas desplegadas, el cual está denotado por P'_{calls}. El nuevo conjunto T_{i+1} puede requerir una evaluación adicional, por lo tanto, el proceso se repite iterativamente mientras se introduzcan nuevos términos.

Respecto al vector de generalización, cuando el control global encuentra un término encabezado por operación con subtérminos marcados con `D` dentro de éste vector; el término debe ser aplanado antes de ser agregado al conjunto de llamadas a evaluar [RSV05a]. Por ejemplo, dado un término $f\ (g\ (x),\ h(y))$, si el vector de generalización contiene una tupla con $(\text{"}\texttt{f}\text{"}, 2, [\texttt{D}, \texttt{D}])$; entonces se agregan las llamadas $f(w1, w2),\ g(x)$ y $h(y)$ al actual conjunto de llamadas a función (a ser) evaluadas parcialmente, es decir, a T_{i+1}, donde $w1, w2$ son variables nuevas que corresponden a la generalización de los términos $g(x)$ y $h(y)$.

Hablando de manera intuitiva, el programa fuente de la Figura 6.2 muestra que la función `fib` puede ser completamente evaluada, es decir,

```
module fib2_ann (Nat(Z,S), main, prod_pe1) where

data Nat = Z | S Nat

main :: b-> a
main v0 = prod_pe1 v0

prod_pe1 :: b-> a
prod_pe1 eval rigid
prod_pe1 Z = Z
prod_pe1 (S v6) = S (S (S (S (prod_pe1 v6))))
```

Figura 6.5: Programa especializado de la sucesión de `fibonacci`.

el programa puede ser especializado evaluando únicamente la función `fib`. Si usamos el programa anotado de la Figura 6.4, podemos ver que se pueden desplegar completamente tanto `sum` como `fib` ya que ambas están anotadas con `UNF`. Si especializamos el programa anotado con nuestro evaluador parcial (**mixpo**) obtenemos el programa que se muestra en la Figura 6.5. **mixpo** genera el programa especializado en lenguaje *FlatCurry*, sin embargo PAKCS [HeAE+04] tiene la facilidad de mostrar este programa en Curry. De hecho, podemos ver que `sum` y `fib` han sido evaluadas completamente. Del mismo modo, la función `prod` ha sido evaluada parcialmente. En resumen, el resultado muestra que por cada unidad indicada en la variable de `main`, esto es, en `v0`; se deben agregar cinco unidades hasta llegar a la regla básica `prod_pe1 Z`, es decir a cero. Por ejemplo, si `v0` = (`S Z`) la llamada de `main` unifica con una instancia de la segunda regla de `prod_pe1`, que a su vez vuelve a llamar `prod_pe1` que ahora unifica con la primera regla. El resultado es `S (S (S (S ( Z))))`. Si `v0` = `S (S Z)` la segunda regla de `prod_pe1` se unifica dos veces antes de instanciar a la primera regla. Lo que genera como resultado `S (S (S (S ( S (S (S (S ( Z))))))))`.

6.3. Implementación de intérpretes

En esta sección introducimos detalles de implementación de los intérpretes, pero primero presentamos la información acerca de las facilidades de metaprogramación de Curry [Han11].

6.3.1. Meta-programación en Curry.

La implementación de nuestro evaluador parcial y nuestro intérprete de Curry están basados en las facilidades de meta-programación del lenguaje Curry. En particular, consideramos el lenguaje intermedio *FlatCurry* para la representación de programas lógico funcionales (disponible por medio del uso de las librerías de Curry *FlatCurry* and *FlatCurry-Tools*). En la librería *FlatCurry*, todas las funciones están definidas en el nivel superior (es decir, las declaraciones de función locales en programas fuente se hacen globales por medio de *lambda lifting*[3]) y la estrategia del emparejamiento de patrones se hace explicita por medio del uso de expresiones *case*. En esta configuración, un programa *FlatCurry* está representado por medio del siguiente tipo de datos:

```
data Prog = Prog String    -nombre del módulo
                [String]   -módulos importados
                [TypeDecl] -declaración de tipos
                [FuncDecl] -declaración de funciones
                [OpDecl]   -declaración de operadores
```

Para simplificar, aquí sólo se muestran los tipos de datos para representar declaraciones de función:

```
data FuncDecl = Func QName      -nombre calificado
                     Int        -aridad
                     Visibility -función pública/privada
                     TypeExpr   -tipo de la expresión
                     Rule       -regla de programa
data Rule = Rule [VarIndex] Expr
```

[3]Es una técnica para transformar un programa funcional con definiciones de función locales, posiblemente con variables libres dentro de las definiciones de función, a un programa que consiste únicamente de definiciones de función globales [Joh85].

Por lo tanto, cada función es representada por una sóla regla cuya parte izquierda contiene índices de diferentes variables (`[VarIndex]`) y cuya parte derecha es una expresión que puede contener variables, literales, llamadas a función y a contructor, disyunciones y expresiones *case*:

```
data Expr = Var VarIndex
          | Lit Literal
          | Comb CombType QName [Expr]
          | Or Expr Expr
          | Case CaseType Expr [BranchExpr]
data CombType   = FuncCall | ConsCall
data CaseType   = Rigid | Flex
data BranchExpr = Branch Pattern Expr
data Pattern    = Pattern QName [VarIndex] | LPattern Literal
```

Considérese, por ejemplo, el programa Curry `reverse.curry`, el cual se muestra a continuación, y que define la función que invierte una lista con un parámetro que se usa como acumulador:

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

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

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

Los programas fuente en Curry pueden ser leidos y traducidos a *FlatCurry* usando la función `readFlatCurry`. Después, se puede manipular la estructura de datos en *FlatCurry*, la cual representa el programa, y finalmente, escribir el programa manipulado a un fichero *FlatCurry* utilizando

la función `writeFCY`.

6.3.2. Semántica operacional para la implementación de intérpretes

Razonando que un intérprete es un tipo de semántica operacional de bajo nivel, y que por lo tanto puede servir como una definición de un lenguaje de programación [JGS93], en la figura 6.6 mostramos la semántica operacional que deben seguir nuestros intérpretes de manera informal. Sin embargo está basada en el cálculo RLNT de [AHV02] y en la sintaxis del subconjunto del lenguaje *flat* mostrado en la sección 6.2. Creemos que esto es suficiente para implementar nuestros intérpretes a ser especializados. Es necesario comentar que no consideramos algunas características de Curry tales como: Orden superior ni programación concurrente para estas implementaciones, pero algunos *built-ins* y *constraints* sí lo están.

Vamos a describir brevemente nuestra semántica operacional: Los símbolos "$[\![$" y "$]\!]$" en una expresión como $[\![e]\!]$, no denotan una función semántica, únicamente son usados para identificar qué parte de una expresión debe ser aún evaluada. Las dos reglas clasificadas dentro de HNF se aplican cuando el término considerado está en forma normal en cabeza, es decir, se trata de una variable o un término encabezado por constructor. Function Eval, en el cálculo RLNT original representaba el desplegado de una llamada a función, esto es considerando un desplegado puramente funcional ya que todos los argumentos en las partes izquierdas de las reglas son variables. Sin embargo, ya que estamos tomando en cuenta algunos *built-ins* y *constraints* como: "==", "+" , "-" , "*"; donde éstos también son funciones, entonces las expresiones con estos símbolos podrían ser evaluados al momento. Por lo tanto realizamos acciones directas para aplicar estos *built-ins* en el intérprete con el objeto de tener código simple y sencillo por el momento. **mixpo** también es capaz de procesar algunos *built-ins* y *constraints*. Analizaremos algunos ejemplos de intérpretes en la sección 6.4.

Case Eval primero evalúa el argumento del *case* creando una llamada a éste subtérmino, como podemos ver esta regla excluye la evaluación de una expresión *case* cuyo argumento sea una variable o un término

HNF

$$[\![e]\!] \Rightarrow \text{e Si e} \in \mathcal{V} \text{ o } e = c() \text{ con } c \in \mathcal{C}$$
$$[\![c(e_1, \ldots, e_n)]\!] \Rightarrow c([\![e_1]\!], \ldots, [\![e_n]\!])$$

Function Eval

$$[\![g(\overline{e_n})]\!] \Rightarrow [\![\sigma(r)]\!] \text{ Si } g(\overline{x_n}) = r \in \mathcal{R} \text{ es una definición de función con variables frescas y } \sigma = \{\overline{x_n \mapsto e_n}\}$$

Case Eval

$$[\![(f)case\ e\ of\ \{\overline{p_k \to e_k}\}]\!] \Rightarrow [\![(f)case\ e'\ of\ \{\overline{p_k \to e_k}\}]\!] \text{ Si } [\![e]\!] \Rightarrow [\![e']\!],\ e \notin \mathcal{V},\ root(e) \notin \mathcal{C}, \text{ y } e \neq (f)case\ (\_)\ of\ \{\ldots\}$$

Figura 6.6: Semántica operacional para la implementación de intérpretes.

encabezado por constructor, o cualquier otra expresión *case*.

6.3.3. Descripción de la implementación de intérpretes

En el ejemplo del programa de la sucesión de `fibonacci` podemos ver que incluye la definición de la función `main`, la que **mixpo** busca por omisión para comenzar la especialización. Así pues, si queremos especializar un intérprete con respecto a un *programa*, este último debe estar inmerso dentro del mismo archivo que el intérprete. Ponemos en la parte derecha de `main` la llamada a la función principal del intérprete con los siguientes argumentos: una bandera booleana, el *programa* y la expresión a ser evaluada. Previamente debemos traducir el *programa* a *FlatCurry* para incluir únicamente la propia lista de funciones del *programa* en formato *flat*. Vamos a aclarar esto, en el apéndice A-1, podemos ver un ejemplo de un intérprete en Curry, ahí tenemos la función:

```
int p e = do (Prog _ _ _ funs _ ) <- readFlatCurry p
             print (ieval True funs e)
```

la cual esta encerrada por {--}. Ésta se usa sólo para probar el intérprete, esta es la razón por la que se pone como un comentario en el intérprete. `int` realiza dos acciones, lee el *programa* e imprime la evaluación de la expresión `e`. El tipo de datos para representar un programa Curry en la

forma intermedia, esto es, en *FlatCurry*, tiene el contructor `Prog` y cinco argumentos donde `funs` es el conjunto de las funciones del *programa*. Así pues damos sólo la estructura `funs` al intérprete como *programa* a ser interpretado, `ieval` es la función principal del intérprete. Si observamos la parte derecha de la función `main`, uno de los argumentos de `ieval` es:

```
[Func (“rev2”,“main´´) 0 Public (TCons (“Prelude",“Int") [ ])
   (Rule [ ]
    (Comb FuncCall (“Prelude”,“+”) [Comb FuncCall ("Prelude",“-”)
      [Comb FuncCall (“Prelude","*") [Lit (Intc 3),Lit (Intc 4)],
            Lit (Intc 1)],Lit (Intc 2)]))]
```

el cual representa la lista de funciones (`funs`) en *FlatCurry* de la siguiente definición Curry:

```
main = 3*4-1+2
```

esta definición también forma parte de un comentario en el intérprete. Podemos especializar este sencillo programa con **mixpo**, capaz de especializar *built-ins*. Tal como esperamos se obtiene el siguiente programa:

```
{- Program file: metaintf/examples/rev2_ann_pe -}
module rev2_ann(main) where
main :: a
main = 13
{-  end of module metaintf/examples/rev2_ann_pe -}
```

Ahora, si especializamos el intérprete del apéndice, es decir, primero anotarlo y luego especializarlo, obtenemos el siguiente programa:

```
module int_arith_ann(main,ieval_pe1) where
import FlatCurry
main :: a
main = ieval_pe1
ieval_pe1 :: a  ieval_pe1 =
     FlatCurry.Comb FlatCurry.FuncCall (“Prelude”,“+”)
       [FlatCurry.Comb FlatCurry.FuncCall (“Prelude”,“-”)
          [FlatCurry.Lit (FlatCurry.Intc 12),FlatCurry.Lit
            (FlatCurry.Intc 1)],
                 FlatCurry.Lit (FlatCurry.Intc 2)]
```

Observamos que todo el código del intérprete haya sido eliminado, pero tenemos como resultado de la evaluación parcial sólo dos definiciones

de función: la función `main` y la función `ieval` evaluada parcialmente.

Siguiendo con la descripción de la implementación de la estructura de los intérpretes (que se refiere al ejemplo del apéndice A-1), tenemos dos reglas para evaluar términos encabezados por constructor, sólo una de ellas despliega; la regla con el argumento `True`, el desplegado se detiene hasta que alcanza un término HNF (forma normal en cabeza). `ievalList` evalúa la lista de expresiones función/constructor. Obsérvese las siguientes reglas:

```
ieval True v2  (Comb ConsCall v8 v9) =
                      (Comb ConsCall v8 (ievalList True v2 v9) )
ieval False _   (Comb ConsCall c es) = Comb ConsCall c es
```

En tanto que `ieval False _ (Comb ConsCall c es)` se utiliza para tener algún control sobre la evaluación del argumento de una expresión `Case`, esto es, para restringir el desplegado de las posibles llamadas a contructor `ConsCall`. Veamos el siguiente segmento de código Curry:

```
ieval top funs (Case ctype e ces) =
    case (ieval False funs e) of
      Comb ConsCall f es -> ieval top funs (matchBranch ces f es)
      Lit l -> ieval top funs (matchBranchLit ces l)
```

Buscando tener, por el momento, un intérprete simple, tomamos en cuenta sólo dos posibles casos del resultado de la expresión `Case`, ambos resultados hacen una llamada a la evaluación de la respectiva rama del `Case`, dependiendo del resultado de la evaluación de la expresión `Case` se selecciona la alternativa a seguir evaluando, ya sea la expresión que contiene `matchBranch` o `matchBranchLit`.

Con respecto a la evaluación de funciones decidimos verificar la evaluación de *built-ins* con valores estáticos antes de proceder a desplegar la respectiva función. Si el encabezado de la llamada a función se refiere a los símbolos: "==", "+", "-", "*" o "failed"; el intérprete podría hacer una llamada a: `ieval_EQ`, `ieval_ARITH` o devolver una expresión `"failed"`, en otro caso el intérprete hace una llamada a desplegar, es decir, llama a `ieval top funs (matchiRHS funs (mn,f) es)`. Se puede verificar esto en la siguiente regla:

```
ieval top funs (Comb FuncCall (mn,f) es) =
     if mn == "Prelude"
```

```
then if f == ``failed''
  then Comb FuncCall (mn,f) es
  else if f == ``==''
        then (ieval_EQ top funs (mn,f) es)
        else case f of
                ``+'' -> ieval_ARITH top funs (mn,f)  es
                ``-'' -> ieval_ARITH top funs (mn,f)  es
                ``*'' -> ieval_ARITH top funs (mn,f)  es
else ieval top funs (matchiRHS  funs (mn,f) es)
```

`ieval_EQ` evalúa un caso simple de igualdad estricta, si los argumentos no son literales enteras, ésta trata de evaluar la lista de argumentos. `ieval_ARITH` evalúa funciones aritméticas simples, si los argumentos no son literales enteras, tal como se hizo con `ieval_EQ`; `ieval_ARITH` espera evaluar la lista de argumentos.

`matchiRHS`, primero busca la función a desplegar en el conjunto de la estructura de funciones, esto es, en el *programa*. Cuando el intérprete encuentra la regla correspondiente a la función; devuelve la expresión de la parte derecha de esa regla con la sustitución de todas las ocurrencias de variables que hay en la parte izquierda de la misma regla por la expresión que aplique. Obsérvese las siguientes reglas:

```
matchiRHS [ ] (_,_) _ = Comb FuncCall (``Prelude'',``failed'') [ ]
matchiRHS (Func (_,fname) _ _ _ funrule : fds) (mn,name) es =
   if fname==name then matchRHS_aux funrule es
                  else matchiRHS fds (mn,name) es

matchRHS_aux (Rule vars rhs) es = substitute vars es rhs
```

Para entender un poco que pasa con la sustitución de variables por la expresión correspondiente analice el ejemplo 6.3.

Ejemplo 6.3 *En este ejemplo vamos a describrir cómo despliega* **mixpo**. *Sabemos que nuestros intérpretes despliegan de la misma forma que* **mixpo**. *Vamos a considerar el programa anotado de la figura 6.4. Para iniciar la especialización de cualquier programa anotado,* **mixpo** *busca la función* `main` *por omisión, en éste caso* **mixpo** *construye la siguiente llamada:*

```
[(Comb FuncCall ("fib_ann","main") [(Var 0)])]
```

que está en FlatCurry, y que corresponde a (`main v0`) *en Curry. Ponemos una instrucción* trace *dentro de* **(mixpo)** *para generar una traza y conocer las llamadas a desplegar, las primeras tres son:*

```
unfoldpo: (main v0)
unfoldpo: (MEM (prod v0 (UNF (fib (S (S (S (S (S (Z ))))))))))
unfoldpo: (MEM (prod v0 (Case (S (S (S (S (S (Z )))))) of
   (Z  -> (Z ))
   (S v105 -> (Case v105 of
     (Z  -> (S (Z )))
     (S v106 -> (UNF (sum (UNF (fib (S v106)))
                          (UNF (fib v106)))))
  ))
)))
```

Dada la llamada a desplegar (`main v0`), `matchiRHS`*; primero busca esta función para desplegarla. Podemos ver que la parte derecha de* (`main v0`) *corresponde a la segunda llamada a desplegar de la traza, y la tercera llamada a desplegar corresponde a la misma expresión de la segunda llamada a desplegar pero con la expresión* ((`fib` (`S` (`S` (`S` (`S` `Z`)))))) *ya desplegada. Nótese que en la última llamada a desplegar hay una expresión* `Case` *que corresponde a la parte derecha de la función* `fib`*, en formato flat. Recuerde que la anotación y la especialización son realizadas en este formato. ¿Pero qué pasa con las sustituciones? La primera sustitución sucede cuando* `matchiRHS` *encuentra la estructura de la función* `main`*, después las variables de la parte derecha; indicadas en los índices en la parte izquierda de* `main`*, en este caso* [1]*, son reemplazadas por las expresiones de la llamada a desplegar. En este caso la lista de expresiones de la llamada es* [(`Var 0`)]*, obsérvese la llamada* (`main v0`) *en formato flat. Por lo tanto* `v1` *es reemplazada por* `v0` *y se muestra en el resultado del primer desplegado. La segunda sustitución sucede cuando* ***mixpo*** *requiere el segundo desplegado:*

```
(MEM (prod v0 (UNF (fib (S (S (S (S (S (Z ))))))))))
```

primero `matchiRHS` *trata de desplegar la función* `fib`*, por lo tanto busca la estructura de esta función, un código como:*

```
fib v1 =  case v1 of
   (Z  -> (Z ))
```

```
(S v2 -> (case v2 of
  (Z  -> (S (Z )))
  (S v3 -> (UNF (sum (UNF (fib (S v3))) (UNF (fib v3)))))))
```

después las variables de la parte derecha indicadas en los índices de la izquierda de `fib`*, en este caso* [1]*, son reemplazados por la expresión de la llamada a desplegar. En este caso la lista de expresiones de la llamada es* (`S` (`S` (`S` (`S` (`S` `Z`))))), *por lo tanto* `v1` *se reemplaza por* (`S` (`S` (`S` (`S` (`S` `Z`))))), *resultando un código muy similar al que se muestra en la tercera llamada a desplegar de la traza en éste ejemplo.*

En términos generales ésta es la descripción de la implementación de los intérpretes a especializar.

6.4. Especialización de intérpretes incluyendo *built-ins* y *constraints*

Es posible compilar por medio de evaluación parcial si consideramos la especialización de un intérprete con respecto a un sólo *programa* fijo, produciendo un programa `pobjeto` en el lenguaje de salida del evaluador parcial —*FlatCurry* en este caso ($_{\texttt{fcy}}$). Una virtud de la compilación por evaluación parcial es que siempre genera código objeto correcto [JGS93]. Así pues, si tenemos un *programa* en *FlatCurry* $\texttt{p}_{\texttt{fcy}}$ que acepta una llamada a función `q` con argumentos estáticos y dinámicos, y un programa $\texttt{int}_{\texttt{Cy}}$ escrito en lenguaje Curry para interpretar programas en el lenguaje intermedio *FlatCurry*. Entonces podemos representar un proceso de interpretación particular como:

$$\begin{aligned}
\texttt{salida} &= [\![\texttt{p}_{\texttt{fcy}}]\!]\ \texttt{q} \\
&= [\![\texttt{int}_{\texttt{Cy}}]\!]\ [\texttt{p}_{\texttt{fcy}},\ \texttt{q}] \\
&= [\![\ \underbrace{[\![\texttt{mixpo}_{\texttt{Cy}}]\!]\ [\texttt{int},\ \texttt{p}_{\texttt{fcy}}]}_{\texttt{pobjeto}}\ ]\!]\ \texttt{q} \\
&= [\![\texttt{pobjeto}_{\texttt{fcy}}]\!]\ \texttt{q}
\end{aligned}$$

Si ejecutamos el intérprete $\texttt{int}_{\texttt{Cy}}$ con entradas $\texttt{p}_{\texttt{fcy}}$ y `q`; debe producir el mismo resultado que ejecutar $\texttt{p}_{\texttt{fcy}}$ con la llamada `q`. Nuestro evaluador parcial **mixpo** acepta programas anotados en *FlatCurry* y genera

programas también en *FlatCurry*, de esta forma aplicamos la *primera proyección de Futamura* [JGS93] y la representamos como se muestra a continuación:

$$\texttt{pobjeto}_{\texttt{fcy}} = [\![\ \textbf{mixpo}\ ]\!]\ [\texttt{int},\ \texttt{p}_{\texttt{fcy}}]_{\texttt{fcy}}$$

Donde $\texttt{pobjeto}_{\texttt{fcy}}$ es el resultado de la especialización de $[\texttt{int},\ \texttt{p}_{\texttt{fcy}}]$. En la práctica tenemos dentro del intérprete $\texttt{int}_{\texttt{Cy}}$ una definición $\texttt{main}_{\texttt{i}}$ que es la llamada principal al mismo. Y como argumentos de esta llamada; el *programa* $\texttt{p}_{\texttt{fcy}}$ a ser interpretado y una llamada $\texttt{q}$ a $\texttt{p}_{\texttt{fcy}}$, ambos argumentos están en formato *FlatCurry*. Para aclarar más esta explicación podemos especificar las siguientes:

$$\begin{array}{lcl} \texttt{main}_{\texttt{i}} & = & \texttt{ieval True p}_{\texttt{fcy}}\ \texttt{q} \\ \texttt{p}_{\texttt{fcy}} & \equiv & \text{lista de funciones de } \texttt{p}_{\texttt{fcy}} \text{ en } \textit{FlatCurry} \\ \texttt{q} & \equiv & \text{llamada a } \texttt{main}_{\texttt{p}} \text{ en } \textit{FlatCurry} \\ \texttt{main}_{\texttt{p}} & \equiv & \text{llamada principal a } \texttt{p}_{\texttt{fcy}} \end{array}$$

Podemos ver un ejemplo de intérprete en el apéndice A-1, en la parte derecha de la definición de función `main`, esto es, en la llamada a función `ieval`. En [And92] también usan diferentes estructuras de datos para introducir programas y datos de entrada en la compilación por evaluación parcial.

Analicemos la especialización de un intérprete usando la conjunción (secuencial), esto es, utilizando el tipo *built-in* `&&` en el siguiente *programa* a interpretar.

```
main = (3 == 3) && (1 == 1)
```

La función `main` del intérprete se ve así:

```
main = ieval True [Func (“andexam”,“main”) 0 Public
    (TCons (“Prelude”,“Bool”) [ ])
    (Rule [ ] (Comb FuncCall (“Prelude”,“&&”)
     [Comb FuncCall (“Prelude”,“==”) [Lit (Intc 3),Lit (Intc 3)],
      Comb FuncCall (“Prelude”,“==”) [Lit (Intc 1),Lit (Intc 1)]]
      ))] (Comb FuncCall (“andexam”,"main") [ ])
```

Si solicitamos a PAKCS la evaluación de la expresión incluida en el *programa*, esto es, `(3 == 3) && (1 == 1)`, el resultado es “True”. Para interpretar esta expresión necesitamos hacer algunos cambios, primero la

evaluación de funciones debe ser capaz de procesar el tipo *built-in* `&&`, así pues cambiamos esa parte de código del intérprete:

```
ieval top funs (Comb FuncCall (mn,f) es) =
     if mn == "Prelude"
      then if f == "failed"
        then Comb FuncCall (mn,f) es
        else if f == ‘‘==’’
              then (ieval_EQ top funs (mn,f) es)
              else case f of
                       ‘‘+’’ -> ieval_ARITH top funs (mn,f)  es
                       " -> ieval_ARITH top funs (mn,f)  es
                       "*" -> ieval_ARITH top funs (mn,f)  es
                       "&&" -> ieval_SAND top funs (mn,f)  es
      else ieval top funs (matchiRHS  funs (mn,f) es)
```

Nótese que agregamos la llamada `ieval_SAND` precisamente para evaluar el *built-in* referenciado. Esta última definición de función se muestra abajo:

```
ieval_SAND :: Bool -> [FuncDecl] -> QName -> [Expr] -> Expr
   ieval_SAND    top       fns         (mn,fn)  [e1,e2] =
      case (ieval True fns e1) of
           (Comb ConsCall (Pre,Tru) [])-> (ieval top fns e2)
           (Comb ConsCall (Pre,Fal) []) ->
                                (Comb ConsCall (Pre,Fal) [])
```

Por experiencia hemos visto que una cadena de caracteres se especializa más facilmente si se define una declaración simple. Así pues hemos definido las siguientes declaraciones:

```
Pre = "Prelude"
Fal = "False"
Tru = "True"
```

Y después de especializar el intérprete con los cambios realizados obtenemos el siguiente programa:

```
module int_sand_ann(main,ieval_pe1) where
import FlatCurry
main :: a
main = ieval_pe1
ieval_pe1 :: a
```

```
ieval_pe1 eval rigid
ieval_pe1 = case (Comb ConsCall ("Prelude","True") []) of
        Comb v141 v142 v143 -> case v141 of
          ConsCall -> case v142 of
            (v144,v145) -> case v143 of
              [] -> Comb ConsCall ("Prelude","True") []
```

Podemos verificar el resultado cargando este programa *FlatCurry* a PAKCS y después requerir la evaluación de la función `main`, comprobando que:

```
Comb ConsCall ("Prelude","True") []
```

es el resultado del requerimiento en *FlatCurry* lo cual es equivalente a "True" en Curry y también es el resultado buscado.

Hasta ahora, hemos visto la evaluación parcial de programas simples sin argumentos dinámicos, ahora trataremos con la especialización de intérpretes que contienen el programa de la sucesión de Fibonacci para números naturales (mostrado en la figura 6.2). Tal como en la última evaluación parcial del intérprete debemos cambiar el programa a interpretar. Tal que la función main del intérprete ahora se declara como sigue:

```
main x = ieval True  [Func ("fib7","main") 1 Public (FuncType
(TCons ("fib7","Nat") []) (TCons ("fib7","Nat") [])) (Rule [1]
(Comb FuncCall ("fib7","prod") [Comb FuncCall ("fib7","fib")
[Var 1],Comb FuncCall ("fib7","fib") [Comb ConsCall ("fib7","S")
[Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","S") [Comb
ConsCall ("fib7","S") [Comb ConsCall ("fib7","S") [Comb ConsCall
("fib7","Z") [][][][][][][][][]])),

Func ("fib7","fib") 1 Public (FuncType
(TCons ("fib7","Nat") []) (TCons ("fib7","Nat") [])) (Rule [1]
(Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") []) (Comb
ConsCall ("fib7","Z") []),Branch (Pattern ("fib7","S") [2]) (Case
Rigid (Var 2) [Branch (Pattern ("fib7","Z") []) (Comb ConsCall
("fib7","S") [Comb ConsCall ("fib7","Z") [][]),Branch (Pattern
("fib7","S") [3]) (Comb FuncCall ("fib7","sum") [Comb FuncCall
("fib7","fib") [Comb ConsCall ("fib7","S") [Var 3]],Comb FuncCall
("fib7","fib") [Var 3]])])])),
```

```
 Func ("fib7","prod") 2 Public
(FuncType (TCons ("fib7","Nat") []) (FuncType (TCons ("fib7",
"Nat") []) (TCons ("fib7","Nat") []))) (Rule [1,2] (Case Rigid
(Var 1) [Branch (Pattern ("fib7","Z") []) (Comb ConsCall ("fib7",
"Z") []),Branch (Pattern ("fib7","S") [3]) (Comb FuncCall ("fib7"
,"sum") [Var 2,Comb FuncCall ("fib7","prod") [Var 3,Var 2]])])),

 Func ("fib7","sum") 2 Public (FuncType (TCons ("fib7","Nat") [])
(FuncType (TCons ("fib7","Nat") []) (TCons ("fib7","Nat") [])))
(Rule [1,2] (Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") [])
(Var 2),Branch (Pattern ("fib7","S") [3]) (Comb ConsCall ("fib7",
"S") [Comb FuncCall ("fib7","sum") [Var 3,Var 2]])]))]
(Comb FuncCall ("fib7","main") [x])
```

hemos separado con un salto de párrafo cada definición de función del *programa* a interpretar, i.e. `main`, `fib`, `prod`,`sum` (El programa completo de éste intérprete se encuentra en el apéndice A-2). Podemos ver que la variable `x` de la llamada de función del *programa* a interpretar, representa un valor dinámico del intérprete, esto es, un valor probablemente conocido en tiempo de evaluación parcial. Dicho valor dinámico se propaga a través del intérprete cuando se ejecuta el BTA, entonces nuestro BTA devuelve la siguiente división:

```
[("main",1,[D]),(ïeval",3,[S,S,D]),(ïeval_ARITH",2,[D,D]),
(ïeval_EQ",1,[D]),(ïeval_args",2,[S,D]),("matchBranch",3,[D,D,D]),
("matchBranchLit",2,[D,D]),("matchiRHS",3,[S,D,D]),
("matchRHS_aux",2,[S,D]),("substituteAll",3,[D,D,D]),
(‘‘replaceVar’’,3,[D,D,D]),("substituteAllArgs",3,[D,D,D]),
("substituteAllCases",3,[D,D,D]),("substituteAllCase",3,[D,D,D])]
```

El análisis size-change (SCA) produce como resultado trece multigrafos idempotentes, las siguientes funciones: `ieval_EQ`, `ieval_ARITH` y `matchRHS_aux` e inclusive `main`, no están involucradas en el resultado del SCA, es decir, no tienen multigrafos idempotentes asociados, entonces las llamadas a estas funciones quedan anotadas con `UNF` (durante el proceso de anotación) porqué éstas no introducen bucles infinitos. `matchiRHS` tiene un multigrafo idempotente pero su primer arco, o sea, el correspondiente a su primer argumento, tiene una etiqueta con orden de reducción estricto ($\succ$) y además es estático. Por lo tanto las llamadas a función de

`matchiRHS` también están estiquetadas con `UNF`. El resto de las llamadas a función están anotadas con `MEM`. Por lo tanto, aún sin considerar el vector de generalización, podemos deducir que la especialización de este intérprete puede no ser buena ya que tiene muchas funciones que no serán desplegadas en tiempo de evaluación parcial. Sin embargo, el resultado de la evaluación parcial de nuestros intérpretes que aceptan una llamada con argumentos dinámicos; es una mezcla del intérprete y del programa a interpretar y contiene partes derivadas de ambos [JGS93]. Por ejemplo, después de la especialización del intérprete, la regla de la definición `main` se transforma en la siguiente:

```
main :: b -> a
main v0 = ieval_pe1 (Comb FuncCall ("fib7","main") [v0])
```

Muy buena especialización de la definición `main` del intérprete, pero `matchiRHS` tiene ahora cuatro casos de sustitución indicadas en la parte derecha; las alternativas a desplegar son: `main`, `fib`, `prod` y `sum`. Para entender mejor esto puede ver el apéndice A-3, donde se muestra el *programa* especializado. También hemos obtenido dos casos para la definición `ieval`, es decir, que ha crecido un poco el tamaño del código obtenido. Podríamos esperar que el *programa* especializado se ejecute más rápido que la interpretación original, pero sólo hemos logrado casi el mismo promedio de tiempo de ejecución. La interpretación del programa ciertamente se ha mejorado puesto que el proceso de desplegado ha sido transferido directamente a la función que es reponsable de hacerlo (`matchiRHS` por ejemplo), es decir, se han ahorrado algunos pasos de la interpretación. Aunque en este caso, la especialización no ha reducido suficientemente el código del intérprete ya que la mayoría de las funciones no pueden ser desplegadas a un valor o simplificadas en el mejor de los casos.

6.5. Trabajo relacionado

Considerando únicamente trabajos de evaluación parcial *offline* que han experimentado con la especialización de intérpretes están los siguientes: En [Jør91], Jørgensen genera compiladores a partir de intérpretes por evaluación parcial e implementa intérpretes (en lenguajes funcionales estrictos) a partir de lenguajes formales, él utiliza Similix; un auto-

intérprete para un gran subconjunto de Scheme, el lenguaje objeto es Scheme pero traduce a partir de BAWL[4]. El BTA de Similix es monovariante tal como el nuestro. En [And92], Andersen desarrolló un evaluador parcial para un subconjunto significativo de lenguaje C sin considerar un BTA, él transformó el programa a un lenguaje intermedio llamado core C, de manera análoga nosotros traducimos a *FlatCurry*. En [And92] también fue reportada la primera implementación de evaluador parcial autoaplicable para un lenguaje imperativo. Tempo es un especializador *offline* para programas C [TBC+98] capaz de especializar tanto intérpretes de *bytecode* como de lenguaje estructurado y genera excelentes mejoras en los tiempos de ejecución. En [LCBV04], Leuschel et. al. presentan LOGEN como un auto-intérprete para programas lógicos. En dicho trabajo han logrado la *optimalidad de Jones* [Glü02] en forma sistemática. De manera análoga a [LCBV04] hemos organizado el proceso de evaluación parcial[5] en dos fases, BTA y fase de especialización. Nosotros usamos un algoritmo muy similar para la propia especialización. En la tesis de [Ram07] presenta una aproximación a la compilación por NPE pero los intérpretes se refieren a librerías de un lenguaje de dominio específico más que a un lenguaje de propósito general como aquí se presenta.

6.6. Conclusiones y trabajo futuro

Hemos presentado los primeros experimentos en la especialización de intérpretes usando evaluación parcial *offline* dirigida por *narrowing*. Nuestro proceso de evaluación parcial consta de un BTA monovariante, un análisis de terminación de primer orden llamado análisis *size-change* y el propio proceso de evaluación parcial *offline* puro (**mixpo**). El análisis de tiempo de enlace (BTA, SCA, el anotador) y el especializador están escritos en Curry. Los intérpretes aceptan programas *FlatCurry* y el lenguaje objeto también es *FlatCurry*. Los intérpretes que ejecutan programas fuente sin argumentos dinámicos se especializan bien. Los intérpretes que ejecutan programas con argumentos dinámicos también son

[4]Un lenguaje funcional perezoso fuertemente tipado[Jør92b].

[5]El paradigma de programación lógica introdujo el término "*partial deduction*" para reemplazar el término "*partial evaluation*".

especializados, hasta el momento obtenemos una mezcla del intérprete y el programa a interpretar pero los tiempos de ejecución se mantienen casi en el mismo promedio.

Aún hay mucho trabajo por hacer para mejorar este trabajo, dado que nuestro evaluador parcial tiene funciones de orden superior no es auto-aplicable. Primero necesitamos un BTA polivariante de orden superior (HO) y también implementar un análisis de terminación de HO. Hemos participado en un trabajo reciente para obtener una aproximación a un BTA polivariante de programas de HO por desfuncionalización [ARTV09], dicho trabajo está especificado en el capítulo 5. Tenemos que analizar la factibilidad de utilizar esta transformación conjuntamente con el actual análisis de terminación o bien implementar el SCA extendido para programas funcionales de HO [Ser07].

Capítulo 7

Generación de código CNC a partir de un DSL en Curry

El Control Numérico Computarizado (CNC) es un lenguaje de aplicación industrial para la manufactura de productos. Los programas CNC son series de código que consisten de instrucciones semejantes al lenguaje ensamblador, consecuentemente, son programas de bajo nivel que requieren programadores especializados con el fin de aumentar la productividad en la escritura de tales programas [ARS11].

En este capítulo introducimos un lenguaje de dominio específico (DSL)[1] para la generación de programas CNC. El DSL ha sido desarrollado en Curry, un lenguaje lógico funcional declarativo. Nuestro DSL incluye un conjunto de funciones que encapsulan las instrucciones CNC elevando el nivel de abstracción, y por lo tanto, mejorando la productividad. El DSL está diseñado de tal forma que los usuarios no expertos pueden escribir programas CNC. También se muestra cómo el uso de un DSL permite realizar la captura de requisitos de sistemas CNC y de este modo reducir la brecha entre los requerimientos y la creación del primer prototipo del sistema. A partir de nuestro lenguaje de dominio específico, generamos código CNC muy cercano al de aplicaciones del mundo real. Finalmente, hemos hecho algunas adecuaciones a la librería de funciones CNC, tal como eliminar la capacidad de entrada/salida con el fin de poder especializar dicha librería. Además puesto que la aplicación trata con números

[1]Por sus siglas en ingles: *Domain Specific Language*.

de punto flotante, hemos dado la capacidad de especializar nuestra librería DSL al evaluador parcial, introducido en el capítulo 6, incluyendo además varios *built-ins* que tratan con este tipo de números.

7.1. Introducción

La escritura de programas de aplicación a partir de lenguajes de programación, es una tarea técnica que generalmente es delegada a programadores especializados. Cada lenguaje de programación está fundamentado sobre un conjunto de características técnicas que influyen en el estilo particular de escritura de los programas.

Históricamente, elevar el nivel de abstracción de los lenguajes de programación ha sido un objetivo común en todos los paradigmas. El cambio en el nivel de abstracción permite ocultar los detalles difíciles y engorrosos más cercanos a la máquina, y de este modo posibilita a los programadores no-expertos construir soluciones. Un método simple para elevar el nivel de abstracción de programas, es produciendo interfaces que oculten el bajo-nivel de las instrucciones, agrupándolas en módulos y componiendo las llamadas librerías. Un método más sofisticado consiste en desarrollar un nuevo lenguaje de programación que incluya principios y abstracciones de una forma consistente; que esté inspirado en el dominio de la aplicación.

Los así llamados Lenguajes de Domino Específico (DSLs) [Hud98] proporcionan una poderosa solución para realizar abstracciones de alto nivel. Un DSL incorpora las abstracciones más comunes de un dominio, ofrece combinadores que permiten la construcción de programas y que se utilizan para producir interacciones entre abstracciones.

Por otro lado, cuando las empresas desarrollan nuevos sistemas de software grandes (o medianos), es muy común descubrir que los prototipos no cumplen con los requerimientos solicitados por el cliente en la definición del proyecto. Esto genera una situación crítica que implica tiempo y costes adicionales, lo que podría ser evitado realizando una captura de requerimientos adecuada. De hecho, el diseño de lenguajes de dominio específico se considera una actividad para la captura de requisitos [AW05]. Sin embargo, aunque algunos DSLs son buenos para modelar

las abstracciones del dominio, algunas veces no producen el código final necesario, y por lo tanto sólo son útiles como lenguajes de especificación.

En este capítulo mostramos el desarrollo de un lenguaje de dominio específico para control Numérico Computarizado (CNC). En la actualidad, las máquinas CNC han llegado a ser la base de muchos procesos industriales. Las máquinas CNC incluyen robots, líneas de producción y todas aquellas máquinas que son controladas por dispositivos digitales. Típicamente, las máquinas CNC tienen una *Unidad de Control de Máquina* (MCU) la cual ingresa un programa CNC y controla el comportamiento y los movimientos de todos los componentes de la máquina. Consideramos que elevar el nivel de abstracción del lenguaje CNC nos permitirá:

1. Poder desarrollar programas CNC más amigablemente ya que utilizamos abstracciones de alto nivel

2. Ser más productivos puesto que proponemos una librería de funciones, cada una de las cuales encapsulará muchas instrucciones CNC simples, y

3. Para demostrar que los DSLs son útiles para producir código real y por lo tanto permiten capturar requerimientos ejecutables.

Presentamos nuestro lenguaje como un conjunto de funciones que encapsulan instrucciones CNC, que a su vez pueden generar código CNC y con el fín de producir aplicaciones del mundo real.

Como lenguaje anfitrión de nuestro DSL, proponemos Curry [Han06], un lenguaje lógico funcional multiparadigma. Nuestra elección se basa en el hecho de que Curry es un lenguaje de alto nivel que proporciona un marco de trabajo adecuado para produccir, analizar (formalmente) y verificar los programas. Además, tiene muchas características modernas, tales como: evaluación perezosa (lo que nos permite definir estructuras de datos infinitas), definiciones de orden superior (es decir, el uso de funciones como elementos simples del lenguaje, lo cual permite definir fácilmente combinadores complejos), tipos de datos que permiten construir tipos de datos abstractos, etc. Parte del desarrollo del DSL fue inspirado en [RSV04], donde los autores presentan un DSL aproximado para la especificación de ruteadores también desarrollado en Curry. Algunas ideas de nuestro diseño están basadas en [AOSV04].

7.2. CNC: Un breve repaso

En la actualidad para diseñar programas CNC, los procesos de producción industrial incluyen el uso de aplicaciones de CAD/CAM que permiten modelar piezas en tres dimensiones (3D) y que posteriormente las máquinas CNC tienen que fabricar. Una vez que las piezas han sido modeladas y que satisfacen todos los requisitos de diseño especificados, los modelos 3D son traducidos a programas CNC por la aplicación CAD/CAM. Tal como lo establece el estandar ISO 6983 [fSTCITS82], los programas CNC que pueden ser interpretados por los MCUs, están formados por un código parecido al lenguaje ensamblador y constituidos por instrucciones individuales llamadas *códigos G* (véa un ejemplo en la Figura 7.1).

Debido a la amplia varidedad de funciones y herramientas que las máquinas CNC ofrecen y, en general, a que cada fabricante de máquinas CNC introduce algunas extensiones al estándar de los *códigos G* —uno de los principales problemas de la programación CNC es la falta de portabilidad. Por lo tanto, cuando es necesario reutilizar un programa CNC, los programadores tienen que depurarlo de acuerdo al MCU específico de sus máqinas CNC. Por ejemplo, si tenemos las máquinas HASS VF-0 y DM2016, a pesar de que ambas son máquinas fresadoras, los *códigos G* que éstas aceptan son diferentes puesto que son de diferentes fabricantes [AOSV04] (por ejemplo, la primera es más reciente y es capaz de llevar a cabo una mayor diversidad de tareas). La programación CNC es una tarea compleja ya que los *códigos G* representan un lenguaje de bajo nivel sin declaraciones de control, procedimientos y muchas otras ventajas de los lenguajes modernos de alto nivel. A fin de proporcionar portabilidad a los programas CNC y de elevar el nivel de abstracción del lenguaje, han habido algunas propuestas de lenguajes intermedios, tales como APL [Ott00] y OMAC [MBY+00] a partir de los cuales los *códigos G* pueden ser automáticamente generados con compiladores y pos-procesadores. El control numérico computarizado es el proceso de tener una computadora controlando la operación de una máquina [WWK01].

Un programa CNC es una serie de bloques que contienen una o más instrucciones escritas en un formato parecido al ensambador [Sea94]. Estos bloques son ejecutados en orden secuencial paso a paso. Cada ins-

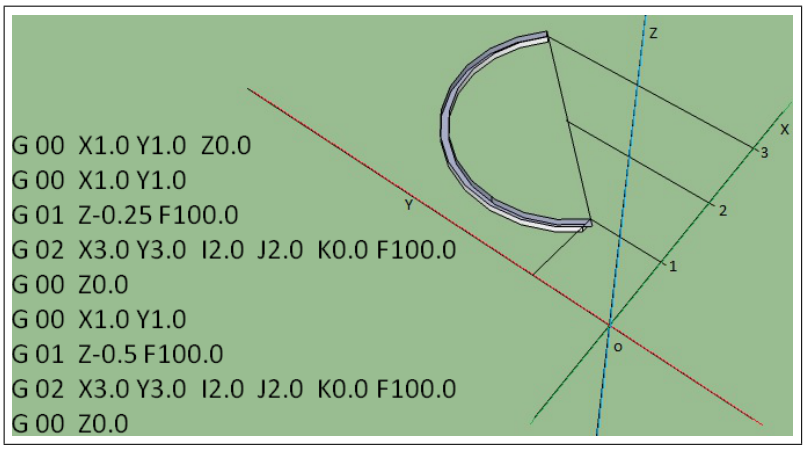


Figura 7.1: Un programa CNC simple.

trucción tiene un significado especial, y a medida que se obtienen; son traducidas en una instrucción específica para la máquina. Por lo general, comienzan con una letra que indica el tipo de la actividad que la máquina está destinada a realizar, tales como F para velocidad de avance de corte, S para velocidad de giro del husillo[2], X, Y y Z para el movimiento de los ejes. Para cualquier tipo de máquina CNC dada, hay cerca de 40 a 50 instrucciones que se pueden utilizar de forma regular.

7.2.1. Códigos G y M

Las instrucciones de los programas CNC se forman con palabras G y M. Las palabras G, comúnmente llamadas Códigos G, son los principales códigos de direccionamiento para las funciones preparatorias, las cuales comprenden movimiento de herramientas y eliminación de material. Entre las mismas se incluyen movimientos rápidos, movimientos de avance lineal y circular, así como ciclos fijos de cajeado, roscado y taladrado. Las palabras M, llamadas también Códigos M, son los principales códigos para funciones misceláneas que indican varias instrucciones sin implicar movimientos dimensionales de la herramienta de corte. Éstas incluyen encendido y apagado de giro del husillo, cambio de herramienta, activación/desactivación de líquido refrigerante, y otras funciones similares.

[2]Es común llamar "husillo" al conjunto del cabezal que porta la herramienta de corte en una fresadora.

La mayoría de los códigos G y M han sido estandarizados, pero algunos de ellos aún tienen diferente significado para controladores específicos. Como se mencionó al inicio, un programa CNC es una serie de bloques, donde cada bloque contiene una o varias instrucciones. Por ejemplo,

```
N0030 G01 X3.0 Y1.7
```

es un bloque con una instrucción, que indica a la máquina hacer un movimiento (de interpolación líneal) en el plano coordenado X Y. La Figura 7.1 muestra un ejemplo de un programa CNC simple para cortar un arco circular por medio de interpolación lineal y circular indicada con *códigos G*: G 01 y G 02.

7.2.2. Un ejemplo

En esta sección, se ilustra un ejemplo de programación CNC. Considérese una máquina fresadora CNC simple que puede mover la torreta en los ejes X, Y y Z. La máquina también maneja la torreta con posicionamiento absoluto e incremental. Un programa CNC para esta máquina consiste de un encabezado y un cuerpo, el encabezado es opcional y es generalmente un breve comentario, mientras que el cuerpo es una lista de bloques, donde cada bloque es identificado por un número (`Nnnnn`). Este número puede ser opcional y puede contener una o más instrucciones o un comentario, donde los comentarios están siempre entre paréntesis. Una instrucción contendrá uno de los códigos CNC que se muestran en la Figura 7.2.

A continuación se muestra un programa CNC para cortar un arco circular de (1,1) a (3,3) con centro en (2,2):

G01:	Representa una interpolación lineal. Mueve el cabezal a lo largo de los ejes XYZ. Puede ser completado por los códigos X, Y, Z.
G02:	Representa una interpolación circular en el sentido de las manecillas del reloj. Mueve el cabezal en un recorrido circular en un plano paralelo a uno de los tres planos principales de referencia.
G90:	indica que se utilizará posicionamiento absoluto.
G91:	indica que se utilizará posicionamiento incremental.
X(-)nn:	se utiliza para indicar movimiento del cabezal a lo largo del eje X.
Y(-)nn:	se utiliza para indicar movimiento del cabezal a lo largo del eje Y.
Z(-)nn:	se utiliza para indicar movimiento del cabezal a lo largo del eje Z.

Donde `nn` es un número cuyo significado depende del tipo de posicionamiento que se está manejando:

- En el caso de posicionamiento *absoluto*, el número indica la nueva posición absoluta en el eje correspondiente, donde (0,0,0) es el punto de referencia sobre la mesa.
- En el caso de posicionamiento *incremental*, el número indica las unidades que está siendo desplazada la herramienta de corte en el eje actual.

Figura 7.2: Conjunto de instrucciones para una máquina fresadora CNC simple.

```
N0010 (corte de un arco circular en el plano XY)
N0020 G90
N0030 G00 X1,0 Y1,0 Z0,0
N0040 G01 Z − 0,25 F50,0
N0050 G02 X3,0 Y3,0 I2,0 J2,0 K0,0 F50,0
N0070 G00 X1,0 Y1,0
N0080 G01 Z − 0,5 F50,0
N0090 G02 X3,0 Y3,0 I2,0 J2,0 K0,0 F50,0
N0095 G00 Z0,0
```

En este ejemplo, el bloque `N0010` denota un comentario; `N0020` instruye a la máquina CNC para que se utilice posición absoluta; `N0030` mueve la torreta a (1,1,0), 0 en el eje Z representa la superficie de la pieza a mecanizar; `N0040` mueve la torreta 0.25 mm bajo la superficie en el eje Z, hace un agujero en (1, 1); `N0050` inicia la interpolación circular desde la última posición a la posición final (3, 3). Las palabras I, J, K definen el centro del arco. F representa el movimiento de avance; `N0060` mueve hacia arriba la torreta; `N0070` mueve la torreta a la posición (1, 1) (el ini-

cio del arco); `N0080` mueve hacia abajo la torreta, esta vez 0.5 milímetros hacia abajo en el eje Z; `N0090` inicia nuevamente el arco circular 0.5 mm debajo del eje Z; finalmente `N0095` coloca la torreta en Z 0.0. De esta forma tenemos un arco circular cortado a 0.5 mm de profundidad (véase la Figura 7.1).

7.3. Un DSL para programación CNC incrustado en Curry

En esta sección presentamos un lenguaje de dominio específico para la generación de código CNC que se basa en la norma ISO 6983 [fSTCITS82]. Nos centramos sobre todo en aspectos de diseño (teniendo en cuenta que el DSL se desarrolla en Curry), explicando las principales funciones desarrolladas y mostrando ejemplos para ilustrar la aplicabilidad de nuestro DSL.

7.3.1. Introducción a Curry

Curry es un lenguaje de programación universal que amalgama los más importantes paradigmas de programación declarativa [Han94], esto es, progamación funcional y programación lógica. Curry combina de manera incondicional características de la programación funcional (expresiones anidadas, evaluación perezosa, funciones de orden superior), de la programación lógica (variables lógicas, estructuras de datos parciales) y de la programación concurrente (evaluación concurrente de constantes con sincronización de variables lógicas). Además Curry provee de características adicionales en comparación a los lenguajes puros (comparando con la programación funcional ofrece: búsqueda, cómputo con información parcial. Comparando a la programación lógica: cómputo más eficiente debido a la evaluación determinista de funciones) [Han06].

Un programa Curry especifica la semántica de expresiones, donde las metas u objetivos, tal como ocurren en programación lógica, son expresiones particulares. La ejecución de un programa Curry significa simplificar una expresión hasta que un valor o solución sea calculada. Para distinguir entre valores y expresiones reducibles, Curry tiene una dis-

tinción estricta entre constructores (de datos) y operaciones o funciones definidas sobre esos datos. Por lo tanto, un programa Curry consiste de un conjunto de declaraciones de tipos y funciones. Las declaraciones de tipos definen el dominio computacional (constructores) y la declaración de funciones las operaciones sobre esos dominios. Los predicados en el sentido de la programación lógica pueden ser considerados como restricciones, esto es, funciones con tipo `Success` como resultado. Los lenguajes funcionales modernos como Haskell [PJ03], SML [MTH90] permiten la detección de muchos errores de programación en tiempo de compilación debido al uso de sistemas de tipos polimórficos. En los lenguajes lógicos modernos se utilizan sistemas de tipos similares (Gödel [HL94], λProlog [NM88]). Curry sigue el mismo enfoque, es decir, un lenguje fuertemente tipado con un sistema de tipos polimórfico tipo Hindley/Milner. Cada objeto en un programa tiene un tipo único, donde los tipos de las variables y de las operaciones pueden ser omitidos y reconstruidos por un mecanismo de inferencia de tipos. Se recomienda ver la sección 1.3 donde se muestran algunas de las características sintácticas de Curry.

7.3.2. Uso de Curry como lenguaje anfitrión del DSL

La estructura de datos especificada en Curry para soportar un programa CNC se muestra en la Figura 7.3.

En nuestra definición, un programa CNC está formado por un encabezado (`Header`) y un cuerpo (`Body`) [Arr04]. Un `Header` es un comentario —un `comentario` es un cadena de caracteres—, en caso de que éste no se ponga, se utiliza un constructor `Nothing`. Un `Body` es un conjunto de bloques, y cada bloque se define con un `Command` o por un conjunto de instrucciones. Finalmente una `Instruction` se especifica como una o varias de las posibles palabras definidas en el estándar ISO 6983 [fSTCITS82].

Por ejemplo, podemos invocar la función `DrawHole` de la siguiente forma:

```
DrawHole (1.0,1.0,0.0,1.5,10.0,0.5)
```

Este comando perfora un agujero en las coordenadas (1,1,0) con 1.5 unidades de profundidad y una herramienda de corte de 0.5 unidades de

```
data CNCprogram = Header Body
data Header  = Maybe Comment
type Comment = String
data Maybe a = Nothing | Just a
type Body    = [Command]
type Command = [Instruction]
data Instruction = N Int | G String | X Float | Y Float | Z Float
| U Float | V Float | W Float | P Float | Q Float | R Float | A Float
| B Float | C Float | I Float | J Float | K Float | F Float | S Float
| T Float | M String
```

Figura 7.3: Estructura de datos Curry para representar un programa CNC.

ancho, ésta función produce el siguiente código CNC:

$[(\texttt{G}\ \text{"}\texttt{00}\text{"}), (\texttt{X}\ 1{,}0), (\texttt{Y}\ 1{,}0), (\texttt{Z}\ 0{,}0)]$
$[(\texttt{G}\ \text{"}\texttt{01}\text{"}), (\texttt{Z}\ (-0{,}25)), (\texttt{F}\ 10{,}0)]$
$[(\texttt{G}\ \text{"}\texttt{01}\text{"}), (\texttt{Z}\ (-0{,}5)), (\texttt{F}\ 10{,}0)]$
$[(\texttt{G}\ \text{"}\texttt{01}\text{"}), (\texttt{Z}\ (-0{,}75)), (\texttt{F}\ 10{,}0)]$
$[(\texttt{G}\ \text{"}\texttt{01}\text{"}), (\texttt{Z}\ (-1{,}0)), (\texttt{F}\ 10{,}0)]$
$[(\texttt{G}\ \text{"}\texttt{01}\text{"}), (\texttt{Z}\ (-1{,}25)), (\texttt{F}\ 10{,}0)]$
$[(\texttt{G}\ \text{"}\texttt{01}\text{"}), (\texttt{Z}\ (-1{,}5)), (\texttt{F}\ 10{,}0)]$

Cuando el MCU interpreta el código generado; mueve la torreta a la coordenada (1,1,0), enseguida hace un agujero con una interpolación lineal de -0.25 unidades en el eje Z a una velocidad de avance de 10.0 unidades por minuto. En este caso prácticamente hace seis agujeros en la misma posición ya que el ancho de la herramienta es menor que la profundidad de la perforación.

Analizando la función `DrawHole` (véase la Figura 7.4), ésta inicia con el procedimiento *InitPOS*, el cual llama a la función *writefile* que a su vez toma a la función *GotoXYZ* como parámetro. Puesto que Curry es un lenguaje de orden superior; acepta sin mayor trámite las funciones como argumentos. *GotoXYZ* produce el primer comando CNC. Después ejecuta el procedimiento *Perforate*, ya que el ancho de la herramien-

```
{-       Auxiliar Function        -}
 FAUX_DrawHole :: (Float, Float, Float, Float) -> IO()
 FAUX_DrawHole(Height, Feedrate, halfdx, control) = do Draw
 where
  Draw =
   if ((Height >= control)&&((Height-.control) >= halfdx))
       ||(Height==control)
      then do
         writefile(DrawZ(control,Feedrate))
         FAUX_DrawHole(Height,Feedrate,halfdx,(control+.halfdx))
      else if ((Height >= control)&&((Height-.control) < halfdx))
              then do
               writefile(DrawZ(control,Feedrate))
               FAUX_DrawHole(Height,Feedrate,halfdx,
                             (control+.(Height-.control)))
              else done
 - Main Function: Drilling function acts in X,Y
 - Specify X,Y,Z  coordinates and depth (Height)
 - Cutting speed is Feedrate and width tool is dx
 DrawHole :: (Float,Float,Float,Float,Float,Float) -> IO()
 DrawHole (InitX,InitY,InitZ,Height,Feedrate,dx) =
         do InitPOS
            Perforate
 where
  InitPOS = writefile(GotoXYZ(InitX,InitY,InitZ))
  Perforate = if (Height<dx)
               then do
                    writefile(DrawZ(Height,Feedrate))
              - width of tool is less than the depth
               else do
                FAUX_DrawHole(Height,Feedrate,(0.5*.dx),(0.5*.dx))
```

Figura 7.4: Ejemplo de la función DSL `DrawHole`.

```
- erasefile cleans CNC file
erasefile :: IO()
erasefile = writeFile "CNCCapture.txt" "%\n"
- Go to X, Y, Z
GotoXYZ :: (Float, Float, Float) -> Command GotoXYZ
GotoXYZ(x,y,z) = [G "00", X x, Y y, Z z]
- Drill Z with a feedrate
DrawZ :: (Float, Float) -> Command
DrawZ (z,feedrate) = [G "01", Z (0-.z), F feedrate]
- writefile writes CNC commands in a text file
writefile :: Command -> IO()
writefile(NewCommand) =
             appendFile "CNCCapture.txt" (show(NewCommand)++"\n")
```

Figura 7.5: `writefile` y otras funciones auxiliares.

ta es menor que la profundidad de la perforación; `DrawHole` llama a *FAUX_DrawHole* para producir el resto de los comandos CNC hasta que se logra la profundidad solicitada (puede encontrar más detalles en la URL http://www.ciidet.edu.mx/Sitio_DSL/CNC_DSL.htm). Vale la pena señalar que la función *writefile* pega un comando CNC en un archivo cada vez que ésta se ejecuta. Algunas funciones auxiliares se muestran en la Figura 7.5.

7.3.3. Funciones del DSL

En esta sección se especifican las definiciones de función de nuestro DSL y se muestra el código CNC que genera, a partir de una lista de parámetros simples. La intención de dichas funciones es simplificar la manera de realizar la programación para manufacturar una pieza. En general todas las partes implicadas en un proceso de fabricación se pueden formar a partir de la combinación de figuras geométricas simples, realizadas a partir de funciones tales como interpolaciones lineales, circulares y parabólicas.

A partir de esta idea, se está diseñando un DSL cuyas funciones generan código CNC basado en el estándar ISO 6983 [fSTCITS82]. Estas funciones DSL producen: cortes lineales, circulares, figuras geométricas simples (rectángulos, círculos), así como ciclos fijos de cajeado circular y

rectangular. Los parámetros que se solicitan en todas las funciones son características relacionadas con el tamaño y posición de la figura, por lo que el usuario deberá realizar un bosquejo adicional en donde diseñe la pieza que desea maquinar y la exprese por medio de figuras geométricas; cuyos parámetros se utilizarán en el DSL, en una secuencia ordenada para generar la pieza deseada. Otros parámetros que se solicitan en las funciones DSL son valores relacionados con la máquina CNC en particular, como son velocidad de corte y diámetro de la herramienta de corte. A continuación se describen algunas funciones representativas con sus parámetros.

Funciones de configuración

Estas funciones se usan para especificar los parámetros generales a utilizar durante el proceso de fabricación de una pieza. Por ejemplo, pueden indicar que el posicionamiento es absoluto o relativo, o bien que se requieren medidas en milímetros o pulgadas y así sucesivamente. Algunas definiciones en lenguaje Curry se muestran en el siguiente cuadro.

```
—Selecciona el tipo de movimiento
Movement :: (String "Absolute" | "Relative") –> IO()
—Seleccion de el tipo de compensación
Compensation :: (String "Center" | "Left" | "Right") –> IO()
—Selección de la unidad de medida
Unit :: (String "Millimeters" | "Inches") –> IO()
```

Función para hacer un agujero en un punto determinado

```
DrawHole :: (Float,Float,Float,Float,Float,Float) –> IO()

DrawHole (InitX,InitY,InitZ,Height,Feedrate,dx)
```

donde *InitX, InitY, InitZ* representan la posición inicial de la herramienta de la máquina CNC, *Height* especifica la profundidad de la perforación, *Feedrate* es la velocidad promedio de avance de la herramienta y *dx* representa el diámetro de la herramienta de corte.

Función para corte lineal

```
DrawLine :: (Float, Float, Float, Float, Float, Float, Float, Float) -> IO()

DrawLine (InitX, InitY, InitZ, EndX, EndY, Height, Feedrate, dx)
```

donde *InitX, InitY, InitZ* indican la posición inicial de la herramienta de la máquina CNC, *EndX, EndY* representa la posición final del corte, *Height* representa la profundidad del corte, *Feedrate* representa la velocidad de avance de la herramienta de la máquina CNC y *dx* representa el diámetro de la herramienta de corte.

Función para hacer un arco de círculo

```
DrawArc :: (Float, Float, Float, Float, Float, Float, Float, Float, Float, String, Float,

            Float) -> IO

DrawArc (Initx, Inity, Initz, Endx, Endy, i, j, k, Height, TSpin, Feedrate, dx)
```

donde *InitX, InitY, InitZ* indican la posición inicial de la herramienta de la máquina CNC, *Endx, Endy* representa la coordenada final del arco en el plano X,Y; *i, j, k* representa las coordenadas del centro del círculo, *Height* representa la profundidad del corte, *TSpin* representa el sentido del giro de la interpolación circular, *Feedrate* representa la velocidad de avance de la herramienta de la máquina CNC y *dx* representa el diámetro de la herramienta de corte.

Función para realizar un cajeado rectangular

```
DrawBox :: (Float, Float, Float,Float, Float, Float, Float,Float) -> IO()

DrawBox (Initx, Inity, Initz, Length, Width, Height, Feed, dx)
```

donde *InitX, InitY, InitZ* indican la posición inicial de la herramienta de la máquina CNC, *Length* representa la longitud de la caja, *Width* representa el ancho de la caja, *Height* representa la profundidad del corte, *Feedrate* representa la velocidad promedio de avance de la herramienta de corte y *dx* representa el diámetro de la herramienta de corte.

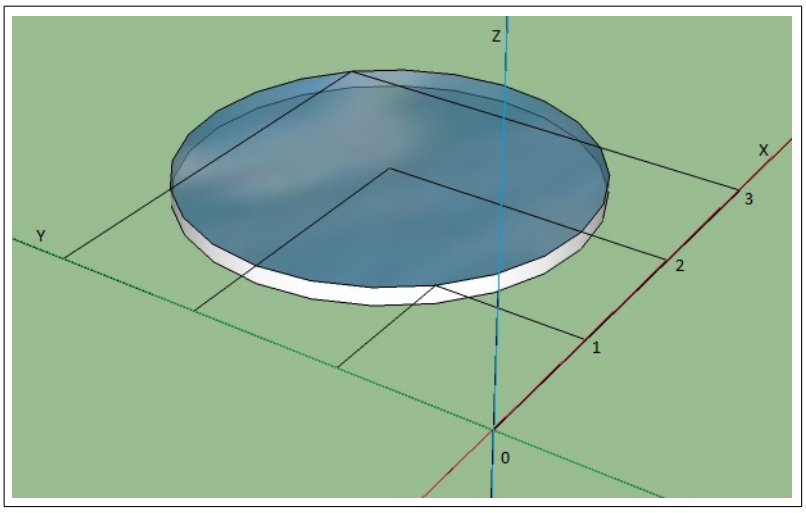


Figura 7.6: Cajeado circular.

Función para realizar un cajeado circular

DrawCylinder :: (Float, Float, Float, Float, Float, Float, Float)->IO()

DrawCylinder (i, j, k, Height, Radius, Feedrate, dx)

deonde *i, j, k* indican la coordenada del centro del círculo, *Height* representa la profundidad del corte, *Radius* representa el radio del círculo, *Feedrate* representa la velocidad promedio de avance de la herramienta de corte y *dx* representa el diámetro de la herramienta de corte.

7.3.4. Uso de las funciones DSL

La siguiente función DSL:

```
DrawCylinder (2.0,2.0,0.0,0.25,1.0,0.5)
```

realiza un ciclo de cajeado circular (véa la Figura 7.6). Ésta define el centro de un sector circular en (2,2,0) con 0.25 unidades de profundidad, 1.0 unidades de radio y una herramienta de corte de 0.5 unidades de ancho. Esta función DSL específica genera el siguiente código CNC:

```
[(G "00"),(X 2.0),(Y 2.0),(Z 0.0)]
[(G "01"),(Z (-0.25)),(F 40.0)]
[(G "01"),(X 2.5),(F 120.0)]
```

```
[(G "03"),(X 2.5),(I 2.0),(J 2.0),(K 0.0),(F 120.0)]
[(G "01"),(X 3.0),(F 120.0)]
[(G "03"),(X 3.0),(I 2.0),(J 2.0),(K 0.0),(F 120.0)]
[(G "00"),(X 2.0),(Y 2.0),(Z 0.0)]
```

`G 03` realiza una interpolación circular en sentido de las manecillas del reloj, y ya que la herramienta de corte es de 0.5 unidades de diámetro; con un agujero en el centro y sólo dos comandos `G 03` son suficientes para hacer el ciclo cajeado circular.

Otro ejemplo se muestra en la Figura 7.1, donde el código CNC para producir un arco circular es generado con una sola llamada a la siguiente función DSL:

```
DrawArc (1.0,1.0,1.0,3.0,3.0,2.0,2.0,0.0,0.5,"CW",100.0,0.5)
```

Ésta genera el siguiente código CNC:

```
[(G "00"),(X 1.0),(Y 1.0),(Z 1.0)]
[(G "00"),(X 1.0),(Y 1.0)]
[(G "01"),(Z (-0.25)),(F 100.0)]
[(G "02"),(X 3.0),(Y 3.0),(I 2.0),(J 2.0),(K 0.0),(F 100.0)]
[(G "00"),(Z 1.0)]
[(G "00"),(X 1.0),(Y 1.0)]
[(G "01"),(Z (-0.5)),(F 100.0)]
[(G "02"),(X 3.0),(Y 3.0),(I 2.0),(J 2.0),(K 0.0),(F 100.0)]
[(G "00"),(Z 1.0)]
```

Claramente, el DSL mejora la legibilidad y aumenta el nivel de abstracción de los programas, además reduce el tamaño del código. En éste ejemplo utilizando el DSL nos permite ejecutar todo el trabajo con una sola instrucción DSL. La Figura 7.7 muestra el código Curry que implementa la función DSL `DrawArc`. Obsérvese que una ventaja importante del DSL es que podemos re-implementar todas las funciones para otra máquina CNC específica, manteniendo inclusive, la misma signatura, es decir, la forma de describir las definiciones del prototipo. Esto significa que el DSL ofrece portabilidad que nos permite usar los mismos programas DSL en diferentes máquinas CNC.

```
{- DrawArc draws an arc where i,j,k are the origin of the radio -}
{- TSpin:  CW=ClockWise CCW=CounterClockWise  -}
DrawArc :: (Float, Float, Float, Float, Float,Float,Float,Float,
            Float, String, Float, Float) -> IO()
DrawArc (Initx, Inity, Initz, Endx, Endy, i, j, k, Height, TSpin,
          Feedrate, dx) = do InitPOS
                             Perforate
 where
  InitPOS = writefile(GotoXYZ(Initx, Inity, Initz))
  Perforate =
    if (Height<dx) then do
      writefile(DrawZ(Height,Feedrate))
      writefile([Spin(TSpin),X Endx, Y Endy, I i, J j, K k,
                 F Feedrate])
{- if tool width is less than depth -}
       else
        do FAUX_DrawArc(Initx, Inity, Initz, Endx, Endy, i, j, k,
                     Height, TSpin, Feedrate,(0.5*.dx),(0.5*.dx))
```

Figura 7.7: Código Curry de la función DSL `DrawArc`.

7.4. Especialización del DSL

Puesto que nuestro evaluador parcial especializa únicamente programas de primer orden, y no tiene la capacidad de tratar con el sistema monádico de entrada/salida de Curry [Han06]. Para especializar la librería fue necesario hacer varias modificaciones al DSL original. Una de las principales definiciones del DSL es *writefile*, ya hemos dicho que pega un comando CNC a un fichero cada vez que ésta se ejecuta, es decir, es básicamente una función de entrada/salida que envía un comando DSL a un archivo. Ahora *writefile* se acompaña de una lista de comandos CNC —un argumento llamado *cncp*— y pega un nuevo comando CNC a dicha lista de comandos. Vemos en la siguiente definición que *writefile* acepta como argumentos dos listas polimórficas y genera una lista del mismo tipo:

```
writefile :: [a] -> [a] -> [a]
writefile NewCommand cncp = cncp ++ NewCommand
```

Originalmente las funciones como: *DrawArc*, *DrawHole*, *DrawLine*, etc., generaban un tipo `IO()`, al modificar la librería para evaluarla par-

```
 {- DrawArc draws an arc where i,j,k are the origin of the radio -}
 {- TSpin:  CW=ClockWise CCW=CounterClockWise  -}
DrawArc (Initx, Inity, Initz, Endx, Endy, i, j, k, Height, TSpin,
         Feedrate, dx) cncp =
    if (Height<dx)
       then  {- if tool width is greater than depth -}
           writefile([Spin(TSpin),X Endx, Y Endy, I i, J j, K k,
                      F Feedrate]) Dz
       else  {- if tool width is less than depth -}
          FAUX_DrawArc(Initx,Inity,Initz,Endx,Endy,i,j,k,Height,
                       TSpin,Feedrate,(0.5*.dx),(0.5*.dx)) cncp
 where
  InitPOS = writefile(GotoXYZ(Initx, Inity, Initz)) cncp
  Dz  = writefile(DrawZ(Height,Feedrate)) InitPOS
```

Figura 7.8: Código Curry de la función DSL `DrawArc`.

cialmente ahora generan un tipo `[Instruction]`. Si verificamos la definición de la estructura de un programa CNC en la Figura 7.3 un tipo `Command` equivale a una lista `[Instruction]`, así pues el DSL genera un conjunto de `[Instruction]` en lugar de un fichero del mismo tipo. Podemos comparar el código de la función *DrawArc* antes y después de la modificación de la librería, véa las Figuras 7.7 y 7.8.

Una de las principales diferencias es que hemos agregado el argumento *cncp* a la función *DrawArc*, inicialmente se trata de una lista vacía y corresponde a la lista de `[Instruction]` que finalmente será el programa CNC generado. También se puede observar que hemos eliminado la secuencia de operaciones de entrada/salida provistas con la notación `do` en la Figura 7.7, es decir `InitPOS` y `Perforate`, prácticamente hemos reemplazado la parte derecha de *DrawArc* con la parte derecha de la definición original de `Perforate`. Si traducimos la función *DrawArc* a *FlatCurry* notaremos que el código de la Figura 7.7 es más complejo que el correspondiente de la Figura 7.8, ambos generan una expresión `Let [(VarIndex,Expr)] Expr`, sin embargo la lista `[(VarIndex,Expr)]` de la función original es más compleja que la correspondientes de la Figura 7.8 incluyendo la función de entrada/salida lo que nos imposibilita para especializar dicha función. Hemos hecho modificaciones muy similares con el resto de las funciones DSL.

Con respecto a los cambios realizados al evaluador parcial para soportar programas con operadores aritméticos que traten con números de punto flotante, se realizaron algunos arreglos para evaluar los siguientes *built-ins*: "*.", "+.", "-.", ">.", "<=.". El evaluador parcial soporta sólo algunos casos con este tipo de operadores.

Al especializar la librería incluyendo al inicio de la misma una definición *main* con una llamada como la siguiente:

```
main =DrawArc(1.0,1.0,1.0,3.0,3.0,2.0,2.0,0.0, 0.5,"CW", 5.0, 0.5) []
```

El evaluador parcial genera el siguiente programa:

```
module
CNCDSLcc_ann(Comment,Body,Command,CNCprogram(Header),Header(Maybe),
 Maybe(Nothing,Just),Instruction(N,G,X,Y,Z,U,V,W,P,Q,R,A,B,C,I,J,K,
 F,S,T,M),main,FAUX_DrawArc_pe1)
where
import CLPR
type Comment = String
type Body = [[Instruction]]
type Command = [Instruction]
data CNCprogram = Header [[Instruction]]
data Header = Maybe String
data Maybe a = Nothing | Just a
data Instruction = N Int | G String | X Float | Y Float | Z Float | U
Float | V Float | W Float | P Float | Q Float | R Float | A Float | B
Float | C Float | I Float | J Float | K Float | F Float | S Float | T
Float | M String
main :: a
main = FAUX_DrawArc_pe1
FAUX_DrawArc_pe1 :: a
FAUX_DrawArc_pe1 = [G "00",X 1.0,Y 1.0,G "01",Z (-0.25),F 5.0,G "02",
X 3.0,Y 3.0,I 2.0,J 2.0,K 0.0,F 5.0,G "00",Z 1.0,G "00",X 1.0,Y 1.0,
G "01",Z (-0.5),F 5.0,G "02",X 3.0,Y 3.0,I 2.0,J 2.0,K 0.0,F 5.0,
G "00",Z 1.0]
{- end of module /home/garroyo/2rev/CNCDSLcc_ann_pe  -}
```

Donde podemos ver que el evaluador parcial ha especializado completamente la llamada de la función *main*, si separamos los códigos G de la lista que se muestra en la parte derecha de `FAUX_DrawArc_pe1` tenemos el siguiente programa:

```
[G "00",X 1.0,Y 1.0,
G "01",Z (-0.25),F 5.0,
G "02",X 3.0,Y 3.0,I 2.0,J 2.0,K 0.0,F 5.0,
G "00",Z 1.0,
G "00",X 1.0,Y 1.0,
G "01",Z (-0.5),F 5.0,
G "02",X 3.0,Y 3.0,I 2.0,J 2.0,K 0.0,F 5.0,
G "00",Z 1.0]
```

El cual es muy similar al código generado por la función DSL *DrawArc* original mostrado en la sección anterior, lo que demuestra la funcionalidad de nuestro DSL y la utilidad del proceso de evaluación parcial.

7.5. Conclusiones

En este capítulo hemos introducido un DSL para el diseño de programas CNC, utilizando Curry como lenguaje anfitrión. Hemos mostrado que el uso del DSL ofrece importantes ventajas sobre la programación CNC pura. Además hemos mostrado la posibilidad de ser especializada nuestra aplicación DSL incluyendo la capacidad de evaluar algunas operaciones de punto flotante.

El DSL provee dos principales ventajas en el desarrollo de programas CNC. Primero, ofrecemos la posibilidad de programar con un alto nivel de abstracción, es decir, con menos instrucciones y mayor legibilidad logramos proporcionar más comandos propios a la máquina CNC (más código CNC), con lo cual se mejora la productividad. Este objetivo ha sido alcanzado ya que las funciones DSL encapsulan procedimientos que automáticamente producen código CNC para llevar a cabo una tarea específica y compleja.

Segundo, hemos reducido la brecha entre los requerimientos del usuario y el desarrollo de los prototipos CNC. Una situación muy común en el campo de la ingeniería del software, ya que el analista que recopila los requerimientos del dominio de un problema y el programador que desarrolla el prototipo; comúnmente son personas diferentes. El uso de los DSLs durante el análisis permite al analista producir prototipos preliminares sin la necesidad de ser un especialista en programación CNC. Esto puede llevarse a cabo gracias al alto nivel de abstracción del los DSLs que permiten a los requerimientos llegar a ser los llamados *requerimien-*

tos ejecutables. Una ventaja secundaria es el dar una aplicación a nuestro evaluador parcial al adecuar tanto el especializador como la aplicación DSL, para habilitar la especialización de la librería DSL para diseño de programas CNC con respecto a una llamada a dicha librería.

Los experimentos preliminares son alentadores y señalan la utilidad de nuestro enfoque. Sin embargo aún hay mucho trabajo por hacer, tal como aumentar la librería con otras funciones para realizar figuras geométricas, definir funciones para otras máquinas CNC (tornos, fresadoras, etc), definir un entorno gráfico para simplificar la tarea de diseño de los programas CNC. Algunos otros ejemplos y el código fuente de nuestra librería DSL está públicamente disponible en la siguiente URL:

http://www.ciidet.edu.mx/Sitio_DSL/CNC_DSL.htm

Capítulo 8

Conclusiones y trabajo futuro

A continuación presentamos las conclusiones, incluyendo un resumen de las principales contribuciones de la tesis e indicando algunas posibles líneas de trabajo futuro.

8.1. Conclusiones

Hemos presentado una significativa mejora en la especialización de programas, con respecto a la primera aproximación *offline* de evaluación parcial dirigida por *narrowing* [RSV05a], para lo cual implementamos dos procedimientos de anotación automáticos. Éstos últimos incluyen un BTA estándar monovariante, y una aproximación del principio de terminación de los grafos *size-change* adaptado del paradigma funcional al lógico funcional, con lo que se obtiene un principio de cuasi-terminación. Puesto que son dos criterios diferentes de anotación, fue necesario implementar dos especializadores, de lo que resultan un evaluador parcial que denominamos *offline puro* y otro *híbrido*.

Comparando las dos estrategias *offline*, mencionadas en el parrafo anterior, con el evaluador parcial *online* de [AHV02], hemos corroborado que el esquema de evaluación parcial *online* es más preciso, aunque es relativamente más lento en el proceso de especialización, que los esquemas de evaluación parcial *offline*. Asimismo, hemos implementado algunas aplicaciones utilizando el evaluador parcial *offline* puro. Brevemente, las principales contribuciones de la tesis son:

1 Fundamentos para garantizar la terminación de la EP *offline*.
Hemos establecido las bases para realizar la primera etapa de un evaluador parcial *offline* de una manera clara y precisa tal como se dicta en la literatura, esto es, una primera etapa que consta de: un BTA monovariante, completamente automático y, como consecuencia de la adecuación del principio de terminación de los grafos *size change* —del paradigma funcional al paradigma lógico funcional— la formulación de un principio de terminación de la evaluación parcial, lo que garantiza la terminación del proceso de especialización. El resultado del BTA y del análisis *size change*, mejora la precisión del proceso de anotación y por ende tiende a perfeccionar la segunda etapa de la evaluación parcial, esto es, de la propia especialización.

2 Aproximaciones de anotación automática.
Los principios teóricos indicados en la contribución (1), expresamente en la primera etapa del evaluador parcial *offline*, derivan en una declaración de evaluación parcial terminante. El procedimiento de anotación implementado está basado en hacer terminantes los programas que violen dicha declaración. En un principio, se agregaron sólo anotaciones de generalización, tal como la anotación de los SRTs *no-crecientes* de [Ram07]. En una segunda implementación de aproximación de anotación, se agregaron dos tipos de anotación más, esto es, que nuestro principio de terminación es capaz de inferir cuales declaraciones de función son susceptibles de ser completamente desplegadas (lo que nos lleva a obtener un control local *offline* puro), cuales no y cuales términos deben ser generalizados para que no violen la declaración de evaluación parcial terminante. Todo lo anterior desde la primera etapa de evaluación parcial, lo que contrasta con la primera aproximación de evaluación parcial *offline* de [Ram07].

3 Aproximación de NPE *offline* con programas de orden superior.
Hemos llevado a cabo una aproximación al primer evaluador parcial *offline* dirigido por *narrowing* que trata con programas de orden superior y que produce especializaciones polivariantes. En términos generales hemos aplicado un algoritmo de desfuncionalización

para hacer explícita la información de orden superior del programa, después con el programa desfuncionalizado generamos una copia de cada definición de función del programa que posea valores de tiempo de enlace diferentes, con lo que obtenemos un programa politransformado de primer orden. Luego al aplicar nuestro BTA monovariante sobre este programa obtenemos una aproximación a un resultado que generaría un BTA polivariante sobre el programa original (de orden superior). Posteriormente se realiza el análisis *size-change* sobre el mismo programa doblemente transformado y con ambos resultados tanto del BTA como del SCA se efectúa la anotación. Finalmente, con el programa anotado se lleva a cabo el proceso de especialización. Los resultados de nuestro prototipo muestran un proceso de evaluación parcial eficiente y bastante preciso.

4 Aproximación a la compilación por EP de programas LF.
Se definió una sencilla semántica operacional para un subconjunto de *FlatCurry*, con el objeto de poder especificar algunos intérpretes que sigan dicha semántica. También se extendió la implementación del evaluador parcial *offline* puro dando la capacidad para que procese angunos *built-ins* y *constraints*, y a su vez, para que acepte especializar un conjunto mayor del lenguaje *FlatCurry*. Con esa definición, es posible realizar algunos intérpretes para ese subconjunto del lenguaje intermedio, e inspirados en el propio proceso de especialización, consecuentemente, llevar a cabo una aproximación a la autoaplicación, es decir, realizar la evaluación parcial del tales intérpretes dando como segundo parámetro un programa Curry de primer orden. Con todo lo anterior se obtuvo la primera aproximación a la compilación por evaluación parcial *offline* de programas lógico funcionales, de acuerdo a la primera proyección de Futamura.

5 Especialización de un DSL.
Otra aplicación que hemos realizado es la especialización de un DSL que genera programas para una máquina de Control Numérico Computarizado. Puesto que la aplicación maneja preponderantemente números de punto flotante, hemos considerado varios *built-*

ins en nuestro evaluador parcial para que soporte especializar algunos programas generados por este DSL. La librería que compone el DSL original, se adaptó a primer orden para hacer posible su especialización. Hemos definido una llamada a alguna de las funciones dentro de la librería para llevar a cabo la evaluación parcial y nuestro EP se encarga de especializar el DSL de acuerdo a la llamada definida.

8.2. Trabajo futuro

Respecto a las líneas de trabajo futuro que podríamos mencionar para dar continuidad a la presente investigación, tenemos las siguientes:

1. Definir un BTA polivariante.
   Es posible mejorar el grado de especialización de un programa si asociamos más de una secuencia de valores de tiempo de enlace a los argumentos de una función, con lo que tedríamos más de una división de valores de tiempo de enlace. Hemos presentado una aproximación a un BTA polivariante; desfuncionalizando un programa de orden superior, y luego aplicando nuestro BTA monovariante al programa desfuncionalizado, presentando una mejora en la rapidez de ejecución de la mayoría de los ejemplos de primer orden, es decir, una mejora en la especialización con programas de primer orden. Obtuvimos un resultado más conservador con programas de orden superior. Realizar un BTA polivariante eliminaría un problema indeseable como lo es el incremento del tamaño del código que se obtiene con el tratamiento del proceso de desfuncionalización.

2. Mejorar la autoaplicación del evaluador parcial *offline*
   Hasta el momento hemos aproximado la autoaplicación definiendo una semántica para un subconjunto de Curry y especializando algunos intérpretes elaborados a partir de dicha semántica, aún así, la primera etapa del evaluador parcial no ha sido suficientemente precisa para lograr una especialización adecuada con intérpretes que ejecutan programas con argumentos dinámicos. Se ha detectado que el SCA no es muy preciso al manejar literales enteras en los

programas a especializar, asimismo en la etapa de especialización al realizar la sustitución de variables por la expresión que corresponde, en el proceso de desplegado, no está desplegando como se espera. Quedando indicadas en el programa residual algunas operaciones que deberían ser desplegadas. Por lo tanto es necesario optimizar aún más el actual SCA y el proceso de especialización del evaluador parcial *offline*.

3. Definir un proceso de evaluación parcial para programas de orden superior.
   De acuerdo a nuestro actual esquema de evaluación parcial *offline* y de manera intuitiva, para llevar a cabo esta propuesta sería necesario definir tanto un BTA como un análisis de terminación para procesar programas de orden superior (HO), contrastando con la contribución (3) donde el programa de HO a especializar se desfuncionaliza transformándolo a primer orden. Una vez que se logre anotar el programa HO; es necesario extender el proceso de la etapa de especialización para que acepte el programa anotado de HO. Es probable realizar por etapas esta propuesta. Primero hacer un estudio sobre la implementación del BTA (monovariante o polivariante) de HO, después intentar adaptar un análisis de terminación de HO del paradigma funcional al logico funcional. Posteriormente extender el propio proceso de especialización para programas de HO. Finalmente integrar las tres etapas en un prototipo de EP *offline* de HO.

Apéndice A

Apéndice

A-1. Ejemplo de un Intérprete en Curry

```
   module int_arith where
import FlatCurry -metaprogramming facilities (e.g., data structure for flat progs)
{-
main = 3*4-1+2
-}
main = ieval True [Func (“rev2”,"main") 0 Public (TCons ("Prelude",Ïnt") [ ])
(Rule [ ] (Comb FuncCall  (“Prelude”,“+”) [Comb FuncCall ("Prelude",“-”)
[Comb FuncCall ("Prelude","*") [Lit (Intc 3),Lit (Intc 4)],Lit (Intc 1)],
Lit (Intc 2)]))] (Comb FuncCall (“rev2”,“main”) [ ])
-these functiond are only used by the partial evaluator to annotate some expressions:
UNF x = x
MEM x = x
-function int is only used to test the interpreter (the partial evaluator calls
-directly to ieval to avoid having I/O function calls
{-
int p e = do (Prog _ _ _ funs _ ) <- readFlatCurry p
             print (ieval True funs e)
-}
-ieval: this is the main function of the interpreter
-arguments: a boolean flag, true for the top expression and false otherwise
-           the program
-           the expression to be evaluated
-vars (we use no environment!)
ieval _ _ (Var v) = Var v
-literals
ieval _ _ (Lit l) = Lit l

-constructor calls (observe the use of the Boolean flag)
ieval True v2  (Comb ConsCall v8 v9) = (Comb ConsCall v8 (ievalList True v2 v9) )
-OJO: stops in HNF!
-ievalList is only used to evaluate sequentially the arguments of
-a constructor call or an arithmetic operation
ievalList _  _ [ ] = [ ]
ievalList top funs (e:es) = ievalListaux top funs (ieval top funs e) es
ievalListaux top funs ne es = ne : (ievalList top funs es)

ieval False _   (Comb ConsCall c es) = Comb ConsCall c es

ieval top funs (Comb FuncCall (mn,f) es) =
  if mn == "Prelude"
   then if f == "failed"
     then Comb FuncCall (mn,f) es
     else if f == “==”
           then (ieval_EQ top funs (mn,f) es)
           else case f of
                    “+” -> ieval_ARITH top funs (mn,f)  es
                    “-” -> ieval_ARITH top funs (mn,f)  es
                    “*” -> ieval_ARITH top funs (mn,f)  es
   else ieval top funs (matchiRHS  funs (mn,f) es)

ieval top funs (Case ctype e ces) =
```

```
  case (ieval False funs e) of
    Comb ConsCall f es -> ieval top funs (matchBranch ces f es)
    Lit l -> ieval top funs (matchBranchLit ces l)
-problema con el renaming!
ieval_ARITH :: Bool -> [FuncDecl] -> QName -> [Expr] -> Expr
ieval_ARITH    top       fns         (mn,fn)  [e1,e2] =
  if (isLitInt e1) && (isLitInt e2)
     then (ieval_ARITH_aux (mn,fn) [e1,e2])
     else
        Comb FuncCall (mn,fn) (ievalList top fns [e1,e2])
-evaluation of simple arithmetic functions
ieval_ARITH_aux (_,f) [(Lit (Intc e1)),(Lit (Intc e2))] =
             case f of
                  ("*") -> Lit (Intc (e1*e2))
                  ("+") -> Lit (Intc (e1+e2))
                  ("-") -> Lit (Intc (e1-e2))
isLitInt :: Expr -> Bool
isLitInt e = case e of
                 (Lit (Intc _)) -> True
                 _              -> False
ieval_EQ :: Bool -> [FuncDecl] -> QName -> [Expr] -> Expr
ieval_EQ    top       fns         (mn,fn)  [e1,e2] =
  if (isLitInt e1) && (isLitInt e2)
     then (ieval_EQ_aux [e1,e2])
     else
        if (isLitInt e1) && (isVar e2)
           then Comb ConsCall ("Prelude","False") [ ]
           else ieval top fns (Comb FuncCall (mn,fn) (ievalList top fns [e1,e2]))
ieval_EQ_aux [(Lit (Intc e1)),(Lit (Intc e2))] =
             case (e1==e2) of
                  True -> Comb ConsCall ("Prelude","True") [ ]
                  _    -> Comb ConsCall ("Prelude","False") [ ]
-matchBranch and matchBranchLit are used to select the matching branch
-of a case expression:
matchBranch cbranches c es =
     case cbranches of
         [ ]     -> (Comb FuncCall ("Prelude","failed") [ ])
         (Branch (Pattern p vars) e):ces ->
                 if p==c then substitute vars es e
                         else matchBranch ces c es
matchBranchLit cbranches c =
     case cbranches of
         [ ]     -> (Comb FuncCall ("Prelude","failed") [ ])
         (Branch (LPattern p) e):ces ->
                 if p==c then e
                         else matchBranchLit ces c
--------------------------------------
- CALL UNFOLDING:
- match a right-hand side of a given function:
matchiRHS [ ] (_,_) _ = Comb FuncCall ("Prelude","failed") [ ]
matchiRHS (Func (_,fname) _ _ _ funrule : fds) (mn,name) es =
   if fname==name then matchRHS_aux funrule es
                  else matchiRHS fds (mn,name) es
matchRHS_aux (Rule vars rhs) es = substitute vars es rhs
```

```
substitute :: [Int] -> [Expr] -> Expr -> Expr
substitute vars exps expr = substituteAll vars exps expr
- substitute all occurrences of variables by corresponding expressions:
- * substitute all occurrences of var_i by exp_i in expr
-   (if vars=[var_1,...,var_n] and exps=[exp_1,...,exp_n])
- * leave all other variables unchanged (i.e., variables in case patterns)
-
substituteAll :: [Int] -> [Expr] -> Expr -> Expr
substituteAll vs es x =
    case x of
        (Var i)  -> replaceVar vs es i
        (Lit (Intc l))  -> Lit (Intc l)
        (Lit (Charc l))  -> Lit (Charc l)
        (Comb ConsCall c exps) -> Comb ConsCall c (mapsAll vs es exps)
        (Comb FuncCall c exps) -> Comb FuncCall c (mapsAll vs es exps)
-        (Comb (FuncPartCall ma) c exps) ->
-                        Comb (FuncPartCall ma) c (mapsAll vs es exps)
        (Case ctype e cases) -> Case ctype (substituteAll vs es e)
                                           (substituteAllCases vs es cases)
        (Or e1 e2) ->  (Or (substituteAll vs es e1) (substituteAll vs es e2))
        (Let [(lhs,rhs)] e) ->
             Let (mapsAllLet vs es [(lhs,rhs)]) (substituteAll vs es e)
        (Free vars e) -> Free vars (substituteAll vs es e)

replaceVar [ ] [ ] var = Var var
replaceVar (v:vs) (e:es) var  = if v==var then e
                                          else replaceVar vs es var
substituteAllCases _  _ [ ]              = [ ]
substituteAllCases vs es (tbranch:cases) =
   (substituteAllCase vs es tbranch) : (substituteAllCases vs es cases)
substituteAllCase vs es x =
             case x of
                  (Branch (Pattern (l,o) pvs) e) ->
                          Branch (Pattern (l,o) pvs) (substituteAll vs es e)
                  (Branch (LPattern l) e) ->
                          Branch (LPattern l) (substituteAll vs es e)
mapsAll vs es [ ]        = [ ]
mapsAll vs es (exp:exps) = (substituteAll vs es exp): (mapsAll vs es exps)
mapsAllLet vs es [ ] = [ ]
mapsAllLet vs es ((lhs,rhs):bindings) =
              (substituteAllLet vs es (lhs,rhs)):(mapsAllLet vs es bindings)
substituteAllLet :: [Int] -> [Expr] -> (VarIndex,Expr) -> (VarIndex,Expr)
substituteAllLet vs es (var,e)= (var,(substituteAll vs es e))

isVar :: Expr -> Bool
isVar e = case e of
               (Var _) -> True
               _       -> False
```

A-2. Ejemplo de un intérprete en Curry que ejecuta el programa de la sucesión de Fibonacci

```
module int_fib7 where
import System -for benchmarking
import FlatCurry -metaprogramming facilities (e.g., data structure for flat progs)
-these are just a test examples for the partial evaluator in order to
-avoid using I/O facilities which are not allowed
data Nat = Z | S Nat
{-
data Nat = Z | S Nat
main x = prod (fib x)  (fib (S (S (S (S (S Z))))))
fib x = case x of
            Z -> Z
            S Z -> S Z
            S (S n) -> sum (fib (S n)) (fib n)
prod x  n  = case x of
                 Z -> Z
                 (S v) ->  sum n (prod v n)
sum x  n = case x of
               Z -> n
               (S v) -> S (sum v n)
-}
main :: Expr -> Expr
main x = ieval True  [Func ("fib7","main") 1 Public (FuncType (TCons ("fib7","Nat") [])
(TCons ("fib7","Nat") [])) (Rule [1] (Comb FuncCall ("fib7","prod") [Comb FuncCall
("fib7","fib") [Var 1],Comb FuncCall ("fib7","fib") [Comb ConsCall ("fib7","S")
[Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","S")
[Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","Z") [][][][][][][][][]]),
Func ("fib7","fib") 1 Public (FuncType (TCons ("fib7","Nat") []) (TCons ("fib7","Nat") []))
(Rule [1] (Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") []) (Comb ConsCall
("fib7","Z") []),Branch (Pattern ("fib7","S") [2]) (Case Rigid (Var 2) [Branch
(Pattern ("fib7","Z") []) (Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","Z") []]),
Branch (Pattern ("fib7","S") [3]) (Comb FuncCall ("fib7","sum") [Comb FuncCall
("fib7","fib") [Comb ConsCall ("fib7","S") [Var 3]],Comb FuncCall ("fib7","fib")
[Var 3]])])])),
Func ("fib7","prod") 2 Public (FuncType (TCons ("fib7","Nat") [])
(FuncType (TCons ("fib7","Nat") []) (TCons ("fib7","Nat") []))) (Rule [1,2]
(Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") []) (Comb ConsCall ("fib7","Z") []),
Branch (Pattern ("fib7","S") [3]) (Comb FuncCall ("fib7","sum") [Var 2,Comb FuncCall
("fib7","prod") [Var 3,Var 2]])])),
Func ("fib7","sum") 2 Public (FuncType (TCons ("fib7","Nat") [])
(FuncType (TCons ("fib7","Nat") []) (TCons ("fib7","Nat") []))) (Rule [1,2]
(Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") []) (Var 2),
Branch (Pattern ("fib7","S") [3]) (Comb ConsCall ("fib7","S")
[Comb FuncCall ("fib7","sum") [Var 3,Var 2]])])]  (Comb FuncCall ("fib7","main") [x])
-these functions are only used by the partial evaluator to annotate some expressions:
UNF x = x
MEM x = x
-function int is only used to test the interpreter (the partial evaluator calls
```

```
-directly to ieval to avoid having I/O function calls
{-
int p e = do (Prog _ _ _ funs _ ) <- readFlatCurry p
             putStrLn (show (ieval True funs e))
-}
-ieval: this is the main function of the interpreter
-arguments: a boolean flag, true for the top expression and false otherwise
-           the program
-           the expression to be evaluated
-returns a pair (value,current value of n)
ieval :: Bool -> [FuncDecl] -> Expr -> Expr
-vars (we use no environment!)
ieval _ _ (Var v) = Var v
-literals
ieval _ _ (Lit l) = Lit l
-constructor calls (observe the use of the Boolean flag)
ieval True funs (Comb ConsCall c es) = Comb ConsCall c (ieval_args funs es)
ieval False _   (Comb ConsCall c es) = Comb ConsCall c es
ieval top funs (Comb FuncCall (mn,f) es) =
  if mn == "Prelude"
     then if f == "failed"
             then Comb FuncCall (mn,f) es
             else if f == "=="
                     then ieval_EQ (ieval_args funs es)
                     else case f of
                            "+" -> ieval_ARITH (mn,f) (ieval_args funs es)
                            " -> ieval_ARITH (mn,f) (ieval_args funs es)
                            "*" -> ieval_ARITH (mn,f) (ieval_args funs es)
     else ieval top funs (matchiRHS  funs (mn,f) es)
-case expressions:
ieval top funs (Case tc e ces) =
  case (ieval False funs e) of
    Comb ConsCall f es -> ieval top funs (matchBranch ces f es)
    Lit l -> ieval top funs (matchBranchLit ces l)
-evaluation of simple arithmetic functions
ieval_ARITH (_,f) [(Lit (Intc e1)),(Lit (Intc e2))] =
             case f of
                  ("*") -> Lit (Intc (e1*e2))
                  ("+") -> Lit (Intc (e1+e2))
                  ("-") -> Lit (Intc (e1-e2))
ieval_EQ [(Lit (Intc e1)),(Lit (Intc e2))] =
             case (e1==e2) of
                  True -> Comb ConsCall ("Prelude","True") []
                  _    -> Comb ConsCall ("Prelude","False") []
-ieval_args is only used to evaluate sequentially the arguments of
-a constructor call or an arithmetic operation
ieval_args funs es = case es of
                          [] -> []
                          (e:exps) -> (ieval True funs e) : (ieval_args funs exps)
-matchBranch and matchBranchLit are used to select the matching branch
-of a case expression:
matchBranch brs c es  =
     case brs of
          [] -> (Comb FuncCall ("Prelude","failed") [])
```

```
          (Branch (Pattern p vars) e : ces) ->
               if p==c then substituteAll vars es e else matchBranch ces c es
matchBranchLit brs c =
     case brs of
          [] -> (Comb FuncCall ("Prelude","failed") [])
          (Branch (LPattern p) e : ces) ->
               if p==c then e else matchBranchLit ces c
-------------------------------------
- CALL UNFOLDING:
- match a right-hand side of a given function:
matchiRHS funcs (mn,name) es =
     case funcs of
          [] -> Comb FuncCall ("Prelude","failed") []
          (Func (_,fname) _ _ _ funrule : fds) ->
                if fname==name then matchRHS_aux funrule es
                               else matchiRHS fds (mn,name) es
matchRHS_aux (Rule vars rhs) es = substituteAll vars es rhs
- substitute all occurrences of variables by corresponding expressions:
- * substitute all occurrences of var_i by exp_i in expr
-   (if vars=[var_1,...,var_n] and exps=[exp_1,...,exp_n])
- * leave all other variables unchanged (i.e., variables in case patterns)
-
- here we assume that the new variables in case patterns
- do not occur in the list "vars" of replaced variables!
substituteAll :: [Int] -> [Expr] -> Expr -> Expr
substituteAll vs es x =
    case x of
        (Var i)  -> replaceVar vs es i
        (Lit l)  -> Lit l
        (Comb ctyp (c,o) exps) -> Comb ctyp (c,o) (substituteAllArgs vs es exps)
        (Case ctype e cases) -> Case ctype (substituteAll vs es e)
                                           (substituteAllCases vs es cases)
        (Or e1 e2) -> (Or (substituteAll vs es e1) (substituteAll vs es e2))
replaceVar [] [] var = Var var
replaceVar (v:vs) (e:es) var  = if v==var then e
                                          else replaceVar vs es var
substituteAllArgs vs es exps =
          case exps of
               [] -> []
               (e:exprs) -> (substituteAll vs es e) : (substituteAllArgs vs es exprs)
substituteAllCases vs es cases =
          case cases of
               [] -> []
               (tbranch:brs) ->
                      (substituteAllCase vs es tbranch) : (substituteAllCases vs es brs)
substituteAllCase vs es x =
             case x of
                  (Branch (Pattern (l,o) pvs) e) ->
                          Branch (Pattern (l,o) pvs) (substituteAll vs es e)
                  (Branch (LPattern l) e) ->
                          Branch (LPattern l) (substituteAll vs es e)
```

A-3. Ejemplo de un intérprete especializado para el caso particular del programa de la sucesión de Fibonacci

```
module int_fib7_ann(Nat(Z,S),main,ieval_pe1,ieval_args_pe2,ieval_pe3,ieval_EQ_pe7,
ieval_ARITH_pe10,matchiRHS_pe16,substituteAll_pe19,replaceVar_pe20,
substituteAllArgs_pe21,substituteAllCases_pe22,matchBranch_pe34,matchBranchLit_pe35)
where
import FlatCurry
import System
data Nat = Z | S Nat
main :: b -> a
main v0 = ieval_pe1 (FlatCurry.Comb FlatCurry.FuncCall ("fib7","main") [v0])
ieval_pe1 :: b -> a
ieval_pe1 eval rigid&flex - CONFLICTING!!
ieval_pe1 (FlatCurry.Var v42) = FlatCurry.Var v42
ieval_pe1 (FlatCurry.Lit v43) = FlatCurry.Lit v43
ieval_pe1 (FlatCurry.Comb FlatCurry.ConsCall v45 v46) =
            FlatCurry.Comb FlatCurry.ConsCall v45 (ieval_args_pe2 v46)
ieval_pe1 (FlatCurry.Comb FlatCurry.FuncCall (v47,v48) v46) =
  case (v47 == "Prelude") of
     True -> case (v48 == "failed") of
       True -> FlatCurry.Comb FlatCurry.FuncCall (v47,v48) v46
       False -> case (v48 == “==”) of
         True -> ieval_EQ_pe7 (ieval_args_pe2 v46)
         False -> case v48 of
           v49 : v50 -> case (v49 == '+') of
             True -> case v50 of
               [] -> ieval_ARITH_pe10 v47 v48 (ieval_args_pe2 v46)
             False -> case (v49 == '-') of
               True -> case v50 of
                 [] -> ieval_ARITH_pe10 v47 v48 (ieval_args_pe2 v46)
               False -> case (v49 == '*') of
                 True -> case v50 of
                   [] -> ieval_ARITH_pe10 v47 v48 (ieval_args_pe2 v46)
     False -> ieval_pe1 (matchiRHS_pe16 v47 v48 v46)
ieval_pe1 (FlatCurry.Case v57 v58 v59) = case (ieval_pe3 v58) of
     FlatCurry.Comb v60 v61 v62 -> case v60 of
       FlatCurry.ConsCall -> ieval_pe1 (matchBranch_pe34 v59 v61 v62)
     FlatCurry.Lit v65 -> ieval_pe1 (matchBranchLit_pe35 v59 v65)
ieval_args_pe2 :: b -> a
ieval_args_pe2 eval rigid
ieval_args_pe2 [] = []
ieval_args_pe2 (v757 : v758) = (ieval_pe1 v757) : (ieval_args_pe2 v758)
ieval_pe3 :: b -> a
ieval_pe3 eval rigid&flex - CONFLICTING!!
ieval_pe3 (FlatCurry.Var v770) = FlatCurry.Var v770
ieval_pe3 (FlatCurry.Lit v771) = FlatCurry.Lit v771
ieval_pe3 (FlatCurry.Comb FlatCurry.ConsCall v773 v774) =
            FlatCurry.Comb FlatCurry.ConsCall v773 v774
ieval_pe3 (FlatCurry.Comb FlatCurry.FuncCall (v775,v776) v774) =
```

```
  case (v775 == "Prelude") of
     True -> case (v776 == "failed") of
       True -> FlatCurry.Comb FlatCurry.FuncCall (v775,v776) v774
       False -> case (v776 == ``==") of
         True -> ieval_EQ_pe7 (ieval_args_pe2 v774)
         False -> case v776 of
           v777 : v778 -> case (v777 == '+') of
             True -> case v778 of
               [] -> ieval_ARITH_pe10 v775 v776 (ieval_args_pe2 v774)
             False -> case (v777 == '-') of
               True -> case v778 of
                 [] -> ieval_ARITH_pe10 v775 v776 (ieval_args_pe2 v774)
               False -> case (v777 == '*') of
                 True -> case v778 of
                   [] -> ieval_ARITH_pe10 v775 v776 (ieval_args_pe2 v774)
     False -> ieval_pe3 (matchiRHS_pe16 v775 v776 v774)
ieval_pe3 (FlatCurry.Case v785 v786 v787) = case (ieval_pe3 v786) of
     FlatCurry.Comb v788 v789 v790 -> case v788 of
       FlatCurry.ConsCall -> ieval_pe3 (matchBranch_pe34 v787 v789 v790)
     FlatCurry.Lit v793 -> ieval_pe3 (matchBranchLit_pe35 v787 v793)
ieval_EQ_pe7 :: b -> a
ieval_EQ_pe7 eval rigid&flex - CONFLICTING!!
ieval_EQ_pe7 [Lit (Intc v9153),Lit (Intc v9157)] =
  case (v9153 == v9157) of
     True -> Comb ConsCall ("Prelude","True") []
     False -> Comb ConsCall ("Prelude","False") []
ieval_ARITH_pe10 :: d -> c -> b -> a
ieval_ARITH_pe10 eval rigid&flex - CONFLICTING!!
ieval_ARITH_pe10 v775 (v9162 : v9163) [Lit (Intc v9157),Lit (Intc v9161)] =
  case (v9162 == '*') of
     True -> case v9163 of
       [] -> FlatCurry.Lit (FlatCurry.Intc (v9157 * v9161))
     False -> case (v9162 == '+') of
       True -> case v9163 of
         [] -> FlatCurry.Lit (FlatCurry.Intc (v9157 + v9161))
       False -> case (v9162 == '-') of
         True -> case v9163 of
           [] -> FlatCurry.Lit (FlatCurry.Intc (v9157 - v9161))
matchiRHS_pe16 :: d -> c -> b -> a
matchiRHS_pe16 eval rigid
matchiRHS_pe16 v775 v776 v774 =
  case ("main" == v776) of
     True -> substituteAll_pe19 [1] v774 (Comb FuncCall ("fib7","prod")
        [Comb FuncCall ("fib7","fib") [Var 1], Comb FuncCall ("fib7","fib")
         [Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","S")
          [Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","S")
           [Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","Z") [][][][][][][][]]
     False -> case ("fib" == v776) of
       True -> substituteAll_pe19 [1] v774
               (Case Rigid (Var 1) [Branch (Pattern ("fib7","Z") [])
                 (Comb ConsCall ("fib7","Z") []),Branch (Pattern ("fib7","S") [2])
                   (Case Rigid (Var 2) [Branch (Pattern ("fib7","Z") [])
                      (Comb ConsCall ("fib7","S") [Comb ConsCall ("fib7","Z") []]),
                       Branch (Pattern ("fib7","S") [3]) (Comb FuncCall ("fib7","sum")
```

```
                        [Comb FuncCall ("fib7","fib") [Comb ConsCall ("fib7","S")
                        [Var 3]], Comb FuncCall ("fib7","fib") [Var 3]])])])
       False -> case ("prod" == v776) of
         True -> substituteAll_pe19 [1,2] v774
                 (Case Rigid (Var 1)
                             [Branch (Pattern ("fib7","Z") [])
                              (Comb ConsCall ("fib7","Z") []),
                              Branch (Pattern ("fib7","S") [3])
                               (Comb FuncCall ("fib7","sum") [Var 2,
                                Comb FuncCall ("fib7","prod") [Var 3,Var 2]])])
         False -> case ("sum" == v776) of
           True -> substituteAll_pe19 [1,2] v774
                   (Case Rigid (Var 1)
                    [Branch (Pattern ("fib7","Z") []) (Var 2),
                     Branch (Pattern ("fib7","S") [3]) (Comb ConsCall ("fib7","S")
                                 [Comb FuncCall ("fib7","sum") [Var 3,Var 2]])])
           False -> Comb FuncCall ("Prelude","failed") []
substituteAll_pe19 :: d -> c -> b -> a
substituteAll_pe19 eval rigid
substituteAll_pe19 v806 v807 (Var v2212) = replaceVar_pe20 v806 v807 v2212
substituteAll_pe19 v806 v807 (Lit v2213) = Lit v2213
substituteAll_pe19 v806 v807 (Comb v2214 (v2217,v2218) v2216) =
              Comb v2214 (v2217,v2218) (substituteAllArgs_pe21 v806 v807 v2216)
substituteAll_pe19 v806 v807 (Case v2219 v2220 v2221) =
                Case v2219 (substituteAll_pe19 v806 v807 v2220)
                           (substituteAllCases_pe22 v806 v807 v2221)
substituteAll_pe19 v806 v807 (Or v2222 v2223) =
                  Or (substituteAll_pe19 v806 v807 v2222)
                  (substituteAll_pe19 v806 v807 v2223)
replaceVar_pe20 :: d -> c -> b -> a
replaceVar_pe20 eval rigid&flex - CONFLICTING!!
replaceVar_pe20 [] [] v2229 = FlatCurry.Var v2229
replaceVar_pe20 (v3633 : v3634) (v3635 : v3636) v2229 = case (v3633 == v2229) of
     True -> v3635
     False -> replaceVar_pe20 v3634 v3636 v2229
substituteAllArgs_pe21 :: d -> c -> b -> a
substituteAllArgs_pe21 eval rigid
substituteAllArgs_pe21 v2231 v2232 [] = []
substituteAllArgs_pe21 v2231 v2232 (v3637 : v3638) =
                            (substituteAll_pe19 v2231 v2232 v3637) :
                            (substituteAllArgs_pe21 v2231 v2232 v3638)
substituteAllCases_pe22 :: d -> c -> b -> a
substituteAllCases_pe22 eval rigid
substituteAllCases_pe22 v2236 v2237 [] = []
substituteAllCases_pe22 v2236 v2237 (v3642 : v3643) =
 (case v3642 of
     Branch v5063 v5064 -> case v5063 of
       Pattern v5065 v5066 -> case v5065 of
         (v5067,v5068) -> Branch (Pattern (v5067,v5068) v5066)
                                 (substituteAll_pe19 v2236 v2237 v5064)
       LPattern v5069 -> Branch (LPattern v5069)
                                (substituteAll_pe19 v2236 v2237 v5064)
  ) : (substituteAllCases_pe22 v2236 v2237 v3643)
matchBranch_pe34 :: d -> c -> b -> a
```

```
matchBranch_pe34 eval rigid
matchBranch_pe34 [] v806 v807 = Comb FuncCall ("Prelude","failed") []
matchBranch_pe34 ((Branch (Pattern v2215 v2216) v2214) : v2212) v806 v807 =
  case (v2215 == v806) of
       True -> substituteAll_pe19 v2216 v807 v2214
       False -> matchBranch_pe34 v2212 v806 v807
matchBranchLit_pe35 :: c -> b -> a
matchBranchLit_pe35 eval rigid
matchBranchLit_pe35 [] v809 = Comb FuncCall ("Prelude","failed") []
matchBranchLit_pe35 ((Branch (LPattern v2216) v2215) : v2213) v809 =
  case (v2216 == v809) of
       True -> v2215
       False -> matchBranchLit_pe35 v2213 v809
- end of module int_fib7_ann_pe
```

Bibliografía

[AA97] S. Antoy and Z.M. Ariola. Narrowing the Narrowing Space. In *In Proc. of the 9th Int'l Symposium on Programming Languages: Implementations, Logics, and Programs (PLILP'97)*, pages 1–15. Springer LNCS 1292, 1997.

[AAF+98a] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Improving Control in Functional Logic Program Specialization. In *Proc. of the Int'l Static Analysis Symposium, SAS'98*, pages 262–277. Springer LNCS 1503, 1998.

[AAF+98b] E. Albert, M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Polygenetic Partial Evaluation of Lazy Functional Logic Programs. In *Proc. of Appia-Gulp-ProDe, AGP'98*, pages 151–164, 1998.

[AAHV99a] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. A Partial Evaluation Framework for Curry Programs. In *Proc. of LPAR'99*, pages 376–395. Springer LNAI 1705, 1999.

[AAHV99b] E. Albert, M. Alpuente, M. Hanus, and G. Vidal. Partial Evaluation of Residuating Functional Logic Programs. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 376–395, 1999.

[AAV00] E. Albert, S. Antoy, and G. Vidal. A formal approach to reasoning about the Effectiveness of Partial Evaluation. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000*, 2000.

[AAV01] E. Albert, S. Antoy, and G. Vidal. Measuring the Effectiveness of Partial Evaluation in Functional Logic Languages. In *Proc. of the 10th Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'00)*, pages 103–124. Springer LNCS 2042, 2001.

[AEH94] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.

[AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[AFJV97] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Specialization of Lazy Functional Logic Programs. In *Proc. of the ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, volume 32, 12 of *Sigplan Notices*, pages 151–162, New York, 1997. ACM Press.

[AFJV03] M. Alpuente, M. Falaschi, P. Julián, and G. Vidal. Uniform lazy narrowing. *Journal of Logic and Computation*, 13:287–312, 2003.

[AFMV04] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + strategies for transforming lazy functional logic programs. *Theor. Comput. Sci.*, 311(1-3):479–525, 2004.

[AFRV93] M. Alpuente, M. Falaschi, M.J. Ramis, and G. Vidal. Narrowing Approximations as an Optimization for Equational Logic Programs. In *Proc. of PLILP'93*, pages 391–409. Springer LNCS 714, 1993.

[AFV98] M. Alpuente, M. Falaschi, and G. Vidal. Partial Evaluation of Functional Logic Programs. *ACM TOPLAS*, 20(4):768–844, 1998.

[AHLV99] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Inductively Sequential Functional Logic Programs.

In *Proc. of ICFP'99*, volume 34.9 of *ACM Sigplan Notices*, pages 273–283. ACM Press, 1999.

[AHLV05] M. Alpuente, M. Hanus, S. Lucas, and G. Vidal. Specialization of Functional Logic Programs Based on Needed Narrowing. *Theory and Practice of Logic Programming*, 2005. To appear (`http://www.dsic.upv.es/users/elp/german/papers.html`).

[AHV00a] E. Albert, M. Hanus, and G. Vidal. Realistic Program Specialization in a Multi-Paradigm Language. In *Proc. of 9th Int'l Workshop on Functional and Logic Programming, WFLP'2000*, 2000.

[AHV00b] E. Albert, M. Hanus, and G. Vidal. Using an Abstract Representation to Specialize Functional Logic Programs. In *Proc. of the 7th Int'l Conf. on Logic for Programming and Automated Reasoning (LPAR'00)*, pages 381–398. Springer LNAI 1955, 2000.

[AHV01] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluator for a Multi-Paradigm Declarative Language. In *Proc. of 5th Int'l Symp. on Functional and Logic Programming (FLOPS'01)*, pages 326–342. Springer LNCS 2024, 2001.

[AHV02] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[AHV03] E. Albert, M. Hanus, and G. Vidal. A Residualizing Semantics for the Partial Evaluation of Functional Logic Programs. *Information Processing Letters*, 85(1):19–25, 2003.

[And92] L. O. Andersen. Partial evaluation of C and automatic compiler generation (extended abstract). In *Compiler Construction. 4th International Conference. (Paderborn, Ger-*

many). Lecture Notes in Computer Science, pages 251–257. Springer-Verlag, 1992.

[Ant92] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming (ALP'92)*, pages 143–157. Springer LNCS 632, 1992.

[Ant97] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. of the Int'l Conference on Algebraic and Logic Programming, ALP'97*, pages 16–30. Springer LNCS 1298, 1997.

[AOSV04] G. Arroyo, C. Ochoa, J. Silva, and G. Vidal. Towards CNC Programming Using Haskell. In Lemaitre Christian, Reyes Carlos A., and Gonzalez Jesus A., editors, *Advances in Artificial Intelligence-IBERAMIA 2004*, pages 386–395. Springer LNCS 3315, 2004.

[Apt97] K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[Arr04] G. Arroyo. Diseño de un Lenguaje de Especificación para CNC. IP Report (in Spanish), DSIC-UPV, 2004.

[ARS11] G. Arroyo, J.G. Ramos, and J. Silva. Generating CNC Code From a Domain Specific Language. In *Research in Computer Science (ENC'11)*, pages 151–161. C. Zepeda, R. Marcial, A. Sánchez, J.L.Zechinelli and M. Osorio (Eds.), 2011.

[ARSV06a] G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Proc. of the 16th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 55–61. Università Ca' Foscari di Venezia, 2006. Extended version to appear in Springer LNCS.

[ARSV06b] G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation Using

Size-Change Graphs. Technical report, Technical University of Valencia, 2006. Available from the following URL: `http://www.dsic.upv.es/users/elp/german/papers.html`.

[ARSV07] G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation Using Size-Change Graphs. In *Logic-Based Program Synthesis and Transformation. Revised and Selected Papers from LOPSTR'06*, pages 60–76. Springer LNCS 4407, 2007.

[ARTV07] G. Arroyo, J.G. Ramos, S. Tamarit, and G. Vidal. Offline Narrowing-Driven Specialization in Practice. In *ACTAS de las VII Jornadas sobre Programación y Lenguajes (PROLE'07)*, pages 137–146, 2007. II CONGRESO ESPAÑOL DE INFORMÁTICA (CEDI 2007).

[ARTV09] G. Arroyo, J.G. Ramos, S. Tamarit, and G. Vidal. A Transformational Approach to Polyvariant BTA of Higher-Order Functional Programs. In *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'08)*, pages 40–54. Springer LNCS 5438, 2009.

[AT99] S. Antoy and A. Tolmach. Typed Higher-Order Narrowing without Higher-Order Strategies. In *Proc. of the 4th Fuji Int'l Symp. on Functional and Logic Programming, FLOPS'99*, pages 335–352. Springer LNCS 1722, 1999.

[AV02] E. Albert and G. Vidal. The Narrowing-Driven Approach to Functional Logic Program Specialization. *New Generation Computing*, 20(1):3–26, 2002.

[AW05] A. Aurum and C. Wohlin. *Engineering and Managing Software Requirements*. Springer-Verlag, 2005.

[BBLM84] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. On the integration of logic programming and functional programming. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 160–166, 1984.

[BBLM86] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. LEAF: A Language which Integrates Logic, Equations and Functions. In *Logic Programming: Functions, Relations, and Equations*, pages 201–238. 1986.

[BCF91] A. Bossi, N. Cocco, and M. Fabris. Proving Termination of Logic Programs by Exploiting Term Properties. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT'91*, pages 153–180. Springer LNCS 494, 1991.

[BD77] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BL86] M. Bellia and G. Levi. The relation between logic and functional languages. *Journal of Logic Programming*, 3:217–236, 1986.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[Bul93] M.A. Bulyonkov. Extracting Polyvariant Binding Time Analysis from Polyvariant Specializer. In *Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 59–65. ACM, New York, 1993.

[CD89] C. Consel and O. Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30:79–86, 1989.

[CD93] C. Consel and O. Danvy. Tutorial notes on Partial Evaluation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 493–501. ACM, New York, 1993.

[CGLH05] S.J. Craig, J. Gallagher, M. Leuschel, and K.S Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of the 14th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'04)*. Springer LNCS, 2005. To appear.

[Chr92] J. Christian. Some termination criteria for narrowing and E-narrowing. In *Proc. of CADE-11*, pages 582–588. Springer LNCS 607, 1992.

[CK96] W.N. Chin and S.C. Khoo. Better Consumers for Program Specializations. *Journal of Functional and Logic Programming*, 1996(4), 1996.

[Con90] C. Consel. Binding time analysis for high order untyped functional languages. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 264–272, New York, NY, USA, 1990. ACM.

[CR91] J. Chabin and P. Réty. Narrowing directed by a graph of terms. In *Proc. of the 4th Int'l Conf. on Rewriting Techniques and Applications (RTA'91)*, pages 112–123. Springer LNCS 488, 1991.

[CT99] M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.

[DDL+97] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K.F. Sagonas. Termination Analysis for Tabled Logic Programming. In *In Proc. of the 7th Int'l Workshop on Logic Programming Synthesis and Transformation (LOPSTR'97)*, pages 111–127. Springer LNCS 1463, 1997.

[DDL+98] S. Decorte, D. De Schreye, M. Leuschel, B. Martens, and K.F. Sagonas. Termination Analysis for Tabled Logic Programming. In *Proc. of LOPSTR'97*, pages 111–127. Springer LNCS 1463, 1998.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

[DN01] O. Danvy and L.R. Nielsen. Defunctionalization at Work. In *PPDP*, pages 162–174, 2001.

[DS88] N. Dershowitz and G. Sivakumar. Goal-Directed Equation Solving. In *Proc. of 7th National Conf. on Artificial Intelligence*, pages 166–170. Morgan Kaufmann, 1988.

[fSTCITS82] International Organization for Standardization. Technical Committee: ISO 6983-1/TC 184/SC 1. Numerical control of machines – Program format and definition of address words, September 1982.

[Fut71] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[Fut99] Y. Futamura. Partial Evaluation of Computation Process—An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprint of article in *Systems, Computers, Controls* 1971.

[Gal93] J. Gallagher. Tutorial on Specialisation of Logic Programs. In *Proc. of the ACM Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 88–98. ACM, New York, 1993.

[GCGL02] S. Genaim, M. Codish, J.P. Gallagher, and V. Lagoon. Combining Norms to Prove Termination. In *Proc. of 3rd Int'l Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'02)*, pages 126–138. Springer LNCS 2294, 2002.

[GHGRA92] J. C. González-Moreno, M. T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing

as the operational semantics of functional logic programming. In *Proc. of CSL'92*, pages 216–230. Springer LNCS 702, 1992.

[Gho11] D. Ghosh. DSL for the Uninitiated. *Communications of the ACM*, 9(6):10:10–10:21, 2011.

[GJ96] A.J. Glenstrup and N.D. Jones. BTA Algorithms to Ensure Termination of Off-Line Partial Evaluation. In *Proc. of the 2nd Int'l Andrei Ershov Memorial Conf. on Perspectives of System Informatics*, pages 273–284. Springer LNCS 1181, 1996.

[GJ05] A.J. Glenstrup and N.D. Jones. Termination analysis and specialization-point insertion in offline partial evaluation. *ACM TOPLAS*, 27(6):1147–1215, 2005.

[GJMS96] R. Glück, J. Jørgensen, B. Martens, and M.H. Sørensen. Controlling Conjunctive Partial Deduction of Definite Logic Programs. In *Proc. Int'l Symp. on Programming Languages: Implementations, Logics and Programs, PLILP'96*, pages 152–166. Springer LNCS 1140, 1996.

[GLMP91] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.

[Glü02] R. Glück. Jones Optimality, BindingTime Improvements, and the Strength of Program Specializers. In *Proceedings of the Symposium on Partial evaluation and semantics-based program manipulation*, ASIA-PEPM '02, pages 9–19. ACM, 2002.

[GM93] J.C. González-Moreno. A correctness proof for Warren's HO into FO translation. In *Proc. of 8th Italian Conf. on Logic Programming, GULP'93*, pages 569–585, 1993.

[GS94] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *Proc. of PLILP'94*, pages 165–181. Springer LNCS 844, 1994.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han06] M. Hanus. Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~mh/curry/`, 2006.

[Han11] M. Hanus. Flatcurry: An intermediate representation for Curry programs. `http://www.informatik.uni-kiel.de/~curry/flat/`, May 2011.

[HeAE+04] M. Hanus (ed.), S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.6.0: The Portland Aachen Kiel Curry System—User Manual. Technical report, University of Kiel, Germany, 2004.

[HGU01] T. Hortalá-González and E. Ullán. An Abstract Machine Based System for a Lazy Narrowing Calculus. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming (FLOPS 2001)*, pages 216–232. Springer LNCS 2024, 2001.

[HH82] G. Huet and J.M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, 1982.

[HK96] M. Hanus and H. Kuchen. Integration of functional and logic programming. *ACM Comput. Surv.*, 28(2):306–308, 1996.

[HKMN95] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Work-*

shop on Visions for the Future of Logic Programming, pages 95–107, 1995.

[HL92] G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.

[HL94] P.M. Hill and J.W. Lloyd. The Godel Programming Languaje. 1994.

[Hol91] C.K. Holst. Finiteness Analysis. In *Proc. of Functional Programming Languages and Computer Architecture*, pages 473–495. Springer LNCS 523, 1991.

[HP99] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[Hud98] P. Hudak. Modular Domain Specific Languages and Tools. In *Procedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[Hul80] J.M. Hullot. Canonical Forms and Unification. In *Proc of 5th Int'l Conf. on Automated Deduction*, pages 318–334. Springer LNCS 87, 1980.

[JG05] N.D. Jones and A. Glenstrup. Partial Evaluation Termination Analysis and Specialization-Point Insertion. *ACM TOPLAS*, 2005. To appear.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Englewood Cliffs, NJ, 1993.

[Joh85] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture (Nancy, France)*, pages 190–203. Springer LNCS 201, 1985.

[Jon96] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, Sept. 1996.

[Jon04] N.D. Jones. Transformation by Interpreter Specialisation. *Science of Computer Programming*, 52:307–339, 2004.

[Jør91] J. Jørgensen. Compiler generation by partial evaluation. Technical report, Master thesis, DIKU, 1991.

[Jør92a] J. Jørgensen. Generating a Compiler for a Lazy Language by Partial Evaluation. In *Proc. of 19th ACM Symp. on Principles of Programming Languages*, pages 258–268. ACM, New York, 1992.

[Jør92b] J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *POPL*, pages 258–268, 1992.

[JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Jul00] P. Julián. *Especialización de Programas Lógico-Funcionales Perezosos*. PhD thesis, DSIC-UPV, May. 2000. In spanish.

[Kom82a] H. J. Komorowski. QLOG: The Programming Environment for Prolog in Lisp. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 315–322. Academic Press, London, 1982.

[Kom82b] H.J. Komorowski. Partial Evaluation as a Means for Inferencing Data Structures in an Applicative Language: A Theory and Implementation in the Case of Prolog. In *Proc. of 9th ACM Symp. on Principles of Programming Languages*, pages 255–267, 1982.

[Laf98] L. Lafave. *A Constraint-Based Partial Evaluator for Functional Logic Programs and its Application*. PhD thesis, University of Bristol, 1998.

[LCBV04] M. Leuschel, S.-J. Craig, M. Bruynooghe, and W. Vanhoof. Specialising Interpreters Using Offline Partial Deduction. In *Program Development in Computational Logic*, pages 340–375. Springer LNCS 3049, 2004.

[LDdW96] M. Leuschel, D. De Schreye, and A. de Waal. A Conceptual Embedding of Folding into Partial Deduction: Towards a Maximal Integration. In M. Maher, editor, *Proc. of JICSLP'96*, pages 319–332. The MIT Press, Cambridge, MA, 1996.

[Leu02] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, pages 379–403. Springer LNCS 2566, 2002.

[Leu07] M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks, 2007. Available at URL:
`www.ecs.soton.ac.uk/~mal/systems/dppd.html`.

[LG97] L. Lafave and J.P. Gallagher. Partial Evaluation of Functional Logic Programs in Rewriting-based Languages. Technical Report CSTR-97-001, Department of Computer Science, University of Bristol, Bristol, England, March 1997.

[LJBA01] C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.

[LJVB04] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline Specialisation in Prolog using a Hand-Written Compiler Generator. *Theory and Practice of Logic Programming*, 4(1-2):139–191, 2004.

[LK99] W. Lux and H. Kuchen. An Efficient Abstract Machine for Curry. In *Proc. of the 8th Int'l Workshop on Functional and Logic Programming (WFLP'99)*, pages 171–181, 1999.

[Llo95] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.

[LLR93] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *Proc. of PLILP'93*, pages 184–200. Springer LNCS 714, 1993.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[LS97] N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. of Int'l Conf. on Logic Programming (ICLP'97)*, pages 63–77. The MIT Press, 1997.

[Lux03] W. Lux. Münster Curry 0.9.6—User's Guide. Technical report, University of Münster, Germany, November 2003.

[MBY+00] J. Michaloski, S. Birla, C.J. Yen, R. Igou, and G. Weinert. An Open System Framework for Component-Based CNC Machines. *ACM Computing Surveys*, 32(23), 2000.

[MG95] B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. of the 12th Int'l Conf. on Logic Programmin (ICLP'95)*, pages 597–611. MIT Press, 1995.

[Mog89] T. Mogensen. *Binding Time Aspects of Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989.

[MR92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *J. Logic Programming*, 12(3):191–224, 1992.

[MS97] T. Æ. Mogensen and P. Sestoft. Partial evaluation. In Allen Kent and James G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 247–279. Marcel Dekker, 270 Madison Avenue, New York, New York 10016, 1997.

[MTH90] R. Milner, M. Tofte, and R. Harper. The Definition of Standard ML. In *Seminar on Concurrency*. MIT Press, 1990.

[NM88] G. Nadathur and D. Miller. An Overview of λProlog. In *Proc. of Fifth Int'l Conf. on Logic Programming*, pages 810–827. The MIT Press, Cambridge, MA, 1988.

[Ott00] T.P. Otto. An apl compiler. In International Conference on APL, editor, *Proceedings of the international conference on APL-Berlin-2000 conference*, pages 186–193. ACM Press - New York, NY, USA, 2000.

[PJ03] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

[PP94] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[PP96] A. Pettorossi and M. Proietti. A Comparative Revisitation of Some Program Transformation Techniques. In *Proc. of the 1996 Dagstuhl Seminar on Partial Evaluation*, pages 355–385. Springer LNCS 1110, 1996.

[Pre94] C. Prehofer. Higher-order narrowing. In *Proc. Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 507–516. IEEE Computer Society Press, 1994.

[Ram30] F. Ramsey. On a problem of formal logic. In *Proc. of the London Mathematical Society*, volume 30, pages 264–286, 1930.

[Ram07] J. G. Ramos. *Una Aproximación Offline a la Evaluación Parcial dirigida por Narrowing.* PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 2007.

[Red85] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of the Symposium on Logic Programming (SLP'85)*, pages 138–151. IEEE Press, 1985.

[Rey98] J.C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–297, 1998. Reprinted from the proceedings of the 25th ACM National Conference (1972).

[RS82a] J. Robinson and E. Sibert. LOGLISP: Motivation, design and implementation. In K. Clark and S. A. Taernlund, editors, *Logic programming*, pages 299–313. Academic Press, London, 1982.

[RS82b] J.A. Robinson and E.E. Sibert. LOGLISP: An Alternative to PROLOG. *Machine Intelligence*, 10:399–419, 1982.

[RSV04] J. G. Ramos, J. Silva, and G. Vidal. An Embedded Language Approach to Router Specification in Curry. In *SOFSEM*, pages 277–288, 2004.

[RSV05a] J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 228–239. ACM Press, 2005.

[RSV05b] J.G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems.

ACM SIGPLAN Notices (Proc. of ICFP'05), 40(9):228–239, 2005.

[Sea94] W. Seames. *CNC: Concepts and Programming.* Delmar Learning, 1994.

[Ser07] D. Sereni. Termination Analysis and Call Graph Construction for Higher-Order Functional Programs. In *Proc. of the 12th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP'07)*, pages 71–84. ACM, 2007.

[SGJ96a] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[SGJ96b] M.H. Sørensen, R. Glück, and N.D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[SGJ+99] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.

[SKGST08] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting, 2008. Submitted for publication.

[Sla74] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[TBC+98] S. Thibault, L. Bercot, C. Consel, R. Marlet, G. Muller, and J. Lawall. Experiments in program compilation by interpreter specialization. Technical Report 3588, INRIA, December 1998.

[TG03] R. Thiemann and J. Giesl. Size-Change Termination for Term Rewriting. In *Proc. of the 14th Int'l Conf. on Rewriting Techniques and Applications (RTA'03)*, pages 264–278. Springer LNCS 2706, 2003.

[TG05] R. Thiemann and J. Giesl. The Size-Change Principle and Dependency Pairs for Termination of Term Rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 16(4):229–270, 2005.

[Tur86] V.F. Turchin. Program Transformation by Supercompilation. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, 1985*, pages 257–281. Springer LNCS 217, 1986.

[Van00] W. Vanhoof. Binding-Time Analysis by Constraint Solving. A Modular and Higher-Order Approach for Mercury. In *LPAR*, pages 399–416, 2000.

[Vid96] G. Vidal. *Semantics-Based Analysis and Transformation of Functional Logic Programs.* PhD thesis, DSIC-UPV, Sept. 1996. In spanish.

[Vid02] G. Vidal. Cost-Augmented Narrowing-Driven Specialization. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'02)*, pages 52–62. ACM Press, 2002.

[Vid04] G. Vidal. Cost-Augmented Partial Evaluation of Functional Logic Programs. *Higher-Order and Symbolic Computation*, 17(1-2):7–46, 2004.

[VSD01] S. Verbaeten, K. Sagonas, and D. De Schreye. Termination Proofs for Logic Programs with Tabling. *ACM Transactions on Computational Logic*, 2(1):57–92, 2001.

[Wad90] P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[War82] D. H. D. Warren. Higher-Order Extensions to Prolog – Are they needed? In Michie Hayes-Roth and Pao, editors, *Machine Intelligence*, volume 10. Ellis Horwood, 1982.

[WWK01] M. Weck, J. Wolf, and D. Kiritsis. STEP-NC The STEP Compliant NC Programming Interface: Evaluation and Improvement of the Modern Interface. In *Proc. of the ISM Project Forum 2001*, 2001.