Optimizando la evaluación parcial *offline* dirigida por *narrowing*

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

Memoria presentada para optar al título de:

Master en Ingeniería del Software, Métodos Formales
y Sistemas de Información

Director:

Dr. Germán F. Vidal Oriola

Valencia, Enero de 2008

Dedicada a:
Felisa y Florentino

Prólogo

Esta investigación ofrece el conocimiento más actual del esquema *offline* de evaluación parcial de programas dirigida por narrowing en el ámbito del paradigma de la programación lógico funcional. Describe el tema desde su origen con el esquema *online* de evaluación parcial y se refiere ampliamente a la primera aproximación de especialización *offline* en el mencionado paradigma; así mismo, especifica detalladamente cómo se ha logrado mejorar dicha aproximación, concretamente se han desarrollado prototipos en lenguaje Curry basándose en sus lenguajes de nivel intermedio AbstractCurry y FlatCurry. Finalmente, se plantea la posibilidad de realizar la compilación por evaluación parcial extendiendo el prototipo implementado y construyendo un metaintérprete.

Descripción del contenido

El Capítulo 1 describe los conceptos generales de evaluación parcial *online* y *offline*, así como el esquema de evaluación parcial dirigido por *narrowing* (NPE). Este capítulo también hace referencia a la especialización de programas lógicos y funcionales. Se hace una reseña de la evaluación parcial de lenguajes lógico funcionales donde se aplica el mecanismo del *narrowing* y además se menciona la formalización de NPE y de algunas instancias del mismo. Se presenta una breve evolución del desarrollo de NPE mencionando las semánticas operacionales que se aplican e inclusive la medición del coste de ejecución de los programas especializados hasta llegar al primer esquema offline de NPE.

El Capítulo 2 presenta la definición los conceptos usados a los largo de todo el documento, los cuales están basados en el marco conceptual de la reescritura de términos, por lo que se recomienda al lector como un capítulo fundamental para la comprensión del resto del trabajo.

El Capítulo 3 presenta detalladamente el primer esquema de evaluación

parcial *offline* el cual asegura cuasi-terminación introduciendo una caracterización denominada *no creciente* aplicada sobre sistemas de reescritura de términos; aquí se introduce un algoritmo de anotación para una clase más amplia de programas, los llamados SRT's inductivamente secuenciales, basado en la caracterización *no creciente*. Se aplica dicho algoritmo con el propósito de convertir en cuasi-terminante el SRT en caso que éste no lo sea y se define una extensión del narrowing necesario: el *narrowing necesario generalizante*, para soportar la especialización del programa anotado.

En el Capítulo 4 se extiende el esquema *offline* de NPE presentado en el capítulo 3 introduciendo una nueva estrategia de anotación la cual esta basada en una combinación del principio de los grafos size-change, que sirve para identificar una forma particular de cuasi-terminación, junto con el resultado de un análisis binding-time que ayuda a mejorar la precisión de la anotación. El resultado de los experimentos indican, en promedio, que los programas residuales se ejecutan más rápido que los obtenidos con el prototipo del primer esquema *offline* de NPE.

El Capítulo 5 esta relacionado con la descripción pormenorizada de las estrategias de control usadas en la fase de especialización del evaluador parcial *offline* dirigido por narrowing, así como de la presentación de un nuevo procedimiento de anotación basado en el análisis de cuasi-terminación del Capítulo 4 y que permite definirla como una estrategia de especialización *puramente offline*.

El Capítulo 6 plantea las contribuciones de la tesis y sugiere algunas líneas para trabajo futuro.

Agradecimientos

Primero, antes que todo, quiero agradecer a mi director de tesis Germán F. Vidal por su acendrado compromiso para culminar este proyecto. Quiero agradecer además de forma particular a J. Guadalupe Ramos, por toda su ayuda, a Josep Silva, Claudio Ochoa y Salvador Tamarit, con quienes he participado en reuniones de trabajo y en conjunto hemos resuelto los obstáculos que implican esta responsabilidad. También a los profesores Isidro Ramos, María Alpuente, María José Ramírez, Matilde Celma, Salvador Lucas y Oscar Pastor por acercarnos a la calidad del conocimiento que debe tener una institución de primera clase. Y al resto de los miembros de grupo ELP del DSIC, gracias por su compañerismo, son una verdadera sociedad civilizada.

A las autoridades del CIIDET, José Carlos Paz, Roberto de la Torre, Juan Manuel Ricaño, a toda la comunidad del mismo; gracias por su apoyo. A las

instituciones de México que me apoyaron: SES-ANUIES, DGEST-SEP. A las instituciones en España: Vicerrectorado de la Fundación UPV, en especial al personal de administración del DSIC. A todos mis familiares en especial a Ma. Abraham. A Angélica mi esposa, verdadero pilar de la familia.

Enero 2008 GAD

Índice general

Capítulo 1

Introducción

La evaluación parcial ha sido objeto de un rápido incremento en la actividad en la década pasada ya que ofrece un paradigma unificador de un amplio espectro de trabajo en optimización de programas, interpretación, compilación, otras formas de generación de programas e incluso 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 una definición interpretativa de un lenguaje de programación, pero la evaluación parcial tiene importantes aplicaciones en la computación científica, metaprogramación y sistemas expertos [Jon96].

La evaluación parcial de programas es una técnica formal para la especialización y optimización de programas. Un evaluador parcial toma un programa y *parte* de sus datos de entrada (los llamados *datos estáticos*) e intenta llevar a cabo todas las computaciones que sean posibles a partir de tales datos. El evaluador parcial devuelve un programa nuevo, denominado programa *residual*, el cual se ejecuta generalmente de manera más eficiente que el programa original, ya que las computaciones que dependen de los datos estáticos se han realizado en la fase evaluación parcial de una vez y para siempre [Ram07]. 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. En [AV02] se examinan los fundamentos de la evaluación parcial dirigida por *narrowing*.

Un aspecto clave para asegurar la terminación del proceso de especialización es la adecuada selección de las partes del programa que deben computarse en tiempo de especialización [GJ05]. La atención se debe centrar en aquellas

partes (normalmente llamadas a función) que podrían producir computaciones infinitas. Para evitarlas, se aplica a menudo alguna forma de generalización. La decisión acerca de qué llamadas a función se deberían generalizar puede tomarse de modo *online* u *offline*. Los evaluadores parciales *online* deciden qué llamadas a función se deben generalizar y cuáles no durante el proceso de especialización, mientras que los evaluadores parciales *offline* proceden en dos etapas: la primera hace un análisis estático del programa y "anota" aquellas llamadas a función que se deberían generalizar; la segunda (la etapa de especialización propiamente dicha) sólo sigue las anotaciones y de esa manera el evaluador parcial *offline* es normalmente más rápido (pero generalmente menos preciso) [Ram07].

En lo que se refiere a especialización de programas lógicos, Gallagher [Gal93] introdujo un algoritmo básico para evaluación parcial de programas lógicos con respecto a un objetivo (una llamada a función); el algoritmo es iterativo y consta originalmente de dos fases independientes.

1. un nivel global, en el que se determinan las llamadas a especializar, y
2. un nivel local, en el que se despliega una llamada tanto como sea posible.

Dicho algoritmo se ha adaptado tanto para especialización *online* como para especialización *offline* en el paradigma de los lenguajes lógico funcionales.

A grandes rasgos, dado un programa lógico P y una llamada inicial Q, un evaluador parcial debe construir un árbol SLD finito—posiblemente incompleto—para Q con P tal que cada átomo, en las hojas, es una instancia de un átomo previamente seleccionado. La terminación de este árbol SLD puede ser garantizada ya sea *online* u *offline*. Algunas técnicas de evaluación parcial (e.g., [LB02, LMD98]) usan chequeos onerosos—tal como la subsumción homeomórfica [Leu02]—para evitar derivaciones infinitas. En contraste, la evaluación parcial *offline* (e.g., [LJVB04]) procede en dos fases: primero el programa es analizado—utilizando el llamado análisis *binding-time*, BTA—lo que regresa un programa *anotado*; las anotaciones son usadas para indicar qué átomos pueden ser desplegados, qué argumentos deben ser generalizados (i.e., reemplazados por variables frescas), etc., para garantizar la terminación del proceso de especialización. Enseguida el segundo paso es una extensión de un interpréte SLD que simplemente obedece las anotaciones. Desafortunadamente no hay todavía un BTA completamente automático para asegurar la terminación de la evaluación parcial *offline*. Recientes avances, incluidos en [CGLH05a], presentan un BTA completamente automático pero sólo ofrece garantía de terminación parcial.

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 y lógico funcionales como Haskell [PJ03] o Curry [Han06]. NPE, emplea un refinamiento del *narrowing* [Sla74] para ejecutar computaciones simbólicas, siendo narrowing necesario (*needed narrowing*) [AEH00] la estrategia que presenta mejores propiedades. En general, el espacio de términos computados por *narrowing* puede ser infinito. Sin embargo, incluso en este caso, NPE puede terminar cuando el programa original es *cuasi-terminante* con respecto a la estrategia de narrowing considerada, i.e., cuando se computan sólo 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 cómputo se pueden evitar insertando una llamada a alguna variante encontrada previamente (una técnica conocida como *inserción de puntos de especialización* en la literatura de evaluación parcial [GJ05]).

1.1. Antecedentes

La evaluación parcial (PE) ha sido ampliamente aplicada en el campo de la programación funcional [CD93, JGS93, Tur86] y de la programación lógica [Gal93, Kom82, LS91, PP94], donde se conoce habitualmente como *deducción parcial*. Aunque los objetivos son similares, los métodos son en general diferentes debido al modelo computacional subyacente. Las técnicas en evaluación parcial convencional de programas funcionales usualmente dependen de la propagación de constantes y de la reducción de expresiones, mientras que las técnicas para los lenguajes lógicos explotan la propagación de parámetros basada en la unificación [GS94]. Las transformaciones de plegado y desplegado, las cuales fueron introducidas por [BD77] para programas funcionales, son explotadas en diferentes formas por las diferentes técnicas de PE. El desplegado (*unfolding*) es esencialmente el reemplazo de una llamada por su definición, con sustituciones apropiadas. El plegado (*folding*) es la transformación inversa, este es el reemplazo de alguna pieza de código por un llamado a función equivalente. En la última década se han establecido algunas correspondencias entre las diferentes técnicas, en particular los casos [GS94, PP96, SGJ96a].

La PE es una técnica de optimización de programas que preserva la semántica de su ejecución, la cual consiste en la especialización del programa con respecto a parte de sus datos de entrada [CD93, JGS93]. En [AFV98] se formaliza el primer algoritmo genérico para la especialización de lenguajes lógico

funcionales (FL). En contraste al enfoque utilizado en los lenguajes puramente funcionales, establecen el mecanismo de computación (basado en unificación) de narrowing tanto para la especialización como para su ejecución. La dimensión lógica del narrowing les permite tratar expresiones que contienen información parcial de manera natural, por medio de variables lógicas y unificación, mientras que la dimensión funcional les permite considerar estrategias de evaluación eficiente tanto como incluir una fase de simplificación determinista, lo cual ofrece mejores oportunidades de optimización y mejora tanto la especialización en general como la eficiencia del método. La PE dirigida por *Narrowing* (NPE) se formaliza dentro del marco teórico establecido por Lloyd y Shepherdson [1991] y Martens y Gallagher [1995] para deducción parcial de programas lógicos, aunque varios conceptos han sido generalizados para tratar con características funcionales tales como llamadas a función anidadas, estrategias de evaluación perezosa e impaciente, o pasos de reducción deterministas.

En [AFJV97, Jul00] se formula una instancia de 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 conjunto 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 similares a las de [GJMS96, LDdW96] para evitar una generalización excesiva a nivel global.

Como hemos introducido, [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 la deducción parcial donde se hace una clara distinción entre control *local* y *global*. A grandes rasgos, el control local involucra la construcción de árboles parciales de narrowing para términos individuales, mientras que el control global está 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. Introducen 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, el cual 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 plano de cerradura. Usando la terminología de Glück y Sørensen [1996], su estrategia de control global les permite producir tanto especializaciones *polivariantes* como *poligenéticas*, i.e., 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 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

(del inglés Lazy Narrowing with definitional Trees). 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, i.e., el cálculo RLNT (Residualizing LNT). A partir del nuevo cálculo es posible implementar un evaluador parcial realista para programas en representación abstracta. En [AHV03] se demuestra la equivalencia entre el cálculo LNT y RLNT. En [AHV01] y [AHV02] se describe el esquema práctico de evaluación parcial dirigido por narrowing para programas abstractos (traducidos a partir de programas Curry) y la forma en que se resuelven las características extendidas del lenguaje: guardas, restricciones, funciones externas, orden superior, etc.

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

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*.

Recientemente, en [RSV05] identificaron una clase sistemas de reescritura cuasi-terminantes (con respecto a narrowing necesario) a los cuales llamaron *no-crecientes*. Esta caracterización es puramente sintáctica y muy fácil de verificar, aunque muy restringida para ser útil en la práctica. Por ello, [RSV05] introdujeron un esquema *offline* para NPE en el cual 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 cual asegura la terminación del proceso).

1.2. Objetivos y desarrollo principal

En esta investigación hemos perfeccionado la caracterización de los sistemas de reescritura *no-crecientes* [RSV05], 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 especifica de cuasi-terminación, i.e., que sólo se pueden producir 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 del uso combinado de los grafos *size-change* y del BTA no nos permite inferir que los sistemas de reescritura cuasi-terminan, procedemos como en [RSV05] y anotamos los subtérminos problematicos para ser generalizados en tiempo de evaluación parcial [ARSV06].

Hasta ahora tenemos dos aproximaciones de anotación a las cuales hemos denominado *híbrida* y *offline pura*. La estrategia *híbrida* realiza 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. Esto se ha superado 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. En este caso, el proceso de especialización está dirigido completamente por las anotaciones en el momento de aplicar el operador de desplegado [ARTV07].

Capítulo 2

Preliminares

La reescritura de términos (term rewriting [BN98]) ofrece un marco apropiado para modelar el componente de primer orden de muchos lenguajes de programación funcionales y lógico funcionales.[1] Consecuentemente, en el resto de este documento seguimos el concepto estándar de reescritura de términos para elaborar nuestros resultados [RSV05].

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*.

Para denotar al conjunto de *términos* y *términos constructores* (ambos incluyendo posiblemente variables de $\mathcal{V}$) usamos $\mathcal{T}(\mathcal{F}, \mathcal{V})$ y $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectivamente. Con $\mathcal{V}ar(t)$ denotamos al conjunto de variables que aparecen en un término t. Se denomina variable *fresca* a una variable nueva que no ha sido empleada con anterioridad. Un término t es *básico* (o *ground*) si $\mathcal{V}ar(t) = \varnothing$. Un término t es una variante de t' si ambos son iguales módulo renombramien-

[1] No obstante, las características de orden superior (higher-order) pueden se modeladas utilizando un operador de aplicación explícito, i.e., por defuncionalización [Rey98].

to 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 $f/n \in D$ 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.

2.1.1. Sistemas de reescritura de términos

Un conjunto de reglas de reescritura (o ecuaciones orientadas) de la forma $l \to r$ tal que l es un término no variable y r es un término cuyas variables aparecen en l es llamado un *sistema de reescritura de términos* (SRT)[2]; los

[2]De acuerdo a su traducción del concepto en inglés: *term rewriting system* (TRS)

términos l y r son llamados parte izquierda y parte derecha de la regla, respectivamente. Dado un SRT $\mathcal{R}$ determinado sobre una signatura $\mathcal{F}$, los símbolos $\mathcal{D}$ *definidos* son los símbolos raíz de las partes izquierdas de las reglas, i.e., son los símbolos de función del SRT, y los *constructores* son $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. Nos limitamos a signaturas y SRTs finitos. Indicamos el dominio de términos y *términos constructores* por medio de $\mathcal{T}(\mathcal{F}, \mathcal{V})$ y $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectivamente, donde $\mathcal{V}$ es un conjunto de variables con $\mathcal{F} \cap \mathcal{V} = \varnothing$.

Un SRT $\mathcal{R}$ esta *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 $\mathcal{V}$ ocurre al menos 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 $\mathcal{R}$ cuya 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 en 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 $root(t) \in \mathcal{C}$). Una *posición* p en un término t esta representada por una secuencia de números naturales, donde ϵ indica la posición raíz. Las posiciones son usadas para localizar los nodos de un término visto como un árbol: $t|_p$ indica el *subtérmino* de t en la posición p y $t[s]_p$ indica el resultado de *reemplazar el subtérmino* $t|_p$ por el término s. Un término t esta definido (*ground*) si $\mathcal{V}ar(t) = \varnothing$. Un término t es una *variante* del término t' si son iguales módulo renombramiento de variables.

Una *sustitución* σ es una representación 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. 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 $\sigma(s) = \sigma(t)$. 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'. En lo subsiguiente, escribimos $\overline{o_n}$ para la *secuencia de objetos* $o_1, \ldots, o_n$.

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 identificación de sus argumentos análogo a la inducción (de la estructura) de un tipo de datos. La especie inductivamente secuencial no es una restricción para la programación. De hecho, la componente de primer orden (first-order) de varios programas (lógico) funcionales escritos en, e.g., Haskell, ML o Curry, es inductivamente secuencial.[3] 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 Considere las siguientes reglas las cuales definen la función menor o igual (less-or-equal) en los números naturales (basados en *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 (left-hand sides) pueden ser organizadas jerárquicamente como sigue:

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

donde los argumentos en las cajas indican los casos a discriminar (esto es similar al concepto de árbol definicional en [Ant92]).

2.1.2. Semántica

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

[3] Curry también acepta *solapamiento (overlapping)* de sistemas inductivamente secuenciales. Esta clase extiende los sistemas inductivamente secuenciales con un operador de disyunción el cual introduce un indeterminismo desconocido. No obstante, las típicas propiedades de secuencialidad inductiva implican asimismo sistemas de solapamiento.

reescritura a un término, i.e., $t \rightarrow_{p,R} s$ si existe una posición p en 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 son omitidos en la notación cuando están claros en el contexto). La instancia $\sigma(l)$ de la parte izquierda de la regla R en cuestión es llamada *redex*[4]. Un término t es *irreducible* o está en *forma normal* si no hay término s con $t \rightarrow s$. Indicamos por medio de $\rightarrow^+$ la cerradura transitiva de $\rightarrow$, 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.

Los programas *lógico* funcionales varían principalmente de los programas puramente funcionales en que las llamadas a función pueden contener variables *libres*. Para evaluar esos términos con variables, es generalmente necesario generar instancias de estas variables a términos apropiados con el propósito de aplicar un paso de reescritura. Esto puede ser hecho usando unificación en lugar de emparejamiento en el paso de reescritura lo cual es conocido como *narrowing*, i.e., el mecanismo del *narrowing* permite descubrir instancias de las variables de manera indeterminista tal que un paso de reescritura sea posible [Han94]. Formalmente, $t \rightsquigarrow_{p,R,\sigma} t'$ es un *paso de narrowing* sii p es una posición novariable de t y $\sigma(t) \rightarrow_{p,R} t'$ (algunas veces se omite p, R y/o σ cuando están claros en el contexto). σ es comúnmente *el unificador mas general*[6] de $t|_p$ y 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 utilizados en un paso de narrowing. Denotamos por $t_0 \rightsquigarrow^*_\sigma t_n$ como una secuencia de pasos narrowing $t_0 \rightsquigarrow_{\sigma_1} \ldots \rightsquigarrow_{\sigma_n} t_n$ 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 descubrir las posibles instancias a diferentes términos. Dada una derivación narrowing $t_0 \rightsquigarrow^*_\sigma t_n$, decimos que t_n es un *valor* computado y σ es un resultado computado para t_0.

Ejemplo 2.2 Considere la siguiente definición de función "+":

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

[4] del inglés *red*ucible *exp*ression, i.e., expresión reducible [Ram07]

[5] *si y sólo si*

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

Dado el término $x + succ(zero)$, narrowing de 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 ejecuciones 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 narrowing pueden ser representadas por un *árbol* de ramificación finita (posiblemente infinita). Formalmente, dado un SRT $\mathcal{R}$ y un término encabezado por operación t, un *árbol narrowing* para t en $\mathcal{R}$ es un árbol que satisface las siguientes condiciones: (a) cada nodo del árbol es un término, (b) el nodo raíz es t, y (c) si s es un nodo del árbol, 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).

Para evitar tratar con estructuras de datos infinitas y cómputos innecesarios, varias estrategias narrowing *perezosas* han adoptado la creación del espacio de búsqueda dirigido–por–demanda [GLMP91, LLR93, MR92]. 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 narrowing perezosa.

Decimos que $s \leadsto_{p,R,\sigma} t$ es un *paso de narrowing necesario* sii $\sigma(s) \rightarrow_{p,R} t$ es un paso de *reescritura necesario* en el sentido de Huet y Lévy [HL92], i.e.,

en cada computo que va desde $\sigma(s)$ a una forma normal, sea $\sigma(s)|_p$ o a uno de sus *descendientes* al que debe ser reducido. Aquí, estamos interesados en una estrategia particular de narrowing, denotada por λ en [AEH00, Def. 13], la cual esta basada en el concepto de *árbol definicional* [Ant92] (una estructura jerárquica incluyendo las reglas de una definición de función, la cual es usada para guiar los pasos de narrowing necesario). Esta estrategia es básicamente equivalente a *narrowing perezoso* [MR92] donde los pasos narrowing son aplicados a la función más externa, si es posible, y las funciones interiores únicamente son reducidas por narrowing si su evaluación es *demandada* por un símbolo constructor en la parte izquierda de alguna regla (i.e., una típica estrategia de evaluación call-by-name). La principal diferencia es que narrowing necesario no calcula el *unificador más general* entre los redex seleccionados y la parte izquierda de la regla sino solo un unificador. Las referencias adicionales son requeridas para asegurar que solo cómputos "needed" son ejecutados (véase, e.g., [AEH00]), de este modo, narrowing necesario generalmente calcula un espacio de búsqueda menor.

Ejemplo 2.3 Considere nuevamente las reglas de la definición de función "$\leqslant$" del Example 2.1.1. 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*[7] (i.e., una variable libre o un término encabezado-por-constructor) dado que todas las reglas arborescentes 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 éste es referenciado a un término encabezado-por-constructor, aquí *zero* o $succ(x)$ y, dependiendo de esta referencia, 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 referencia de esté a un término encabezado-por-constructor (*zero* o

[7](i.e., a una expresión sin símbolo de definición de función en la raíz)[Han06]

$succ(x)$). Dependiendo de ésta referencia, procedemos como en los casos (1) o (2) de arriba.

Observemos que narrowing necesario está únicamente definido sobre términos encabezados-por-operación, i.e., una derivación de narrowing necesario se detiene 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 ser reducida a una secuencia de cómputos de forma normal en cabeza (véase [HP99]).

Para obtener una definición precisa de SRTs inductivamente secuenciales y narrowing necesario, el lector interesado puede encontrar definiciones detalladas en [Ant92, AEH00]). En lo subsiguiente, usamos *narrowing necesario* para referirnos a la estrategia particular λ definida en [AEH00, Def. 13].

Capítulo 3

Evaluación parcial *offline*

3.1. Introducción

Dado un programa y una llamada inicial (conteniendo algún dato conocido), el objetivo de un evaluador parcial es la construcción de un nuevo programa residual especializado para esta llamada. La 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) pueda ser extraído de esta representación. Por ejemplo, dado un programa $\mathcal{P}$ y una llamada inicial a función, $f(t, x)$, donde t es un dato de entrada conocido y x es una variable libre, un evaluador parcial trivial puede regresar un programa residual $\mathcal{P}' = \mathcal{P} \cup \{f_t(x) = f(t, x)\}$ incluyendo una versión especializada f_t de la función f. En tanto que la validez de este evaluador parcial trivial es obvia, es claro que no ocurre así con la eficiencia que pueda alcanzarse. Un reto en la evaluación parcial es la definición de técnicas para construir representaciones finitas del espacio de cómputo de un programa a partir de las cuales puedan extraerse programas residuales eficientes.

La evaluación parcial dirigida por Narrowing (NPE) es una poderosa técnica de especialización para sistemas de reescritura [AV02], i.e., para la 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, i.e., por defuncionalización [Rey98]; ésta estrategia es usada en algunas implementaciones de lenguajes lógico funcionales perezosos, tales como el sistema Portland-Aachen-Kiel Curry System (PAKCS [HeAE+04]) y el Münster Curry Compiler (MCC [Lux03]).

Aunque la NPE puede ser vista como un esquema tradicional de evaluación parcial para la especialización de programas, puede alcanzar un mayor poder de optimización, tal como consigue la deforestación [Wad90], eliminación de funciones de orden-superior (representadas en un ambiente de primer orden por defuncionalización), etc. Un evaluador parcial dirigido por narrowing esta actualmente integrado en el entorno de PAKCS para Curry (una evaluación experimental se encuentra en [AHV02]).

En el núcleo del esquema NPE encontramos un método para construir una representación finita en 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 ésta 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, i.e., 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 puede ser evitada, insertando una llamada a una variante previamente encontrada.

Los evaluadores parciales se clasifican en dos grandes categorías, *online* y *offline*, de acuerdo al momento en que son considerados los aspectos de terminación. Los evaluadores parciales online son usualmente más precisos ya que disponen de mayor información. Por ejemplo, el esquema original del 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 genera 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, i.e., sin riesgo de no terminación); después, la segunda etapa —la propia especialización— sólo debe obedecer las anotaciones y, por lo 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 incluyendo variables libres al momento de la ejecución. Por lo tanto, la primera etapa del esquema de evaluación parcial *offline* garantiza la terminación aún cuando todos los argumentos sean dinámicos (i.e., desconocidos).

Contribuciones. Las principales contribuciones de la primera aproximación offline a la evaluación parcial de programas lógico funcionales son las siguientes. Primero, se identifica una clase de SRTs, llamados *no-crecientes*, facilitando condiciones suficientes. Esto es un resultado interesante por si mismo ya que en la literatura no aparece 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*. A continuación, se define una relación extendida del narrowing necesario, *el narrowing necesario generalizante*, en la cual 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 un concepto general e informal de la aproximación a la evaluación parcial de programas (lógico) funcionales.

En este escenario, los datos de entrada al evaluador parcial son un un sistema de reescritura—un típico programa funcional de primer orden—y un llamado a función inicial, el cual suele contener algún dato conocido (los llamados

datos *estáticos*). Por ejemplo, considere el siguiente sistema de reescritura:

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

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* (i.e., especializando la función *add* con el primer argumento estático $succ(zero)$).

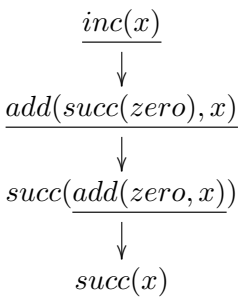
Ambos evaluadores parciales tanto *online* como *offline* deben construir alguna forma de *árbol de ejecución simbólica*. Se dice *simbólica* porque los términos pueden contener variables libres y, por lo tanto, a menudo se requiere un mecanismo de ejecución no estándar. Además, se obtiene una estructura 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]) comprenden algoritmos que iterativamente (1) toman una llamada a función, (2) ejecutan algunas evaluaciones simbólicas, y (3) de la expresión evaluada parcialmente extraen el conjunto de llamadas a función pendientes por resolver—los denominados *sucesores* de la llamada a función inicial—en la siguiente iteración del algoritmo. Observe 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 identificación de instancias para las variables libres (para una definición formal véase el capítulo 2). Además, en el escenario de la programación lógico funcional, puede ser utilizado 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):



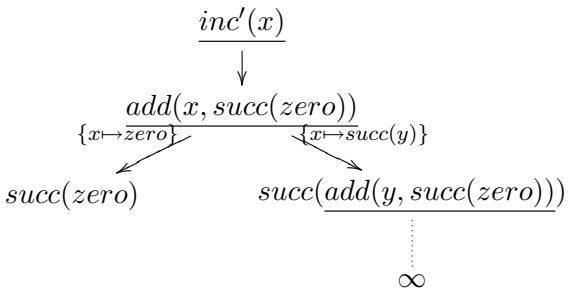
Aquí, no fue necesario calcular posibles instancias de las variables libres; por lo tanto, tenemos una evaluación determinista. El programa residual asociado puede ser fácilmente extraído a partir de las computaciones que van de la raíz a las hojas en el árbol de ejecución simbólica. Del ejemplo anterior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

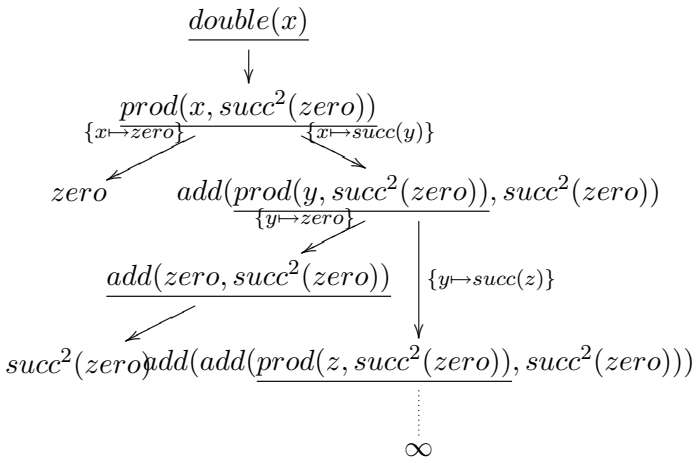
$$\begin{array}{rcl} inc'(x) & \rightarrow & add(x, succ(zero)) \\ add(zero, succ(zero)) & \rightarrow & succ(zero) \\ add(succ(y), succ(zero)) & \rightarrow & succ(add(y, succ(zero))) \end{array}$$

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 ser garantizado cuando las computaciones simbólicas son *cuasi-terminantes*, i.e., cuando se obtiene un número finito de términos diferentes —módulo renombramiento de variables. Note 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. Considere la siguiente definición de función:

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

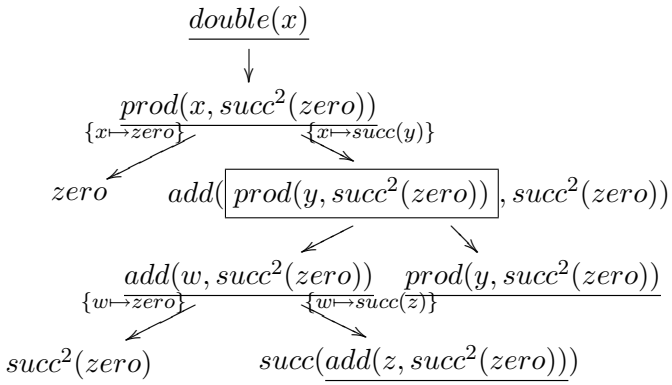
Dada la llamada inicial $double(x)$, el árbol simbólico asociado es infinito (se usa $succ^2(zero)$ como una abreviación para $succ(succ(zero))$):



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 partial *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 *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 ser garantizada generalizando 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 $succ^2(zero)$ (i.e., no contienen llamadas a función) o contienen una llamada a función que es una variante de una llamada a función previa dentro del árbol (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, puede ser extraído 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

Narrowing es usado como mecanismo de computación simbólica en el esquema NPE 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 NPE original garantiza que los cómputos son cuasi-terminantes en el esquema *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:

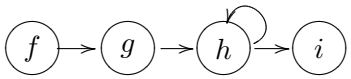
Definición 1 (grafo de dependencias funcionales) *Dado un SRT $\mathcal{R}$, su grafo de dependencias funcionales, en símbolos $\mathcal{G}(\mathcal{R})$, contiene nodos etiquetados con los símbolos de función en $\mathcal{D}$ y existe un arco del nodo f al nodo g sii hay una llamada a la función g desde la parte derecha de alguna regla en la definición de función f.*

Definición 2 (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 Considere 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*, i.e., una función que siempre *consume* sus parámetros o los deja inalterados:

Definición 3 (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 (i.e., 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 mantienen las siguientes relaciones:

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

i.e., 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-terminate para un conjunto de términos T con respecto a narrowing necesario* sii son cuasi-terminantes todas las derivaciones de narrowing necesario que provienen de los términos en T. Ahora, se ofrece una condición suficiente para cuasi-terminación de SRTs:

Definición 4 (SRT no-crecientes) *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 (i.e., 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 5 (SRT cuasi-terminante [RSV05]) *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, considere el siguiente SRT:

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

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 efectivas 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 solo aseguran que *algunos* parámetros decrezcan (pero no todos ellos), lo cual no es suficiente en nuestro contexto donde todos los parámetros pueden ser desconocidos (i.e., variables libres). La linealidad por la derecha es un requerimiento esencial inclusive para funciones muy simples. Por ejemplo, considere 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*:

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

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

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

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 considere 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 el modo *online* (véase, e.g., [AHV02, AV02, AFV98]), i.e., durante el proceso de evaluación parcial se utilizan operadores de generalización y test de terminación apropiados para garantizar que sólo sean computados 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 tiempo y espacio); por lo tanto, no es muy

adecuado para especializar problemas realistas. Para remediar esta situación, en [RSV05] se introduce un método NPE más rápido el cual 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 5). Este esquema produciría una herramienta NPE muy rápida—puesto que solo requeriría verificaciones de igualdad módulo renombramiento de variables—aunque desafortunadamente 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, i.e. los programas inductivamente secuenciales, la clase de programas para la cual NPE fue originalmente definida, y se define un algoritmo que *anote* las expresiones que puedan causar la no cuasi-terminación de las derivaciones del *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:

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

i.e., dado un término anotado $t \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$, la expresión $gen(t) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ regresa 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 $aterms(t) \subseteq \mathcal{T}(\mathcal{F}_\bullet, \mathcal{V})$ regresa 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 sencillos:

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

La siguiente definición introduce el concepto de transformación para anotar programas inductivamente secuenciales. Intuitivamente, se revisaran las partes derechas de las reglas y se anotarán aquellos argumentos correspondientes a subtérminos encabezados por función definida, i.e., 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 6 ($ann(\mathcal{R})$) *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 Fig. 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:

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

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

Figura 3.1: Funciones Auxiliares qs y qs'

- 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.

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 han modificado.

Note que la Definición 6 es indeterminista ya que no precisa qué variable no debe ser anotada 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 no debe ser anotada.

Ejemplo 3.3 Considere 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:

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

Observe 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

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

i.e., $g(x, w_1)$, $f(x, w_2)$, 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 7 (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)

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

(decomposición)

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

Una derivación de narrowing necesario generalizante $s \rightsquigarrow_\sigma^* t$ esta compuesta de pasos de *narrowing necesario* (para términos no anotados 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 etiquetando los pasos propios de *narrowing necesario*. Note que, *narrowing necesario* únicamente computa una *forma normal en cabeza* (i.e., una variable o un término encabezado por constructor), la regla de descomposición se requiere para

asegurar que todas las posibles funciones internas sean, a la larga, evaluadas parcialmente. Algunos ejemplos del cálculo de narrowing necesario generalizante pueden ser examinados en la siguiente sección.

Notemos 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 8 (SRT anotado cuasi-terminante [RSV05]) *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, el cual esta basado en la anotación de programas procesados con narrowing necesario generalizante. Luego, se ilustra el nuevo esquema por medio de algunos ejemplos seleccionados. Finalmente, se presenta 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, en 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 narrowing necesario generalizante considerado, donde la función *rann* simplemente elimina las ocurrencias de símbolo "$\bullet$" 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 representan 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 9 (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.

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

Ejemplo 3.4 Considere 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 8) garantizan 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 $\{s \mid s \leadsto_\sigma t\}$, i.e., para los términos a los cuales 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 10 (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 Considere 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 regresa el término renombrado $g'(x, f(a, s'(b)))$.

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

Definición 11 (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 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 $\{s \mid s \rightsquigarrow_\sigma s' \in \tau\}$.*

[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.

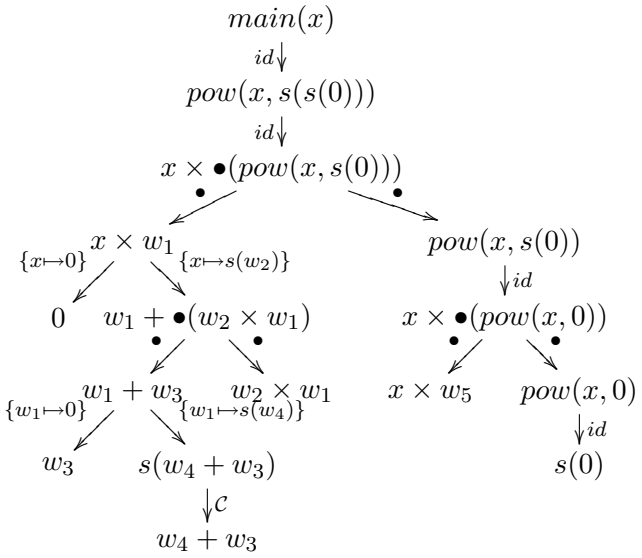


Figura 3.2: Árbol de narrowing necesario generalizante para $main(x)$

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. De hecho, el prototipo implementado sigue este refinamiento.

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

Teorema 12 *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 11 siempre termina al computar un SRT inductivamente secuencial $\mathcal{R}'$ tal que narrowing necesario calcula los mismos resultados para $f(\overline{x_n})$ en $\mathcal{R}$ y en $\mathcal{R}'$.*

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 del programa. El primer ejemplo ilustra el uso del método NPE offline para especialización de programas. Considere el siguiente SRT

el cual ha sido anotado de acuerdo a las Definición 6:

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

Dado el término inicial $main(x)$, se construye el árbol de narrowing necesario generalizante representado en la Figura 3.2. Así pues, el SRT residual asociado contiene las siguientes reglas:

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

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

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

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

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

ya que las funciones pow_2, pow_1, y pow_0 son únicamente funciones intermedias (i.e., sólo hay una llamada a cualquiera de ellas). 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*.

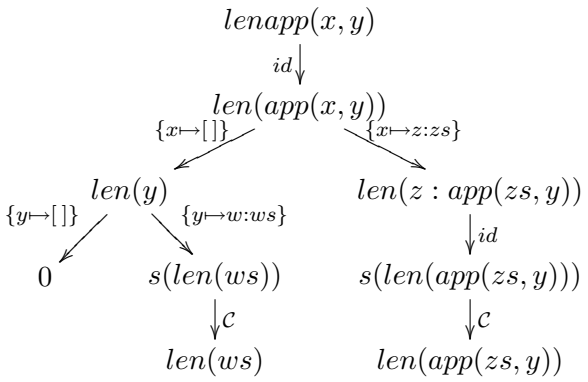


Figura 3.3: Árbol de narrowing necesario generalizante para $lenapp(x, y)$

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

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

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:

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

el SRT asociado es como sigue:

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

[2] Un post-proceso estándar de desplegado conocido como: *transition compression* [JGS93]

junto con la definición de la función original *len*. Y, tal como en el ejemplo anterior, una simple transformación de post-desplegado podría eliminar la función intermedia $la2$. Nótese que el SRT residual esta completamente deforestado (i.e., 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], i.e., 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 6, incluye la definición de la conocida función de orden superior *map*:

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

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

Observe que, en este ejemplo, se ha anotado la ocurrencia que está más a la izquierda de la variable f 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. Note 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).

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

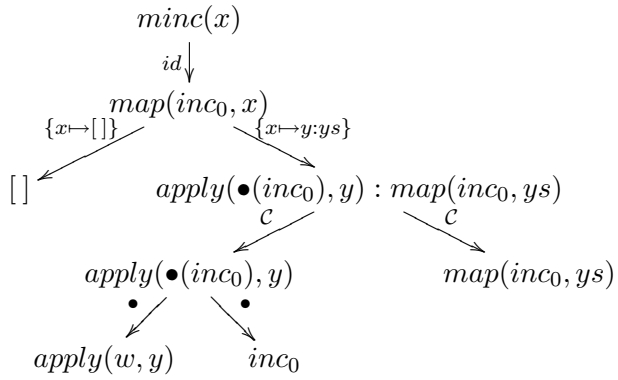


Figura 3.4: Árbol de narrowing necesario generalizante para $minc(x)$

Dado el siguiente renombramiento independiente:

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

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

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

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

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

donde el operador de aplicación explícito *apply* ya no es necesario. Note que esta transformación constantemente 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 estándares de comparación 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-in's), etc.

El cuadro 3.1 muestra los resultados de una comparativa en la que se han considerado los siguientes ejemplos:

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

`allones`: El objetivo de este ejemplo es producir automáticamente una función nueva que transforme todos los elementos 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, los 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 aprox-

imación *offline* propaga la información lo mismo que el esquema *online* (el cual 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 (i.e., el programa de entrada al intérprete ha sido completamente evaluado).

3.6. Trabajo relacionado y discusión

A pesar de la relevancia de *narrowing* como mecanismo de computación simbólica, se encontró 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*, i.e., que todos los argumentos sean 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

[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

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 presento 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, se ha presentado una nueva caracterización para SRTs que asegura la cuasi-terminación de computaciones de narrowing necesario. Este es un problema difícil de interés independiente que no ha sido atacado anteriormente. Dado que la clase de SRTs considerada es muy restrictiva, se consideró entonces los programas inductivamente secuenciales—una clase mucho más amplia—y se introdujo un algoritmo que anota aquellos subtérminos que pueden causar la no cuasi-terminación del narrowing necesario. También se introdujo una extensión generalizante del narrowing necesario la cual esta dirigida por anotaciones agregadas al programa. Finalmente, se describe 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, nuestros desarrollos podrían ser fácilmente extendidos a sistemas inductivamente secuenciales *solapantes* y narrowing inductivamente secuencial [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 si-

guientes dos reglas individuales:

$$\begin{aligned} f(x) &\rightarrow t_1 \\ f(x) &\rightarrow t_2 \end{aligned}$$

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 esta basado en los grafos de cambio de tamaño (*grafos size-change* [LJBA01] los cuales 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 SRT's no crecientes por los SRT's EP-terminantes [ARSV06] logrados a partir de la extensión del esquema de los *grafos size-change* de programas funcionales a NPE con programas lógico funcionales .

Por otro lado, el algoritmo de anotación para SRTs es independiente del término considerado para la evaluación parcial. Esto significa que un SRT necesita ser anotado sólo una vez, y entonces puede ser evaluado parcialmente, con respecto a diferentes términos, sin calcular nuevas anotaciones. Sin embargo, esto también significa que no estamos explotando la estructura conocida del término considerado para la evaluación parcial. Por lo tanto, sería interesante estudiar la combinación de la primera etapa con un tradicional proceso de análisis de tiempo de enlace (BTA, del inglés binding-time analysis). 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., [CGLH05b, LJVB04]).

Capítulo 4

Optimizando la evaluación parcial *offline*

4.1. Introducción

Narrowing [Sla74] extiende el principio de reducción de los lenguajes funcionales reemplazando emparejamiento por unificación. La evaluación parcial dirigida por narrowing [AV02] es una técnica potente de especialización para el componente de primer orden varios lenguajes funcionales y lógico funcionales tales como Haskell o Curry.

En el capítulo anterior se identificaron una clase de sistemas de reescritura cuasi-terminates (con respecto a narrowing necesario) los cuales son llamados *no crecientes*. Esta caracterización es puramente sintáctica y muy fácil de verificar, aunque también muy restrictiva para ser útil en la práctica. Así pues, en el capítulo anterior se presento un esquema offline para NPE tomando en cuenta 1) la anotación de expresiones de programa *que violan la propiedad no creciente*, y 2) considerando una ligera extensión del narrowing necesario para ejecutar computaciones parciales y que los subtérminos anotados sean *generalizados* en tiempo de especialización (lo cual asegura la terminación del proceso).

En este capítulo, se mejora la caracterización de los sistemas de reescritura no crecientes mediante los *grafos size-change* [LJBA01], los cuales rastrean los cambios en la talla de los parámetros en las llamadas a función. Más detalladamente, se usa la información de los grafos size-change para identificar una forma particular de cuasi-terminación, i.e., que sólo pueden ser producidos 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 binding-time estándar (BTA) para contar con la información sobre qué argumentos de función son *estáticos* (y por lo tanto conocidos) o *dinámicos*. Cuando la información coleccionada del uso combinado de los grafos size-change y del BTA no permite inferir que el sistema de reescritura cuasi-termina, se procede como en el capítulo anterior y se anotan los subtérminos problemáticos para ser generalizados en tiempo de evaluación parcial. Al final del presente capítulo se presenta la comparación de algunos estándares de programación basados en la implementación de este nuevo análisis.

El análisis que se muestra comparte muchas similitudes con [GJ05], donde se utiliza un análisis de cuasi-terminación basado en los grafos size-change para garantizar la terminación de un evaluador parcial offline de primer orden en programas funcionales. Sin embargo, no es sencillo trasladar el esquema de Jones y Glenstrup a NPE y programas lógico funcionales. Por ejemplo, NPE propaga los enlaces de variables hacia adelante en los cómputos parciales, de esta forma (comparado con [GJ05]) son necesarios algunos requerimientos adicionales para asegurar la cuasi-terminación.

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

En esta sección, se retomarán algunos conceptos básicos sobre los grafos size-change a partir de [LJBA01, TG05]; después, introduciremos nuestra nueva aproximación para garantizar cuasi-terminación.

En adelante, diremos que un orden "$\succ$" es *cerrado bajo sustituciones* (o *estable*) si $s \succ t$ implica $\sigma(s) \succ \sigma(t)$ para todo término $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ y sustitución σ.

Definición 13 (par reducción) *Decimos que $(\succsim, \succ)$ es un par reducción si $\succsim$ es un cuasi-orden y $\succ$ es un orden bien fundamentado sobre términos donde ambos $\succsim$ y $\succ$ son cerrados bajo sustituciones y compatibles (i.e., $\succsim \circ \succ \subseteq \succ$ o $\succ \circ \succsim \subseteq \succ$ pero $\succsim \subseteq \succ$ no es necesario).*

Definición 14 (grafos size-change) *Sea $(\succsim, \succ)$ un par reducción. Para cada regla $f(\overline{s_n}) \to r$ de un SRT $\mathcal{R}$ y cada subtérmino $g(\overline{t_m})$ de r donde $g \in \mathcal{D}$, definimos un grafo size-change como sigue. El grafo tiene n nodos de salida marcados con $\{1_f, \ldots, n_f\}$ y m nodos de entrada marcados con $\{1_g, \ldots, m_g\}$. Si $s_i \succ t_j$, entonces hay un arco marcado con $\succ$ y dirigido desde el nodo i_f al*

nodo j_g. *En caso contrario, si* $s_i \succsim t_j$, *entonces hay un arco marcado con* $\succsim$ *desde* i_f *a* j_g.

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

Para analizar los bucles de un programa se introducen los conceptos de *concatenación y multigrafo maximal* en la siguiente definición:

Definición 15 (multigrafo, concatenación, multigrafo maximal) *Cada grafo size-change de* $\mathcal{R}$ *es un multigrafo de* $\mathcal{R}$ *y si* $G = (\{1_f, \ldots, n_f\}, \{1_g, \ldots, m_g\}, E_1)$ *y* $H = (\{1_g, \ldots, m_g\}, \{1_h, \ldots, p_h\}, E_2)$ *son multigrafos de* $\mathcal{R}$ *con respecto al mismo par reducción* $(\succsim, \succ)$, *entonces la concatenación* $G \cdot H = (\{1_f, \ldots, n_f\}, \{1_h, \ldots, p_h\}, E)$ *es también un multigrafo de* $\mathcal{R}$. *Para* $1 \leq i \leq n$ *y* $1 \leq k \leq p$, E *contiene un arco desde* i_f *a* k_h *sii* E_1 *contiene un arco desde* i_f *a algún* j_g *y* E_2 *contiene un arco desde* j_g *a* k_h. *Si hay un determinado* j_g *donde el arco de* E_1 *o* E_2 *está etiquetado con "*$\succ$*", entonces el arco en* E *es etiquetado también con "*$\succ$*". De otro modo, se etiqueta con "*$\succsim$*".*

Un multigrafo G *de* $\mathcal{R}$ *es llamado un multigrafo maximal de* $\mathcal{R}$ *si sus nodos de entrada y salida están ambos etiquetados con* $\{1_f, \ldots, n_f\}$ *para alguna función* f *y si éste es idempotente, i.e.,* $G = G \cdot G$.

A grandes rasgos, dado el conjunto de los grafos size-change de un programa, primero calculamos su cierre transitivo bajo el operador de concatenación, de esta manera se produce un conjunto finito de multigrafos. Después, sólo necesitamos concentrarnos en los multigrafos maximales porque éstos representan los bucles del programa.

Ejemplo 4.1 *Considere el siguiente SRT simbolizando un programa que devuelve los elementos de una lista en posición invertida respecto a la lista original:*

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

donde "$[\,]$*" y ":" son constructores de lista. En este ejemplo, consideramos un par reducción* $(\succsim, \succ)$ *definido como sigue:*

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

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

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

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

donde $c \in \mathcal{C}$ es un término constructor con aridad $n \geqslant 0$. Entonces, los grafos size-change correspondientes para este programa son los siguientes:

$$G_1: \quad 1_{rev} \xrightarrow{\succ} 1_{rev} \qquad G_2: \quad 1_{rev} \begin{array}{l} \searrow^{\succ}_{\sim} \\ \searrow \end{array} \begin{array}{l} 1_{app} \\ 2_{app} \end{array} \qquad G_3: \quad \begin{array}{l} 1_{app} \xrightarrow{\succ} 1_{app} \\ 2_{app} \xrightarrow{\succsim} 2_{app} \end{array}$$

donde G_1 y G_3 son, así mismo, los multigrafos maximales del programa.

Definición 16 (EP-terminación, SRT EP-terminante) *Un cómputo de narrowing necesario es EP-terminante si el desplegado de funciones genera sólo un número finito de llamadas a función diferentes (i.e., redexes) módulo renombramiento de variables. Un SRT es EP-terminante si cada posible computación por narrowing necesario es EP-terminante.*

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

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

Aunque esta derivación consta de un número infinito de términos diferentes, existe sólo un número finito de llamadas a función diferentes módulo renombramiento de variables. Afortunadamente, este hecho es suficiente para asegurar la terminación en muchos esquemas de evaluación parcial.

En adelante, consideramos que tenemos disponible el resultado de un análisis binding-time (BTA) monovariante. Hablando informalmente, dado un SRT y la información sobre cuáles parámetros de la llamada a función inicial son estáticos y cuáles dinámicos, un BTA propaga dicha cualidad de los argumentos y realiza una correspondencia congruente con la lista de valores estático/dinámico que entrega el BTA, correspondientes a los argumentos de cada

función del programa. Aquí, consideramos que un parámetro estático es absolutamente conocido en tiempo del proceso de especialización (por lo tanto es un valor básico), mientras que un parámetro dinámico es posiblemente desconocido en tiempo de especialización.

A continuación, requeriremos que el componente $\succsim$ de un par reducción $(\succsim, \succ)$ sea de *particionamiento finito*[1], i.e., que el conjunto $\{s \mid t \succsim s\}$ debe contener un número finito de términos no variantes para cualquier término t.

El siguiente teorema establece las condiciones suficientes para asegurar la propiedad de EP-terminación de SRTs:

Teorema 17 *Sea $\mathcal{R}$ un SRT y $(\succsim, \succ)$ un par reducción. $\mathcal{R}$ es EP-terminante si cada multigrafo maximal 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

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

Además, requerimos que $\mathcal{R}$ sea lineal por la derecha con respecto a variables dinámicas, i.e., no se permiten ocurrencias repetidas de la misma variable dinámica en la parte derecha de la regla.

La última condición sobre la linealidad por la derecha es requerida para evitar situaciones como la siguiente: dado el SRT

$$\begin{array}{lcl} double(x) & \to & add(x, x) \\ add(zero, y) & \to & y \\ add(succ(x), y) & \to & succ(add(x, y)) \end{array}$$

a pesar de que *double* y *add* parecen claramente terminates (y por lo tanto cuasi-terminantes), es posible que se llegue a derivar la siguiente computación infinita:

$$\begin{array}{lll} \underline{double(x)} \rightsquigarrow \underline{add(x, x)} & \rightsquigarrow_{\{x \mapsto succ(x')\}} & succ(\underline{add(x', succ(x'))}) \\ & \rightsquigarrow_{\{x' \mapsto succ(x'')\}} & succ(succ(\underline{add(x'', succ(succ(x'')))})) \\ & \rightsquigarrow_{\{x'' \mapsto succ(x''')\}} & \cdots \end{array}$$

la cual no es cuasi-terminante ni EP-terminante.

[1] del concepto *bounded* de [DDL+97]

En lugar de requerir programas fuente que satisfagan las condiciones del teorema anterior, utilizamos este resultado para para definir una nuevo procedimiento de anotación de programa que asegure EP-terminación para NPE offline.

Básicamente, se toma cada símbolo de función f/n tal que f tiene un multigrafo maximal, y se ejecuta una de las siguientes acciones:

1) si las condiciones del Teorema 17 se mantienen, no se agregan anotaciones;
2) de otro modo, tenemos dos posibilidades:

- si la función f tiene un parámetro estático (el cual no decrece en el multigrafo maximal), digamos que el i-esimo parámetro, entonces se anota el i-esimo argumento de cada llamada a función f en el programa;
- de otra forma, si todos los parámetros de f son dinámicos, entonces se anota el j-esimo argumento de cada llamada a función f en el programa, donde j varia dentro del rango de los parámetros de f que no tengan un arco $j_f \xrightarrow{R} j_f$, etiquetado con $R \in \{\succsim, \succ\}$;

3) finalmente, si hay más de una ocurrencia de la misma variable dinámica (que no se encuentre anotada) en la parte derecha de una regla de programa, entonces se anotan todas las ocurrencias excepto una (e.g., la que se encuentre más a la izquierda).

4.3. 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 [RSV05]. Este evaluador parcial ha sido implementado así mismo en Curry [Han06]. En esta implementación solo se ha considerado un subconjunto de Curry. La extensión del resto de las características de Curry (e.g., restricciones, funciones de orden superior, builtin's, etc.) está en proceso. Los programas fuente del evaluador parcial 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. Por cada ejemplo, mostramos el tamaño (en bytes) de cada programa (codesize), el tiempo de ejecución del programa residual especializado con el evaluador parcial offline y que usa el procedimiento de anotación simple (simple peval), el tiempo

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	425	1.12
`gauss`	2904	308	320	0.96	252	1.22
`inc_list`	817	937	834	1.12	730	1.28
`inser_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: Resultados de estándares de programación

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 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). Los programas 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. 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 (i.e., 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.

Capítulo 5

Implementación del evaluador parcial *offline*

En éste capítulo presentamos un procedimiento de anotación basado en el análisis de cuasiterminación del capítulo anterior. Enseguida introducimos un algoritmo de especialización que distingue dos niveles diferentes. El nivel *global* garantiza que el número de las diferentes funciones especializadas se mantiene finito. El nivel *local* toma un llamado a función y construye una evaluación finita (posiblemente parcial) de ésta llamada. El método resultante es offline puro y de esta manera muy eficiente.

Finalmente, también discutimos un algoritmo híbrido que incluye algunos chequeos simples durante la evaluación parcial, de forma que la calidad de los programas residuales pueda ser mejorada.

5.1. 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 traducidas a su representación flat, nuestro enfoque cubre recientes propues-

tas para la programación multi-paradigma lógico funcional. La sintaxis para programas en la representación flat es como sigue:

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

Aquí, escribimos $\overline{o_n}$ para la secuencia de objetos $o_1, \ldots, o_n$. De esta forma, un programa $\mathcal{R}$ consiste de una secuencia de definiciones de función D tales que la parte izquierda es lineal y tiene únicamente variables como argumentos, i.e., el emparejamiento de patrones se compila a expresiones *case*. La parte derecha de cada definición de función es una expresión e compuesta por variables ($\mathcal{X}$), constructores ($\mathcal{C}$), llamadas a función ($\mathcal{F}$), y expresiones *case* para emparejamiento de patrones. Las variables son denotadas por $x, y, z \ldots$. La forma general de una expresión case es:

$$\begin{array}{rcl} (f)case\ e\ of\ \{ & c_1(\overline{x_{n_1}}) \rightarrow e_1 & ; \\ & \ldots & ; \\ & c_k(\overline{x_{n_k}}) \rightarrow e_k & \} \end{array}$$

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 ésta 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

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

Figura 5.1: Función auxiliar *pairs*

ser escrita en la representación flat mediante la siguiente regla:

```
app (x, y) = fcase x of {
                 [ ] → y;
           (z : zs) → z : app (zs, y) }
```

La semántica operacional de los programas flat esta basada en el cálculo LNT (Lazy Narrowing with definitional Trees) [HP99]. En la Sección 5.3.2 representamos una ligera extensión de esta semántica para ejecutar computaciones en tiempo de evaluación parcial.

5.2. Análisis de cuasi-terminación y anotación de programas

Primero adaptamos el análisis de cuasi-terminación del capítulo anterior, introducido originalmente para sistemas de reescritura de términos, al lenguaje flat. Posteriormente presentamos un procedimiento de anotación que está basado en este análisis de cuasi-terminación.

Para adaptar el concepto original de grafo size-change a programas flat; tomamos las definiciones de: **par reducción**, **grafo size-change**, **multigrafo**, **concatenación**, y **multigrafo maximal** del capítulo anterior. A continuación especificamos la siguiente definición auxiliar:

Definición 18 (par llamada) *Dada una definición de función* $f(\overline{x_k}) = e$*, tenemos que los pares* $pairs(f(\overline{x_k}), e)$ *es el conjunto asociado de* par llamada*, el cual es inductivamente definido como se muestra en Fig. 5.1.*[1]

[1] Aquí, asumimos que las expresiones *case* solo 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].

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

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

Figura 5.2: Ejemplo de deforestación `applast` y sus **par llamadas**

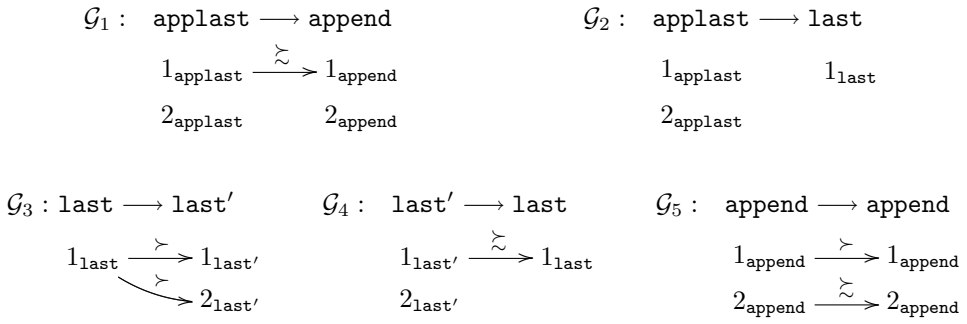


Figura 5.3: Grafos size-change de `applast`

Ejemplo 5.1 *Considere el ejemplo de deforestación mostrado en Fig. 5.2 y su conjunto de par llamada asociado 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 acuerdo a los conceptos referidos, tenemos cinco grafos asociados a las cinco par llamadas mostradas in Fig. 5.3. Finalmente, al igual que en el capítulo anterior, calculamos el cierre transitivo de los grafos size-change bajo el operador de concatenación y obtenemos los tres multigrafos idempotentes mostrados in Fig. 5.4.

Centraremos ahora nuestra atención en un concepto de terminación denominado *EP-terminación* especificado en la Definición 16 (un caso particular de cuasi-terminación), adaptado al contexto de los programas flat.

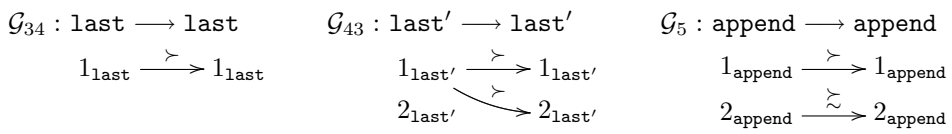


Figura 5.4: Multigrafos idempotentes de `applast`

Definición 19 (EP-terminación) *Una computación es EP-terminante si el desplegado de funciones que origina, genera sólo un número finito de llamadas a función diferentes. Un programa flat es EP-terminante si cada posible computación es EP-terminante.*

La siguiente definición es una extensión directa del Teorema 17 derivado como resultado del capítulo anterior e inducido a los programas flat.

Teorema 20 *Sea $\mathcal{R}$ un programa flat y sea $(\succsim, \succ)$ un par reducción. $\mathcal{R}$ es EP-terminante con respecto a cualquier término lineal si cada multigrafo idempotente asociado a una 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 sea estático, o

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

También, requerimos que $\mathcal{R}$ sea lineal por la derecha con respecto a variables dinámicas , i.e., sin ocurrencias repetidas de la misma variable dinámica en la parte derecha de la regla.

5.2.1. 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 $\mathcal{R}$, el par reducción $(\succsim, \succ)$,[2] los multigrafos idempotentes de $\mathcal{R}$, y el resultado del BTA como parámetros globales.

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

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

Figura 5.5: Función de anotación ann^u

Definición 21 (anotación de programa) *Un programa se anota reemplazando cada regla* $f(\overline{x_n}) = e$ *en* $\mathcal{R}$ *por una nueva regla definida como* $f(\overline{x_n}) = ann^l(ann^g(ann^u(e)))$[3]. *La función* ann^u *se usa para agregar anotaciones de* unfolding *como se muestra en la Fig. 5.5: 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*).*[4]

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 Fig. 5.6, donde* $\mathcal{F}^{an}$ *indica el dominio de las funciones anotadas.*

Finalmente, la función ann^l *es usada para linealizar expresiones tal como se muestra en la Fig. 5.7.*

Ejemplo 5.2 *Considere el programa* `applast` *mostrado en la Fig. 5.2. Consideremos su especialización con respecto al valor conocido del primer argumento de* `applast`*. En este caso, el BTA podría regresar 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 dinámico.*

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

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

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

Figura 5.6: Función de anotación ann^g

$$ann^l(e) = \begin{cases} (f)case\ x\ of\ \{\overline{p_k \to ann^l(e_k)}\} & \text{if } e \equiv (f)case\ x\ of\ \{\overline{p_k \to e_k}\} \\ linear(e) & \text{de otra manera} \end{cases}$$

donde la función *linear* anota cada ocurrencia de una variable dinámica no anotada aún excepto una (e.g., la de más a la izquierda)

Figura 5.7: Función de anotación ann^l y la función auxiliar *linear*

Nuestro procedimiento de anotación devuelve

$$\begin{aligned}
&\texttt{applast(xs,x)} = \texttt{last}^{\texttt{m}}(\texttt{append}^{\texttt{u}}(\texttt{xs}, [\texttt{x}])) \\
&\texttt{last(xs)} \quad = \texttt{fcase xs of} \\
&\qquad \{\ (\texttt{y} : \texttt{ys}) \to \texttt{last}'^{\texttt{m}}(\texttt{ys}, \texttt{y})\ \} \\
&\texttt{last}'\texttt{(ys,y)} \quad = \texttt{fcase ys of} \\
&\qquad \{\ [\,] \to [\texttt{y}]; \\
&\qquad\ \ (\texttt{w} : \texttt{ws}) \to \texttt{last}^{\texttt{m}}(\texttt{w} : \texttt{ws})\ \ \} \\
&\texttt{append(xs,ys)} = \texttt{fcase xs of} \\
&\qquad \{\ [\,] \to \texttt{ys}; \\
&\qquad\ \ (\texttt{w} : \texttt{ws}) \to \texttt{w} : \texttt{append}^{\texttt{u}}(\texttt{ws}, \texttt{ys})\ \}
\end{aligned}$$

5.3. 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.

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

Figura 5.8: Procedimiento genérico para NPE

El procedimiento genérico se muestra en la Figura 5.8. 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 (i.e., llamadas a función), el nivel de control local aplica un operador de desplegado *unfold* de tal forma que regresa un conjunto de reglas residuales como resultado (ver Sección 5.3.2). El operador de desplegado debe asegurar que el proceso de desplegado es 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 el conjunto finito de términos encabezados por operación T_i y entonces agrega apropiadamente el conjunto de subtérminos encabezados por operación de las llamadas ya desplegadas, el cual esta denotado por $\mathcal{R}'_{\mathit{calls}}$. Puede ser necesario evaluar adicionalmente el nuevo conjunto T_{i+1}; de esta forma, el proceso se repite iterativamente mientras nuevos términos sean introducidos.

Observe que este procedimiento no regresa 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 regresado y, después, renombrar la reglas usando una fase estándar de post-desplegado (ver Sección 5.3.2)

5.3.1. Control global

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

do de acuerdo a la Definición 21 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 lo que sigue, 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 22 (operador de abstracción) *Sean T_1, T_2 conjuntos finitos de términos. Entonces, $abstract(T_1, T_2)$ esta definido como sigue:*

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

5.3.2. Control local

Ahora, introducimos nuestro operador de desplegado. Éste está dirigido por 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 cual 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 Fig. 5.9. Primero, note que los símbolos "$[\![$" y "$]\!]$" en una expresión como $[\![e]\!]$ son puramente sintácticos (i.e., 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 ignorar las anotaciones de generalización. Observe 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

Unfold

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

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

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

Figura 5.9: El cálculo RLNT offline

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]).

Observe que los cómputos RLNT con un programa anotado son siempre finitos como una consecuencia simple del Teorema 20.

Nuestro operador de desplegado puede ser definido como sigue:

Definición 23 (operador de desplegado) *Dado un programa flat $\mathcal{R}$ y un conjunto de términos T, tenemos que $unfold(T, \mathcal{R}) =$*

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

Observe que el operador de desplegado no regresa 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 5.3 *Considere nuevamente el programa anotado del Ejemplo 5.2. Dado el conjunto inicial de llamadas* $T_0 = \{\texttt{applast}([\texttt{1}], \texttt{x})\}$ *la evaluación parcial produce la siguiente secuencia de llamadas (de acuerdo al algoritmo de la Figura 5.8):*

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

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:

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

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

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

5.3.3. 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:

Unfold

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

Memo

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

Figura 5.10: El cálculo RLNT híbrido

benchmark	original runtime	Híbrido spec. time	Híbrido runtime spec.	Híbrido speedup	Offline spec. time	Offline runtime spec.	Offline speedup	Online spec. time	Online runtime spec.	Online 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 5.1: Resultados de Benchmarks

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 Fig. 5.10. 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 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 5.1 muestra los resultados de una evaluación experimental de ambas estrategias. En general, logran mejoras similares y son igualmente eficientes.

Ejemplo 5.4 *Considere nuevamente el ejemplo 5.2 pero tomando en cuenta el refinamiento del control local. Entonces, tenemos el siguiente computo.*

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

así pues el resultado renombrado es:

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

i.e., la llamada fue completamente desplegada.

5.4. 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 5.3.2 y otro para la especialización *híbrida* expuesta en la sección 5.3.3. Así mismo hemos implementado las dos aproximaciones de especialización asociadas.

Los dos módulos de anotación usan el resultado de un mismo análisis binding-time para saber qué argumentos son estáticos y cuales 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 de las llamadas a función que no tengan un arco asociado en el multigrafo
- anotar con GEN las ocurrencias múltiples de la misma variable excepto una.

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:

Cuadro 5.2: Acciones de la Especialización

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

- 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 5.2: En el caso la aproximación *offline pura*, 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 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, i.e., 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 subtérminos 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 se han adaptado los ejemplos para ambas clases de especializadores. En el cuadro 5.1 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. Así mismo, 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`

5.5. Conclusiones

En este capítulo, hemos introducido las estrategias de control apropiadas para diseñar un evaluador parcial offline dirigido por narrowing. Se ha iniciado una implementación del evaluador parcial siguiendo las ideas presentadas hasta el momento, siendo alentadores los resultados preliminares.

Capítulo 6

Conclusiones y trabajo futuro

A continuación concluimos resumiendo las principales contribuciones de la tesis e indicando algunas líneas de trabajo futuro.

6.1. Conclusiones

Hemos presentado la evaluación parcial offline dirigida por narrowing como una técnica de especialización de programas lógico funcionales de primer orden. Se ha corroborado que el esquema de evaluación parcial online es más preciso aunque es relativamente más oneroso en el proceso de especialización que el esquema de evaluación parcial offline. Esta última característica indica que el esquema offline NPE resulta más adecuado para la especialización de intérpretes [Jon04] o la compilación por evaluación parcial.

Hemos presentado, además, una mejora en la caracterización de sistemas de reescritura *no crecientes* mediante los *grafos size-change* y el uso de un BTA estándar automático. Aunado a esto, se presentaron dos procedimientos de anotación del cual uno de ellos produce un programa anotado muy intuitivo para especializar resultando un método de especialización *offline puro*, así como un algoritmo de especialización híbrido. Brevemente, las principales contribuciones de la tesis son:

1. Una fase de anotación automática.
   Dado que la fase de anotación en el primer esquema de evaluación parcial produce un programa anotado en lenguaje intermedio FlatCurry, lo cual no permite observar las anotaciones en lenguaje fuente Curry, se dio a la tarea de automatizar dicho proceso desarrollando un módulo de anotación para el lenguaje intermedio AbstractCurry. Dicho proceso genera

un programa anotado en AbstractCurry, el cual es traducido fácilmente a Curry y es así susceptible de ser analizado antes de ser especializado.

2. Una aproximación offline NPE mejorada.
   Para lograr este propósito se incluyó el nuevo procedimiento de anotación dentro del evaluador parcial (EP) offline para programas Curry especificado en [RSV05] utilizando el BTA automático. En este nuevo esquema se aplicaron las anotaciones de generalización indicadas en la Sección 4.2 considerando además el Teorema 17 de la misma Sección. Su fase de especialización usa el mismo algoritmo de especialización que el EP offline de [RSV05], sólo que se ha adaptado para que las anotaciones "•" sean identificadas ahora con la anotación GEN. Pudimos comprobar que la especialización del EP offline usando un BTA y un análisis *size-change* en su proceso de anotación (para asegurar terminación) es más precisa, ya que los tiempos de ejecución de los programas especializados fueron, en promedio, menores que los respectivos generados por el EP offline que utiliza la caracterización *no creciente* (véase el cuadro 4.1). Por lo tanto se ha mejorado la primera aproximación de evaluación parcial offline para programas lógico funcionales.

3. Desarrollo de prototipo de anotación.
   Se han definido dos procedimientos de anotación con sus correspondientes procesos de especialización. A grandes rasgos, ambos procedimientos siguen la misma técnica y principio de anotación descrito en la optimización anterior, i.e., el uso de un BTA monovariante y el análisis size-change. Sin embargo, ahora se ha definido un proceso de anotación más intuitivo para realizar la generalización, es decir, con este nuevo proceso desde la anotación se indica claramente qué llamadas a función pueden desplegarse sin riesgo de no terminación (se les agrega anotación UNF) y se muestra cuáles no pueden ser desplegadas (estas llamadas a función se distinguen con la anotación MEM). Finalmente, por cada regla, se anotan con GEN las variables repetidas en la parte derecha excepto una, i.e., se linealiza cada regla. Con este esquema de anotación más completo ya no es necesario realizar ningún chequeo de terminación al momento de especializar (i.e., puede ser considerada como especialización *offline pura*). El segundo proceso de anotación identifica únicamente con anotación GEN los términos que introducen no terminación en las llamadas a función; tales términos son generalizados en la fase de especialización. En este caso sí es necesario hacer un chequeo de terminación en tiempo de

especialización para evitar una derivación indefinida. Por lo tanto, esta especialización se considera *offline híbrida*. En la implementación del prototipo se tuvo la oportunidad de comparar los dos especializadores *offline* (tanto *híbrido* como *puro*) contra el especializador *online*. En resumen los ejemplos especializados con el EP *online* se desempeñaron mejor que los respectivos especializados con ambos especializadores *offline*; sin embargo, el tiempo de especialización promedio consumido por los EP *offline* fue menor que el respectivo del EP *online*, véase la Sección 5.4.

6.2. Trabajo futuro

Respecto a las líneas de investigación podríamos mencionar las siguientes para dar continuidad al presente trabajo:

1. Definición de un BTA polivariante.
   Es posible mejorar la precisión de las anotaciones si consideramos que puede haber varias clasificaciones estático/dinámico para los argumentos de una función, es decir, que algún argumento puede tomar el valor estático en un momento dado y el valor dinámico en otro momento. El BTA que se ha definido aquí es monovariante, es decir, sólo considera un conjunto de valores estático/dinámico global.

2. Extender el esquema *offline* NPE
   Hasta el momento el prototipo de evaluación parcial no está definido para evaluar parcialmente las funciones predefinidas (built in's) tales como: $+, -, *, mod$, etc., ni las restricciones esto es: $(=:=), (\&), (\&>)$, etc. Incorporar estas definiciones permitiría evaluar parcialmente un mayor conjunto de programas Curry y, al mismo tiempo, se daría flexibilidad al evaluador parcial para intentar realizar alguna aplicación.

3. Implementación de un meta-intérprete
   Si construimos un meta-intérprete (`int`) para Curry y luego se especializa dicho meta-intérprete con respecto a un programa fuente en Curry, es posible "compilar" dicho programa fuente:

   $$\texttt{pobjeto} = [\![\text{mix}]\!]_{Curry}[\texttt{int}_{Curry}, \texttt{pfuente}]$$

   donde `pobjeto` se espera sea más eficiente que `pfuente`. En general `pobjeto` será una mezcla de `int` y `pfuente` conteniendo partes derivadas de ambos [JGS93]. La ecuación anterior es comunmente llamada primera proyección de Futamura (reportada originalmente en [Fut71]).

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.

[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.

[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.

[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.

[ARSV06] G. Arroyo, J.G. Ramos, J. Silva, and G. Vidal. Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In *Proc. of the 16th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 55–61. Università Ca' Foscari di Venezia, 2006. Extended version to appear in Springer LNCS.

[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).

[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.

[BD77] R.M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BN98] F. Baader and T.Ñipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[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.

[CGLH05a] S.-J. Craig, J. Gallagher, M. Leuschel, and K.S. Henriksen. Fully Automatic Binding Time Analysis for Prolog. In *Proc. of the Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 53–68. Springer LNCS 3573, 2005.

[CGLH05b] 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.

[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.

[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.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1&2):69–115, 1987.

[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.

[Fut71] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[Fut99] Yoshihiko 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.

[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.

[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.

[HeAE+04] M. Hanus (ed.), S. Antoy, M. Engelke, K. Höppner, J. Koj, P.Ñiederau, 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.

[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.

[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.

[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.

[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.

[Jul00] P. Julián. *Especialización de Programas Lógico-Funcionales Perezosos*. PhD thesis, DSIC-UPV, May. 2000. In spanish.

[Kom82] 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.

[LB02] M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4-5):461–515, 2002.

[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.

[LMD98] M. Leuschel, B. Martens, and D.De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.

[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.

[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.

[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.

[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.

[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).

[RSV05] 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.

[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.

[Sla74] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[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 Alge-*

bra in Engineering, Communication and Computing, 16(4):229–270, 2005.

[Tur86] V.F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

[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.

[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.