Universidad Politécnica de Valencia

Departamento de Sistemas
Informáticos y Computación

Tesis de Máster

Slicing Condicional de Programas Funcionales

Autor: Diego Cheda

Director: Germán Vidal

– Septiembre de 2008 –

Este trabajo está parcialmente financiado por la Unión Europea (FEDER - Fondo Europeo de Desarrollo Regional) y el Ministerio de Educación y Ciencia de España (MEC) bajo la subvención TIN2005-09207-C03-02, TIN2008-06622-C03-02, por *Acción Integrada* HA2006-0008 y por ALFA LERNet AML/19.0902/97/0666/II-0472-FA.

Dirección del autor:

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

Divide et impera
Cayo Julio César

Resumen

La *depuración* de programas es una de las tareas que más tiempo y esfuerzo consume durante el desarrollo del software. Esta situación se agrava en los lenguajes funcionales debido a que —como asevera Wadler en [Wad98] y, recientemente, en [HHJW07]— no se dispone de herramientas lo suficientemente evolucionadas para facilitar la depuración y la dificultad que representa la construcción de las mismas.

Cualquier aproximación para la depuración de programas debe tener por finalidad incrementar la habilidad de los programadores para identificar la naturaleza o la localización de un error y reducir, de esta forma, el tiempo y esfuerzo destinado a tal labor.

Una de las estrategias utilizadas para este fin se basa en la técnica denominada *fragmentación de programas* (conocida generalmente por su término en inglés: *slicing*) [Wei84]. Los programadores, durante la depuración de un programa, dividen a éste en piezas de código coherentes y, normalmente, no contiguas. Esas porciones de código —o *slices*— afectan potencialmente los valores calculados para algún punto de interés del programa, al que se denomina *criterio de slicing*. De este modo, si un programa calcula un valor erróneo para una variable, sólo aquellas sentencias del slice con respecto al punto de interés son las que, posiblemente, contribuyeron al cálculo de ese valor.

En el campo de los lenguajes imperativos el slicing de programas fue ampliamente investigado (e.g. debugging, testing, integración, comprensión y mantenimiento de programas, paralelismo, detección y eliminación de código muerto, reuso, etc.). En cambio, en el ámbito de los lenguajes funcionales existen pocos desarrollos.

El presente trabajo se enfoca como un paso hacia la mejora de las técnicas para la depuración de programas funcionales. Para ello, definiremos una técnica para realizar *slicing condicional* en programas funcionales de primer orden y que puede ser empleada no sólo en la depuración de éstos, sino también aplicada a los contextos mencionados en el párrafo anterior.

Básicamente, un *slice condicional* consiste en un subconjunto de sentencias del programa original que preserva el comportamiento del mismo con respecto a un conjunto de estados iniciales caracterizados por una fórmula lógica de primer orden sobre las variables de entrada del programa. La idea fue desarrollada originalmente en el paradigma imperativo [CPPGM91, CCLL94, LFM96, CCL98], sin embargo hasta el momento no ha sido definida en el contexto funcional.

Por lo tanto, esta investigación constituye un aporte original en el ámbito de los lenguajes funcionales.

Índice general

1

Introducción

1.1. Motivación

Los programadores emplean un tiempo y esfuerzo considerable en la *depuración*[1] de sus programas. Se calcula que entre el veinticinco y el cincuenta por ciento del coste destinado al desarrollo de software se consume en las fases de prueba y depuración [Zel78]. Por otra parte, la comprensión de un sistema de software existente conlleva entre el cincuenta y el noventa por ciento del tiempo dedicado a su mantenimiento [LFM96].

A pesar de esta realidad, muchas de las técnicas desarrolladas hace más de veinte años aún siguen siendo empleadas en la actualidad, tanto en el ámbito de los lenguajes imperativos como en el de los funcionales. En relación a lo anterior, en [Wad98], Wadler plantea las razones por las cuales los lenguajes funcionales no son ampliamente usados. Entre las posibles causas apunta a la necesidad de contar con herramientas para el debugging y a la dificultad que representa la construcción de las mismas (e.g., debido a la *evaluación perezosa* o *lazy*).

Pese a que han transcurrido diez años desde la aparición del controversial artículo de Wadler, la situación no ha cambiado mucho. Si bien existen diferentes esfuerzos orientados a conseguir herramientas para la depuración de programas funcionales (e.g. Buddha [Pop98], DDT [Cab05], HOOD [Gil00], Hat [WCBR01], entre otras), las herramientas de depuración aún no han alcanzado la madurez de otras áreas [HHJW07]. Además, su uso no se ha generalizado entre los programadores como consecuencia de la complejidad que acarrea el manejo de las mismas, por tratarse de versiones poco estables y que se encuentran en fase de investigación y desarrollo, por estar limitadas a un subconjunto de lenguaje en cuestión o, simplemente, por no cumplir con las espectativas del usuario [CS08].

Así, cualquier aproximación para la depuración de programas debe tener por finalidad incrementar la habilidad de los programadores para identificar la naturaleza o la localización de un error y reducir, de esta forma, el tiempo y esfuerzo destinado al debugging de programas.

[1] Utilizaremos indistintamente el término en castellano "*depuración*" como su equivalente en inglés "*debugging*".

Una de las estrategias utilizadas para tal fin se basa en la técnica conocida como *fragmentación de programas*[2] [Wei84]. Los programadores, durante la depuración de un programa, dividen a éste en piezas de código coherentes y, en general, no contiguas. Esas porciones de código —o *slices*— afectan potencialmente los valores calculados para algún punto de interés del programa, al que se denomina *criterio de slicing*. De este modo, si un programa calcula un valor erróneo para una variable, sólo aquellas sentencias del slice con respecto al punto de interés son las que, posiblemente, contribuyeron al cálculo de ese valor.

La noción de *slicing estático*, expuesta por Weiser [Wei84], permite obtener aquellas sentencias del programa que afectan (directa o indirectamente) al valor de una variable de interés sin considerar ninguna ejecución en particular. Mediante la examinación de las sentencias involucradas en el slice es posible identificar dónde se encuentra el error. Sin embargo, durante la depuración, el programador generalmente analiza el programa considerando los casos de prueba que provocan el error y no en forma genérica.

Korel y Laski [KL88] introdujeron el concepto de *slicing dinámico* que tiene en cuenta los valores de las variables de interés durante la ejecución del programa. Los slices obtenidos mediante esta técnica contienen todas las sentencias que realmente influyen en el valor que alcanza una variable en un punto del programa durante una ejecución específica del mismo. Consecuentemente y debido a que se conoce el flujo real de control para la entrada particular, el slicing dinámico logra slices más pequeños, es decir de mayor calidad y precisión. Por esta razón, el programador puede concentrarse en las sentencias que están relacionadas con la salida incorrecta y así efectuar un debugging más eficiente.

Posteriormente, Canfora et al [CCL98] definieron un marco general para la comprensión de programas basado en *slicing condicional*. Un slice condicional consiste en un subconjunto de sentencias del programa original que preserva el comportamiento del mismo con respecto a un conjunto de ejecuciones del programa. El conjunto de estados iniciales que caracteriza esas ejecuciones se especifica en términos de una fórmula lógica de primer orden sobre las variables de entrada del programa. Esta técnica permite una mejor descomposición del programa brindando la posibilidad de analizar los fragmentos de código desde diferentes perspectivas. Una de las ventajas del slicing condicional es que subsume a las técnicas tradicionales de slicing. Es decir, tanto el slicing estático como el dinámico pueden ser expresados usando slicing condicional.

Mientras que en el campo de los lenguajes imperativos el slicing de programas fue ampliamente investigado (e.g. debugging [LW87, ADS93, KSF90, GBF99], testing [Las90, FSKG92, BH93, BG96, Bin98], integración [HPR89], comprensión [LA93, HDS95] y mantenimiento de programas [GL91], paralelismo [Wei84], detección y eliminación de código muerto [RT96, LS03], reuso [LV96] e ingeniería

[2]Habitualmente se emplea la palabra inglesa "*slicing*" para referirse a ella.

reversa [BE93], etc.), en el ámbito de los lenguajes funcionales existen pocos desarrollos.

Considerando lo antes descripto, el trabajo se enfoca como un paso hacia la mejora de las técnicas para la depuración de programas funcionales. Con ese objetivo en mente, definiremos una técnica para realizar slicing condicional de programas funcionales de primer orden y que puede ser empleada no sólo en la depuración de éstos, sino también aplicada a los contextos mencionados en el párrafo anterior. Esta técnica, definida originalmente en el ámbito de los lenguajes imperativos, no tiene su contraparte en el paradigma funcional. Por lo tanto, el presente trabajo constituye un aporte y desarrollo original en el ámbito de los lenguajes funcionales.

1.2. Contribuciones

A continuación, mencionamos brevemente las contribuciones principales de la investigación:

- **La definición de slicing condicional para programas funcionales de primer orden**. El objetivo de este método es obtener, desde un programa, un subprograma que contiene todas las posibles derivaciones que pueden producirse por la ejecución de dicho programa cuyos argumentos de entrada están caracterizados por una condición lógica de primer orden. Además, tanto el programa original como el slice tienen el mismo comportamiento cuando son ejecutados con valores en sus argumentos que satisfacen la condición establecida.

- **La formulación de un algoritmo para el cálculo de los slices condicionales**. El cálculo de slices está dividido en dos fases. En primer lugar, extendemos la semántica original del lenguaje con el fin de recolectar aquellas partes del programa de interés de acuerdo al criterio de slicing condicional. Mediante el uso de la semántica aumentada y considerando la condición lógica, ejecutamos simbólicamente el programa desde una función inicial. El resultado de este proceso es un programa simplificado de acuerdo a la condición sobre los valores de entrada. Finalmente, a partir de este programa condicionado y utilizando la técnica de slicing estática propuesta en [CSV07] se obtiene el slice condicional.

- **El desarrollo de un prototipo que implementa la técnica de slicing condicional para el lenguaje multiparadigma *Curry* [HAB+06]**. Presentamos un prototipo desarrollado en Curry que permite obtener slices condicionales de acuerdo a la técnica propuesta.

1.3. Estructura del Trabajo

El trabajo está organizado de la siguiente manera:

- La primera parte constituye una revisión de algunos conceptos que serán utilizados a lo largo del trabajo. Principalmente, explicaremos —con mayor detalle— en qué consiste la técnica de slicing, algunas nociones útiles sobre sistemas de reescritura de términos y narrowing, y del lenguaje multiparadigma Curry.

- Luego, en la segunda parte, presentamos la técnica para slicing condicional de programas funcionales de primer orden. Inicialmente, introducimos el concepto de criterio de slicing condicional y de slice condicional. Ambos conceptos no han sido definidos previamente en el contexto declarativo.

  Además, desarrollamos un método y un algoritmo para calcular los slices condicionales. Para esa finalidad, extendemos la semántica del lenguaje funcional considerado con el fin de recolectar aquellas partes del programa que son relevantes de acuerdo al criterio de slicing condicional propuesto.

  Finalmente, describimos los detalles de la implementación del slicer.

- Para concluir, comentamos las líneas de investigación futuras y una discusión sobre los resultados de la investigación.

1.4. Publicaciones

El presente trabajo se elaboró basándonos en los siguientes artículos:

- **D. Cheda** y S. Cavadini, *Conditioned Slicing for Firsts Order Functional Logic Languages*. En *Proc. of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, 2008.

- **D. Cheda** y J. Silva, *An Evaluation of Algorithmic Debugger Implementations*. En *Proc. of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, 2008. A publicarse en Electronic Notes in Theoretical Computer Science (ENTCS).

- **D. Cheda** y J. Silva, *State of the Practice in Algorithmic Debugging*. En *VII Jornadas sobre Programación y Lenguajes (PROLE 2007)*, 2007.

- **D. Cheda**, J. Silva, y G. Vidal, *Static Slicing of Rewrite Systems*. En *Proc. of the 15th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)*. Electronic Notes in Theoretical Computer Science, 2006.

- **D. Cheda**, J. Silva, y G. Vidal, *Term Dependence Graph*. En *VI Jornadas sobre Programación y Lenguajes (PROLE 2006)*, 2006.

Otros trabajos publicados por el autor de este trabajo, pero no referenciados en la memoria:

- S. Cavadini y **D. Cheda**, *Run-time Information Flow Monitoring based on Dynamic Dependence Graphs*, 3th International Conference on Availability, Reliability and Security (ARES 2008). En *IEEE Proceedings*, pp. 586-591, 2008.

- S. Cavadini y **D. Cheda**, *Program Slicing Based on Sentence Executability*. En *XIII Congreso Argentino de Ciencias de la Computación (CACIC 2007)*, 2007.

- S. Cavadini y **D. Cheda**, *The Necessary Condition for Execution and its Use in Program Slicing*. En *VII Jornadas sobre Programación y Lenguajes (PROLE 2007)*, 2007.

- **D. Cheda** y S. Cavadini, *Refactoreo de Diagramas de Clases UML empleando Slicing de Modelos*. En *XII Congreso Argentino de Ciencias de la Computación (CACIC 2006)*, 2006.

2

Técnicas de Depuración

A diferencia de otros paradigmas de programación, una de las ventajas que poseen los lenguajes funcionales es que pueden evitarse muchos de los errores (o *bugs*) que se producen en tiempo de ejecución. Esto se debe a que cuentan con ciertas características intrínsecas (como ser el hecho de no contener *efectos secundarios*, la *transparencia referencial*, el *manejo implícito de la memoria* y el *sistema de inferencia de tipos*) que impiden la aparición de una gran cantidad de errores en un programa [Hug89].

Sin embargo, ningún paradigma está exento de errores. Entre los más comunes en los programas funcionales se encuentran: las excepciones (por ejemplo, una división por cero), errores en la lógica y aquellos provocados por el agotamiento de los recursos de memoria [Spa96].

Seguidamente describiremos el estado del arte de las técnicas de depuración en el paradigma funcional que permiten abordar estos tipos de errores.

2.1. Depuración Tradicional

Para la depuración de programas funcionales se puede emplear un acercamiento similar al debugging convencional utilizado en lenguajes imperativos: recurrir a sentencias que muestren los valores de las variables durante la evaluación del programa. Si bien la técnica es simple, suele resultar tediosa: requiere modificar y recompilar el programa cada vez que se introduce una función para imprimir los valores deseados. Además, esto puede ocasionar una alteración en la semántica del programa con la posible obtención de resultados incorrectos.

Algunas herramientas de depuración permiten especificar *puntos de interrupción* (o *breakpoints*) [SH99, EJ03, MK06, IM07] en ciertas partes del programa de tal forma que, al alcanzar uno de ellos durante la ejecución, el compilador detiene el programa y permite al usuario ver el estado actual de las variables deseadas.

Un método semejante al descripto anteriormente, consiste en utilizar una sentencia de impresión específica que no modifica la semántica del programa para observar las variables y estructuras de datos del mismo [Gil00, BCHH04].

En las estrategias tradicionales, el programador debe decidir qué funciones desea examinar para localizar un error en el programa. Por otra parte, el orden de ejecución en los lenguajes con evaluación lazy no es predecible fácilmente desde el código fuente e incluso puede variar dependiendo de la computación demandada. Estas particularidades provocan que los métodos tradicionales no sean efectivos para la depuración sobre todo porque los resultados obtenidos son extensos y difíciles de interpretar.

2.2. Profiling

Básicamente, esta técnica calcula la cantidad de recursos (tiempo, memoria, etc.) que un programa usa durante su ejecución y, teniendo en cuenta la medición realizada, permite detectar aquellas secciones de código ineficientes. Así, es posible localizar los errores provocados por el consumo excesivo de los recursos de memoria.

En [SJ97] se plantea una técnica de profiling que consiste en etiquetar el código fuente del programa con lo que se denomina *centros de costes*. A cada centro de coste se le atribuyen los recursos de tiempo y espacio de la tarea que se desea medir. Luego, a partir de las mediciones obtenidas, el programador puede concentrarse en mejorar el rendimiento de la sección de código que consume la mayoría de los recursos.

Otras aproximaciones similares, pero enfocadas para lenguajes multiparadigma pueden encontrarse en [AV02b, BHH+04].

```
(0)   main = sort ([2500..5000]++[1..2499])

(1)   sort [] = []
(2)   sort (x:xs) = insert x (sort xs)

(3)   insert x [] = [x]
      insert x (y:ys)
(4)     | x <= y = (x:y:ys)
(5)     | x > y  = (y:insert x ys)
```

Figura 2.1: Ordenamiento por inserción `insort`.

Ejemplo 2.2.1 *Consideremos el programa de la figura 2.1 que realiza el ordenamiento por inserción de una lista. Utilizando la herramienta de profiling incorporada en GHC (Glasgow Haskell Compiler) es posible obtener la cantidad de recursos de memoria y tiempo de ejecución que utilizan ambas funciones del*

```
total time  =         0.50 secs
total alloc =  176,086,248 bytes

 COST CENTRE   MODULE    entries   %time  %alloc        bytes
   main         Main           1     0.0     0.2       270100
    sort        Main        5001     0.0     0.0        60000
     insert     Main     6254999   100.0    99.5    175119948
```

Figura 2.2: Recursos de tiempo y espacio consumidos por `insort`.

programa. En la figura 2.2 se muestra un extracto de las mediciones obtenidas para cada centro de coste. En este caso, la función `insert` *es la que más recursos consume a lo largo de la ejecución.*

2.3. Tracing

La técnica se sustenta en la generación de una estructura de datos que almacena la traza (*trace*) de ejecución del programa y que se construye a medida que transcurre la misma. Luego, el programador puede explorarla para obtener información referida a la ejecución efectuada. Esta estructura de datos es un grafo que preserva las reducciones (*redex*) que el compilador realiza durante la evaluación del programa y las relaciones existentes entre ellas [SR97, Chi01].

Las herramientas de tracing presentan al usuario diferentes vistas y formas de recorrer los redex [Bre01, WCBR01, DC05]. En la práctica, la traza de un programa puede ser muy compleja y extensa. Esto último dificulta —sin dudas— el análisis y comprensión de la misma. Además, pueden ocupar enormes cantidades de memoria, lo que las hace en ocasiones intratables.

```
length :: [Int] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)

main = print $ length [1,2,3]
```

Figura 2.3: Definición de la función `length`.

Ejemplo 2.3.1 *La figura 2.4 muestra la traza del programa* `length` *que calcula la longitud de la lista* `[1,2,3]`*. La traza mostrada corresponde a una simplificación de la generada por el trazador Hat para Haskell. Dicha estructura de datos es*

explorada con la ayuda de diferentes herramientas de visualización que permiten recorrer la computación del programa.

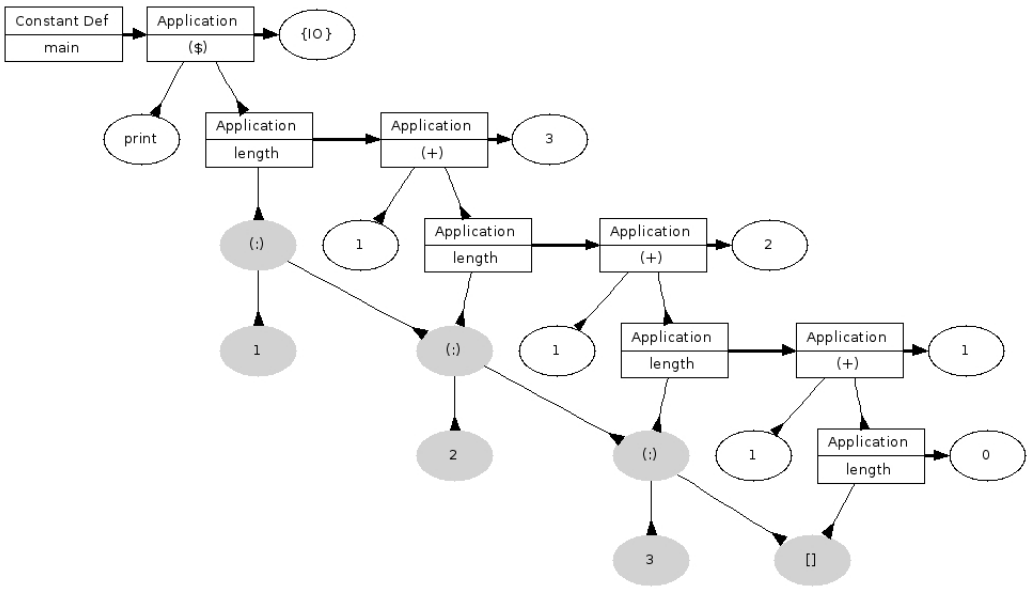

Figura 2.4: Traza del programa `length`.

2.4. Depuración Algorítmica

La *depuración algorítmica* [Sha83] (también llamada *depuración declarativa*) es un método basado en las respuestas de un *oráculo* (típicamente el programador) a una serie de preguntas —generadas automáticamente por la herramienta— con respecto a la corrección o no de las computaciones efectuadas por un programa. El depurador emplea dichas respuestas para guiar la búsqueda del error hasta aislar una sección de código que lo contiene.

Ejemplo 2.4.1 *Consideremos el programa erróneo de la figura 2.5. El bug está ubicado en la función* `insert` *en la línea* `(4)`*. La regla correcta debería ser:*

```
insert x (y:ys) | x <= y = (x:y:ys)
```

En la figura 2.6 podemos ver una típica sesión de depuración algorítmica usando una estrategia top-down left-to-right [Llo87] del programa 2.5. Para esto, primero se construye el árbol de ejecución que se muestra en la figura 2.7. Luego, la depuración se lleva a cabo con un recorrido de arriba hacia abajo y de izquierda a derecha desde la raíz del árbol. El oráculo debe responder a las preguntas

```
(0)   main = sort [3,1,2]

(1)   sort [] = []
(2)   sort (x:xs) = insert x (sort xs)

(3)   insert x [] = [x]
      insert x (y:ys)
(4)     | x <= y = (x:ys)
(5)     | x > y  = (y:insert x ys)
```

Figura 2.5: Definición incorrecta del ordenamiento por inserción `insort`.

realizadas por el depurador con el objetivo de localizar un error en el programa. Debido a que el resultado de `insert` *en el nodo* `(5)` *no es el esperado y, además, no tiene hijos, entonces el error se encuentra en esa función. En la figura 2.7 el nodo erróneo está coloreado en gris.*

```
Starting algorithmic debugging session

main = [1,3] ? NO
sort [3,1,2] = [1,3] ? NO
sort [1,2] = [1] ? NO
sort [2] = [2] ? YES
insert 1 [2] = [1] ? NO

Bug found in the function ïnsert" in rule (4).
```

Figura 2.6: Sesión de depuración del programa `insort`.

Esencialmente, un depurador algorítmico está constituido por dos fases: en la primera, se construye una estructura de datos —denominada *árbol de ejecución* [Nil98] (*execution tree*, por su siglas: *ET*)— y luego, en la segunda fase, ésta es recorrida empleando alguna estrategia (ver [Sil06]). La figura 2.8 muestra la estructura típica de un depurador algorítmico.

Cada nodo del ET contiene una ecuación formada por la llamada a una función con sus argumentos totalmente evaluados y el resultado devuelto por la misma.

Una vez que el ET es generado, el depurador lo atraviesa utilizando alguna estrategia de búsqueda (e.g. *top-down* [Llo87], *divide and query* [Sha83, HH93], *more rules first* [Sil06], *divide by rules and query* [Sil06], etc.) mientras pregunta al oráculo si la ecuación es o no correcta. Luego de responder una pregunta, los restantes nodos del ET son sospechosos —i.e., podrían contener un error— y éstos

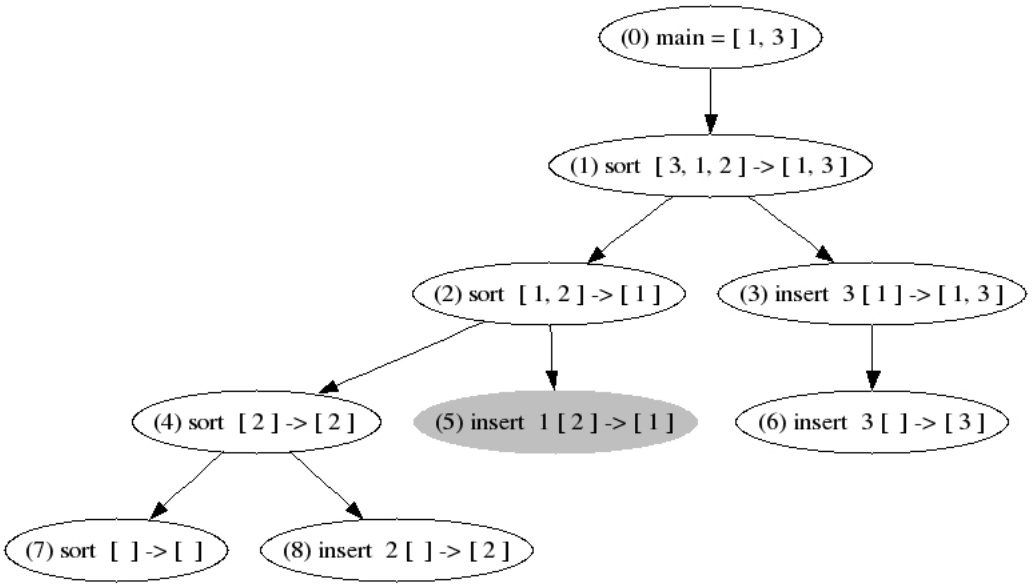

Figura 2.7: árbol de ejecución del programa `insort`.

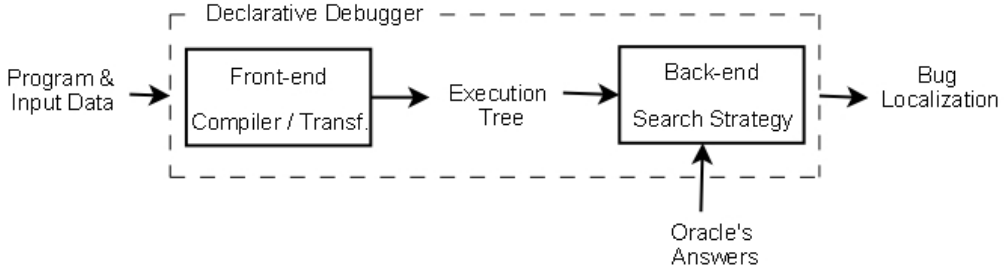

Figura 2.8: Esquema de un depurador algorítmico.

serán los próximos a examinar. Cuando todos los hijos de un nodo (si es que tiene alguno) son correctos, entonces el nodo es erróneo y el depurador ha localizado el error en la definición del mismo [Sha83].

La técnica garantiza que —dado un programa con un comportamiento incorrecto y después de haber respondido a las preguntas realizadas— el bug (eventualmente) será encontrado. El depurador sólo puede encontrar un error por sesión de depuración, pero, una vez que éste es removido, puede aplicarse nuevamente para buscar otro error si lo hubiere.

Aunque surgió como una herramienta de depuración para Prolog [Sha83], recientemente se han desarrollado depuradores de este tipo para Haskell (Buddha [Pop98] y Hat-Delta [DC05]), Curry (integrado al compilador *Münster Curry* [CL02]), *Toy* [CRA04, Cab05] y *Mercury* [Mac05]. Además, la técnica se ha trasladado a los lenguajes imperativos como *Java* [CHK07]. Una comparativa

entre las características funcionales y de rendimiento de distintos depuradores puede encontrarse en [CS08].

La ventaja más importante de la depuración algorítmica es que permite al usuario concentrarse en la semántica declarativa de los programas en lugar de hacerlo en sus aspectos operacionales. Durante el proceso de depuración, el usuario solamente tiene que decidir si una función particular aplicada a unos datos específicos produce un resultado correcto o no. Sin embargo, uno de los inconvenientes que presenta la depuración algorítmica es la complejidad de las preguntas que se deben responder. Además, las estrategias de búsqueda producen una serie de interrogantes que suelen estar desconectados semánticamente (i.e., debido a que se refieren a partes diferentes e independientes de la computación) haciendo el proceso muy complicado.

2.5. Slicing de Programas

El concepto —introducido por Weiser [Wei79] para el paradigma imperativo— plantea que los programadores, durante la depuración de un programa, dividen a éste en piezas de código coherentes y no necesariamente contiguas. Estas porciones de código, denominadas *slices*, afectan potencialmente los valores calculados para algún punto de interés del programa, al que se conoce como *criterio de slicing*. En otros términos, el slice de un programa P con respecto al criterio de slicing (p, v), donde p es una posición o sentencia determinada del programa y v la variable de interés, es un subprograma P' que contiene aquellas sentencias que afectan al valor de v calculado en p.

La aplicación original del slicing fue la de asistir a los programadores durante la depuración de un programa. Sin embargo, su uso se extendió a otras áreas tales como *testing*, integración, comprensión y mantenimiento de programas, paralelismo, reuso e ingeniería reversa, etc. Esto último ha dado lugar a diversas definiciones de slice, así como diferentes métodos de cálculo dependiendo del área en cuestión.

2.5.1. Slicing Estático

La definición original de Weiser es estática debido a que no considera ninguna ejecución particular del programa.

Ejemplo 2.5.1 *El programa imperativo de la figura 2.9 calcula la suma y el producto de los* `n` *primeros números positivos. Un slice de este programa computado estáticamente con respecto al criterio de slicing* `(9,sum)` *contendrá las sentencias en negro, mientras que las coloreadas en gris serán suprimidas. El slice contiene todas aquellas partes del programa que son necesarias para calcular el valor de la variable* `sum` *en la línea* `(9)` *sin considerar ningún dato de entrada en particular.*

```
(1)    sumProd(n)
(2)      i := 1;
(3)      sum := 0;
(4)      prod := 1;
(5)      while (i<n) do
(6)        sum := sum + i;
(7)        prod := prod * i;
(8)        i := i + 1;
(9)      write(sum);
(10)     write(product);
```

Figura 2.9: Slice estático del programa `sumProd`.

La noción esencial en el slicing es la de *dependencia*. Los valores calculados por la sentencia de interés indicada en el criterio sólo pueden verse afectados por aquellos de los cuales depende. Es posible diferenciar entre dos tipos de dependencias:

- La *dependencia de datos* se presenta cuando una variable puede tomar un valor incorrecto si se cambia el orden de ejecución entre dos sentencias. Por ejemplo:

  ```
  (1)  a = b * c
  (2)  d = a * e
  ```

  La sentencia `(2)` depende de `(1)` ya que si se ejecuta `(2)` antes que `(1)` puede resultar que `(2)` use un valor incorrecto de `a`.

- Existe *dependencia de control* entre una sentencia y un predicado de control si el valor del predicado controla inmediatamente la ejecución de la sentencia. Por ejemplo:

  ```
  (1)  if (c) then
  (2)        a = b * d
       endif
  ```

  La sentencia `(2)` depende del predicado `c` puesto que el valor tomado por el mismo determina si `(2)` se ejecuta o no.

Es posible representar de manera explícita estas dependencias mediante el *grafo de dependencia del programa* (*PDG* por *Program Dependence Graph*) y transformar el slicing estático en un problema de alcanzabilidad de grafos

[OO84, FOW87]. Un PDG es un grafo dirigido cuyos nodos se corresponden con sentencias y sus arcos representan las dependencias de datos y de control entre dichas sentencias. El criterio de slicing coincide con un nodo del PDG y el slice con respecto a ese nodo está formado por todos aquellos nodos desde los cuales puede ser alcanzado.

Ejemplo 2.5.2 *La figura 2.10 es el PDG del programa 2.9. Las líneas sólidas representan las dependencias de control y las de puntos, las de datos. Los nodos están etiquetados con el número de sentencia del programa que respresentan. Haciendo uso del PDG, un slice de este programa con respecto al criterio de slicing* `(9,sum)` *contendrá los nodos en negro, mientras que los coloreados en gris serán suprimidos. El slice contiene todas aquellos nodos que alcanzan a* `(9)` *siguiendo arcos de dependencia de control y de datos.*

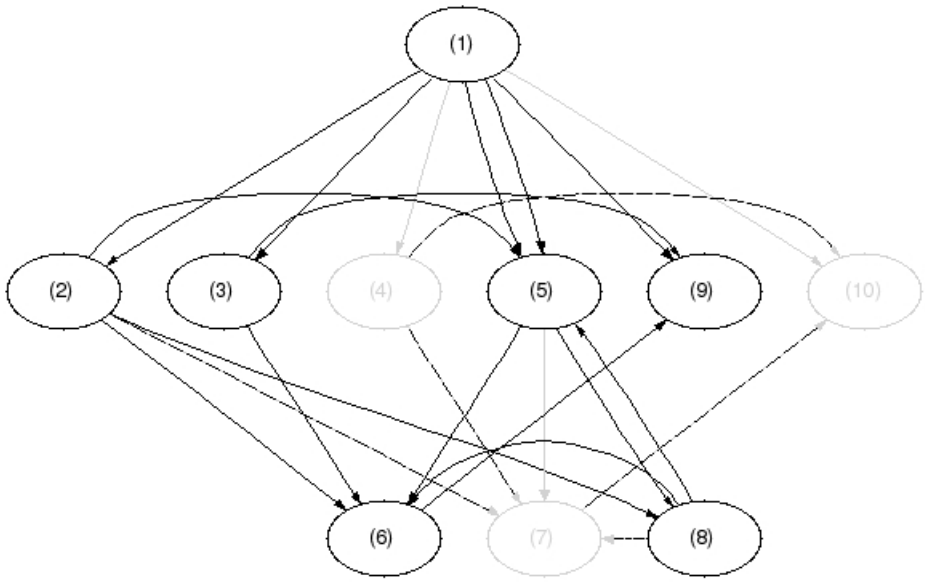

Figura 2.10: Grafo de Dependencias del Programa `sumProd`.

El slice del ejemplo anterior fue producido siguiendo los arcos de datos y control desde el nodo correspondiente al criterio de slicing hacia *atrás*. Este tipo de slicing se conoce como *backward slicing* y permite obtener aquellas sentencias de las que depende cierto punto del programa. Por otra parte, también es posible conocer qué sentencias se ven afectadas por un punto de interés si se recorre el grafo desde el criterio de slicing hacia *adelante*, es decir efectuando *forward slicing* [BC85].

Ejemplo 2.5.3 *Un forward slice del programa de la figura 2.9 con respecto al criterio de slicing* `(3,sum)` *se muestra en la figura 2.11.*

```
(3)       sum := 0;
          ⋮
(6)         sum := sum + i;
          ⋮
(9)       write(sum);
```

Figura 2.11: Forward slice del programa `sumProd`.

2.5.2. Slicing Dinámico

En general, los programadores durante la depuración se concentran en una ejecución específica de un programa, en aquella que causó el error. Es decir, consideran la ejecución para unos datos de entrada determinados (y no para cualquier ejecución como con slicing estático).

Korel y Laski [KL88] introdujeron el concepto de *slicing dinámico* que tiene en cuenta los valores de las variables de interés durante la ejecución del programa. Los slices obtenidos mediante esta técnica contienen todas las sentencias que realmente influyen en el valor que alcanza una variable en un punto del programa durante una ejecución específica del mismo. Consecuentemente y debido a que se conoce el flujo real de control para la entrada particular, el slicing dinámico logra slices más pequeños, es decir de mayor calidad y precisión. Razón por la cual, el programador puede concentrarse en las sentencias que están relacionadas con la salida incorrecta y así efectuar un debugging más eficiente.

Durante la ejecución de un programa, la misma sentencia puede ser ejecutada varias veces con diferentes valores para las variables. Por eso, el criterio de slicing debe ser modificado para poder distinguir una ejecución particular. Por lo tanto, quedará conformado por la tripla $(\{x_1, ..., x_n\}, I^q, V)$, donde $x_1, ..., x_n$ son los valores inciales del programa, q es el número de ocurrencia de la sentencia I a lo largo de la ejecución del programa y V es el conjunto de variables de interés.

Ejemplo 2.5.4 *El slicing dinámico del programa 2.9 con respecto a* $(\{\texttt{n=0}\}, 9^1, \texttt{sum})$ *se muestra en la figura 2.12. El ejemplo permite observar cómo el slicing dinámico sólo considera aquellas sentencias que realmente son necesarias para el cómputo de la variable de interés. Como* `n=0`*, entonces el ciclo que se inicia en la sentencia* `(5)` *jamás se ejecutará y, por lo tanto, no es necesario para obtener el valor de* `sum`*.*

Aunque el slicing dinámico produce slices más precisos, el costo implicado en la computación de las dependencias es considerablemente mayor (en cuanto a tiempo y espacio) y requiere de estructuras de datos más complejas (por ejemplo, una

```
(1)    sumProd(n)
(2)      i := 1;
(3)      sum := 0;
(4)      prod := 1;
(5)      while (i<n) do
(6)        sum := sum + i;
(7)        prod := prod * i;
(8)        i := i + 1;
(9)      write(sum);
(10)     write(product);
```

Figura 2.12: Slice dinámico del programa `sumProd`.

traza de ejecución —redex— como las mencionadas en la sección 2.3). También, dependiendo de cómo se recorran estas estructuras de datos (hacia atrás o hacia adelante), se pueden tener backward o forward slicing de manera dinámica.

2.5.3. Slicing Condicional

El slicing condicional [CCL98] es una aproximación intermedia entre el slicing estático y el dinámico. Mientras que en el slicing estático no se tiene información sobre las entradas concretas del programa y en el slicing dinámico se emplea un conjunto de valores específicos para los argumentos, en el slicing condicional el criterio contiene adicionalmente una fórmula lógica que caracteriza a un conjunto de valores de entrada.

Por lo tanto, el slice condicional del programa estará compuesto por aquellas partes del programa que pueden ser (potencialmente) ejecutadas cuando la condición sobre las entradas se cumple.

Ejemplo 2.5.5 *El programa de la figura 2.13 computa la suma y el producto de los* `n` *primeros números positivos o negativos. El slicing condicional de dicho programa con respecto al criterio* `(14,sum,{n` $\geq$ `0})` *se muestra en la misma figura. Las sentencias que pertenecen al slice están marcadas con negro mientras que en gris aparecen aquellas que no forman parte del slice. El slicing condicional sólo considera aquellas sentencias que realmente son necesarias para el cómputo de la variable de interés cuando los argumentos satisfacen la condición establecida. Como la condición indica que* `n` $\geq$ `0`*, entonces el bloque de la sentencia* `if` *desde la línea* `(10)` *hasta la línea* `(12)` *jamás se ejecutará y, por ende, dichas sentencias no son necesarias para obtener el valor de la variable de interés* `sum`*.*

Una característica importante de este tipo de slice es que las ejecuciones del

```
(1)    sumProd2(n)
(2)      i := 1;
(3)      j := abs(n);
(4)      sum := 0;
(5)      prod := 1;
(6)      while (i < j) do
(7)        if (n ≥ 0) then
(8)           sum := sum + i;
(9)           prod := prod * i;
(10)       else if (n < 0) then
(11)          sum := sum + (-i);
(12)          prod := prod * (-i);
(13)       i := i + 1;
(14)     write(sum);
(15)     write(product);
```

Figura 2.13: Slice condicional del programa `sumProd2`.

programa y su slice condicional, con valores para las entradas que satisfacen la condición, son indistinguibles. Es decir, ambos programas tienen el mismo comportamiento.

La relevancia de esta técnica radica en el hecho de que proporciona un marco común de slicing, permitiendo de ésta forma computar slices estáticos y dinámicos [CCL98] mediante la adecuación del criterio utilizado. En resumen, el slicing condicional subsume a ambas técnicas.

Ejemplo 2.5.6 *El slice estático calculado en la figura 2.9 puede representarse mediante el criterio de slicing condicional* `(9,sum,{True})`*. De la misma forma, el slice dinámico de la figura 2.12 puede ser computado usando el criterio* `(14,sum,{n = 0})` *—i.e., colocando un valor concreto en cuenta de una condición para las variables de entrada del programa—.*

2.5.4. Slicing de Programas Funcionales

En la sección anterior describimos las técnicas tradicionales de slicing (estático, dinámico y condicional) en el paradigma imperativo. En esta sección presentamos los mismos conceptos, pero aplicados al paradigma funcional. Nos concentraremos principalmente en las técnicas descriptas en [OSV04, CSV07] para slicing estático y dinámico y explicaremos nuestra idea de slicing condicional en programas funcionales. Para tal fin, utilizaremos el programa de la figura 2.14. Dicho programa, escrito en el lenguaje multiparadigma Curry [HAE+07], calcula

```
data Nat = Zero | Succ Nat
data Triangle = Equilateral | Isosceles | Scalene
data Pairs = Pair Nat Triangle

main = triangle (Succ Zero) (Succ Zero) (Succ Zero)
triangle x y z = Pair (sum x y z) (kind x y z)

sum x y z = add x (add y z)

add Zero x = x
add (Succ x) y = (Succ (add x y))

kind x y z = if (x == y) then
               (if (y == z) then
                   Equilateral
                else
                   Isosceles)
             else
               (if (y == z) then
                   Isosceles
                else
                   Scalene)
```

Figura 2.14: Slice estático con respecto a `(triangle x y z,Pair` $\perp$ $\top$`)`.

el perímetro de un triángulo y su clasificación con respecto a la longitud de sus lados (Equilatero, Isósceles y Escaleno), y produce un par que contiene dicha información.

Tal como dijimos anteriormente, durante el slicing estático los valores de los argumentos del programa son desconocidos. Por lo tanto, la simplificación del programa es realizada teniendo en cuenta todas las posibles entradas. Por ejemplo, de acuerdo a la aproximación definida en [CSV07], el slice estático del programa de la figura 2.14 con respecto al criterio `(triangle x y z,Pair` $\perp$ $\top$`)` es el subprograma compuesto por las sentencias mostradas en color negro. En este caso, el criterio de slicing estático es un par *(función,* π*)* donde *función* es la función desde la que se iniciará el slicing y π denota qué parte de la estructura de un término constructor se ignora y cuál resulta relevante para el slicing. Para eso se emplean dos símbolos especiales: $\perp$ que indica una subexpresión cuya computación no es relevante, y $\top$ para subexpresiones relevantes (esta notación, denominada *patrones de slicing* [OSV04], está basada en la idea de "patrones de

vivacidad" —*liveness patterns*— [LS03]).

El slice contiene sólo aquellas partes del programa que afectan al cálculo del segundo argumento del constructor `Pair` —es decir, a la clasificación del triángulo— partiendo desde la función inicial `triangle x y z`. De color negro se marcan aquellas sentencias que pertenecen al slice, mientras que en gris las que no están relacionadas con el mismo.

A diferencia del slicing estático, en el slicing dinámico la simplificación es llevada a cabo teniendo en cuenta valores específicos para los argumentos de entrada del programa. Es decir, la técnica requiere un conjunto de valores usados como entradas para una ejecución específica del programa en cuestión.

Por ejemplo, la figura 2.15 muestra[1] el slice dinámico del programa con respecto al criterio de slicing (`main,Pair _ Equilateral,Pair` $\bot$ $\top$), realizado usando la técnica propuesta en [OSV04]. Aquí, el criterio de slicing es una tupla *(función, valor parcial,* π*)*, donde *función* es la llamada a la función con valores concretos para sus argumentos, *valor parcial* es un resultado parcial obtenido desde la llamada a la *función*, y π indica qué parte del resultado de la *función* es relevante o no.

Debido a que el flujo de control es conocido, los slices dinámicos son, en general, más pequeños que los obtenidos de manera estática.

El slicing condicional [CCL98] es una aproximación intermedia entre el slicing estático y el dinámico. Mientras que en el slicing estático no se tiene información sobre las entradas concretas del programa y en el slicing dinámico se emplea un conjunto de valores específicos para los argumentos, en el slicing condicional el criterio contiene adicionalmente una fórmula lógica que caracteriza a un conjunto de valores de entrada.

La figura 2.16 muestra un ejemplo de esta clase de slicing. El criterio de slicing es la tupla *(función,*ϕ*,*π*)* que contiene una llamada a la función inicial, una fórmula lógica sobre los argumentos y un patrón de slicing para denotar aquellas partes relevantes del resultado. En el ejemplo, la fórmula lógica indica que dos de los lados deben tener la misma longitud.

El slice del programa está compuesto por aquellas partes del programa que pueden ser (potencialmente) ejecutadas cuando la condición sobre las entradas se cumple.

Las ejecuciones del programa y su slice condicional, con valores para las entradas que satisfacen la condición, son indistinguibles. Es decir, ambos programas tienen el mismo comportamiento.

Como hemos afirmado previamente, el slicing conditional no ha sido definido en el paradigma declarativo. En el captítulo 4 introduciremos nuestra aproximación a la técnica de slicing condicional para programas funcionales de primer

[1] Como en el ejemplo de slicing estático, las partes relevantes del programa están señaladas en color negro, mientras que las descartadas en gris.

```
data Nat = Zero | Succ Nat
data Triangle = Equilateral | Isosceles | Scalene
data Pairs = Pair Nat Triangle

main = triangle (Succ Zero) (Succ Zero) (Succ Zero)
triangle x y z = Pair (sum x y z) (kind x y z)

sum x y z = add x (add y z)

add Zero x = x
add (Succ x) y = (Succ (add x y))

kind x y z = if (x == y) then
               (if (y == z) then
                   Equilateral
                else
                   Isosceles)
             else
               (if (y == z) then
                   Isosceles
                else
                   Scalene)
```

Figura 2.15: Slice dinámico con respecto a `(main,Pair _ Equilateral,Pair ⊥ ⊤)`.

orden.

2.5.5. Otros Trabajos Relacionados

Mientras que en el campo de los lenguajes imperativos el slicing de programas fue ampliamente investigado, en el ámbito de los lenguajes funcionales existen pocos desarrollos. A continuación describimos brevemente los más relevantes:

Reps y Turnidge [RT96] presentan un algoritmo de slicing estático para programas funcionales de primer orden aplicado en la especialización de programas. Para este fin, utilizan una función de proyección como criterio de slicing. La función de proyección se emplea para caracterizar qué información debe ser "descartada" y cuál "retenida" a partir de los valores que el programa computa. Luego, esta proyección se propaga hacia atrás —a través del cuerpo del programa— desde el resultado de la función principal, a fin de simplificar apropiadamente las partes que no son requeridas para computar dicho resultado final.

Biswas [Bis97] busca extraer slices ejecutables de programas de orden superior

```
data Nat = Zero | Succ Nat
data Triangle = Equilateral | Isosceles | Scalene
data Pairs = Pair Nat Triangle

main = triangle (Succ Zero) (Succ Zero) (Succ Zero)
triangle x y z = Pair (sum x y z) (kind x y z)

sum x y z = add x (add y z)

add Zero x = x
add (Succ x) y = (Succ (add x y))

kind x y z = if (x == y) then
               (if (y == z) then
                   Equilateral
                else
                   Isosceles)
             else
               (if (y == z) then
                   Isosceles
                else
                   Scalene)
```

Figura 2.16: Slice condicional con respecto a `(triangle x y z,x==y,Pair` $\bot\top$`)`.

(con asignaciones y excepciones) mediante slicing dinámico basado en una traza de la ejecución.

Field y Tip [FT98] plantean una noción de slice en sistemas de reescritura de términos que utiliza la relación entre contextos (i.e., un fragmento de un término) en una reducción.

Ahn y Han [AH99] exponen formalmente un método para slicing estático de lenguajes funcionales de primer orden que se sustenta en una semántica operacional empleada para diferenciar aquellas partes del programa que pertenecerán al slice.

Liu y Stoller [LS03] definen un algoritmo para slicing estático en lenguajes funcionales de primer orden. Su principal interés es eliminar *código muerto*. De manera similar a la empleada en [RT96], utilizan *patrones de vivacidad* (*liveness patterns*) para indicar qué parte de una computación corresponde a código "vivo" (i.e., utilizado durante la ejecución) o muerto.

Hallgren [Hal03] exhibe, como uno de los resultados del proyecto *Programati-*

ca[2], el desarrollo de una herramienta para slicing en programas Haskell, pero no proporciona mayores detalles sobre la misma.

Vidal [Vid03, SV07] introduce la técnica de forward slicing para programas declarativos utlizando una extensión de la evaluación parcial. En este enfoque es posible realizar slicing estático o dinámico dependiendo del grado de instanciación del criterio de slicing.

En [OSV04], Ochoa, Silva y Vidal presentan un método para realizar slicing dinámico de programas funcionales basado en una extensión de los redex-trails que permite almacenar la ubicación en el programa de cada expresión reducida. A través de la semántica operacional instrumentada se genera una traza de la computación. Luego, se obtiene el slice dinámico recolectando todos los nodos de la traza de acuerdo al criterio de slicing. Además, la técnica está implementada para el lenguaje multiparadigma Curry[3].

Rodrigues y Barbosa [RB05] implementan en la herramienta *HaSlicer*[4], un slicer para Haskell que utiliza una estructura de datos denominada *Grafo de Dependencia Funcional* (*FDG* por *Functional Dependence Graph*). El FDG contiene las relaciones existentes entre las llamadas a función que se emplean para identificar las dependencias entre componentes de distintos módulos.

En [CSV07], Cheda, Silva y Vidal describen una técnica para realizar slicing estático y dinámico de programas funcionales de primer orden que se basa en una noción de *dependencia* entre los términos del mismo. Además, plantean una aproximación que utiliza una estructura de datos llamada *Grafo de Dependencias de Términos* (*TDG* por *Term Dependence Graph*). Este grafo permite representar las posibles computaciones de un programa y, a partir del TDG construido, obtener el slice del programa con respecto al criterio dado.

Los pocos desarrollos existentes con respecto al slicing de programas funcionales convierten a este campo en un interesante espacio de investigación.

[2]http://www.cse.ogi.edu/PacSoft/projects/programatica/
[3]http://www.dsic.upv.es/∼jsilva/slicer/
[4]http://labdotnet.di.uminho.pt/HaSlicer/HaSlicer.aspx

3

Preliminares

A continuación introducimos las nociones básicas de *Sistemas de Reescritura de Términos* (*Term Rewriting Systems*, *TRS*) extraídas de [BN98, DP01], y de Narrowing [Sla74, AV02a] que serán usadas a lo largo del trabajo. Al final del capítulo presentamos la relación existente entre TRS y programas funcionales y, finalmente, la sintaxis y semántica del lenguaje funcional que emplearemos durante el resto del trabajo.

3.1. Términos y Signaturas

Una *signatura* $\mathcal{F}$ es un conjunto de *símbolos de función* donde cada $f \in \mathcal{F}$ tiene definida su *aridad* n con $n \geq 0$. Los símbolos con aridad cero se denominan *símbolos constantes*. Un símbolo que se emplea para construir un dato es un *constructor*. En los ejemplos, se utilizarán letras mayúsculas para los constructores y minúsculas para los símbolos de función y variables.

Ejemplo 3.1.1 *Supongamos que queremos representar la suma de números naturales (simbolizados por los términos construidos desde cero y sus sucesores). Para ello necesitamos los siguientes símbolos:*

- *el símbolo de función binario* `+` *de la suma,*
- *el símbolo constructor unario* `S`*,*
- *el símbolo constante* `0`*.*

Entonces, $\mathcal{F} = \{+, S, 0\}$.

Dada una signatura $\mathcal{F}$ y un conjunto de variables $\mathcal{V}$ tal que $\mathcal{F} \cap \mathcal{V} = \emptyset$, el conjunto $\mathcal{T}(\mathcal{F}, \mathcal{V})$ de todos los *términos* de $\mathcal{F}$ sobre $\mathcal{V}$ se define inductivamente como:

- $\mathcal{V} \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$ (i.e. cada variable es un término),

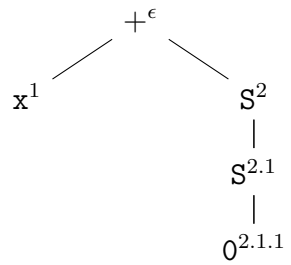

Figura 3.1: Representación del término `x + S(S(0))` como un árbol.

- $f(t_1, ..., t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ para todo $n \geq 0$, $f \in \mathcal{F}$ y $t_1, ..., t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ (i.e. la aplicación de símbolos de función a términos también son términos).

Ejemplo 3.1.2 *Ejemplos de términos sobre* $\mathcal{F} = \{+, S, 0\}$ *y un conjunto de variables* $\mathcal{V} = \{x\}$ *son* $S(S(0))$, $x + S(S(0))$, $S(S(0)) + 0$, $S(0) + S(0)$. *No son términos los siguientes:* S, $+$, $S(0, 0)$.

$Var(t)$ es el conjunto de variables que aparecen en el término t. Un término t es *básico* si no tiene variables[1], i.e. $\mathcal{V}ar(t) = \emptyset$.

Ejemplo 3.1.3 *Sea* $t = x + S(0)$, *entonces* $\mathcal{V}ar(t) = \{x\}$.

3.2. Posiciones

La estructura de un término puede ser vista como un árbol, donde los símbolos de función son nodos y los arcos apuntan a los argumentos de la misma. Los nodos del árbol pueden enumerarse con una secuencia de números enteros positivos a los que se conoce como *posición* de un término.

El símbolo ϵ es la *posición raíz* y el término etiquetado en esa posición se denomina *término raíz*. En el trabajo indicaremos la raíz del término t con $root(t)$.

Ejemplo 3.2.1 *El ejemplo de la figura 3.1 muestra la representación del término* $x + S(S(0))$ *como un árbol. Al lado de cada nodo se encuentra su posición.*

El conjunto de posiciones $Pos(t)$ de un término se define de la siguiente manera:

$$Pos(t) \;=\; \begin{cases} \{\epsilon\} & \text{si } t \in X \\ \{\epsilon\} \;\cup\; \{i.p \mid p \in Pos(t_i) \wedge 1 \leq i \leq n\} & \text{si } t = f(t_1, ..., t_n) \end{cases}$$

[1]Se utiliza también la palabra "*ground*" para denominarlos.

Ejemplo 3.2.2 *Siguiendo el ejemplo 3.2.1, el conjunto de posiciones del subtérmino* $s = S(S(0))$ *es:* $Pos(s) = \{2, 2.1, 2.1.1\}$

El *subtérmino* t en la posición p se indica con $t|_p$ con $p \in Pos(t)$ y se define como sigue:

$$t|p \;=\; \begin{cases} t & \text{si } p = \epsilon \\ t_i|q & \text{si } p = i.q \text{ y } t = f(t_1, ..., t_n) \text{ con } 1 \leq i \leq n \end{cases}$$

Ejemplo 3.2.3 *Se accede a los términos y subtérminos mediante las posiciones. Si* $t = x + S(S(0))$*, entonces:* $t|_\epsilon = x + S(S(0))$, $t|_1 = x$, $t|_2 = S(S(0))$, $t|_{2.1} = S(0)$, $t|_{2.1.1} = 0$.

El *reemplazo* de un subtérmino de t en la posición p por un término s, i.e. $t[s]_p$, se define:

$$t[s]_p \;=\; \begin{cases} s & \text{si } p = \epsilon \\ f(t_1, ..., t_i[s]_q, ..., t_n) & \text{si } p = i.q \text{ y } t = f(t_1, ..., t_n) \text{ con } 1 \leq i \leq n \end{cases}$$

Ejemplo 3.2.4 *Dado el término* $t = x + S(S(0))$*, podríamos reemplazar el subtérmino* $t|_1$ *por* $s = S(0)$*. Entonces,* $t[s]_1 = S(0) + S(S(0))$.

3.3. Sustituciones

Una *sustitución* $\sigma = \{x_1 \mapsto s_1, ..., x_n \mapsto s_n\}$ es una función que permite realizar una correspondencia entre variables y términos de tal forma que la aplicación de σ a un término t reemplaza de forma simultánea todas las ocurrencias de las variables x_i por los respectivos términos s_i.

Un término t' es una *instancia* del término t si existe una sustitución σ tal que $\sigma(t) = t'$.

Ejemplo 3.3.1 *Con el término* $t = x + S(S(x))$ *y la sustitución* $\sigma = \{x \mapsto 0\}$*, si aplicamos la sustitución a* t*:* $\sigma(t) = 0 + S(S(0))$.

Diremos que dos términos s y t *unifican* con una sustitución σ si $\sigma(s) = \sigma(t)$.

Ejemplo 3.3.2 *Si consideramos los términos* $s = +(x, 0)$ *y* $t = +(S(S(0)), 0)$ *y la sustitución* $\sigma = \{x \mapsto S(S(0))\}$*,* t *unifica con* s *debido a que* $\sigma(s) = \sigma(t)$.

3.4. Sistemas de Reescritura de Términos

Un *sistema de reescritura de términos*[2] $\mathcal{R}$ sobre una signatura finita $\mathcal{F}$ es un conjunto finito de reglas (o ecuaciones orientadas) $l \to r$ de tal forma que el término l no es una variable y r es un término cuyas variables aparecen en l. Los términos l y r se conocen como lado izquierdo y lado derecho de una regla. Cada regla del TRS será enumerada para identificarla de manera única.

Dado un TRS, consideraremos la partición de la signatura $\mathcal{F} = \mathcal{D} \cup \mathcal{C}$, donde los símbolos *definidos* $\mathcal{D}$ son a las raíces de los lados izquierdos de las reglas y los símbolos *constructores* el resto: $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. De acuerdo a esto, el domino de términos y de *términos constructores* es $\mathcal{T}(\mathcal{F}, \mathcal{V})$ y $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectivamente, donde $\mathcal{V}$ es un conjunto de variables y cumple que $\mathcal{F} \cap \mathcal{V} = \emptyset$.

Ejemplo 3.4.1 *El ejemplo siguiente muestra el sistema de reescritura que define la suma de números naturales.*

$$\begin{array}{lll} (R1) & +(x, 0) & \to \quad x \\ (R2) & +(x, S(y)) & \to \quad S(+(x, y)) \end{array}$$

En este caso, $\mathcal{D} = \{+\}$, $\mathcal{C} = \{S\}$ *y* $\mathcal{V} = \{x, y\}$. *Algunos ejemplos de* $\mathcal{T}(\mathcal{F}, \mathcal{V})$ *son* $S(S(0))$, $x + S(S(0))$, $S(S(0)) + 0$, $S(0) + S(0)$.

Un paso de reescritura es la aplicación de una regla a un término, i.e., el subtérmino $t|_p$ se reescribe al término s con la regla R, en símbolos $t \to_{p,R} s$, si:

$$\exists\ l \to r \in R,\ p \in Pos(t),\ \sigma.\ t|_p = \sigma(l) \wedge s = t[\sigma(r)]_p$$

Una *expresión reducible* o *redex* (*reducible expression*) es una instancia de un lado izquierdo de una regla, i.e. $\sigma(l)$. Reducir un redex significa reemplazarlo por su correspondiente instancia del lado derecho de la regla, i.e. $t[\sigma(l)]_p$ a $t[\sigma(r)]_p$. En los ejemplos, subrayaremos los redex seleccionados en cada paso de reducción. Diremos que un término t es *irreducible* o está en *forma normal* si no existe un término s de manera que desde t sea posible obtener s, i.e. $t \not\to s$.

Denotaremos con $\to^+$ a la clausura transitiva de $\to$ y por $\to^*$ a la clausura transitiva y reflexiva. Dado un TRS $\mathcal{R}$ y un término t, diremos que t se *evalúa* a s si y sólo si $t \to^* s$ y s está en forma normal.

Ejemplo 3.4.2 *A partir del TRS del ejemplo 3.4.1 podemos calcular la suma de dos números naturales. En este caso, sumaremos 2 y 1 (representados por los números* $S(S(0))$ *y* $S(0)$ *en notación de Peano, respectivamente) mediante la aplicación sucesiva de las reglas del TRS hasta obtener una forma normal que corresponda al resultado de la operación.*

$$\underline{+(S(S(0)), S(0))} \ \to_{R2}\ S(\underline{+(S(S(0)), 0)}) \ \to_{R1}\ S(S(S(0)))$$

[2] *TRS* por las siglas en inglés para *term rewriting system*.

Podemos decir que el término $t = +(S(S(0)), S(0))$ *se evalúa a* $s = S(S(S(0)))$ *ya que* $t \to^* s$ *y* s *es irreducible.*

3.5. Narrowing

Si una llamada a una función contiene variables *libres* en sus argumentos, entonces es necesario instanciar esas variables a un término apropiado con el fin de evaluar dicha función a un término en forma normal. Este proceso es conocido como *narrowing* [Sla74]. Básicamente, para evaluar un término que contiene variables, narrowing instancia indeterminísticamente las variables del término de forma que sea posible realizar un paso de reescritura. La técnica computa el unificador más general entre un subtérmino y la parte izquierda de alguna regla y reemplaza éste por la parte derecha instanciada de la regla [Han94].

Entonces, un *paso de narrowing* consiste de una unificación —entre un subtérmino no variable del término actual con el lado izquierdo de una regla— y luego una reducción —que reemplaza el subtérmino instanciado por el lado derecho instanciado de la regla (i.e., la aplicación de la regla)—. Formalmente, $t \leadsto_{(p,R,\sigma)} t'$ es un paso de narrowing si p es la posición de un subtérmino no variable del término t y $\sigma(t) \to_{p,R} t'$ es un paso de reescritura.

Una derivación narrowing se denota con $t_0 \leadsto_\sigma^* t_n$ si existe una secuencia de pasos de narrowing $t_1 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ con $\sigma = \sigma_n \circ \ldots \circ \sigma_1$ (si $n = 0$ entonces $\sigma = id$).

En la programación funcional estamos interesados en computar valores (términos en forma normal) y sus respuestas asociadas (sustituciones), por lo tanto sólo consideraremos las derivaciones que finalicen en términos en forma normal o encabezados por un símbolo constructor (en inglés, *head normal form*).

Ejemplo 3.5.1 *El siguiente TRS que define la función booleana "menor o igual que" aplicada en números naturales representados por términos construidos con los símbolos 0 (cero) y S (sucesor) (números de Peano):*

$$\begin{array}{lllll} \textit{(R1)} & \texttt{0} \leq \texttt{x} & \to & \texttt{true} \\ \textit{(R2)} & \texttt{S(x)} \leq \texttt{0} & \to & \texttt{false} \\ \textit{(R3)} & \texttt{S(x)} \leq \texttt{S(y)} & \to & \texttt{x} \leq \texttt{y} \end{array}$$

Desde el término $x \leq 0$*, narrowing efectúa las siguientes derivaciones:*

$$\texttt{x} \leq \texttt{0} \leadsto_{1,R1,\{x \mapsto 0\}} \texttt{true}$$

$$\texttt{x} \leq \texttt{0} \leadsto_{1,R2,\{x \mapsto S(x_1)\}} \texttt{S}(\texttt{x}_1) \leq \texttt{0} \leadsto_{1.1,R2,\{x_1 \mapsto 0\}} \texttt{false}$$

$$\texttt{x} \leq \texttt{0} \leadsto_{1,R2,\{x \mapsto S(x_1)\}} \texttt{S}(\texttt{x}_1) \leq \texttt{0} \leadsto_{1,R2,\{x_1 \mapsto S(x_2)\}} \texttt{S}(\texttt{S}(\texttt{x}_2)) \leq \texttt{0} \leadsto_{1.1.1,R2,\{x_2 \mapsto 0\}} \texttt{false}$$

$\ldots$

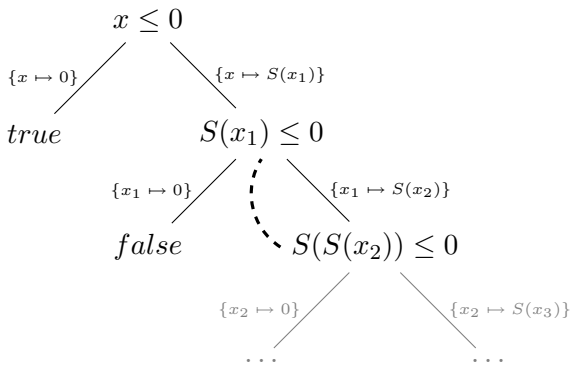

Figura 3.2: Árbol de narrowing de $x \leq 0$.

Luego, $x \leq 0$ *computa de forma no determinística los valores: true con respuesta* $\{x \mapsto 0\}$*, false con* $\{x \mapsto S(0)\}$*, false con* $\{x \mapsto S(S(0))\}$*, etc.*

Las derivaciones narrowing pueden representarse mediante árboles (posiblemente infinitos). La figura 3.2 muestra el árbol de narrowing construido desde la llamada inicial $x \leq 0$. Las ramas infinitas están en gris (Por ahora, ignoraremos las flechas con líneas de puntos).

Pese a que pueden ser infinitos, los pasos de una derivación narrowing pueden representarse de forma finita. Es posible identificar algunas derivaciones en el árbol que son instancias de derivaciones previas. En la figura 3.2 el nodo gris etiquetado con $S(S(x_2)) \leq 0$ (y sus descendientes) son cubiertos por $S(x_1) \leq 0$. Para prevenir derivaciones infinitas durante la construcción del árbol narrowing es necesario determinar si un término es cubierto por otro. Para este propósito, utilizamos la noción de *cerradura* (*closeness*). Diremos que un término t es cerrado con respecto a un conjunto de términos S, denotado por t-cerrado, si t es una instancia de $t' \in S$. La figura 3.2 muestra una flecha con línea de puntos desde $S(S(x_2)) \leq 0$ a $S(x_1) \leq 0$ que representa esta situación.

Además, es necesario representar todas las posibles derivaciones narrowing. Con el fin de computar todas las soluciones con la formulación original de narrowing, se deben aplicar todas las reglas a todos los subtérminos no variables en forma paralela (es decir, no determinísticamente). Esta *estrategia de narrowing* conduce a un enorme e infinito espacio de búsqueda.

Diremos entonces que una *estrategia de narrowing* determina la posición donde el siguiente paso de narrowing debería aplicarse con el fin de evitar computaciones innecesarias. Diversas mejoras a la estrategia original han sido propuestas (ver [Han94]). Sin embargo, *needed narrowing* [AEH00] es actualmente la mejor estrategia de narrowing perezoso debido a que realiza sólo aquellos pasos que son necesarios para computar las soluciones.

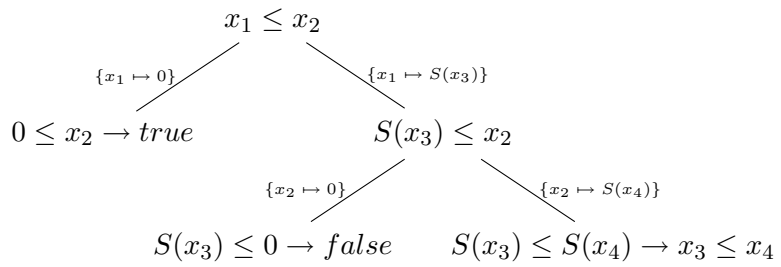

Figura 3.3: Árbol definicional del ejemplo 3.5.1.

Un paso de narrowing $t \leadsto_{(p,R,\sigma)} t'$ se denomina *necesario*[3] sii $\sigma(t) \rightarrow_{p,R} t'$ es un paso de reescritura necesario, i.e. en cada computación desde $\sigma(t)$ a un término en forma normal, debe ocurrir que $\sigma(t)|_p$ o uno de sus *descendientes*[4] debe ser reducido.

Esta estrategia se basa en una estructura de datos denominada *árbol definicional* [Ant92] —i.e., una estructura de datos jerárquica que contiene todas las reglas que definen a una función— y que es usada para guiar los pasos de narrowing. En [Han97] se presenta un algoritmo para la construcción de árboles definicionales.

Ejemplo 3.5.2 *Consideremos nuevamente el TRS del ejemplo 3.5.1 y el término inicial* $0 \leq t$*, donde* t *es cualquier término (e.g., una llamada a función). La figura 3.3 muestra el árbol definicional construido a partir del TRS. Debido a que* 0 *es el primer argumento de la llamada de la función inicial, entonces se evalúa* $0 \leq t$ *con la regla* $0 \leq x_2 \rightarrow true$*. Por lo tanto, es posible aplicar la regla sin evaluar el término* t*.*

3.6. Programación Funcional

Es usual ver a los programas funcionales como una clase restringida de sistemas de reescritura. Básicamente, un programa funcional es un conjunto de ecuaciones de la forma $f_i(x_1, ..., x_k) = t_i$ con $i = 1, ..., n$, tal que:

- $\mathcal{V}ar(t_i) \subseteq \{x_1, ..., x_k\}$[5],
- todas las variables $x_1, ..., x_k$ son distintas.

En este trabajo, nos limitaremos a TRS *basados en constructores* y *lineales a izquierda* [Ant92], a los que llamaremos *programas*. Además, cada función tiene asociado un árbol definicional [BN98, Klo92].

[3]En el sentido de Huet y Lévy [HL92].

[4]Un concepto de *descendientes* puede encontrarse en el trabajo de Klop y Middeldorp [KM91].

[5]En lo sucesivo, escribiremos $\overline{o_n}$ para denotar la *secuencia de objetos* $o_1, \ldots, o_n$.

$$
\begin{array}{rrcl}
\textbf{(program)} & \mathcal{R} & ::= & \mathcal{D}_1 \ldots \mathcal{D}_n \\
\textbf{(functions)} & \mathcal{D} & ::= & f(\overline{x_n}) = e \\
\textbf{(expressions)} & e & ::= & t \mid \texttt{case}\ t\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\} \\
\textbf{(terms)} & t & ::= & x \mid c(\overline{t_n}) \mid f(\overline{t_n}) \\
\textbf{(patterns)} & p & ::= & c(\overline{x_n})
\end{array}
$$

Figura 3.4: Sintaxis del lenguaje funcional de primer orden utilizado a lo largo del trabajo.

$$
\begin{array}{rl}
\textbf{(fun eval)} & \texttt{f}(\overline{t_n}) \Longrightarrow_{id} \sigma(e) \\
& \text{if } \texttt{f}(\overline{x_n}) = e \in \mathcal{R}, \text{ and } \sigma = \{\overline{x_n \mapsto t_n}\} \\
\textbf{(case select)} & \texttt{case}\ c(\overline{t_n})\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\} \Longrightarrow_{id} \sigma(e_i) \\
& \text{if } p_i = c(\overline{x_n}),\ c \in \mathcal{C}, \text{ and } \sigma = \{\overline{x_n \mapsto t_n}\} \\
\textbf{(case guess)} & \texttt{case}\ x\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\} \Longrightarrow_{\sigma} \sigma(e_i) \\
& \text{if } \sigma = \{x \mapsto p_i\}, \text{ and } i \in \{1, \ldots, k\} \\
\textbf{(case eval)} & \texttt{case}\ t\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\} \Longrightarrow_{\sigma} \sigma(\texttt{case}\ t'\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\}) \\
& \text{if } t \text{ is not in } \textit{head normal form} \text{ and } t \Longrightarrow_{\sigma} t'
\end{array}
$$

Figura 3.5: Semántica operacional (basada en cálculo LNT).

Decimos que un TRS $\mathcal{R}$ está *basado en constructores* si los lados izquierdos de sus reglas tienen la forma $f(s_1, \ldots, s_n)$ y cada uno de los s_i son términos constructores, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, para todo $i = 1, \ldots, n$. Un término t está enraizado por una operación si $root(t) \in \mathcal{D}$ (en inglés, se conoce como *operation-rooted*). Un término t está enraizado por un símbolo constructor si $root(t) \in \mathcal{C}$ (en inglés, se conoce como *constructor-rooted*).

Un TRS $\mathcal{R}$ es *linear a izquierda* si todos los lados izquierdos de sus reglas no contienen apariciones múltiples de la misma variable.

Ejemplo 3.6.1 *El TRS del ejemplo 3.4.1 es linear a izquierda ya que en cada lado izquierdo de sus reglas no existen apariciones múltiples de las mismas variables.*

3.7. Lenguaje Funcional

La figura 3.4 muestra la sintaxis del lenguaje funcional que utilizaremos a lo largo del trabajo. La sintaxis establece que un programa $\mathcal{R}$ es una secuencia de definiciones de función $\mathcal{D}_i$. Cada función se define mediante una única regla de tal forma que el lado izquierdo tiene *variables* como argumentos, y el lado derecho es una *expresión*. Las expresiones se construyen con *términos* y expresiones *case*. Las

expresiones *case* se utilizan para realizar el emparejamiento por patrones (pattern-matching) donde cada patrón p_k es un constructor del tipo de x. Los términos pueden ser *variables* (e.g., x, y, z), *constructores* (e.g., A, B, C), o *llamadas a función* (e.g., f, g, h).

Ejemplo 3.7.1 *La función booleana "menor o igual que" representada como un TRS en el ejemplo 3.5.1 puede especificarse en este lenguaje como:*

```
x ≤ y = case x of
        Z    → true
        S(n) → case y of
                 Z    → false
                 S(m) → n ≤ m
```

La semántica del lenguaje se muestra en la figura 3.5. Está basada en el cálculo *narrowing perezoso con árboles definicionales* (*Lazy Narrowing with definitional Trees* (LNT)) [HP99, AHV03]. Concisamente, describimos las reglas de la semántica:

fun eval: realiza el *desplegado* (*unfolding*) de una llamada a una función. Las variables son renombradas en cada paso de evaluación.

case select: esta regla implementa el pattern-matching y selecciona la rama apropiada del *case* cuando el argumento del mismo es un término enraizado con un constructor (constructor-rooted).

case guess: si el argumento del *case* es una variable x, entonces es posible aplicar (no determinísticamente) cualquiera de las n reglas del mismo.

case eval: si el argumento del *case* es una llamada a una función, entonces dicha función debe evaluarse en primer lugar.

La relación de transición $e \Longrightarrow_\sigma e'$ se etiqueta con la sustitución computada en cada paso. Una *derivación LNT*, denotada $e_0 \Longrightarrow_\sigma^* e_n$, es una secuencia $e_0 \Longrightarrow_{\sigma_1} \ldots \Longrightarrow_{\sigma_n} e_n$, donde $\sigma = \sigma_n \circ \ldots \circ \sigma_1$, y es *exitosa* cuando e_n está encabezada por un símbolo constructor (head normal form).

Ejemplo 3.7.2 *Utilizando el programa del ejemplo 3.7.1 y la llamada inicial* `Z ≤ y`*, el cálculo LNT computa la siguiente derivación:*

```
Z ≤ y ⟹id case Z of                    (fun eval)
        Z → true
        S(n) → case y of
                 Z → false
                 S(m) → n ≤ m
     ⟹id true                          (case select)
```

4

Slicing Condicional de Programas Funcionales

En este capítulo presentamos la técnica para slicing condicional de programas funcionales de primer orden. A continuación, introduciremos el concepto de slice de un programa. Este concepto nos permitirá presentar nuestra aproximación a la definición de slicing condicional y un método para su cómputo. Como hemos afirmado en las secciones previas de este trabajo, la definición de slicing condicional en el ámbito declarativo no ha sido tratada hasta el momento. Por ese motivo, este trabajo resulta un aporte original.

4.1. Slice de un programa

En general, diremos que un slice es un subprograma obtenido desde el programa original mediante la eliminación de uno o más términos de éste. A continuación presentamos formalmente el concepto de slice que utilizaremos en el trabajo. Para tal fin, introducimos un constructor especial $\perp$ para denotar los términos eliminados en el lado derecho de la definición de una función. Simbólicamente,

Definición 4.1.1 (Slice [SV07]) *Dado un programa* $\mathcal{R} = \overline{\mathcal{D}_n}$ *con* $\mathcal{D}_i = (f(\overline{x_m}) = e)$*, diremos que el programa* $\mathcal{R}' = \overline{\mathcal{D}'_n}$ *con* $\mathcal{D}'_i = (f(\overline{x_m}) = e')$ *es un slice de* $\mathcal{R}$*, y escribiremos* $\mathcal{R}' \preceq \mathcal{R}$*, sii* $e' \preceq e$ *donde*

$$e' \preceq e \;=\; \begin{cases} true & si\ e' = \perp\ o\ e' = e \\ true & si\ e' = \text{case } x \texttt{ of } \{\overline{p_k \mapsto e'_k}\}, \\ & \quad e = \text{case } x \texttt{ of } \{\overline{p_k \mapsto e_k}\},\ y \\ & \quad e'_k \preceq e_k \text{ para todo } k. \\ false & en\ otro\ caso \end{cases}$$

La definición establece que un slice de un programa es un subprograma obtenido desde el original en el que pueden haberse eliminado expresiones del lado derecho de la definición de una función, términos o ramas de un *case*.

En el paradigma imperativo, un *slice condicional* es un slice o un subprograma que incluye todos los caminos que pueden producirse durante la ejecución de un programa cuyos argumentos de entrada cumplen cierta condición lógica de primer orden. Por lo tanto, las ejecuciones del programa y de su slice condicional, siempre que se efectúen con valores para las entradas que satisfagan la condición establecida, son indistinguibles —i.e., ambos programas presentan el mismo comportamiento—. Por ese motivo, nuestra definición de slicing en el paradigma funcional debe contemplar esta característica. A continuación, definimos el criterio de slicing y luego, presentamos la noción de slicing condicional.

4.2. Slice Condicional

En primer lugar, introducimos nuestro *criterio de slicing condicional*. El criterio es una tupla que permite especificar una llamada a una función inicial, una fórmula lógica que satisfacen los argumentos de entrada de la función, y un patrón que indica cuales partes del resultado son de interés.

Definición 4.2.1 (Criterio Slicing de Condicional) *Dado un programa $\mathcal{R}$, el criterio de slicing condicional para $\mathcal{R}$ es la tupla $(f(\overline{x_n}), \phi, \pi)$ donde*

- *f es un símbolo de función tal que $f(\overline{x_n}) = e \in \mathcal{R}$,*
- *ϕ es una fórmula lógica de primer orden sobre los argumentos $x_1, \ldots, x_n$ de la función f, y*
- *π es un* patrón de slicing *[OSV04] definido como sigue:*

$$\pi \ \in \ \bot \mid \top \mid c(\overline{\pi_k})$$

donde

  - *c es un símbolo constructor de aridad $k \geq 0$,*
  - *$\bot$ denota una subexpresión cuya computación no es relevante y*
  - *$\top$ es una subexpresión relevante.*

Los patrones de slicing son similares a los *patrones de vivacidad* (*liveness patterns*) usados para la eliminación de código muerto en [LS03]. Básicamente, son términos abstractos que usaremos para determinar qué parte de la estructura de un término constructor se ignora y cuál resulta relevante para el slicing. Por ejemplo, para el siguiente término constructor $\mathtt{C}(\mathtt{A}, \mathtt{B})$ —dependiendo de la información disponible y de los fragmentos que se consideren importantes— algunos casos de patrones que podríamos emplear son: $\top$, $\bot$, $\mathtt{C}(\top, \top)$, $\mathtt{C}(\top, \bot)$, $\mathtt{C}(\bot, \top)$, $\mathtt{C}(\bot, \bot)$, $\mathtt{C}(\mathtt{A}, \top)$, $\mathtt{C}(\mathtt{A}, \bot)$, $\mathtt{C}(\top, \mathtt{B})$, $\mathtt{C}(\bot, \mathtt{B})$, $\mathtt{C}(\mathtt{A}, \mathtt{B})$.

Además, es necesario contar con una *función de concretización* γ que se aplicará a los términos abstractos. El resultado de $\gamma(\pi)$ será un conjunto (normalmente infinito) de términos que pueden ser obtenidos desde π mediante el reemplazo de todas las ocurrencias de $\top$ y $\bot$ por cualquier término constructor. Por ejemplo, $\gamma(\mathtt{C}(\mathtt{A},\top)) = \gamma(\mathtt{C}(\mathtt{A},\bot)) = \{\mathtt{C}(\mathtt{A},\mathtt{A}), \mathtt{C}(\mathtt{A},\mathtt{B}), \mathtt{C}(\mathtt{A},\mathtt{D}), \mathtt{C}(\mathtt{A},\mathtt{C}(\mathtt{A},\mathtt{A})), \mathtt{C}(\mathtt{A},\mathtt{C}(\mathtt{A},\mathtt{B})), \ldots\}$.

Informalmente, diremos que un programa $\mathcal{R}'$ es un *slice condicional* de $\mathcal{R}$ con respecto al criterio $(f(\overline{x_n}), \phi, \pi)$ si y sólo si se cumple:

- $\mathcal{R}'$ es un slice de $\mathcal{R}$ de acuerdo a la definición 4.1.1,
- todos los argumentos satisfacen la condición establecida ϕ,
- la ejecución de $\mathcal{R}$ y de $\mathcal{R}'$ con sus argumentos satisfaciendo la condición son indistinguibles, y
- sólo estamos interesados en aquellas partes del resultado v de la llamada a función $f(\overline{x_n})$ de acuerdo al patrón de slicing π.

Definición 4.2.2 (Slice Condicional) *Dado un programa $\mathcal{R}$, y un criterio condicional de slicing $SC = (f(\overline{x_n}), \phi, \pi)$, $\mathcal{R}'$ es un slice condicional con respecto a $(f(\overline{x_n}), \phi, \pi)$, en símbolos $\lhd(\mathcal{R}, SC)$, sii*

1. *$\mathcal{R}' \preceq \mathcal{R}$, y*
2. *$\overline{x_n}$ satisface ϕ, $f(\overline{x_n}) \Longrightarrow^*_\sigma v$ con $\mathcal{R}$, $f(\overline{x_n}) \Longrightarrow^*_{\sigma'} v'$ con $\mathcal{R}'$, donde v y v' son valores encabezados por un constructor (i.e., head normal form) tal que $v = v'$, y $\sigma = \sigma'$, y*
3. *$v' \in \gamma(\pi)$ y para todos los subtérminos $v|_p$ y $\pi|_q$ / $p = q \wedge \pi|_q \neq \bot$ donde $\gamma(\pi)$ es una concretización de π —i.e. v' tiene la* forma *de π—.*

Entonces, un slice condicional $\mathcal{R}'$ es un subprograma de $\mathcal{R}$ en el que se han sustituido por $\bot$ (o borrado) cero o más funciones, términos o ramas de expresiones *case* debido a que no forman parte de ninguna derivación iniciada desde la función $f(\overline{x_n})$ cuyos argumentos cumplen la condición ϕ. Además, únicamente se consideran aquellas partes relevantes del resultado v de acuerdo con π. Una característica fundamental es que ambos programas $\mathcal{R}$ y $\mathcal{R}'$ presentan idéntico comportamiento cuando se ejecutan con entradas que satisfacen ϕ —i.e. se evalúan al mismo resultado v—.

Ejemplo 4.2.3 *Consideremos el programa $\mathcal{R}$ del ejemplo 3.7.1, el slice condicional $\lhd(\mathcal{R}, SC)$ con $SC = (x \leq y, \phi, \pi)$, donde $\phi = y > Z$ y $\pi = \top$, es el subprograma $\mathcal{R}'$:*

```
x ≤ y = case x of
          Z    → true
          S(n) → case y of
                   Z    → false
                   S(m) → n ≤ m
```

La regla `Z → false` *del* case*, que aparece marcada en gris, puede reemplazarse por* `Z` $\rightarrow \bot$ *—i.e. puede ser borrada— debido a que no existe ninguna derivación a la que pueda pertenecer dicha rama cuando las entradas del programa satisfacen la condición* ϕ*. Es decir, la expresión no es necesaria para computar el resultado del programa dada la llamada inicial* $y > Z$*. Por otra parte,* $\pi = \top$ *indica que estamos interesados en todo el resultado.*

Ejemplo 4.2.4 *Consideremos ahora el programa* $\mathcal{R}$ *de la figura 2.16 y su slice condicional* $\lhd(\mathcal{R}, SC)$ *con* $SC = (main, \phi, \pi)$*, donde* $\phi = \{x == y\}$ *y* $\pi = Pair \bot \top$*.*

Todas aquellas partes del programa que aparecen en gris pueden ser borradas debido a que no son necesarias para computar el resultado desde la llamada inicial main con sus argumentos satisfaciendo $\phi = \{x == y\}$ *—i.e., cuando los triángulos son isósceles—. Además,* $\pi = Pair \bot \top$ *indica que sólo estamos interesados en la segunda parte del par que se genera como resultado de la función —i.e., la clasificación del triángulo—.*

4.3. Cálculo de Slices Condicionales

Pettorossi et al [PP96] establecieron una metodología común para las técnicas de transformación de programas basándose en las similitudes entre ellas e identificando los siguientes pasos fundamentales de dichas transformaciones:

- Computación simbólica,
- búsqueda de regularidades, y
- extracción del programa.

El slicing de programas es un tipo de transformación de programas y, por lo tanto, puede caracterizarse con los pasos mencionados anteriormente.

La forma natural de realizar *computación simbólica* en los lenguajes multiparadigmas es a través de narrowing ya que éste constituye el mecanismo de computación sobre el que se basan. En particular, estamos interesados en construir árboles de narrowing finitos —comenzando desde una llamada inicial—

para representar todas las posibles ejecuciones y emplearlos con el fin de guiar el proceso de slicing. Debido a que las ejecuciones pueden ser infinitas es necesario asegurar que la construcción (iterativa) de los árboles de narrowing termina. En este paso se requiere la *búsqueda de regularidades* para garantizar la finalización de la construcción de los árboles de narrowing usando alguna condición de terminación (e.g., análisis de covertura). Por último, en la etapa de *extracción del programa*, un programa es generado a partir de los árboles de narrowing obtenidos. El proceso descripto anteriormente es una extensión de la técnica de slicing dirigida por narrowing explicada en [SV07].

En el paper introductorio de la técnica de slicing condicional, Canfora et al [CCL98] establecen que un slice condicional de un programa es equivalente al slice (forward) de su programa condicionado —i.e., el programa ejecutado de manera simbólica considerando la condición lógica establecida—. Por este motivo, proponemos un proceso de slicing formado por dos fases:

- Extendemos la semántica del lenguaje con el fin de recolectar aquellas partes del programa que pueden ser ejecutadas cuando una fórmula lógica se cumple. Entonces, ejecutamos simbólicamente el programa desde la función inicial empleando la semántica extendida y considerando la condición lógica. El resultado de este proceso es un programa condicionado que contiene un conjunto de partes que podrían ser ejecutadas cuando la condición establecida es verdadera. Esta aproximación se presentará en la siguiente sección.

- Una vez que el programa es simplificado con respecto a la condición lógica sobre las entradas del programa, se procede a realizar un slicing (forward) del program condicionado con respecto al slicing pattern establecido con el fin de obtener el slice condicional [CCL98]. Para este fin, utilizaremos la técnica de slicing estático definida en [CSV07].

De acuerdo a lo anterior, es posible descomponer el criterio de slicing condicional $(function, \phi, \pi)$ en dos partes. Por un lado, en la primera fase de nuestra aproximación, consideramos un criterio de slicing compuesto por $(function, \phi)$ para obtener el programa condicionado. En la segunda fase, usamos $(function, \pi)$ como criterio de slicing para efectuar un slicing (forward) del programa obtenido en la primera fase.

4.3.1. Extendiendo la semántica para computar slices condicionales

Con el fin de computar el slice de un programa con respecto al criterio de slicing $(function, \phi)$, extenderemos la semántica de la figura 3.5 de manera que

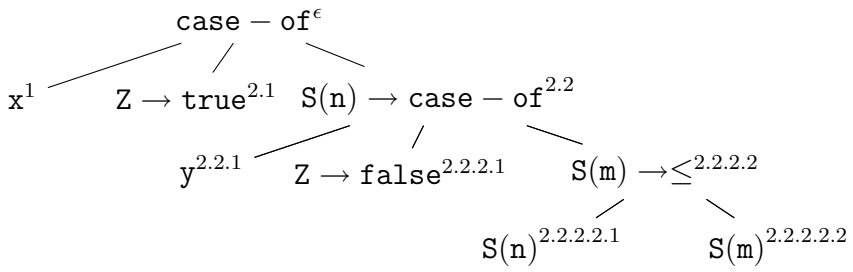

Figura 4.1: Posiciones de programa del lado derecho del ejemplo 3.7.1.

sea posible recolectar aquellas *posiciones de programa* [OSV04] correspondientes al lado derecho de cada regla del programa capaces de ser ejecutadas cuando la fórmula lógica ϕ se cumple —i.e., aquellas partes del programa que no deberían ser eliminadas o reemplazadas por $\perp$ porque son necesarias para realizar una ejecución que satisface ϕ—.

A continuación, presentamos la definición de *posiciones de programa* y un ejemplo explicativo del concepto.

Definición 4.3.1 (Posición de Programa [OSV04]) *Una* posición de programa *es un par* (f, q) *donde* f *es un símbolo de función tal que* $f(\overline{x_n}) = e$ *y* q *es la posición de un subtérmino de* e*.*

La figura 4.1 muestra las posiciones de programa de las reglas del ejemplo 3.7.1.

En nuestra extensión de la semántica, tanto el lado izquierdo como el derecho de cada regla de la misma estará constituida por una tupla de la forma $\langle e, f, P, \phi \rangle$, donde

- e es una expresión a ser evaluada,
- f es el símbolo de función a la cual la expresión e pertenece,
- P es un conjunto que acumula las posiciones de programa, y
- ϕ es la condición de slicing.

La figura 4.2 muestra la semántica extendida usada para recolectar las *posiciones de programa* de aquellas partes relevantes con respecto al criterio de slicing (f, ϕ).

La semántica extendida y la original son muy similares. De hecho, la extensión propuesta preserva el comportamiento estándar ya que no se ha impuesto ninguna restricción a la semántica original.[1]

[1]Debido a esto también puede suceder que un programa ejecutado bajo la semántica extendida entre en un bucle infinito.

(fun eval) $\langle f(\overline{t_n}), g, P, \phi\rangle \Longrightarrow_{id} \langle \sigma(e), f, pp \cup P, \phi\rangle$
if $f(\overline{x_n}) = e \in \mathcal{R}$, $\sigma = \{\overline{x_n \mapsto t_n}\}$, and
if $f \in SC$ and $\phi \Rightarrow \sigma(\overline{x_n})$ then $pp = \{(f, \epsilon)\}$ else $pp = \emptyset$

(case select) $\langle case\ c(\overline{t_n})\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\}, g, P, \phi\rangle \Longrightarrow_{id} \langle \sigma(e_i), g, pp \cup P, \phi\rangle$
if $p_i = c(\overline{x_n})$, $c \in \mathcal{C}$, and $\sigma = \{\overline{x_n \mapsto t_n}\}$ then $pp = \{(g, 2.j) \mid j \neq i\}$

(case guess) $\langle case\ x\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\}, g, P, \phi\rangle \Longrightarrow_{\sigma} \langle \sigma(e_i), g, pp \cup P, \phi'\rangle$
if $\sigma = \{x \mapsto p_i\}, i \in \{1, \ldots, k\}$, and
if $g \in SC$ and $\phi \Rightarrow \sigma(x)$ then $pp = \{(g, 2.i)\}$, and $\phi' = \phi \wedge (x = p_i)$
else $pp = \emptyset$

(case eval) $\langle case\ t\ \texttt{of}\ \{\overline{p_k \rightarrow e_k}\}, g, P, \phi\rangle \Longrightarrow_{\sigma} \langle \sigma(case\ t'\ \texttt{of}\ \{\overline{p_k \mapsto e_k}\}), g, P, \phi\rangle$
if t is not in *head normal form* and $t \Longrightarrow_{\sigma} t'$

Figura 4.2: Semántica operacional extendida (basada en Cálculo LNT).

A continuación describimos brevemente las reglas de la semántica:

fun eval: realiza el desplegado (unfolding) de una llamada a función. Cuando la función que se está desplegando pertenece al criterio de slicing y el predicado ϕ es satisfecho por los valores de los argumentos mapeados por la sustitución σ, entonces recolecta la posición raíz de f —esto implica que la función puede ser ejecutada—.

case select: selecciona la rama apropiada del *case* cuando el argumento del *case* es un término encabezado por un constructor, y recolecta las posiciones de programa correspondientes a la rama seleccionada.

case guess: si el argumento de la expressión *case* es una variable x, entonces es posible aplicar (no determinísticamente) cualquiera de las n reglas que definen al *case*. Cuando la función que se está desplegando pertenece al criterio de slicing y el predicado ϕ es satisfecho por los valores de los argumentos mapeados por la sustitución σ, entonces se recolectan las posiciones de programa de la rama seleccionada —esto implica que puede ser ejecutada—. Caso contrario, la regla de la semántica se comporta de manera equivalente con la original. Además, con el fin de incrementar la precisión del proceso de slicing, si el predicado ϕ es satisfecho por los valores mapeados por la sustitución, entonces ϕ es aumentada con $x = p_i$.

case eval: si el argumento del *case* es una llamada a función, entonces ésta debe evaluarse primero.

Con el propósito de realizar las computaciones necesarias, el proceso comienza desde un estado inicial y el cálculo computa de forma indeterminista —mediante la aplicación de las reglas de la figura 4.2— todas las derivaciones *exitosas*.

$\langle x \leq y, \leq, \emptyset, \phi \rangle$
$\Longrightarrow_{id}$
$\langle$`case x of {Z→true;S(n)→case y of {Z→false; S(m)→n ≤ m}}`$, \leq, \{(\leq, \epsilon)\}, \phi\rangle$ (function eval)
$\Longrightarrow_{\{x \mapsto Z\}}$
$\langle$`true`$, \leq, \{(\leq, \epsilon), (\leq, 2.1)\}, \phi\rangle$ (case guess)

$\langle x \leq y, \leq, \emptyset \rangle$
$\Longrightarrow_{id}$
$\langle$`case x of {Z→true;S(n)→case y of {Z→false;S(m)→n ≤ m}}`$, \leq, \{(\leq, \epsilon)\}, \phi\rangle$ (fun eval)
$\Longrightarrow_{\{x \mapsto S(x_1)\}}$
$\langle$`case y of {Z→false; S(m)→n ≤ m}`$, \leq, \{(\leq, \epsilon), (\leq, 2.2)\}, \phi \wedge x = S(x_1)\rangle$ (case guess)
$\Longrightarrow_{\{y \mapsto Z\}}$
$\langle$`false`$, \leq, \{(\leq, \epsilon), (\leq, 2.2)\}, \phi \wedge x = S(x_1) \wedge y = Z\rangle$ (case guess)

$\langle x \leq y, \leq, \emptyset \rangle$
$\Longrightarrow_{id}$
$\langle$`case x of {Z→true;S(n)→case y of {Z→false;S(m)→n ≤ m}}`$, \leq, \{(\leq, \epsilon)\}, \phi\rangle$ (fun eval)
$\Longrightarrow_{\{x \mapsto S(x_1)\}}$
$\langle$`case y of {Z→false; S(m)→n ≤ m}`$, \leq, \{(\leq, \epsilon), (\leq, 2.2)\}, \phi \wedge x = S(x_1)\rangle$ (case guess)
$\Longrightarrow_{\{y \mapsto S(x_2)\}}$
$\langle$`x`$_1 \leq$ `x`$_2, \leq, \{(\leq, \epsilon), (\leq, 2.2), (\leq, 2.2.2.2)\}, \phi \wedge x = S(x_1) \wedge y = S(x_2)\rangle$ (case guess)
...

Figura 4.3: Un ejemplo de derivación empleando la semántica extendida.

El *estado inicial* $I = \langle f(\overline{x_n}), f, \emptyset, \phi \rangle$ donde $f(\overline{x_n})$ pertenece a $SC = (f(\overline{x_n}), \phi)$, `f` es el símbolo de función actual, el conjunto de posiciones de programa se inicializa con el conjunto vacío y ϕ es el predicado que debe ser satisfecho por los argumentos de entrada $\overline{x_n}$ de la función inicial.

El *estado final* F tiene la forma $\langle e, g, P, \phi \rangle$ donde `e` está en head normal form, g es el símbolo de función actual, y P contiene las posiciones de programa identificadas como necesarias.

Ejemplo 4.3.2 *Nuevamente utilizaremos el programa $\mathcal{R}$ del ejemplo 3.7.1 y el criterio de slicing $SC = (x \leq y, \phi)$ con $\phi = y > Z$, las derivaciones desde $\langle x \leq y, \leq, \emptyset \rangle$ se muestran en la figura 4.3.*

Las posiciones de programa recolectadas son $\{(\leq, \epsilon), (\leq, 2.1), (\leq, 2.2), (\leq, 2.2.2.2)\}$. Estas posiciones de programa corresponden a aquellas partes del programa que pueden ser ejecutadas cuando ϕ se cumple. Sin embargo, la posición de programa $\{(\leq, 2.2.2.1)\}$ puede eliminarse ya que no es necesaria y no aparecerá en ninguna derivación desde la llamada inicial con sus argumentos satisfaciendo ϕ.

4.3.2. Algoritmo de Slicing Condicional

En la sección 4.3 sucintamente delineamos la idea básica de cómo computar slices condicionales basándonos en un proceso de tres etapas. En esta sección, describiremos con mayor detalle este algoritmo inspirado en el método presentado en [SV07]. La figura 4.4 presenta un pseudocódigo de tal proceso.

```
Input: programa R y SC = (f(x_n), φ)
Output: slice del programa S
i = 0;
S_i = { f(x_n) };
repeat
   R' = unfold(S_i, R, φ);
   S_{i+1} = abstract(S_i, R');
   i = i + 1;
until (S_i = S_{i-1});
return  build_slice(S_i,R);
```

Figura 4.4: Algoritmo de Slicing Conditional.

Básicamente, dado un programa $\mathcal{R}$ y el criterio de slicing $(f(\overline{x_n}), \phi)$ con $\overline{x_n}$ (parcialmente) instanciado, el algoritmo procede desplegando (aplicando el operador de *unfolding*) y obteniendo los árboles de narrowing de aquellas partes del programa que son alcanzables desde la llamada inicial $f(\overline{x_n})$ y que satisfacen ϕ.

Luego, la operación de abstracción (*abstract*) asegura que el número de árboles sea finito añadiendo al conjunto actual S_i de llamadas a función sólo aquellas que aún no han sido consideradas.

A continuación, definiremos las funciones *unfold* y *abstract* y sus problemas de terminación inherentes.

El operador de desplegado *unfold* calcula un árbol de narrowing finito comenzando desde $f(\overline{x_n})$. Con la finalidad de detener la construcción de los árboles de narrowing, el proceso debe decidir si un término será evaluado (o no) en el siguiente paso de derivación cuando éste no es cubierto por otros términos en el árbol. Para este fin, empleamos la relación de *inclusión homeomórfica* (*homeomorphic embedding*) [Leu98] para determinar cuándo un término está embebido por otro o no. Informalmente, s está embebido en t, $s \triangleright t$, si s puede ser obtenido desde t borrando ciertas partes de t —i.e., borrando operadores de t—.

Definición 4.3.3 (Unfold) *Dado un conjunto T de estados, un programa R y una condición ϕ, la función unfold se define de la siguiente manera:*

$$unfold(T, \mathcal{R}, \phi) = \bigcup unfold'(t, \emptyset) \text{ para cada } t \in T$$

$$unfold'(\langle t, f, S, \phi\rangle, D) = \begin{cases} \{\langle t', f', S', \phi'\rangle\} & si\ \langle t, f, S, \phi\rangle \Longrightarrow \langle t', f', S', \phi'\rangle \\ & y\ t' \triangleright\ s \text{ donde } s \in D \\ \{\langle t', f', S', \phi'\rangle\} \cup & \\ unfold'(\langle t', f', S', \phi'\rangle, D \cup t') & si\ \langle t, f, S, \phi\rangle \Longrightarrow \langle t', f', S', \phi'\rangle \\ & y\ t' \not\triangleright s \text{ para cada } s \in D \end{cases}$$

La función *abstract* toma el conjunto S_i de términos evaluados y el conjunto $\mathcal{R}'$ de términos producidos por unfolding para generar un nuevo conjunto S_{i+1}. Este nuevo conjunto es una aproximación segura de $S_i \cup \mathcal{R}'$ tal que el término t de cada estado $\langle t, g, P, \phi\rangle \in S_i \cup \mathcal{R}'$ es $t-$closed con respecto a S_{i+1}. Es decir, cada término t es una instancia de otro término t' cuyo estado pertenece a S_{i+1}.

Definición 4.3.4 (Abstract) *Dado un conjunto S de estados y otro conjunto R de estados a ser agregados a S, la función de abstracción se define como sigue:*

$$abstract(S, R) = \bigcup\ abstract'(S, t)\ para\ cada\ t\ \in\ R$$

$$abstract'(S, t)\ =\ \begin{cases} \{t\} & si\ t \not\triangleright t'\ para\ cada\ t' \in S \\ \emptyset & si\ t \triangleright t'\ y\ t\ es\ t\text{-closed} \\ abstract'(S, msg(t, t')) & en\ otro\ caso \end{cases}$$

donde $msg(t, t')$ es la generalización más específica.[2]

Una vez concluido el proceso de desplegado-abstracción (unfold-abstract), el conjunto S_i de estados incluye todas las posibles computaciones desde la llamada inicial $f(\overline{x_n})$.

Por lo tanto, es posible extraer el programa condicionado eliminado aquellas posiciones de programa que no satisfacen la condición ϕ.

Ejemplo 4.3.5 *Consideremos el programa $\mathcal{R}$ del ejemplo 3.7.1, el slice condicional $\lhd(\mathcal{R}, SC)$ con $SC = (x \leq y, \phi, \pi)$, donde $\phi = y > Z$ y $\pi = \top$, es el subprograma $\mathcal{R}'$:*

```
x ≤ y = case x of{ϵ}
          Z    → true{2.1}
          S(n) → case y of{2.2}
                   Z    → false{2.2.2.1}
                   S(m) → n ≤ m{2.2.2.2}
```

Para este ejemplo, el algoritmo recolecta las posiciones de programa $\{(\leq, \epsilon), (\leq, 2.1), (\leq, 2.2), (\leq, 2.2.2.2)\}$ y elimina aquellas que no pertencen a dicho conjunto —i.e., la posición $\{\leq, 2.2.2.1\}$ no es considerada en el slice debido a que no se ejecutará esa rama del case *siempre que se cumpla ϕ.*

[2]Un término t es la generalización de t' y t'' si ambos son instancias de t. Además, t es la generalización más específica si dada cualquier otra generalización s de t' y t'', t es una instancia de s.

Definición 4.3.6 (build_slice) *Dado un conjunto S de estados y un programa $\mathcal{R}$, el slice se construye de la siguiente manera:*

$$build\_slice(S,R) \;=\; \begin{cases} R & si\ S = \emptyset \\ f(\overline{x_n}) = e' & en\ otro\ caso \\ & donde\ f(\overline{x_n}) \in R, \\ & e' = deleteProgPos(PP, R), \\ & PP = extractProgPos(S) \end{cases}$$

La función *extractProgPos* toma un conjunto de estados y extrae los componentes correspondientes a la posición de programa; y la función *deleteProgPos* recibe como entrada un conjunto de posiciones de programa de interés PP y un programa R con el objetivo de eliminar aquellas posiciones de programa p de R tales que $p \notin PP$.

Claramente, existen dos problemas de terminación en el algoritmo:

- La terminación del proceso de unfolding (termination local), y
- la terminación del ciclo de repeat-until (terminación global).

Ambos problemas de terminación son abordados usando la relación de inclusión homeomórfica. En la terminación local es necesario asegurar que cada derivación de narrowing (posiblemente infinita) termina. Entonces, empleando inclusión homeomórfica, se decide cuando un término t debe ser evaluado en el siguiente paso de derivación. Si t es cubierto por otro térmio en el árbol, entonces el proceso es detenido; caso contrario, t es evaluado.

En la terminación global debemos garantizar que el número de árboles generados sea finito. Para conseguir esto también utilizamos la inclusión homeomórfica. En este caso, la función de abstracción —que controla la terminación global— decide cuándo agregar un nuevo término para desplegar, cuándo aplicar una generalización, o detener el proceso de construcción de árboles de narrowing (i.e., cuando no hay más funciones para desplegar).

4.3.3. Forward slicing

Una vez que el programa se simplifica con respecto a la condición ϕ, debemos obtener el slice (forward) a partir del programa condicionado teniendo en cuenta el patrón de slicing π para, finalmente, conseguir el slice condicional. Por lo tanto, el criterio de slicing considerado en la aplicación del slicing es $(function, \pi)$.

Por ejemplo, en el programa de la figura 2.16 el programa condicionado que obtuvimos de la primera fase contiene todas las partes del mismo que pueden ser ejecuadas cuando sus entradas satisfacen la condición $x == y$. Pero, estamos

únicamente interesados en la segunda parte del par de acuerdo al patrón de slicing $Pair\top\bot$ —i.e., la clasificación del triángulo y no el perímetro—. Para realizar el slicing forward empleamos la aproximación descripta en [CSV07] que devuelve un slice conteniendo aquellas partes del programa que son alcanzables desde el criterio de slicing $(main, Pair\top\bot)$.

4.4. Slicing Estático y Dinámico

La importancia del slicing condicional radica en el hecho de que provee un marco común para realizar slicing estático y dinámico [CCL98]. Es decir, la técnica subsume a ambas.

De esta forma, el slice estático puede computarse usando el criterio de slicing condicional como $(function, True, \pi)$.

Por otra parte, el slice dinámico puede ser calculado con el criterio $(function, \phi, \pi)$ donde ϕ se compone de valores concretos para los argumentos de entrada en cuenta de una fórmula lógica.

Por ejemplo, el slice estático para el programa de la figura 2.14 puede ser representado mediante el criterio de slicing condicional como `(main,True,Pair` $\bot$ $\top$`)` y el slice dinámico de la figura 2.15 como `(main,{x=(Succ Zero), y=(Succ Zero), z=(Succ Zero)},Pair` $\bot$ $\top$`)`.

4.5. Trabajos Relacionados

En este trabajo hemos presentado la primera definición de slicing condicional en programas funcionales de primer orden. Además, introdujimos una aproximación para su cálculo. Ambos aportes están inspirados en los desarrollos realizados en el ámbito imperativo.

Como se mencionó anteriormente, aunque en el campo de los lenguajes imperativos el slicing de programas fue ampliamente investigado, en el ámbito de los lenguajes funcionales existen pocos desarrollos. Las aproximaciones más cercanas a nuestro desarrollo se mencionan a continuación.

La idea de slicing con una precondición fue introducida en [CPPGM91] —y posteriormente desarrollada en [CCLL94], [LFM96] y [CCL98]—. La técnica utiliza la ejecución simbólica del programa mediante la cual propaga las condiciones a través del mismo. El estado inicial del programa —que satisface la precondición— es propagado a todos los puntos en el programa. En cada rama condicional se efectúa un test para comprobar si es posible inferir el valor booleano de la condición desde el correspondiente estado del programa.

La misma aproximación fue adoptada en [HHF$^+$01], pero añadiendo un probador automático de teoremas para asistir en el chequeo automático de la

condición.

Equivalente al método anterior, en [CLYK01] se propone utilizar un cálculo para propagar el estado inicial del programa.

La técnica expuesta en [CCKJ02] evita el uso de probadores de teoremas mediante la aplicación de interpretación abstracta con el fin de razonar acerca de los efectos de la precondición. Esto permite realizar un proceso de slicing precondicional totalmente automático, pero con la consiguiente pérdida de precisión.

Los trabajos antes mencionados pueden emplearse como base para futuros desarrollos en el contexto declarativo.

La técnica propuesta en la presente investigación guarda similitudes con el algoritmo de evaluación parcial propuesto por Gallagher [Gal93] y con la evaluación parcial dirigida por narrowing de Alpuente et al [AFV96]. La relación existente entre el slicing y la evaluación parcial fue analizada previamente en distintos artículos. Reps y Turnidge [RT96] informalmente muestran que el slicing de programas realiza cierto tipo de especialización que es complementaria con las operaciones realizadas mediante evaluación parcial. Recientemente, Binkley et al [BDH$^+$06] comparan formalmente ambas técnicas empleando un marco común. En ese artículo demuestran que —para programas terminantes— el programa residual producido utilizando evaluación parcial es semánticamente equivalente al slicing condicional.

Sin embargo, en distintas actividades involucradas en la ingeniería de software tales como testing, comprensión de programas y debugging (entre otras aplicaciones) es muy importante conservar tanto la semántica como así también su sintaxis. A diferencia de la evaluación parcial, el método propuesto cumple con esta condición.

Con respecto a la aproximación presentada, podemos decir que es una extensión de la técnica de slicing introducida por Silva and Vidal [SV07]. En [SV07] se utiliza una extensión de la evaluación parcial para efectuar slicing, pero no se considera el uso de una condición lógica de primer orden como criterio de slicing. En ese sentido, nuestra propuesta es original.

Por otra parte, el *patrón de slicing* fue tomado del trabajo de Ochoa, Silva y Vidal en [OSV04] que definen y emplean dicho concepto para realizar slicing dinámico basado en redex-trails. La idea está inspirada en los *patrones de vivacidad* ideados por Liu y Stoller [LS03] con el objetivo de eliminar código muerto. Otra aproximación similar, es la función de proyección utilizada por Reps y Turnidge [RT96] para determinar qué información debe ser "descartada" y cuál "retenida" durante el slicing. A diferencia de éste último enfoque cuyo objetivo es la especialización de programas, nuestro *criterio de slicing* permite especificar cualquier función del programa como punto de interés, mientras que en [RT96] el criterio es más rígido y sólo es posible obtener el slice partiendo desde la función principal para identificar las partes que no son requeridas en el cómputo del resultado final.

5

Implementación

El slicer está desarrollado en y para el lenguaje Curry. La herramienta implementa el algoritmo descripto en el capítulo 4. Seguidamente presentamos brevemente las características del lenguaje Curry y, luego, detallamos las particularidades del prototipo construido.

5.1. El lenguaje multiparadigma *Curry*

Los lenguajes funcionales han evolucionado hacia lenguajes que combinan sus características con las de los lenguajes lógicos. Ejemplos de éstos lenguajes —denominados multiparadigmas— son *Curry* [HAB+06], *Toy* [CSe06], *Escher* [Llo95], *Mercury* [Mac05].

Particularmente, Curry es un lenguaje que reúne las propiedades más importantes de los lenguajes funcionles y lógicos:

- Programación funcional: expresiones anidadas, evaluación perezosa, funciones de orden superior.
- Programación lógica: variables lógicas, estructuras de datos parciales y métodos búsqueda incorporados.

Para el desarrollo del slicer utilizamos *PAKCS*[1] (Portland Aachen Kiel Curry System) [HAE+07], la implementación de Curry desarrollada por las universidades de Kiel, Aachen y Portland. PAKCS provee facilidades que pueden ser usadas para una multiplicidad de aplicaciones. Concretamente, nos interesan aquellas proporcionadas para la manipulación de programas Curry —i.e. metaprogramación—.

5.1.1. *Flat y Abstract Curry*

A medida que ha evolucionado el lenguaje Curry surgió la necesidad de contar con una interface común para conectar diferentes herramientas que manipulan

[1] http://www.informatik.uni-kiel.de/~pakcs/

programas Curry.

Con el objetivo de brindar soporte para la metaprogramación (i.e., manejar programas Curry con Curry), PAKCS incluye dos módulos: *FlatCurry* y *AbstractCurry* que definen tipos de datos para la representación interna de programas Curry.

Ambos módulos contienen operaciones que permiten analizar sintácticamente un programa fuente (i.e., en Curry, Abstract o Flat Curry) y representarlos en una estructura de datos. Además, existen funciones mediante las cuales es posible leer un archivo de un programa escrito en Curry y traducirlo al código intermedio. Una vez que tenemos el programa en código intermedio, podemos manipular las expresiones que representan al programa de acuerdo al proceso que deba aplicarse al mismo y, finalmente, almacenarlo en un archivo.

En nuestro caso, usaremos FlatCurry para representar los programas introducidos al slicer.

5.2. Estructura del Slicer

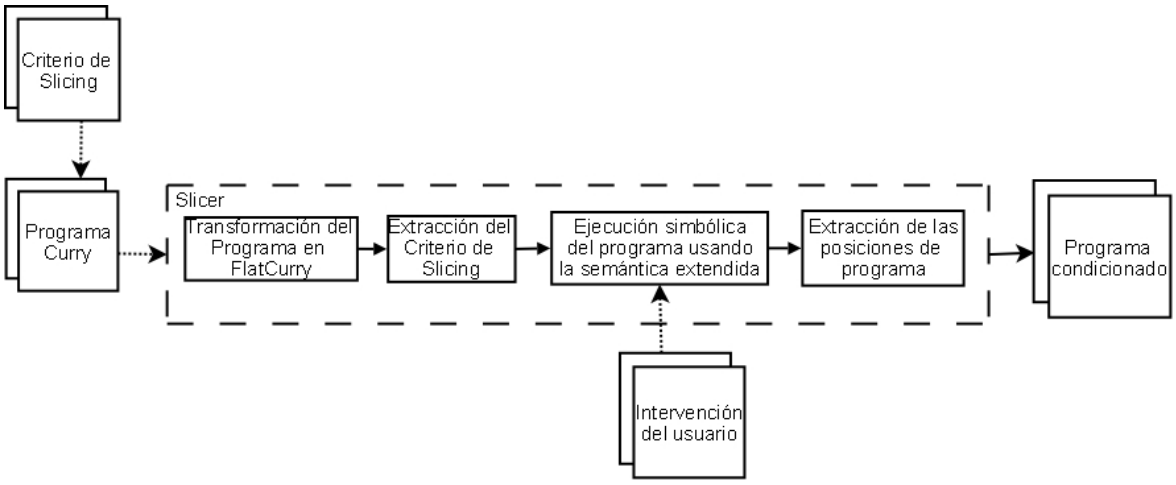

Figura 5.1: Estructura básica de la herramienta.

La estructura básica del slicer se muestra en la figura 5.1. El slicer consta de una interfaz por línea de comandos mediante la cual se lanza el slicer y se obtienen los resultados.

Una vez determinado el programa y su criterio, éstos son proporcionados como argumentos al slicer que evaluará simbólicamente el programa mediante la semántica aumentada definida en la sección 4.3.1 y empleando el criterio establecido. El slicer consulta interactivamente al usuario acerca de la veracidad

de la condición en distintos puntos del programa con el fin de recolectar aquellas posiciones de programa que serán ejecutadas cuando se cumpla la condición establecida en el criterio de slicing condicional.

Finalmente, utilizando las posiciones de programa recolectadas, el slicer genera el programa condicional —i.e., simplificado de acuerdo a la condición—.

El prototipo del slicer fue implementado mediante el reuso de código del evaluador parcial para Curry desarrollado por Albert, Vidal y Hanus [AHV00, AHV01, AHV02].

5.3. Sesión de Slicing

A continuación mostraremos una sesión típica de slicing condicional con el método propuesto.

El siguiente programa define el operador booleano `and`:

```
and True True = True
and False True = False
and True False = False
and False False = False
```

El criterio de slicing es una anotación que se realiza en el programa y que debe ser incluida como una función especial llamada `CSLICE`. La función `CSLICE` debe definirse en el programa como una función identidad, tal como se muestra a continuación:

```
goal x = CSLICE (and x False) (x == False)
CSLICE x y = x
```

La función `CSLICE` tiene dos argumentos:

- `(and x False)` es la llamada inicial, y
- `(x == False)` es la fórmula lógica de primer orden sobre las variables de entrada del programa.

El criterio de slicing condicional indica que el slicing debe comenzar desde la función inicial `(and x False)` y que sólo estamos interesados en aquellas partes del programa que satisfacen la condición `(x == False)`. Es decir, la única regla que satisface esta condición para el ejemplo dado es `and False False = False`.

El slicer se lanza desde PAKCS convocando al comando `:load` (o de manera abreviada `:l`):

```
prelude>:l condslicer
```

Luego, para computar el slice del programa debe ejecutarse una llamada similar a esta:

```
condslicer>main ‘‘Examples/example1’’
```

donde `Examples/example1.curry` es el archivo donde está almacenado el programa y la anotación que contiene el criterio de slicing condicional.

El slicer convierte el programa en formato FlatCurry y lo muestra. Aquí puede observarse cómo las reglas de la función `and` son convertidas a una única regla que define la función mediante el uso de *cases* anidados:

```
example1.and x1 x2 = (FCase x1 of
   (True  -> (FCase x2 of
     (True  -> (True))
     (False  -> (False))
  ))
   (False  -> (FCase x2 of
     (True  -> (False))
     (False  -> (False))
```

A continuación, interactivamente el slicer pregunta si la expresión que se está evaluando en ese momento satisface la condición indicada en el criterio de slicing:

```
In this function call:
(and x1 (False))

this arguments x1, (False),  satisfy (== x1 (False)),
? [y / n] y


In this program fragment:
(FCase x1 of
   (True  -> (FCase (False) of
     (True  -> (True))
     (False  -> (False))
  ))
)

if x1 is substituted by (-> x1 (True)),  satisfy (== x1 (False)),
? [y / n] n
```

```
In this program fragment:
(FCase x1 of
   (False  -> (FCase (False) of
     (True  -> (False))
     (False  -> (False))
  ))
)

if x1 is substituted by (-> x1 (False)),  satisfy (== x1 (False)),
? [y / n] y
```

Por último, el slicer muestra las posiciones de programa recolectadas a través del proceso anterior y que se corresponden con aquellas partes del programa que satisfacen la fórmula lógica:

```
[(‘‘example1.and’’,[]),(‘‘example1.and’’,[2,2]),(‘‘example1.and’’,[2,2,2,2])]
```

Tales posiciones de programa corresponden al siguiente programa en FlatCurry:

```
example1.and x1 x2 = (FCase x1 of
   (False  -> (FCase x2 of
     (False  -> (False))
```

6

Conclusiones y Trabajo Futuro

Este trabajo, hasta donde sabemos, es el primero en el que se define el *slice condicional* para lenguajes funcionales de primer orden. Junto con la definición, proporcionamos un algoritmo para la extracción de slices condicionales.

El objetivo de este método de slicing es obtener, a partir de un programa, un subprograma que codifica todas las posibles derivaciones que puede producirse mediante la ejecución del programa cuyas entradas están caracterizadas por una condición lógica de primer orden. De esta manera, ambos programas producen las mismas secuencias de derivación si se ejecutan con una entrada que cumpla la fórmula lógica de primer orden.

Actualmente, hemos desarrollado un prototipo que implementa la técnica propuesta. El slicer nos permitió chequear la viabilidad del enfoque desarrollado en este trabajo y nos posibilitará probar diferentes alternativas para mejorar la precisión de la técnica.

En ese sentido, con el fin de obtener slices más pequeños planeamos combinar la propuesta con diferentes enfoques. A largo plazo objetivo es generalizar la investigación con el fin de calcular otros tipos de slices (estáticos, dinámicos, quasiestáticos, etc.) como en [LFM96].

Existen varias direcciones hacia donde puede continuar la investigación presentada. Concretamente, en cuanto a la técnica de slicing definida, algunos de los trabajos pendientes son:

- Finalizar las pruebas formales de completitud del algoritmo empleado para calcular el slice de un programa.
- Extender la definición de la técnica para contemplar funciones de orden superior.
- Indagar la posibilidad de introducir un probador de teoremas para automatizar el proceso de slicing.
- Investigar la aproximación mediante el uso de interpretación abstracta en cuenta de usar la ejecución simbólica mediante evaluación parcial.

Otra extensión que resulta atractiva es adaptar la herramienta a Haskell. Como vimos en la sección 2.5.4, aparte del proyecto Programatica [Hal03] y de HaSlicer [RB05], ésta es un área poco explorada.

De manera independiente a la depuración, una aplicación inmediata es la especialización de programas basándose en slicing. Además de indagar sobre el uso de nuestra técnica para la especialización, puede resultar interesante compararla frente a la especialización basada en slicing dinámico y establecer las ventajas y desventajas de cada método.

La motivación de este trabajo es conseguir mejoras que permitan abordar con mayor eficacia y eficiencia la detección de errores en programas funcionales. En [Wad98], Wadler indica, como una de las causas que inciden en el uso de los lenguajes funcionales, la esacasez de herramientas para la depuración. Recientemente, en [HHJW07] se señala que los esfuerzos en el desarrollo de herramientas para depuración en Haskell aún no han prosperado. Por estos motivos, la línea de investigación no se limita únicamente a las posibilidades que ofrece el slicing, sino que puede ampliarse para abarcar otras técnicas de utilidad —como se expuso en el capítulo 2—.

Bibliografía

[ADS93] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software—Practice and Experience*, 23(6):589–616, 1993.

[AEH00] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[AFV96] M. Alpuente, M. Falaschi, and G. Vidal. Narrowing-driven Partial Evaluation of Functional Logic Programs. In H. Riis Nielson, editor, *Proc. of the 6th European Symp. on Programming, ESOP'96*, pages 45–61. Springer LNCS 1058, 1996.

[AH99] J. Ahn and T. Han. Static slicing of a first-order functional language based on operational semantics, 1999.

[AHV00] 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.

[AV02a] E. Albert and G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Gen. Comput.*, 20(1):3–26, 2002.

[AV02b] E. Albert and G. Vidal. Symbolic Profiling of Multi-Paradigm Declarative Languages. In *Proc. of Int'l Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'01)*, pages 148–167. Springer LNCS 2372, 2002.

[BC85] J. F. Bergeretti and B. A. Carré. Information-Flow and Data-Flow Analysis of While-Programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985.

[BCHH04] B. Brassel, O. Chitil, M. Hanus, and F. Huch. Observing functional logic computations. In B. Jayaraman, editor, *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, LNCS 3057, pages 193–208. Springer, June 2004.

[BDH+06] D. Binkley, S. Danicic, M. Harman, J. Howroyd, and L. Ouarbya. A formal relationship between program slicing and partial evaluation. *Form. Asp. Comput.*, 18(2):103–119, 2006.

[BE93] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[BG96] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[BH93] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. of 20th Annual ACM Symp. on Principles of Programming Languages*, pages 384–396, New York, NY, USA, 1993. ACM Press.

[BHH+04] B. Brassel, M. Hanus, F. Huch, J. Silva, and G. Vidal. Runtime Profiling of Functional Logic Programs. In *Proc. of the 14th Int'l Symp. on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, pages 178–189, 2004.

[Bin98] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11-12):583–594, 1998.

[Bis97] S. K. Biswas. *Dynamic Slicing in Higher-Order Programming Languages*. PhD thesis, Department of CIS, University of Pennsylvania, 1997.

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

[Bre01] T. Brehm. A toolkit for multi-view tracing of haskell programs. Master's thesis, RWTH Aachen, 2001.

[Cab05] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*, pages 8–13, New York, NY, USA, 2005. ACM Press.

[CCKJ02] I. Chung, B.-M. Chang, B. Kim, and Jang-Wu Jo. Abstract program slicings. In *Twentieth IASTED International Conference on Applied Informatics*. ACTA Press, 2002.

[CCL98] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998. Special issue on program slicing.

[CCLL94] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, and Giuseppe A. Di Lucca. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance, ICSM 1994, Victoria, BC, Canada, 1994*, pages 424–433, 1994.

[Chi01] O. Chitil. A Semantics for Tracing. In *13th Int'l Workshop on Implementation of Functional Languages (IFL 2001)*, pages 249–254. Ericsson Computer Science Laboratory, 2001.

[CHK07] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of java programs. *Electronic Notes in Theoretical Computer Science*, 2007. To appear.

[CL02] R. Caballero and W. Lux. Declarative debugging for encapsulated search. *Electronic Notes in Theoretical Computer Science*, 76, 2002.

[CLYK01] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon. Program slicing based on specification. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 605–609, New York, NY, USA, 2001. ACM Press.

[CPPGM91] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli. Software specialization via symbolic execution. *IEEE Trans. Softw. Eng.*, 17(9):884–899, 1991.

[CRA04] R. Caballero and M. Rodríguez-Artalejo. DDT: A Declarative Debugging Tool for Functional-Logic Languages. In *Proc. of 5th Int'l Symp. on Functional and Logic Programming (FLOPS'04)*, volume 2998 of *LNCS*, pages 70–84. Springer, 2004.

[CS08] D. Cheda and J. Silva. Static Slicing of Rewrite Systems. In *Proc. of the 17th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, 2008.

[CSe06] R. Caballero and J. Sanchez (eds.). Toy: A multiparadigm declarative language, version 2.2.3. Technical report, UCM, Madrid, July 2006.

[CSV07] D. Cheda, J. Silva, and G. Vidal. Static Slicing of Rewrite Systems. In *Proc. of the 15th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)*, volume 177, pages 123–136. Electronic Notes in Theoretical Computer Science, 2007.

[DC05] T. Davie and O. Chitil. Hat-delta — one right does make a wrong. In Colin Runciman, editor, *Hat Day 2005: work in progress on the Hat tracing system for Haskell*, pages 6–11. Tech. Report YCS-2005-395, Dept. of Computer Science, University of York, UK, October 2005.

[DP01] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.

[EJ03] R. Ennals and S. Peyton Jones. Hsdebug: debugging lazy programs by not being lazy. In *Haskell '03: Proc. of the 2003 ACM SIGPLAN Workshop on Haskell*, pages 84–87, New York, NY, USA, 2003. ACM Press.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[FSKG92] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimóthy. Generalized Algorithmic Debugging and Testing. *LOPLAS*, 1(4):303–322, 1992.

[FT98] J. Field and F. Tip. Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing. *Information and Software Technology*, 40(11-12):609–634, 1998.

[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.

[GBF99] T. Gyimóthy, Á. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 303–321, London, UK, 1999. Springer-Verlag.

[Gil00] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. In *Proc. of the 4th Haskell Workshop*. Technical report of the University of Nottingham, 2000.

[GL91] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[HAB+06] M. Hanus, S. Antoy, B. Braßel, H. Kuchen, F. J. López-Fraguas, W. Lux, J. J. Moreno Navarro, and F. Steiner. Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~curry/`, 2006.

[HAE+07] M. Hanus, S. Antoy, M. Engelke, B. Braßel, K. Höppner, , J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS 1.8: The Portland Aachen Kiel Curry System—User Manual. Technical report, U. Kiel, Germany, 2007.

[Hal03] T. Hallgren. Haskell Tools from the Programatica Project. In *Proc. of the ACM Workshop on Haskell (Haskell'03)*, pages 103–106. ACM Press, 2003.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[Han97] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM, 1997.

[HDS95] M. Harman, S. Danicic, and Y. Sivagurunathan. Program comprehension assisted by slicing and transformation, 1995.

[HH93] V. Hirunkitti and C. J. Hogger. A Generalised Query Minimisation for Program Debugging. In *AADEBUG, Springer Lecture Notes in Computer Science, vol 749*, pages 153–170, 1993.

[HHF+01] Mark Harman, Rob Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 138, Washington, DC, USA, 2001. IEEE Computer Society.

[HHJW07] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[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.

[HP99] M. Hanus and C. Prehofer. Higher-Order Narrowing with Definitional Trees. *Journal of Functional Programming*, 9(1):33–75, 1999.

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[Hug89] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[IM07] J. Iborra and S. Marlow. Examine your laziness. Technical report, Universidad Politécnica de Valencia, 2007. To be published.

[KL88] B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[Klo92] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[KM91] J.W. Klop and A. Middeldorp. Sequentiality in Orthogonal Term Rewriting Systems. *Journal of Symbolic Computation*, pages 161–195, 1991.

[KSF90] M. Kamkar, N. Shahmehri, and P. Fritzson. Bug localization by algorithmic debugging and program slicing. In *PLILP*, pages 60–74, 1990.

[LA93] P. E. Livadas and S. D. Alden. A toolset for program understanding. In Bruno Fadini and Vaclav Rajlich, editors, *Proceedings of the IEEE Second Workshop on Program Comprehension*, pages 110–118, 1993.

[Las90] J. Laski. Data flow testing in stad. *J. Syst. Softw.*, 12(1):3–14, 1990.

[Leu98] M. Leuschel. Homeomorphic embedding for online termination, 1998.

[LFM96] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In A. Cimitile and H. A. Müller, editors, *Proceedings: Fourth Workshop on Program Comprehension*, pages 9–18. IEEE Computer Society Press, 1996.

[Llo87] J. W. Lloyd. Declarative error diagnosis. *New Gen. Comput.*, 5(2):133–154, 1987.

[Llo95] J. W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.

[LS03] Y. A. Liu and S. D. Stoller. Eliminating Dead Code on Recursive Data. *Science of Computer Programming*, 47:221–242, 2003.

[LV96] F. Lanubile and G. Visaggio. Extracting reusable functions by program slicing. Technical Report CS-TR-3594, 1996.

[LW87] J. R. Lyle and M. D. Weiser. Automatic Program Bug Location by Program Slicing. In *In Proceeding of the Second International Conference on Computers and Applications*, pages 877–882, Peking, China, June, June 1987.

[Mac05] I. MacLarty. *Practical Declarative Debugging of Mercury Programs.* PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.

[MK06] O. Murk and L. Kolmodin. Rectus: Locally eager haskell debugger. To be published, 2006.

[Nil98] H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, may 1998.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.

[OSV04] C. Ochoa, J. Silva, and G. Vidal. Dynamic Slicing Based on Redex Trails. In *Proc. of the ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation (PEPM'04)*, pages 123–134. ACM Press, 2004.

[Pop98] B. Pope. Buddha: A declarative debugger for haskell, 1998.

[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.

[RB05] N. Rodrigues and L. S. Barbosa. Component Identification Through Program Slicing. In Electronic Notes in Theoretical Computer Science, editor, *In Proc. Formal Aspects of Component Software (FACS05)*. Elsevier, 2005.

[RT96] T. Reps and T. Turnidge. Program Specialization via Program Slicing. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, pages 409–429. Springer LNCS 1110, 1996.

[SH99] Z. Somogyi and F. Henderson. The implementation technology of the mercury debugger. *Electronic Notes in Theoretical Computer Science*, 30(4), 1999.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[Sil06] J. Silva. Strategies for Algorithmic Debugging. Informes Investigación DSIC-II/09/06, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, April 2006.

[SJ97] M. Sansom and S. L. Peyton Jones. Formally based profiling for higher-order functional languages. *ACM Trans. Program. Lang. Syst.*, 19(2):334–385, 1997.

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

[Spa96] J. Sparud. A transformational approach to debugging lazy functional programs. Master's thesis, Chalmers University, 1996.

[SR97] J. Sparud and C. Runciman. Tracing Lazy Functional Computations Using Redex Trails. In *Proc. of the 9th Int'l Symp. on Programming Languages, Implementations, Logics and Programs (PLILP'97)*, pages 291–308. Springer LNCS 1292, 1997.

[SV07] J. Silva and G. Vidal. Forward slicing of functional logic programs by partial evaluation. *Theory and Practice of Logic Programming*, 7:215–247, 2007.

[Vid03] G. Vidal. Forward Slicing of Multi-Paradigm Declarative Programs Based on Partial Evaluation. In *Logic-based Program Synthesis and Transformation (revised and selected papers from the 12th Int'l Workshop LOPSTR 2002)*, pages 219–237. Springer LNCS 2664, 2003.

[Wad98] P. Wadler. Why no one uses functional languages. *Sigplan Notices*, 33(8):23–27, 1998.

[WCBR01] M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. Universiteit Utrecht UU-CS-2001-23, 2001.

[Wei79] M. D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

[Wei84] M. D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[Zel78] M. V. Zelkowitz. Perspectives in software engineering. *ACM Comput. Surv.*, 10(2):197–216, 1978.

Apéndices

A

Trabajos desarrollados en el marco de esta Tesis

El presente trabajo se elaboró basándonos en los siguientes artículos:

- **D. Cheda** y S. Cavadini, *Conditioned Slicing for Firsts Order Functional Logic Languages*. En *Proc. of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, 2008.
- **D. Cheda** y J. Silva, *An Evaluation of Algorithmic Debugger Implementations*. En *Proc. of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008)*, 2008. A publicarse en Electronic Notes in Theoretical Computer Science (ENTCS).
- **D. Cheda** y J. Silva, *State of the Practice in Algorithmic Debugging*. En *VII Jornadas sobre Programación y Lenguajes (PROLE 2007)*, 2007.
- **D. Cheda**, J. Silva, y G. Vidal, *Static Slicing of Rewrite Systems*. En *Proc. of the 15th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP 2006)*. Electronic Notes in Theoretical Computer Science, 2006.
- **D. Cheda**, J. Silva, y G. Vidal, *Term Dependence Graph*. En *VI Jornadas sobre Programación y Lenguajes (PROLE 2006)*, 2006.

Otros trabajos publicados por el autor de este trabajo, pero no referenciados en la memoria:

- S. Cavadini y **D. Cheda**, *Run-time Information Flow Monitoring based on Dynamic Dependence Graphs*, 3th International Conference on Availability, Reliability and Security (ARES 2008). En *IEEE Proceedings*, pp. 586-591, 2008.
- S. Cavadini y **D. Cheda**, *Program Slicing Based on Sentence Executability*. En *XIII Congreso Argentino de Ciencias de la Computación (CACIC 2007)*, 2007.

- S. Cavadini y **D. Cheda**, *The Necessary Condition for Execution and its Use in Program Slicing*. En *VII Jornadas sobre Programación y Lenguajes (PROLE 2007)*, 2007.
- **D. Cheda** y S. Cavadini, *Refactoreo de Diagramas de Clases UML empleando Slicing de Modelos*. En *XII Congreso Argentino de Ciencias de la Computación (CACIC 2006)*, 2006.