Fragmentación de Programas Erlang Secuenciales

César Tomás Franco

Departamento de Sistemas Informáticos y Computación

Universidad Politécnica de Valencia

Memoria presentada para optar al título de:
Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Director:
Josep Silva Galiana

Valencia, Septiembre 2014

Índice general

Capítulo 1

Introducción

Desde que Mark Weiser acuñó el término fragmentación de programas (del inglés *Program Slicing*) se han desarrollado gran cantidad de trabajos y aplicaciones en torno a esta técnica de análisis de programas. La finalidad principal del slicing es la de extraer una parte del programa (fragmento o *slice*) que influye o es influenciada por un determinado punto de interés (criterio de fragmentación o *criterio de slicing*) [23, 21, 20]. Inicialmente fue propuesta como una técnica de depuración para mejorar la comprensión de la porción de código que contiene el error.

La técnica de slicing de programas puede ser dinámica (si tiene en cuenta una única ejecución concreta del programa) o estática (se tienen en cuenta todas las posibles ejecuciones). De esta forma, la versión dinámica esta basada en una estructura de datos que representa una ejecución particular (o traza) [14, 1], la versión estática ha estado tradicionalmente basada en una estructura de datos llamada *grafo de dependencia de programa* (PDG, del inglés *program dependence graph*) [9] que representa todas las instrucciones de un programa mediante nodos y sus dependencias de control y datos mediante arcos. Cuando el PDG se ha construido, el slicing de programas se reduce a un problema de alcanzabilidad dentro del grafo de tal forma que los slices pueden ser obtenidos en un tiempo lineal.

Desafortunadamente, el PDG es impreciso en el caso de los programas interprocedurales. Es por eso que se definió una versión mejorada denominada *grafo de dependencia de sistema* (SDG, del inglés *system dependence graph*) [12]. El SDG ofrece como ventaja que almacena el contexto de cada una de las llamadas a funciones de forma que puede distinguir entre distintas llamadas. Esto nos permite definir un algoritmo mucho mas preciso en los programas intraprocedurales.

En este trabajo se adapta el SDG al lenguaje funcional Erlang. Esta adaptación es muy interesante porque se trata de la primera adaptación del

SDG a un lenguaje funcional. Los lenguajes funcionales imponen dificultades adicionales al SDG y a la definición del algoritmo para producir slices precisos. Concretamente Erlang impone restricciones tales como asignación única de variables, ajuste de patrones o funciones de alto nivel. Es por eso que se necesita una redefinición no trivial del SDG que contemple todas estas características.

1.1. Erlang Secuencial

Erlang es un lenguaje pequeño simple y eficiente que fue concebido para programar aplicaciones industriales concurrentes robustas a gran escala. El subconjunto de programación secuencial de Erlang es un lenguaje funcional, con evaluación estricta, asignación única y tipado dinámico. Fue diseñado por la compañía Ericsson para realizar aplicaciones distribuidas, tolerantes a fallos, *soft-real-time* y de funcionamiento ininterrumpido (ofrece la funcionalidad de cambio en caliente de código sin necesidad de parar el sistema). La primera versión fue desarrollada por Joe Armstrong en 1986, pero no fue cedida como código abierto hasta 1998.[2]

La creación y gestión de procesos es trivial en Erlang, mientras que, en muchos lenguajes, los hilos de ejecución se consideran un apartado complejo y propenso a errores. En Erlang toda concurrencia es explícita, los distintos procesos se comunican usando el paso de mensajes asíncrono mediante variables compartidas, sin tener la necesidad de usar bloqueos. Los mecanismos de concurrencia en Erlang requieren muy poca memoria permitiendo así soportar aplicaciones con un número muy elevado de procesos concurrentes. Además, Erlang tiene una gestión de la memoria muy eficiente gracias a su asignación dinámica y su recolector de basura en tiempo real.

Cuando Erlang fue diseñado, se concibió como un lenguaje simple de comprender y aprender. Es por esto que por razones de eficiencia, se evitaron incluir muchas características presentes en los lenguajes lógicos o funcionales modernos, tales como currificación, evaluación perezosa, variables lógicas, etc. No obstante su ausencia no representa un inconveniente a la hora de programar aplicaciones típicas de control industrial. El uso de la sintaxis de ajuste de patrones (*pattern matching*), recursividad y modularización, la presencia únicamente de dos estructuras de control, estructuras de datos mediante listas y tuplas, junto con la propiedad de asignación única de variables de Erlang dan como resultado programas claros, cortos y fiables.

Como la mayoría de lenguajes funcionales Erlang requiere un análisis del problema y una forma de diseñar la solución diferente a como se haría en un lenguaje de programación imperativo. Dicha forma de solucionar el problema

requerido se basa en una sintaxis más matemática que programática por lo que tiende más a la resolución de problemas que a la ordenación y ejecución de órdenes. [13]

Erlang es un lenguaje basado en Prolog y tiene una sintaxis muy particular. Algunos ejemplos de estas particularidades son que Erlang no soporta bucles tales como `while`, `repeat` o `for` sino que son implementados mediante recursión, las variables solo pueden ser asignadas una vez o incluso que el ajuste de patrones se usa para seleccionar la cláusula deseada en una función. Adicionalmente, se pueden utilizar funciones de alto nivel y otras construcciones sintácticas no presentes en los lenguajes imperativos. En el capítulo 3 veremos en más detalle el subconjunto de instrucciones secuenciales que se ha elegido como el lenguaje vehicular para la adaptación de la técnica de fragmentación de programas.

En los lenguajes imperativos la sintaxis se basa en una serie de instrucciones que el programador indica a la máquina mediante el código. Tanto en Erlang como en otros lenguajes funcionales, la sintaxis se diseña en formato de funciones matemáticas o proposiciones lógicas. Así, cada elemento dentro de la función tiene la finalidad de obtener un valor. El resultado estará formado por el conjunto de todos esos valores, con o sin procesamiento.

Es una realidad que el lenguaje Erlang, a pesar de haber sido diseñado hace mas de treinta años, se va asentando en el entorno tanto empresarial como en las comunidades de software libre, siendo los desarrollos en este lenguaje cada vez mas visibles, sobretodo en el entorno para el que fue concebido: la concurrencia y el software distribuido y robusto a gran escala. Claros ejemplos de este creciente interés en el marco empresarial son los desarrollos del chat de las redes sociales *Facebook* o *Tuenti* que soportan intercambio de mensajes de mas de 70 millones de usuarios. O incluso el programa de intercambio de mensajes entre smartphones *WhatsApp* emplean a nivel de servidor sistemas Erlang. En el entorno de Software libre se han desarrollado proyectos de gran envergadura en ámbitos tan distintos como por ejemplo bases de datos distribuidas, servidores web, sistemas de gestión de contenidos, chats o colas de mensajes.

1.2. Fragmentación de programas

La técnica de fragmentación de programas o *program slicing*[1] fue definida por Mark Weiser en su tesis doctoral [25] junto con algoritmo para poder calcular los slices.

[1] En adelante utilizaremos el término fragmentación o slicing indistintamente, siendo el último el más comúnmente usado en la comunidad científica.

La aplicación natural del slicing fue inicialmente la de auxiliar a los programadores en las tareas de depuración de programas (*debugging*). Sin embargo, una vez reconocida su utilidad, se percibió que también ofrecer grandes beneficios su aplicación en otras áreas como testing [3], integración [11], comprensión [17], paralelismo, obtención de métricas [24], ingeniería reversa [4], etc. En cada una de estas áreas se requiere obtener slices con ciertas características o propiedades especiales, que ha provocado la aparición de distintas definiciones de slice que difieren en mayor o menor grado con la original junto con sus correspondientes métodos de cálculo. A continuación vemos algunos de las principales clasificaciones de los tipos de slicing de programas dependiendo de su naturaleza y características particulares:

Slicing Dinámico y Estático: Mediante el slicing estático los slices son calculados sin necesidad de ejecutar el programa, ya que se tienen en cuenta todas las posibles ejecuciones del programa para todas las posibles entradas del mismo.

El concepto de slicing dinámico fue introducido por [14], en el que el slice contiene únicamente aquellas instrucciones que realmente afectan al valor de una variable en un punto del programa durante una ejecución concreta del mismo. Para calcularlo se ha de tener en cuenta los datos de entrada del programa para la ejecución en particular.

***Forward y Backward* Slicing:** La definición de slicing convencional siempre ha sido"hacia atrás"(o *Backward Slicing*), en la que se obtienen todas aquellas instrucciones que influyen en un determinado punto del programa [23]. Es decir, se calcula realizando un análisis "hacia atrás-especto al punto de interés seleccionado.

Sin embargo, en [5] se acuña el concepto de slicing "hacia adelante"(o *Forward Slicing*) mediante el cual se obtiene el conjunto de sentencias que resultarían afectadas si se realizara un cambio en el valor de la variable de interes o criterio de slicing.

Slice ejecutable y de clausura: El slice ejecutable es un subconjunto de las instrucciones del programa original sintácticamente válido, es decir, se podría compilar y ejecutar sin problemas. Los slices de clausura no requieren ser sintácticamente válidos, con lo que su número de instrucciones puede ser menor y su obtención menos costosa al no exigir esta correctitud sintáctica.

Para realizar el cálculo del slice se han definido multitud de métodos y algoritmos que intentan aproximarse en mayor o menor grado a la obtención del slice mínimo (slice ideal con el menor número de sentencias posible),

aunque lamentablemente ninguno de ellos puede alcanzar a obtenerlo ya que está demostrado que es un problema indecidible [10]. En el presente documento nos vamos a centrar en el análisis de los métodos de cálculo para la técnica de slicing de programas estática. De esta forma, para poder calcular el slice del programa deseado es necesario obtener las dependencias de información entre las distintas instrucciones que componen el programa. El concepto de dependencia es esencial en la noción del cálculo del slice. La forma más natural para poder representar y posteriormente analizar dichas dependencias del programa tradicionalmente ha sido mediante grafos. Así, cada instrucción se representa mediante un nodo, y las distintas dependencias entre las instrucciones mediante arcos. A continuación detallamos algunos de los principales grafos utilizados para representar las dependencias entre instrucciones de un programa:

- **El Grafo de Flujo de Control (*CFG* o *Control Flow Graph*)** es una estructura de datos que representa todas las dependencias de control de las instrucciones de un programa, de forma que un arco dirigido desde el nodo i hasta el nodo j indica un flujo de control desde el primero hasta el último.

- Sin embargo, para poder representar de forma mas precisa las dependencias de información entre las instrucciones de un programa se necesita tener en cuenta las dependencias de datos, característica que no se refleja en el CFG. Para solucionar esta carencia, el **Grafo de Dependencia de Programa (*PDG* o *Program Dependence Graph*)** [18] extiende al CFG incluyendo dichas dependencias de datos. Como puede observarse en la Figura 1.1, en este PDG se incluyen tanto las dependencias de control (arcos continuos) como las dependencias de datos (arcos discontinuos).

El cálculo del slice de un programa se reduce a un problema de alcanzabilidad dentro del PDG, recorriendo en sentido inverso (*backward*) los arcos del PDG iterativamente desde el nodo que representa el criterio de slicing. Así, el conjunto de nodos alcanzados representa el slice, consiguiendo obtenerlo en tiempo lineal con el número de nodos del grafo [18]. En la Figura 1.2 podemos observar remarcado en el PDG el conjunto de nodos que forman parte del slice en el caso de haber seleccionado como criterio de slicing el nodo `print sum`

- El PDG obtiene slices precisos en programas intraprocedurales, sin embargo, en el caso de programas interprocedurales el PDG incluye nodos innecesarios en el slice calculado. Esta pérdida de precisión se produce al no

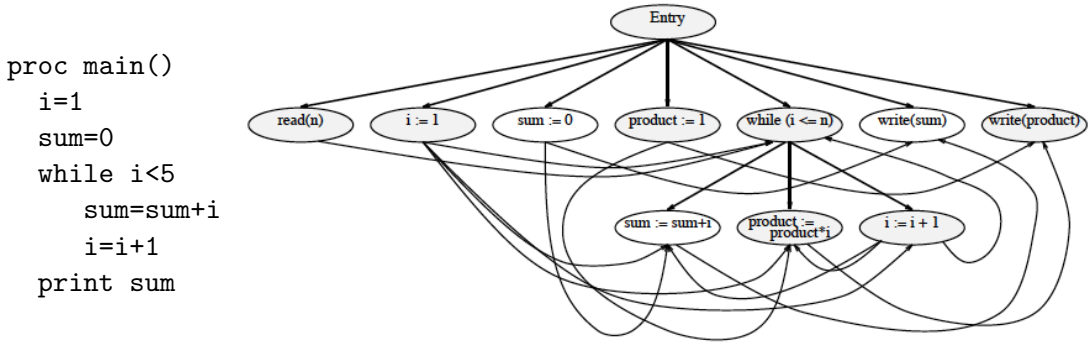


Figura 1.1: *Representación de las dependencias de control y datos de un programa mediante el PDG*

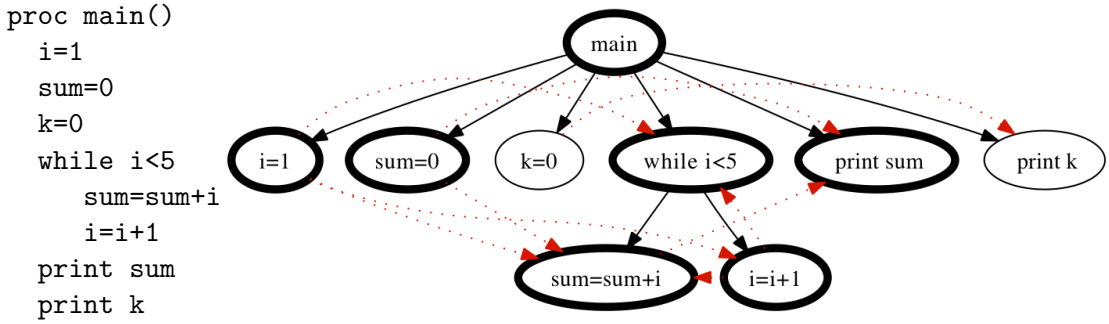


Figura 1.2: *Slice obtenido (negrita) a partir del PDG si se ha seleccionado como criterio de slicing el nodo* `print sum`

tener en cuenta las estructuras llamada-resultado de las rutas de ejecución interprocedurales. Para solucionar este problema Horwitz, Reps y Binkley [12] propusieron un algoritmo para computar slices interprocedurales precisos basándose en el denominado **Grafo de Dependencia de Sistema (*SDG* o *System Dependence Graph*)**. Para conseguir esto, el SDG representa un PDG por cada procedimiento del programa de forma que se conectan entre ellos mediante nuevos tipos de nodos y nuevos tipos de arcos interprocedurales de datos y control. Los nuevos tipos de nodos modelan los valores de entrada y salida, tanto en la llamada como en el procedimiento invocado:

I) *Nodos reales de entrada y salida* dependientes de control del nodo de la llamada (*call*). Se añade uno por cada parámetro implicado en la entrada / salida de la llamada.

II) *Nodos formales de entrada y salida* dependientes de control del nodo de entrada del procedimiento invocado (*entry*). Se añade uno por cada

argumento de entrada / salida del procedimiento invocado.

Los nuevos tipos de arcos representan el flujo de información y de control interprocedural. Por cada llamada a procedimiento tenemos:

i) Un *arco de control interprocedural* que une el nodo de la llamada (*call*) con el nodo de entrada del procedimiento invocado (*entry*)

ii) *Arcos de parámetros de entrada* entre los nodos reales y formales de entrada correspondientes.

iii) *Arcos de parámetros de salida* entre los nodos reales y formales de salida correspondientes.

iv) *Arcos de resumen* entre algunos nodos reales de entrada y de salida. Representan dependencias de datos interprocedurales transitivas. La presencia de este tipo de arco indica el hecho de que el valor del parámetro de salida está influenciado por el valor que toma el correspondiente parámetro de entrada.

Estos nuevos tipos de arcos y nodos modelan una interfaz capaz de capturar el contexto de cada llamada. El Ejemplo 1.3 muestra el SDG correspondiente a un programa interprocedural con los nuevos tipos de nodos y arcos.

Junto con la definición del SDG, en [12] se definió un algoritmo capaz de producir slices interprocedurales precisos consistente en dos fases. La primera fase atraviesa todos los arcos del grafo en sentido inverso (*backwards*) excepto los de salida añadiendo al slice todos los nodos alcanzados. Es decir, obtenemos todos los nodos que se pueden alcanzar sin atravesar las llamadas a procedimientos. En la segunda fase se atraviesan en sentido inverso todos los tipos de arcos excepto los de entrada, obteniendo los nodos restantes en los procedimientos invocados. Gracias a los arcos de resumen se descartan aquellas llamadas que pueden ser eludidas, provocando un aumento considerable en la precisión del slice.

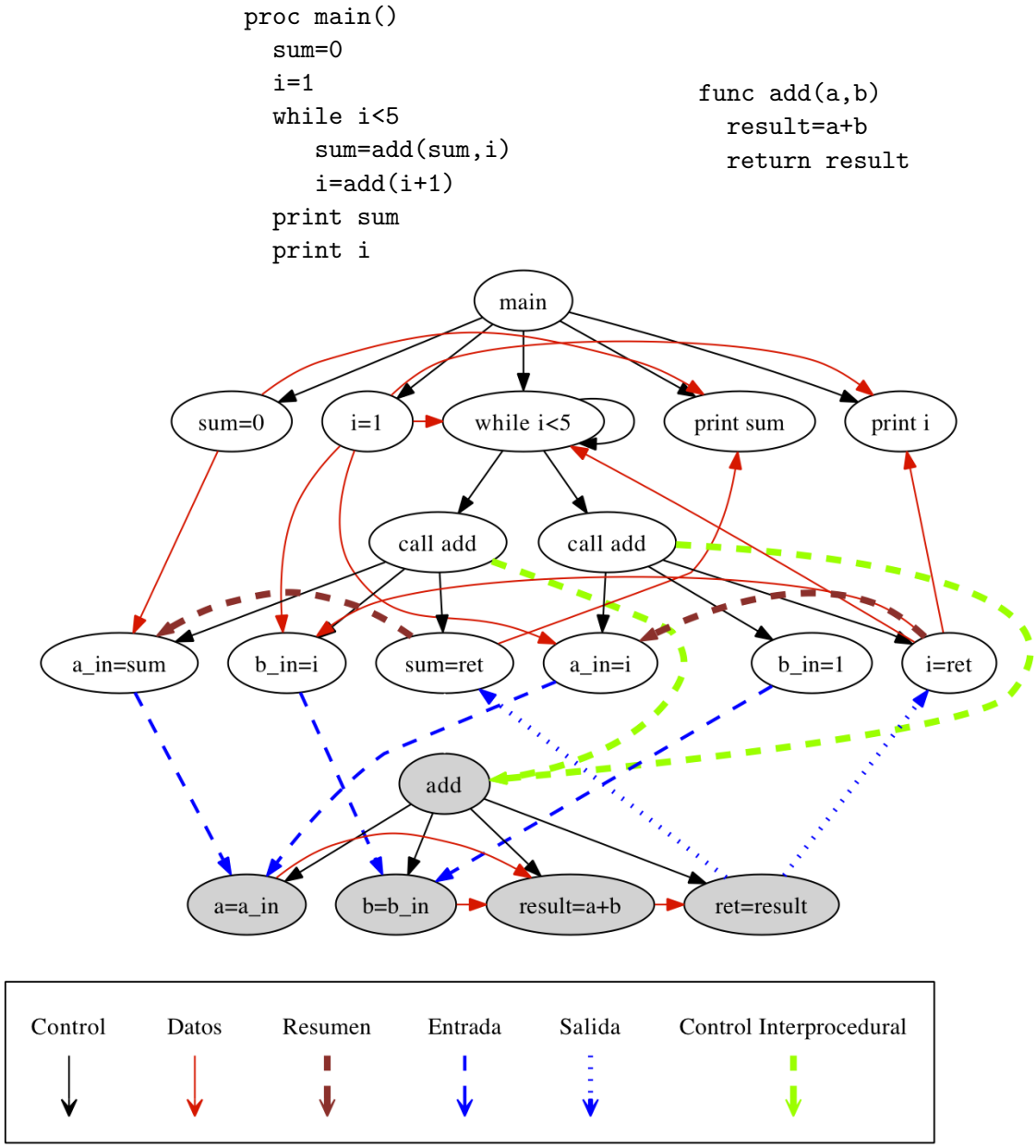


Figura 1.3: *SDG obtenido a partir del código*

Capítulo 2

Estado del Arte

La técnica de slicing de programas ha estado tradicionalmente ligada al paradigma imperativo, siendo muy pocos los trabajos existentes para lenguajes funcionales. A continuación estudiaremos el estado del arte relacionado con el ámbito del presente documento, analizando las principales características de las distintas aproximaciones.

Component identification through program slicing [19]**:** Este trabajo presenta una adaptación del PDG para lenguajes funcionales, concretamente para el lenguaje Haskell, el *Functional Dependence Graph*. Se define también un algoritmo basándose en dicha estructura de datos para poder computar los slices de programas escritos en este lenguaje y compuesto por cinco fases.

Sin embargo, el FDG es muy útil a niveles de abstracción mas elevados, por ejemplo es posible aplicar la técnica de slicing sobre módulos o funciones. Es por este motivo por el que esta técnica resulta insuficiente para Erlang, ya que no se puede aplicar sobre expresiones.

Static Slicing of Rewrite Systems [8]**:** Basándose en sistemas de reescritura de términos, este trabajo presenta una aproximación a la técnica de slicing para programas funcionales de primer orden. Para aplicar la técnica de slicing en sistemas de reescritura de términos, se define en este trabajo una noción de dependencia adaptada al contexto funcional, así como la definición de la estructura de datos *Term Dependence Graph*. También se define un algoritmo para poder computar los slices basándose en dicha estructura de datos.

Este tipo de grafo únicamente considera sistemas de reescritura de términos con llamadas a funciones y constructores de datos, sin tener en cuenta es-

tructuras mas complejas tales como expresiones `if` o `case`, llegando incluso a no ser capaces de trabajar con programas de alto nivel.

Tool Support For Refactoring Haskell Programs [7]**:** Esta tesis doctoral utiliza como lenguaje vehicular el lenguaje funciona Haskell, y provee un conjunto de refactorizaciones aplicables a programas de dicho lenguaje.

Se presenta una serie de refactorizaciones relacionadas con la técnica de slicing, proporcionando una nueva definición de dicha técnica para programas Haskell. Sin embargo, no se define una nueva estructura de datos, sino que se usa el *Abstract Syntax Tree* de Haskell ampliado con anotaciones extra sobre las dependencias de datos.

Grafos de flujo en Erlang Secuencial [26]**:** En este artículo, al igual que el presente trabajo, los autores proponen un grafo de flujo para la parte secuencial de Erlang y describen una herramienta para poder generar dichos grafos a partir del código.

Este grafo es utilizado para ayudar en la tarea de testeo ya que nos permite determinar el conjunto de las diferentes rutas de flujo que los casos de test deben cubrir.

Sin embargo, este grafo no está basado en el SDG y no contiene la información necesaria para poder aplicar de forma precisa la técnica de slicing. Por ejemplo, no contiene arcos de resumen y no descompone las expresiones de forma que en muchos casos no es posible seleccionar variables simples como criterio de slicing. A pesar de esto, este grafo consigue solucionar el problema de la dependencia de flujo.

Dependence Graph [22]**:** Por último se analiza la aproximación de Melinda Tóth e István Bozó [22], en la que definen un grafo de dependencia que se construye composicionalmente a partir de otros grafos para poder aplicar la técnica de slicing. Esta aproximación se utiliza básicamente para poder aplicar la técnica de Forward Slicing sobre el código y detectar el alcance de los valores de las variables en la herramienta de refactorización *refactorErl* [6].

En la Figura 2.1, podemos ver los grafos en los que se basa esta aproximación para poder construir composicionalmente el *Dependence Graph*.

A continuación analizamos a grandes rasgos los distintos grafos en los que se basa este trabajo:

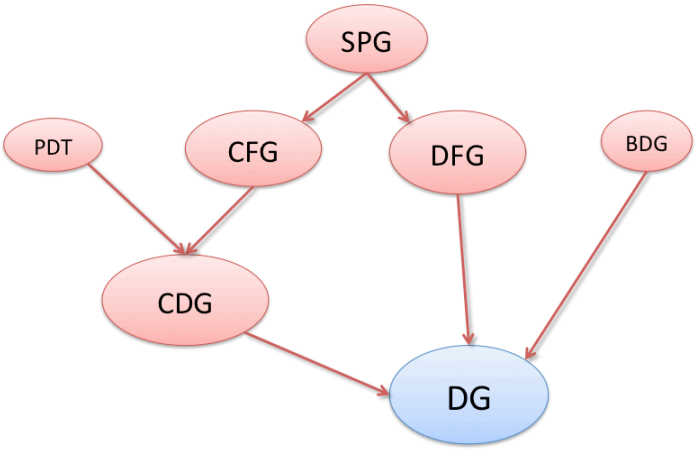


Figura 2.1: *Construcción composicional del DG a partir de otros grafos.*

- **SPG *Semantic Program Graph*:** Grafo constituido por tres capas que almacenan información léxica (tokens, espacios, comentarios...), sintáctica (AST) y semántica de los programas (enlace de variables...).
- **DFG *Data Flow Graph*:** Extensión del SPG con arcos de flujo de datos centrándose en la alcanzabilidad.
- **BDG *Behaviour Dependence Graph*:** Grafo que representa el flujo de comportamiento del programa. Este tipo de dependencia analiza si el comportamiento de un nodo influye en el comportamiento de otro.
- **CFG *Control Flow Graph*:** Basado en la semántica del lenguaje, a partir del SPG es un grafo intrafuncional y contiene todos los caminos de ejecución de una función.
- **PDT *Post Dominator Tree*:** Un nodo del CFG post-domina a otro si todos los caminos de ejecución que parten del segundo contienen al primero, de forma que en este grafo se reflejan este tipo de dependencias.
- **CDG *Control Dependence Graph*:** Se construye a partir de los CFG de las funciones implicadas y de los PDT.
- **DG *Dependence Graph*:** El CDG compuesto (interfuncional) se extiende con dependencias de datos adicionales (sacadas del DFG) y con las dependencias de comportamiento del BDG dando lugar al DG.

En este trabajo los slices obtenidos no son ejecutables y la sintaxis incluida engloba tanto la parte secuencial como la concurrente de Erlang, así como otras características del lenguaje como el manejo de errores. Sin embargo, esta aproximación no produce slices precisos en el caso interprocedural y el proceso de construcción del grafo tiene un coste muy elevado.

Hay que destacar que todos los trabajos recién analizados han sido diseñados para aplicar la técnica de slicing en programas intraprocedurales, pero al mismo tiempo presentan pérdidas de precisión en el caso de programas interprocedurales. El presente trabajo presenta la que es hasta el momento la primera adaptación del SDG a un lenguaje funcional.

Capítulo 3

Preliminares

En esta sección vamos a estudiar en mas profundidad el subconjunto de instrucciones del lenguaje Erlang seleccionado para llevar a cabo la adaptación de la técnica de slicing, así como la definición de algunos conceptos previos necesarios para poder abordar el resto del trabajo aquí expuesto.

En la Figura 3.1 podemos ver la especificación de la sintaxis del subconjunto del lenguaje Erlang sobre el que vamos a trabajar:

pr	$::=$	$\overline{f_n}$	(Programa)
f	$::=$	$\overline{atom\ fc_n}$	(Definición de Función)
fc	$::=$	$(\overline{p_m}) \rightarrow \overline{e_n} \mid (\overline{p_m})$ `when` $\overline{g_o} \rightarrow \overline{e_n}$	(Cláusula de Función)
p	$::=$	$l \mid v \mid \langle\overline{p_n}\rangle \mid [\overline{p_n}] \mid p_1 = p_2$	(Patrón)
g	$::=$	$l \mid v \mid \langle\overline{g_n}\rangle \mid [\overline{g_n}] \mid g_1\ op\ g_2 \mid op\ g$	(Guarda)
e	$::=$	$l \mid v \mid \langle\overline{e_n}\rangle \mid [\overline{e_n}]$	(Expresión)
	$\mid$	$e_1\ op\ e_2 \mid op\ e \mid e(\overline{e_n}) \mid p{=}e$	
	$\mid$	`if` $\overline{ic_n}$ `end` $\mid$ `case` e `of` $\overline{cc_n}$ `end`	
	$\mid$	`fun` $atom/number \mid$ `fun` $\overline{fc_n}$ `end`	
	$\mid$	`begin` $\overline{e_n}$ `end` $\mid [e \mid\mid \overline{fg_n}]$	
l	$::=$	$number \mid string \mid atom$	(Literal)
ic	$::=$	$\overline{g_m} \rightarrow \overline{e_n}$	(Cláusula if)
cc	$::=$	$p \rightarrow \overline{e_n} \mid p$ `when` $\overline{g_m} \rightarrow \overline{e_n}$	(Cláusula case)
fg	$::=$	$p \leftarrow e \mid e$	(Filtros y Generadores)
op	$::=$	$+ \mid - \mid * \mid / \mid div \mid rem \mid ++ \mid --$	(Operación)
	$\mid$	$not \mid and \mid or \mid xor \mid == \mid /=$	
	$\mid$	$=< \mid < \mid >= \mid > \mid =:= \mid =/=$	

Figura 3.1: *Sintaxis del subconjunto del lenguaje Erlang seleccionado.*

Cabe destacar que, por simplicidad, se han dejado fuera del subconjunto del lenguaje secuencial Erlang con el que se va a trabajar en este estudio algunas instrucciones y características más avanzadas del lenguaje Erlang completo. Algunos ejemplos de estas características son los Records (`-record(person, name = , phone = [], address)`, constructores binarios (`Bin = «E1, E2, ... En»`), Map Expressions (`M = #{{"w", 1} =>f()}`) o las expresiones `Try`, `Catch` y `Throw`.

A continuación detallamos cada uno de los componentes que forman la sintaxis mostrada. Tener en cuenta que mediante el uso de la notación $\overline{f_n}$ se representa la secuencia $f_1 \dots f_n$.

- Un programa Erlang es una conjunto de definiciones de funciones ($pr ::= \overline{f_n}$)

- Cada definición de función esta formada por una secuencia de n pares $atom\ fc$ donde $atom$ representa el nombre de la función con aridad n y fc es una cláusula de función.

- Al mismo tiempo, las cláusulas de función están formadas por secuencias de patrones encapsuladas entre paréntesis ($\overline{p_m}$) seguidas de una secuencia de guardas $\overline{g_o}$. La declaración de la función se separa mediante una flecha a la derecha del cuerpo de la misma, formada por una secuencia de expresiones $\overline{e_n}$ tal y como se ve en el Ejemplo 1

Ejemplo 1

$f(X,\ Y,\ Z)$ `when` $X > 0; Y > 1; Y < 5\ \rightarrow$
$X + Y,$
$Z.$

- Un patrón puede ser un literal (un número, una cadena de texto o simplemente un átomo), una variable, un ajuste de patrones entre dos patrones $p_1 = p_2$ o incluso una tupla o lista de otros patrones $\langle \overline{p_n} \rangle$ $[\overline{p_n}]$.

- Las variables en erlang se representa por una secuencia alfanumérica encabezada por una letra en mayúscula o un guión bajo. Las variables anónimas se representan únicamente por un guión bajo, sin embargo, por claridad de la notación se representarán por el símbolo $\perp$ en el presente documento.

- Siguiendo la misma pauta que en el punto anterior, las tuplas se representarán con $\langle \overline{e_n} \rangle$ en lugar de $\{\overline{e_n}\}$.

- Las guardas son muy parecidas a los patrones con la salvedad de que representan un valor booleano y no permiten realizar ajuste de patrones.

- Una expresión pueden ser:
  - un literal o variable
  - una tupla o lista de expresiones: $\langle\overline{e_n}\rangle$, $[\overline{e_n}]$
  - una operación básica (aritméticas, lógicas y de comparación) : e_1 *op* e_2, *op* e
  - una asignación: $p{=}e$
  - una aplicación de función: $e(\overline{e_n})$
  - una instrucción de control if o *case* formada por una secuencia de clausulas específicas de cada instrucción $\overline{ic_n}$, $\overline{cc_n}$
  - un identificador de función átomo/aridad
  - una declaración de función anónima, formada por una secuencia de cláusulas de función, de la misma forma que en las definiciones de función.
  - un bloque de instrucciones, en el que el valor retornado es la última expresión evaluada.
  - o una lista de compresión (o *list comprehension*), formada por generadores ($p \leftarrow e$) ,filtros sobre las listas generadas y la expresión final devuelta.

Después de ver la sintaxis, hay que destacar que una de las características más importantes de Erlang es la de la selección de cláusulas de función mediante ajuste de patrones. Cuando una llamada a función es evaluada en Erlang, el compilador intenta realizar ajuste de patrones entre la llamada y la primera cláusula de la definición de la función asociada. En caso de no poder realizarse el ajuste con esta cláusula, se continua con las siguientes hasta que una de ellas permite realizar el ajuste de patrones de forma satisfactoria. Cuando se realiza el ajuste de patrones con una de las cláusulas, se procede a evaluar las expresiones del cuerpo de dicha cláusula y las siguientes cláusulas de la función son ignoradas. En el caso de que ninguna cláusula realice ajuste de patrones se lanza una excepción.

A partir de este punto se va a asumir la existencia de una serie de funciones y conjuntos necesarios para poder entender algunas definiciones básicas que se detallarán en las próximas secciones y capítulos del documento:

- La función *pos* asigna un identificador (que llamaremos posición del programa) para cada construcción sintáctica del programa (por ejemplo, patrones, guardas, expresiones, etc.). Las posiciones de un programa se usarán para identificar de forma inequívoca cada elemento del programa. Concretamente, la posición de programa de un elemento identifica la fila y la columna donde esta empieza y la fila y columna donde esta termina.

- Análogamente usaremos la función *comp*, que proporciona la funcionalidad inversa de la función *pos*, devolviendo el componente asociado a una posición específica.

- Por último, se usarán los conjuntos finitos *Vars*, *Literal* y *Ops*, que contienen todas la variables, literales y operadores de un programa.

Capítulo 4

ERLANG DEPENDENCE GRAPH

En esta sección se va a mostrar la adaptación realizada del SDG para el subconjunto del lenguaje Erlang especificado en el capítulo 3. Su definición esta basada en la representación gráfica de cada uno de los componentes de un programa.

4.1. Representación gráfica de los componentes de un EDG

Definición 4.1.1 (Representación gráfica) *La representación gráfica de un programa Erlang es un grafo etiquetado* $(\mathcal{N}, \mathcal{C})$ *donde* $\mathcal{N}$ *son los nodos y* $\mathcal{C}$ *son los arcos. Adicionalmente, las siguientes funciones están asociadas al grafo:*

$type : \mathcal{N} \to \mathcal{T}$
$pos : \mathcal{N} \to \mathcal{P}$
$function : \mathcal{N} \to \{(atom, number)\}$
$child : (\mathcal{N}, number) \to \mathcal{N}$
$lasts : \mathcal{N} \to \{\mathcal{N}\}$
$rootLasts : \mathcal{N} \to \{\mathcal{N}\}$

Para cada función del programa existe una definición de función en el grafo que se construye composicionalmente de acuerdo a ciertos casos.

- La función *type* devuelve el tipo de un nodo, donde $\mathcal{T}$ es el conjunto de tipos de nodos posibles en un EDG, compuesto por:
`function`$_{\texttt{in}}$, `clause`$_{\texttt{in}}$, `pm`, `guards`, `fid` (identificador de función), `var`, `lit`, `block`, `case`, `if`, `tuple`, `list`, `op`, `call`, `lc` (lista de compresión), y `return`.

- La función *pos* devuelve la posición de programa asociada a un nodo.

- La función parcial *function* está definida únicamente para nodos de tipo $\texttt{function}_{\texttt{in}}$. Devuelve una tupla que contiene el nombre de la función y su aridad.

- La función *child* devuelve el hijo de un nodo posicionado en la posición especificada por el número que se le pasa como parámetro.

- *lasts* devuelve el nodo final asociado a el nodo en cuestión.

- Dado un nodo que tiene asociado un conjunto de grafos finales, la función *rootLast* devuelve los nodos iniciales de los grafos finales asociados al nodo en cuestión. Esa función es muy útil para obtener los nodos iniciales de todos los argumentos de una cláusula de función.

La definición del EDG se basa en la representación gráfica de cada uno de los componentes en los que se divide un programa. Tal y como se ve en la Figura 4.1, los distintos componentes de un EDG se pueden agrupar en cuatro categorías:

- Definiciones de función (*Function Definition*)

- Clausulas (*Clause (c)*)

- Expresiones (*Expression (e)*)

- Patrones (*Pattern (p)*)

Cada grafo de las distintas categorías mostradas representa una construcción sintáctica distinta, de forma que el EDG se construye composicionalmente a partir de estos grafos. La composición se construye reemplazando algunos nodos por un grafo particular de la Figura 4.1, de esta forma se consigue representar definiciones sintácticas complejas como por ejemplo definiciones completas de funciones. De forma mas específica, podemos realizar:

- Los nodos etiquetados con una `c` deben ser sustituidos por cualquiera de los grafos de la categoría de las cláusulas.

- Los nodos etiquetados con una `e` deben ser sustituidos por uno de los grafos que representan expresiones o definiciones de función (en el caso de las funciones anónimas)

- Los nodos etiquetados con `p` deben ser sustituidos por uno de los grafos que representan los patrones.

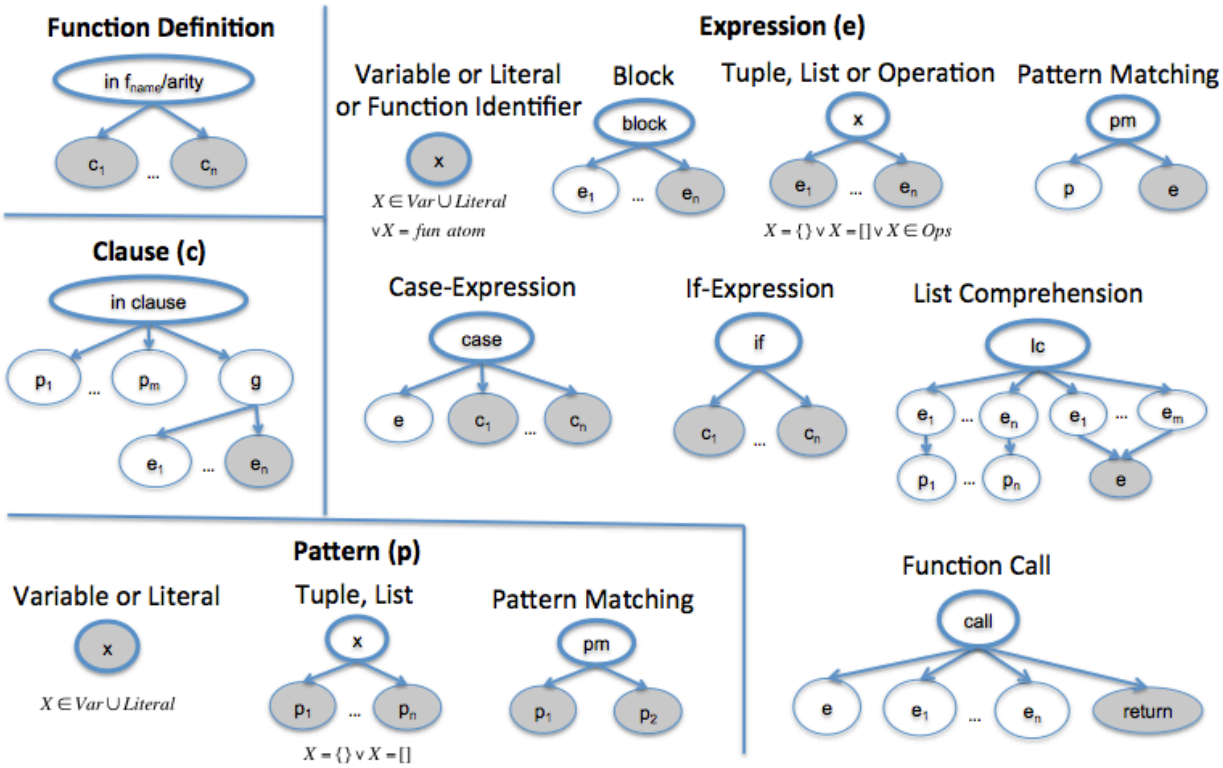


Figura 4.1: *Patrones de los grafos de los componentes de un EDG*

Para poder realizar la sustitución de un nodo por uno de los grafos se conectan todos de los arcos de entrada del nodo al nodo inicial del grafo que va a sustituirlo. En la Figura 4.1, podemos ver estos nodos iniciales de cada grafo destacados con una línea más gruesa. Asimismo, se deben conectar todos los arcos de salida del nodo a reemplazar con los nodos finales del grafo. En la Figura vemos estos nodos finales de cada grafo diferenciados mediante un sombreado. En el caso de que uno de estos nodos finales se reemplace al mismo tiempo por otro subgrafo, los nodos finales pasan a ser recursivamente los nodos finales de dicho subgrafo.

A continuación se va a explicar brevemente cada grafo por separado:

Definición de Función (*Function Definition*): En este grafo el nodo inicial incluye información sobre el nombre de la función y su aridad. En el caso de las funciones anónimas, f_{name} toma el valor de $\perp$. Las diferentes cláusulas de la función están representadas por los nodos $\texttt{c}_1 \ldots \texttt{c}_n$ y debe existir al menos una en cada definición de función.

Cláusula (*Clause*): Las cláusulas se usan tanto en la definición de funciones como en las expresiones de control `case` y `if`. En el caso de las cláusulas de función, cada una contiene cero o mas patrones (los nodos $p_1 \dots p_m$) que representan los argumentos de dicha función. En el caso de las expresiones `case` cada cláusula tiene exactamente un patrón. Sin embargo, en el caso de las expresiones `if` no existen patrones.

El nodo `g` (único) representa todas las guardas (cero o mas) de la cláusula. Las guardas imponen restricciones adicionales en la selección de los distintos patrones.

Por último, existe un grafo por cada expresión ($e_1 \dots e_n$) en el cuerpo de la cláusula.

Podemos ver un ejemplo de la definición de una función junto con sus cláusulas, guardas y expresiones en la Figura 4.2

```
add(X,0)->X;
add(X,Y)->X+Y.
```

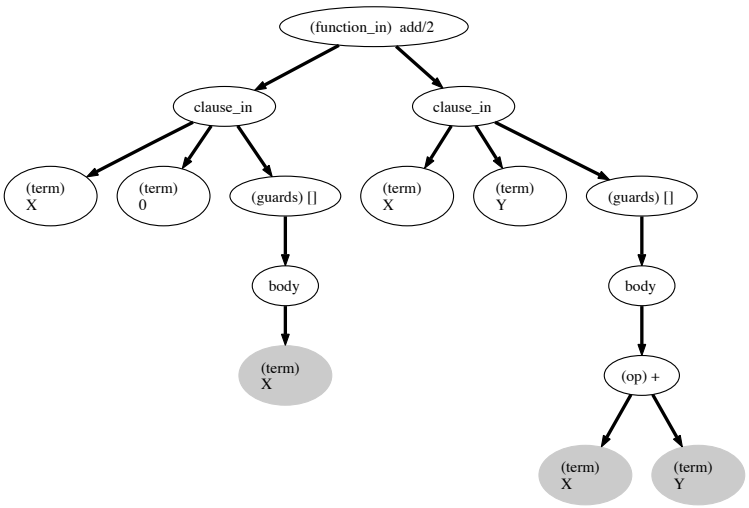


Figura 4.2: *Ejemplo de declaración de función con sus cláusulas, guardas y expresiones.*

Variable/Literal: Las variables o literales se pueden usar tanto en los patrones como en las expresiones. Están representadas por un único nodo que tiene el rol tanto de nodo inicial como de final.

Identificador de función (*Function Identifier*): Este grafo se usa en las llamadas a funciones de alto nivel de forma que identifica a una función con su nombre y aridad. Mediante este tipo de expresiones se permite por

ejemplo elegir en tiempo de ejecución qué función va a ser invocada. Está representado también por un único nodo, tanto inicial como final.

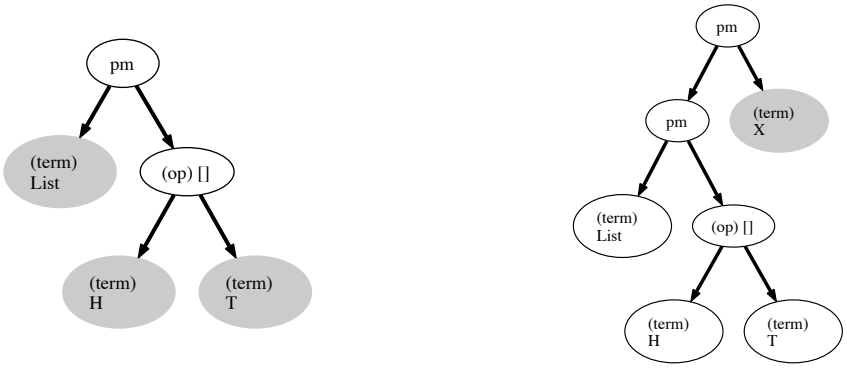


Figura 4.3: *Ejemplos de grafos de ajustes de patrones*

Ajuste de patrones (*Pattern Matching*): El ajuste de patrones puede ser utilizado tanto como un patrón o como una expresión. Como podemos ver en la Figura 4.3, la única diferencia reside en que si se trata de un patrón, los nodos finales son los dos subpatrones de la expresión, tanto la parte derecha como la parte izquierda. Sin embargo, en el caso de que sea una expresión, los nodos finales son los de la subexpresión, es decir, los de la parte derecha del ajuste de patrones.

Bloque (*Block*): Este grafo contiene un número de expresiones ($\mathtt{e}_1 \ldots \mathtt{e}_n$), en las que el nodo final es el último nodo de la última expresión ($\mathtt{e}_n$).

Tupla/Lista/Operación (*Tuple/List/Operation*): Tanto las tuplas como las listas pueden ser patrones o expresiones, mientras que las operaciones únicamente pueden ser expresiones. El nodo inicial puede ser la tupla ({}), la lista ([]) o el operador ($+$, $*$, etc.) respectivamente. Los nodos finales de estas expresiones son los nodos finales de todas las subexpresiones implicadas ($\mathtt{e}_1 \ldots \mathtt{e}_n$). Los grafos de la Figura 4.4 muestran ejemplos de representaciones gráficas de Tuplas, Operaciones y Bloques.

Expresión case (*Case-Expression*): En este tipo de instrucciones, la expresión evaluada se representa por `e`, y el último nodo de sus cláusulas son sun nodos finales. En la Figura 4.5 podemos ver un ejemplo del grafo de esta expresión.

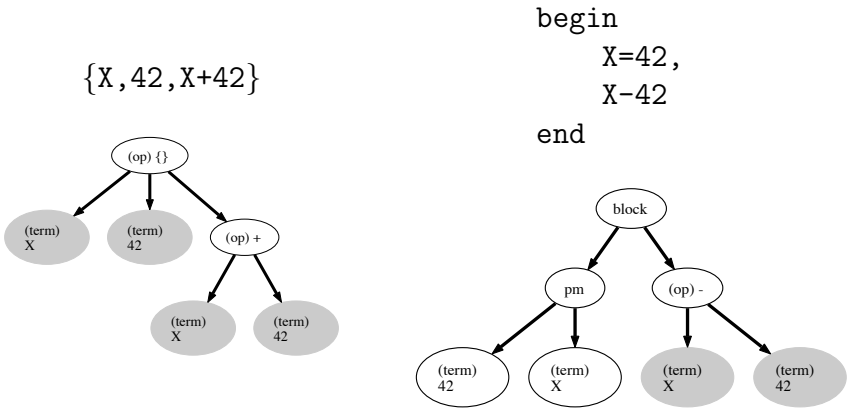


Figura 4.4: *Ejemplo de grafo: tuplas, operaciones y bloques*

Expresión if (*If-Expression*): El grafo de esta instrucción es muy parecido al de la expresión case, exceptuando que en esta caso no existe la expresión a evaluar.

```
case X of
    0 -> X;
    Y -> X+Y
end.
```

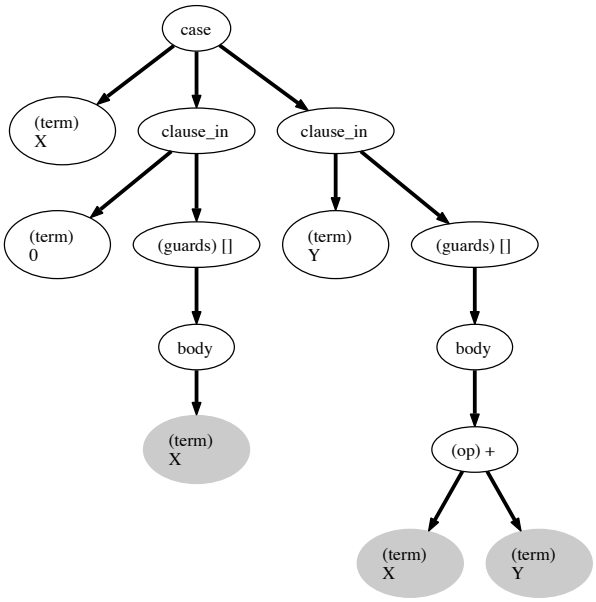


Figura 4.5: *Ejemplo de instrucción case*

Llamada a función (*Function Call*): La función se representa en este caso por `e` y los argumentos son $\texttt{e}_1 \ldots \texttt{e}_n$. En cuanto al nodo final, se trata

del nodo `return`, que representa la salida de la llamada a función. Tal y como puede verse en los ejemplos de la Figura 4.6, las llamadas a funciones anónimas se comportan de la misma manera, exceptuando que del nodo que representa el nombre de la función se ha sustituido por la declaración de la función en cuestión.

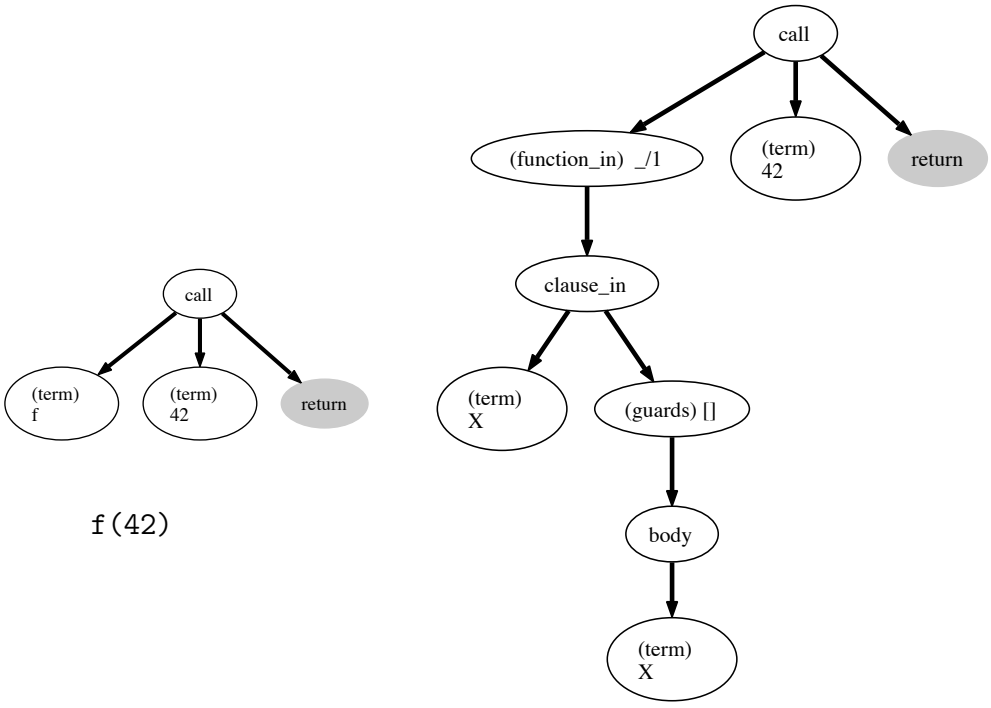


Figura 4.6: *Ejemplos de llamadas a funciones*

Listas de compresión (*List Comprehension*): Una lista de compresión contiene n generadores formados por una expresión y un patrón; m filtros ($\mathtt{e}_1 \ldots \mathtt{e}_m$) que se aplican sobre los generadores y una expresión final (`e`)

4.2. Tipos de Arcos del EDG

4.2.1. Arcos de Control

La representación gráfica de un programa mediante el uso de grafos define implícitamente las dependencias de control entre los distintos componentes del programa.

Definición 4.2.1 (Dependencia de control) *Dada la representación gráfica de un programa Erlang* $(\mathcal{N}, \mathcal{C})$ *y dos nodos* $n, n' \in \mathcal{N}$*, podemos afirmar que* n' *es* dependiente de control *de* n *si y solo si* $(n \to n') \in \mathcal{C}$*.*

En todos los ejemplos de componentes vistos hasta el momento, los nodos están unidos mediante arcos de control.

4.2.2. Arcos de Datos

Al contrario que en los lenguajes imperativos, en Erlang la definición de dependencia de datos se complica. Esto principalmente está causado por la presencia del ajuste de patrones. A continuación explicamos por separado los cuatro casos en los que aparece dependencia de datos:

Dependencia de datos producida por relaciones de flujo de datos.

En Erlang, al igual que en el paradigma imperativo, las relaciones de dependencia de flujo son provocadas por flujos de dependencia de datos. Este tipo de dependencias se basan en los conjuntos $Def(n)$ y $Use(n)$, que en Erlang contienen la variable definida o utilizada respectivamente en el nodo $n \in \mathcal{N}$.

Dados dos nodos $n, n' \in \mathcal{N}$, afirmamos que n' es *dependiente de flujo* de n si y solo si $Def(n) = Use(n')$, n' sea alcanzable por n y además n y n' pertenecen a la misma clausula de función.

Definimos el conjunto $\mathcal{D}_f$ como el conjunto que contiene todas aquellas dependencias de datos de este tipo, por ejemplo, $\mathcal{D}_F = \{(n, n') \mid n' \text{ es } \textit{dependiente de flujo} \text{ de } n\}$.

Dependencia de datos producida por ajuste de patrones

En esta sección, cuando hablamos de ajuste de patrones nos referimos al ajuste que se produce entre una expresión con un patrón. Por ejemplo, el grafo de `{X,Y}` se ajusta al grafo de `{Z,42}` con tres nodos; {} con {}, `X` con `Z`

y `Y` con 42. Otro ejemplo podría ser el grafo de la expresión `if X>1 ->true; _ ->false end`, que se ajusta al patrón `Y` de dos formas: `Y` con `true` y `Y` con `false`.

El ajuste de patrones se usa en tres situaciones distintas:

- En expresiones case para ajustar la expresión a evaluar del case con los patrones de las distintas cláusulas.
- En las expresiones de ajuste de patrones. Por ejemplo `List=[H|T]`.
- En las llamadas a funciones, para ajustar cada uno de los parámetros a los argumentos de las cláusulas de la función llamada.

En este punto únicamente consideraremos los dos primeros casos ya que el tercero se representa con otro tipo de arco que será expuesto en la sección 4.2.3.

Dado el nodo inicial de un patrón (n_p) y el nodo inicial de una expresión (n_e) podemos calcular todos los pares de ajustes en el grafo con la función *match* recursivamente definida de la siguiente forma:

$$
\begin{aligned}
&match(n_p, n_e) = \\
&\{(n_e, n_p) \mid type(n_e) = \texttt{var} \vee \\
&\qquad (\ type(n_p), type(n_e) \in \{\texttt{lit}, \texttt{fid}\} \\
&\qquad\quad \wedge\ elem(pos(n_p)) = elem(pos(n_e))\ )\} \cup
\end{aligned}
$$

$$
\begin{aligned}
&\{(last_e, n_p) \mid (\ type(n_p) = \texttt{var} \vee \\
&\qquad\qquad (type(n_p) = \texttt{lit} \ \wedge\ type(n_e) \in \{\texttt{op}, \texttt{call}\}) \vee \\
&\qquad\qquad (type(n_p) = \texttt{tuple} \ \wedge\ type(n_e) = \texttt{call}) \vee \\
&\qquad\qquad (type(n_p) = \texttt{list} \ \wedge\ type(n_e) \in \{\texttt{op}, \texttt{call}, \texttt{lc}\})\ ) \\
&\qquad\quad \wedge\ last_e \in lasts(n_e)\} \cup
\end{aligned}
$$

$$
\begin{aligned}
&\{edge \mid (\ (type(n_p) \in \{\texttt{lit}, \texttt{tuple}, \texttt{list}\} \wedge type(n_e) \in \{\texttt{case}, \texttt{if}, \texttt{pm}, \texttt{block}\}) \\
&\qquad \wedge\ edge \in \bigcup_{n'_e \in rootLasts(n_e)} match(n_p, n'_e)\ ) \vee \\
&\qquad (\ type(n_p) = \texttt{pm} \ \wedge\ edge \in \bigcup_{n'_p \in rootLasts(n_p)} match(n'_p, n_e)\ )\} \cup
\end{aligned}
$$

$$
\begin{aligned}
&\{edge \mid type(n_p) \in \{\texttt{tuple}, \texttt{list}\} \ \wedge\ type(n_e) = type(n_p) \\
&\qquad \wedge\ |\{n' \mid (n_e \to n') \in \mathcal{C}\}| = |\{n' \mid (n_p \to n') \in \mathcal{C}\}| \\
&\qquad \wedge \bigwedge_{i \in 1...|children(n_e)|} match(child(n_p, i), child(n_e, i)) \neq \emptyset \\
&\qquad \wedge\ edge \in (\ (n_e, n_p) \ \cup\ (\bigcup_{i \in 1...|children(n_e)|} match(child(n_p, i), child(n_e, i))\ )\}
\end{aligned}
$$

De esta forma, los arcos de datos producidos por ajuste de patrones se construyen uniendo los nodos que realizan el ajuste. El conjunto formado por todos estos arcos son el conjunto $\mathcal{D}_{pm}$.

En la Figura 4.7 mostramos dos ejemplos sencillos de arcos de datos producidos por ajuste de patrones:

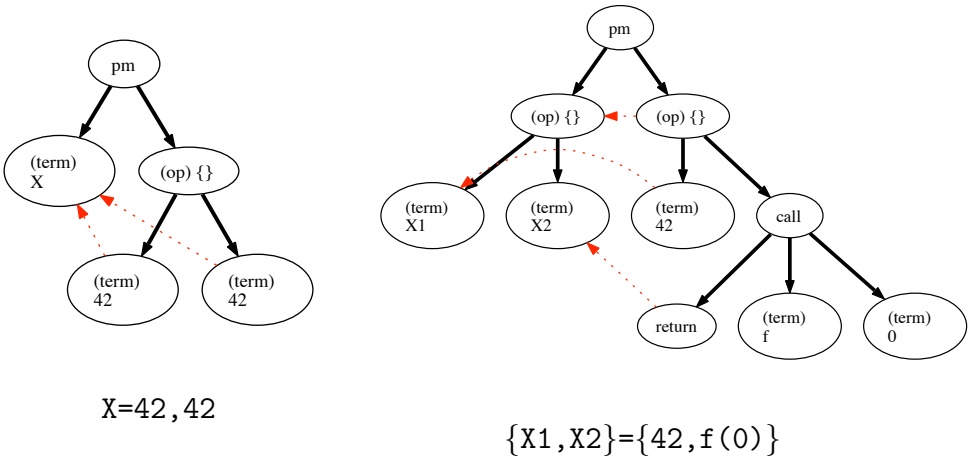


Figura 4.7: *Ejemplos de arcos de datos producidos por ajuste de patrones*

Dependencia de datos producida por restricciones impuestas por los patrones

Los patrones que aparecen en las cláusulas pueden imponer restricciones a los posibles valores de las expresiones que pueden ajustarse a estos patrones. Por ejemplo, los patrones utilizados en la definición de función `foo(X,X,0,Y) ->Y` imponen dos restricciones que deben ser tenidas en cuenta para calcular el valor de retorno `Y`: *(1)* Los primeros dos argumentos deben tener el mismo valor, y *(2)* el tercer argumento debe ser `0`.

Este tipo de restricciones se pueden representar en el EDG con un arco de datos que enlaza el patrón que impone dicha restricción y el nodo que representa la guarda de la cláusula. Esto significa que para poder evaluar las guardas, las restricciones de los nodos a los que apunta deben ser tenidas en cuenta. El conjunto de todas los arcos de datos producidos por este tipo de restricciones en un grafo se denomina $\mathcal{D}_r$, y puede ser calculado fácilmente con la función *constraints*. Esta función toma el nodo inicial de un patrón y el conjunto de variables repetidas en los parámetros de la cláusula asociada a dicho patrón, devolviendo todos los nodos en el patrón que imponen restricciones.

$$constraints(n, RVars) =$$

$$\begin{cases} \{n\} & type(n) = \texttt{lit} \ \vee \\ & (type(n) = \texttt{var} \ \wedge \ comp(pos(n)) \in RVars) \\ \{n\} \cup \bigcup\limits_{n' \in children(n)} constraints(n', RVars) & type(n) \in \{\texttt{list}, \texttt{tuple}\} \\ \bigcup\limits_{n' \in children(n)} constraints(n', RVars) & type(n) = \texttt{pm} \\ \emptyset & otherwise \end{cases}$$

En la Figura 4.9 podemos ver un ejemplo de grafo con cuatro arcos de datos provocados por restricciones impuestas por patrones. Dos de ellos nacen de las dos apariciones de la variable repetida `X`, otro desde el nodo que impone la restricción de valor 0 y por último un arco que une el nodo tupla del patrón con la guarda al imponer una restricción estructural.

```
g({X,0},X) ->X.
```

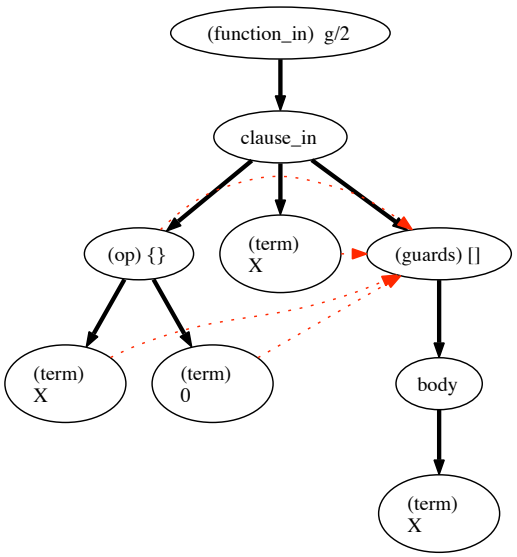


Figura 4.8: *Ejemplos de arcos de datos producidos por restricciones en los patrones*

Dependencias producidas en las llamadas a funciones

El valor de retorno de una llamada de función siempre depende de la función que ha sido llamada. Para poder representar este tipo de dependencia

añadimos un arco de cada nodo que puede representar el nombre de la función que esta siendo llamada al nodo que representa el valor de retorno en la llamada de la función. Esto ocurre en Erlang debido a la característica de las funciones de alto nivel, en las que puede ocurrir que no sepamos de forma estática el nombre de la función que va a ser invocada. Hay que tener en cuenta que el nombre de una función siempre está representado por un nodo de tipo `atom`, `variable` o `fid`.

Este tipo de dependencias se calcula utilizando una función simple que intenta encontrar los nodos que definen las posibles funciones que pueden ser aplicada en una llamada. Esta función considera distintos casos:

- Para el caso de nodos de identificadores de función, átomos (si se aplican directamente) o variables, el conjunto de nodos únicamente incluye este nodo.

- En expresiones if, case y ajuste de patrones, la búsqueda continua utilizando el nodo raíz del conjunto de los *lasts* de estas expresiones.

- En el resto de los casos (incluyendo las funciones anónimas) no se introducen ninguna dependencia de este tipo.

Representaremos el conjunto que contiene todas las dependencias producidas en las llamadas a funciones por $\mathcal{D}_{fc}$.

Llegados a este punto, estamos en condiciones de definir la noción de dependencia de datos en Erlang

Definición 4.2.2 (Dependencia de Datos) *Dada la representación gráfica de un programa Erlang* $(\mathcal{N}, \mathcal{C})$ *y dos nodos* $n, n' \in \mathcal{N}$*, afirmamos que* n' *es* dependiente de datos *de* n *si y solo si* $(n, n') \in (\mathcal{D}_f \cup \mathcal{D}_{pm} \cup \mathcal{D}_r \cup \mathcal{D}_{fc})$.

4.2.3. Arcos de Entrada / Salida

Los arcos de entrada y salida representan el flujo de información en las llamadas a funciones. Existen diversas complicaciones adicionales que aparecen en lenguajes funcionales como Erlang con respecto a otros lenguajes funcionales que aparecen de forma estática cuando se intenta determinar el flujo de información interprocedural de los programas. Uno de estos problemas es que las funciones de alto nivel pueden ocultar el nombre de la función

que se está llamando. Incluso conociendo el nombre de la función, no siempre es posible saber con certeza la cláusula que se ajustará a la llamada de la función.

Ejemplo 2 En el siguiente programa, es imposible conocer de forma estática cual de las cláusulas se ajustará a la llamada de la función. Es por esto que necesitamos conectar la llamada de la función a todas las cláusulas que pueden realizar ajuste de patrones en tiempo de ejecución.

```
-export(f/1).

f(X) -> g(X).

g(1)-> a;
g(X)-> b.
```

Determinar todas las posibles cláusulas que pueden hacer ajuste de patrones con respecto a una llamada es un problema indecidible ya que depende de la terminación de una llamada de función, y probar dicha terminación es, al mismo tiempo, también indecidible. Es por eso que estamos ante un punto clave en cuanto a la precisión del grafo se refiere. Conceptualmente, se puede asumir la existencia de una función `clauses(call)` que devuelve todas aquellas cláusulas que realizan ajuste de patrones con una determinada llamada de función. En la práctica, se puede utilizar algún tipo de análisis estático para poder aproximar este problema. Así, en la implementación de la técnica de slicing de este trabajo se hace uso de la herramienta Typer [15] que usa el sistema de inferencia de tipos de Dialyzer [16], que permite producir una aproximación completa.

Definición 4.2.3 (Arcos de Entrada) *Dado un grafo $(\mathcal{N}, \mathcal{C})$, definimos el conjunto $\mathcal{I}$ de arcos de entrada como un conjunto de arcos dirigidos. Por cada grafo correspondiente a una llamada de función call, se debe analizar el ajuste entre el subgrafo de cada parámetro de la llamada con respecto a cada subgrafo de los argumentos de las cláusulas que pertenecen a clauses(call). De esta forma existirá un arco en $\mathcal{I}$ por cada par de nodos que se ajustan. Además, existirá un arco desde el nodo* `return` *de la llamada al nodo* $\texttt{clause}_{\texttt{in}}$ *de la cláusula.*

Definición 4.2.4 (Arcos de Salida) *Dado un grafo* $(\mathcal{N}, \mathcal{C})$*, definimos el conjunto* $\mathcal{O}$ *de arcos de salida como un conjunto de arcos dirigidos. Por cada grafo correspondiente a una llamada de función call y cada cláusula que pertenece a* `clauses(call)` *existirá un arco en* $\mathcal{O}$ *desde cada nodo final del grafo de la cláusula hasta el nodo* `return` *de la llamada.*

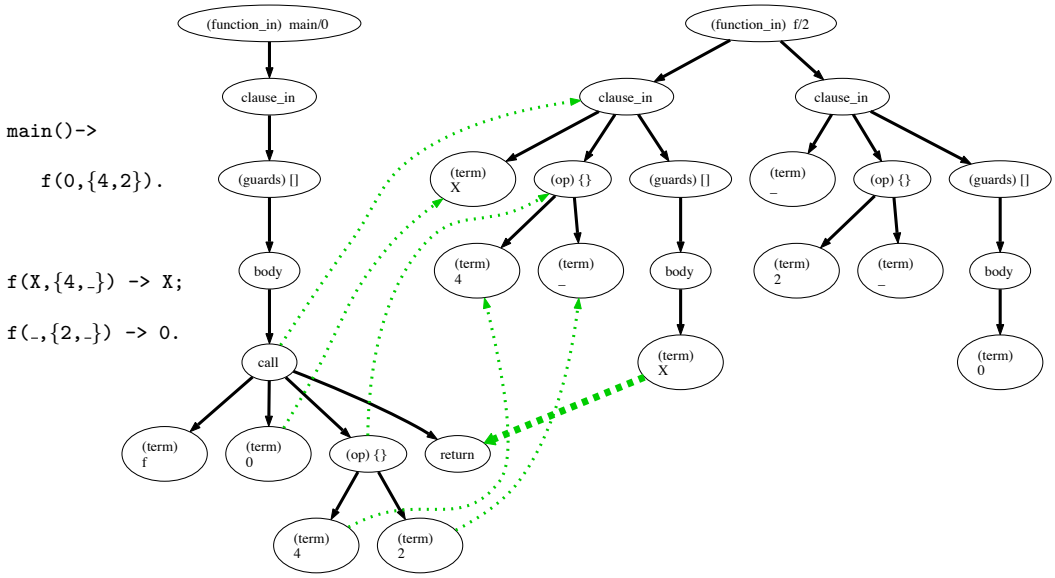


Figura 4.9: *Ejemplos de arcos de entrada (arco fino punteado) y salida (arco grueso) entre distintas funciones*

4.2.4. Arcos de Resumen

Los arcos de resumen se usan para capturar de forma precisa dependencias inter-funcionales. Básicamente, este tipo de arcos nos indican los argumentos de una función que tienen influencia en el resultado de dicha función (para una explicación mas detallada consultar [12]). Al igual que en el paradigma imperativo, los arcos de resumen han de ser calculados una vez se han calculado todas las otras dependencias del programa.

Definición 4.2.5 (Arcos de Resumen) *Dado un grafo* $(\mathcal{N}, \mathcal{C})$*,definimos el conjunto* $\mathcal{S}$ *de arcos de resumen como un conjunto de arcos dirigidos. Existirá un arco de resumen entre dos nodos* n, n' *de un grafo si* n *pertenece al subgrafo de un parámetro de una llamada de función,* n' *es el nodo de tipo return de dicha llamada, y existe un arco de entrada saliendo de* n *que alcanza a uno de los nodos finales de la cláusula de la función.*

En el ejemplo de la Figura 4.10 podemos ver como existe un arco de resumen entre el parámetro de la llamada 0 y el nodo *return* de la misma. Esto es debido a que en la llamada existe un arco de salida que parte desde el nodo 0 y alcanza (a través de un arco de datos) el nodo final X de la cláusula de la función. Por claridad, en este ejemplo se han ocultado los nodos y arcos que no están involucrados en la explicación del ejemplo.

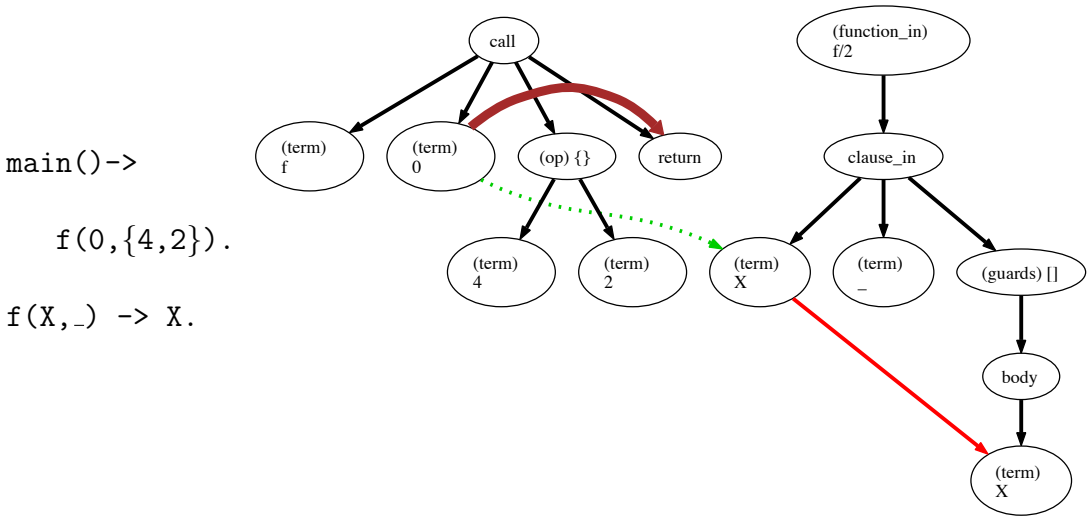


Figura 4.10: *Ejemplo de arco de resumen (color marrón con trazo grueso)*

4.3. EDG

Llegados a este punto, estamos en disposición de formalizar la definición del Erlang Dependence Graph (EDG)

Definición 4.3.1 (Erlang Dependence Graph) *Dado un programa Erlang $\mathcal{P}$, el* Erlang Dependence Graph (EDG) *es un grafo etiquetado dirigido $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ donde $\mathcal{N}$ son los nodos y $\mathcal{E} = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{O}, \mathcal{S})$ son los arcos.*

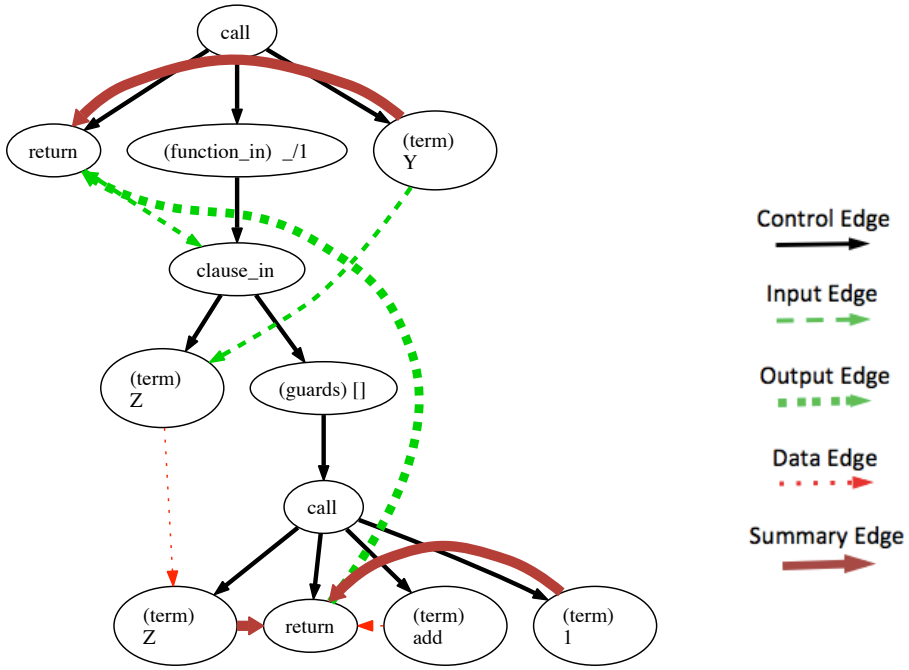


Figura 4.11: *EDG asociado a la expresión* `fun(Z)->add(Z,1) end(Y)` *del Ejemplo 5.1.*

Ejemplo 3 El EDG correspondiente a la expresión `fun(Z)->add(Z,1) end(Y)` en la línea `(9)` del código de la Figura 5.1 se muestra en la Figura 4.11. Como puede verse, este se trata de un buen ejemplo del EDG ya que la representación gráfica de esta instrucción engloba casi todos los tipos de arcos recién explicados (únicamente no aparecen $\mathcal{D}_{pm}$ y $\mathcal{D}_r$)

Capítulo 5

Técnica de Fragmentación en Erlang

Como hemos podido ver en las secciones previas, el EDG es una herramienta muy potente para realizar distintos análisis estáticos sobre programas Erlang, ofreciendo aún más utilidad cuando aplicamos la técnica de slicing de programas. En este capítulo mostramos que nuestra adaptación del SDG al lenguaje Erlang consigue mantener la propiedad mas importante del SDG: el cálculo del slice de un programa a partir de un EDG tiene un coste lineal con el tamaño (número de nodos) del EDG. Esto significa que, después de construir el EDG, podemos obtener el slice resultante recorriendo una única vez los nodos de dicho grafo, en otras palabras, accediendo una única vez a cada nodo del EDG

Hay que destacar que el algoritmo que se va a definir para recorrer el EDG esta basado en el algoritmo básico del SDG descrito en [12]. Sin embargo, dicho algoritmo no es el estándar, se necesita incluir una pequeña modificación que nos permite mejorar la precisión en el slice obtenido.

Una de las ventajas mas destacables del EDG respecto al SDG es el hecho de que minimiza el nivel de granularidad del mismo, de forma que en el EDG todas las construcciones sintácticas se descomponen al nivel máximo (átomos, literales, variables, etc.) Tradicionalmente, en los trabajos realizados para lenguajes imperativos cada nodo del grafo representa una línea completa en el código. Sin embargo, el nivel de granularidad utilizado en el EDG nos permite obtener aquellas partes del programa que afectan a una subexpresión en una posición concreta del programa.

Definición 5.0.2 *Dado un EDG $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, un criterio de slicing para $\mathcal{G}$ es un nodo $n \in \mathcal{N}$.*

En la práctica, la construcción del EDG es transparente para el programador, el cual se encarga únicamente de seleccionar el criterio de slicing en el código fuente. Concretamente, en la implementación de este trabajo, esto se lleva a cabo remarcando simplemente una expresión en el código, tal y como veremos en el capítulo 6. Esta acción se transforma automáticamente en una posición del programa, que al mismo tiempo equivale con un identificador de un nodo en el EDG. Dicho nodo se trata de la entrada del Algoritmo 1 que nos permitirá extraer los slices a partir del EDG.

Básicamente, el Algoritmo 1 funciona de la siguiente manera:

- En la primera fase el algoritmo incluye en el slice todos aquellos nodos alcanzables desde el criterio de slicing siguiendo todos los arcos de los tipos $\mathcal{C} \cup \mathcal{D} \cup \mathcal{I}$ en sentido inverso (*backwards*).

- En la segunda fase del algoritmo, se añaden al slice todos aquellos nodos alcanzables desde los nodos ya incluidos siguiendo todos los arcos de los tipos $\mathcal{C} \cup \mathcal{D} \cup \mathcal{O}$.

- En ambas fases se incluyen los nodos alcanzables siguiendo en orden inverso los arcos de resumen ($\mathcal{S}$) únicamente en el caso de que estén conectados a un nodo que ya pertenece al slice a través de un arco de entrada.

Algorithm 1 Slicing de programas interprocedurales

Entrada: Un EDG $\mathcal{G} = (\mathcal{N}, \mathcal{E} = (\mathcal{C}, \mathcal{D}, \mathcal{I}, \mathcal{O}, \mathcal{S}))$ y un criterio de slicing $\mathcal{SC}$
Salida: Un conjunto de nodos $Slice \in \mathcal{N}$
Devuelve traverse(traverse($\{\mathcal{SC}\}, \mathcal{I}$), $\mathcal{O}$)

function traverse($Slice, X$)
  repeat
    $Slice = Slice \cup \{n' \mid (n' \to n) \in (\mathcal{C} \cup \mathcal{D} \cup X)$ con $n \in Slice\}$
      $\cup \{n_2 \mid (n_2 \to n_1) \in \mathcal{S} \wedge (n_2 \to n_3) \in \mathcal{I}$ con $n_1, n_3 \in Slice\}$

  until se alcanza un punto fijo
return $Slice$

Como puede verse, la definición del algoritmo es similar a la estándar del SDG definida en [12] exceptuando el tratamiento que se realiza cuando se atraviesan los arcos de resumen en sentido inverso. En el algoritmo del SDG, los arcos de resumen van desde los parámetros de entrada hasta los

parámetros de salida de la función, y se atraviesan en todos los casos. Además, dichos parámetros no pueden ser descompuestos. Sin embargo, en el lenguaje Erlang los argumentos de una función pueden ser al mismo tiempo estructuras de datos compuestas, por lo que es posible que únicamente una parte de esta estructura influencie en el criterio de slicing. Para tener este aspecto en cuenta, en el algoritmo recién expuesto únicamente se atraviesan los arcos de resumen si provienen de nodos que realmente se requieren. La forma de saber si dichos nodos son requeridos es observar el arco de entrada que sale de ellos, en el caso de que el nodo alcanzado pertenezca al slice se podrán atravesar los arcos de resumen en cuestión. Por supuesto, para poder conocer esta información es necesario haber realizado un análisis previo de la función que es llamada.

Por último, una vez se tienen todos los nodos que pertenecen al slice, es muy sencillo mapear dichos nodos otra vez en el código fuente. Para un programa $\mathcal{P}$, la colección exacta de posiciones (líneas y columnas) que pertenecen al slice es $\{pos(n) \mid n \in Slice(\mathcal{P})\}$ donde la función *Slice* implementa el Algoritmo 1.

Para poder asegurar que el programa o slice devuelto por el algoritmo es ejecutable, teneos que reemplazar todas la expresiones que no pertenecen al slice por el átomo `undef` y todos os patrones no usados por la variable anónima " _ ".

Ejemplo 4 El código del programa 5.2 se trata del slice obtenido a partir de la aplicación del algoritmo 1 sobre el código del programa interprocedural 5.1 tomando como criterio de slicing la expresión compuesta por la llamada a la función `add(Z,1)` de la función `while`, línea `(9)`.

Se observa como el slice obtenido es completamente ejecutable. Esto se ha conseguido gracias a la sustitución de las instrucciones que no forman parte del slice y cuya línea entera no puede ser eliminada, por el átomo `undef` (líneas `(4)(10)(14)`). Por otro lado puede verse como todos los patrones no usados se han sustituido por la variable anónima " _ "(línea `(4)(6)(14)`).

Cabe destacar que este ejemplo se trata de una adaptación al lenguaje Erlang del ejemplo que se propuso en [12] para mostrar el funcionamiento del algoritmo de obtención de slice sobre el SDG. Como puede observarse, el resultado obtenido es correcto y con la precisión adecuada.

En el capítulo 7 estudiaremos un caso de estudio en mas profundidad, en el cual se detallará paso a paso la aplicación del algoritmo en la implementación desarrollada.

```
(1)   main() ->
(2)      Sum = 0,
(3)      I = 0,
(4)      {Result, _} = while(Sum, I, 11),
(5)      Result.

(6)   while(Sum, I,Top) ->
(7)      if I /= Tope ->
(8)            NSum = add(Sum, I),
(9)            NI = (fun (Z) -> add(Z,1) end)(I),
(10)           while(NSum, NI, Top-1);
(11)       I == Top ->
(12)           {Sum,Top}
(13)     end.

(14)  add(A,0) -> A;
(15)  add(A,B) -> A+B.
```

Figura 5.1: *Programa Original*

```
(1)   main() ->
(2)
(3)      I = 0,
(4)      _ = while(undef, I, 11).
(5)

(6)   while(_, I, Top) ->
(7)      if I /= Tope ->
(8)
(9)            NI = (fun (Z) -> add(Z,1) end)(I),
(10)           while(undef, NI, Top-1)
(11)
(12)
(13)     end.

(14)  add(_,0) -> undef;
(15)  add(A,B) -> A+B.
```

Figura 5.2: *Slice obtenido tras aplicar el Algoritmo 1*

Capítulo 6

Implementación

Para aplicar la técnica de slicing de programas expuesta se ha desarrollado una herramienta llamada *slicerl* que está disponible de forma pública desde la página web http://kaz.dsic.upv.es/slicErlang.html.

Esta implementación ha sido desarrollada íntegramente en el lenguaje Erlang con un total de 1500 líneas de código aproximadamente. En las siguientes secciones se detallan algunas de las características mas importantes de esta herramienta así como algunos detalles de su implementación:

6.1. Algunos detalles de la implementación

Para poder extraer de los programas Erlang la información necesaria para ser capaces de construir la representación gráfica del mismo es necesario recurrir a una representación alternativa de dicha información como son los Árboles de Sintaxis Abstracta (o *AST*, del inglés *Abstract Syntax Tree*). De esta forma, se ha decidido trabajar con la representación estándar de los Árboles de Sintaxis Abstracta de Erlang o *Abstract Format*.

6.1.1. Abstract Format

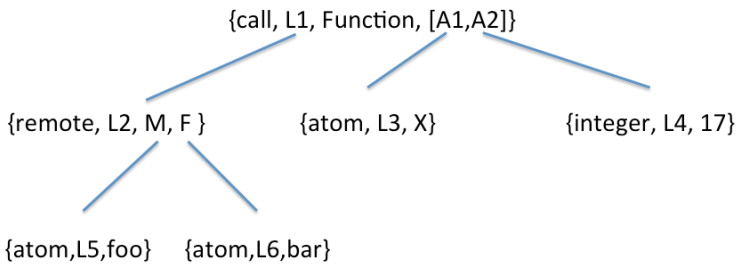
Todas las expresiones de Erlang se representan en Abstract Format mediante tuplas anidadas en forma de Árbol. En el Ejemplo 5 podemos ver una instrucción Erlang y su representación en Abstract Format, así como una representación mas visual de esta representación arbórea.

Ejemplo 5 La siguiente llamada a función en Erlang:

```
foo:bar(X,17).
```

Se representa en Abstract Format con la siguiente tupla:

{call, 1, {remote, 1, {atom, 1, foo}, {atom, 1, bar}}, [{atom, 1, X}, {integer, 1, 17}]}.
Como se puede ver en la siguiente imagen, la tupla forma una estructura de Árbol



La jerarquía la estructura de representación de la información en el Abstract Format sigue el siguiente patrón o esquema:

- Un módulo se representa en Abstract Format con una lista $[F_1 \ldots F_n]$ donde cada F representa un **form**.

- Un **form** puede ser un **atributo** o bien una **declaración de función**:

  - Por ejemplo, el atributo `-module(Mod)` se representa en Abstract Format {`attribute,LINE,module,Mod`}
  - EL Abstract Format correspondiente a una declaración de función es {`function,LINE,Name,Arity,` $[FC_1 \ldots FC_n]$} donde cada FC es el Formato Abstracto de una **cláusula de función**
    - Así, cada cláusula de función se representa por {`clause,LINE,` $[P_1 \ldots P_n], [G_1 \ldots G_n], [E_1 \ldots E_n]$} donde:
      - Cada [P] es la representación de los Patrones de una cláusula en Abstract Format
      - Cada [G] es la representación de las guardas
      - Cada [E] es la representación de las expresiones del cuerpo de la cláusula.
    - Por su parte, los patrones, guardas y expresiones son listas de expresiones (por ej. expresiones `case`, `list`, `lc`, `fun`, `block`, `var`... etc.) en Abstract Format, con la salvedad de que tanto patrones como guardas solo aceptan un subconjunto de estas expresiones.

Para poder convertir el código Erlang a su representación en Abstract Format, se ha elegido la librería estándar de Erlang *SyntaxTools* que contiene gran cantidad de módulos para el manejo de los AST junto con utilidades para la lectura de archivos de código fuente, impresión de los AST y reversión al código fuente.

La gran variedad de funcionalidad que ofrecen las distintas librerías de la distribución oficial de Erlang OTP en cuanto a la manipulación de código en tiempo de ejecución (o metaprogramación) queda mermada asimismo por la complejidad y la densidad de los módulos que contienen. Es por este motivo que se decidió utilizar el módulo SMERL (*Simple Metaprogramming for Erlang*), librería que facilita y abstrae el uso de las librerías relacionadas con el área de la metaprogramación en Erlang. En el Ejemplo 6 podemos ver que con dos líneas de código se puede obtener la representación en Abstract Format de las declaraciones de función del módulo de un fichero.

Ejemplo 6 Se tiene el siguiente módulo guardado en un fichero `test.erl`:

```
-module(test).
-export([fac/1]).

fac(0) -> 1;
fac(N) when N>0 -> mult(N, fac(N-1)).

mult(N1,N2) -> N1 * N2.
```

Mediante el uso de las siguientes dos líneas de código Erlang:

```
{ok,AbstractForm} = smerl:for_file("test.erl"),
FunctionForms = [Form||Form={function,_,_,_,_}<-smerl:get_forms(AbstractForm)].
```

Obtenemos la representación en Abstract Format del módulo:

```
[{function,7,mult,2,
     [{clause,7,
          [{var,7,'N1'},{var,7,'N2'}],
          [],
          [{op,7,'*',{var,7,'N1'},{var,7,'N2'}}]}]},
 {function,4,fac,1,
     [{clause,4,[{integer,4,0}],[],[{integer,4,1}]},
      {clause,5,
          [{var,5,'N'}],
          [[{op,5,'>',{var,5,'N'},{integer,5,0}}]],
          [{call,5,
               {atom,5,mult},
               [{var,5,'N'},
                {call,5,
                    {atom,5,fac},
                    [{op,5,'-',{var,5,'N'},{integer,5,1}}]}]}]}]}]
```

La definición completa del Asbtract Format esta disponible públicamente en http://www.erlang.org/doc/apps/erts/absform.html

Una vez se tiene la representación en Abtract Format del programa, resulta muy sencillo recorrer la estructura Arbórea que conforma extrayendo la información de las distintas expresiones del código. Para realizar el recorrido se hace uso de funciones recursivas que recorren desde los Forms que forman la definición del módulo hasta los átomos literales y variables de las distintas expresiones. Mediante el uso del ajuste de patrones y la potencia que ofrecen las listas de compresión en Erlang resulta muy sencillo recorrer dicho árbol y extraer la información necesaria para poder construirlos nodos y los arcos del grafo.

6.1.2. Representación Visual del Grafo

Para poder ver la representación en grafo de la información extraída del recorrido del Abstract Format del programa se ha utilizado el lenguaje de descripción de grafos en texto plano *DOT*. Este es un lenguaje muy potente que permite especificar de forma muy sencilla los nodos y arcos que forman un grafo. De esta forma, mediante una transformación simple de los nodos y arcos obtenidos del recorrido del Abstract Format, se puede transformar al formato .dot la representación del grafo.

Cabe destacar que la representación en forma de grafo resulta muy útil para poder analizar programas de poco tamaño (<50 líneas de código) sin excesivo flujo de información entre sus distintas instrucciones. Esto se debe

a la gran cantidad de nodos y sobretodo arcos que se generan para un fragmento de código, hecho que provoca que para programas grandes no sea útil trabajar con la representación gráfica de los EDGs, tal y como se puede ver en la Figura 7.1. Para programas de mayor tamaño, esta opción puede ser transparente al usuario, pasando directamente a la selección del criterio de slicing y a la obtención del slice en cuestión. A pesar de ello, en la implementación desarrollada se ha incluido la funcionalidad de visualización de la representación gráfica de los EDGs de los programas introducidos o de sus slices obtenidos.

6.2. Arquitectura

En esta sección se detalla la arquitectura elegida para realizar el desarrollo de la implementación de la herramienta *slicErl*, las fases por las que se atraviesa así como los distintos módulos y ficheros implicados en el ciclo de vida del proceso de slicing desarrollado.

La implementación se ha dividido en tres grandes módulos separando de forma lógica las tres grandes funcionalidades que ofrece la aplicación:

Módulo slicErlang: *(800 líneas de código aprox.)* Es el módulo encargado de crear el grafo recorriendo la estructura del Abstract Format obtenida del código original.

Módulo slicErlangDot: *(200 líneas de código aprox.)* Módulo encargado de crear el archivo .dot para la visualización gráfica del grafo

Módulo slicErlangSlice: *(400 líneas de código aprox.)* En este módulo se implementa la técnica de slicing de programas a partir del criterio de slicing seleccionado y se restaura el código del slice obtenido.

En la Figura 6.1 podemos ver de forma gráfica los pasos por los que se atraviesa en la implementación, que a continuación detallamos brevemente:

1. Se lee el fichero Erlang creado a partir del código Erlang introducido por el usuario en la aplicación con la ayuda de la librería *smerl* descrita anteriormente. En este punto, el módulo `slicErlang` se encarga de obtener el Abstract Format del código y de recorrerlo para construir los diferentes nodos y arcos que forman el EDG. A continuación se detalla el orden seguido para dicha generación de nodos y arcos del grafo:

   a) Arcos de control: Se recorre el abstract format (Forms, Funciones, Clausulas, Expresiones... etc) mediante una serie de funciones recursivas que

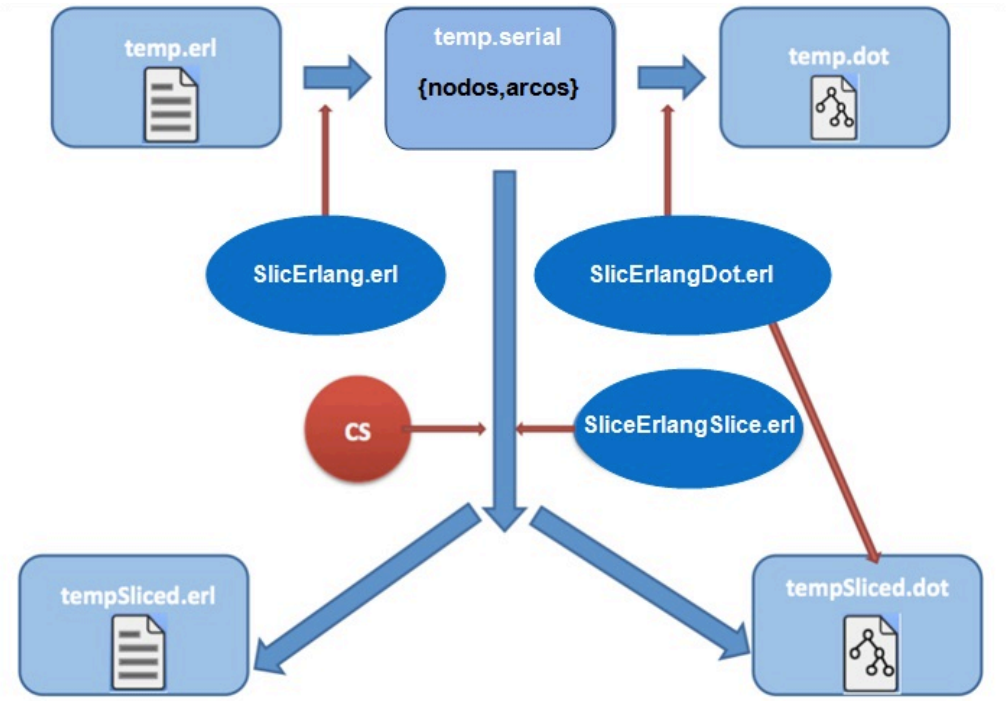


Figura 6.1: *Proceso General que se atraviesa al utilizar la aplicación slicErl*

van creando una lista de nodos y una lista de arcos a la que vamos añadiendo los distintos arcos de control encontrados.

b) Arcos de datos: Al mismo tiempo que se va recorriendo el Abstract Format se va rellenando una estructura de datos Diccionario, en la que se guarda donde se declara una variable y sus utilizaciones posteriores. De esta forma después es mucho más sencillo construir los distintos arcos de datos.

c) Arcos de entrada / salida: Se mantiene información de todas las llamadas a función así como de todas las cláusulas de función encontradas. Se realiza el patter-matching analizando la correspondencia entre argumentos de llamada y los parámetros de las distintas cláusulas asociadas. Tal y como hemos descrito en capítulos anteriores, este proceso esta apoyado el uso de la herramienta Typer [15] que usa el sistema de inferencia de tipos de Dialyzer [16].

d) Arcos de Resumen: Con toda la información recopilada en los puntos anteriores, se realiza un estudio de la alcanzabilidad entre los nodos `pattern` y el nodo de salida de la declaración de la función. En caso de existir dicho camino se unen los nodos `return` y el argumento

correspondiente de las llamadas asociadas.

Como salida de este proceso se obtiene una tupla con dos elementos, el primero es la lista de nodos del grafo y el segundo es la lista de arcos del mismo, esta información se almacena en un fichero temporal. Como funcionalidad adicional se puede elegir los tipos de arcos que se desean mostrar en el grafo seleccionándolos de una lista.

2. A partir de este punto se pueden realizar distintas acciones:

- El usuario puede querer visualizar el grafo, con lo que se utilizaría el módulo `SlicErlangDot` para transformar la tupla {nodos,arcos} al formato DOT y que pueda ser mostrado por un programa de visualización de grafos. El grafo se puede visualizar directamente en el navegador, guardar en formato .jpg o incluso guardar en el formato .dot. Como entrada del proceso de construcción del fichero dot también tenemos un fichero (*Shows.txt*) que indica los tipos de arcos a mostrar a partir de la selección realizada por el usuario en la aplicación.
- Por otra parte el usuario puede aplicar la técnica de slicing sobre el código que ha introducido en la aplicación, funcionalidad implementada en el módulo `SlicErlangSlice`. Para ello ha de resaltar el texto del criterio de slicing sobre el mismo código introducido. Obviamente, el slice seleccionado ha de ser sintácticamente válido para poder aplicar la técnica de slicing. Una vez seleccionado un criterio de slicing válido, para aplicar la técnica de slicing primeramente se ha de encontrar la localización del criterio de slicing en el código. Posteriormente se aplica el Algoritmo 1 sobre la lista de nodos y arcos que nos habíamos guardado en el paso 1 recorriendo en dos fases y en sentido inverso los arcos especificados en el Algoritmo. Como resultado se obtiene la lista de nodos y arcos que conforman el slice.

3. Una vez obtenida la lista de nodos y arcos que conforman el slice obtenido después de la aplicación de la técnica, se procede a restaurar el código a partir de dicha lista de nodos y arcos. Para llevar esto a cabo se realiza el proceso inverso al que se había realizado en el proceso de construcción de la lista de nodos y arcos del grafo a partir del código original. Una vez se obtiene el código asociado al grafo del slice se muestra al usuario para que lo compare con el código original.

4. Por último, y al igual que ocurre sobre el grafo del código original, también se puede visualizar el grafo del slice obtenido, guardarlo en formato .jpg o .dot.

6.3. Uso de la herramienta

A continuación detallamos las principales características de la interfaz y mostramos cómo sería la utilización básica de la herramienta *SlicErl* por parte del usuario.

6.3.1. Interfaz de usuario

La herramienta *SlicErl* se ha integrado en una página web simple y sencilla desde la que se pueden realizar las distintas opciones disponibles así como visualizar los distintos resultados obtenidos.

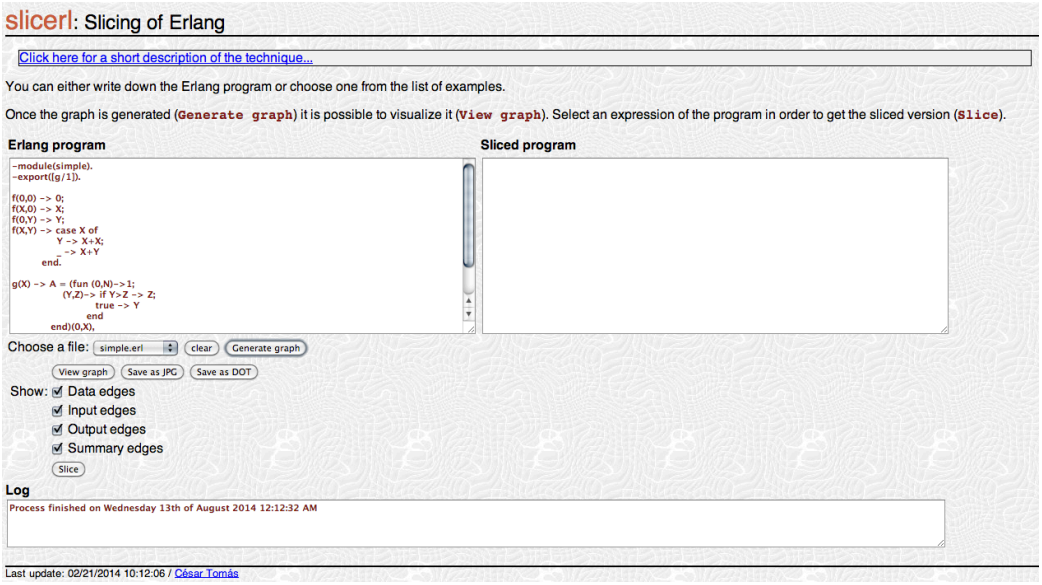


Figura 6.2: *Interfaz de la herramienta slicErl*

Como puede verse en la Figura 6.2 la interfaz de la herramienta *SlicErl* es muy simple, a continuación se detallan los aspectos mas importantes de dicha interfaz:

- Existen dos cuadros de texto, el de la derecha sirve para introducir el código sobre el que se quiere obtener el EDG o sobre el que se quiere aplicar la técnica de slicing. El cuadro de texto de la izquierda sirve para visualizar el slice obtenido después de la aplicación de la técnica de slicing.

- En la parte inferior de los cuadros de texto tenemos las distintas opciones que puede marcar el usuario como por ejemplo la generación y visualización del grafo, los arcos que se desean visualizar o la aplicación de la técnica de slicing con un criterio de slicing ya seleccionado.

- En la parte inferior vemos un cuadro de texto en el que se muestra un log informativo de las distintas acciones que realiza el usuario.

6.3.2. Utilización de SlicErl

El primer paso que sigue el usuario al acceder al sitio web de la implementación *slicerl* es la introducción del código. Para ello, el usuario tiene dos opciones, la introducción manual de dicho código o la selección de uno de los programas precargados en el sistema. Para ello, el usuario ha de seleccionarlo de entre los programas disponibles en el control desplegable ubicado en la parte inferior del cuadro de texto del código. Como puede verse en la Figura 6.3, podemos observar que se ha seleccionado el programa `simple.erl` de la lista introduciéndose automáticamente el código de dicho programa en el cuadro de texto destinado a la introducción del código a analizar.

Una vez introducido el código, el siguiente paso a seguir es el de la generación del grafo, para ello el usuario ha de pulsar sobre el botón *Generate Graph*. En este momento, el sistema muestra un mensaje en el cuadro de texto de visualización del log de los procesos, tanto si el código se ha generado correctamente como si se ha encontrado algún problema durante el proceso de generación del Árbol. En el caso de que la generación haya resultado satisfactoria el sistema mostrara un log con el mensaje:

```
Process finished on ...
```

Si por el contrario se ha encontrado un problema en la generación del grafo el sistema mostrará un mensaje con el descriptivo del error encontrado, como por ejemplo:

```
typer: Analysis failed with error report:
./temp_tt.erl:3: function g/1 undefined
```

En el caso de que la generación haya resultado satisfactoria, el usuario podrá visualizar el grafo o incluso guardarlo como imagen o como fichero `.dot`. Primeramente, el usuario debe elegir los arcos que quiere ver en la visualización del grafo, para ello ha de marcar de la lista los tipos de arcos que quiere visualizar, exceptuando los arcos de control que aparecerán siempre en los grafos. En la imagen de la Figura 6.4 podemos ver una porción del grafo del código del ejemplo `simple.erl` únicamente con la visualización de los arcos de input seleccionados.

Una vez seleccionados los arcos para la visualización, se puede visualizar el grafo directamente, guardarlo en formato JPG o incluso guardarlo en

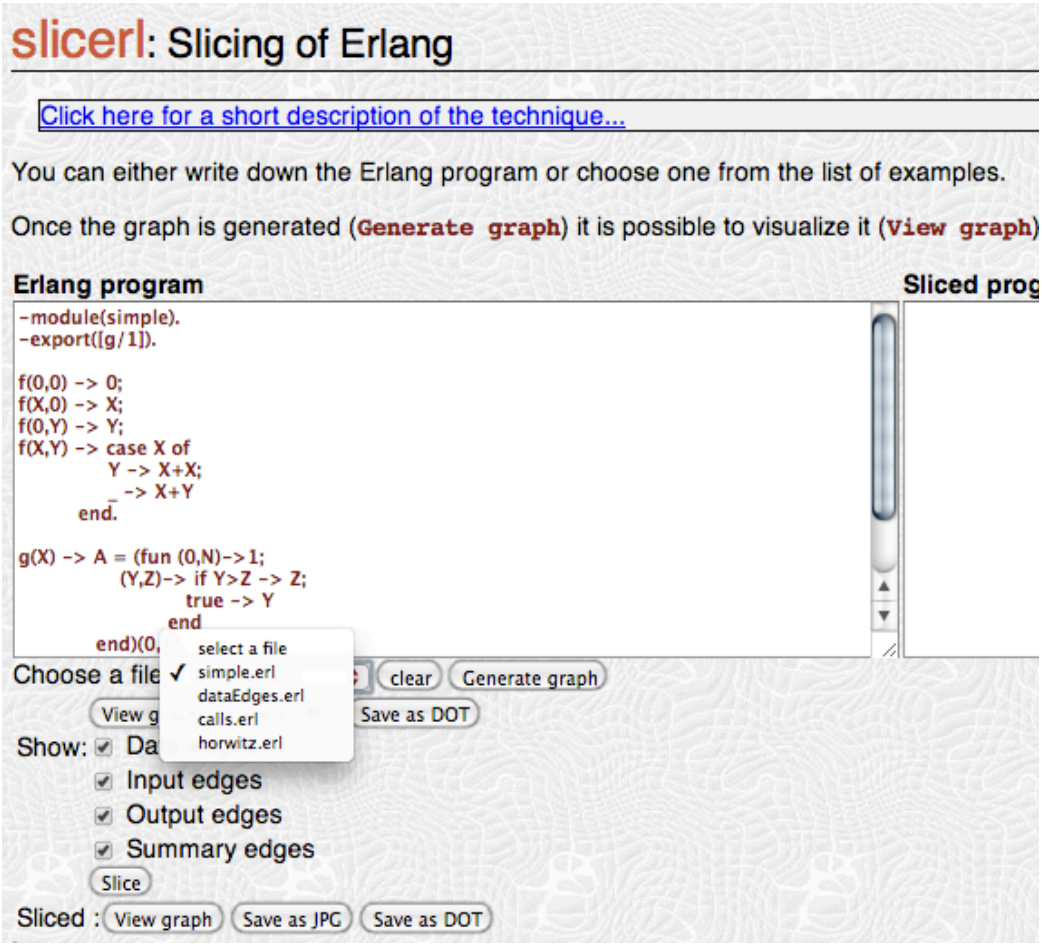


Figura 6.3: *Selección de un programa de ejemplo*

formato DOT pulsando el botón correspondiente. En el código de la Figura 6.5 podemos ver una porción del grafo correspondiente al código del ejemplo `simple.erl` en formato DOT. En este ejemplo podemos ver como en primer lugar se muestra como se representan los nodos en lenguaje de descripción de grafos DOT y posteriormente se muestra como se enlazan dichos nodos mediante arcos.

Después de la generación y la visualización del EDG generado a partir del código introducido, el usuario podrá aplicar la técnica de slicing sobre dicho código. Para ello, el usuario ha de seleccionar el criterio de slicing remarcando con el cursor el slice de código que forma dicho criterio y pulsando después sobre el botón slice. Al igual que ocurrió en la generación del grafo, el sistema muestra un mensaje en el cuadro de texto de visualización del log de los procesos indicando si el proceso se ha llevado a cabo correctamente o si

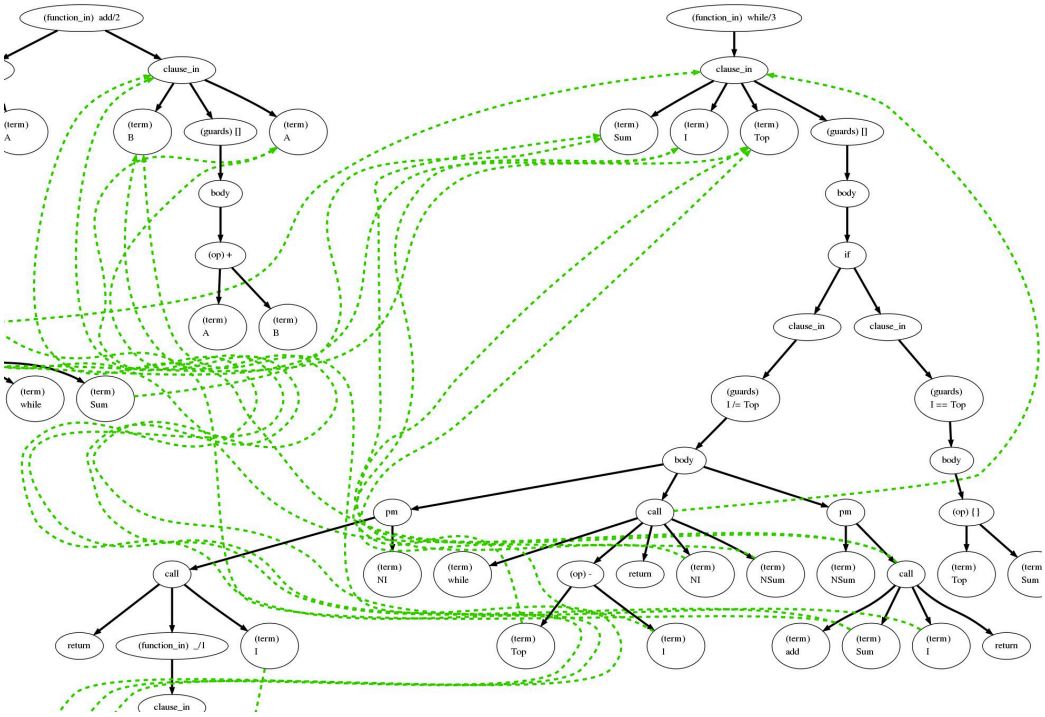


Figura 6.4: *Visualización de un grafo únicamente con la opción de los arcos de entrada marcada*

se ha encontrado algún problema. Por ejemplo, el siguiente texto se muestra cuando el proceso se ha realizado correctamente seleccionando un criterio de slicing válido:

```
Slice from nodes [66]
Process finished on ...
```

En caso de no haber seleccionado un criterio de slicing correcto, el sistema mostrará un mensaje similar a:

```
Selected code is not valid to perform slicing
```

Una vez realizado el proceso de aplicación de la técnica de slicing sobre el código, se puede visualizar los resultados de forma gráfica de la misma forma que se realizó para la visualización del grafo generado para el código inicial.

```
digraph PDG {
    0 [shape=ellipse, label="(function_in)  main/0"];
    1 [shape=ellipse, label="clause_in"];
    2 [shape=ellipse, label="(guards) []"];
    3 [shape=ellipse, label="body"];
    4 [shape=ellipse, label="pm"];
    5 [shape=ellipse, label="(term)  \lSum\l"];
    6 [shape=ellipse, label="(term)  \l0\l"];
    7 [shape=ellipse, label="pm"];
    8 [shape=ellipse, label="(term)  \lI\l"];
    9 [shape=ellipse, label="(term)  \l0\l"];
...
    0 -> 1 [color=black, penwidth=3];
    1 -> 2 [color=black, penwidth=3];
    2 -> 1 [color=red, constraint=false, style="dotted"];
    2 -> 3 [color=black, penwidth=3];
    3 -> 4 [color=black, penwidth=3];
    3 -> 7 [color=black, penwidth=3];
    3 -> 10 [color=black, penwidth=3];
    3 -> 20 [color=black, penwidth=3];
    4 -> 5 [color=black, penwidth=3];
    4 -> 6 [color=black, penwidth=3];
...
}
```

Figura 6.5: *Fragmento del fichero DOT generado para un grafo en slicErl*

Capítulo 7

Caso de Estudio

Para poder analizar realmente la potencia que ofrece la aplicación de a técnica de slicing de programas, en este capítulo se muestra un ejemplo completo de la utilización de la misma sobre un programa concreto así como la evaluación de los resultados obtenidos por la herramienta *slicerl*.

Cabe destacar que por motivos de simplicidad y claridad del caso de uso que se va a detallar, se ha optado por mostrar un ejemplo muy sencillo ya que con ejemplos mas complejos resulta imposible interpretar de forma clara los distintos grafos que se van a utilizar. Como muestra, podemos ver como en la Figura 7.1 se muestra la gran cantidad de nodos y arcos entrelazados que se generan con el programa de la Figura 5.1 mostrado de el Capítulo 5, resultando prácticamente imposible interpretarlo de forma gráfica si no se posee un dispositivo de visualización adecuado.

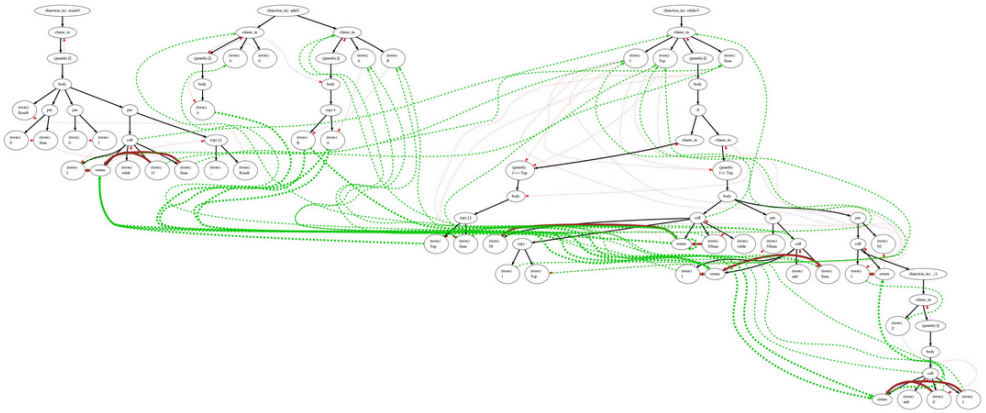


Figura 7.1: *EDG obtenido a partir del código 5.1*

7.1. Programa inicial e introducción en sli-cErl

Para recopilar y mostrar toda la información descrita en los capítulos anteriores hemos decidido realizar el caso de estudio con el código inter-procedural de la Figura 7.3. Este ejemplo muestra todos los tipos de arcos explicados y al no ser extenso permite diferenciar claramente los pasos por los que atraviesa la aplicación de la técnica de slicing.

En la Figura 7.5 podemos ver como el usuario ha introducido el código en el campo de texto habilitado para tal fin en el sitio web de *slicErl* y ha pulsado sobre el botón *Generate Graph*. También puede verse en el cuadro de texto del log que el programa ha reconocido correctamente el programa y ha realizado el proceso de construcción del EDG sin problemas.

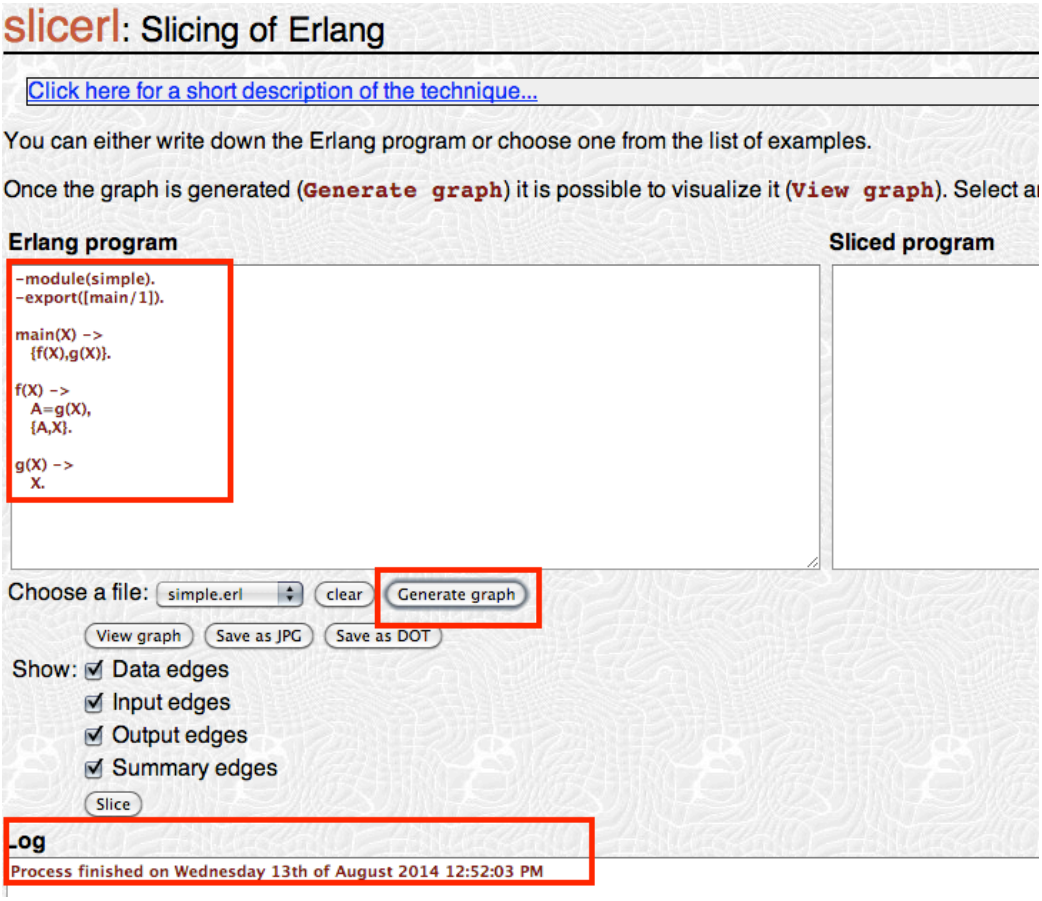


Figura 7.2: *El usuario introduce el código de programa 7.3 y genera el grafo correctamente*

```
main(X) ->
  {f(X),g(X)}.

f(X) ->
  A=g(X),
  {A,X}.

g(X) ->
  X.
```

Figura 7.3: *Código del Caso de Estudio*

```
main(X) ->
  {f(X),undef}.

f(X) ->
  A=g(X),
  {A,undef}.

g(X) ->
  X.
```

Figura 7.4: *Slice del Caso de Estudio obtenido*

7.2. Visualización del EDG

En la Figura 7.5 podemos ver el grafo generado por *slicErl* con el código introducido y después de pulsar sobre el botón *Generate Graph*. Como podemos observar, este ejemplo contiene arcos de todos los tipos explicados en la Sección 4.1 y al no ser un código muy extenso se pueden diferenciar con relativa facilidad dichos arcos.

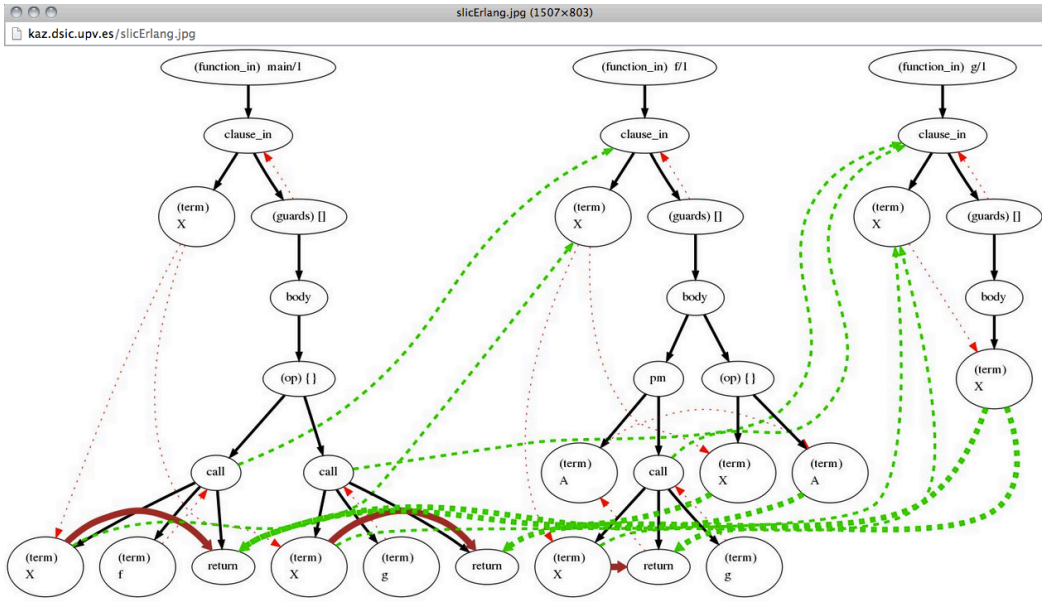


Figura 7.5: *EDG generado por SlicErl a partir del código 7.3*

7.3. Selección del criterio de slicing

Para aplicar la técnica de slicing sobre el caso de estudio expuesto, elegiremos la variable `A` del primer elemento de la tupla de la última instrucción de la función `f` (resaltado en color azul en el código de la Figura 7.3.

Para seleccionar dicho criterio de slicing en *SlicErl*, el usuario únicamente ha de destacar el texto que compone dicho criterio. En la imagen 7.9 podemos ver como el usuario ha marcado la variable `A` en el código.

7.4. Aplicación de la técnica de slicing

Antes de mostrar como el usuario obtiene el slice a partir del criterio de slicing seleccionado en *slicErl* vamos a analizar cuales serían los pasos que atraviesa el Algoritmo 1 con el código del caso de estudio:

1. En la Figura 7.6 vemos destacado el nodo que representa el criterio de slicing en el EDG.

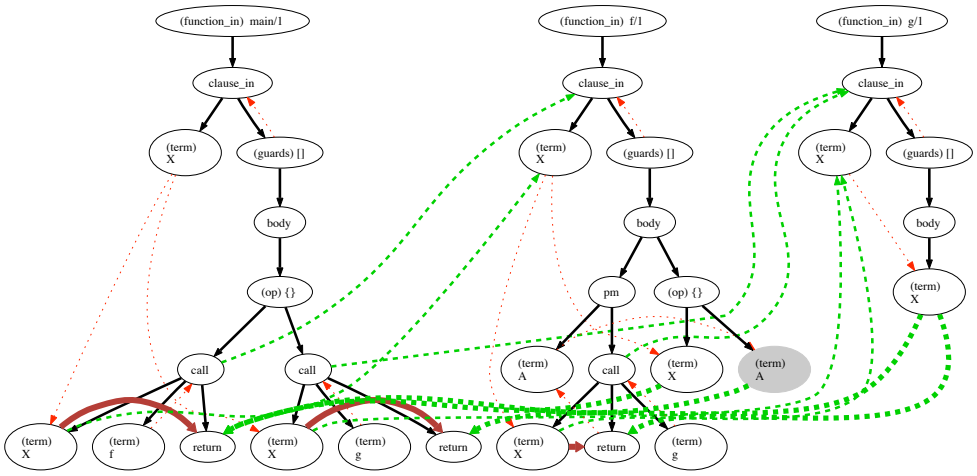


Figura 7.6: *EDG con el criterio de slicing seleccionado*

2. **Primera Fase:** A partir de este nodo seguimos en sentido inverso todos los arcos de Control, Datos, Input y Summary (siempre que se cumpla el caso que se detalló en la explicación del Algoritmo 1). En la Figura 7.7 vemos que se han añadido al slice (nodos oscuros) todos los nodos alcanzados en esta primera fase.

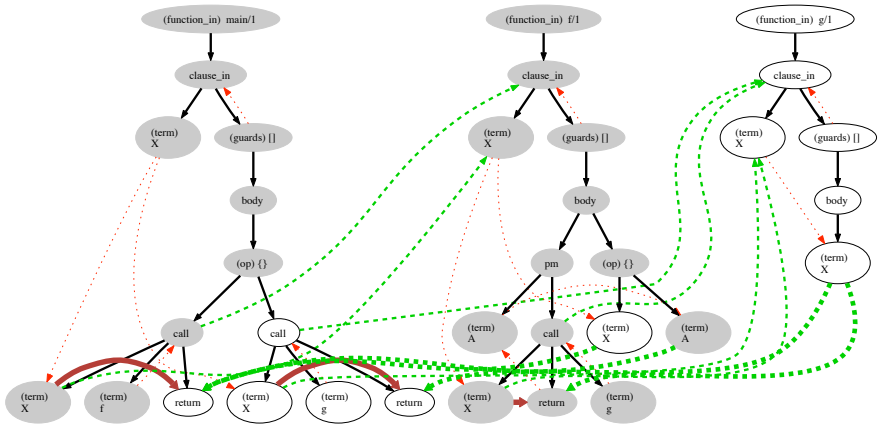


Figura 7.7: *Conjunto de nodos alcanzados a partir del criterio de slicing en la Fase 1 del algoritmo.*

3. **Segunda Fase:** A partir de los nodos alcanzados en la Fase 1 del algoritmo, seguimos en sentido inverso todos los arcos de Control, Datos, Output y Summary (con la misma regla aplicada en la Fase 1). De esta forma, podemos ver en la Figura 7.8 el conjunto de nodos alcanzados en esta segunda Fase, que serán los nodos que compondrán el slice obtenido después de la aplicación de la técnica de slicing

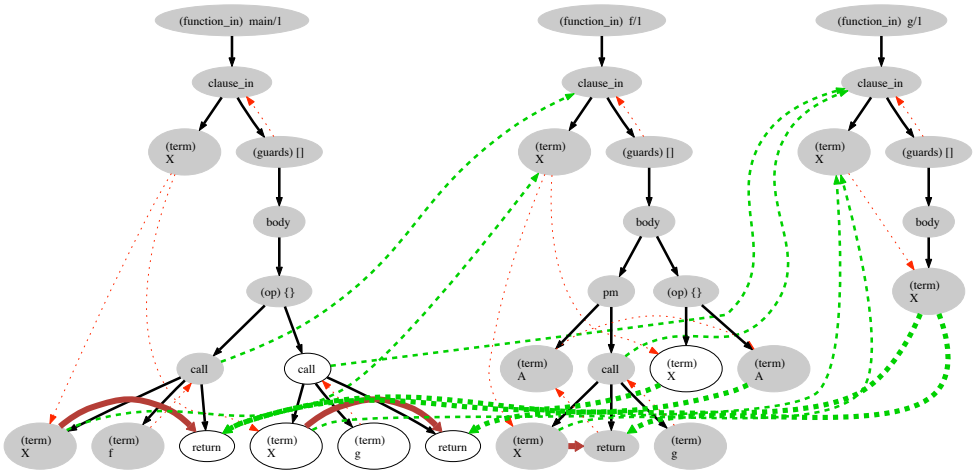


Figura 7.8: *Conjunto de nodos que forman el slice después de la Fase 2 del algoritmo.*

7.5. Obtención e interpretación de los resultados

Una vez seleccionado el criterio de slicing en *slicErl*, el usuario pulsa sobre el botón *Slice*, generándose correctamente el slice, tal y como vemos en la Figura 7.9. Podemos ver como en el cuadro de texto derecho se visualiza el código que conforma el slice obtenido y en el cuadro de teto del Log se indica al usuario que el proceso ha resultado satisfactorio.

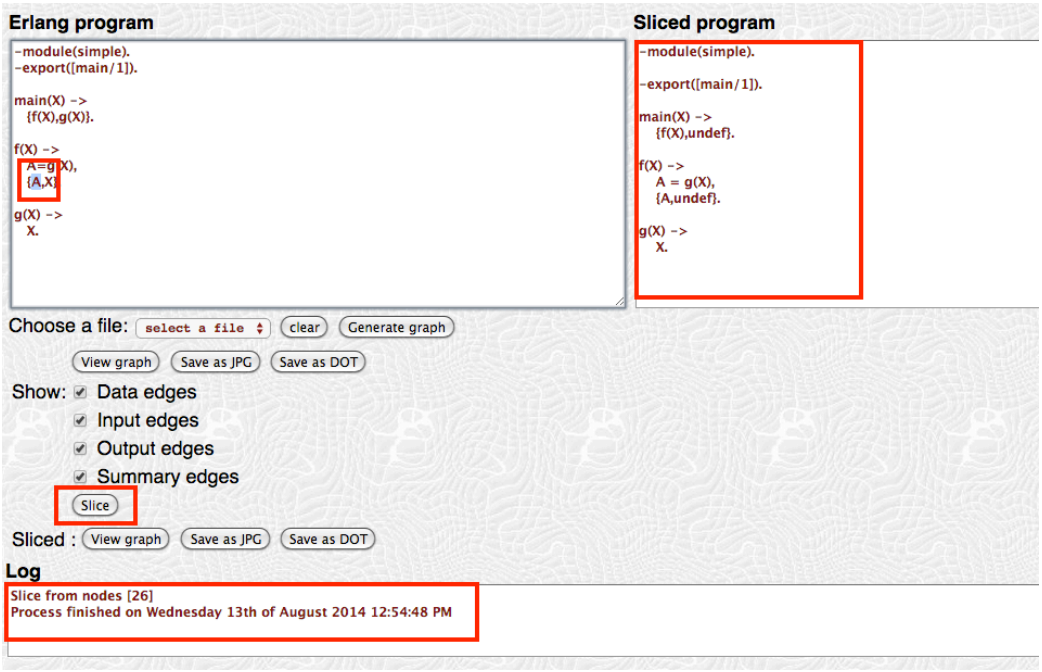


Figura 7.9: *El usuario remarca el criterio de slicing deseado y genera el slice correctamente.*

Cabe destacar que el slice obtenido por *slicErl* coincide con el esperado que mostrábamos en la Figura 7.4. En este código se ha sustituido las dos apariciones de la variable `X` por el átomo `undef` tanto en la llamada a la función `g` de la segunda posición de la tupla de retorno de la función `main`, como en la segunda posición de la tupla de retorno de la función `f`. Esta sustitución nos indica que el valor que contenía esta expresión no afecta al valor del criterio de slicing seleccionado. Si nos paramos a analizar con mayor detenimiento el código podremos comprobar que efectivamente así es y que mediante el uso de *slicErl* podemos aplicar la técnica de slicing a cualquier programa Erlang siempre que se adapte a las restricciones sintácticas expuestas en la Figura 3.1.

Capítulo 8

Conclusiones

En este trabajo se consigue adaptar el SDG para poder ser usado con programas Erlang. Basándose en esta adaptación se define un algoritmo para aplicar la técnica de slicing que produce slices precisos de programas interprocedurales Erlang.

Cabe destacar que esta es la primera adaptación del SDG para un lenguaje funcional que, aunque se ha realizado para el lenguaje Erlang, puede ser fácilmente trasladada a otros lenguajes funcionales con pequeñas modificaciones.

Los slices producidos por la técnica expuesta son completamente ejecutables. Este hecho propicia que esta técnica pueda ser utilizada como fase de preproceso por otros análisis y herramientas, simplificando el programa de entrada y consiguiendo así producir resultados más precisos y/o acelerar el siguiente proceso de transformación.

Se ha implementado un herramienta para aplicar la técnica de slicing a programas Erlang, que también puede generar y visualizar los EDGs correspondientes. Esta herramienta se llama `Slicerl` y está disponible públicamente en:

http://kaz.dsic.upv.es/slicerl

La implementación actual de `Slicerl` acepta más construcciones sintácticas de las que se han descrito en este trabajo. De hecho, `Slicerl` es capaz de producir slices de su propio código.

A pesar de que esta aproximación consigue solucionar la pérdida de precisión interprocedural mediante el uso de los arcos de resumen junto con el algoritmo propuesto, existe otra pérdida de precisión que esta herramienta no es capaz de solucionar y se produce en la expansión y la compresión de estructuras de datos como tuplas o listas.

Ejemplo 7 Considerar el programa Erlang de la izquierda y el criterio de slicing `Y` de la línea `(4)`:

```
(1)  main() ->          (1)  main() ->          (1)  main() ->
(2)    X={1,2},         (2)    X={1,2},         (2)    X={1,_},
(3)    {Y,Z}=X,         (3)    {Y,_}=X,         (3)    {Y,_}=X,
(4)    Y.               (4)    Y.               (4)    Y.
```

El Algoritmo 1 produce el slice que se muestra en el centro, sin embargo no es capaz de producir el slice más preciso que se muestra a la derecha, produciéndose una pérdida de precisión.

La pérdida de precisión mostrada en el Ejemplo 7 se debe al hecho de que el EDG no provee ningún mecanismo para poder seguir el flujo de datos cuando una expresión forma parte de una estructura que se comprime en una variable y posteriormente se expande de nuevo. En este ejemplo existe una dependencia entre la variable `Y` y la variable `X` de la línea `(3)`. Esta dependencia nos indica que "*El valor de* `Y` *depende del valor de* `X`". Desafortunadamente, esto no es totalmente cierto, ya que el significado real es "*El valor de* `Y` *depende de una parte del valor de* `X`". Se está trabajando actualmente para definir una nueva dependencia llamada *dependencia parcial* que solucione este problema.

Bibliografía

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation (PLDI)*, pages 246–256, 1990.

[2] Joe Armstrong. A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[3] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In PeterA. Fritzson, editor, *Automated and Algorithmic Debugging*, volume 749 of *Lecture Notes in Computer Science*, pages 206–222. Springer Berlin Heidelberg, 1993.

[4] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[5] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, January 1985.

[6] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. Refactorerl–source code analysis and refactoring in erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools, Tallin, Estonia*, 2011.

[7] C. Brown. *Tool Support for Refactoring Haskell Programs*. PhD thesis, School of Computing, University of Kent, Canterbury, Kent, UK, 2008.

[8] Diego Cheda, Josep Silva, and Germán Vidal. Static slicing of rewrite systems. *Electron. Notes Theor. Comput. Sci.*, 177:123–136, June 2007.

[9] 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.

[10] M. Harman and S. Danicic. Amorphous program slicing. In *Program Comprehension, 1997. IWPC '97. Proceedings., Fifth Iternational Workshop on*, pages 70–79, Mar 1997.

[11] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, July 1989.

[12] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Programming Languages and Systems*, 12(1):26–60, 1990.

[13] Manuel Angel Rubio Jimenez. *Erlang/OTP Volumen I: Un Mundo Concurrente*. Creative Commons Atribucion, 2013.

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

[15] Tobias Lindahl and Konstantinos F. Sagonas. Typer: a type annotator of erlang code. In Konstantinos F. Sagonas and Joe Armstrong, editors, *Erlang Workshop*, pages 17–25. ACM, 2005.

[16] Tobias Lindahl and Konstantinos F. Sagonas. Practical type inference based on success typings. In Annalisa Bossi and Michael J. Maher, editors, *PPDP*, pages 167–178. ACM, 2006.

[17] P.E. Livadas and S.D. Alden. A toolset for program understanding. In *Program Comprehension, 1993. Proceedings., IEEE Second Workshop on*, pages 110–118, Jul 1993.

[18] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, April 1984.

[19] Nuno F. Rodrigues and Luï¿½is S. Barbosa. Component identification through program slicing. In *In Proc. of Formal Aspects of Component Software (FACS 2005). Elsevier ENTCS*, pages 291–304. Elsevier, 2005.

[20] Josep Silva. A vocabulary of program slicing-based techniques. *ACM Computing Surveys*, 44(3), 2012.

[21] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[22] Melinda Toth and Istvan Bozo. Building dependency graph for slicing erlang programs. *7th Conference of PhD Students in Computer Science*, 2010.

[23] M. Weiser. Program Slicing. In *Proceedings of the 5th international conference on software engineering*, pages 439–449. IEEE Press, 1981.

[24] Mark Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, July 1984.

[25] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[26] Manfred Widera. Flow graphs for testing sequential erlang programs. In *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, ERLANG '04, pages 48–53, New York, NY, USA, 2004. ACM.

[27] Manfred Widera and Fachbereich Informatik. Concurrent erlang flow graphs. In *In Proceedings of the Erlang/OTP User Conference 2005*, 2005.