Sistema de identificación y explotación de paralelismo en programas lógico-funcionales

Memoria que presenta para optar al grado de Doctor en Ciencias Físicas

Fernando Sáenz Pérez

Dirigida por el profesor

José J. Ruz Ortiz

Departamento de Informática y Automática
Facultad de Ciencias Físicas
Universidad Complutense de Madrid

Septiembre 1995

A mi familia...

Agradecimientos

Muchas veces me detengo a pensar lo afortunado que soy con las personas que me rodean y que me ayudan en muchos sentidos a cumplir tareas como ésta, una tesis fruto de varios años de trabajo. No puedo por menos nombrar en primer lugar a Pepe, que ha sido con su esfuerzo, apoyo y continuos consejos mi guía durante todo este tiempo. Fue a causa de él por lo que decidí abandonar un puesto fijo en la televisión pública y dar un giro a mi vida para dedicarme al entorno académico, de lo cual estoy seguro que jamás me arrepentiré. Reconozco y aprecio sinceramente su valiosa influencia, de la que espero quede constancia en este trabajo.

La colaboración con el Lehrstuhl für Informatik II empezó *simplemente* como tal, pero acabó dejando en Alemania muchos amigos que han hecho con su ayuda prosperar mi trabajo y hacerme sentir allí como en mi propia casa. Hablo por ejemplo de Klaus, que me ayudó mucho y me trató como a un amigo más, y de su mujer Rosalía (una madrileña encantadora) de los cuales guardo el más grato recuerdo. También de Herbert Kuchen, que me acogió en su casa con su familia cuando no pude disponer de alojamiento y me dio a conocer Alemania. Sé que debí ser una carga para Rita Loogen cuando, en plena preparación de su habilitación, me recibió, atendió y dispuso todo lo necesario para que pudiese instalarme y empezar a trabajar en el instituto. Y cómo no recordar a mis amigos Stephan, Werner y la revoltosa Katia, compañeros de trabajo en el instituto y de encuentros en la ciudad: los partidos de tenis, los pubs, las comidas, los viajes, ... Recuerdo el agradable ambiente que se respiraba en el instituto, con las continuas celebraciones por aniversarios, bodas y los más peregrinos motivos. Agradezco, en suma, la hospitalidad que me brindaron todos y cada uno de los miembros del instituto de Aquisgrán durante el medio año que disfruté junto a ellos. Pero no sólo a ellos, sino a toda la gente que conocí en Alemania y me ayudó cuando tuve problemas. Por cierto, quiero poner una pica en Flandes para romper el tópico de la frialdad alemana, no es como la pintan.

Aquí en Madrid, mis amigos los *politécnicos* han estado soportándome por todas mis dudas acerca de su trabajo, facilitándome continuamente el andar en el mundo horrible de las estaciones de trabajo. Me refiero a Paco, con el que he mantenido muchas charlas inspiradoras y con el que he compartido muchas diversiones. También a María, que me ha ayudado a comprender temas claves y ha sufrido la lectura de algún *papel* que le pasé. Y también a Manuel, lástima que disponga siempre de tan poco tiempo. En general me refiero al grupo CLIP, siempre dispuesto a algún cine y a echarme un cable.

Creo que si Mario no nos hubiese *dado la lata* con Babel, ahora esta tesis estaría hablando de otra cosa totalmente diferente, por lo cual quiero dejar constancia no sólo de su influencia en este trabajo, sino también por el ánimo que inspira hacia la investigación *divertida*.

Lourdes me ha estado aclarando con su trabajo las consultas que le hacía vía Internet acerca de su sistema paralelo. Aquí Antonio también tiene un sitio especial, con su capacidad para encontrar mis múltiple fallos en el uso escrito del inglés. Y para el inglés hablado, las pacientes Ana y Marisa estuvieron soportando algunas de las peroratas que había que soltar en los congresos internacionales para *limar* mi pronunciación. Agradezco también a Kike su sinceridad cuando me hacía ver que la estructura de un artículo no tenía ni pies ni cabeza y los ratos agradables que hemos pasado juntos en empresas emocionantes. Y cómo no hablar de Boni, adalid de causas perdidas, siempre preocupado por nosotros (con su carga sarcástica e irónica), los pobres becarios y desvalidos.

Aunque debo reconocer que se debe tener vocación para dedicarse a la labor investigadora en España, no debo por menos agradecer la oportunidad que brinda el MEC a través de sus becas para la formación de investigadores.

Sin embargo, a quienes más debo es a mis padres, porque me han apoyado e incentivado a hacer mi camino durante todos estos años.

Espero que este trabajo sirva como una pequeña y sincera muestra de

mi agradecimiento a todos ellos.

Fernando Sáenz Pérez
Facultad de Ciencias Físicas, UCM [1]
Madrid, 23:45 del 24 Junio de 1995

[1] Sí, ya sé que es Sábado, un poco tarde y que debería irme a la cama o de juerga, pero esto también es divertido. Por cierto, creo que he olvidado agradecer a Carolina que, sin su aportación, esta tesis quizás se hubiese terminado antes.

Contenido

Prólogo xviii
Resumen del trabajo xix
Organización del trabajo xx

1 Introducción 1
1.1 Lenguajes de programación declarativa 1
1.1.1 La programación funcional 2
1.1.2 La programación lógica 6
1.1.3 La integración de la programación lógica y la programación funcional 10
1.2 Paralelismo en lenguajes declarativos 13
1.2.1 Paralelismo conjuntivo 15
1.3 Implementaciones de lenguajes lógico-funcionales 17
1.4 Mejora de rendimiento 19
1.5 Propósito del trabajo 19
Sumario 21

I Identificación de paralelismo 23

2 Paralelización de Babel 25
2.1 Paralelismo conjuntivo independiente en Babel 25
2.1.1 Modelo de evaluación paralela 27
2.1.2 Expresión de dependencias: el grafo de dependencias condicionales (*CDG*) 31
2.1.3 Transformaciones del *CDG* 35
2.2 Obtención de reglas Babel paralelas 44
2.2.1 Reglas de transformación 45
2.2.2 Estrategias de transformación 48

2.2.3 Transformación de reglas secuenciales Babel en paralelas 56
2.3 Medida de rendimiento del sistema de paralelización automática 58
2.3.1 Simulador 59
2.3.2 Resultados 64
Sumario 68

3 Análisis de independencia **71**
3.1 Interpretación abstracta 71
3.1.1 Interpretación abstracta de programas funcionales . . 75
3.1.2 Interpretación abstracta de programas lógicos 76
3.2 Interpretación abstracta de lenguajes lógico-funcionales . . . 76
3.2.1 Validación de la extensión 86
3.2.2 Tratamiento de las funciones no estrictas 89
3.3 Niveles de análisis 90
3.3.1 Dominio abstracto $\mathcal{D}_1$ 90
3.3.2 Dominio abstracto $\mathcal{D}_2$ 93
3.3.3 Dominio abstracto $\mathcal{D}_3$ 96
3.4 Resultados 100
Sumario 102

II Explotación de paralelismo **105**

4 La máquina abstracta paralela PEBAM **107**
4.1 Introducción 107
4.1.1 Máquinas abstractas secuenciales basadas en pilas . . 108
4.1.2 Extensión de máquinas abstractas secuenciales basadas en pilas a paralelas 111
4.2 Modelo de ejecución de la máquina abstracta paralela 113
4.2.1 Cómputo hacia adelante 113
4.2.2 Cómputo hacia atrás (*backtracking*) 114
4.2.3 Planificación de trabajo 114
4.2.4 Descarte paralelo de cómputos 119
4.3 Arquitectura de la PEBAM 124
4.3.1 Áreas y estructuras de datos 125
4.3.2 El repertorio de instrucciones 131
4.4 La compilación 135
4.4.1 Compilación del programa (*progtrans*) 136
4.4.2 Compilación de funciones (*functrans*) 136

4.4.3 Compilación de reglas (*ruletrans*) 137
4.4.4 Compilación de expresiones (*exprtrans*) 137
4.4.5 Compilación de la unificación (*unifytrans*) 143
4.4.6 Compilación de la igualdad (*equality_prelude*) 143
4.4.7 Compilación de las *PEUs* (*partrans* y *strictpartrans*) . 144
4.4.8 Compilación del guarda (*guardtrans*) 144
4.4.9 Compilación de primitivas aritméticas y de comparación (*ho_prelude*) 145
4.4.10 Compilación de la impresión (*print_prelude*) 145
4.4.11 Ejemplo de compilación 146
Sumario 150

5 Implementación y resultados **153**
5.1 Requerimientos de la máquina PEBAM 153
5.2 Sistema de ejecución paralela 154
5.2.1 Bus del sistema 157
5.2.2 Elementos de proceso 158
5.2.3 Memoria compartida 159
5.2.4 Protocolos de coherencia caché 160
5.3 Simulación del sistema multiprocesador 162
5.3.1 El lenguaje de descripción *hardware* VHDL 163
5.3.2 Simulación VHDL del sistema multiprocesador 164
5.3.3 Medidas de rendimiento 171
Sumario 181
Conclusiones 185
Principales aportaciones 186
Sugerencias para trabajos futuros 187

Conclusiones **184**

A Sintaxis y semántica de Babel **189**
A.1 Sintaxis de Babel 189
A.2 Semántica operacional de Babel 192

B Programas de prueba **195**
B.1 Sintaxis 195
B.2 Listado de los programas de prueba 196

C Especificación formal de S **227**
C.1 Esquema S^* 231

D Especificación de la máquina abstracta paralela **233**
D.1 El lenguaje de especificación 233
D.2 Operaciones relacionadas con la explotación de paralelismo . 234
D.2.1 Consecuencia calculada con éxito 235
D.2.2 Fallo 238
D.2.3 Procedimiento de descarte (*kill*) 245
D.2.4 Planificación de trabajo 248
D.2.5 Operaciones misceláneas 251

Publicaciones **254**

Índice de tablas

2.1 Medidas resultado de la simulación de los programas de prueba. I parte. 66
2.2 Medidas resultado de la simulación de los programas de prueba. II parte. 67

3.1 Resultados del análisis de los programas de prueba. 101

5.1 Comparación de las estrategias de paralelización. I parte. . . 177
5.2 Comparación de las estrategias de paralelización. II parte. . . 178

Índice de figuras

0.1 Visión de conjunto. xxi

1.1 Circuito combinacional para un sumador binario completo. . 5
1.2 Paralelismo de argumentos en un lenguaje funcional impaciente. 15

2.1 *CDG*. 34
2.2 Distinción de casos de la función *slink*. 43
2.3 Orden parcial de pesos asignados a la evaluación de condiciones. 44
2.4 *ECDG* correspondiente a la regla $h(X,Y) := u(a(X), b(Y), c, d(X,Y))$. 52

3.1 Dominios estándar y abstracto de los enteros. 74
3.2 Interpretaciones estándar y abstracta. 74
3.3 Relación entre los conjuntos $\mathcal{Z}_\perp$ y $\mathcal{Z}^*$. 75
3.4 Corrección de una interpretación abstracta. 77
3.5 Nodos del árbol abstracto de evaluación. 79
3.6 Tabla de memoria. 81
3.7 Comparación entre los diferentes algoritmos. 83
3.8 Árbol de evaluación abstracta de f. 84
3.9 Subárboles para los casos lógico y lógico-funcional. 88

4.1 Ejemplo de gestión de la pila. 110
4.2 Problema de la explotación de paralelismo. 112
4.3 Máquina abstracta paralela. 113
4.4 Árboles de evaluación para los casos secuencial y paralelo. . . 116
4.5 Condiciones en la planificación de trabajo. 117
4.6 Cola de tareas imbuida en la pila. 120
4.7 Áreas de datos de un trabajador. 126
4.8 Áreas y estructuras de datos de un trabajador. 127
4.9 Estructuras de datos y sus contenidos. 128

5.1 Referencias entre las áreas de datos de los trabajadores. . . . 155

5.2 Áreas de datos. 156
5.3 Formato de las direcciones. 156
5.4 Sistema multiprocesador de memoria compartida. 156
5.5 Bus con arbitraje por sondeo. 157
5.6 Bus con arbitraje por petición independiente. 158
5.7 Elemento de proceso. 159
5.8 Memoria caché. 159
5.9 Módulo de memoria compartida. 160
5.10 Diagrama de transición de estados del protocolo de Goodman. 161
5.11 Diagrama de transición de estados del protocolo Firefly. . . . 162
5.12 Entidad y arquitectura del controlador del bus. 166
5.13 Entidad y arquitectura del elemento de proceso. 168
5.14 Entidad y arquitectura del módulo de memoria compartida. . 169
5.15 Porcentaje de aciertos en memoria caché. 173
5.16 Tasa de retardo debido al uso compartido del bus. 174
5.17 Implementación de sondeo con protocolo de coherencia Goodman. 175
5.18 Protocolo Goodman *vs.* Firefly. 176
5.19 Efecto de la política de planificación en la carga de trabajo. . 180
5.20 Efecto de la condición de precedencia en la planificación. . . . 182

Prólogo

Los lenguajes declarativos en general y los lógicos y funcionales en particular disponen de una base matemática bien fundada y son referencialmente transparentes. Ello permite el desarrollo de programas concisos y el uso de métodos formales para asistir al diseño, mantenimiento y su análisis para la mejora de rendimiento, con mayor capacidad expresiva que los imperativos pero más ineficientes en tiempo de cómputo y uso de memoria. Puesto que los programas conservan con mayor integridad el paralelismo de las aplicaciones, el análisis de independencia permite detectar las expresiones que se pueden evaluar simultáneamente en un sistema de ejecución paralela.

El objetivo de este trabajo es el aumento de la eficiencia de los lenguajes lógico-funcionales mediante la identificación automática de paralelismo y su explotación, reteniendo las optimizaciones conseguidas en los sistemas secuenciales.

Resumen del trabajo

En primer lugar se desarrolla un procedimiento para la extracción de paralelismo en *programas l'ogico-funcionales* secuenciales a partir de los grafos de dependencias condicionales de cada regla del programa, bajo el *modelo de ejecuci'on paralela* que describimos. Un grafo expresa las dependencias locales de evaluación entre las expresiones de una regla. Se presentan tres estrategias para la generación de *reglas paralelas* a partir de los grafos. Cada estrategia establece un compromiso entre dos factores: paralelismo identificado y coste de su identificación en tiempo de ejecución. Además, se presenta un procedimiento para la incorporación de información de granularidad de manera que se evite la evaluación paralela de expresiones de baja carga computacional. El proceso de paralelizaci'on admite la incorporación de información global de independencia para simplificar las reglas paralelas.

En segundo lugar se desarrolla un *análisis de independencia* de los pro-

gramas para obtener esta informaci'on de independencia. Para ello se utiliza la interpretaci'on abstracta extendiendo los resultados obtenidos en programaci'on l'ogica. Se estudian tres niveles de análisis con distintos grados de informaci'on y se comparan en términos de la informaci'on útil inferida y el tiempo necesario para obtenerla.

En tercer lugar se diseña una *máquina abstracta paralela* de memoria compartida basada en pilas para la explotaci'on del paralelismo identificado. Esta máquina retiene las optimizaciones de las máquinas secuenciales, fundamentalmente durante el c'omputo hacia atrás en la desasignaci'on de memoria. Su comportamiento es similar al de una máquina funcional cuando se determinan c'omputos deterministas estáticamente, mediante análisis, o dinámicamente, mediante el corte dinámico. Se describen las áreas de datos, el repertorio de instrucciones y la compilaci'on de programas paralelos Babel a instrucciones máquina.

Finalmente, se realiza una implementación de la máquina abstracta paralela sobre un *sistema multiprocesador simulado de memoria compartida.* Se estudian los componentes del sistema y se plantean diferentes alternativas de diseño, que afectan al protocolo de arbitraje de bus y a la política de coherencia de la memoria caché. Se utiliza el lenguaje de descripción *hardware* VHDL para validar el comportamiento funcional de la máquina y para obtener medidas de rendimiento del sistema.

En la figura 0.1 se muestra una síntesis del desarrollo de este trabajo.

Organización del trabajo

En la memoria se introducen en el capítulo 1 los lenguajes declarativos, los tipos de paralelismo que presentan, sus implementaciones y su mejora de rendimiento.

La primera parte del trabajo comienza en el capítulo 2, en el que se desarrolla el procedimiento de extracción de paralelismo en programas lógico-funcionales secuenciales.

En el capítulo 3 se describe el procedimiento de análisis para la simplificación de las reglas paralelas.

La segunda parte comienza en el capítulo 4, en el que se presenta la máquina abstracta paralela, su repertorio de instrucciones y la compilación.

En el capítulo 5 se desarrolla la implementación de la máquina sobre una simulación de un sistema multiprocesador.

Finalmente, en el capítulo 5.3.3 se resumen las conclusiones del trabajo

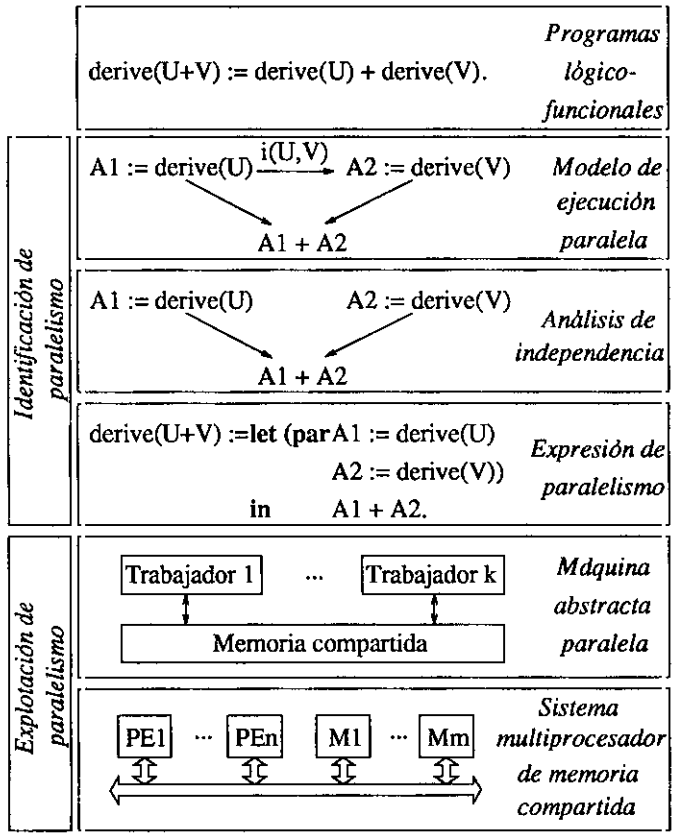


Figura 0.1: Visión de conjunto.

manifestando las aportaciones principales y el sentido en que se pueden encauzar futuros trabajos.

Capítulo 1

Introducción

En este primer capítulo se introducen los tipos más representativos de programación declarativa: funcional, lógica y lógico-funcional. Se ponen de manifiesto los diferentes tipos de paralelismo que presentan estos lenguajes y se estudian las técnicas de implementación que han permitido conseguir realizaciones eficientes. Asimismo se presenta la forma de mejorar el rendimiento secuencial y paralelo de las implementaciones incorporando en tiempo de compilación información global obtenida por análisis de programas.

1.1 Lenguajes de programación declarativa

Con los lenguajes declarativos se pretende que el programador se ocupe más de la especificación de su aplicación que del diseño del algoritmo que la implementa. Con un lenguaje imperativo se especifica explícitamente el flujo de control necesario para resolver un problema. Con un lenguaje declarativo tan sólo se declara el problema, siendo el sistema el encargado de aplicar un mecanismo general de resolución. A esta ventaja de los lenguajes declarativos se pueden añadir las siguientes:

- *Concisión.*
  Los programas declarativos son más concisos que sus homólogos imperativos, esto es, proporcionan una expresividad mayor que los lenguajes imperativos a un nivel de complejidad equivalente [117].

- *Análisis.*
  Facilitan su análisis gracias a la *transparencia referencial* que los caracteriza, es decir, al hecho de que el significado de un trozo de programa dependa sólo del significado de sus partes y no de la historia de la

computación realizada hasta el momento de su evaluación[1]. Por ello se les pueden aplicar más fácilmente técnicas de transformación para mejorar su rendimiento.

- *Paralelismo.*
  La mayor distancia semántica de los lenguajes lógico-funcionales de la arquitectura de von Neumann [158] en comparación con los lenguajes imperativos se puede compensar aprovechando su mayor grado de paralelismo. En los lenguajes declarativos, al debilitarse el concepto de secuencialidad, el paralelismo de la aplicación se conserva con mayor integridad en el programa, dejando abierta la posibilidad de identificarlo automáticamente para traducirlo en mejora de rendimiento[2].

1.1.1 La programación funcional

Históricamente, la programación funcional [81, 59, 67] es anterior a la programación lógica. Sus fundamentos son el λ-cálculo [32], la lógica ecuacional [84] y la reescritura [53]. La forma en que se construye un programa funcional es análoga a la especificación jerárquica que se realiza en ingeniería de la programación. El elemento básico de programación es la función, que hereda las propiedades de las funciones matemáticas. Lisp [109], Haskell [82], Miranda [154] y ML [65] son algunos de los lenguajes de programación funcional más conocidos. En [10] y [84] se pueden encontrar los fundamentos de la programación funcional.

Se pueden destacar las siguientes características de los lenguajes funcionales:

- *Aplicaciones parciales de funciones.*
  Esto es, la posibilidad de aplicar una función a un número menor o igual de argumentos que su aridad. Esto permite, por ejemplo, tener datos parcialmente especificados[3] o funciones parciales que se pasen como argumentos a otras funciones[4].

[1] La ausencia de transparencia referencial en los lenguajes imperativos se debe a la sentencia de asignación, que permite que a una misma variable se le puedan asignar valores diferentes en más de un punto del programa. El resultado es que el significado de una variable se hace dependiente de la posición, dificultando la manipulación y transformación del programa.

[2] Por contra, la naturaleza secuencial de los lenguajes imperativos obliga a *destruir* componentes importantes del paralelismo natural de la aplicación.

[3] Un dato se considera como una función que se evalúa a sí misma.

[4] Para expresar las aplicaciones parciales se sigue una notación currificada [42].

- *Orden superior.*
  Las funciones pueden tener como argumentos otras funciones o devolver como resultado una función, lo que permite una mayor expresividad del lenguaje, puesto que se pueden especificar funciones más genéricas.

- *Mecanismos de evaluación.*
  Los lenguajes de programación funcional usan como mecanismo de cómputo la reescritura de términos [83, 10]. La reescritura de un término consiste en la sustitución de sus subtérminos utilizando las reglas de definición de funciones predefinidas o definidas por el usuario. El mecanismo por el que se comprueba que un término puede reescribirse con una regla se denomina *encaje de patrones.* En esta operación se comprueba la igualdad sintáctica de los parámetros formales con los argumentos de la regla, haciendo básicas las variables de la parte izquierda de una regla, esto es, asignándoles valores que cumplan la igualdad. La reescritura de términos es determinista puesto que se calculan expresiones que son formas normales y, por lo tanto, únicas.

  La evaluación impaciente[5] y la perezosa son los mecanismos principales de evaluación. Según el primero se realiza la evaluación de los argumentos antes de la aplicación de la función. Con el segundo se retrasa la evaluación de las expresiones hasta que sean demandadas, permitiendo el manejo de objetos potencialmente infinitos[6].

  Los mecanismos de evaluación perezosa permiten cómputos terminantes en programas más expresivos en cuanto que admiten datos potencialmente infinitos. Por el contrario, las estrategias impacientes encuentran en muchos casos las soluciones *prácticas* con la ventaja de poder ser implementadas más eficientemente.

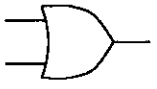
En el siguiente ejemplo mostramos un programa Hope que describe el funcionamiento del circuito combinacional que implementa al sumador completo de la figura 1.1. En este ejemplo se observa la declaración de tipos de datos enumerados (**data**), la declaración de las funciones y sus tipos (**dec**) y la declaración de las reglas de definición de función (que comienzan con - - -). En la declaración de los tipos de datos enumerados, el símbolo ++ separa las constantes. En la declaración de los tipos de funciones, el tipo

[5]Llamada también *voraz* por algunos autores.

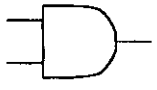
[6]Se seleccionan primero los términos más exteriores (*outer* y *outermost*) [59, 64].

de los argumentos se separa con el símbolo #, mientras que el símbolo → precede al tipo del resultado. En la declaración de funciones el símbolo <= separa la cabeza del cuerpo de la función, que termina en ;. Los comentarios van precedidos por el símbolo !.

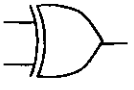
```
data  level == 0 ++ 1 ;

! Puerta Or
dec or : level # level → level ;
--- or(0,0) <= 0;
--- or(0,1) <= 1;
--- or(1,0) <= 1;
--- or(1,1) <= 1;
```



```
! Puerta And
dec and : level # level → level ;
--- and(0,0) <= 0;
--- and(0,1) <= 0;
--- and(1,0) <= 0;
--- and(1,1) <= 1;
```



```
! Puerta Or exclusiva
dec xor : level # level → level ;
--- xor(0,0) <= 0;
--- xor(0,1) <= 1;
--- xor(1,0) <= 1;
--- xor(1,1) <= 0;
```



```
! Sumador completo de un bit
dec adder : level # level # level → level # level;
--- adder(I1,I2,Cin) <= (xor (xor (I1, I2), Cin),
                         or (and (Cin, xor (I1, I2)), and (I1, I2)) ) ;
```

En este ejemplo se muestra la capacidad expresiva de un lenguaje funcional para especificar circuitos combinacionales. En él se utiliza la aplicación de funciones para implementar de manera natural el circuito combinacional.

El programa anterior se puede consultar con una expresión como $adder(1,0,1)$, que da como resultado el par $(0,1)$. El primer componente de este par es el resultado de la reescritura de la expresión $xor(xor(1,0),1)$, donde los argumentos $I1$, $I2$ y Cin se han sustituido respectivamente por 1, 0 y 1. La

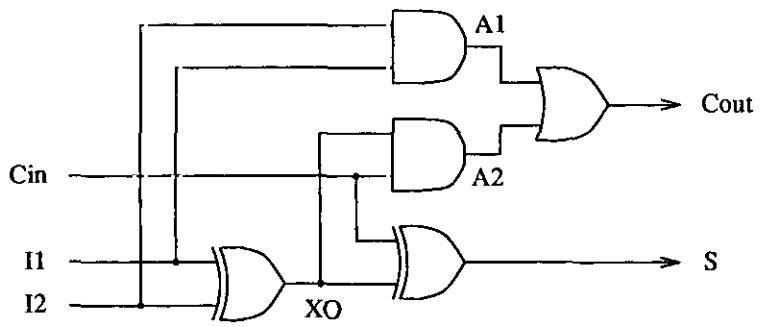


Figura 1.1: Circuito combinacional para un sumador binario completo.

subexpresión $xor(1,0)$ se reescribe con el lado derecho de la tercera regla de la función xor, i.e., 1, con lo cual la expresión queda reescrita como $xor(1,1)$. Esta expresión se reescribe a 0 con la aplicación de la cuarta regla, quedando calculado el primer componente del resultado. El segundo componente se calcula de forma similar con resultado 1.

La concatenación de listas, clásico ejemplo de la programación declarativa, se escribe en Hope:

```
dec   append: list(alpha) # list(alpha) → list(alpha)
- - -  append(nil, l) <= l;
- - -  append(x::y::nil, l) <= x::append(y,l);
```

El símbolo :: se usa como constructor de listas, siendo *nil* la lista vacía. En este ejemplo se declara que tanto los argumentos como el resultado de la función son de tipo lista. Los componentes de la lista se declaran como polimórficos con la variable de tipo *alpha*. Esto es, la función *append* acepta listas de cualquier tipo de datos, si bien todas las componentes deben ser del mismo tipo.

Es posible consultar este programa con la siguiente expresión:

append("Programación ","funcional")

que se reduce al término "Programación funcional".

En el siguiente programa, se muestra la función de orden superior *map*:

```
dec   map: (alpha → beta) # list(alpha) → list(beta)
- - -  map(f, nil) <= nil;
- - -  map(f, x::l) <= f(x)::map(f,l);
```

En este ejemplo, el tipo del primer argumento se declara como una función cuyo primer argumento es de tipo *alpha* y el resultado de tipo *beta*.

Se puede observar la potencia expresiva del orden superior planteando la siguiente expresión, que combina llamadas a las dos funciones anteriores.

map(append(a::nil), (b::nil)::(b::c::nil)::nil)

Su evaluación genera una lista de listas que comienzan con el término *a*. Durante la evaluación se realiza la aplicación *append(a::nil)* a cada uno de los argumentos de la lista *(b::nil)::(b::c::nil)::nil*, generándose la lista $(a :: b :: nil) :: (a :: b :: c :: nil) :: nil$.

Por último, para mostrar la utilidad de la evaluación perezosa, presentamos el siguiente ejemplo. En él se define la función *from*, que calcula la lista infinita de enteros a partir de uno dado, y la función *sum*, que suma los n primeros números de una lista de enteros.

```
dec   from: num → list(num)
- - -  from(n) <= n::from(n+1);

dec   sum: num # list(num) → num
- - -  sum(n,x::l) <= if n=0 then 0 else x + sum(n-1,l);
```

Puesto que bajo este mecanismo de evaluación sólo se demanda la reducción de las expresiones necesarias, es posible plantear la siguiente expresión:

sum(3,from(1))

cuya evaluación da como resultado 6, objetivo que no se puede alcanzar en un esquema de evaluación impaciente.

1.1.2 La programación lógica

Al igual que la programación funcional, la programación lógica [91, 149] también dispone de firmes fundamentos matemáticos. Está basada en la lógica de predicados de primer orden (cláusulas de Horn).

La programación lógica nació de la confluencia de dos líneas de investigación: una inglesa, encabezada por Kowalski [91], centrada en demostración automática de teoremas y otra francesa, encabezada por Colmerauer [37], dedicada al procesamiento del lenguaje natural. De ambos procede la idea de que *la lógica puede utilizarse como un lenguaje de programación.* A ello hay que añadir el trabajo fundamental de Robinson [135] sobre el principio de resolución SLD: una regla de inferencia con una operación de encaje de patrones denominada *unificación* [133]. Los trabajos de Kowalski llevaron

al diseño de un sistema de inferencia basado en el principio de resolución. El grupo de Colmerauer propuso el primer intérprete práctico de un lenguaje de programación lógica, Prolog (acrónimo de *PROgramming in LOGic*). Este es el lenguaje de programación lógica que ha prevalecido y que se considera como su representante más importante. Los fundamentos teóricos de este estilo de programación se pueden encontrar en [99].

Se pueden citar las siguientes características de la programación lógica.

- Unificación.
  La unificación es una operación que integra varias operaciones de la programación imperativa: el paso de parámetros en llamada a procedimientos, la asignación, la comparación y la bifurcación. El paso de parámetros se realiza mediante la unificación de los argumentos de un literal con los de la cabeza de una cláusula, la asignación mediante la unificación de *variables lógicas* y términos, la comparación mediante unificación de constantes o funtores y, finalmente, la bifurcación mediante el proceso de acierto o fallo en la comparación.

- Manejo de términos parcialmente construidos.
  Son términos que contienen variables lógicas sin vincular. Esta característica permite manejar términos generales que se pueden especializar durante el cómputo mediante la unificación. Esto permite implementar la concatenación de listas y la inserción en colas como operaciones de tiempo constante [34, 35]. También es posible resolver problemas de referencias adelantadas en una sola pasada, como la gestión de tablas de símbolos de compiladores.

- Bidireccionalidad de las variables lógicas.
  Admiten un uso de entrada o de salida en la llamada a procedimientos a diferencia de los lenguajes funcionales.

- Cómputo relacional.
  Se implementa con un mecanismo general de búsqueda.

- Simplicidad.
  Una sintaxis y una semántica claras y sencillas acerca la programación lógica a los programadores, que encuentran una herramienta potente a un bajo costo de aprendizaje y programación. Ello hace de la programación lógica una excelente herramienta de prototipado rápido.

Los lenguajes de programación lógica basan su mecanismo de cómputo en el principio de resolución SLD de Robinson [135]. Éste es un mecanismo

de selección ordenada[7] de soluciones alternativas indeterminista. Cuando se evalúa un predicado, la solución no se restringe a la elección de un *caso* (i.e., una cláusula) de la definición de dicho predicado como ocurre en programación funcional cuando se selecciona una regla de definición de función. Por el contrario, cualquiera de las cláusulas es candidata a ofrecer una solución alternativa. En determinados casos es deseable *acotar* la búsqueda exhaustiva en caminos que no dan lugar a soluciones, para lo cual se utiliza el corte [35, 149]. El corte (*cut*) es una característica no declarativa que mejora el comportamiento operacional de los programas a costa, en algunos casos, de alterar la semántica del programa con los llamados *cortes rojos* [149].

Un programa lógico es un conjunto de declaraciones de predicados. Un predicado es un conjunto enumerable de cláusulas que lo definen. Cada cláusula es un caso diferenciado de la semántica del predicado.

Siguiendo el ejemplo del circuito combinacional de la sección anterior, mostramos un programa homólogo en la notación Prolog DEC-10 [52].

```
/* Puerta Or */
or(0,0,0).
or(0,1,1).
or(1,0,1).
or(1,1,1).
/* Puerta And */
and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).
/* Puerta Or exclusiva */
xor(0,0,0).
xor(0,1,1).
xor(1,0,1).
xor(1,1,0).
/* Sumador completo de un bit */
adder(I1,I2,Cin,S,Cout):-
                        xor(I1,I2,XO),
                        and(I1,I2,A1),
                        xor(Cin,XO,S),
                        and(XO,Cin,A2),
                        or(A1,A2,Cout).
```

[7] Usualmente en profundidad y de izquierda a derecha.

En este programa, la semántica de las funciones combinacionales se representa con *hechos*, i.e., relaciones de cuerpo vacío. Los dos primeros argumentos de los predicados *or*, *and* y *xor* representan las entradas de las puertas y el último argumento representa la salida. La cláusula *adder* contiene tres variables lógicas auxiliares ($A1$, $A2$ y XO) que se utilizan para conectar las puertas. Estas variables representan la salida de las puertas que se conectan a las entradas de otras.

Este programa se puede utilizar en el mismo sentido que el funcional, i.e., con instancias básicas de los argumentos de entrada para calcular la salida del circuito. Gracias a la componente lógica es posible consultar este programa con instancias libres de los argumentos de entrada. Por ejemplo, con el objetivo $adder(I1, I2, Cin, 1, 0)$ se obtienen los posibles valores de las entradas que cumplen la suma. En la única cláusula de la definición del predicado *adder*, se instancian las variables $I1$, $I2$ y $X0$ en el primer objetivo, $xor(I1, I2, XO)$. El mecanismo de resolución selecciona la primera cláusula del predicador *xor*, esto es, $xor(0, 0, 0)$ para resolver el objetivo, instanciando $I1$, $I2$ y $X0$ a 0, 0 y 0 respectivamente. El siguiente objetivo ($and(0, 0, A1)$) se resuelve con la primera cláusula del predicado *and*, instanciando $A1$ a 0. El objetivo $xor(Cin, 0, 1)$ se resuelve con la instanciación de Cin a 1, y $and(0, 1, A2)$ con $A2$ a 0. Por último, el objetivo $or(0, 0, 0)$ tiene éxito. Es posible solicitar una nueva solución al sistema activando el cómputo hacia atrás, lo que provoca la búsqueda de la alternativa pendiente que se encuentra en el primer objetivo de *adder*, que se puede resolver con la segunda cláusula del predicado *xor*. Con la resolución del resto de los objetivos se instancian $I1$, $I2$ y Cin respectivamente a 0, 1 y 0. La última alternativa corresponde a la instanciación de las variables $I1$, $I2$ y Cin a 1, 0 y 0. Si se solicita una nueva solución se produce *fallo* puesto que no hay alternativas pendientes.

Este mecanismo de búsqueda de soluciones no es posible en el programa funcional expuesto en la sección anterior[8]. Por ello, Prolog resulta una herramienta muy apropiada para diseñar sistemas incompletamente definidos que requieren un desarrollo incremental, facilitando la depuración del sistema y la construcción de prototipos. Sin embargo, la expresividad del caso funcional se pierde en cuanto que es necesario introducir variables lógicas

[8]Sí sería posible reformulando el programa funcional, lo que evidencia el menor poder declarativo de la programación funcional que la lógica en este tipo de casos. Véase [117] para una discusión detallada acerca de la expresividad de los diferentes paradigmas.

auxiliares.

En el siguiente ejemplo mostramos un programa de concatenación de listas homólogo al planteado en la sección anterior.

```
append([ ], L, L).
append([X|Xs], Y, [X|Zs]) :- append(Xs, Y, Zs).
```

Los símbolos [y] encierran los elementos de una lista, el símbolo | separa la cabeza de la cola de una lista y la lista vacía se representa con [ ].

Gracias a la componente lógica de Prolog es posible dividir una lista utilizando este mismo predicado formulando el objetivo $append(X, Y, [1, 2, 3])$, lo que daría como resultado las siguientes soluciones:
$\{X/[], Y/[1, 2, 3]\}$, $\{X/[1], Y/[2, 3]\}$, $\{X/[1, 2], Y/[3]\}$ y $\{X/[1, 2, 3], Y/[]\}$.
Cada una de las soluciones es una sustitución de variables lógicas, esto es, un conjunto de pares X/v, donde X es una variable y v su valor de vinculación. La respuesta muestra todas las posibles divisiones de la lista $[1, 2, 3]$ en dos sublistas.

1.1.3 La integración de la programación lógica y la programación funcional

Durante la última década ha crecido el interés en la integración de ambos estilos de programación [12, 51] para aunar sus ventajas. Se han seguido principalmente dos vías para realizar la integración. La primera, incorporando las funciones en un lenguaje de programación lógica [155, 20, 19, 18], añadiendo la evaluación funcional a la resolución. La segunda, incorporando la variable lógica en un lenguaje de programación funcional [98, 116, 7, 43] mediante la sustitución del encaje de patrones por la unificación y el tratamiento del indeterminismo. En [64] se pueden encontrar los fundamentos de este paradigma.

Algunas ventajas de este estilo de programación son:

- Inversión de funciones.
  Esto es posible gracias a la bidireccionalidad inherente a los lenguajes de programación lógica, que se incorpora en el mecanismo de evaluación de funciones.

- Evaluaciones más deterministas.
  Las funciones permiten evaluaciones más deterministas que la resolución de predicados. Esto puede evitar algunas de las características no declarativas de Prolog como el corte [149] o los recortes (*snips*) [14, 6].

- Aumento de la expresividad.
  Añadir las ventajas de ambos estilos de programación desemboca de manera natural en un incremento del poder expresivo [117].

El estrechamiento (*narrowing* [147]) es el mecanismo de evaluación más utilizado en la integración de los lenguajes lógico-funcionales. Es un caso especial de paramodulación [134, 87] en las teorías ecuacionales, que integra la reescritura de términos y la resolución SLD. El proceso de estrechamiento añade un grado de libertad extra (indeterminismo) al tener que seleccionar el subtérmino al que aplicar una ecuación[9]. Puesto que un cómputo funcional debe ser *confluente*, se imponen restricciones sintácticas [53] que lo aseguran.

Una instancia de la programación lógico-funcional: Babel

Babel [116, 119] es un lenguaje lógico-funcional diseñado para integrar la programación lógica en la funcional de una forma cómoda y matemáticamente bien fundamentada. Está basado en disciplina de constructoras [128] y sus tipos elementales se reducen a dos: términos construidos y valores booleanos. Los predicados lógicos se identifican con funciones booleanas, con las que se pueden especificar conectivas proposicionales. La negación booleana con la que se puede trabajar se acerca más a la negación lógica que la negación por fallo finito que dispone Prolog. Babel utiliza como mecanismo de cómputo el estrechamiento de términos, que se proporciona en las versiones impaciente y perezosa. Asimismo, se dispone de las versiones de primer orden, orden superior (en el sentido de [133]) y con variables lógicas de orden superior (como en [114]). La resolución SLD (y, por lo tanto, cualquier cómputo Prolog) se simula con el estrechamiento. Babel dispone de una aproximación computable a la igualdad [116, 119]. La semántica declarativa del lenguaje usa interpretaciones basadas en dominios de Scott [142].

Un programa Babel es un conjunto de definiciones de funciones. Una función es un conjunto enumerable de reglas de definición de función. Cada regla es un caso diferenciado de la semántica de la función.

En lo que sigue, nos centraremos en la presentación de la versión de orden superior, con mecanismo de cómputo de estrechamiento impaciente en profundidad de izquierda a derecha [93, 94]. Esto no supone pérdida de generalidad de la aplicación de los resultados de este trabajo a otros lenguajes lógico-funcionales con la misma estrategia de cómputo. En el apéndice A se describe formalmente la sintaxis y la semántica de esta versión.

[9] Esta situación ocurre en los casos en los que exista más de un posible unificador.

En esta sección introduciremos el lenguaje de forma más intuitiva y práctica a través de algunos ejemplos.

El siguiente es el programa Babel del circuito combinacional planteado anteriormente.

```
datatype
      level := 0 | 1.

/* Puerta Or */
fun or: level → level → level.
    or(0, 0) := 0.
    or(0, 1) := 1.
    or(1, 0) := 1.
    or(1, 1) := 1.
/* Puerta And */
fun and: level → level → level.
    and(0, 0) := 0.
    and(0, 1) := 0.
    and(1, 0) := 0.
    and(1, 1) := 1.
/* Puerta Or exclusiva */
fun xor: level → level → level.
    xor(0, 0) := 0.
    xor(0, 1) := 1.
    xor(1, 0) := 1.
    xor(1, 1) := 0.
/* Sumador completo de un bit */
fun xor: level → level → level → level → level.
    adder(I1, I2, Cin) := (xor(xor(I1, I2), Cin),
                           or(and(Cin, xor(I1, I2)), and(I1, I2)));
```

Los tipos de datos definidos por enumeración de constantes se declaran empezando con **datatype**, seguido del tipo y de la lista de constantes (separadas por |). En el ejemplo, las constantes 0 y 1 pertenecen al tipo *level*. Las funciones se declaran con **fun**, seguido del nombre de función, el tipo de los argumentos y el resultado, separados por →.

La expresión funcional de este tipo de problemas, más adecuada que la notación relacional, permite retener la expresividad de los lenguajes funcionales con la ventaja de realizar cómputos al estilo Prolog. Por ejemplo,

es posible formular una expresión tal como $adder(I1, I2, Cin) = (1, 0)$, obteniendo los posibles valores de las entradas $I1$, $I2$ y Cin que verifican el resultado $(1, 0)$.

En el siguiente ejemplo podemos ver la ventaja de la integración de las características de los dos estilos de programación definiendo funciones de orden superior para la aplicación de funciones a elementos de una lista y para la concatenación de listas.

```
fun map: (A → B) → list A → list B.
        map F [ ] := [ ].
        map F [X|Xs] := [F X | (map F Xs)].

fun append: list A → list A → list B.
        append [ ] L := L.
        append [H|T] L := [H | (append T L)].
```

En la declaración de tipos, *list* declara el tipo lista y los identificadores en mayúscula denotan variables de tipo. Las listas se escriben con la notación de Prolog.

Es posible formular la expresión objetivo *map* (*append* X) $[Y, [3]] = [[0, 1, 2], Z]$ en el programa anterior. En este ejemplo se hace uso del orden superior en la aplicación de la función *append* a una lista, y de la potencia de las variables lógicas para calcular listas diferencia (X e Y) y concatenación de listas (en Z). La evaluación de la expresión tiene como soluciones:
$< true, \{X/[\,], Y/[0, 1, 2], Z/[3]\} >$
$< true, \{X/[0], Y/[1, 2], Z/[0, 3]\} >$
$< true, \{X/[0, 1], Y/[2], Z/[0, 1, 3]\} >$
$< true, \{X/[0, 1, 2], Y/[\,], Z/[0, 1, 2, 3]\} >$

1.2 Paralelismo en lenguajes declarativos

Con frecuencia, cuando se habla de paralelismo se utilizan como sinónimos los términos paralelismo y concurrencia. Sin embargo, aquí se seguirá el criterio de algunos autores de asociar significados diferentes a cada término. Se hablará de paralelismo cuando se trate de aprovechar la independencia de diferentes partes de un programa para ejecutarlas simultáneamente en el tiempo y aumentar así la velocidad de ejecución. Esta traducción de la

independencia en mejora de rendimiento se realiza de forma transparente al usuario, manteniendo la semántica secuencial del programa. Por el contrario, se hablará de concurrencia cuando se trate de las construcciones de un lenguaje que permiten expresar las acciones simultáneas (concurrentes), por ejemplo, la creación y comunicación de procesos. La concurrencia no implica necesariamente ejecución simultánea en el tiempo. Ambos conceptos, paralelismo y concurrencia, son independientes aunque coexisten en muchas ocasiones. Nosotros explotaremos el paralelismo implícito de programas lógico-funcionales de un lenguaje no concurrente, proporcionando los mecanismos internos a la implementación de la sincronización que garantice la integridad semántica.

En general, un programa declarativo presenta tres tipos de paralelismo, dos de grano grueso, el conjuntivo y el disyuntivo, y uno de grano fino, el paralelismo de unificación.

El paralelismo disyuntivo procede del indeterminismo implícito que introduce la componente relacional o lógica del lenguaje. Se trata de un paralelismo de grano grueso compuesto por tareas que buscan soluciones alternativas al problema planteado. Se presenta fundamentalmente en programas que implementan aplicaciones con problemas combinatorios importantes, por ejemplo, un programa para realizar la integración simbólica de una función. La independencia de evaluación de las alternativas en diferentes entornos de cómputo ha propiciado la aparición de diferentes modelos de ejecución [162, 144, 141]

El paralelismo conjuntivo [50, 72] procede de la independencia de subtareas que colaboran en la obtención de una determinada solución. Se trata de un paralelismo de grano grueso pero más fino que el disyuntivo. Se presenta en programas deterministas susceptibles de descomponerse en subtareas independientes como, por ejemplo, un programa para calcular la derivada simbólica de una función.

Por último, el paralelismo de unificación [11] se corresponde con el paralelismo a nivel de instrucción de los lenguajes imperativos, ya que esta operación lógica contempla como casos particulares la mayoría de las operaciones imperativas (asignación de variables, test sobre condiciones, paso de parámetros en llamada a procedimientos, ...).

Estas componentes del paralelismo de un programa escrito en un lenguaje declarativo son ortogonales entre sí, en el sentido de que se pueden explotar de manera simultánea e independiente sobre una arquitectura paralela adecuada. Este trabajo trata de la identificación y explotación del paralelismo conjuntivo, si bien los resultados están incorporados también en

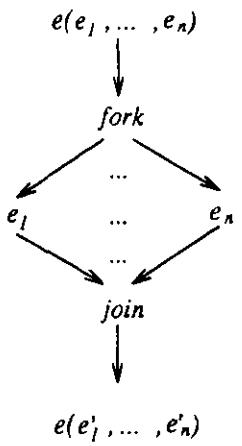


Figura 1.2: Paralelismo de argumentos en un lenguaje funcional impaciente.

la combinación del paralelismo conjuntivo y disyuntivo [63].

1.2.1 Paralelismo conjuntivo

El paralelismo conjuntivo en programación lógica se corresponde con el paralelismo implícito de los objetivos del cuerpo de una cláusula. Análogamente, en el ámbito de la programación funcional se encuentra el *paralelismo de argumentos* o *expresiones*. En un modelo simple de paralelismo para un lenguaje funcional con estrategia de cómputo impaciente se admite la evaluación paralela de los argumentos de una expresión, como se muestra en la figura 1.2. El mecanismo de reducción paralela genera un punto *fork* (activación de paralelismo) para la reducción de los argumentos de la función. Cuando se ha completado la reducción de los argumentos, que a su vez pueden contener más reducciones paralelas, se alcanza el punto *join* (punto de reunión) donde se realiza la composición de los resultados en la expresión original. Esta restricción que impone el par *fork-join* implementa las dependencias implícitas en el lenguaje, ya que expresa la dependencia de evaluación entre una expresión y sus argumentos (i.e., la expresión no se puede reducir hasta que sus argumentos se hayan reducidos[10]), así como la independencia de evaluación entre los argumentos.

Si se considera un lenguaje con mecanismo de reducción perezoso, apa-

[10]Esta condición se relaja cuando se introducen, por ejemplo, funciones predefinidas no estrictas como el bicondicional.

rece el problema de conocer la *necesidad* de cómputo de un determinado argumento. La evaluación ciega de los argumentos conduce en general a cómputos inútiles e incluso erróneos, puesto que en el esquema perezoso se demanda la evaluación de un argumento hasta el nivel requerido por la evaluación de la función (por ejemplo, forma normal de cabeza). Se han propuesto análisis de estricticidad [1, 25] para determinar los argumentos que son necesarios reducir y análisis de demanda [118] para inferir el nivel de reducción de los argumentos. La información derivada de estos análisis se incorpora en el esquema de reducción paralela para eliminar o relajar algunas condiciones de dependencia. Por ejemplo, conocer la estricticidad de un conjunto de argumentos supone la libertad de ejecutarlos en paralelo, mientras que conocer una cota del nivel necesario de reducción supone que dicho argumento puede ser reducido en paralelo hasta el nivel de reducción inferido evitando la sincronización hasta esa cota.

La presencia de variables lógicas en un lenguaje lógico-funcional como Babel, añade un nuevo nivel de dependencias al esquema de reducción paralela funcional. Análogamente a lo que ocurre en programación lógica [50], la evaluación paralela de expresiones que contengan variables comunes puede dar lugar a conflictos de vinculación cuando una variable compartida se instancie a diferentes valores. En lugar de seguir un modelo productor-consumidor (paralelismo *stream*), en el que se permita paralelismo *no estricto*[11], vamos a extender las ideas de [72][12] en programación lógica a la programación lógico-funcional; de manera que evitamos la costosa y compleja sincronización de tareas del esquema productor-consumidor, sobre todo en presencia de *backtracking*[13]. Bajo este esquema se restringe el paralelismo conjuntivo de manera que se permite la ejecución paralela de objetivos *independientes* (cfr. [72]).

Este trabajo se centra en la explotación de paralelismo conjuntivo independiente adecuado a la programación lógico-funcional. Para ello, se identifican las expresiones de la parte derecha de una regla que puedan ser evaluadas en paralelo, expresando las dependencias entre ellas. Estas expresiones son aquéllas que tienen como símbolo raíz el de una función. Por lo

[11] En el sentido de [73, 74], que permiten la paralelización de objetivos que compartan variables mientras no se produzcan conflictos de vinculación (objetivos conocidos gracias a un análisis de programa o a una anotación manual) o incluso en el sentido más general de Prolog concurrente [144].

[12] Basado en el modelo de paralelismo *estricto* de DeGroot [50] de programación lógica.

[13] En [130] se propone la paralelización de lenguajes de programación lógico-funcional con mecanismo de evaluación perezosa en donde se admite una forma de paralelismo productor-consumidor de argumentos, pero se rechaza el *backtracking*.

tanto, no se considera tan sólo el caso conjuntivo de Prolog, puesto que un lenguaje lógico-funcional contiene funciones arbitrarias además de la conjunción (término del que se deriva el nombre 'paralelismo conjuntivo'). Aunque sería más adecuado llamar a este tipo de paralelismo, paralelismo de argumentos o paralelismo de expresiones, se mantendrá el término paralelismo conjuntivo para seguir la nomenclatura previa [92, 140, 95].

1.3 Implementaciones de lenguajes lógico-funcionales

La alternativa inmediata para implementar un lenguaje lógico-funcional es el diseño de un intérprete, que debido a su ineficiencia se ve sustituida fundamentalmente por las dos siguientes:

- La compilación en otro lenguaje de alto nivel, e.g., Prolog, para el que existan implementaciones eficientes [155, 30].
- La definición de una *máquina abstracta*, que facilita la correspondencia entre el programa y la arquitectura donde se ejecuta, actuando de vehículo de ajuste entre ambos extremos principalmente en lo relativo a la gestión de memoria [161, 96, 101, 17].

Según la primera alternativa se han propuesto traducciones de lenguajes lógico-funcionales a Prolog, en donde el mecanismo de estrechamiento se ha simulado con la resolución SLD [20, 9]. Sin embargo, esta alternativa no se demuestra completa en cuanto que parte de la información de determinismo debida a las dependencias funcionales [64] presente en los lenguajes lógico-funcionales se pierde. Esta desventaja se resuelve con el diseño de máquinas abstractas especializadas para los lenguajes lógico-funcionales.

Se han utilizado dos tipos de máquinas abstractas, las basadas en grafos [93, 57] y las basadas en pilas [161, 101]. Las primeras son menos eficientes que las segundas puesto que en éstas se aprovecha la estructura de pila durante el *backtracking* en la desasignación eficiente de memoria, reduciendo además su fragmentación. De ello se deriva una ventaja adicional: la recolección de basura es una operación a realizar mucho menos frecuentemente. En este sentido, el exponente más significativo ha sido la máquina abstracta de Warren (WAM) [161], diseñada para el lenguaje de programación lógica Prolog, que se ha convertido en el estándar *de facto* en las implementaciones de lenguajes lógicos.

Se han propuesto máquinas abstractas para lenguajes lógico-funcionales desde dos perspectivas diferentes: diseñadas como extensiones de máquinas

lógicas [17] y como extensiones de máquinas funcionales [101]. Ambos tipos de aproximaciones alcanzan grados de eficiencia similares. Sin embargo, el paso de argumentos por pila que se efectúa usualmente en las máquinas funcionales se aprovecha en [101], que sigue la última alternativa, para gestionar los puntos de elección más eficientemente. En ésta, además, los cómputos funcionales se tratan casi con la misma eficiencia que en una máquina puramente funcional gracias al *corte dinámico* [102]. El corte dinámico ejerce la misma función que el corte de Prolog en tanto que es un corte *verde* [14], si bien el descarte de alternativas se produce automáticamente, sin intervención del programador. En este trabajo se seguirá esta última alternativa, reteniendo sus ventajas en la extensión a la programación lógico-funcional.

Recientemente se han propuesto extensiones paralelas de máquinas abstractas secuenciales. Dado que la representación de grafos en las máquinas funcionales se hace de forma *desordenada* (en contraposición a la ordenación *LIFO* de las representaciones en pilas), es posible extender más fácilmente una máquina basada en grafos para dar soporte al paralelismo que una máquina basada en pilas. Así, [92, 95] describen máquinas basadas en grafos para explotar paralelismo conjuntivo independiente de lenguajes lógico-funcionales. Sin embargo, la extensión de máquinas basadas en pilas se muestra como una técnica muy eficiente aún cuando más compleja en su diseño[15].

Por el modelo de memoria utilizado se pueden distinguir las implementaciones paralelas sobre memoria distribuida [16, 4, 157], memoria compartida [130, 69] e híbridas [8]. Las arquitecturas de memoria distribuida se han demostrado adecuadas para la explotación de paralelismo disyuntivo en los modelos de copia y recomputación [4, 89] y basado en procesos [38]. Bajo estos modelos las tareas paralelas progresan efectuando un bajo número de intercambio de mensajes, debido a la alta granularidad de las tareas. Las arquitecturas de memoria compartida son más adecuadas para la explotación del paralelismo conjuntivo en cuanto que las tareas paralelas acceden frecuentemente a los datos compartidos.

[14] Esto es, un corte que no altera la semántica del programa. Por lo tanto, se consigue el mismo resultado en la evaluación de un programa dado con y sin la presencia de cortes verdes [149].

[15] Véase el trabajo [69], desarrollado en el ámbito de la programación lógica.

1.4 Mejora de rendimiento

Para mejorar el rendimiento de la ejecución de los programas declarativos se especializa el código con información obtenida en tiempo de compilación directamente del programador o mediante análisis[16]. Esta mejora se realiza ortogonalmente en las siguientes vías:

- Mejora del componente secuencial de los programas.
  Se basa principalmente en la optimización de la gestión de memoria (recolección de basura [120], descarte de alternativas [45], gestión de estructuras de control [106, 138], ...) y la generación de código especializado para tareas específicas (unificación [97], indexación [77, 29], ...).

- Mejora del componente paralelo de los programas.
  Se basa en especializar las reglas paralelas con información de independencia que elimine o simplifique su comprobación en tiempo de ejecución. Complementariamente, esta información es útil también en la mejora del componente secuencial de los programas. Por ejemplo, con la información de independencia se puede mejorar la unificación, la recolección de basura y la desasignación de memoria mediante reclamación impaciente. En este trabajo desarrollaremos esta alternativa.

1.5 Propósito del trabajo

El propósito de este trabajo es la identificación automática y la explotación de paralelismo conjuntivo[17] en lenguajes lógico-funcionales. Bajo este paralelismo se admite la evaluación simultánea de expresiones independientes.

Es necesario proponer un sistema de identificación y expresión de paralelismo en el que se puedan expresar las dependencias de los argumentos de las funciones. A partir de estas dependencias se puede expresar de manera explícita el paralelismo implícito en las funciones. Esta expresión se realiza mediante reglas paralelas que contienen las construciones sintácticas necesarias para la expresión de paralelismo. El paso de la expresión de dependencias a la expresión de paralelismo no es inmediato ni ofrece una

[16] Las técnicas de análisis de programas se pueden utilizar tanto para comprobar la corrección o consistencia del programa (e.g., comprobación de tipos) como para *inferir* información implícita en el programa (e.g., inferencia de tipos).

[17] Llamado también paralelismo de expresiones o de argumentos.

solución única. Por lo tanto, propondremos varias alternativas que explotan diferentes grados de paralelismo. Normalmente, la explotación de un mayor grado de paralelismo viene asociada a un mayor coste su identificación. Por ello estudiaremos el compromiso entre estos dos factores.

Puesto que la identificación de paralelismo se produce parcialmente en tiempo de ejecución mediante comprobaciones de independencia, es deseable reducirlos en lo posible incorporando en el sistema de identificación de paralelismo información de independencia. Esta información se obtendrá de manera automática mediante análisis global de independencia basado en interpretación abstracta. Ya que el tiempo de análisis es un factor importante, consideraremos varias alternativas con distintos niveles de refinamiento en la información conseguida, lo que da lugar a diferentes tiempos de análisis. También se estudiará el compromiso entre estos dos factores.

Dado que no es deseable la planificación paralela de tareas que no den lugar a una mejora de rendimiento, se va a admitir la incorporación de información de granularidad, de manera que se impida la explotación paralela de las tareas de baja carga computacional. Se propondrá una estrategia que impida la evaluación paralela de las tareas de baja granularidad.

En cuanto a la explotación de paralelismo nos basaremos en las implementaciones eficientes de pilas para extender esta técnica al caso lógico-funcional, de manera que se consiga un buen rendimiento del sistema principalmente en el cómputo hacia atrás. No obstante, también se retendrán las optimizaciones secuenciales de las máquinas funcionales, lógicas y lógico-funcionales. Por ejemplo, la detección estática y dinámica de cómputos deterministas con reclamación impaciente de estructuras de control y el paso de parámetros y resultados por pila.

Sin embargo, no estamos interesados sólo en el nivel abstracto de cómputo, sino también en cómo realizar una arquitectura de soporte para las máquinas abstractas. Así, consideraremos una arquitectura de memoria compartida para la explotación de paralelismo conjuntivo, puesto que son arquitecturas que se han demostrado eficaces en la explotación de este tipo de paralelismo, en donde es frecuente el acceso a datos compartidos.

Es también deseable realizar una validación y análisis del sistema que realizaremos mediante simulación. Puesto que estamos también interesados en el análisis de su rendimiento, realizaremos una simulación temporal a bajo nivel que dé cuenta de los retardos en el acceso a memoria. Para ello se usará un lenguaje de especificación que disponga de un modelo temporal capaz de cumplir estos requisitos. Realizaremos la medida de diferentes alternativas de diseño en cuanto al tipo de bus y la caché.

En resumen, el propósito del trabajo se centra en contribuir a la viabilidad de uso de la programación lógico-funcional como herramienta práctica y eficiente en la resolución de problemas con fuerte componente declarativa.

Sumario

En este capítulo se ha introducido la programación lógico-funcional mediante el lenguaje Babel, para el que desarrollaremos en los próximos capítulos el sistema de identificación automática y explotación de paralelismo.

I

Identificación de paralelismo

Capítulo 2

Paralelización de Babel

En este capítulo se desarrolla un procedimiento para la extracción de paralelismo en programas secuenciales a partir de los grafos de dependencias condicionales de cada regla del programa. Un grafo expresa las dependencias locales de evaluación entre las expresiones de una regla. Se presentan tres estrategias para la generación de reglas paralelas a partir de los grafos. Cada estrategia establece un compromiso entre dos factores: paralelismo identificado y coste de su identificación en tiempo de ejecución. Además, se presenta un procedimiento para la incorporación de información de granularidad de manera que se evite la evaluación paralela de expresiones de baja carga computacional. El proceso de paralelización admite la simplificaci'on de las reglas paralelas incorporando información global de independencia, que obtenemos mediante el an'alisis descrito en el siguiente cap'itulo. Finalmente, se realiza un estudio de alto nivel del rendimiento de las estrategias.

2.1 Paralelismo conjuntivo independiente en Babel

A partir de una regla secuencial de la versión impaciente de primer orden de Babel[1] como la definición de la derivada simbólica de la suma de funciones:

derive(U+V) := derive(U) + derive(V).

Se pretende transformarla en una regla paralela que haga explícito su paralelismo implícito, por ejemplo en la siguiente:

[1] Al considerar primer orden, se añaden paréntesis a los argumentos de las funciones y constructoras de datos, mejorando la legibilidad.

```
derive(U+V) := let (par A1 := derive(U)
                       A2 := derive(V))
               in A1 + A2.
```

Esta regla explicita que las expresiones *derive(U)* y *derive(V)* se pueden evaluar en paralelo. Estas expresiones son subexpresiones de la expresión suma (+), que se evalúa después de aquéllas. Sin embargo, esta es sólo una de las posibles expresiones de paralelismo, en la que se ha identificado que *derive(U)* y *derive(V)* son efectivamente *independientes.*

Dos expresiones son independientes si:

- no comparten variables lógicas en tiempo de ejecución y
- la evaluación de una no depende del resultado de la otra.

La primera condición evita el problema que aparece cuando una variable, común a varias expresiones, se vincula a diferentes valores. Esta condición es análoga a la que se impone en el caso de la programación lógica. La segunda condición restringe el que una expresión se evalúe simultáneamente con una de sus subexpresiones. Esta condición se impone para impedir las suspensión de cómputo necesaria cuando el argumento funcional requiera la respuesta computada de uno de sus descendientes en el árbol sintáctico de la expresión. De otro modo, sería necesario gestionar la sincronización de las tareas asignadas a la evaluación de expresiones arbitrarias. En nuestro esquema, la sincronización se identifica y anota en tiempo de compilación. En [92] se propone una máquina para explotar paralelismo conjuntivo no restringido, que es capaz de evaluar argumentos funcionales y sus subexpresiones en paralelo, eliminando las condiciones de independencia anteriores. En [95] se presenta un sistema en el que la segunda condición de independencia se relaja de manera que un argumento funcional y sus descendientes se pueden evaluar en paralelo. El análisis de las implementaciones de estos sistemas demostraría cuál es el costo de las tareas de gestión de la sincronización necesaria.

Si no se puede afirmar la independencia de ambas expresiones en la transformación, podríamos generar la siguiente regla alternativa para el ejemplo anterior.

```
derive(U+V) := let (cpar i(U,V)
                        A1 := derive(U)
                        A2 := derive(V))
```

```
                    in A1 + A2.
```

En esta regla, la función $i(U, V)$ es responsable de comprobar *en tiempo de ejecución* la independencia entre las expresiones *derive(U)* y *derive(V)*, lo que se reduce a comprobar si las variables U y V son independientes. Si lo son, la evaluación de las expresiones se realizará en paralelo, en caso contrario se realizará secuencialmente. Sin embargo, cuando la carga computacional (*granularidad*) de las expresiones independientes no alcance una determinada cota, se planificarán secuencialmente, como en la siguiente regla:

```
derive(U+V) := let (seq A1 := derive(U)
                       A2 := derive(V))
               in A1 + A2.
```

En resumen, en la explotación de paralelismo hay que considerar dos factores esenciales:

- *Corrección.* Es necesario asegurar que el cómputo paralelo calcula los mismos resultados que el cómputo secuencial, o lo que es igual, que se preserve la semántica de los programas.

- *Eficiencia.* El cómputo paralelo de un programa debe, en el peor de los casos, calcular el resultado en el mismo tiempo que el correspondiente cómputo secuencial.

2.1.1 Modelo de evaluación paralela

En esta sección se presenta el modelo de evaluación paralela en el cómputo hacia adelante y hacia atrás (*backtracking*) de expresiones con funciones estrictas y no estrictas[2]. Nuestro propósito es conservar el mismo conjunto de respuestas y en el mismo orden que el mecanismo de evaluación secuencial, pero en menor tiempo.

[2]Consideraremos las constructoras de datos como funciones estrictas que se evalúan a sí mismas.

Cómputo hacia adelante

Si durante la evaluación de una expresión se alcanza un punto de activación de paralelismo[3] (punto *fork*), se produce la evaluación paralela de las expresiones correspondientes. Si consideramos que estas expresiones son argumentos de funciones estrictas, sólo pueden ocurrir dos situaciones:

- Se alcanza el punto de reunión (punto *join*).
- La evaluación de alguna de las expresiones produce fallo y es necesario activar el cómputo hacia atrás.

Sin embargo, cuando consideramos la evaluación paralela de una función no estricta, un fallo en la evaluación de alguno de sus argumentos no conduce necesariamente a la activación del *backtracking*. Supongamos que se produzca fallo en el argumento e_i de la función lógica conjunción $(e_1, \ldots, e_i, \ldots, e_n)$. Si se han evaluado todos los argumentos a su izquierda y es el único fallo computado, entonces se debe activar el cómputo hacia atrás. En caso contrario, cuando resta la evaluación de al menos una expresión e_j a la izquierda de e_i $(j < i)$, es posible que se produzca un resultado *definitorio*[4] en la posición más a la izquierda j para la función conjunción. Si el resultado de e_j es definitorio, sólo se tienen en cuenta las sustituciones debidas a la evaluación de $e_1, \ldots, e_j$ y el cómputo hacia adelante puede continuar.

En nuestro esquema se admite la evaluación paralela de los argumentos de las funciones estrictas y las lógicas no estrictas. Sin embargo, no se admite la evaluación paralela de los argumentos de la función no estricta condicional, puesto que estamos interesados sólo en la explotación del paralelismo conjuntivo. Si lo permitiésemos, estaríamos manejando una versión de paralelismo disyuntivo, puesto que las dos ramas del condicional comparten en general variables, siendo necesario mantener diferentes sustituciones hasta averiguar la rama efectiva para el cómputo.

En resumen, distinguimos los siguientes casos en el cómputo hacia adelante:

- Funciones estrictas:
  Durante la evaluación paralela pueden ocurrir las siguientes situaciones:

[3] Cumpliéndose las condiciones necesarias para que se puedan evaluar en paralelo, i.e., de independencia y granularidad, que se introducirá más adelante.

[4] Un valor definitorio es el que es suficiente para dar un valor definido a una función. Por ejemplo, *false* para la función lógica conjunción, que define el valor de la función como *false*.

  - Todos los argumentos se han evaluado.
    Es este caso se continúa el cómputo hacia adelante (se ha alcanzado con éxito el punto *join*).
  - Falla la evaluación de un argumento.
    En este caso se activa el mecanismo de cómputo hacia atrás (no se alcanza el punto *join* correspondiente).

- Funciones lógicas no estrictas ($e_1 \circ \ldots \circ e_n$, donde $\circ$ representa la conjunción (,) o la disyunción (;)):
  Durante la evaluación paralela pueden ocurrir las siguientes situaciones:

  - Todos los argumentos se han evaluado.
    Es este caso se continúa el cómputo hacia adelante (a lo sumo se ha calculado un valor definitorio para el argumento e_n).
  - Falla la evaluación de un argumento e_i:
    Es necesario esperar a la evaluación de los argumentos a la izquierda de e_i.
    * Si se calcula un valor definitorio para una posición j ($j < i$), tal que los valores $k < j$ no son definitorios, se reanuda el cómputo hacia adelante con las respuestas computadas correspondientes a $e_1, \ldots, e_j$ (se ha alcanzado con éxito el punto *join*).
    * Si no se calcula este valor definitorio, se debe descartar la evaluación del resto de argumentos, puesto que, al ser independientes, no es posible que puedan generar otra sustitución que impida el fallo de e_i. En este caso, se activa el mecanismo de cómputo hacia atrás (no se alcanza el punto *join* correspondiente).

Cómputo hacia atrás

Cuando se ha calculado fallo para un argumento e_i de una función $f(e_1, \ldots, e_i, \ldots, e_n)$ antes de alcanzar el punto *join*, se actúa como en el caso secuencial solicitando una nueva alternativa a la primera función a la izquierda de e_1 que disponga de ellas. Esto es válido tanto para funciones estrictas como para las no estrictas en las que no se ha calculado un valor definitorio para e_i.

Si el curso del *backtracking* alcanza el punto *join* de una evaluación paralela de los argumentos de una función estricta, entonces se debe solicitar una

alternativa de la expresión más a la derecha que disponga de alternativas[5]. Si no hay alternativas en la llamada paralela, se deben buscar otras alternativas en expresiones anteriores (hayan sido evaluadas secuencialmente o en paralelo). Si se encuentra tal expresión, se reactiva el cómputo hacia adelante a partir de la expresión e_i con alternativas, permitiéndose la evaluación paralela de e_i con $e_{i+1}, \ldots, e_n$. En el caso de que la llamada paralela corresponda a una función lógica no estricta, se busca el primer argumento e_i con alternativas a la izquierda del argumento definitorio (si no existe, se busca a partir del e_n). Si se encuentra, se reanuda la evaluación paralela hacia adelante de los argumentos $e_i, \ldots, e_n$[6].

En la activación del cómputo hacia atrás, se produce una búsqueda de la última expresión e con alternativas. Distinguimos los siguientes casos:

- La expresión corresponde a una llamada secuencial:
  Se actúa como en el caso secuencial, i.e., se descartan las vinculaciones hasta la expresión e que disponga de alternativas y se solicita una nueva , reanudándose el cómputo hacia adelante.

- La expresión corresponde a una llamada paralela:

  - Si la llamada paralela corresponde a la evaluación de los argumentos de una función estricta, se solicita una nueva alternativa para e, descartándose todos los cómputos a partir de e. A continuación, se reanuda el cómputo hacia adelante mediante la evaluación paralela de e y el resto de argumentos a su derecha.
  - Si la llamada paralela corresponde a la evaluación de los argumentos de una función lógica no estricta, se solicita una nueva alternativa para e. Se descartan todos los cómputos a partir de e hasta el argumento definitorio más a la izquierda. A continuación, se reanuda el cómputo hacia adelante mediante la evaluación paralela de e y el resto de argumentos a su derecha.

Esta descripción completa la presentación del modelo de evaluación paralela para funciones estrictas y predefinidas no estrictas. En los próximos apartados, presentaremos cómo identificamos y expresamos el paralelismo implícito en las reglas Babel, dando definiciones formales para los conceptos que se han introducido (expresiones paralelas, independencia, ...).

[5] Este comportamiento es similar al caso de la programación lógica descrito en [71].

[6] Nótese que es posible optimizar el esquema de *backtracking* evitando el descarte de los cómputos deterministas, de manera que se reutilicen.

2.1.2 Expresión de dependencias: el grafo de dependencias condicionales (*CDG*)

El *CDG* es una estructura que expresa el orden en que pueden evaluarse las subexpresiones de una regla debido a las dependencias locales. Usaremos este grafo para identificar el paralelismo de las reglas y expresarlo en reglas paralelas.

Parte del trabajo de este capítulo es una extensión a la programación lógico-funcional de las ideas de Jacobs *et al.* expuestas en [86], por lo que hemos mantenido parte de la notación allí introducida. Por ejemplo, utilizamos la misma notación para el grafo de dependencias condicionales (*CDG*) y algunas funciones de transformación, si bien el *CDG* es una estructura comúnmente utilizada en la expresión de dependencias como en [50, 122].

Unidades evaluables (*PEUs*)

Definiciones previas:

- Una variable auxiliar ($A_i \in AVar$) es una variable lógica que no pertenece al conjunto de las variables lógicas (Var) del programa ($Var \cap AVar = \emptyset$).
- Una expresión extendida es una expresión que puede contener variables auxiliares.

Definimos una *PEU* como la asignación de una variable auxiliar A_i a una expresión extendida e, i.e., la asignación $A_i := e$. Los argumentos de e pueden ser variables, términos de datos o variables auxiliares. Hay una *PEU* para cada símbolo de función de una parte derecha de una regla (*rhs*) excepto para la función raíz[7] (si existe).

> **Ejemplo 1.** PEUs *asociadas a una rhs.*
> $rhs \equiv d(a(X, b(Y)), c(X, Y), t(X))$, donde a, b, c y d son símbolos de función y t es un símbolo de constructora. Las *PEUs* asociadas a esta expresión son:
> $A_1 := b(Y) \quad A_2 := a(X, A_1) \quad A_3 := c(X, Y)$
> Donde A_1, A_2 y A_3 son variables auxiliares nuevas.

[7] No hay necesidad de construir una *PEU* para la función raíz de una *rhs*, ya que no se puede evaluar en paralelo con ninguna otra *PEU* de la misma regla. Esta imposición se deriva de la segunda condición de dependencia.

Grafo de dependencias condicionales

El *CDG* expresa las dependencias de evaluación entre las *PEUs* asociadas a una regla, de forma que se cumplan dos restricciones naturales:

- Las expresiones dependientes no se deben evaluar simultáneamente.
- Las expresiones dependientes deben preservar el orden de evaluación *innermost.*

Antes de dar su definición daremos unas definiciones previas.

- Independencia entre variables auxiliares[8]:
  - A_i depende de $A_j, i \neq j$, si A_j aparece en la subexpresión asignada a A_i (relación transitiva).
  - A_i depende condicionalmente de $A_j, i \neq j$, si A_i y A_j están vinculadas a términos no básicos[9] (relación transitiva y simétrica).

  Ejemplo 2. *Dependencia entre* PEUs.
  Siguiendo el ejemplo 1, tenemos:
  A_2 depende de A_1
  A_2 depende condicionalmente de A_1
  A_3 depende condicionalmente de A_1
  A_3 depende condicionalmente de A_2

- Independencia entre variables lógicas.
  Dos variables lógicas X e Y son independientes si los términos a los que se instancian no comparten variables.

  Ejemplo 3. *Dependencia entre variables lógicas.*
  En el ejemplo 1 observamos que Y y X son independientes para la sustitución $\{X/t(Z), Y/g\}$ y dependientes para la sustitución $\{X/s(Z), Y/t(Z)\}$.

- *Orden natural* entre *PEUs.*
  A diferencia del orden total que se puede establecer entre los objetivos de Prolog, de izquierda a derecha, no es posible establecerlo en un

[8] Esta independencia se deriva de las condiciones de dependencia expuestas anteriormente.

[9] Es decir, *pueden* compartir variables.

lenguaje lógico-funcional con funciones predefinidas no estrictas. Estas funciones son el condicional, el bicondicional o las lógicas (conjunción y disyunción). Sin embargo, sí se puede establecer un orden parcial inducido por el mecanismo de evaluación impaciente. Decimos que una *PEU* u es anterior a (o está a la izquierda de) otra *PEU* v si el mecanismo de evaluación impaciente evalúa la subexpresión de u antes de la subexpresión de v. Se puede definir entonces la función $nat(\Gamma, v)$, que devuelve la tupla de *PEUs* hasta el nodo v del *CDG* Γ según el orden de evaluación impaciente.

- Condición simple.
  Una condición simple puede ser:
  - *true*, que representa el valor lógico cierto.
  - *false*, que representa el valor lógico falso.
  - $i(X, Y)$, cuyo valor lógico es *true* si la variable lógica X es independiente de Y y *false* en caso contrario.
  - $g(X)$[10], cuyo valor lógico es *true* si la variable X es básica.

Definición del *CDG* Un *CDG* es un grafo dirigido acíclico cuyos nodos son las *PEUs* asociadas a la parte derecha de una regla, que cumple las siguientes condiciones. Para las *PEUs* $u \equiv A_j := E_j$ y $v \equiv A_i := E_i$:

1. Hay arcos *incondicionales* (sin etiquetar) de u a v si:
   - A_i depende de A_j y
   - v es anterior a u.
2. Hay arcos *condicionales* (etiquetados con una condición simple c) de u a v si:
   - A_i depende condicionalmente de A_j y
   - v es anterior a u.

   Estos arcos se etiquetan con las siguientes condiciones simples:

   (a) $i(X, X), \forall X$ tal que $X \in A_i, X \in A_j$. ($X \in A_i$ si la variable lógica X aparece en la subexpresión asignada a la variable auxiliar A_i).

[10] Sinónimo de $i(X, X)$.

(b) $i(X,Y), \forall X$ tal que $X \in A_i, X \notin A_j$ y $\forall Y$ tal que $Y \notin A_i, Y \in A_j$[11].

3. Hay arcos etiquetados con T de u a v si:

   (a) $u \equiv A_i := A_{i1} \rightarrow A_{i2}$ (condicional) y $v \equiv A_{i2} := e$, siendo e la expresión extendida asociada al segundo argumento de $\rightarrow /2$[12].

   (b) $u \equiv A_i := A_{i1} \rightarrow A_{i2} \square A_{i3}$ (bicondicional) y $v \equiv A_{i2} := e$, siendo e la expresión extendida asociada al segundo argumento de $\rightarrow /3$.

   (c) $u \equiv A_i := A_{i1}, A_{i2}$ (conjunción) y $v \equiv A_{i2} := e$, siendo e la expresión extendida asociada al segundo argumento de $, /2$.

4. Hay arcos etiquetados con F de u a v si:

   (a) $u \equiv A_i := A_{i1} \rightarrow A_{i2} \square A_{i3}$ y $v \equiv A_{i3} := e$, siendo e la expresión extendida asociada al tercer argumento de $\rightarrow /3$.

   (b) $u \equiv A_i := A_{i1}; A_{i2}$ (disyunción) y $v \equiv A_{i2} := e$, siendo e la expresión extendida asociada al segundo argumento de $; /2$.

Ejemplo 4. CDG *asociado a la rhs de una regla Babel.*
La *rhs* de la regla $h(X,Y) := u(a(X), b(Y), (c \rightarrow s \square t), d(X,Y))$. (donde a, b, c, d, f y h son símbolos de función, y s, t y u son símbolos de constructora) tiene asociado el *CDG* de la figura 2.1.

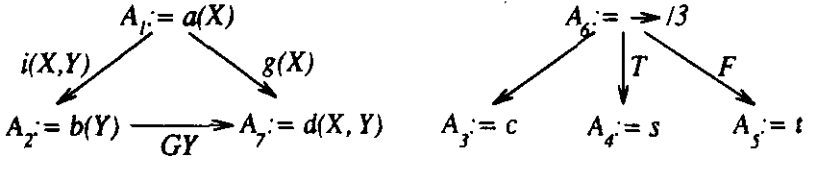


Figura 2.1: *CDG.*

[11] La inclusión de $i(X,X)$ evita la necesidad de arcos etiquetados como $i(X,Y)$ entre A_i y A_j, dado $X \in A_i, X \in A_j$ y Y tal que $Y \in A_i, Y \notin A_j$.

[12] La notación ϕ/a denota a una función o constructora ϕ de aridad a (número de argumentos).

Modelo de ejecución paralela de *CDGs* La ejecución de un *CDG* es un proceso cíclico en el que cada ciclo comienza tan pronto como finaliza la evaluación de una *PEU*. Un ciclo se compone de los siguientes pasos:

1. Eliminación de arcos:
   - Se eliminan todos los arcos cuyo origen haya sido evaluado.
   - Se elimina el arco de entrada T a cada nodo cuyo nodo padre tenga su primer hijo evaluado a *true*.
   - Se elimina el arco de entrada F a cada nodo cuyo nodo padre tenga su primer hijo evaluado a *false*.
2. Llamada al mecanismo de estrechamiento para las subexpresiones de las *PEUs* a las que no llegan arcos.

2.1.3 Transformaciones del *CDG*

En esta sección presentamos una serie de transformaciones que aplicamos a los *CDGs* con los siguientes objetivos:

1. Simplificación de las condiciones con información global.
2. Incorporación de información de granularidad.

Para ello introducimos en primer lugar algunos conceptos que serán necesarios para las transformaciones.

- Un *hecho* es una relación entre dos variables lógicas que representa la independencia entre ellas. $i(X,Y)$ representa que X e Y son independientes[13]. $d(X,Y)$ representa que X e Y son dependientes.
- Un *contexto* C de un *CDG* Γ es un conjunto de hechos que son ciertos antes de la evaluación de Γ.
- Un *ECDG* es un grafo de dependencias condicionales extendido con un contexto.

[13]Utilizamos la misma sintaxis para hechos y para condiciones. Sin embargo, esto no provoca ambigüedad puesto que el significado se deduce trivialmente.

A continuación definimos una serie de funciones para manejar contextos y simplificar *CDGs*[14].

1. Función *maintained facts (mf)*.
   Esta función devuelve el conjunto de hechos que se conocen como ciertos en tiempo de compilación después de la evaluación de una expresión bajo un contexto. mf extiende un contexto C con los hechos de independencia derivados de la basicidad de los términos instanciados a las variables lógicas. Esta función elimina del contexto los hechos que no se pueden asegurar como ciertos después de la evaluación de una *PEU*.

   Esta función hace uso de una estructura que almacena una entrada para cada llamada a función del programa. Cada entrada contiene los modos de las variables de una función después de su evaluación. Esta información se dispone como pares $X/ground$, X/Y o X/top. El par $X/ground$ significa que la variable X se instanciará en tiempo de ejecución a un término básico. El par X/Y significa que la variable X es dependiente de la variable Y. El par X/top significa que la variable X puede ser instanciada en tiempo de ejecución a cualquier término[15]

   La especificación de la función mf es la siguiente:

$$\begin{aligned} mf(C,F) = \{&i(X,v) \text{ tal que } X/ground \in mt(F)\}_{\forall v \in V_F} \\ \cup\{&f(X,Y) \text{ tal que } X/ground \notin mt(F), Y/ground \notin \\ &mt(F), i(X,V_F) \subseteq C, i(Y,V_F) \subseteq C\}_{\forall f(X,Y) \in C} \end{aligned}$$

   Donde V_F es el conjunto de variables lógicas de la función F, $f(X,Y)$ denota un hecho genérico acerca de las variables X e Y (i.e., $i(X,Y)$ o $d(X,Y)$), C es un contexto e $i(X,V_F)$ denota $\bigcup_{\forall v_i \in V}\{i(X,v)_i$ tal que $i(X,v)_i \in C\}$.

2. Función *Post*.
   Devuelve el conjunto de hechos mantenidos tras la ejecución de un

[14] A partir de este momento, consideramos *CDGs* en donde sólo aparecen funciones estrictas. En la sección 2.2.3 mostraremos cómo manejar en este marco las funciones predefinidas no estrictas. Sin embargo, esto no implica falta de generalidad del marco. Alternativamente, las funciones predefinidas no estrictas se podrían haber considerado desde este momento.

[15] Como veremos en el capítulo 3, estos pares se derivan directamente de las *sustituciones abstractas* obtenidas mediante análisis.

CDG bajo un determinado contexto. Se calcula como los hechos que se mantienen tras la ejecución de los nodos del *CDG* en el orden natural.

$$post(C,\Gamma) = \begin{cases} C & \text{si } nat(\Gamma) =<> \\ post(\Gamma - \{v_1\}, mf(v_1, C)) & \text{tal que } nat(\Gamma, v_1) =< v_1 > \text{ e.o.c.} \end{cases}$$

El sentido del operador '-' sobre un grafo Γ y un conjunto de nodos V es el siguiente:
$\Gamma - V =< V_\Gamma - V, E' >$, donde:
$\Gamma =< V_\Gamma, E >, E' = \{< u, c, v >\in E / STu \in V, v \in V$.
El operador '-', por lo tanto, está sobrecargado para las operaciones de diferencia de conjuntos (como en $V_\Gamma - V$) y eliminación de nodos de un grafo (como en $\Gamma - V$).

3. Función *simplify*. Esta función devuelve el *CDG* Γ simplificado bajo el contexto C. La simplificación consiste en la eliminación de los arcos cuyas condiciones se satisfacen[16] bajo C.

   Introducimos la función de evaluación de condiciones *eval* de una condición c bajo un contexto C.

$$eval(c,C) = \begin{cases} false & \text{si } c = i(X,Y), d(X,Y) \in C \\ true & \text{si } c = i(X,Y), i(X,Y) \in C \\ c & \text{e.o.c.} \end{cases}$$

   La especificación de la función de simplificación es la siguiente:
   $simplify(\Gamma, C) =< V, C_s >$, donde $\Gamma =< V, E >$,
   $C_s = \{< v_i, c, v_j > \text{ tal que } < v_i, c, v_j >\in E, nat(\Gamma, v_i) =< v_i, \ldots > , eval(c, C) = true\} \cup E_s\} >$ y $simplify(\Gamma - < v_i, c, v_j >, C) =< V, E_s >$[17]

[16] Es decir, se interpretan como ciertas.

[17] El operador '-' se ha sobrecargado también para la eliminación de arcos de un grafo.

Nótese que se sigue el orden natural de los nodos en Γ cuando se aplica la función *eval*. Esto es necesario puesto que, en caso contrario, se podrían invalidar hechos del contexto tras la evaluación de una *PEU*, como se puede comprobar en el siguiente ejemplo.

Ejemplo 5.
X es una variable libre en la regla $h(Y) := t(a(X,Y), b(Y), c(X))$., cuyo contexto inicial es $\{i(X,Y)\}$. Tras la evaluación de $a(X,Y)$, $i(X,Y)$ se elimina del contexto.

Transformación basada en las variables libres

Las variables libres son las que no aparecen en la parte izquierda de una regla. La *PEU p* que contenga la primera aparición de una variable libre X es independiente (con respecto a X) de las *PEUs* anteriores, puesto que no es posible que se haya producido *aliasing*[18] con otras variables. Este *aliasing* es sólo posible a partir de la evaluación de la *PEU p*. El procedimiento de construcción del *CDG* genera arcos con condiciones de independencia sobre las primeras apariciones de estas variables. El propósito de esta transformación es la eliminación de estos arcos, puesto que se conoce en tiempo de compilación la independencia de la primera aparición de una variable libre de las variables que aparecen en las *PEUs* anteriores. Además, ya que la *PEU p* que contenga la primera aparición de una variable libre X es la generadora de la instanciación de X, hasta que no se haya evaluado p, no es posible conocer si X puede ser independiente de las variables que aparezcan en *PEUs* posteriores y, en particular, de sí misma, esto es, que sea básica. Esto significa que los arcos que salgan de p con una condición de independencia sobre X se pueden etiquetar con la condición *false*[19]. La especificación de la función de transformación de variables libres (fvt) es la siguiente[20]:

$fvt(\Gamma) = \Gamma'$, donde $\Gamma =< V, E >, \Gamma' =< V, E' >$, siendo:

$E' =$
$E - \{< v_i, i(X,Y), v_j >\in E$ tal que $X \in vars(v_i), X \in \mathcal{E}, occ(X, v_i, \Gamma) = 1\}\cup$
$\{< v_i, false, v_j >$ tal que $X \in \mathcal{E}, < v_i, i(X,X), v_j >\in E, occ(X, v_i, \Gamma) = 1\}-$
$\{< v_i, i(X,Y), v_j >\in E$ tal que $X \in vars(v_j), X \in \mathcal{E}, occ(X, v_j, \Gamma) = 1\}$

Donde $\mathcal{E}$ es el conjunto de variables libres de Γ y $occ(X, v, \Gamma)$ es la función que devuelve el número de nodos en los que ha aparecido X hasta el nodo v, según el orden parcial de Γ.

---

[18] Hay *aliasing* entre dos variables si los términos a los que están ligadas comparten variables.

[19] No hay lugar de preguntarse acerca de si es posible iniciar la evaluación paralela de p y una *PEU* posterior que contenga a X, puesto que el grafo transformado es semánticamente equivalente al original.

[20] Obviamente, esta transformación puede eliminarse y en su lugar modificar el procedimiento de obtención del *CDG*. Sin embargo, hemos mantenido aislada esta transformación del procedimiento por modularidad.

Transformación basada en información de independencia

Con un análisis global es posible inferir la independencia de variables como veremos en el capítulo 3. En concreto, se puede conocer el estado de independencia o dependencia de las variables antes y después de la evaluación de una *PEU* y, en particular, en el punto previo a la evaluación de una regla. Esta información se puede incorporar al *CDG* inicial y transformarlo de manera que se simplifiquen las condiciones que se conozcan en tiempo de compilación gracias al análisis de independencia[21], que es el propósito de esta transformación. Para ello se simplifica el *CDG* inicial con los hechos derivados de la información independencia. La función *df* (*derived facts*) calcula los hechos que se pueden inferir de la información de independencia en *mt*. Su especificación es la siguiente:

$$df(F) = \{c \text{ tal que } X/ground \in mt(F)\}_{\forall c \in i(X,V_F)} \cup \{d(X,Y) \text{ tal que } X/Y \in mt(F)\}$$

La especificación de la función de transformación basada en información de independencia (*ibt*) es la siguiente:

$ibt(\Gamma) = \Gamma'$, siendo:

$\Gamma' = simplify(\Gamma, C)$ y $C = df(F)$ es el conjunto de hechos derivados de la información de independencia en el punto previo a la evaluación del caso F correspondiente a la función definida por la regla.

Transformación basada en información de granularidad

La granularidad es una medida de la carga de proceso que una tarea va a generar al ser evaluada. No se debe permitir la explotación paralela de tareas cuyo coste de planificación sea superior a la ganancia que se obtiene por la evaluación paralela. Por lo tanto, es interesante proporcionar un mecanismo que evite la explotación paralela de *PEUs* cuya granularidad no sea suficiente para obtener una ganancia de velocidad.

En diferentes ámbitos se han presentado trabajos relativos a la estimación de granularidad. Por ejemplo, en programación funcional se han presentado varios trabajos sobre análisis de *complejidad*[22] [131, 136, 15, 159, 76, 113]. En programación lógica se han propuesto análisis que son inherentemente más complejos debido al indeterminismo provocado por las variables lógicas [153, 47, 48, 49, 103, 104].

[21] Y eventualmente, a una anotación manual, que si bien no se descarta, no se aconseja.

[22] La complejidad es un término que se emplea en programación funcional, análogo al término granularidad que se emplea en el ámbito de la programación lógica.

Con la presente transformación, permitimos la inclusión de la información de granularidad derivada de un análisis estático o de una anotación manual. Asumimos que las *PEUs*[23] con una carga computacional inferior a una determinada cota se han anotado como nodos de *grano fino* y el resto como nodos de *grano grueso*. Mediante esta transformación evitamos la paralelización de los nodos de grano fino. El objetivo es *secuencializar* el cómputo de los nodos de grano fino que pertenezcan a un determinado *CDG*. Sin embargo, un grupo secuencializado de nodos de grano fino sí puede tener suficiente carga computacional (suficiente grano) como para ser evaluados en paralelo con otras tareas. El problema que resolvemos es cómo secuencializar los nodos de grano fino de una manera razonable. No es posible dar una solución óptima en tiempo de compilación a este problema, puesto que no se tiene una medida certera de la carga computacional, sino de una determinada cota superior. Por otra parte, hay que considerar que la identificación de paralelismo en tiempo de ejecución pasa por la evaluación de condiciones de independencia entre nodos o grupos de nodos. La manera en que se agrupen nodos afectará a la independencia condicional entre otros grupos de nodos. La identificación de paralelismo debe ser lo más rápida posible para no restar rendimiento a su explotación. Por lo tanto, una de nuestras premisas en la agrupación de nodos es que las condiciones entre grupos de nodos que se deben resolver en tiempo de ejecución sean fáciles de evaluar. A continuación propondremos una estrategia que se basa en los siguientes puntos:

- Secuencialización de los nodos en el orden natural. Este criterio está justificado porque la instanciación de las variables lógicas es mayor, en general, a medida que se resuelven subexpresiones en el orden natural. Por otra parte, el proceso de agrupación de nodos se simplifica, puesto que al no considerar un orden arbitrario, se evita una exploración de independencias entre grupos, lo que supone una complejidad exponencial del algoritmo.

- Los grupos de grano fino se secuencializan con nodos de grano grueso, de manera que la independencia entre grupos se rompa lo menos posible. Para ello se permite dividir un grupo de nodos de grano fino en dos, de manera que uno de los grupos se secuencializa con un nodo de grano grueso anterior y el otro se secuencializa con un nodo de grano grueso posterior.

[23] A partir de aquí también nos referiremos a las *PEUs* como nodos.

Descripción de la estrategia Sea $\mathcal{C}$ el conjunto de nodos de grano grueso de un *CDG* Γ y $\mathcal{F}$ el conjunto de nodos de grano fino del mismo *CDG* Γ. Según el orden natural, en general encontramos grupos de nodos pertenecientes a $\mathcal{C}$ alternados con grupos de nodos pertenecientes a $\mathcal{F}$. Dividimos un grupo de nodos G perteneciente a $\mathcal{F}$ en dos subgrupos G_1 y G_2. Secuencializamos G_1 con el nodo u_i, perteneciente a $\mathcal{C}$, inmediatamente anterior a G en el orden natural. De igual forma, secuencializamos G_2 con el nodo u_j, perteneciente a $\mathcal{C}$, inmediatamente posterior a G en el orden natural. Sea $\hat{G_1} = G_1 \cup u_i$ y $\hat{G_2} = G_2 \cup u_j$. Los subgrupos G_1 y G_2 se eligen de forma que las condiciones que etiquetan los arcos de $\hat{G_1}$ a $\hat{G_2}$ sean condiciones de independencia de bajo coste, de manera que si se genera una condición de independencia que haya que resolver en tiempo de ejecución, se trate de una condición rápida de evaluar. Esta estrategia es una instancia de la solución al problema no decidible más general cuando se considera un orden arbitrario en la agrupación de nodos.

La especificación formal de la transformación basada en información de granularidad es la siguiente:
$tgi(\Gamma) =< V, E' >$, donde $\Gamma =< V, E >$

El conjunto de arcos E' resultado de la transformación se calcula en base al conjunto E'' que se especifica a continuación.

$$E'' = \begin{cases} E & \neg\exists v_i \in \mathcal{F} \\ link(V) & \neg\exists v_i \in \mathcal{C} \\ E \cup link(\{v_1, \ldots, v_i\}) \cup slink(\{v_i, \ldots, v_n\}) & v_1, \ldots, v_{i-1} \in \mathcal{F}, v_i \in \mathcal{C} \\ E \cup slink(\{v_1, \ldots, v_n\}) & \text{e.o.c.} \end{cases}$$

E'' es el conjunto de arcos que son resultado de la aplicación de la estrategia propuesta.

$E' = E'' - \{< v_i, c, v_j > \in E$ tal que $< v_i, false, v_j > \in E'', c \neq false\}$

La función $link(V)$ se limita a secuencializar los nodos en el conjunto V según el orden natural.

$link(V) = \{< v_1, false, v_2 >\} \cup link(\{v_2, \ldots, v_n\}), nat(V) =< v_1, \ldots, v_n >$

La función *slink* secuencializa los nodos de V según la estrategia propuesta. En la figura 2.2 se muestran ordenados cada uno de los casos que distingue. Los casos del 1 al 5 en la figura se encuentran ordenados de arriba a abajo en la especificación formal de *slink*. La secuencialización se muestra con un arco $\rightarrow$ que conecta dos nodos u y v como $u \rightarrow v$, que significa que la evaluación de v es posterior a la evaluación de u. Las llaves encierran nodos que pertenecen a un determinado grupo de nodos, i.e., a los conjuntos $\mathcal{C}$ y $\mathcal{F}$, nodos de grano grueso y nodos de grano fino, respectivamente. En el primer caso no hay secuencialización porque todos los nodos de V son de

grano grueso. En el siguiente caso se asume que hay un número determinado de nodos de grano grueso seguidos en el orden natural por un conjunto de nodos de grano fino. En este segundo caso se secuencializan los nodos de grano fino con el nodo de grano grueso más a la derecha. Los casos tercero y cuarto distinguen la situación en que un nodo de grano fino se encuentra entre dos conjuntos de nodos de grano grueso. El nodo de grano grueso se asocia con el nodo de grano grueso a su izquierda o derecha (caso 3 o 4, respectivamente) de manera que la dependencia condicional se rompa lo menos posible, lo cual se calcula mediante la función *weight*. El último caso recoge la situación en la que un conjunto de nodos de grano fino se encuentra entre dos conjuntos de grano grueso. La estrategia propuesta se aplica en este caso de la siguiente forma:

- Se secuencializan los nodos de grano fino en dos grupos. El grupo de la izquierda se secuencializa con el nodo de grano grueso inmediatamente a su izquierda.
- El grupo de nodos de grano fino de la derecha se secuencializa con el nodo de grano grueso inmediatamente a su derecha.

La estrategia se aplica recursivamente al conjunto de nodos a la derecha que restan (donde, obviamente, el nodo más a la izquierda es un nodo de grano grueso). Nótese que el nodo más a la izquierda en el orden natural del conjunto V, argumento de *slink*, es de grano grueso.

La especificaci'on de esta funci'on es la siguiente:

$$
slink(V) = \begin{cases}
\{\} \\
\quad \text{si } \neg\exists v_i \in \mathcal{F}, v_i \in V \\
link(\{v_{i-1}, \ldots, v_n\}) \\
\quad \text{si } v_i, \ldots, v_n \in \mathcal{F}, v_i = leftmost(V, \mathcal{F}) \\
link(\{v_{i-1}, v_i\}) \cup slink(\{v_{i+1}, \ldots, v_n\}) \\
\quad \text{si } v_i = leftmost(V, \mathcal{F}), v_{i+1} \in \mathcal{C}, \\
\quad weight(\{v_{i-1}\}, \{v_i, v_{i+1}\}) \geq_w weight(\{v_{i-1}, v_i\}, \{v_{i+1}\}) \\
link(\{v_i, v_{i+1}\}) \cup slink(\{v_{i+1}, \ldots, v_n\}) \\
\quad \text{si } v_i = leftmost(V, \mathcal{F}), v_{i+1} \in \mathcal{C}, \\
\quad weight(\{v_{i-1}\}, \{v_i, v_{i+1}\}) <_w weight(\{v_{i-1}, v_i\}, \{v_{i+1}\}) \\
link(\{v_{i-1}, \ldots, v_{l-1}\}) \cup link(\{v_l, \ldots, v_{k+1}\}) \\
\cup slink(\{v_{k+1}, \ldots, v_n\}) \\
\quad \text{si } v_i = leftmost(V, \mathcal{F}), v_i, \ldots, v_k \in \mathcal{F}, v_{k+1} \in \mathcal{C}, k > i, \\
\quad weight(\{v_{i-1}, \ldots, v_{l-1}\}, \{v_l, \ldots, v_{k+1}\}), (i \leq l \leq k > 1) \\
\quad \text{es un minimal}
\end{cases}
$$

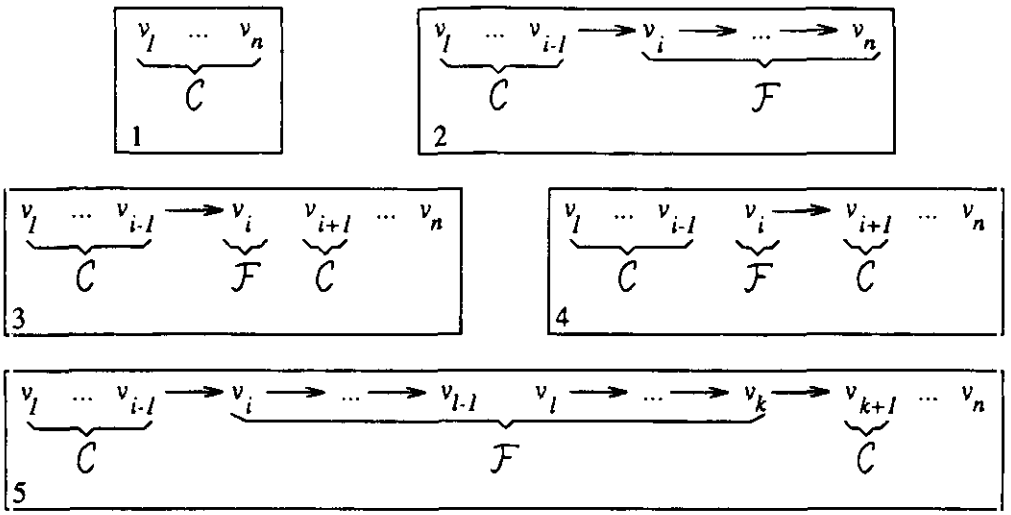


Figura 2.2: Distinción de casos de la función *slink*.

A continuación damos las especificaciones que restan.

- $min(V, N)$.
  Esta función devuelve el nodo v_i de V más a la izquierda en el orden natural de los nodos de V.

- Orden parcial $>_w$ (más costoso de evaluar).
  Una condición de basicidad es, en general, menos costosa de evaluar en tiempo de ejecución que una condición de independencia entre dos variables diferentes. Como es imposible conocer en tiempo de compilación si una condición simple de independencia es menos costosa que una condición de basicidad (conjunción de condiciones simples de basicidad), hemos tomado el criterio de asignar un "peso" mayor a una condición simple de independencia frente a cualquier número de condiciones de basicidad. Esta función hace uso de un orden parcial que da cuenta del costo de evaluación de condiciones. Así, definimos el orden parcial $>_w$ sobre $\{T, A, B, L\}$ como[24]:

  $T >_w A >_w L$
  $T >_w B >_w L$
  $nA + mB \geq_w n'A + m'B$ si y sólo si $n \geq n', m \geq m'$
  $nB >_w mA$

[24] En lo que sigue, nX significa n veces X y m, n, m', n' son números naturales.

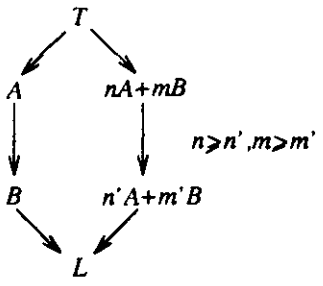


Figura 2.3: Orden parcial de pesos asignados a la evaluación de condiciones.

Donde T es la cota superior del orden parcial (peso superior a cualquier otro), L es la cota inferior del orden parcial (peso inferior a cualquier otro), A es el peso asociado a la evaluación de una condición de basicidad y B es el peso asociado a la evaluación de una condición de independencia. La figura 2.3 representa el orden parcial descrito, en donde una flecha $X \rightarrow Y$ representa que X es más costoso de evaluar que Y.

- Suma de pesos ($\sum$).
  Esta función es necesaria para la suma de los pesos de las condiciones que etiquetan los arcos. Su definición es:

  $\sum\{\} = L$
  $\sum\{L, x_1, \ldots, x_n\} = \sum\{x_1, \ldots, x_n\}$
  $\sum\{T, x_1, \ldots, x_n\} = T$
  $\sum\{x_1, x_2, \ldots, x_n\} = x_1 + \sum\{x_2, \ldots, x_n\}$ tal que $x_1 \in \{A, B\}$

- *weight*.
  $weight(V_1, V_2)$ es una función que calcula el "peso" de las condiciones que etiquetan los arcos que conectan los conjuntos V_1 y V_2.

  $weight(V_1, V_2) = \sum\{wc(c_i)\}_{\forall u \in V_1, \forall v \in V_2 \text{ tal que } <u,c_i,v> \in E}$, donde:

  $$wc(c) = \begin{cases} T & \text{si } c = false \\ A & \text{si } c = i(X,X) \\ B & \text{si } c = i(X,Y) \end{cases}$$

2.2 Obtención de reglas Babel paralelas

En esta sección veremos cómo obtener reglas paralelas Babel a partir de los *CDG* que expresen el paralelismo identificado estáticamente.

El modelo de ejecución de un *CDG* explota el paralelismo máximo de una regla. Sin embargo, la implementación de este modelo es ineficiente porque en cada ciclo de ejecución es necesario comprobar las condiciones de independencia. En su lugar, las *PEUs* se agrupan de modo que estas comprobaciones sólo se realizan cuando finaliza la evaluación de algún grupo. El problema de la agrupación no admite solución óptima en tiempo de compilación, por lo que presentaremos diferentes estrategias.

En primer lugar, veremos unas reglas de transformación que se aplican a los *CDG* para dividirlos y especializarlos. A continuación presentamos las estrategias de agrupación. Por último, veremos la transformación de las reglas secuenciales Babel en paralelas, de modo que expresen el paralelismo identificado.

2.2.1 Reglas de transformación

Antes de presentar las reglas de transformación, introduciremos dos nuevos tipos de expresiones que aparecerán en las transformaciones del *CDG*. En primer lugar, se definen inductivamente las expresiones de grafo de ejecución (*EGEs*).

- Una *PEU* es una *EGE*.
- Para las *EGEs* $E_1, \ldots, E_k$ y la condición P (conjunción de condiciones simples sobre variables), las siguientes expresiones son también *EGEs*:
  - $(par\ E_1 \ldots E_k)$
  - $(seq\ E_1 \ldots E_k)$
  - $(if\ P\ E_1\ E_2)$
  - $(cpar\ P\ E_1 \ldots E_k)$

La evaluación de una EGE se realiza en profundidad y de izquierda a derecha. La evaluación de una expresión *par* activa la evaluación paralela de sus argumentos. La evaluación de una expresión *seq* activa la evaluación secuencial de sus argumentos. La evaluación de una expresión if activa la evaluación de su segundo argumento en el caso de que la condición representada en su primer argumento se satisfaga y la de su tercer argumento en caso contrario. La expresión *cpar* es una abreviatura de:

$$(if\ P\ (par\ E_1 \ldots E_k)\ (seq\ E_1 \ldots E_k))$$

El segundo tipo de expresiones que introducimos son las expresiones híbridas (expresiones *HE*), que son *EGEs*, en las que, además de *PEUs*, pueden aparecer alternativamente *ECDGs*.

Las reglas de transformación son las siguentes:

1. *Split.*

   Esta regla devuelve una expresión híbrida resultado de dividir el *ECDG* Ξ en dos subgrafos Γ_1 y Γ_2. La idea es dividir el grafo en Γ_1 y Γ_2, tal que en el peor de los casos, Γ_2 se ejecute después de Γ_1. Por ello, no se admiten arcos de Γ_2 a Γ_1, puesto que en otro caso y, en general, algunas *PEUs* de Γ_1 deberían esperar por *PEUs* de Γ_2. Si no hay arcos entre Γ_1 y Γ_2, ambos subgrafos pueden evaluarse en paralelo. Si los arcos entre Γ_1 y Γ_2 no se pueden satisfacer bajo el contexto[25], la evaluación de Γ_2 debe esperar a la evaluación de Γ_1, puesto que se ha detectado en tiempo de compilación la dependencia entre Γ_1 y Γ_2. Si las condiciones de independencia entre Γ_1 y Γ_2 no se pueden evaluar a *true* o a $false$, entonces se evalúan estas condiciones en tiempo de ejecución para comprobar si es posible una evaluación paralela de los grafos Γ_1 y Γ_2.

   Antes de dar la especificación formal de la regla *split*, introducimos la función de evaluación de condiciones *con* de un conjunto χ de condiciones simples bajo un contexto C.

$$con(\chi, C) = \bigwedge_{c_i \in \chi} eval(c_i, C)$$

   La especificación de la regla *split* es la siguiente:

$$split(\Xi, \Gamma_1, \Gamma_2) = \begin{cases} parA_1 =< \Gamma_1, C > A_2 =< \Gamma_2, C > \\ \quad \text{si } \neg\exists < u, c, v > \in E \text{ tal que } u \in V_1, v \in V_2 \\ seqA_1 =< \Gamma_1, C > A_2 =< \Gamma_2, post(C, \Gamma_1) > \\ \quad \text{si } con(\chi, C) = false \\ cpar\chi A_1 =< \Gamma_1, C > A_2 =< \Gamma_2, post(C, \Gamma_1) > \\ \quad \text{e.o.c.} \end{cases}$$

   Donde:

$$\Xi =< \Gamma, C >, \Gamma =< V, E >, \Gamma_1 =< V_1, E_1 >, \Gamma_2 =< V_2, E_2 >$$

$$V = V_1 \cup V_2, V_1 \cap V_2 = \emptyset, \Gamma_1 = \Gamma - V_2$$

[25] Esto es, $\bigwedge_{<u,c,v> \in E_2} eval(c, C) = false$, donde $\Gamma_2 =< V_2, E_2 >$.

χ es el conjunto de cada una de las condiciones simples que etiquetan los arcos de Γ_1 a Γ_2:

$\chi = \{c \text{ tal que } < u, c, v > \in E, u \in V_1, v \in V_2\}$

A_1 y A_2 son variables auxiliares nuevas y distintas.

La aplicación de esta regla de transformación sobre un grafo produce el particionamiento en dos subgrafos, por lo que el tamaño de la expresión resultante no aumenta. Los subgrafos están relacionados con la constructora *seq*, *par* o *cpar*, que expresan su evaluación secuencial, paralela o condicional respectivamente.

2. *If.*

Esta regla genera una expresión con dos grafos especializados bajo la suposición de que una determinada condición se evalúe en tiempo de ejecución a *true* o a *false*. Se aplica sobre un grafo y una condición simple, generando una expresión if cuyos argumentos son: la condición simple, el subgrafo que se deriva de asumir cierta la condición y el subgrafo derivado de asumir falsa la condición.

La especificación formal de la regla if es la siguiente:

$$if(c, \Xi) = (if\ c\ \ then < simplify(\Gamma, C_t), C_t > \\ else < simplify(\Gamma, C_f), C_f >)$$

Donde:

$\Xi =< \Gamma, C >, C_t = ext(\Xi, c), C_f = ext(\Xi, \neg c)$

La función $ext(\Xi, c)$ devuelve el contexto C de Ξ, extendido con los hechos que se pueden derivar de que la condición c se satisfaga. Su especificación formal es la siguiente:

$ext(\Xi, c) =$

$$\begin{cases} \{i(X,Y)\} \cup C & \text{si } c = i(X,Y) \\ \{i(X,X)\} \cup \{i(X,v)\}_{\forall v \in vars(\Gamma)} \cup C & \text{si } c = i(X,X) \\ \{d(X,Y), d(X,X), d(Y,Y)\} \cup C & \text{si } c = \neg i(X,Y) \\ \{d(X,X)\} \cup \{d(X,v)\}_{\forall v \in vars(\Gamma) \text{ tal que } i(X,v) \notin C} \cup C & \text{si } c = \neg i(X,X) \end{cases}$$

La aplicación de la regla if genera expresiones en donde, además de incluir la constructora if, se generan expresiones para el grafo bajo dos asunciones diferentes, esto es, duplica el tamaño del código.

2.2.2 Estrategias de transformación

En esta sección se presentan tres estrategias de transformación. En el diseño de las estrategias hemos considerado dos factores opuestos:

- Explotar el máximo paralelismo.
- Limitar el coste de su detección en tiempo de ejecución.

Para explotar el m'aximo paralelismo es necesario incorporar en las reglas paralelas condiciones de independencia costosas de evaluar en tiempo de ejecuci'on. Por ello es necesario limitar esta sobrecarga al tiempo de ejecuci'on para no degradar el rendimiento, a costa de reducir las oportunidades de paralelismo. En base a estos dos factores se desarrollan tres estrategias:

1. *Paralelismo incondicional.* Mediante esta estrategia sólo se permite la explotación paralela de *PEUs* cuya independencia se conozca en tiempo de compilación. Por lo tanto, las *EGEs* que se generan con esta estrategia no incorporan condiciones de independencia, por lo que no se invierte ningún esfuerzo en tiempo de ejecución en la detección de paralelismo. Esta estrategia prima el segundo factor, sin tener en cuenta el primero.

2. *Paralelismo máximo.* Con esta estrategia se busca la expresión de todas las oportunidades de paralelismo presentes en un determinado *CDG*, sin importar el número de condiciones en tiempo de ejecución que sea necesario para ello. Esta estrategia sólo tiene en cuenta el primer factor.

3. *Paralelismo básico.* Con esta estrategia se trata de detectar en tiempo de ejecución las oportunidades de paralelismo sólo con condiciones de basicidad. En general, el coste de identificación de paralelismo es inferior al de la segunda estrategia y superior al de la primera, detectándose menores oportunidades de paralelismo que en la segunda pero mayores que en la primera. Esta estrategia supone un compromiso entre los dos factores.

Antes de presentar la especificación formal de las estrategias, introduciremos un paso intermedio para la identificación del paralelismo incondicional que se puede detectar en un determinado *ECDG*. Con este paso se identifican aquellos nodos cuya dependencia o independencia se conoce en tiempo de compilación, por lo que deben evaluarse secuencialmente o pueden evaluarse en paralelo, respectivamente.

Paso de identificación de paralelismo incondicional

Los componentes conexos de un determinado *ECDG* Ξ se pueden evaluar en paralelo puesto que no hay arcos entre ellos. Denominamos U al conjunto de nodos u_i con arcos de salida insatisfactibles bajo un contexto C[26] y N al conjunto de nodos a los que llegan arcos de nodos pertencientes a U. El conjunto de nodos U se debe evaluar después de los nodos de N. Los nodos u_i se pueden evaluar en paralelo puesto que no hay dependencias entre ellos. Sin embargo, algunos arcos entre U y N se pueden satisfacer después de la evaluación de algunos nodos de U y antes de la evaluación de todos ellos. Por lo tanto, es posible que los nodos de N, a los que llegan los arcos que se pueden haber satisfecho, deban esperar innecesariamente a la evaluación del resto de los nodos de U. Esta es una consecuencia de la agrupación de nodos, que se ha descrito en el modelo *RAP* [50] para programación lógica.

Este paso de identificación de paralelismo incondicional se aplica recursivamente a todos los conjuntos N de un *ECDG*. Este procedimiento finaliza cuando los arcos que parten de un conjunto N no se pueden evaluar a $false$, i.e., es posible que se puedan evaluar a *true* y se permita una evaluación paralela. Después, se aplica la estrategia elegida en la compilación. A continuación ofrecemos la especificación del paso de identificación de paralelismo incondicional (*ups*), aplicado a un *ECDG* Ξ, bajo una estrategia h.

$$ups(\Xi, h) = \begin{cases} v & \text{si } V = v \\ \varepsilon_1 & \text{si } \Gamma \text{ sólo tiene un componente conexo} \\ par(\varepsilon_1, \ldots, \varepsilon_n) & \text{e.o.c.} \end{cases}$$

La aplicación de la regla *split* a componentes conexos de Ξ resulta en una expresión *par* tal que:

$split(\Xi, \Gamma_1, \ldots, \Gamma_n) = (par\ \Xi'_1 \ldots \Xi'_n)$, donde Γ_i es un componente conexo de Ξ.

El resultado que *ups* devuelve como argumentos de la expresión *par* son las *EGE's* ε_i que se calculan para las *ECDGs* $\Xi'_1, \ldots, \Xi'_n$.

Dado $\Xi'_i =< \Gamma'_i, C'_i >$, calculamos ε_i como:

$$\varepsilon_i = \begin{cases} v_i & \text{si } \Gamma'_i \text{ contiene un único nodo } v_i \\ (seq\ \varepsilon_{i1}\ \varepsilon_{i2}) & \text{si } con(\chi, C'_i) = false \\ h(\Xi'_i) & \text{e.o.c.} \end{cases}$$

Si las condiciones que etiquetan los arcos que salen de Γ_{i1} son potencialmente satisfactibles bajo C'_i[27], entonces se aplica la estrategia h sobre el

[26] Decimos que un conjunto χ de condiciones simples es insatisfactible bajo un contexto C si $con(\Xi, C) = false$.

[27] Decimos que un conjunto de condiciones χ es potencialmente satisfactible bajo un

ECDG Ξ_i, que corresponde al último caso de la especificación de ε_i.

La expresión *seq* resulta de la aplicación de la regla *split* sobre Ξ_i y dos subgrafos Γ_{i1} y Γ_{i2}. Γ_{i1} es el grafo que contiene los nodos a los que no entran arcos y Γ_{i2} es el grafo que contiene el resto de nodos (que expresamos como diferencia de grafos sobrecargando el operador '-' como $\Gamma_{i2} = \Gamma - \Gamma_{i1}$). Los argumentos de la expresión *seq* son las *EGEs* ε_{i1} y ε_{i2}. Estas *EGEs* se calculan a partir de las *ECDGs* Ξ'_{i1} y Ξ'_{i2} respectivamente. Así:

$split(\Xi_i, \Gamma_{i1}, \Gamma_{i2}) = (seq\ \Xi'_{i1}\ \Xi'_{i2})$, donde:

$\Gamma_i =< V_i, E_i >$

$\Gamma_{i1} =< V_{i1}, \{\} >$, $V_{i1} = \{v \in V_i \text{ tal que } < u, c, v > \notin E_i\}$

$\Gamma_{i1} =< V_{i1}, E_{i2} >$ $\begin{cases} V_{i2} = V_i - V_{i1} \\ E_{i2} = \{e \text{ tal que } e \in E_i, e \neq < u, c, v >, \forall u \in V_{i1}\} \end{cases}$

$\chi = \{c \text{ tal que } e \in E_i, e =< u, c, v >, u \in V_{i1}\}$

ε_{i1} se calcula aplicando la regla *split* a los subgrafos formados por un único nodo. Dará como resultado una expresión *par* cuyos argumentos serán cada uno de tales subgrafos, esto es:

$\varepsilon_{i1} = split(\Xi'_{i1}, \Gamma''_{i1}, \ldots, \Gamma''_{in})$, donde:

$\Gamma''_{ij} =< \{v_j\}, C'_{i1} >$ tal que $v_j \in V'_{i1}$

ε_{i2} i se calcula aplicando el paso de identificación de paralelismo incondicional al *ECDG* Ξ'_{i2} como se especifica a continuación:

$\varepsilon_{i2} = ups(\Xi'_{i2}, h)$

Descripción de las estrategias de transformación

Una estrategia h es una función que devuelve una *EGE* ε a partir de un *ECDG* $< \Gamma, C >$, esto es, $\varepsilon = h(< \Gamma, C >)$. Para calcular la *EGE* aplicando la estrategia h se realiza una llamada al paso de identificación de paralelismo bajo las *directivas* h_s de la estrategia. h_s representa la función que implementa el comportamiento de una determinada estrategia. Denotamos s como *up* para la estrategia de paralelismo incondicional, *mp* para la estrategia de paralelismo máximo y *gp* para la estrategia de paralelismo básico. La especificación formal de una estrategia s sobre un *ECDG* Ξ es en general $s(\Xi) = ups(\Xi, h_s)$. A continuación presentamos las tres estrategias.

1. *Estrategia de paralelismo incondicional (up).*

   Esta estrategia omite las condiciones que etiquetan los arcos del *CDG*, de manera que se realice una planificación de paralelismo totalmente

contexto C si $con(\chi, C) \neq false$.

estática. Así, se elimina cualquier test de independencia en tiempo de ejecución. Esta estrategia puede considerarse como una instancia del paso de identificación de paralelismo incondicional, cuando las condiciones potencialmente satisfactibles que etiquetan los arcos son insatisfactibles. Esta estrategia se puede implementar como *ups* cuando la función *con* devuelve siempre como resultado $false$[28].

La especificación formal de la estrategia es la siguiente:

$up(\Xi) = ups(\Xi, h_{up})$

$h_{up}(\Xi) = (seq\ \varepsilon_1\ \varepsilon_2)$

La expresión *seq* es el resultado de aplicar la regla *split* sobre dos subgrafos de un grafo Ξ'. El grafo Ξ' es el resultado de la transformación de Ξ, donde las condiciones que no se pueden verificar en tiempo de compilación se sustituyen por $false$. Γ_1 es el grafo compuesto por los nodos de Ξ sin arcos de entrada y $\Gamma_2 = \Gamma - \Gamma_1$.

$split(\Xi', \Gamma_1, \Gamma_2) = (seq\ \Xi'_1\ \Xi'_2)$

$\Xi =< \Gamma, C >, \Gamma =< V, E >$

$\Xi_1 =< \Gamma_1, C_1 >$

$$\Gamma_1 =< V_1, E_1 > \begin{cases} V_1 = \{v_i \text{ tal que } < v_j, c_{ji}, v_i > \notin E, \forall v_j \in V\} \\ E_1 = \{\} \end{cases}$$

$$\Xi_2 =< \Gamma_2, C_2 >, \Gamma_2 =< V_2, E_2 > \begin{cases} V_2 = V - V_1 \\ E_2 = \{< v, c, v > \text{ tal que } v \notin V\} \end{cases}$$

$\Xi' =< \Gamma', C >$

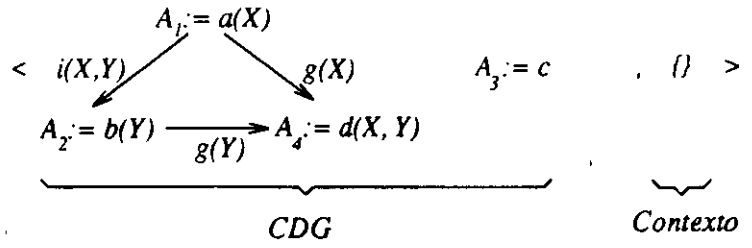
$\Gamma' =< V', E' >$

$$\begin{cases} V' = & V \\ E' = & \{< v_i, c_{ij}, v_j > \text{ tal que } v_i \notin V_1, < v_i, c_{ij}, v_j > \in E\} \cup \\ & \{< v_i, false, v_j > \text{ tal que } v_i \in V_1, v_j \in V_2, < v_i, c_{ij}, v_j > \in V\} \end{cases}$$

Los argumentos ε_1 y ε_2 de la expresión *seq* son el resultado de la aplicación recursiva de *ups* sobre las *ECDGs* que son resultado de la aplicación de la regla *split* bajo la estrategia *up*. Esto es:

$\varepsilon_1 = ups(\Xi'_1, h_{up})$, $\varepsilon_2 = ups(\Xi'_2, h_{up})$.

[28] Nótese que esto no es equivalente a la sustitución de las condiciones del *CDG* por $false$. En general, las variables aumentan su grado de instanciación cuando se evalúan las diferentes *PEUs* que las contienen, por lo que es posible que algunas *PEUs*, condicionalmente independientes, se transformen en independientes.



a, b, c y d funciones estrictas
u constructora de datos
X e Y variables lógicas

Figura 2.4: *ECDG* correspondiente a la regla $h(X,Y) := u(a(X), b(Y), c, d(X,Y))$.

Ejemplo 6. *Aplicación de la estrategia up.*
La estrategia aplicada al *ECDG* de la figura 2.4 devuelve:

$(par\ A_3 := c$
$\quad (seq\ A_1 := a(X)\ A_2 := b(Y)\ A_4 := d(X,Y)))$

2. *Estrategia de paralelismo máximo (mp)*

   El algoritmo que implementa esta estrategia se basa en la simplificación sistemática de los arcos con condiciones potencialmentes satisfactibles. La simplificación consiste en la aplicación de la regla if, que devuelve dos subgrafos, uno asumiendo la condición de if como cierta y otro como falsa. Cada aplicación de la regla if tiene como argumento una de las condiciones potencialmente satisfactibles. Las condiciones se eligen en el orden natural, esto es, desde las que etiquetan los arcos saliendo de nodos más a la izquierda, a las que etiquetan los arcos saliendo de nodos más a la derecha. La especificación formal de esta estrategia es la siguiente:

   $mp(\Xi) = ups(\Xi, h_{mp})$, donde

   $h_{mp}(\Xi) = (if\ c_{ik}\ \varepsilon_t\ \varepsilon_f)$

   Esta expresión if es el resultado de aplicar la regla if a la condición c_{ik} y el *ECDG* Ξ.

   $if(c_{ik}, \Xi) = (if\ c_{ik}\ \Xi'_t\ \Xi'_f)$, donde:

   $\Xi =< \Gamma, C >$, $\Gamma =< V, E >$

La condición c_{ik}, potencialmente satisfactible, etiqueta un arco de Γ tal que:

$i = min(\{ord(\Gamma, v_j)\}_{j=1,\ldots,n})$ tal que $< v_i, c_{ik}, v_k > \in E, eval(c_{ik}, C) \neq false$[29]

La función *ord* calcula la posición del nodo v en el orden natural de Γ. La función *min* calcula el número mínimo de un conjunto de enteros diferentes. Las expresiones ε_t y ε_f son el resultado de la aplicación de *ups* sobre los argumentos de la expresión if bajo la estrategia *mp*. Esto es:

$\varepsilon_t = ups(\Xi'_t, h_{mp})$, $\varepsilon_f = ups(\Xi'_f, h_{mp})$

Ejemplo 7. *Aplicación de la estrategia mp.*
La *EGE* que devuelve la aplicación de esta estrategia sobre el *ECDG* de la figura 2.4 es:

```
(par A3 := c
     (if i(X,Y)
         (if i(X,X)
             (if i(Y,Y)
                 (par A1 := a(X) A2 := b(Y) A4 := d(X,Y))
                 (par  A1 := a(X)
                       (seq A2 := b(Y) A4 := d(X,Y)))
             (if i(Y,Y)
                 (par (seq A1 := a(X) A4 := d(X,Y))
                       A2 := b(Y))
                 (seq (par A1 := a(X) A2 := b(Y))
                       A4 := d(X,Y))))
         (seq A1 := a(X)
              (if i(Y,Y)
                  (par A2 := b(Y) A4 := d(X,Y))
                  (seq A2 := b(Y) A4 := d(X,Y))))))
```

3. *Estrategia de paralelismo básico (gp)*

   En esta estrategia se eligen las condiciones de basicidad que proporcionan una mayor simplificación del grafo, i.e., se elimina el mayor

[29] No hay ningúna condición para $eval = true$ en la elección de i, ya que el grafo está simplificado.

número posible de arcos. Cuando el paso de identificación de paralelismo incondicional finaliza, la estrategia *gp* busca la variable X que cumple lo siguiente:

- X es la variable con el mayor número de apariciones en las condiciones que etiquetan los arcos del grafo.
- $i(X,X)$ debe ser consistente con el contexto.

Una vez seleccionada la variable X, la estrategia *gp* aplica la regla if con la condición de basicidad sobre X. El paso de identificación de paralelismo incondicional se aplica recursivamente bajo *gp*.

La especificación formal de la estrategia *gp* es la siguiente:

$gp(\Xi) = ups(\Xi, h_{gp})$, donde

$h_{gp}(\Xi) = (if\ i(X,X)\ \varepsilon_t\ \varepsilon_f)$

$if(i(X,X),\Xi) = (if\ i(X,X)\ \Xi'_t\ \Xi'_f)$

$\Xi =< \Gamma, C >$, $\Gamma =< V, E >$

La función occ_c calcula el número de apariciones de la variable X en la condición c.

$$occ_c(X,c) = \begin{cases} 0 & \text{si } c = i(Y,Z) \vee c = false \\ 1 & \text{si } c = i(X,Y) \\ 2 & \text{si } c = i(X,X) \end{cases}$$

La función occ_E calcula el número de apariciones de la variable X en el conjunto de arcos E.

$occ_E(X,E) = \sum_{c_i \in \chi} occ_c(X,c_i)$

tal que:

- $\chi = \{c_i \text{ tal que } < v_k, c, c_l > \in E \text{ y } v_k, v_l \in V\}$
- $occ_E(X,E)$ es un maximal[30] de:

$$\mathcal{O} = \{occ_E(X_i,E)\}_{\forall X_i \text{ tal que } <v_k,i(X_i,X_j),v_l> \in E, v_k,v_l \in V, \text{ tal que } eval(i(X,X),\Xi) \neq false.}$$

Las expresiones ε_t y ε_f son el resultado de la aplicación del paso de identificación de paralelismo incondicional sobre los subgrafos Ξ''_t y Ξ''_f

[30] En general habrá variables con el mismo número de apariciones. Una de ellas se elige arbitrariamente.

en los que la condición sobre la que se aplica la regla if se asume como un hecho cierto o falso, respectivamente.

$\varepsilon_t = ups(\Xi''_t, h_{gp})$, $\varepsilon_f = ups(\Xi''_f, h_{gp})$

Donde:

$\mathcal{D} = \{c\}_{\forall c=<u,i(Y,Z),v>\in E \text{ tal que } eval(i(Y,Y),\Xi)=false \wedge occ_E(X,E)\geq occ_E(X,E)}$

$\Xi''_t =< \Gamma'_t, C'_t \cup \{ext(\Xi'_t, \neg c)\}_{\forall c\in\mathcal{D}} >$

$\Xi''_f =< \Gamma'_f, C'_f \cup \{ext(\Xi'_f, \neg c)\}_{\forall c\in\mathcal{D}} >$

Ejemplo 8. *Aplicación de la estrategia gp.*

La aplicación de la estrategia sobre el *ECDG* de la figura 2.4 arroja la siguiente *EGE*:

$$
\begin{array}{l}
(par\ A_3 := c \\
\quad (if\ i(X,X) \\
\quad\quad (if\ i(Y,Y) \\
\quad\quad\quad (par\ A_1 := a(X)\ A_2 := b(Y)\ A_4 := d(X,Y)) \\
\quad\quad\quad (par\ A_1 := a(X)\ (seq\ A_2 := b(Y)\ A_4 := d(X,Y))) \\
\quad\quad (if\ i(Y,Y) \\
\quad\quad\quad (par\ A_2 := b(Y)\ (seq\ A_1 := a(X)\ A_4 := d(X,Y))) \\
\quad\quad (seq\ A_1 := a(X) \\
\quad\quad\quad (if\ i(Y,Y) \\
\quad\quad\quad\quad (par\ A_2 := b(Y)\ A_4 := d(X,Y)) \\
\quad\quad\quad\quad (seq\ A_2 := b(Y)\ A_4 := d(X,Y)))))))
\end{array}
$$

En el marco de la programación lógica se encuentra un trabajo relacionado [122] en donde se proponen diferentes estrategias para explotar el paralelismo conjuntivo de Prolog. Una de ellas está directamente relacionada con *up*, que busca el paralelismo incondicional entre objetivos de una cláusula. Otra de ellas explota paralelismo por conjunto de objetivos tal que si se demuestra la independencia entre conjuntos se realiza una ejecución paralela y en caso contrario, secuencial. Sólo una de ellas tiene en cuenta en tiempo de compilación la información derivada de la ejecución de los objetivos.

2.2.3 Transformación de reglas secuenciales Babel en paralelas

En esta sección presentamos cómo se integran las *EGEs*, que expresan el paralelismo de las *PEUs*, en reglas Babel paralelas. En primer lugar consideraremos las reglas Babel que contengan sólo funciones estrictas y después veremos las reglas con funciones predefinidas no estrictas.

Reglas paralelas para funciones estrictas

La identificación de paralelismo presentada en las secciones anteriores atañe a la parte derecha (rhs) de una regla Babel $lhs := rhs.$. Tras el cálculo de su EGE se calcula la rhs paralela ($prhs$) como:

$$prhs = \begin{cases} t & \text{si } rhs = t \in DT_\Sigma \cup Var \\ let\ Ege_h(rhs)\ in\ Body & \text{e.o.c.} \end{cases}, \text{ donde}$$

$$Body = \begin{cases} f(E_1, \ldots, E_n) & \text{si } rhs \text{ es la aplicación de una función f} \\ c(E_1, \ldots, E_n) & \text{si } rhs \text{ es la aplicación de una constructora c} \end{cases}$$

y $E_i \in Var \cup AVar \cup DT_\Sigma$

DT_Σ es el conjunto de términos de datos de la signatura del programa al que pertenece la regla a traducir. *Body* se construye sustituyendo los argumentos funcionales de la rhs por las variables auxiliares correspondientes que están definidas en $Ege_h(rhs)$. Como argumentos tiene términos de datos, variables lógicas o variables lógicas auxiliares. $Ege_h(E)$ es la función que devuelve una EGE para la expresión E bajo la estrategia h.

Ejemplo 9. *Posibles reglas Babel paralelas para el ejemplo de la figura 2.4.*

$$\begin{array}{l} h(X,Y) := let\ (par\ A_3 := c \\ \qquad\qquad (if\ i(X,X)\ (par\ A_1 := a(X) \\ \qquad\qquad\qquad\qquad (seq\ A_2 := b(Y)\ A_4 := d(X,Y)))) \\ \qquad\qquad\qquad\quad (seq\ A_1 := a(X) \\ \qquad\qquad\qquad\qquad A_2 := b(Y) \\ \qquad\qquad\qquad\qquad A_4 := d(X,Y)))) \\ \qquad in\ f(A_1, A_2, A_3, A_4). \end{array}$$

$$\begin{array}{l} h(X,Y) := let\ (par\ A_3 := c \\ \qquad\qquad (cpar\ \ i(X,X) \wedge i(Y,Y) \\ \qquad\qquad\qquad\ A_1 := a(X) \end{array}$$

$$A_2 := b(Y)$$
$$A_4 := d(X, Y)))))$$
$$in\ f(A_1, A_2, A_3, A_4).$$

Reglas paralelas para funciones predefinidas no estrictas

Para manejar las funciones predefinidas no estrictas que aparezcan en el programa Babel, realizamos una transformación del programa de modo que dichas funciones aparezcan sólo como símbolo raíz de las partes derecha de las reglas. Con esta transformación previa, aplicaremos una estrategia determinada a los argumentos de las funciones no estrictas, de manera que mantenemos encapsulada la identificación de paralelismo a los argumentos estrictos.

La especificación formal de la transformación, que denotamos como función de traducción $\mathcal{S}$, se puede encontrar en el apéndice C. Aquí, introducimos $\mathcal{S}$ mediante un ejemplo.

Ejemplo 10. *Traducción de un programa fuente bajo* $\mathcal{S}$
$h(X, Y) := (a(X) \wedge (b(Y) \vee c)) \rightarrow (d(X \rightarrow t \square f)$.
se transforma en:
$h(X, Y) := b_1(X, Y) \rightarrow e_1(X)$.
$b_1(X, Y) := a(X) \wedge b_2(Y)$.
$b_2(Y) := b(Y) \vee c$.
$e_1(X) := d(X) \rightarrow t \square f$.

A continuación presentamos las diferentes funciones predefinidas no estrictas para las que admitimos una explotación de paralelismo de sus argumentos.

1. *Función lógica conjunción* (b_1, b_2).

   Admitimos la evaluación paralela de b_1 y b_2 si se prueba que son independientes. Las condiciones de independencia entre b_1 y b_2 se anotan en tiempo de compilación. Si se prueban independientes, la condiciones de independencia se evaluan a *true*.

   La regla $L := (b_1, b_2)$. se transforma en:

   $L := and\ \chi\ Ege_h(Ecdg(b_1), \{\})\ Ege_h(Ecdg(b_2), post(Ecdg(b_1), \{\})$

   Donde χ es la conjunción de condiciones simples entre b_1 y b_2, $Ecdg(Exp)$ es la función que devuelve el $ECDG$ correspondiente a la expresión Exp y $Ege_h(\Xi, C)$ es la función que devuelve la EGE correspondiente al $ECDG$ Ξ extendido con el contexto C bajo la estrategia h.

2. *Función lógica disyunción* $(b_1; b_2)$.

   Admitimos la evaluación paralela de b_1 y b_2 si se prueba que son independientes. Las condiciones de independencia entre b_1 y b_2 se anotan en tiempo de compilación.

   La regla $L := (b_1; b_2)$. se transforma en:

   $L := or\ \chi\ Ege_h(Ecdg(b_1), \{\})\ Ege_h(Ecdg(b_2), post(Ecdg(b_1), \{\})$

   Donde χ es la conjunción de condiciones simples entre b_1 y b_2.

El comportamiento operacional de estas funciones es el definido en la sección 2.1.

Esta traducción se incluye en el esquema S dando lugar al nuevo esquema S^* que se puede consultar en la sección C.1 del apéndice C.

2.3 Medida de rendimiento del sistema de paralelización automática

En esta sección realizaremos una medida del rendimiento del procedimiento de paralelización independiente de la implementación, en el mismo sentido que se realiza en [70, 145, 78, 146, 33] en programación lógica. En el capítulo 5 realizaremos un estudio del rendimiento del sistema teniendo en cuenta los mecanismos de implementación[31].

La medida de rendimiento del sistema de identificación de paralelismo se realiza en función de los siguientes parámetros.

1. Ganancia de velocidad (*speed-up*).
   Es el factor más importante que determina la eficiencia global de la explotación de paralelismo.

2. Número de condiciones satisfechas frente al número de condiciones probadas.
   Es un parámetro más concreto que refleja el rendimiento de las *EGEs* en la identificación de paralelismo en tiempo de ejecución.

3. Costo de la identificación de paralelismo.
   Es el tiempo empleado en la evaluación de las condiciones de independencia. Este parámetro da una medida del esfuerzo necesario para la identificación de paralelismo en tiempo de ejecución.

[31] Considerando factores como el diseño del bus del sistema multiprocesador, las políticas de coherencia de memoria caché, etc.

2.3.1 Simulador

Hemos diseñado un simulador de alto nivel que proporciona las medidas de los parámetros considerados. No damos cuenta de la sobrecarga de tiempo debida a la planificación paralela de procesos o del reclamo de procesos en caso de fallo. Asumimos también que disponemos de un número ilimitado de procesadores para estimar cuál sería la máxima ganancia en la evaluación paralela.

Implementación de la evaluación paralela

Hemos implementado el simulador en Prolog, cediendo el control de la unificación y del *backtracking* al propio sistema Prolog. El mecanismo de *backtracking* de Babel se comporta de manera análoga a Prolog. La prueba de diferentes alternativas (reglas) para la evaluación de una función se corresponde con la prueba de las diferentes cláusulas para la resolución de un predicado en su orden textual. Por otra parte, la unificación en Babel como reconocimiento de patrones y vinculación de variables lógicas es idéntica a Prolog.

La resolución de un objetivo en Prolog devuelve *true* y una sustitución de éxito, o bien falla. La evaluación en Babel de primer orden de una expresión devuelve un término de datos y una sustitución de éxito, o bien falla. Un cómputo de estrechamiento de una expresión E devuelve:

- E, si E es una variable o una constante,
- una constructora c con sus argumentos reducidos, si $E = c(E_1, \ldots, E_n)$, o
- el cuerpo reducido de la regla cuya cabeza unifica con $f(E'_1, \ldots, E'_n)$, donde E'_i $(1 \leq i \leq n)$ son los argumentos reducidos de E, si $E = f(E_1, \ldots, E_n)$.

Un simulador secuencial de Babel impaciente de primer orden debe recoger el resultado de la evaluación de las expresiones y la sustitución de éxito. La entrada al simulador será un programa Babel y una expresión a reducir. Escribimos en Prolog el procedimiento `evaluate/2` cuyas cláusulas implementan cada uno de los casos de un paso de narrowing. La cláusula que representa el paso general del procedimiento de evaluación es:

```
evaluate(Expression, Term) :-
```

```
Expression =.. [Function|Arguments],
function(Function),
        % Test of function
evaluate_arguments(Arguments, Evaluated_Arguments),
        % Evaluation of function's arguments
Head =.. [Function|Evaluated_Arguments],
rule(Head, Body),
        % Function call
evaluate(Body, Term).
        % Recursive evaluation of the body
```

Las funciones y constructoras de datos se deben declarar como tales en el fuente Babel, i.e., la declaración de tipos, para que un objetivo como `function/1` identifique que su argumento es una función antes de realizar la aplicación de la función. El procedimiento `evaluate_arguments/2` evalúa de forma impaciente los argumentos de la función. A continuación se construye la aplicación de la función con el operador *univ* (`=..`) para encontrar la regla que unifique con ella. Las reglas Babel están declaradas en el procedimiento `rule/2`. El primer argumento de este procedimiento representa la cabeza de la regla y el segundo representa el cuerpo. La unificación de la expresión Babel se realiza de forma transparente en la resolución del objetivo `rule(Head, Body)`. Prolog guarda los puntos de elección correspondientes a la llamada al procedimiento `rule/2`, por lo que en caso de fallo en cualquier punto del programa simulado, se prueban las diferentes alternativas de las funciones.

Hemos construido el simulador de la evaluación paralela incorporando nuevas cláusulas de evaluación que manejan las constructoras paralelas.

```
evaluate((let EGE in Exp), Term) :-
        % Test of the new constructor let - in -
    execute(EGE),
        % Execution of the EGE
    evaluate(Exp, Term).
        % Evaluation of the expression Exp
```

Las variables auxiliares se representan con variables lógicas Prolog. En ellas se almacenan las subexpresiones evaluadas de la *rhs* de una regla definidas en la *EGE*. La cláusula Prolog que se ocupa de la evaluación de una *PEU* es la siguiente:

```
execute(AVar := Expression) :-
        evaluate(Expression, AVar).
```

Los argumentos de las expresiones *seq*, *par* y *cpar* se representan con listas. La implementación correspondiente a la ejecución de una expresión *par* corresponde a las siguientes cláusulas:

```
execute(par(List_of_EGE)) :-
        par_execution(List_of_EGE).

par_execution([]).
par_execution([EGE|EGEs]) :-
        execute(EGE),
        par_execution(EGEs).
```

Implementación de la medida de los parámetros

1. Ganancia de velocidad.
   Para medir este parámetro elegimos como unidad de tiempo un paso de estrechamiento. El número de pasos de estrechamiento necesarios para un cómputo da una medida estimativa del tiempo de cómputo, sólo a efectos de comparación entre los sistemas secuencial y paralelo. La elección de esta medida de tiempo es razonable en cuanto que se pretende efectuar una medida *relativa* del tiempo de ejecución entre dos sistemas, en lugar de una absoluta, para lo cual se debería seguir una aproximación como la propuesta en el capítulo 4. En [70] se utiliza de manera análoga un paso de resolución para realizar un análisis de alto nivel del rendimiento de &-Prolog [68].

   > **Ejemplo 11.** *Tiempo de reducción de una expresión.* La expresión $media(2,4)$, con la siguiente definición de función:
   > $media(X,Y) := (X+Y)/2$.
   > se evalúa en tres pasos de estrechamiento, es decir, asociamos tres unidades de tiempo como tiempo de cómputo de la evaluación de esta expresión.
   > $media(2,4) \Longrightarrow_\emptyset (2+4)/2 \Longrightarrow_\emptyset 6/2 \Longrightarrow_\emptyset 3$

   Para medir el tiempo de evaluación de una expresión en el caso secuencial basta con incrementar el tiempo en cada paso de estrechamiento. En el caso del simulador paralelo hay que contabilizar el tiempo de

forma diferente. Cuando ocurre una planificación paralela de varios procesos, el tiempo de evaluación corresponderá al tiempo máximo de tales procesos. Utilizamos las siguientes variables internas del simulador para gestionar la medida del tiempo:

- *Tiempo secuencial.* Tiempo empleado en la evaluación secuencial.
- *Tiempo paralelo.* Tiempo empleado en la evaluación paralela.
- *Tiempo base.* Instante en el que ocurre una planificación paralela de procesos.
- *Tiempo privado.* Tiempo de un proceso paralelo relativo a su tiempo base.

Para cada planificación paralela (en presencia de una expresión *par* o *cpar*) fijamos un tiempo base como el tiempo paralelo actual. Sobre el tiempo base se mide el tiempo privado de cada proceso paralelo. El tiempo paralelo se incrementa en cada paso de narrowing de la ejecución de un proceso paralelo y se sitúa en el tiempo base cuando termina el proceso. Cuando todos los procesos paralelos finalizan, el tiempo paralelo se actualiza con el tiempo base más el máximo de los tiempos privados y el tiempo privado del proceso padre se actualiza sumándole el máximo de los tiempos privados.

Las variables internas están implementadas como términos de la base de datos Prolog para mantener su valor en caso de *backtracking*. Estos términos están identificados con una referencia a cada planificación paralela. Las referencias se pasan como un argumento más de los procedimientos de evaluación de expresiones y de ejecución de *EGEs*. El procedimiento `par_execution` con gestión de tiempos es:

```
par_execution(List_of_EGE, Ref) :-
        upd_fork_time(Ref, NewRef),
                % Computes the base time, resets
                % the private time and the maximal
                % private time for a new reference
        xpar_execution(List_of_EGE, NewRef),
        upd_join_time(Ref, NewRef).
                % Updates the father process' private time
                % and the parallel time
```

```
xpar_execution([], _).
xpar_execution([EGE|EGEs], Ref) :-
        reset_priv_time(Ref),
                % Resets the private time
        execute(EGE, Ref),
        keep_max_priv_time(Ref),
                % Keeps the maximal private time
        set_par_time(Ref),
                % Sets the parallel time to the base time
        xpar_execution(EGEs, Ref).
```

2. Número de condiciones satisfechas.
   Se contabilizan con contadores implementados con predicados dinámicos.

3. Costo de la identificación de paralelismo.
   Para realizar la medida de este parámetro daremos una medida del coste necesario para evaluar las condiciones de independencia, i.e., condiciones simples de basicidad de una variable y de independencia entre dos variables. La evaluación de las condiciones de independencia y basicidad se realiza recursivamente sobre la estructura de los términos de datos, de manera que las condiciones sean seguras, no conservadoras como en [70], donde se propone una evaluación de los términos de datos hasta una determinada profundidad[32]. Para evaluar una condición de basicidad de una variable visitamos cada nodo del término de datos vinculado a la variable, comprobando su carácter básico. Si todos los nodos son básicos, entonces el término es básico. El cálculo del tiempo de evaluación de una condición de basicidad lo realizamos incrementando el tiempo de evaluación en cada nodo explorado de la estructura. Para evaluar una condición de independencia exploramos dos términos para extraer sus correspondientes conjuntos de variables y comprobamos que los conjuntos son disjuntos. El tiempo de evaluación de una condición de independencia se incrementa al explorar cada nodo de los términos y cada vez que una variable de un conjunto se compara con otra.

En la próxima sección, presentaremos los resultados que se obtienen al analizar programas Babel paralelos con el simulador.

[32] Si el término explorado tiene una profundidad superior a una predeterminada, el término se considera potencialmente dependiente (aunque no lo sea realmente). La ventaja de esta propuesta se pone de manifiesto cuando se evalúan términos de gran profundidad.

2.3.2 Resultados

Hemos seleccionado varios programas de prueba para analizarlos con el simulador[33]. Cada uno de ellos se ha compilado con las estrategias *up*, *mp* y *gp*. Hemos simulado y comparado diferentes versiones de los programas: la secuencial, la paralela y la paralela con anotación de información de independencia[34]. Se ha incluido también la versión paralelizada a mano de cada programa, es decir, con la máxima información de independencia que en tiempo de compilación se podría inferir. A pesar de que el procedimiento de compilación admite incorporación de información de granularidad, ésta no se ha incluido puesto que el simulador no incorpora una medida precisa del tiempo empleado en la gestión de la explotación de paralelismo. Los programas paralelos, resultado de la compilación bajo las diferentes estrategias, han pasado por la fase de transformación S^*, que reemplaza las funciones no estrictas por su versión paralela y aparecen como raíz en las *rhs*. Las condiciones de independencia entre sus argumentos que se conocen como ciertas o falsas aparecen como *true* o *false*. Como se ha asignado cero como valor del tiempo de evaluación de una condición de independencia que se conoce como *true* o *false*, en la tabla aparecen entradas con un tiempo cero de evaluación para estas condiciones.

En las tablas 2.1 y 2.2 se muestran las medidas resultado de la simulación de los programas de prueba bajo las diferentes estrategias. La primera columna contiene el nombre del programa de prueba. La segunda columna contiene el tiempo empleado en la evaluación secuencial del programa. Las columnas etiquetadas con las estrategias *up*, *mp* y *gp* contienen el tiempo empleado en la evaluación paralela de los programas en dos casos distintos: cuando se ha empleado informacion de independencia (columna etiquetada con *c.i.*) y cuando se ha obviado (columna etiquetada con *s.i.*). La columna etiquetada con *hp* contiene el tiempo empleado en la evaluación paralela de la versión del programa paralelizado a mano. Hay cuatro filas por cada programa, que contienen:

- el tiempo de evaluación,
- la ganancia de velocidad sobre la versión secuencial,
- el número de condiciones que se han probado y que han tenido éxito[35]

[33] Estos programas se pueden encontrar en el apéndice B.

[34] Esta información de independencia se ha obtenido mediante el análisis de independencia que se describe en el capítulo 3, bajo el dominio $\mathcal{D}_1^*$.

[35] El número de condiciones se entiende como el número de conjunciones de condiciones

y

- una medida del tiempo de evaluación de las condiciones[36].

De estas tablas podemos concluir lo siguiente:

- La estrategia *up* sin anotación de modos no consigue ganancia de velocidad sobre la versión secuencial. Además penaliza el tiempo de ejecución con comprobaciones de falsedad en las funciones lógicas[37].
- Añadir información de independencia proporciona una identificación óptima de paralelismo en la mayoría de los casos. Sólo cuando en el procedimiento de análisis para obtener información de independencia se infiere que un argumento puede ser de salida (i.e., *top*) y en la ejecución se utiliza como entrada, como en los casos *Aplanamiento de listas* y *Subárboles*, la información de independencia no provoca ganancia de velocidad. No obstante, las estrategias *mp* y *gp* lo consiguen a costa de la evaluación de las condiciones de independencia.
- Excepto en el programa *Árboles balanceados*, el número de condiciones probadas y con éxito es el mismo. Esto se debe a que, para representar los datos del problema se utilizan estructuras incompletas con variables lógicas sin instanciar, por lo que su comprobación de basicidad falla.
- La ganancia de velocidad conseguida en los problemas puramente paralelos es significativa ya que se usan funciones en un estilo de programación funcional de primer orden, lo que proporciona evidentes oportunidades de paralelismo. La ganancia de velocidad depende de la profundidad y anchura de los árboles datos puesto que se gestionan en paralelo sus subárboles.
- La versión de los programas paralelizados a mano es mejor que la paralelización automática cuando la información de independencia no es suficientemente precisa. Aunque la detección de paralelismo sea igual en la mayoría de los casos, el número de condiciones evaluadas es menor que en las versiones automáticas. Es decir, el procedimiento de

simples.

[36]La unidad de tiempo que en este caso se utiliza no es la misma que la utilizada para las llamadas a función.

[37]Una mejora obvia e inmediata del procedimiento de compilación es sustituir las funciones lógicas paralelas con condiciones de falsedad por las funciones secuenciales correspondientes.

Programa	Versión Sec.	*up*		*mp*		*gp*		*hp*
		s.i.	c.i.	s.i.	c.i.	s.i.	c.i.	
Números de Fibonacci (13^{o})	1881	1881	37	37	37	37	37	37
		1.0	50.8	50.8	50.8	50.8	50.8	50.8
		0/0	0/0	376/376	0/0	376/376	0/0	0/0
		0	0	752	0	376	0	0
Aplanamiento de listas (prof.: 6)	1019	1019	1019	154	154	154	154	154
		1.0	1.0	6.6	6.6	6.6	6.6	6.6
		0/0	0/0	190/190	190/190	190/190	190/190	0/0
		0	0	3206	3206	1990	1990	0
Árboles balanceados (prof.: 8)	514	514	48	48	48	48	48	48
		1.0	10.7	10.7	10.7	10.7	10.7	10.7
		0/255	247/255	247/255	247/255	247/255	247/255	247/255
		0	474	6682	474	3578	474	474
Sub-árboles (prof.: 5)	125	125	125	11	11	11	11	11
		1.0	1.0	11.4	11.4	11.4	11.4	11.4
		0/62	0/62	62/62	62/62	62/62	62/62	62/62
		0	0	6308	6308	6308	6308	0

Tabla 2.1: Medidas resultado de la simulación de los programas de prueba. I parte.

Programa	Versión Sec.	*up*		*mp*		*gp*		*hp*
		s.i.	c.i.	s.i.	c.i.	s.i.	c.i.	
Torres de Hanoi (8 discos)	1274	1274	17	17	17	17	17	17
		1.0	74.9	74.9	74.9	74.9	74.9	74.9
		0/255	255/255	509/509	255/255	509/509	255/255	255/255
		0	0	1147	0	763	0	0
Derivación simbólica (prof.: 4)	191	191	9	9	9	9	9	9
		1.0	21.2	21.2	21.2	21.2	21.2	21.2
		0/0	0/0	79/79	0/0	79/79	0/0	0/0
		0	0	1218	0	577	0	0
Producto vector-matriz (orden: 10)	321	321	31	31	31	31	31	31
		1.0	10.3	10.3	10.3	10.3	10.3	10.3
		0/0	0/0	110/110	0/0	110/110	0/0	0/0
		0	0	5820	0	620	0	0
Circuito combina-cional (3 niveles)	7	7	4	4	4	4	4	4
		1.0	1.7	1.7	1.7	1.7	1.7	1.7
		0/0	0/0	2/2	0/0	2/2	0/0	0/0
		0	0	6	0	2	0	0

Tabla 2.2: Medidas resultado de la simulación de los programas de prueba. II parte.

análisis de independencia no es lo suficientemente preciso para derivar esta información en tiempo de compilación. Sin embargo, el esquema de identificación de paralelismo puede detectar en tiempo de ejecución estas oportunidades de paralelismo que no se han podido detectar en tiempo de compilación.

- El comportamiento de las estrategias *mp* y *gp* es el mismo en la mayoría de los casos, conduciendo a detectar el mayor grado de paralelismo. Sólo cuando la información de independencia no conduce a la eliminación de las condiciones de las *EGEs*, la estrategia *gp* se comporta mejor que la estrategia *mp* si consideramos el tiempo de evaluación de las condiciones, como ocurre en *Aplanamiento de listas.*

- La estrategia *up* con información de independencia conduce a la misma ganancia de velocidad que las estrategias *mp* y *gp* en la mayoría de los casos probados. Si comparamos esta estrategia cuando se dispone de la información de independencia frente a las otras cuando no se dispone de la información de independencia, observamos que con *up* se consigue identificar el mismo grado de paralelismo con un costo menor en tiempo de compilación, lo que sugiere que podría ser una buena alternativa a la identificación de paralelismo. Sin embargo, se perderían las oportunidades de paralelismo que las estrategias *gp* y *mp* pueden detectar en tiempo de ejecución.

Sumario

En este capítulo se ha desarrollado un procedimiento para la extracción de paralelismo de programas secuenciales Babel y se ha medido su rendimiento con un simulador. De las medidas obtenidas se ha concluido que, con una buena información de independencia en tiempo de compilación, las estrategias consiguen una buena identificación de paralelismo a bajo costo. Esta identificación es similar en la mayoría de los casos a la versión paralela manual. Sin embargo, con poca información de independencia, la estrategia *gp* puede identificar el mismo paralelismo que la estrategia *mp* a menor costo en determinados programas. Se concluye, pues, que el rendimiento de las estrategias depende en gran medida del programa y de la información de independencia disponible.

Como puntos de trabajo futuro se pueden considerar los siguientes. En primer lugar, se pueden plantear otras estrategias de incorporación de la

información de granularidad en base a los resultados de la simulación. Por ejemplo, rompiendo la ordenación secuencial de las tareas secuencializadas de modo que se detecten posibles fallos en el camino de cómputo con anticipación. También se puede considerar la propuesta de otras estrategias de identificación de paralelismo como, por ejemplo, forzando la agrupación secuencial de tareas[38] para compararla con las desarrolladas aquí. Por último, sería de interés considerar un lenguaje lógico-funcional con funciones no estrictas definidas por el usuario, i.e., con mecanismo de evaluación perezoso.

[38] Como se propone en el algoritmo MEL de programación lógica [122].

Capítulo 3

Análisis de independencia

Las condiciones de independencia son costosas de evaluar en tiempo de ejecución, por lo que es interesante poder resolver tantas como sea posible en tiempo de compilación. En este capítulo se desarrolla un análisis de independencia para obtener la información que permita simplificarlas.

En primer lugar se introduce la técnica de análisis de programas basada en interpretación abstracta. A continuación se extiende la interpretación abstracta de programas lógicos a lógico-funcionales. Se desarrollan tres niveles de análisis con los que se obtienen distintos grados de información de independencia. Por último, se comparan estos niveles de análisis en términos de la información útil inferida y el tiempo necesario para obtenerla.

3.1 Interpretación abstracta

La interpretación abstracta es una técnica de análisis que permite inferir las propiedades que se verifican en un determinado punto de un programa para cualquier estado de su ejecución. Con esta técnica, en lugar de realizar una ejecución en el dominio real, se efectúa una ejecución (o interpretación del programa) en un dominio abstracto cuyos elementos capturan las propiedades que comparten un conjunto de datos del dominio real. Estas propiedades se utilizan en la compilación de los programas para su mejora de rendimiento.

La interpretación abstracta como técnica de análisis de flujo de datos nace en el marco de la programación imperativa de manos de P. y R. Cousot [41][1]. Bajo su propuesta, los programas se representan como grafos dirigidos

[1] En [90] se presentan también los fundamentos teóricos de esta técnica como análisis

cuyos nodos corresponden a operaciones y los arcos a flujo de control. El punto de programa es el contador de programa, que se representa por un arco. El estado está representado por el contador de programa y la memoria (llamada entorno de programa), que almacena las vinculaciones de las variables. La semántica del programa se define por una función de transición de estados que identifica cada una de las operaciones posibles y cómo afectan al estado del programa. La interpretación abstracta de un programa se calcula como la recolección de todos los entornos que se pueden asociar a un punto de programa durante su ejecución.

Un ejemplo de interpretación abstracta es la regla de los signos [61]. Consideremos la sintaxis abstracta de un lenguaje de expresiones aritméticas definidas inductivamente como sigue:

$$\begin{aligned} Exp :: &= c_n \\ &\mid Exp + Exp \\ &\mid Exp \times Exp \end{aligned}$$

donde $\mathcal{Z} = \{c_n : n \text{ número entero}\}$.

La interpretación estándar ($[[]]$) que asignamos a este lenguaje es:

$$\begin{aligned} [[c_n]] &= \text{entero } n \\ [[+]] &= \text{función suma de enteros} \\ [[\times]] &= \text{función producto de enteros} \end{aligned}$$

Con este lenguaje se pueden describir e inferir los signos derivados de una operación aritmética que incluya sumas y productos sobre números enteros. Sin embargo, si tan sólo estamos interesados en la propiedad que distingue los números positivos de los negativos, conviene definir una interpretación del lenguaje que maneje menos información. Denotamos esta interpretación abstracta con $[[]]^*$ y se define como sigue.

$$[[c_n]]^* = signo([[c_n]])$$

$$\text{donde } signo(n) = \begin{cases} \oplus \text{ si } n > 0 \\ 0 \text{ si } n = 0 \\ \ominus \text{ si } n < 0 \end{cases}$$

En este marco se puede definir la regla de los signos apoyándonos en la función *abstracta* producto ($\times^*$) que se aplica sobre elementos de la interpretación abstracta ($\{\oplus, 0, \ominus\}$), y que se define a continuación.

---

global de programas.

$$\begin{array}{ll} \ominus \times^* \ominus = \oplus & \oplus \times^* 0 = 0 \\ \oplus \times^* \ominus = \ominus & 0 \times^* \ominus = 0 \\ \ominus \times^* \oplus = \ominus & \ominus \times^* 0 = 0 \\ \ominus \times^* \ominus = \oplus & 0 \times^* 0 = 0 \\ 0 \times^* \oplus = 0 & \end{array}$$

Puesto que no se puede inferir en este marco de interpretación abstracta el resultado de la suma abstracta de $\ominus$ y $\oplus$, se introduce el valor $\top$ (*top*) para representar la ausencia de información acerca del resultado de la función abstracta. Las reglas que definen la función abstracta suma ($+^*$) son las siguientes:

$$\begin{array}{lll} \oplus +^* \oplus = \ominus & \oplus +^* 0 = \oplus & 0 +^* \top = \top \\ \oplus +^* \ominus = \top & 0 +^* \ominus = \ominus & \top +^* \oplus = \top \\ \ominus +^* \oplus = \top & \ominus +^* 0 = \ominus & \oplus +^* \top = \top \\ \ominus +^* \ominus = \ominus & 0 +^* 0 = 0 & \ominus +^* \top = \top \\ 0 +^* \oplus = \oplus & \top +^* 0 = \top & \top +^* \ominus = \top \\ & \top +^* \top = \top & \end{array}$$

Al añadir $\top$ al conjunto de elementos del dominio abstracto, añadimos también las siguientes reglas, correspondientes a la función abstracta producto.

$$\begin{array}{ll} \top \times^* 0 = 0 & \oplus \times^* \top = \top \\ 0 \times^* \top = 0 & \ominus \times^* \top = \top \\ \top \times^* \oplus = \top & \top \times^* \ominus = \top \\ \top \times^* \top = \top & \end{array}$$

Por razones de completud, se introduce el valor $\bot$ que representa un entero indefinido (aunque en esta semántica no es estrictamente necesario). El dominio $\mathcal{Z}$ se extiende a $\mathcal{Z}_\bot$, donde:

$$\mathcal{Z}_\bot = \mathcal{Z} \cup \{\bot\}$$

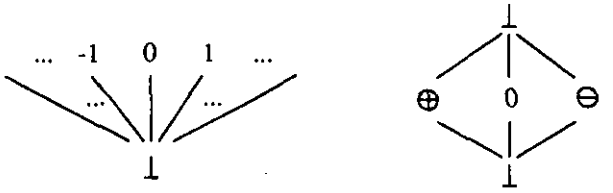
Sobre este dominio se define un orden $\sqsubseteq$ entre sus elementos tal que $z_1 \sqsubseteq z_2$ si y sólo si $z_1 = \bot$ o $z_1 = z_2$.

El dominio *abstracto* $\mathcal{Z}^*$ queda como sigue:

$$\mathcal{Z}^* = \{\top, \oplus, 0, \ominus, \bot\}$$

En la figura 3.1 se muestran el dominio estándar $\mathcal{Z}_\bot$ y el dominio abstracto $\mathcal{Z}^*$, donde éste se ha ordenado para que forme un retículo completo.

En la figura 3.2 mostramos las interpretaciones estándar ($[\![]\!]$) y abstracta ($[\![]\!]^*$) correspondientes a nuestro lenguaje. En dicha figura observamos que lo único que las diferencia es la interpretación de las constantes y de las funciones aritméticas. En la parte derecha de las reglas, denotamos con los símbolos $+$ y $\times$ a las funciones que realizan la adición y la multiplicación de enteros, respectivamente.



Dominio estándar *Dominio abstracto*

Figura 3.1: Dominios estándar y abstracto de los enteros.

$$
\begin{aligned}
&[[]] : Exp \to \mathcal{Z} \\
&[[c_n]] = n \\
&[[Exp_1 + Exp_2]] = [[Exp_1]] + [[Exp_2]] \\
&[[Exp_1 \times Exp_2]] = [[Exp_1]] \times [[Exp_2]]
\end{aligned}
$$

$$
\begin{aligned}
&[[]]^* : Exp \to \mathcal{Z}^* \\
&[[c_n]]^* = signo([[c_n]]) \\
&[[Exp_1 + Exp_2]]^* = [[Exp_1]]^* +^* [[Exp_2]]^* \\
&[[Exp_1 \times Exp_2]]^* = [[Exp_1]]^* \times^* [[Exp_2]]^*
\end{aligned}
$$

Figura 3.2: Interpretaciones estándar y abstracta.

Las funciones de *concretización* (γ) y *abstracción* (α) relacionan los elementos de ambos dominios[2]:

$\gamma : \mathcal{Z}^* \to \mathcal{P}(\mathcal{Z}_\perp)$

$\gamma(z^*) = Z : Z$ es el mayor elemento bajo $\sqsubseteq$ que describe a z^*

$\alpha : \mathcal{P}(\mathcal{Z}_\perp) \to \mathcal{Z}^*$

$\alpha(Z) = z^* : z^*$ es el menor elemento bajo $\sqsubseteq$ que describe a Z

La función de concretización captura los elementos que se abstraen en el elemento z^*, mientras que la función de abstracción devuelve el elemento del dominio abstracto que representa a un conjunto de elementos del dominio real. Por ejemplo:

$$\gamma(\oplus) = \{n : n > 0\} \cup \{\perp\}$$
$$\gamma(\top) = \mathcal{Z}_\perp$$
$$\alpha(\{-1\}) = \ominus$$
$$\alpha(\{0, 1\}) = \top$$

En la figura 3.3 se muestra la relación entre los conjuntos asociados a los dominios estándar y abstracto mediante las funciones α y γ.

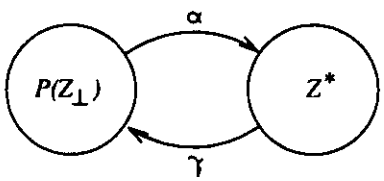


Figura 3.3: Relación entre los conjuntos $\mathcal{Z}_\perp$ y $\mathcal{Z}^*$.

3.1.1 Interpretación abstracta de programas funcionales

La interpretación abstracta fue extendida al marco funcional por Mycroft *et al.* [126]. El nuevo marco incorpora los elementos matemáticos necesarios para manejar análisis más interesantes para la programación funcional, derivados fundamentalmente de la presencia de las funciones no estrictas. Por ejemplo, algunas propiedades de interés para un esquema de evaluación perezoso son las de *estricticidad* y *necesidad* de argumentos [124, 66, 75], que se refieren al requerimiento de la evaluación de argumentos hasta un patrón determinado (i.e., forma normal, primera constructora de cabeza, ...)[3].

[2]En lo que sigue, $\mathcal{P}$ representa el conjunto potencia.

[3]En este marco cada fase previa y posterior a la evaluación de una función es un punto de programa, y el estado está determinado por las asignaciones de las variables.

3.1.2 Interpretación abstracta de programas lógicos

El análisis clásico de flujo de datos aplicado al análisis de programas lógicos [44, 111, 132, 112, 28, 156, 105] se puede incluir como una instancia del marco teórico de interpretación abstracta. La extensión de esta técnica a la programación lógica es más compleja ya que el flujo de control es más complicado debido al *backtracking*. Bruynooghe [21], Jones y Søndergaard [88], y Mellish [110] extendieron formalmente esta técnica al ámbito de la programación lógica. A partir de entonces se han realizado múltiples trabajos acerca de la interpretación abstracta de programas lógicos [46, 129, 152, 137, 127, 163, 24, 22, 121, 123, 62, 160, 23, 148, 36, 60, 40, 39][4]. En el marco lógico, cada fase de la ejecución previa y posterior a la resolución de un objetivo es un punto de programa. El estado está determinado por la sustitución de las variables lógicas. Algunas propiedades que se infieren generalmente son los tipos, los modos, la información de independencia y el determinismo [112].

3.2 Interpretación abstracta de lenguajes lógico-funcionales

En esta sección extendemos la interpretación abstracta de programas lógicos desarrollada por Bruynooghe [22] al análisis de programas lógico-funcionales. Aplicamos esta técnica de análisis a la abstracción de la función semántica de un programa lógico-funcional Babel con funciones primitivas no estrictas.

Comenzaremos con la presentación de la notación necesaria y las ideas básicas.

Sea define E como el conjunto potencia de un conjunto dado de elementos del dominio de cómputo (en adelante, elementos reales en el dominio real) y E_P como el operador de transición de estados de un programa P. La ejecución abstracta de P se guía por el operador abstracto de transición de estados D_P, que se aplica a D (dominio abstracto), un conjunto de propiedades o descripciones de los elementos reales dotado de orden parcial. D_P se debe elegir de manera que, cuando se aplique a D, reproduzca el comportamiento de E_P sobre E. Se define una función de abstracción $\alpha : E \rightarrow D$ que relaciona los elementos de los dominios real y abstracto. El *significado* de un programa P se define como el *menor punto fijo* (*least fixpoint - lfp*)

[4] En [107] se ofrece una buena presentación de los conceptos fundamentales y en [1] se recopilan trabajos representativos acerca de la interpretación abstracta de lenguajes declarativos.

del operador E_P ($lfp(E_P)$). Se obtiene una aproximación conservadora del significado de un programa P calculando la concretización del menor punto fijo de D_P (c.f. [41]), esto es, $\gamma(lfp(D_P))$. El significado de un programa P es un subconjunto del conjunto $\gamma(lfp(D_P))$ que se infiere en el análisis.

Consideremos por ejemplo la función semántica F_P de un programa P, tal que $F_P : E \to E$. Se dice que la función $G_P : \mathcal{P}(D) \to \mathcal{P}(D)$ es una aproximación segura de F_P si $\forall X, X \in \mathcal{P}(D)$ y $G_P(X) \supseteq F_P(X)$. Esto asegura que todas las propiedades ciertas en G lo son también en F_P. Es decir, si para la instancia G de G_P ($G : D \to D$), se cumple $\gamma(G(x)) \supseteq F_P(\gamma(x))$, de lo que se da cuenta en la representación gráfica de la figura 3.4. Al igual que consideramos la función semántica de un programa, podemos considerar cualquier otra operación en este desarrollo, como la unificación.

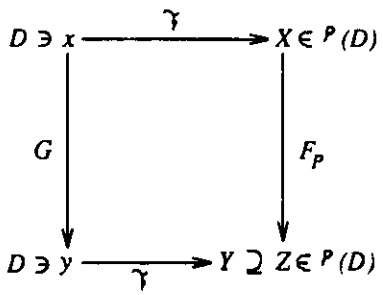


Figura 3.4: Corrección de una interpretación abstracta.

En general, y para asegurar cálculos finitos, se exige que las secuencias de Kleene para las funciones sean finitas, para lo cual se impone que los dominios sean retículos completos o cadenas ascendentes finitas [108].

En un lenguaje lógico-funcional como Babel podemos elegir de manera natural el operador E_P como el operador paso de estrechamiento, de manera análoga a la elección del operador paso de resolución en Prolog. Retrasaremos la presentación del dominio de datos hasta el momento en el que propongamos los dominios abstractos, ya que veremos cómo la elección de un dominio de datos adecuado favorece la implementación de una implementación eficiente.

Comenzaremos considerando Babel sólo con funciones estrictas y más adelante incluiremos las funciones primitivas no estrictas. Bajo esta premisa, la construcción del árbol de evaluación es parecida a la de Prolog.

Los elementos que manejaremos en este marco son sustituciones abstractas, como en el marco de la programación lógica [22], y además una

representación abstracta de los resultados que devuelven las funciones, esto es, resultados abstractos.

Construcción de árboles de evaluación abstracta

Utilizaremos un árbol de evaluación abstracta similar al árbol *And-Or* de programación lógica, en el que se alternan nodos *And* y *Or*, para representar los conjuntos de árboles de evaluación. Un nodo *And* representa un literal del cuerpo de una cláusula (conjunción de objetivos), mientras que un nodo *Or* representa la cabeza de la regla que se puede unificar con éxito con un nodo *And* dado.

La parte izquierda de una regla Babel se compone de funciones y términos de datos y el resultado de su evaluación es un término de datos o una aplicación parcial que se evalúa a sí misma. En el árbol de evaluación abstracta usaremos un nuevo tipo de nodo para representar la parte derecha de una regla, además de los nodos para representar las partes izquierdas de las reglas y los nodos para representar las aplicaciones de función.

Dada la regla $h := e[e_1, \ldots, e_n].$, su parte derecha e (función de las aplicaciones $e_1, \ldots, e_n$) es un *nodo de expresión* E cuyos hijos son los nodos *Or* f_i. Cada nodo f_i representa una aplicación en la parte derecha de la regla, donde $f_i = e_i[f_j/\rho_j], 1 \leq i \leq n, j \in \{1, \ldots, i-1 : f_j \in e_i\}$. Esto significa que cada aparición f_j en e_i se ha reemplazado por el correspondiente resultado abstracto de la aplicación f_j. Cada parte izquierda de una regla que es unificable con un nodo *Or* es un nodo de función F, hijo del nodo *Or*.

La construcción del árbol de evaluación abstracta sigue el esquema de evaluación impaciente, comenzando por la expresión a evaluar. En la figura 3.5 se ilustran las tres clases de nodos junto a las sustituciones abstractas y los resultados abstractos. λ_{call} es la sustitución abstracta de llamada a una función, esto es, la sustitución abstracta previa a la evaluación de la función. λ_{succ} es la sustitución abstracta de éxito, esto es, la sustitución abstracta calculada durante la evaluación abstracta de la función. β_{entry} es la sustitución abstracta de entrada de una regla, esto es, el resultado de la unificación de la parte izquierda de la regla y la llamada a la función (supuesto que los argumentos de la función ya han sido evaluados). β_{exit} es la sustitución abstracta de salida, esto es, la sustitución abstracta calculada después de la evaluación de la parte derecha de la regla. Finalmente, ρ es el resultado abstracto que representa la expresión evaluada[5].

[5] A partir de aquí, escribiremos sustituciones en lugar de sustituciones abstractas, y resultados en lugar de resultados abstractos cuando no haya lugar a ambigüedad.

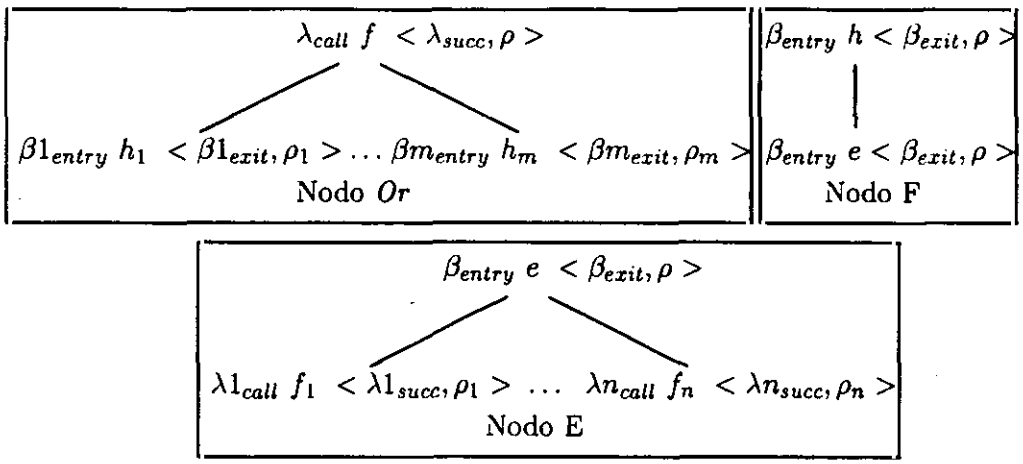


Figura 3.5: Nodos del árbol abstracto de evaluación.

El par de éxito abstracto $<\lambda_{succ},\rho>$ de un nodo *Or* se calcula como el resultado conservador de sus m nodos F $<\beta i_{exit},\rho_i>$ como sigue. La sustitución de éxito se calcula como $\lambda_{succ} = lub(\{\pi(\beta i_{exit},p) : 1 \leq i \leq m\})$, donde la operación *lub* (*least upper bound*) calcula la menor cota superior bajo un dominio dado, y π es la proyección de una sustitución sobre la tupla de argumentos p del nodo F. La sustitución de entrada de un nodo F h se calcula como la unificación abstracta de la parte izquierda h de una regla y la aplicación de función representada en su nodo *Or* padre. Para los nodos *Or* $f_1,\ldots,f_n$ hijos de un nodo E e, la sustitución de llamada de f_1 es la sustitución de entrada de e, y para el resto de nodos se cumple $\lambda i_{call} = \lambda i - 1_{succ}$. El resultado abstracto de un nodo *Or* se calcula como $lub(\{\rho_i : 1 \leq i \leq n\})$, donde ρ_i son los resultados de sus nodos hijos F. La abstracción de una expresión e bajo una sustitución β_{exit} se calcula como $\rho = \alpha(e[e_i/\rho_i])$, esto es, el elemento en el dominio abstracto que representa la expresión e, donde las subexpresiones e_i se han reemplazado por sus resultados ρ_i y las vinculaciones de las variables se proporcionan en β_{exit}. Para mantener la corrección es necesario calcular las sustituciones de los nodos *Or* en el orden de evaluación en el dominio real. Sin embargo, y debido a la conmutatividad de la operación *lub*, las sustituciones de los

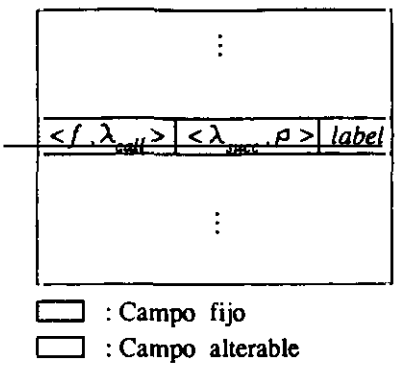


Figura 3.6: Tabla de memoria.

El algoritmo de cálculo de punto fijo es como sigue. El procedimiento de interpretación abstracta comienza con una llamada a función f con una sustitución de llamada λ_{call}. A continuación se busca la correspondiente entrada en la tabla de memoria.

- Si existe la entrada, entonces:
  - Si *label* contiene el valor *complete*, el par de éxito almacenado en la tabla de memoria se usa como par de éxito para $< f, \lambda_{call} >$ puesto que dicho par se ha calculado como el definitivo para $< f, \lambda_{call} >$.
  - Si *label* contiene el valor *fixpoint*, la información almacenada en la tabla de memoria se utiliza como en el punto anterior pero anotando el cómputo actual como aproximado, puesto que dicha información no es definitiva.
  - Si *label* contiene el valor *approximate*, el campo *label* se modifica para que contenga el valor *fixpoint* para prevenir cómputos infinitos (c.f. [121]). Se inicia de nuevo la búsqueda de punto fijo de $< f, \lambda_{call} >$. Cuando se termina el cómputo para $< f, \lambda_{call} >$, el cómputo actual (que en general corresponderá a otra función diferente g) se anota como aproximado, de manera que, tras el cómputo de g, su entrada en la tabla de memoria se anota con el valor *approximate*.
- Si no existe tal entrada se almacena en la tabla de memoria la tupla $<< f, \lambda_{call} >, < \lambda_{seed}, \rho_{seed} >, fixpoint >$, donde el par $< \lambda_{seed}, \rho_{seed} >$ representa el valor inicial sobre el que se aplica el operador de punto

fijo. Este valor inicial se calcula como el resultado más general bajo el dominio dado para el par de éxito de $< f, \lambda_{call} >$. A continuación se realiza el siguiente cálculo de punto fijo.
Sea n el número de reglas unificables con f bajo λ_{call}. Se calcula el par $< \beta i_{exit}, \rho_i >$ del i-ésimo nodo F, y la sustitución βi_{exit} se proyecta sobre el nodo f para obtener $< \lambda i_{succ}, \rho_i >$. El par de éxito $< \lambda lub_{succ}, \rho_{lub} >$ se calcula como $< lub(\{\lambda i_{succ}, \lambda mt_{succ}\}), lub(\{\rho_i, \rho_{mt}\}) >$, donde $< \lambda mt_{succ}, \rho_{mt} >$ es el par de éxito almacenado en la tabla de memoria para $< f, \lambda_{call} >$. Si el par de éxito no cambia durante el cómputo de las n reglas, el par no se puede refinar más y se considera como cálculo definitivo, finalizando el proceso iterativo. Después, y si el cómputo actual se había anotado como aproximado (debido al cálculo de un punto fijo de otra función en el subárbol de evaluación), entonces la entrada para $< f, \lambda_{call} >$ se anota como aproximada (campo *label* con el valor *approximate*). Si el par de éxito cambió debido a alguna de las n reglas, el cómputo de punto fijo se reinicia.

El algoritmo propuesto realiza un reconocimiento general de los cómputos aproximados (mediante la anotación del cómputo actual como aproximado) relativos a un determinado subárbol, en lugar de centrarse únicamente en los nodos que son responsables del cómputo aproximado. Por una parte, nuestro algoritmo no selecciona para las aplicaciones sucesivas (recómputo) de D_P los nodos *Or* responsables del cómputo aproximado como hace el algoritmo propuesto en [121]. En su lugar calculamos el subárbol completo del nodo *Or*, siendo 'revisitados' todos sus hijos y realizando más trabajo. Por otra parte, nótese que este 'recómputo' evita mucho trabajo previamente realizado gracias a la información almacenada en la tabla de memoria. La aproximación propuesta en [121] elimina la 'revisita', pero pagando el costo extra de la gestión de las necesarias estructuras de datos. Nuestra propuesta nos permite además plantear una implementación eficiente de un intérprete abstracto basado en ejecución simbólica, como propusimos en el marco de la programación lógica [139].[7]

[7] Más tarde, se presentó un trabajo similar en [151]. La desventaja principal de dicho trabajo es que no se tienen en cuenta los cómputos aproximados para evitar el recómputo. Su propuesta está basada en el cómputo iterativo del árbol abstracto hasta que se se alcanza el punto fijo. Nuestra propuesta se haya, pues, entre las dos comentadas, puesto que evitamos el recómputo de muchos casos que [151] no detecta, pero no todos los que distingue [121]. La ventaja de nuestra propuesta es que aún podemos plantear sencillamente el esquema de compilación basado en ejecución simbólica manteniendo un algoritmo razonable de búsqueda de punto fijo (véase [139]).

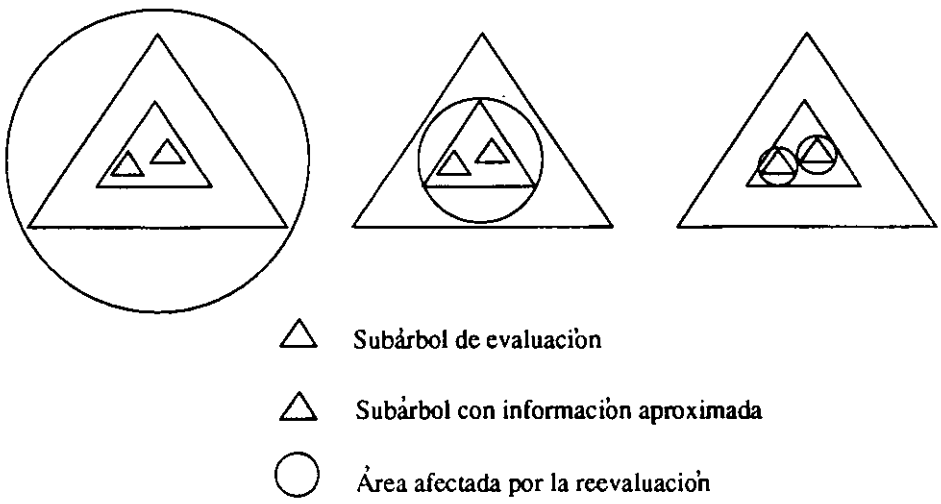


Figura 3.7: Comparación entre los diferentes algoritmos.

Para estudiar el algoritmo de búsqueda de punto fijo proponemos el siguiente ejemplo que incorpora una recursión mutua complicada.
$f := g, h.$
$g := f, h.$
$h := f, g.$

El árbol de evaluación abstracta correspondiente al análisis de la expresión f se muestra en la figura 3.8. Cada nodo del árbol puede estar etiquetado de dos formas:

- ${}^{L_1}_{L_2}e_{L_3}$, siendo L_1 la etiqueta de la entrada en la tabla de memoria para la función e (si no existe tal entrada, L_1 es ?); L_2 es el valor al que se cambia la etiqueta *antes* de la evaluación abstracta de e; y L_3 es el valor de la etiqueta *después* de la evaluación de e. En la figura se ha utilizado `app` y `fp` para denotar respectivamente los valores *aproximate* y *fixpoint*.

- ${}^{e}_{L}$, siendo L la etiqueta de la entrada en la tabla de memoria para la

En la figura 3.7 se muestra un ejemplo que compara el efecto sobre el recómputo de un árbol, que depende de dos subárboles, que tienen los diferentes algoritmos comentados. En el caso de la izquierda [151] se muestra que la reevaluación afecta a todo el árbol. En el caso central, que corresponde a nuestra propuesta, se muestra que la reevaluación sólo afecta al subárbol que contiene a los subárboles responsables de la reevaluación. Finalmente, en el caso de la derecha se muestra que se reevalúa lo estrictamente necesario.

función e. El valor del par de éxito de esta entrada es el que se utiliza como par de éxito aproximado para e en lugar de evaluar de nuevo la función.

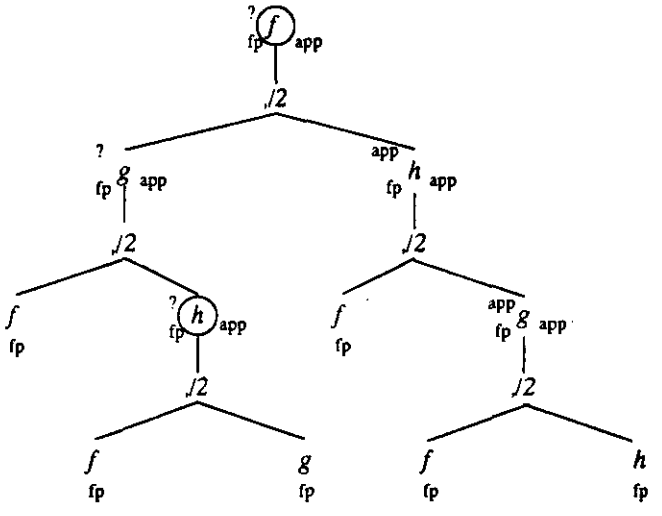


Figura 3.8: Árbol de evaluación abstracta de f.

En base a este ejemplo destacamos algunos comentarios relativos al algoritmo:

- Situaciones en que se modifica el campo *label* para que contenga el valor *approximate*:

  1. Cuando se utiliza una entrada *fixpoint* del par de llamada $< e_1, \lambda_1 >$ para la evaluación del subárbol correspondiente a $< e, \lambda >$, siendo $e \neq e_1$ o $\lambda \neq \lambda_1$. Esto es necesario porque, al usar información aproximada en el cómputo del punto fijo actual procedente del cómputo de otro punto fijo, se debe anotar como aproximado el cómputo del punto fijo actual. La anotación del campo *label* de esta entrada con este valor fuerza a que en el próximo ciclo de cómputo de punto fijo se reevalúe la función, en lugar de tomar el par de éxito de la entrada correspondiente en la tabla de memoria.

     El nodo h, rodeado por un círculo en la figura 3.8, es un ejemplo de este caso, en el que se ha utilizado información aproximada de

los nodos f y g, que son funciones en proceso de cálculo de punto fijo, cuyos campos *label* contenían el valor *fixpoint*.

2. Cuando durante la evaluación de una función g del subárbol de evaluación abstracta correspondiente a f, la entrada en la tabla de memoria para g tiene el campo *label* con el valor *approximate*. Esto es necesario porque también se utiliza información aproximada en la evaluación de la función considerada.

   El nodo f, rodeado con un círculo en la figura 3.8, es un ejemplo de este caso, su entrada en la tabla de memoria se marca como *approximate* a causa de que las entradas en la tabla de memoria de g y h contienen el valor *approximate*.

En ambos casos, el campo *label* se modifica *después* de la evaluación del subárbol correspondiente, i.e., el de los nodos h y f marcados para los casos 1 y 2, respectivamente.

- Cambio del contenido del campo *label* de *approximate* a *fixpoint*.

  Este cambio se produce cuando se evalúa de nuevo la función g en el cómputo de punto fijo.

  El valor *approximate* significa que la entrada etiquetada con él contiene información aproximada que se debe reevaluar en el próximo ciclo del cómputo de punto fijo para obtener, en general, una información más precisa. Sin embargo, ésto sólo debe hacerse una vez en un subárbol determinado puesto que en otro caso el subárbol se repetiría indefinidamente al evaluar una y otra vez el mismo par de llamada.

  Esto se ve claramente en el ejemplo anterior: el subárbol más a la izquierda de f contiene un nodo g que a su vez contiene un nodo g idéntico en el nivel más bajo del subárbol. La entrada para g en la tabla de memoria queda etiquetada como *approximate*. El subárbol de f más a la derecha contiene otro nodo para g idéntico al del subárbol más a la izquierda. Si se reevalúa sin cambiar el campo *label* de g a *fixpoint* previamente a la evaluación de g, entonces aparecería otro nodo g en el subárbol de g cuya entrada en la tabla de memoria estaría etiquetada como *approximate*, el cual se volvería a evaluar, siguiendo el proceso indefinidamente.

- Situaciones en que se realizan evaluaciones innecesarias:

1. Cuando en un ciclo de cómputo de punto fijo para una función f aparecen dos o más nodos para g con los mismos pares de llamada en distintos subárboles.

   En el ejemplo anterior, g y h se reevalúan sin necesidad en un mismo ciclo. La primera vez que se resuelven se deposita en la tabla de memoria el resultado del par de éxito, resultado que, aunque aproximado de forma global, es completo para el ciclo actual de cómputo de punto fijo ya que el par de éxito que se obtenga es el mismo que el almacenado en la tabla de memoria, puesto que la información del par de éxito de f no se actualiza de nuevo hasta completar un ciclo de cómputo de punto fijo.

2. Cuando al evaluar por completo una función f mutuamente recursiva, aparece otra llamada posterior con el mismo par de llamada. En efecto, la tabla de memoria contiene la entrada para f etiquetada como *approximate*. Con el algoritmo utilizado es imposible detectar cuándo el cálculo es *complete*. No obstante, el algoritmo termina en tiempo finito. Para la detección de esta situación sería necesario utilizar listas de nodos como en [121]. Nótese que una reevaluación da lugar a tan sólo un ciclo de cómputo de punto fijo, ya que el par de éxito de tal función que se vuelve a calcular sería el mismo que el almacenado en la tabla de memoria.

3. En el último ciclo de cómputo de punto fijo en funciones totales aplicadas a uno o más argumentos.

   La condición de parada del algoritmo de cómputo de punto fijo de una determinada aplicación de función f es que el par de éxito almacenado en la tabla de memoria asociado a f no haya cambiado. Para detectar este cambio es necesario haber calculado dos veces el mismo par de éxito. Por lo tanto, el último ciclo del cómputo de punto fijo corresponde a un cálculo redundante.

3.2.1 Validación de la extensión

En esta sección se proporciona la guía para la validación de la extensión de la interpretación abstracta en el marco lógico al marco lógico-funcional presentado.

En primer lugar, proponemos un esquema de traducción de programas Babel a programas Prolog, mostrando que el resultado que proporciona la interpretación abstracta en el marco lógico-funcional cubre el resultado pro-

porcionado por la interpretación abstracta en el marco lógico[8].

Consideremos el esquema de traducción $\mathcal{T}$ de programas Babel a programas Prolog. Cada regla $f(t_1,\ldots,t_n) := e.$ en la definición de la función f se traduce bajo $\mathcal{T}$ en una cláusula C que define el predicado $f/n+1$, cuyo $n+1$-ésimo argumento es una variable R que representa el resultado de la función. Los m primeros objetivos de C son las expresiones e'_i que representan las aplicaciones e_i de la expresión e, cuyas aplicaciones e_j se han reemplazado por las correspondientes variables nuevas R_j. Cada expresión e'_i tiene las variables nuevas R_i que representan el resultado de cada aplicación e_i. Las expresiones e'_i se ordenan en C siguiendo el orden de evaluación impaciente de izquierda a derecha. El último objetivo de C es la asignación de R a la expresión que se obtiene de reemplazar las aplicaciones e_i en la expresión e por las variables R_i. La transformación viene dada por el siguiente esquema: $f(t_1,\ldots,t_n) := e[e_1,\ldots,e_m]. \xrightarrow{\mathcal{T}} f(t_1,\ldots,t_n,R) : -e'_1[R_1],\ldots,e'_m[R_m], R = e[e_i/R_i]$.

En la figura 3.9 se muestran los subárboles correspondientes a los casos lógico y lógico-funcional para una llamada f junto a su sustitución de entrada β_{entry}. Suponiendo la misma sustitución de entrada para ambos casos[9] , λi^P_{succ} difiere de λi^B_{succ} sólo en las vinculaciones debidas a $R_1,\ldots,R_i$. Estas vinculaciones no se encuentran en λi^B_{succ} puesto que la información relativa a ellas se encuentra incorporada en el segundo argumento de los pares de éxito y salida ($< \lambda i_{succ}, \rho_i >$, y $< \beta_{exit}, \rho >$ respectivamente). Puesto que las operaciones abstractas sobre sustituciones abstractas se realizan de igual forma en los marcos de Babel y Prolog y, dado que las operaciones relativas a los resultados abstractos son las realizadas sobre los valores a los que están ligados las variables $R_1,\ldots,R_m$, podemos observar que la interpretación abstracta de Babel proporciona los mismos resultados que la interpretación abstracta del correspondiente programa Prolog traducido. Esto es cierto salvo para la información relativa a las vinculaciones $R_1,\ldots,R_m$, que no se incorpora en las sustituciones, pero que están presentes explícitamente en el árbol abstracto de evaluación. De hecho, los resultados proporcionados por la interpretación abstracta en el marco lógico-funcional son en general más

[8] La noción de *cobertura* se debe entender como la conservación de la información derivada en uno de los marcos y, posiblemente, la derivación de información más precisa. La información más precisa se entiende como menos conservadora, i.e., asociada a los elementos del dominio abstracto más cercanos al elemento $\perp$ en el retículo bajo la relación de orden dada. Es de sobra conocido que, para los propósitos del análisis, conforme descendemos en el retículo asociado al dominio abstracto obtenemos información más precisa.

[9] $\beta_{entry} = \beta^P_{entry} = \beta^B_{entry}$, donde los superíndices P y B representan en lo sucesivo a los casos Prolog y Babel, respectivamente.

precisos que los del marco lógico para el mismo dominio. Este hecho es obvio ya que, durante la traducción, la información acerca de la independencia de las nuevas variables no se mantiene.

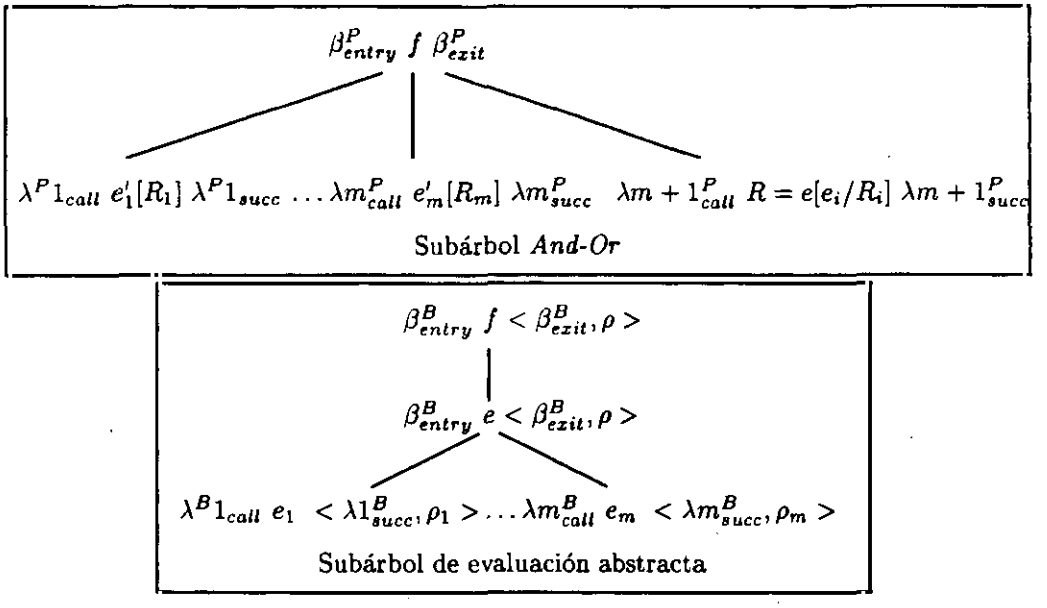


Figura 3.9: Subárboles para los casos lógico y lógico-funcional.

Podemos también considerar que la interpretación abstracta de un programa Babel se podría haber realizado a través de la interpretación abstracta del programa Prolog resultado de la traducción $\mathcal{T}$ del programa Babel. Sin embargo, para mantener los mismos resultados de nuestro marco, debemos incorporar explícitamente en $\mathcal{T}$ la información concerniente a la independencia de las variables nuevas. Pero de este modo estamos obligados a considerar un dominio abstracto con una representación explícita de dependencia. Con nuestra propuesta, el marco de interpretación abstracta no depende del dominio abstracto para incorporar la información de independencia de las aplicaciones en la parte derecha de una regla[10].

[10]Por otro lado, puesto que en los programas Babel se declara implícitamente la información de independencia de estas aplicaciones, comprobamos la mayor expresividad de Babel sobre Prolog.

En [62] se presenta la interpretación abstracta de programas lógico-funcionales introduciendo explícitamente las vinculaciones de las variables nuevas en las sustituciones abstractas. Por las razones expuestas anteriormente, nuestra solución es más general que la propuesta en [62].

3.2.2 Tratamiento de las funciones no estrictas

La interpretación abstracta desarrollada es correcta para las funciones estrictas en todos sus argumentos, pero si consideramos las funciones primitivas no estrictas guarda, condicional, conjunción y disyunción, se debe modificar para considerar los argumentos de las funciones no estrictas que no se evalúan. A través del siguiente ejemplo, podemos ver el tipo de problemas que aparecen en la interpretación abstracta de las funciones no estrictas.

Consideremos un dominio abstracto con información de basicidad, y supongamos que la evaluación de una expresión que contiene la función binaria conjunción cierra su segundo argumento. En la evaluación real se puede dar el caso de que el segundo argumento nunca se evalúe porque se haya calculado un resultado definitorio para el primero. El esquema presentado asume funciones estrictas, por lo que se calcula el resultado de la evaluación del segundo argumento, es decir, se obtiene que se cierra (se hace básico).

Para gestionar las funciones no estrictas de manera correcta seguimos la solución propuesta en [62], que propone una transformación de las reglas del programa en reglas equivalentes con respecto al propósito del análisis, de modo que el resultado de la interpretación abstracta ofrece el resultado correcto esperado. La idea es transformar cada regla que contenga una función primitiva no estricta en un conjunto de reglas que contengan funciones primitivas estrictas. Cada regla de este conjunto contempla cada uno de los casos de la ejecución real de la regla con funciones no estrictas. Todas estas reglas pertenecen a la definición de la función considerada por lo que, al realizar su interpretación abstracta, la operación *lub* calcula el resultado correcto más conservador.

Por ejemplo, la regla con la función binaria no estricta conjunción $h := a, b.$ se transforma en $\{h := a., h := a, b.\}$. La primera regla asume que sólo el primer argumento se calcula, y la segunda, que se calculan ambos argumentos. En la interpretación abstracta se calcula el resultado conservador de la evaluación de ambas reglas.

3.3 Niveles de análisis

El objetivo de este análisis es obtener información de independencia para simplificar las reglas paralelas. Esta información puede ser explícita, incorporando en el dominio elementos para representar el caracter libre de las variables y la dependencia o independencia entre ellas, o implícita, incorporando en el dominio información de modos. Los modos pueden ser de entrada, salida o entrada/salida. Un modo de entrada o básico (*ground*) significa que el argumento estará cerrado, i.e., sin variables libres, por lo que es independiente.

En esta sección presentamos tres dominios abstractos $\mathcal{D}_1$, $\mathcal{D}_2$ y $\mathcal{D}_3$ para obtener diferentes niveles de información de modos y de independencia. En el diseño de los dominios abstractos hemos comenzado con $\mathcal{D}_1$, que representa información de modos. Sobre esta base, hemos extendido el dominio con información de dependencia y libertad, proponiendo $\mathcal{D}_2$. Finalmente, $\mathcal{D}_3$ incorpora la expresividad de sus predecesores, incluyendo abstracciones para términos de datos con profundidad acotada. Esto permite obtener más información de dependencia que en $\mathcal{D}_2$ puesto que el dominio tiene más expresividad.

Para cada dominio describimos las operaciones más destacadas en el análisis, que son:

- Abstracción (α).
- Menor cota superior (*lub*).
- Unificación abstracta ($unif$).

Además, proporcionamos el orden parcial necesario para cada uno.

3.3.1 Dominio abstracto $\mathcal{D}_1$

En este dominio se incluye la información de modos en los elementos abstractos *ground* y *top*. El primer elemento es un modo de salida y el segundo un modo de entrada/salida, que representa cualquier término del dominio real. Para completar el dominio se añade el elemento *bottom* ($\perp$), que representa un punto de programa inalcanzable durante el cómputo real.

Consideramos como objetos de datos las respuestas computadas de las funciones (i.e., resultados y sustituciones reales). Siguiendo la nomenclatura previa, E es el conjunto potencia de los objetos de datos y D es el conjunto de las propiedades que abstraen a los elementos de E. Definimos los objetos

en el dominio abstracto como los resultados abstractos de las funciones y las sustituciones abstractas. Una sustitución abstracta es un conjunto de vinculaciones *variable / elemento abstracto*, donde un elemento abstracto está en el conjunto $\{top, ground, bottom\}$.

Este dominio no es capaz de exportar las vinculaciones de variables más allá del ámbito local de la cláusula, puesto que no tiene representación para las variables. Podemos ver cómo se materializa esta desventaja en el siguiente ejemplo[11].

Ejemplo 12. *Falta de precisión en* $\mathcal{D}_1$

Consideremos las siguientes reglas, donde f, g, h e i son símbolos de función y t es un símbolo de constructora.

$f(X,Y) := t(g(X,Y), h(X), i(Y))$.
$g(X,X) := true$.
$h(true) := true$.
$i(Z) := Z$.

La interpretación abstracta del punto de entrada $f(X,Y)$ con la sustitución $\lambda_{call} = \{X/top, Y/top\}$ resulta:
para $g(X,Y)$: $\lambda_{call} = \{X/top, Y/top\}$, $\lambda_{succ} = \{X/top, Y/top\}$, $\rho = ground$,
para $h(X)$: $\lambda_{call} = \{X/top\}$ [12] , $\lambda_{succ} = \{X/ground\}$, $\rho = ground$,
para $i(Y)$: $\lambda_{call} = \{Y/top\}$, $\lambda_{succ} = \{Y/top\}$, $\rho = top$,
y para $f(X,Y)$: $\lambda_{succ} = \{X/ground, Y/top\}$ y $\rho = top$.
Por lo tanto, podemos comprobar que este dominio no es lo suficientemente expresivo para inferir la basicidad de Y, ya que las vinculaciones entre variables[13] se conocen sólo en el contexto local de la regla, sin propagarse en las llamadas a función.

A continuación describimos el orden parcial y las operaciones abstractas que operan sobre sustituciones y resultados.

- *Orden parcial*
  $bottom \sqsubseteq ground \sqsubseteq top$

[11] Adaptado de [44].

[12] De aquí en adelante y por brevedad escribiremos en las sustituciones sólo las vinculaciones que afecten a la expresión correspondiente.

[13] I.e., *aliasing*. Se dice que existe *aliasing* entre dos variables cuando la instanciación de una afecta a la de la otra.

$$\lambda_1 \sqsubseteq \lambda_2 \Leftrightarrow \begin{cases} \text{si } X/ground \in \lambda_1 \Rightarrow X/ground \in \lambda_2 \vee X/top \in \lambda_2 \\ \text{si } X/top \in \lambda_1 \Rightarrow X/top \in \lambda_2 \end{cases}$$

que es un retículo completo con elemento superior $\{X/top : X \in dom(\lambda)\}$ y elemento inferior $\{X/bottom : X \in dom(\lambda)\}$, donde $dom(\lambda) = \{X : X/d \in \lambda, d \in D\}$.

- *Abstracción (α)*
  - de un término de datos e: $\alpha(e) = \begin{cases} ground & \text{si } var(e) = \emptyset \\ top & \text{e.o.c.} \end{cases}$ [14]
  - de una sustitución θ: $\mathcal{A}(\theta) = \{X/d : X/e \in \theta, d = \alpha(e)\}$
- *Menor cota superior (lub)*
  - de dos términos abstractos d_1 y d_2:

$lub(d_1, d_2)$	*top*	*ground*	*bottom*	d_1
top	*top*	*top*	*top*	
ground	*top*	*ground*	*ground*	
bottom	*top*	*ground*	*bottom*	
d_2				

  - de dos sustituciones λ_1 y λ_2:
    $lub(\lambda_1, \lambda_2) = \{X/lub(d_1, d_2) : X/d_1 \in \lambda_1 \wedge X/d_2 \in \lambda_2\}$
- *Unificación abstracta (unif)* de dos términos abstractos d_1 y d_2, calculada en términos de la siguiente tabla:

$unif(d_1, d_2)$	*top*	*ground*	*bottom*	d_1
top	*top*	*ground*	*top*	
ground	*ground*	*ground*	*ground*	
bottom	*top*	*ground*	*bottom*	
d_2				

[14] $var(e)$ es la función que devuelve el conjunto de variables en un término e.

Mejora del rendimiento en tiempo de ejecución de $\mathcal{D}_1$

Como veremos en la sección 3.4, los tiempos relativos de análisis del dominio $\mathcal{D}_1$ son bastante buenos. No obstante, proponemos una pequeña modificación en los dominios real y abstracto de los objetos de datos para conseguir una aceleración significativa del procedimiento de análisis a expensas de una pequeña pérdida de precisión.

Consideremos un programa con n definiciones de función F_i. Los nuevos objetos de datos que consideramos son las n-tuplas $(T_1, \ldots, T_n)$, donde T_i son los términos de datos construidos con constructoras de datos[15] y variables. El símbolo raíz de T_i es el símbolo de función de F_i. La aridad de T_i es uno más que la aridad de F_i. E es el conjunto potencia de los objetos de datos. Las propiedades se describen por las n-tuplas de *patrones* [44, 137] $(L_1, \ldots, L_n)$, donde L_i tiene el mismo símbolo raíz que F_i y la misma aridad que T_i. Los argumentos de L_i se restringen a que pertenezcan a un conjunto dotado de orden parcial determinado. Elegimos este conjunto como aquél que contiene los elementos de $\mathcal{D}_1$. Los argumentos de L_i representan las abstracciones de los argumentos de F_i y la abstracción de su resultado.

La principal ventaja de este dominio es que las funciones abstractas son más simples que en el dominio anterior. Además, el procedimiento de búsqueda en la tabla de memoria es más rápido al considerarse patrones en lugar de sustituciones.

Si se aplica este nivel de análisis al último ejemplo se obtiene el mismo resultado. Sin embargo, si se consideran programas con estructuras en los argumentos, se obtienen resultados menos precisos.

3.3.2 Dominio abstracto $\mathcal{D}_2$

Para capturar las ligaduras entre variables construimos este nuevo dominio añadiendo a $\mathcal{D}_1$ dos nuevas propiedades:

- la propiedad para caracterizar una variable libre (*freeness*), y
- la propiedad para caracterizar la vinculación entre dos variables (*aliasing*).

Estas propiedades cubrirán la falta de expresividad de $\mathcal{D}_1$ para la propagación de *aliasing* entre la llamada a función y la evaluación de la función. Añadimos los siguientes elementos: *free* para denotar una variable, y X

[15] Nótese que las constantes son constructoras de datos de aridad 0.

($X \in PVar$, el conjunto de variables de programa) para denotar el *aliasing* seguro de X y una variable de programa dada. Con este dominio es posible expresar lo siguiente en una sustitución abstracta λ:

- Si X/Y $(X, Y \in PVar)$ y $Y/d : d \sqsubset top$ pertenece a λ, entonces X e Y son variables definitivamente dependientes.
- Si $X/free$ $(X \in PVar)$ pertenece a λ y Y/X no pertenecen a λ, entonces X es una variable independiente.
- Si X/Y y Y/top pertenecen a λ, entonces X e Y pueden ser dependientes.

Centrándonos en el ejemplo 12, podemos comprobar el tratamiento más preciso del *aliasing*.

Ejemplo 13. *Gestión del* aliasing *en* $\mathcal{D}_2$

Con $\lambda_{call} = \{X/free, Y/free\}$ para $f(X, Y)$ obtenemos lo siguiente:
para $g(X, Y)$: $\lambda_{call} = \{X/free, Y/free\}$, $\lambda_{succ} = \{X/free, Y/X\}$, $\rho = ground$,
para $h(X)$: $\lambda_{call} = \{X/free\}$, $\lambda_{succ} = \{X/ground\}$, $\rho = ground$,
para $i(Y)$: $\lambda_{call} = \{Y/ground\}$, $\lambda_{succ} = \{Y/ground\}$, $\rho = ground$,
y entonces, para $f(X, Y)$ obtenemos $\lambda_{succ} = \{X/ground, Y/ground\}$, $\rho = ground$.
Nótese que no sólo se conoce la basicidad de Y, sino también del resultado.

Esta propuesta es similar a la desarrollada en [24], en la que no sólo se tiene en cuenta el *aliasing* seguro sino también el *aliasing* posible.

A continuación describimos el orden parcial y las operaciones abstractas que operan sobre sustituciones y resultados.

- *Orden parcial*
  $bottom \sqsubseteq ground \sqsubseteq top$
  $bottom \sqsubseteq free \sqsubseteq top$

$bottom \sqsubseteq X \sqsubseteq top$

$$\lambda_1 \sqsubseteq \lambda_2 \Leftrightarrow \begin{cases} \text{si } X_1/free \in \lambda_1 \Rightarrow X_1/free \in \lambda_2 \vee X_1/top \in \lambda_2 \\ \text{si } X_1/ground \in \lambda_1 \Rightarrow X_1/ground \in \lambda_2 \vee X_1/top \in \lambda_2 \\ \text{si } X_1/top \in \lambda_1 \Rightarrow X_1/top \in \lambda_2 \\ \text{si } X_1/X_2 \in \lambda_1 \Rightarrow X_1/X_2 \in \lambda_2 \vee X_1/top, X_2/top \in \lambda_2 \end{cases}$$

que es un retículo completo con los mismos elementos superior e inferior que $\mathcal{D}_1$.

- *Abstracción (α)*
  - de un término de datos $e = t\theta$:
$$\alpha(e) = \begin{cases} ground & \text{si } var(e) = \emptyset \\ X & \text{si } e \in PVar \wedge e/X \in \theta \\ free & \text{si } e \in PVar \wedge \nexists e/X \in \theta \\ top & \text{e.o.c.} \end{cases}$$
  - de una sustitución θ: $\mathcal{A}(\theta) = \{X/d : X/e \in \theta, d = \alpha(e)\}$
- *Menor cota superior (lub)*
  - de dos términos abstractos d_1 y d_2: [16]

$lub(d_1,d_2)$	top	$ground$	$free$	X_1	$bottom$	d_1
$free$	top	top	$free$	top	$free$	
X_2	top	top	top	si $X_1 = X_2$ $\Rightarrow X_1$ top e.o.c.	X_2	
d_2						

  - de dos sustituciones λ_1 y λ_2:
    $lub(\lambda_1, \lambda_2) = \{X/lub(d_1, d_2) : X/d_1 \in \lambda_1 \wedge X/d_2 \in \lambda_2\}$
- *Unificación abstracta (unif)* de dos términos abstractos d_1 y d_2, calculados en términos de las siguientes tablas:

$unif(d_1,d_2)$	$ground$	$free$	X_1	d_1
$free$	$\{d_2/ground\}$	$\{d_2/d_1\}$	$\{d_2/X_1\}$	
X_2	$\{X_2/ground\}$	$\{X_2/d_1\}$	$\{X_2/X_1\}$	
d_2				

[16] De aquí en adelante sólo incluiremos en las tablas las entradas correspondientes a los nuevos elementos abstractos. El resto de las entradas permanecen iguales que en el dominio anterior.

$unif(d_1, d_2)$	top	$bottom$	d_1
$free$	$\{d_2/top\}$	$\{d_2/bottom\}$	
X_2	$\{X_2/top\}$	$\{X_2/bottom\}$	
d_2			

Incluso cuando no proporcionamos representación para las constantes y constructoras en $\mathcal{D}_1$ y en $\mathcal{D}_2$, descartamos aquellas reglas que no son unificables con la expresión a evaluar realizando unificación concreta, mejorando de este modo la precisión del análisis.

3.3.3 Dominio abstracto $\mathcal{D}_3$

Con el dominio anterior éramos capaces de capturar información acerca de términos básicos, variables y *aliasing* seguro, pero también es interesante incorporar una representación de las constructoras de datos para obtener más información de independencia. Sin embargo, la inclusión de constructoras de datos de profundidad no acotada en sus argumentos provocaría la no terminación del procedimiento de análisis puesto que estaríamos considerando un retículo de profundidad infinita del que no podríamos conseguir en general secuencias de Kleene finitas. Por lo tanto, imponemos una cota a la profundidad de los términos de datos. Para construir $\mathcal{D}_3$ hemos añadido a los elementos de $\mathcal{D}_2$, el elemento abstracto $c(E_1, \ldots, E_n)$, donde c es un símbolo de constructora, y $E_1, \ldots, E_n$ son también elementos abstractos que están limitados a tener una profundidad determinada[17].

Ejemplo 14. *Constructoras como términos abstractos.*

Consideremos el siguiente programa[18], donde f, g, h e i son funciones, r, s y t son constructoras de datos, y a es una constante.

$f(X, Y) := r(g(X), h(X), i(Y))$.
$g(s(W)) := true$.
$g(t(a)) := true$.
$h(X) := \ldots$.
$i(X) := \ldots$.

Supongamos que para la llamada $f(X, Y)$, X está ligada a un término $t(Z)$, donde Y y Z son variables libres. Este punto de entrada se abstrae bajo $\mathcal{D}_2$ como $\lambda_{call} = \{X/top, Y/free\}$. El procedimiento de interpretación abstracta arroja $\lambda_{succ} = \{X/top\}$

[17] Esto fué sugerido en [123].
[18] Adaptado de [123].

para $q(X)$. Sin embargo, la abstracción del punto de entrada bajo $\mathcal{D}_3$ es $\lambda_{call} = \{X/g(Z), Z/free\}$ y el procedimiento de interpretación abstracta arroja $\lambda_{succ} = \{X/g(Z), Z/ground\}$[19] para ella. Esto implica que, en el contexto de la regla para f, se conoce la independencia de X e Y después de la evaluación abstracta de $g(X)$. Podemos aprovechar esta información en el modelo de evaluación paralela para permitir la evaluación paralela de $h(X)$ e $i(Y)$ sin necesidad de ningún test de independencia en tiempo de ejecución.

A continuación describimos el orden parcial y las operaciones abstractas que operan sobre sustituciones y resultados.

- *Orden parcial*
  $bottom \sqsubseteq c(c_1, \ldots, c_n) \sqsubseteq ground \sqsubseteq top$ tal que $c_1, \ldots, c_n \sqsubseteq ground$
  $bottom \sqsubseteq free \sqsubseteq top$
  $bottom \sqsubseteq X \sqsubseteq top$
  $bottom \sqsubseteq c(c_1, \ldots, c_n) \sqsubseteq top$
  $c(c_1, \ldots, c_n) \sqsubseteq c(d_1, \ldots, d_n)$ tal que $c_i \sqsubseteq d_i : 1 \leq i \leq n$

$$\lambda_1 \sqsubseteq \lambda_2 \Leftrightarrow \begin{cases} \text{si } X_1/free \in \lambda_1 \Rightarrow X_1/free \in \lambda_2 \vee X_1/top \in \lambda_2 \\ \text{si } X_1/ground \in \lambda_1 \Rightarrow X_1/ground \in \lambda_2 \vee X_1/top \in \lambda_2 \\ \text{si } X_1/top \in \lambda_1 \Rightarrow X_1/top \in \lambda_2 \\ \text{si } X_1/X_2 \in \lambda_1 \Rightarrow X_1/X_2 \in \lambda_2 \vee X_1/top, X_2/top \in \lambda_2 \\ \text{si } X_1/c(d_{11}, \ldots, d_{1n}) \in \lambda_1 \Rightarrow \; X_1/c(d_{21}, \ldots, d_{2n}) \in \lambda_2 \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad d_{1i} \sqsubseteq d_{2i} \vee \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad X_1/ground \in \lambda_2 \wedge d_{1i} \sqsubseteq d_{2i} \end{cases}$$

  que es un retículo completo con los mismos elementos superior e inferior que en los dominios anteriores.

- *Abstracción (α)*

  - de un término de datos $e = t\theta$, definida por inducción:

$$\alpha^0(e) = \begin{cases} e & \text{si } e \in DC^0 \\ ground & \text{si } e \notin DC^0 \wedge var(e) = \emptyset \\ X & \text{si } e \in PVar \wedge e/X \in \theta \\ free & \text{si } e \in PVar \wedge \nexists e/X \in \theta \\ top & \text{e.o.c.} \end{cases}$$

[19] Nótese que la primera regla para g no es unificable con la llamada $q(X)$ y $\lambda_{call} = \{X/g(Z), Z/free\}$.

$$\alpha^k(e) = \begin{cases} e & \text{si } e \in DC^0 \\ X & \text{si } e \in PVar \wedge e/X \in \theta \\ free & \text{si } e \in PVar \\ c(d_1,\ldots,d_n): & \\ d_i = \alpha^{k-1}(e_i) & \text{si } e = c(e_1,\ldots,e_n) \wedge c \in DC^n \\ top & \text{e.o.c.} \end{cases}$$

- de una sustitución θ: $\mathcal{A}^k(\theta) = \{X/d : X/e \in \theta, d = \alpha^k(e)\}$

- *Menor cota superior (lub)*
  - de dos términos abstractos d_1 y d_2:

$lub(d_1,d_2)$	*ground*	*free*	X_1	d_1
$s(d_{21},\ldots,d_{2n})$	si $d_2 \sqsubseteq$ *ground* $\Rightarrow$ *ground* *top* e.o.c.	*top*	*top*	
d_2				

$lub(d_1,d_2)$	$t(d_{11},\ldots,d_{1m})$ con $s \neq t \vee n \neq m$	$s(d_{11},\ldots,d_{1n})$	d_1
$s(d_{21},\ldots,d_{2n})$	si $d_1,d_2 \sqsubseteq$ *ground* $\Rightarrow$ *ground* *top* e.o.c.	$s(d'_1,\ldots,d'_n)$: $d'_k = lub(d_{1k},d_{2k})$	
d_2			

$lub(d_1,d_2)$	*top*	*bottom*	d_1
$s(d_{21},\ldots,d_{2n})$	*top*	$s(d_{21},\ldots,d_{2n})$	
d_2			

  - de dos sustituciones λ_1 y λ_2:
    $lub(\lambda_1,\lambda_2) = \{X/lub(d_1,d_2) : X/d_1 \in \lambda_1 \wedge X/d_2 \in \lambda_2\}$

- *Unificación abstracta (unif)* de dos términos abstractos d_1 y d_2, calculada en términos de la siguiente tabla, en la que se especifica la profundidad m de los términos abstractos de datos:

$unif^m(d_1,d_2)$	$ground$	$free$	d_1
$s(d_{21},\ldots,d_{2n})$	$s(d'_1,\ldots,d'_n)$: $d'_k = unif^{m-1}(ground, d_{2k})$	$s(d_{21},\ldots,d_{2n})$	
d_2			

$unif^m(d_1,d_2)$	$s(d_{11},\ldots,d_{1n})$	$t(d_{11},\ldots,d_{1m})$ con $s \neq t \vee n \neq m$	d_1
$s(d_{21},\ldots,d_{2n})$	$s(d'_1,\ldots,d'_n)$: $d'_k = unif^{m-1}(d_{1k}, d_{2k})$	$bottom$	
d_2			

$unif^m(d_1,d_2)$	top	$bottom$	d_1
$s(d_{21},\ldots,d_{2n})$	$s(d'_1,\ldots,d'_n)$: $d'_k = unif^{m-1}(top, d_{2k})$	$s(d_{21},\ldots,d_{2n})$	
d_2			

$unif^0(d_1,d_2)$	$ground$	$free$	d_1
$s(d_{21},\ldots,d_{2n})$	$ground$	$\alpha^0(s(d_{21},\ldots,d_{2n}))$	
d_2			

$unif^0(d_1,d_2)$	$s(d_{11},\ldots,d_{1n})$	$t(d_{11},\ldots,d_{1m})$ con $s \neq t \vee n \neq m$	d_1
$s(d_{21},\ldots,d_{2n})$	$unif(\ \alpha^0(t(d_{11},\ldots,d_{1m})),$ $\alpha^0(s(d_{21},\ldots,d_{2n})))$	$bottom$	
d_2			

$unif^0(d_1,d_2)$	top	$bottom$	d_1
$s(d_{21},\ldots,d_{2n})$	si $s(d_{21},\ldots,d_{2n}) \sqsubseteq ground$ $\Rightarrow ground$ top e.o.c.	$\alpha^0(s(d_{21},\ldots,d_{2n}))$	
d_2			

Nótese que $unif$ aplicado a términos no unificables arroja como resultado *bottom*, que representa en este caso fallo de unificación.

3.4 Resultados

En esta sección presentamos los resultados del análisis de varios programas Babel significativos que hemos seleccionado. Los resultados se han obtenido usando un intérprete abstracto de programas Babel escrito en Prolog. Se han implementado cada uno de los dominios y las operaciones abstractas asociadas a ellos. Cada programa se ha analizado con $\mathcal{D}_1^*$ (el dominio $\mathcal{D}_1$ mejorado), $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3^1$ ($\mathcal{D}_3$ con cota de profundidad 1), $\mathcal{D}_3^2$ ($\mathcal{D}_3$ con cota de profundidad 2) y con $\mathcal{D}_3^3$ ($\mathcal{D}_3$ con cota de profundidad 3). Los programas con funciones primitivas no estrictas se han transformado como se indicó en la sección 3.2.2, de manera que aparecen nuevas funciones intermedias.

En la tabla 3.1 mostramos las medidas que hemos obtenido del análisis de los programas. La primera columna contiene el nombre del programa y la primera fila el de los dominios. Para cada dominio hay tres campos de información: *Time*, *Call* y *Succ*. El campo *Time* contiene la relación entre el tiempo necesario para el análisis con el dominio $\mathcal{D}_1^*$ y el dominio considerado. El campo *Call* muestra la relación entre los patrones de llamada inferidos y los del dominio $\mathcal{D}_1^*$. El campo *Succ* contiene el porcentaje de modos útiles inferidos (para los propósitos de la paralelización). Aparte de estos campos, en las tres versiones del dominio $\mathcal{D}_3$ añadimos el campo *Term* que muestra una medida de la cantidad de términos de datos compuestos inferidos hasta una profundidad determinada.

Observando la tabla podemos extraer las siguientes conclusiones:

- Para algunos programas de prueba como *Derivación Simbólica*, *Números de Fibonacci* y *Matriz*, obtenemos la información necesaria para simplificar todas las condiciones de las reglas paralelas. El hecho de que ello se consiga con un dominio tan simple como $\mathcal{D}_1^*$ puede parecer sorprendente en el caso de *Matriz*. Sin embargo, esto es debido a la mayor expresividad declarativa del programa Babel, en el que no aparecen las variables auxiliares necesarias en Prolog para guardar resultados intermedios. Estas variables provocan la pérdida de información en la propagación de las sustituciones abstractas que impiden obtener el nivel de información del caso lógico-funcional con un dominio y análisis equivalentes.

- Para algunos programas, con el dominio $\mathcal{D}_1^*$ se obtienen los mismos resultados útiles para la paralelización que en el resto de dominios con un ahorro de tiempo significativo (entre el 17% y el 84% como se puede

Programa	$\mathcal{D}_1^*$		$\mathcal{D}_1$		$\mathcal{D}_2$		$\mathcal{D}_3^1$		$\mathcal{D}_3^2$		$\mathcal{D}_3^3$	
	Time		Time		Time		Time	Term	Time	Term	Time	Term
	Call	Succ	Call	Succ	Call	Succ	Call	Succ	Call	Succ	Call	Succ
Derivación	1.00		1.19		1.22		1.22	0	1.26	0	1.41	0
Simbólica	1.00	100	1.00	100	1.00	100	1.00	100	1.00	100	1.00	100
Función de	1.00		175		235		257	93	290	823	309	1814
Ackermann	1.00	67	2.50	86	3.50	100	4.50	100	4.50	100	4.50	100
Números de	1.00		6.25		5.75		6.32	30	6.51	30	7.01	30
Fibonacci	1.00	100	4.00	100	4.00	100	6.00	100	6.00	100	6.00	100
Árboles	1.00		3.68		15.7		18.0	0	17.9	0	17.9	0
Balanceados	1.00	63	1.00	63	1.50	39	1.50	39	1.50	39	1.50	39
Producto	1.00		1.46		1.51		5.90		6.15		6.03	
vector-matriz	1.00	100	1.00	100	1.00	100	1.00	100	1.00	100	1.00	100

Tabla 3.1: Resultados del análisis de los programas de prueba.

derivar de la tabla 3.1). La clave que explica esta mejora es que $\mathcal{D}_1^*$ gestiona sólo patrones que se construyen con términos de datos.

- $\mathcal{D}_3$ se comporta bastante bien en la inferencia de un número mayor de patrones de llamada y consigue información más refinada acerca de los términos compuestos sin una sobrecarga excesiva si lo comparamos con $\mathcal{D}_2$. La diferencia de tiempo con respecto a $\mathcal{D}_1$ es en ocasiones de dos órdenes de magnitud, que se explica no sólo por el número de patrones inferidos, sino también por las operaciones abstractas más complejas. Como ventaja, el número de términos compuestos inferidos es del orden de 10^3, mientras que con $\mathcal{D}_1^*$ no se infiere ninguno.

- $\mathcal{D}_1$ y $\mathcal{D}_2$ representan un nivel de análisis intermedio, con los cuales se obtiene información para una paralelización eficiente[20], pero menor información de tipos para la mejora del componente secuencial si se compara con $\mathcal{D}_3$.

- Podemos encontrar una paradoja en *Árboles Balanceados* con respecto al número de modos de éxito inferidos. A partir del dominio $\mathcal{D}_2$, el porcentaje de modos de éxito inferidos decrece. Esto se debe a que estamos calculando la relación entre los modos de éxito y los *modos de llamada*. Puesto que el número de modos de llamada crece mientras que no crece el número de modos de éxito inferidos en la misma proporción, el porcentaje disminuye. De hecho, el número de modos de éxito inferidos aumenta.

Sumario

En este capítulo se ha presentado la aplicación de la interpretación abstracta al análisis de independencia de programas Babel impaciente, desarrollando tres dominios abstractos con los que se consiguen diferentes niveles de análisis. Se ha estudiado el rendimiento de estos niveles analizando varios programas de prueba. Se concluye que, para algunos programas, con el dominio $\mathcal{D}_1^*$ se obtienen los mismos resultados útiles para la paralelización que en el resto de dominios. $\mathcal{D}_1$ y $\mathcal{D}_2$ representan un nivel de análisis intermedio, con los cuales se obtiene suficiente información para una paralelización eficiente, pero menor información de tipos para la mejora del componente secuencial si se compara con $\mathcal{D}_3$.

[20] A partir de un 63% de modos de éxito inferidos, alcanzado en ocasiones el 100%.

Un punto de trabajo futuro interesante en términos de la eficiencia en tiempo de análisis sería la implementación de $\mathcal{D}_3$ con la misma aproximación que la tomada en la implementación de $\mathcal{D}_1^*$, en base a considerar sólo patrones en las llamadas y retornos de funciones, en lugar de patrones y sustituciones. Este nivel de análisis se podría implementar como el que planteamos en [139] en el ámbito de la programación lógica. El objetivo es conseguir un analizador rápido que use la misma técnica de las implementaciones basadas en máquinas abstractas, de modo que se analicen los programas mediante ejecución simbólica.

Otro punto interesante es el tratamiento de funciones no estrictas definidas por el usuario (i.e., con el mecanismo de evaluación perezoso) y extender el análisis de forma que incorpore información de estricticidad [125, 126] o demanda [75].

Finalmente, la interpretación abstracta se podría utilizar en el análisis de granularidad de expresiones en la línea de [49], desarrollado en el ámbito de la programación lógica.

II

Explotación de paralelismo

Capítulo 4

La máquina abstracta paralela PEBAM

En este capítulo se presenta una máquina abstracta paralela sobre un modelo de memoria compartida. El capítulo comienza con un breve repaso de las técnicas usadas en implementaciones de máquinas lógicas y funcionales secuenciales, relacionándolas con las técnicas de ejecución paralela utilizadas en programación lógica. Se desarrolla el modelo de ejecución de una máquina paralela basada en pilas que retiene las optimizaciones de las máquinas lógicas y funcionales secuenciales, fundamentalmente durante el cómputo hacia atrás en la desasignación de memoria. Su comportamiento es similar al de una máquina funcional cuando se determinan cómputos deterministas estáticamente, mediante análisis, o dinámicamente, mediante el corte dinámico. Finalmente, se describe su arquitectura, el repertorio de instrucciones y la compilación de programas paralelos Babel a instrucciones máquina.

4.1 Introducción

En esta sección se comentarán las implementaciones secuenciales en programación lógica y funcional y el problema que aparece en la extensión de una implementación de una máquina secuencial a una paralela.

4.1.1 Máquinas abstractas secuenciales basadas en pilas

Máquinas de programación lógica

En 1983, D.H.D Warren diseñó la WAM (*Warren Abstract Machine* [161][1]), una máquina abstracta basada en pilas para la ejecución compilada de Prolog que se ha convertido en el estándar sobre el que se basan diseños posteriores. Su diseño integra con un alto grado de eficiencia el paradigma declarativo de Prolog en una arquitectura tipo Von Neumann.

Esta máquina incorpora las siguientes ideas:

- Codificación en las instrucciones máquina de casos particulares de la unificación.
- Gestión dinámica de memoria con recuperación automática en el cómputo hacia atrás que minimiza la operación de recolección de basura.
- Representación de datos marcados con su tipo para optimizar la unificación y la selección de alternativas.
- Identificación del determinismo de los programas especializando el control de flujo en la compilación.

Más tarde han aparecido otros trabajos que mejoran el diseño original de la WAM, consiguiendo implementaciones aún mas eficientes [137]. Estas implementaciones reducen la granularidad de las instrucciones y optimizan la identificación de determinismo y la especialización de la unificación mediante análisis de flujo de datos.

Máquinas de programación lógico-funcional

Los primeros trabajos sobre la implementación del estrechamiento se basaron en la representación de grafos [93], estructuras costosas en la gestión de memoria dinámica.

En [101] se describe la primera máquina de estrechamiento (*Narrowing Machine (NM)*) basada en pilas que incorpora las ideas principales de la WAM. La *NM* dispone de una *pila de ejecución* (*run-time stack* o simplemente pila (*stack*)) que almacena información de control para el cómputo hacia adelante y hacia atrás. Cada vez que se produce una llamada a una función con varias alternativas no excluyentes se almacena un *punto de elección* (*choice point*) en la pila, que contiene la información de control

[1] En [2] se puede encontrar una descripción de este diseño.

necesaria para probar las siguientes alternativas del procedimiento. Como en las máquinas de reducción, en la pila se almacena cada uno de los *entornos* (*environment*) asociado a una llamada a función, que contienen los argumentos y las variables locales de la función, así como la información de control necesaria para continuar el cómputo hacia adelante después de la evaluación de la función.

En el siguiente ejemplo mostramos un caso representativo de la gestión dinámica de memoria de la *NM*.

Ejemplo 15. *Gestión dinámica de memoria en la* NM
En la figura 4.1 se muestra un programa Babel y la organización de la pila en dos momentos (*(a)* y *(b)*) de la evaluación de la expresión $f(X)$. Las flechas de trazo continuo apuntan al punto de continuación (*Continuation Point (CP)*) en caso de evaluación con éxito, mientras que las flechas de trazo discontinuo apuntan a la siguiente alternativa en caso de fallo.

En *(a)* se describe la situación alcanzada en el cómputo hacia adelante antes de iniciar la evaluación de h en el cuerpo de la regla de f. En la pila se ha creado un entorno para la llamada a f e inmediatamente después el entorno para la evaluación de g. El punto de continuación posterior a la evaluación de g se almacena en el entorno de g. Sobre este entorno se ha creado un punto de elección para g, puesto que las dos reglas de su definición representan alternativas excluyentes[2]. La siguiente alternativa a probar en caso de fallo se almacena en este punto de elección. Finalmente, el entorno correspondiente a h se almacena en la cima de la pila.

En *(b)* se describe la situación alcanzada después de evaluar la segunda alternativa para g. La evaluación de h, cuyo segundo argumento es c (generado por la primera regla de g) produce fallo de unificación. La siguiente alternativa a probar está indicada en el *último* punto de elección almacenado en la pila (apuntado por el registro B). Puesto que la alternativa anotada en el punto de elección es la última posible para la evaluación de g, el punto de elección se elimina después de restaurar el estado. La eliminación del entorno de h y del punto de elección para g se realiza de

[2]La restricción de no ambigüedad (c.f. Apéndice A) se cumple para este programa, puesto que las partes izquierdas de las reglas no son unificables (restricción de no superposición).

manera eficiente restaurando simplemente el contenido de los registros E y B. La restauración del estado previo a la evaluación de g se realiza deshaciendo las unificaciones a partir de ese punto; en este caso, la instanciación de X a c. Estas unificaciones se anotan en el *trail*, una pila que registra las variables que sufren instanciación en el cómputo hacia adelante. El espacio de *trail* utilizado se recupera restaurando el registro de cima de *trail* que se almacena en el entorno. La evaluación de g con la segunda regla devuelve d, que permite el éxito de la unificación de la expresión $h(c, d)$ con la regla de definición para h.

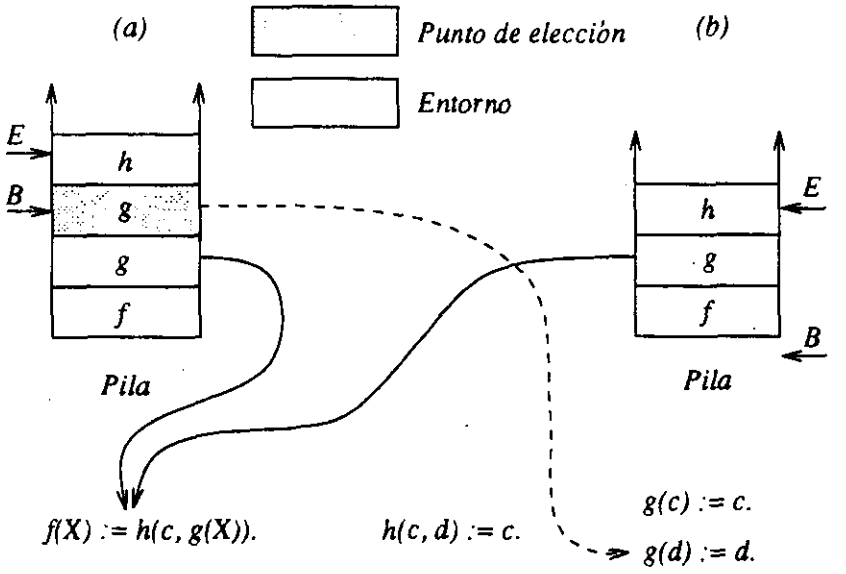


Figura 4.1: Ejemplo de gestión de la pila.

En este ejemplo se muestra la gestión eficiente del cómputo hacia atrás que se realiza en la NM. Ello es posible por las estructuras de datos basadas en pilas que se han utilizado (pila, *trail*, ...). El principio en que se funda esta arquitectura es colocar las estructuras de datos *nuevas* sobre las *viejas* (condición de orden o de *precedencia*), de manera que las que se descarten durante el cómputo hacia atrás sean las que estén en la cima de la pila. Una estructura de datos asociada a una función f es más nueva que otra asociada a otra función g siempre que f se evalúe después de g en el orden de estrechamiento secuencial (en profundidad y de izquierda a derecha). En

particular, el último punto de elección en la pila siempre corresponde a la próxima alternativa a probar en el orden de evaluación secuencial[3].

4.1.2 Extensión de máquinas abstractas secuenciales basadas en pilas a paralelas

Una máquina de estrechamiento secuencial se puede extender al caso paralelo de varias formas:

- Un sistema multisecuencial con un *trabajador* para la evaluación paralela de cada expresión. Cada trabajador es una máquina de estrechamiento secuencial extendida con los mecanismos necesarios para la explotación de paralelismo. Esta solución en la explotación de paralelismo es simple puesto que asume recursos ilimitados y además se mantiene la eficiencia del trabajador secuencial. Sin embargo, la gestión de la memoria dinámica para la creación de trabajadores es costosa[4]. Además, esto provoca de nuevo un problema que se evita con las implementaciones basadas en pilas, es decir, la fragmentación de memoria[5].

- Un sistema multisecuencial con pilas compartidas. En [80] se proponen pilas compartidas (*meshed stacks*) por varios trabajadores para disminuir el consumo de memoria, pero se necesita un recolector de basura que añade un costo computacional extra[6].

- Un sistema multisecuencial con trabajadores para la evaluación consecutiva de diferentes expresiones paralelas. Esto introduce otros problemas en la desasignación de memoria (*garbage slot* y *trapped goal* [71]) si se admite almacenar en una misma pila expresiones sin respetar la condición de precedencia del caso secuencial, es decir, si se solicitan nuevas alternativas de un punto de elección que no se encuentre en la

[3] Las alternativas de cada función se prueban en el orden textual en el que aparecen en el programa.

[4] Si este sistema se implementa en un sistema paralelo que simule el sistema de estrechamiento paralelo, la gestión de la memoria dinámica se cede al sistema operativo subyacente y su eficacia recaerá en última instancia en el sistema operativo.

[5] En el diseño de sistemas paralelos no se debe olvidar la capa software o hardware que subyace a la implementación, que puede significar el trasladar un problema de un nivel abstracto a otro (diseño del sistema paralelo al sistema operativo subyacente).

[6] En definitiva, se pierde realmente la pila como estructura de datos y pasa a ser un nuevo *heap*.

cima de la pila. Para prevenir estos problemas, se debe mantener el orden de evaluación secuencial de las estructuras de datos en la pila, de forma que las alternativas en la pila se encuentren en el orden de evaluación secuencial de *backtracking*. Sin embargo, no se puede establecer un orden entre cualquier par de expresiones paralelas como mostramos en el siguiente ejemplo.

Ejemplo 16. *Problema de la explotación de paralelismo.*
En la figura 4.2 se muestra la evaluación de una expresión *exp* en la que intervienen los trabajadores T_1 y T_2. *exp* contiene las expresiones *f*, *g* y *h* en el orden de estrechamiento secuencial. En el ejemplo se muestra que el trabajador T_1 ha finalizado con éxito la evaluación de *f* y *g* mientras *h* se está evaluando en T_2. Si la evaluación de *h* provoca fallo, la siguiente alternativa en el orden de evaluación secuencial está en el punto de elección de *f*, no el de *g*, que se encuentra en la cima de la pila de T_1.

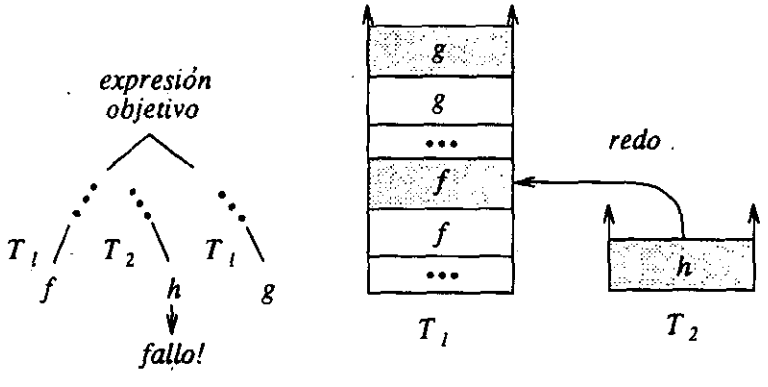


Figura 4.2: Problema de la explotación de paralelismo.

Estas consideraciones muestran el problema que existe en la extensión del caso secuencial al paralelo. Se han propuesto varios sistemas paralelos para Babel basados en grafos [92, 140, 95], pero ninguno retiene las características de un sistema basado en pilas como el sistema paralelo de Prolog descrito en [69]. En las próximas secciones se desarrolla una máquina abstracta paralela extendiendo la máquina secuencial descrita en [101] con técnicas de ejecución paralela similares a las desarrolladas en [69].

4.2 Modelo de ejecución de la máquina abstracta paralela

En esta sección proponemos el modelo de ejecución de la PEBAM (*Parallel Eager Babel Abstract Machine*), una máquina abstracta paralela basada en pilas (figura 4.3) que es un sistema multisecuencial de n trabajadores que cooperan a través de un soporte de comunicaciones[7]. Cada trabajador es una máquina de estrechamiento secuencial extendida con los mecanismos necesarios para la evaluación consecutiva de expresiones en el entorno paralelo.

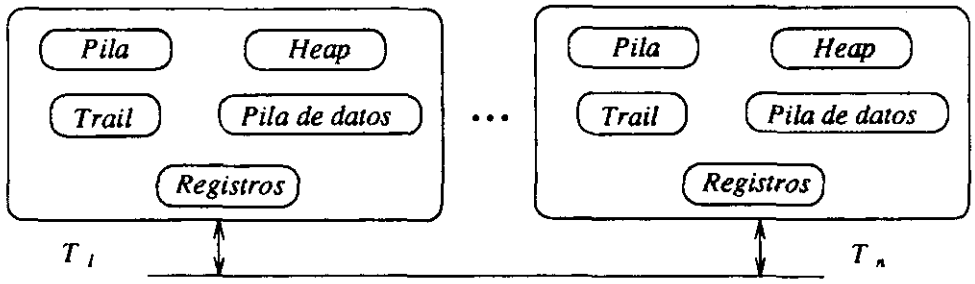


Figura 4.3: Máquina abstracta paralela.

4.2.1 Cómputo hacia adelante

Para evaluar la expresión objetivo se designa un trabajador que actúa como una máquina de estrechamiento secuencial hasta que se alcanza un punto de activación de paralelismo (*fork point*). Hasta este punto, el orden de las estructuras de datos en su pila es el de estrechamiento secuencial. A continuación, el trabajador pone a disposición del resto de trabajadores las expresiones paralelas, conservando la primera en el orden de evaluación secuencial para su evaluación local. Dependiendo de los recursos disponibles, la totalidad o un subconjunto de las expresiones paralelas disponibles se ejecutarán en trabajadores remotos. Después de la evaluación local es posible que aún queden expresiones que no hayan sido aún evaluadas. El trabajador inicia entonces la evaluación de la expresión más vieja disponible para permitir subsiguientes evaluaciones locales. Los trabajadores remotos que

[7]En este nivel de abstracción consideramos el soporte de comunicaciones sin hacer referencia a su implementación, por ejemplo, con enlaces serie, bus o multibús, etc.

hayan finalizado su cómputo pueden buscar más trabajo de manera que respeten el orden de las estructuras de datos en sus pilas. Cuando ha finalizado con éxito la evaluación de las expresiones paralelas se alcanza el punto de reunión (*join point*).

4.2.2 Cómputo hacia atrás (*backtracking*)

El cómputo hacia atrás se activa cuando se produce fallo o se solicitan nuevas alternativas. Pueden ocurrir los dos casos siguientes:

- La evaluación de una expresión paralela de un punto de activación de paralelismo falla (estado *inside* [69]). Debido a que las expresiones paralelas son independientes, ninguna otra alternativa de las expresiones del punto de activación de paralelismo puede solucionar el fallo calculado. En este caso se puede descartar la evaluación de estas expresiones, es decir, se realiza *backtracking semi*-inteligente [69].

- Un cómputo secuencial no dispone de más alternativas debido a que el punto de elección inmediatamente inferior es anterior al último punto de reunión (estado *outside* [69]). Se busca la siguiente alternativa en el orden de evaluación secuencial explorando las expresiones correspondientes a este punto a partir de la expresión que provocó el punto de reunión. Cuando se encuentra, se descartan las expresiones posteriores a la que posee alternativas (operación de descarte de cómputos) y se pide una nueva alternativa de un modo similar al secuencial, esto es, inspeccionando el punto de elección correspondiente. Si se producen subsiguientes fallos se buscan nuevas alternativas en el mismo orden.

4.2.3 Planificación de trabajo

Para mantener el orden del *backtracking* secuencial en las estructuras de datos alojadas en las pilas se impone una restricción en las expresiones que un trabajador pueda evaluar (condición de precedencia). El procedimiento de planificación de trabajo determina si una expresión paralela cumple la condición de precedencia. Para ello es necesario plantear un orden entre las estructuras de datos que se alojan el la pila. Describiremos en primer lugar una representación para el orden del caso secuencial y a continuación la extenderemos al paralelo.

Orden entre estructuras de datos

En lo que sigue un nodo representa las estructuras de datos que se colocan en la pila necesarias para iniciar la evaluación de una función.

- *Orden del caso secuencial.*
  Cada estado intermedio de un cómputo de estrechamiento secuencial se representa por medio de un árbol *And-Or*, en el que se alternan nodos *And* y *Or* como se puede ver en la figura 4.4-a. Un nodo *And* representa la parte izquierda de una regla cuyos hijos son las aplicaciones de su parte derecha. Estas aplicaciones son los nodos *Or* cuyos hijos son las cabezas de las reglas unificables con cada nodo *Or*. La construcción de este árbol está guiada por el mecanismo de cómputo impaciente. Se define un orden parcial cuya relación de orden es la siguiente: un nodo a es más nuevo que (o está a la derecha de) un nodo b siempre que b se encuentre primero que a en el árbol de evaluación según el orden de estrechamiento impaciente en profundidad y de izquierda a derecha[8].

- *Orden del caso paralelo.*
  Cada estado intermedio de un cómputo de estrechamiento paralelo se representa añadiendo al caso anterior nodos *Fork* y *Join*. Un nodo *Fork* representa el punto de activación de paralelismo de sus hijos *Or*. Un nodo *Join* representa un punto de reunión tras una evaluación paralela (véase la figura 4.4-b). En este esquema, un nodo *And* puede tener nodos hijos *Or* y *Fork*. Se establece una relación de orden "es más viejo que" entre los nodos *Or* y sus hijos, entre los hijos de un nodo *Fork* y entre los hijos de un nodo *And* (en estos dos últimos casos en el orden de estrechamiento secuencial). Sin embargo, no es posible aplicar esta relación entre los descendientes de un nodo *Fork* hasta que no se alcance el nodo *Join* correspondiente. Esto es debido a que la configuración del árbol de evaluación puede variar antes de alcanzar la configuración de éxito cuando, en presencia de *backtracking*, se prueban diferentes configuraciones del árbol de evaluación.

Condiciones de evaluación

Seguimos el orden parcial expuesto para asignar la evaluación de una expresión a un trabajador que cumpla la condición de precedencia, esto es, que

[8]La relación es simétrica para "es más viejo que" (o "está a la izquierda de").

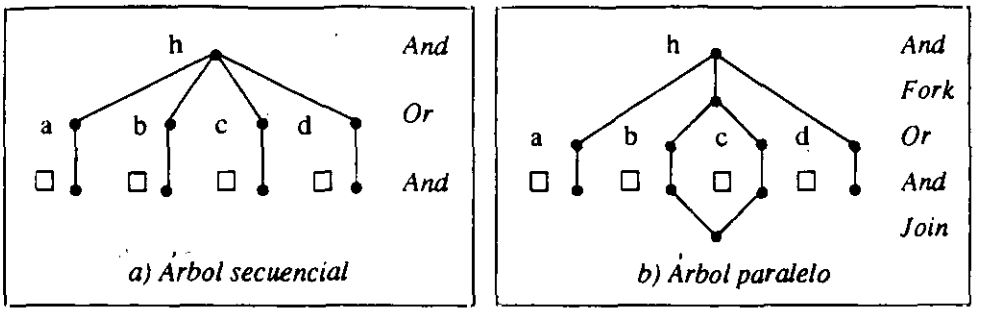


Figura 4.4: Árboles de evaluación para los casos secuencial y paralelo.

durante el *backtracking* tenga en la cima de su pila la estructura de datos a descartar o con alternativas pendientes.

Los siguientes casos condicionan la evaluación de expresiones.

- Si un trabajador ha finalizado la evaluación de una expresión e de su punto de activación de paralelismo activo (i.e., no se ha alcanzado el punto de reunión) sólo puede evaluar expresiones del punto de activación a la derecha de e.

- Si un trabajador ha evaluado una expresión e de un punto de activación remoto que aún contiene otras expresiones, se permite que evalúe otra expresión a la derecha de e. Esta situación se muestra en la figura 4.5-a, en donde se representan árboles de evaluación paralela con arcos etiquetados con el identificador del trabajador T que ha evaluado una determinada expresión. En esta figura las expresiones que T puede evaluar se anotan con *Permitido*, mientras que las que no cumplen la condición de precedencia se anotan con *Prohibido*). Las expresiones a su izquierda no son adecuadas puesto que son más viejas en el orden de *backtracking*.

- Si un trabajador T ha evaluado una expresión de un punto de activación de paralelismo, que puede haber alcanzado el punto de reunión, y que pertenece a un segmento computacional de otro punto de activación ancestro, no se permite que evalúe expresiones de éste hasta que el segmento computacional haya alcanzado el punto de reunión (véase la figura 4.5-c).

- Finalmente, un trabajador que no haya evaluado ninguna expresión puede evaluar cualquiera disponible.

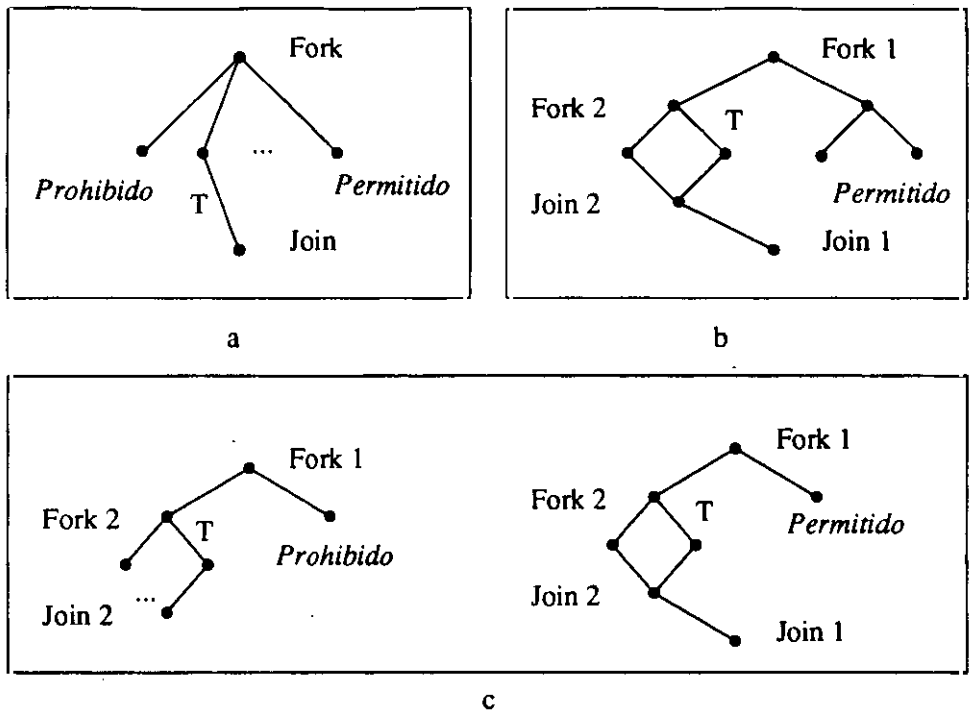


Figura 4.5: Condiciones en la planificación de trabajo.

Como conclusión, decimos que un trabajador T puede evaluar una expresión de un determinado punto de activación si todos los segmentos computacionales en los que ha tomado parte el trabajador T han alcanzado los correspondientes puntos de reunión. Es decir, puede evaluar no sólo los hijos inmediatos del correspondiente punto de activación de paralelismo, sino cualquier descendiente (véase la figura 4.5-b). Aun cuando la implementación de estas condiciones pueda parecer costosa, se puede comprobar que la estrategia de planificación de trabajo que desarrollamos se puede implementar eficientemente (véase el apéndice D).

Selección de las expresiones

Una vez identificadas las expresiones que cumplen las condiciones de evaluación en un determinado trabajador, presentamos el criterio por el que se selecciona una de ellas para su evaluación.

Hay dos casos en la selección de una expresión en un punto de activación de paralelismo para que la evalúe un trabajador:

- Si no hay ninguna expresión en evaluación a la izquierda de otra, ésta es la más adecuada puesto que, al ser la expresión más antigua disponible, el trabajador puede realizar subsiguientes evaluaciones de expresiones de este punto de activación.

- Si hay expresiones en evaluación a la izquierda de otras expresiones disponibles, la estrategia a seguir no es tan evidente como en el caso anterior. En este caso se pueden elegir dos alternativas:

  - Seleccionar la expresión más a la izquierda realizando una búsqueda de trabajo, que no asegura que tenga éxito.
  - Seleccionar la expresión disponible más a la izquierda, que tiene definitivamente trabajo.

  Hemos elegido la segunda alternativa por dos motivos.

  - Para asegurar que se puede evaluar el máximo número de expresiones en el trabajador remoto.
  - Para evitar el trabajo especulativo debido a las expresiones a la derecha que no es útil cuando se calcula un resultado definitorio o de fallo a la izquierda.

Estrategia de planificación de trabajo

Podemos considerar dos estrategias principales de planificación de trabajo.

- *Centralizada.*
  Un controlador se hace cargo del reparto de trabajo en base a la información que se posea de la carga de trabajo del sistema [4].

- *Distribuida.*
  Los trabajadores son los encargados de explorar su entorno en busca de trabajo [100].

La primera alternativa es adecuada para un sistema en el que la transmisión de la información necesaria del estado de los trabajadores no sea costosa, como en un sistema de memoria compartida. Los sistemas centralizados son más complejos en su diseño puesto que se trata de predecir de manera global la planificación que produce una carga de trabajo más homogénea.

La segunda alternativa es adecuada para sistemas en los que la transmisión de esa información sea costosa, como en sistemas de memoria distribuida. De esta manera, el conocimiento del estado de la carga de trabajo del sistema es preferentemente local en cuanto que para eliminar el tráfico de mensajes se prescinde de cierta información global.

Hemos desarrollado una planificación global gestionada localmente por los trabajadores, ya que consideramos una máquina sobre un modelo de memoria compartida. Los detalles de la implementación de esta estrategia se pueden encontrar en el apéndice D.

Representación de la información de trabajo disponible

Usualmente se utilizan estructuras de datos como pilas o colas para la información del trabajo disponible. En su lugar, hemos utilizado la cadena de marcadores de paralelismo en los que se guarda la información necesaria del punto de activación de paralelismo al que corresponde, obviando otras estructuras de datos. La diferencia fundamental con la gestión de otras estructuras de datos es que en lugar de la búsqueda en la cola de tareas siguiendo direcciones consecutivas, se sigue la cadena de marcadores de paralelismo en un esquema de indirección (las tareas se encuentran por búsqueda en el marcador hijo de paralelismo) en la búsqueda de trabajo.

En la figura 4.6 se puede observar cómo se proyecta una cola de tareas típica sobre la pila, en la que residen una serie de marcadores de paralelismo (*PF - parcall frames*) que contienen las expresiones disponibles para evaluarse (*slot ready*). Las expresiones que se evalúan primero son las más viejas, como en la cola de tareas que se presenta en la misma figura.

Podemos exponer algunas ventajas de nuestra propuesta. En primer lugar, el trabajador local (que en general está realizando trabajo) se libera de la gestión de la cola de tareas y, en segundo lugar, se necesita menos gestión de memoria. Sin embargo, el esquema de indirección es más lento que el acceso directo a una cola de tareas.

4.2.4 Descarte paralelo de cómputos

El descarte de cómputos puede ocurrir en los casos siguientes:

- En el fallo de la evaluación de uno de los argumentos de una función estricta (estado *inside*). En este caso se descarta el resto de argumentos.

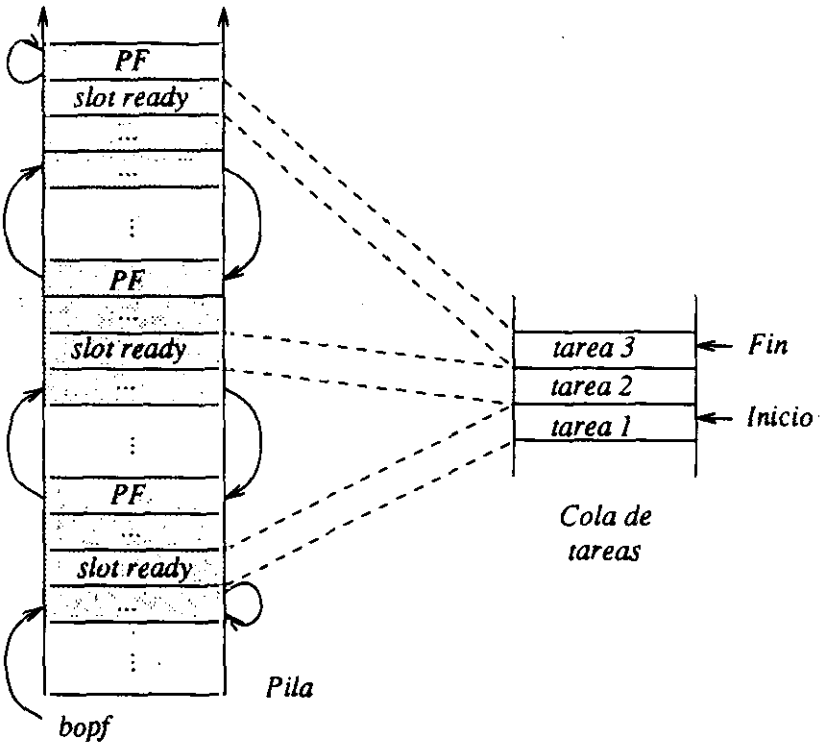


Figura 4.6: Cola de tareas imbuida en la pila.

- En el fallo de la evaluación de uno de los argumentos de una función no estricta (estado *inside*). En este caso se descartan los argumentos a la derecha del que provocó el fallo.
- En el cálculo de un resultado definitorio de una función no estricta (estado *inside*). En este caso se descartan los argumentos a la derecha de la evaluación definitoria.
- En la búsqueda de alternativas en un punto de activación durante el cómputo hacia atrás (estado *outside*). En este caso se descartan todos los argumentos a la derecha del que posee alternativas.

Hay dos fases en el descarte de cómputos:

- *Fase de notificación.*
  Consiste en el envío de la información necesaria para el descarte de segmentos computacionales alojados en las pilas de los trabajadores implicados.

- *Fase de desasignación.*
  Consiste en la desasignación de estructuras de datos y desvinculación de variables, similar al caso secuencial.

Estas dos fases se describen en el procedimiento de descarte de cómputos.

Procedimiento de descarte de cómputos

Se puede comprobar que existe un invariante del número de notificaciones de descarte que llegan a un trabajador debido al orden establecido en la pila. Gracias a ello podemos diseñar un procedimiento eficiente de descarte de cómputos que difunde las notificaciones de descarte esperando sólo cuando es realmente necesario. El procedimiento permite el envío de notificaciones a todos los trabajadores que han intervenido en un punto de activación, lo que implica un descarte paralelo de cómputos. Una vez que se han enviado todas las notificaciones de descarte se produce una espera hasta que se hayan descartado todos los cómputos. No es necesario que estas notificaciones contengan también el segmento al que van referidas puesto que el número de notificaciones de descarte que llegan a un trabajador coincide con el número de segmentos computacionales que se deben descartar. En general, el orden en el que las notificaciones de descarte llegan a un trabajador no coincide con el orden de los segmentos computacionales alojados en la pila. Sin embargo, esto no presenta ningún problema tal como se muestra en la siguiente distinción de casos posibles.

- Descarte local de expresiones. Este caso corresponde al descarte de los cómputos locales o evaluados por trabajadores remotos.
  - Evaluación local de todas las expresiones.
    El procedimiento de descarte es similar al caso secuencial.
  - Evaluación remota.
    * Si una de las expresiones se ha evaluado en un trabajador remoto y éste no ha evaluado ninguna otra expresión (por la condición de precedencia, porque no haya terminado su evaluación o porque no haya más trabajo disponible), el trabajador remoto recibe una única notificación de descarte que corresponde al segmento computacional superior en su pila[9].

[9]Este trabajador, a su vez, puede haber generado otros puntos de activación y las notificaciones de descarte en este caso se reducen al caso general.

  - Si varias expresiones se han evaluado en el mismo trabajador remoto, a éste le llegará el número de notificaciones de descarte correspondiente al número de expresiones que ha evaluado[10]. En este caso, si las notificaciones se envían de las expresiones más antiguas a las más nuevas, las notificaciones llegan al trabajador remoto en sentido inverso al orden en que se alojan los segmentos computacionales respectivos en la pila.

- Descarte remoto de expresiones.
  Este caso corresponde a la llegada de notificaciones de descarte desde trabajadores ancestros.

  Debido a la condición de precedencia, un trabajador ha podido intervernir en los puntos de activación del segmento computacional, pero no de otros correspondientes a otros segmentos. Esto significa que el número de notificaciones de descarte que recibe el trabajador corresponde a las expresiones que han evaluado y que se encuentran en la cima de la pila. Si el trabajador se ha involucrado en un segmento computacional diferente, sólo puede haber sido en la evaluación de una expresión del punto de activación al que pertenece el anterior segmento computacional, por lo que si éste debe ser descartado también debe serlo el otro, puesto que es más nuevo.

Con este procedimiento se evita el problema que aparece cuando la velocidad de creación de tareas es semejante a la velocidad de descarte de tareas, lo cual da lugar a segmentos computacionales inútiles que avanzan ocupando recursos sin llegar a ser descartados en un tiempo razonable[11].

Repercusión del descarte de cómputos en la planificación de trabajo

Para mantener la coherencia del marcador de paralelismo, éste se bloquea cada vez que un trabajador lo actualiza. Esto implica la suspensión del procedimiento de búsqueda de trabajo cuando un marcador de paralelismo

[10] Es posible que la última expresión se encuentre aún evaluándose.

[11] E incluso a no llegar siquiera a ser descartados si la velocidad de creación de tareas es igual o superior a la velocidad de descarte de tareas.

En [56] se describen dos alternativas a la propagación de notificaciones de descarte que solucionan los problemas que aparecen en un sistema que no hace uso de las técnicas aquí mencionadas, donde es necesario proponer algoritmos más complejos.

bloqueado se intenta consultar. Cuando se desbloquea el marcador, pueden ocurrir tres casos:

- Se cierra el marcador (marcador inactivo).
- El marcador permanece con trabajo pendiente (marcador activo).
- Se descarta el marcador.

Los dos primeros casos no representan ningún problema para el procedimiento de planificación de trabajo (tan sólo se continúa con la búsqueda de trabajo), pero se debe observar cuidadosamente el caso en que se produce una notificación de descarte cuando se realiza la búsqueda de trabajo.

Si la notificación de descarte llega a un trabajador en el que se está buscando trabajo, se puede considerar la alternativa de recorrer la cadena de marcadores de paralelismo en orden inverso buscando más trabajo disponible. Sin embargo, este proceso tiene algunas desventajas: en primer lugar hay que prevenir la interferencia con otras notificaciones de descarte al recorrer la cadena y, en segundo lugar, la estrategia se complica en términos de consumo de memoria y de tiempo de cómputo. Para prevenir estas desventajas, optamos por la interrupción del procedimiento de búsqueda de trabajo *en presencia de una notificación de descarte, seguido de su activación desde* la situación inicial.

Repercusión del descarte de cómputos en la finalización de un cómputo

Cuando un trabajador finaliza la evaluación de una expresión paralela con resultado de éxito o fallo, intenta acceder al marcador de paralelismo correspondiente para actualizar la información adecuada (la respuesta computada, la presencia de alternativas[12], el resultado de fallo, ...). Si el marcador de *paralelismo está bloqueado, el trabajador debe esperar hasta que se reciba* una notificación de descarte o que se desbloquee el marcador de paralelismo. El primer caso corresponde a que el trabajador que ha evaluado una expresión hermana o antepasada ha provocado una notificación de descarte para el cómputo actual. El segundo corresponde a que el trabajador que

[12] Como veremos, esta operación consiste tan sólo en comprobar si el registro del punto de elección contiene un valor superior al del registro del marcador correspondiente (marcador de evaluación local o marcador de evaluación remota).

ha evaluado una expresión hermana ha terminado la actualización del marcador de paralelismo[13]. Cada trabajador que termina una evaluación es el responsable del análisis de la situación y de ejecutar las acciones pertinentes, en lugar de interrumpir el proceso del trabajador local.

Repercusión del descarte de cómputos en la petición de alternativas

Una petición de alternativas se puede resolver activando tan sólo un registro binario del trabajador remoto correspondiente. Esto es posible porque éste sólo puede esperar una petición de alternativas o una notificación de descarte. Si se recibe una petición de alternativas, el cómputo que hubiese podido ser asignado en la pila después de dicha expresión habría sido descartado definitivamente por una operación de descarte previa, lo que está asegurado por la condición de precedencia.

En el apéndice D se puede encontrar la descripción de cada una de los procedimientos discutidos en esta sección junto a una explicación más detallada de las consecuencias en términos de eficiencia del sistema.

4.3 Arquitectura de la PEBAM

La PEBAM consta de un conjunto de trabajadores que evalúan expresiones secuencialmente en un entorno paralelo. Un trabajador es una extensión de la máquina abstracta secuencial descrita en [101] con los mecanismos adecuados para coexistir en un entorno paralelo, que incluyen nuevos registros, estructuras de datos e instrucciones.

En el diseño de la PEBAM se debe tomar en consideración el modelo de memoria sobre el que se va a soportar, compartida o distribuida. Los multiprocesadores de memoria compartida utilizan una memoria común accesible por todos los elementos de proceso del sistema. La frecuencia de acceso en estos sistemas está limitada por el ancho de banda de la memoria. El acceso a la memoria supone un cuello de botella cuando el número de procesadores aumenta significativamente. En los multiprocesadores de memoria distribuida cada elemento de proceso dispone de una memoria local y está conectado al resto a través de una red con una determinada topología.

[13] Más adelante veremos que este no es el único caso en el que se bloquea un marcador de paralelismo.

Utilizan un sistema de paso de mensajes para la comunicación. El sistema *Sequent* [143] es representativo del modelo de memoria compartida, mientras que el sistema *Supernode*, basado en *transputers* [150], lo es del modelo de memoria distribuida.

El mecanismo de explotación del paralelismo conjuntivo aconseja la utilización de un modelo de memoria compartida puesto que el acceso a datos compartidos es más frecuentemente que en el paralelismo disyuntivo [4, 26, 164][14]. En efecto, la evaluación de las unidades paralelas de una regla debe confluir en ausencia de fallo en el punto de reunión donde se compone el resultado de la función. Esto implica no sólo el acceso compartido a los resultados de las funciones, sino también a las estructuras de datos que se construyen en las sustituciones de las variables. Por lo tanto, se produce comunicación entre los trabajadores que intervienen en el cómputo de una expresión durante la planificación paralela, la evaluación de las unidades paralelas (en la consulta de estructuras básicas o independientes) y en la composición del resultado. Utilizamos, pues, en un modelo de memoria compartida para el diseño de la PEBAM.

4.3.1 Áreas y estructuras de datos

En este apartado se describen las áreas de datos de los trabajadores (registros y áreas de memoria) y las estructuras de datos que contienen. En la figura 4.7 se muestran las áreas de datos de un trabajador que se comentan a continuación.

Registros

Distinguimos cuatro tipos de registros:

- *Registros de estrechamiento.* Se utilizan en implementaciones típicas de máquinas secuenciales de estrechamiento basadas en pilas. El registro e denota el entorno superior en la pila, b apunta al punto de elección superior (registro de *backtracking*), h a la cima del *heap*, t a la cima del *trail*, d a la cima de la pila de datos y, finalmente, ic apunta a la siguiente instrucción a ejecutar del programa.

- *Registros de control de paralelismo.* Se utilizan para acceder a las áreas de datos que gestionan el paralelismo. El registro pf se usa para

[14] No obstante, para determinados sistemas se comprueba que la memoria distribuida puede ser eficaz en este caso [4, 31]. Uno de los factores que explican este hecho es que el coste de la transmisión se puede amortizar cuando se envían grandes paquetes de memoria.

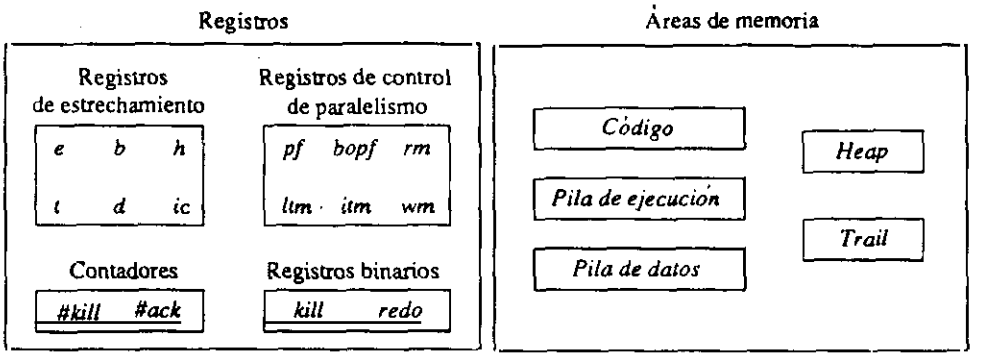


Figura 4.7: Áreas de datos de un trabajador.

denotar el marcador de paralelismo activo, mientras que *bopf* se utiliza para apuntar al primer marcador de paralelismo asignado en la pila. Los registros *ltm*, *itm* y *wm* apuntan al último marcador de evaluación local, al último marcador de evaluación remota y al último marcador de reunión, respectivamente. Finalmente, el registro *rm* refleja el estado del trabajador (i.e., en ejecución, buscando trabajo, efectuando una operación de descarte de cómputos o en estado de espera).

- *Contadores.* Un trabajador dispone de dos contadores que se utilizan para gestionar la sincronización necesaria para las operaciones de descarte de cómputos. *#kill* almacena el número n de las notificaciones de descarte entrantes que corresponden a los segmentos computacionales que delimitan los n marcadores de evaluación remota superiores. Este contador siempre lo incrementa un trabajador remoto. El contador *#nack* almacena el número esperado de reconocimientos a las notificaciones de descarte enviadas. Este contador lo decrementa el trabajador remoto que ha completado la operación de descarte de cómputos.

- *Registros binarios.* El registro *kill* se activa cada vez que se efectúa una operación local de descarte de cómputos. El registro *redo* se activa cada vez que se requiere una nueva alternativa. El trabajador correspondiente está en modo de espera y, debido a la condición de precedencia, la notificación *redo* siempre corresponde al último marcador de evaluación remota.

Áreas de memoria

A continuación describimos las áreas de memoria de los trabajadores, que se muestran en la figura 4.8. Las estructuras de datos que pueden alojarse en las áreas de memoria se muestran en la figura 4.9.

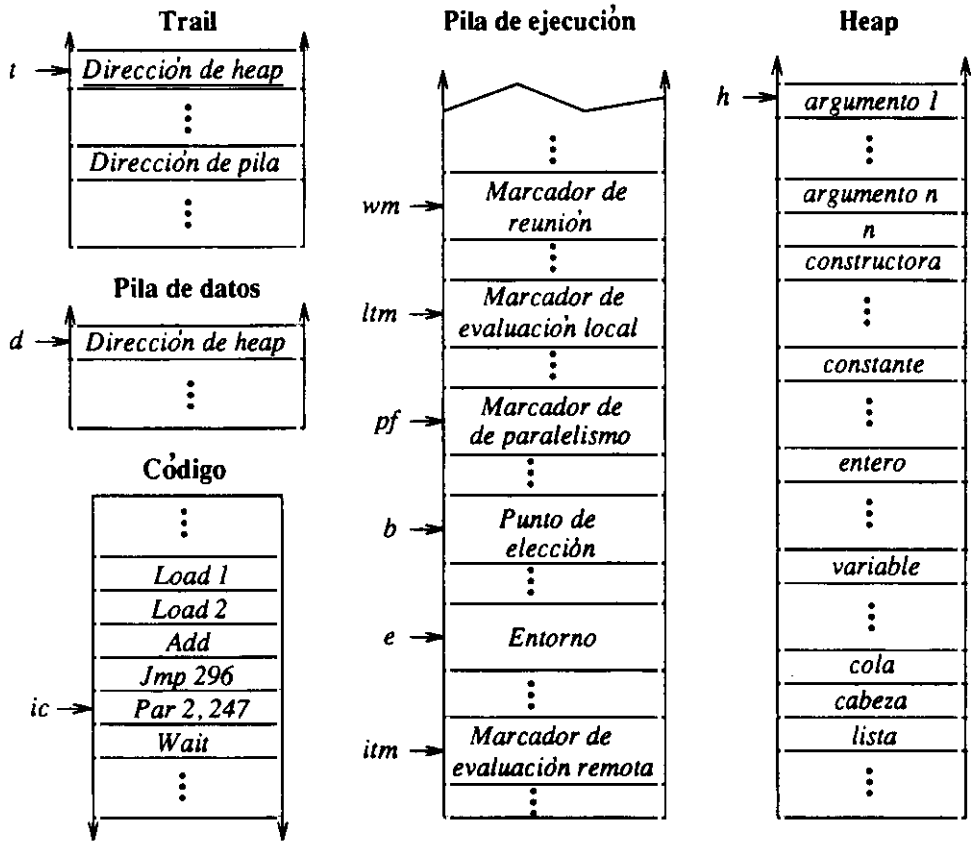


Figura 4.8: Áreas y estructuras de datos de un trabajador.

- *Código.* En este área se almacena el programa Babel compilado a instrucciones de la máquina abstracta.
- *Pila de datos.* En ella se almacenan los argumentos de llamada a función y los resultados obtenidos de la la evaluación de funciones. La pila de datos del trabajador inicial (el que evalúa la expresión objetivo) contiene el resultado de la evaluación al finalizar el cómputo.

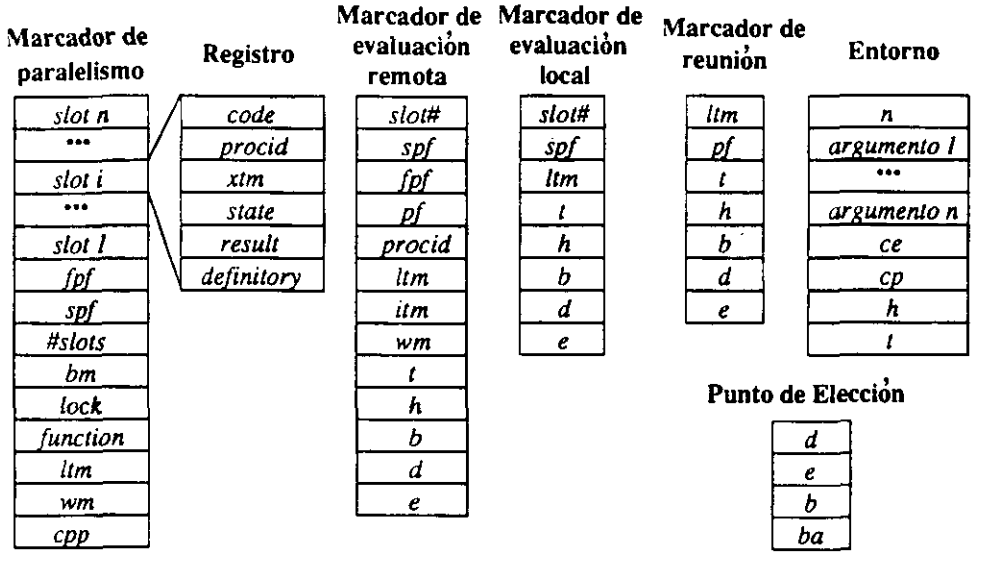


Figura 4.9: Estructuras de datos y sus contenidos.

- *Pila de ejecución.* Almacena las estructuras de datos de control para la gestión del cómputo hacia adelante y hacia atrás tanto en la ejecución secuencial como paralela.

  Las estructuras de datos usuales en una máquina secuencial (de inferencia o de estrechamiento) basada en pilas son los puntos de elección para el control de la selección de alternativas de una función y los entornos para el control de las llamadas a función.

  Un entorno tiene varios campos para almacenar el número de argumentos de llamada y los propios argumentos (en la figura: *n*, *argumento 1*, ..., *argumento n*), el entorno de continuación *ce* (i.e., el entorno padre que se recupera cada vez que se desasigna un entorno), el punto de continuación *cp* (i.e., la dirección de retorno) y los registros de *heap* y *trail*. *h* y *t* se alojan en el entorno en lugar del punto de elección para aprovechar la detección dinámica de determinismo [102].

  El punto de elección almacena el registro de la cima de la pila de datos *d*, el registro del entorno actual *e*, el registro del punto de elección *b* y la dirección de *backtracking ba*, que apunta a la siguiente alternativa disponible.

Las estructuras de datos que se describen a continuación son las necesarias para la gestión del cómputo en presencia de constructoras paralelas.

Marcador de paralelismo. Representa un punto de activación de paralelismo y contiene los siguientes campos:

- *#slots* (registros) denota el número de expresiones de la llamada paralela. Un marcador de paralelismo contiene tantos registros como número de expresiones paralelas. Cada registro contiene la información necesaria para conocer el estado computacional de la expresión, así como la información del trabajador que evalúa la expresión, que se almacenan en los siguientes campos:
  - *code*. Contiene la dirección de código que evalúa la expresión.
  - *procid*. Almacena el identificador del trabajador que evalúa la expresión.
  - *xtm*. Apunta al marcador (*marker*) correspondiente (un marcador de evaluación local si la expresión la evalúa el trabajador local, o un marcador de evaluación remota si la evalúa un trabajador remoto).
  - *state*. Almacena uno de los siguientes valores: *ready* si la expresión está preparada para ser evaluada por cualquier trabajador, *running* si la expresión está siendo evaluada, *alt* cuando la expresión ha sido evaluada y tiene alternativas pendientes, *noalt* cuando la expresión ha sido evaluada y es determinista, *failed* cuando ha fallado el cómputo, *killing* cuando se está descartando por una operación de descarte de cómputos y *killed* cuando ya ha sido descartada.
  - *result*. Apunta al resultado de la evaluación de la expresión, que se encuentra en el *heap*.
  - *definitory*. Es un campo binario que denota si la expresión ha devuelto un resultado definitorio para la función.
- *spf* y *fpf*. Almacenan las direcciones del marcador de paralelismo hijo y del marcador de paralelismo padre, respectivamente. Con el campo *spf* se puede recorrer la cadena de marcadores de paralelismo desde antepasados a descendientes, lo cual es necesario en el procedimiento de búsqueda de trabajo. El campo *fpf*

se usa durante el cómputo hacia atrás al desasignar los marcadores de paralelismo para reconocer los marcadores de paralelismo antepasados.

- *bm.* Almacena el modo de *backtracking*: *inside* cuando aún no se ha alcanzado el punto de reunión en la llamada paralela y *outside* cuando se ha alcanzado alguna vez.
- *lock.* Es un campo binario que denota el estado bloqueado o desbloqueado de un marcador de paralelismo.
- *function.* Almacena la clase de la función que es el nodo raíz de la expresión, i.e., estricta o no estricta (*strict* o *non-strict*, respectivamente).
- *ltm* y *wm.* Almacenan el marcador de evaluación local y el marcador de reunión previos a la llamada paralela. Los registros *ltm* y *wm* se recuperan de estos campos cuando se desasigna el marcador de paralelismo durante el cómputo hacia atrás.
- *cpp.* Es el punto de continuación posterior al punto de reunión. El cómputo secuencial hacia adelante se continúa en la dirección de código que se almacena en este campo.

Marcador de evaluación local. Delimita el segmento computacional de un registro en un marcador de paralelismo y contiene los siguientes campos:

- *slot#.* Almacena el identificador del registro al que pertenece el segmento computacional.
- *spf.* Es el marcador de paralelismo hijo, i.e., el siguiente marcador de paralelismo en el segmento computacional actual.
- *ltm.* Es el marcador de evaluación local previo. Durante el cómputo hacia atrás, el registro *ltm* se recupera de este campo.
- *t*, *h*, *b*, *d* y *e*. Son campos que almacenan los contenidos de los registros *t*, *h*, *b*, *d* y *e*, respectivamente, en la creación del marcador de evaluación local.

Marcador de evaluación remota. Delimita un segmento computacional que corresponde a un marcador de paralelismo remoto y contiene los siguientes campos:

- *slot#*. Almacena el identificador del registro al que pertenece el segmento computacional.
- *spf* y *fpf*. Apuntan al marcador de paralelismo hijo y padre, respectivamente.
- *ltm*, *itm*, *wm*, *t*, *h*, *b*, *d* y *e*. Son los campos que almacenan los registros *ltm*, *itm*, *wm*, *t*, *h*, *b*, *d* y *e*, respectivamente. El estado computacional del trabajador previo al segmento computacional indicado por el marcador de evaluación remota se recupera de estos campos durante el cómputo hacia atrás.

El marcador de reunión indica el punto de reunión de un cómputo paralelo. Contiene los siguientes campos:

- *pf*. Apunta al marcador de paralelismo al que pertenece el marcador de reunión.
- *ltm*, *t*, *h*, *b*, *d* y *e*. Son los campos que almacenan los registros *ltm*, *t*, *h*, *b*, *d* y *e*, respectivamente. El estado computacional del trabajador previo al punto de reunión indicado por el marcador de reunión se recupera de estos campos durante el cómputo hacia atrás.

- *Heap*. El *heap* contiene los datos que se construyen en la unificación de términos o en la reducción de las expresiones. Las entradas del *heap* pueden ser variables, constantes, listas[15] y constructoras de datos.
- *Trail*. En este área de datos se almacenan punteros a las variables que se han vinculado a fin de deshacer la vinculación si el cómputo hacia atrás lo demanda.

4.3.2 El repertorio de instrucciones

En este apartado exponemos la descripción de la semántica operacional de las instrucciones.

Instrucciones de grafos

- **Load** *i*. Almacena la *i*-ésima variable local del entorno en la pila de datos. Esta instrucción asegura que la dirección almacenada está *desreferenciada*, esto es, se almacena la representación de una variable o

[15] Como se hace usualmente, hemos distinguido estas constructoras de las constructoras de datos genéricas para especializar las operaciones en las que están implicadas.

un término como la dirección que se obtiene a través de la cadena de punteros que comienza en la celda de la variable local.

- `StoreConstr` *c n*. Genera un nuevo nodo de constructora en el *heap*. Las direcciones de sus *n* argumentos se encuentran en la pila de datos y serán reemplazados por la dirección del nodo de constructora. El nodo de constructora en el *heap* se compone del símbolo de constructora y sobre él cada uno de sus argumentos.
- `StoreConst` *c*. Genera un nuevo nodo de constante en el *heap*, que consiste en el símbolo de constructora.
- `StoreApp` *f n*. Crea un nodo de aplicación en el *heap*. Este nodo contiene la dirección de la función *f* así como los *n* argumentos de la aplicación parcial.
- `InitGuardVars` *free n*. Inicializa las variables libres del cuerpo de una regla de función, alojándolas en el *heap*.

Instrucciones de unificación

- `UnifyVar` *i*. Copia el puntero de la cima de la pila de datos a la *i*-ésima variable local en el entorno. La instrucción se aplica cada vez que la variable local se conoce como libre en tiempo de compilación, lo cual se asegura por la linealidad por la izquierda de las reglas Babel.
- `UnifyConstr` *c n*. Intenta unificar el elemento desreferenciado en la cima de la pila de datos con la constructora *n*-aria *c*. Si la unificación tiene éxito, se aloja en la pila de datos cada uno de los argumentos de la constructora que están almacenados en el *heap*. Si el puntero referencia una variable libre se genera un término de constructora, se vincula la variable a él y se inicializan sus componentes a variables libres. Si hay fallo de unificación se activa el cómputo hacia atrás.
- `CheckEq` *l*. Representa el procedimiento de unificación general que unifica recursivamente en los casos de listas o constructoras mediante el cambio de flujo de control a la dirección *l*.

Instrucciones de control de flujo hacia adelante

- `Call` *f n*. Representa una llamada secuencial a una subrutina. Se asigna un nuevo entorno en la cima de la pila de ejecución con los argumentos correspondientes que se toman de la pila de datos.

- `Proceed` f n. Es equivalente a `Call` pero con la optimización de la recursión de cola.

- `Apply` n. Aplica un nodo de aplicación a los n argumentos que se hallan sobre él en la pila de datos. Si la lista de argumentos está completa se genera un nuevo entorno y se ejecuta el código de la función correspondiente. En otro caso, la lista de argumentos se inserta en una copia del nodo de aplicación.

- `Return`. Finaliza la llamada a función almacenando en el contador de programa la dirección de retorno y recuperando el entorno previo. El entorno actual se descarta.

- `Jump` l, `JumpIfTrue` l y `JumpIfFalse` l. Realizan saltos directos y condicionales. Las instrucciones de salto condicional inspeccionan el valor lógico que se encuentra en la cima de la pila de datos.

Instrucciones de control de flujo hacia atrás

Las primeras seis instrucciones se corresponden con las instrucciones de gestión de alternativas de la WAM.

- `TryMeElse` l. Almacena el estado actual del trabajador creando un punto de elección en la cima de la pila de ejecución. El punto de elección contiene toda la información necesaria para recuperar el estado cuando el mecanismo de cómputo hacia atrás solicite una nueva alternativa en la dirección de programa l. La ejecución continúa con la instrucción siguiente.

- `RetryMeElse` l. Recupera el estado almacenado en el punto de elección, actualiza la dirección de *backtracking* del punto de elección con l y continúa con la ejecución de la siguiente instrucción.

- `TrustMe`. Recupera el estado almacenado en el punto de elección en la cima de la pila, descartándolo a continuación. Finalmente, se prueba la última alternativa continuando con la ejecución de la siguiente instrucción.

- `Try` l. Almacena el estado actual del trabajador creando un punto de elección en la cima de la pila de ejecución. La dirección de *backtracking* se actualiza a la siguiente instrucción y la ejecución se continúa en la dirección de programa l.

- `Retry` *l*. Recupera el estado almacenado en el punto de elección, se actualiza la dirección de *backtracking* a la siguiente instrucción y la ejecución continúa en la dirección de programa *l*.

- `Trust` *l*. Recupera el estado almacenado en el punto de elección y lo descarta. Se prueba la última alternativa continuando la ejecución en la dirección de programa *l*.

- `Fail`. Provoca el cambio del flujo de control a cómputo hacia atrás.

- `DynamicCut`. Comprueba si no ha habido vinculaciones de variables durante la ejecución del código de la instrucción actual. Si no se han producido vinculaciones, la regla actual es la única que es aplicable para la evaluación de la función, lo que se asegura mediante la restricción de no ambigüedad de Babel. Así, los últimos entornos y puntos de elección en la cima de la pila de ejecución pueden ser eliminados con seguridad [102].

Instrucciones de control de paralelismo

- `Par` *n l*. Crea un marcador de paralelismo con *n* slots e inicializa todos los componentes necesarios. Finalmente, se realiza un salto a la dirección *l*, que representa el punto de reunión de la llamada paralela.

- `Wait` *n*. Espera a la terminación de las *n* expresiones del marcador de paralelismo actual. Esta instrucción representa el punto de reunión de la llamada paralela. Si el marcador de paralelismo tiene aún trabajo disponible, el procesador puede robar trabajo para su ejecución local.

- `WaitTab` *f dd n*. Representa una tabla para cada expresión paralela, conteniendo la dirección *f* del código que evalúa la expresión *n* y el máximo espacio necesario de la pila de datos *dd*.

- `ConjWait` y `DisjWait`. Espera a la terminación de los argumentos de las funciones no estrictas conjunción y disyunción siguiendo la semántica no estricta. Estas instrucciones representan el punto de reunión de estas funciones.

- `SendResult` *n*. Finaliza la ejecución (con éxito) de la *n*-ésima expresión paralela. Actualiza el registro del marcador de paralelismo correspondiente con el resultado de la evaluación así como el tipo de resultado (determinista, definitorio, ...).

- `SendFail` *n*. Informa al trabajador padre de que la evaluación de la expresión ha fallado.
- `Idle`. Se cambia el estado del procesador a *idle*, iniciándose la búsqueda de trabajo.

Primitivas

- `Add`, `Subtract`, `Multiply`, `Divide` y `Modulo`. Son las primitivas aritméticas binarias, que realizan la operación correspondiente sobre los argumentos que están en la cima de la pila de datos.
- `GreaterThan`, `GreaterOrEqual`, `LessThan` y `LessOrEqual`. Son las primitivas de comparación que operan sobre los dos argumentos que están en la cima de la pila de datos.
- `And`, `Or`, `Xor` y `Not`. Son las primitivas lógicas de bits que toman sus operandos de la pila de datos.

Otras instrucciones

- `More`. Representa la pregunta al usuario por la búsqueda de nuevas soluciones.
- `Stop`. Finaliza el cómputo.
- `InitPrint`, `Print`, `PrintChar` *c*. Son las instrucciones reponsables de la presentación de las soluciones.
- `JumpIfEol` *l*, `JumpIfList` *l*, and `JumpIfData` *l*. Realizan saltos condicionales y se utilizan por razones técnicas en el procedimiento de presentación de datos.

4.4 La compilación

En esta sección se describe la traducción de programas Babel a código PEBAM. Utilizaremos una descripción funcional del esquema de traducción en lugar de la representación imperativa más habitual. Empleamos la cursiva en los identificadores de los esquemas de traducción, mientras que para las instrucciones máquina y los argumentos usamos el tipo de letra normal. El símbolo % se utilizan para denotar un comentario. Las etiquetas locales de

un fragmento de la especificación de la compilación comienzan por l y las globales por L.

4.4.1 Compilación del programa (*progtrans*)

La compilación de un programa produce el código inicial para situar al trabajador en el estado de búsqueda de trabajo (instrucción `Idle`), para parar el trabajador (instrucción `Stop`) y un conjunto de instrucciones para gestionar la presentación de datos (*print_prelude*) y algunas primitivas (según los esquemas *equality prelude* y *ho_prelude*). A continuación produce la traducción de cada una de las funciones del programa según el esquema *functrans*. El punto de entrada para la evaluación de una expresión objetivo, que se traduce según el esquema *exprtrans*, es L_{Start}. Después se genera un salto a la sección de impresión (etiquetado con L_{Print}), que se encuentra en el esquema *print_prelude*..

progtrans:(Rule*)* →Expr→PBAMCode
progtrans(<<f^j $t^j_{i,1} \ldots t^j_{i,n^j} := e^j_i \mid 1 \leq i \leq r^j >\mid 1 \leq j \leq p >$,e) ::=

```
        Idle
L_Stop: Stop
        print_prelude
        equality_prelude
        ho_prelude
        functrans(<f^1 t^1_{i,1} ... t^1_{i,n^1} := e^1_i | 1 ≤ i ≤ r^1 >)
        ⋮
        functrans(<f^p t^p_{i,1} ... t^p_{i,n^p} := e^p_i | 1 ≤ i ≤ r^p >)
L_Start: InitBam    lv_e n dd L_Print L_Stop
        exprtrans(e, 0)
        Jump        L_Print
```

4.4.2 Compilación de funciones (*functrans*)

La primera regla del esquema muestra la traducción de una única regla de definición de función, mientras que la segunda corresponde a la traduccción de más de una regla de definición de función. La traducción de alternativas es similar al caso de la WAM. Con la instrucción `Tab` se reserva espacio para los datos relativos al número de variables locales para la función y su aridad.

functrans:Rule* →PBAMCode
functrans(<f $t_{1,1} \ldots t_{1,n} := e_1$ >) ::=
L_f: `Tab` lv_f $arity_f$
ruletrans(f $t_{1,1} \ldots t_{1,n} := e_1$)
functrans(<f $t_{i,1} \ldots t_{i,n} := e_i \mid 1 \leq i \leq r$ >)::=
L_f: `Tab` lv_f $arity_f$
`TryMeElse` l_2
ruletrans(f $t_{1,1} \ldots t_{1,n} := e_1$)
l_2: `RetryMeElse` l_3
ruletrans(f $t_{2,1} \ldots t_{2,n} := e_2$)
⋮
l_r: `TrustMe`
ruletrans(f $t_{r,1} \ldots t_{r,n} := e_r$)

4.4.3 Compilación de reglas (*ruletrans*)

La traducción de una regla comienza por el esquema *unifytrans* para cada argumento de la parte izquierda de la regla, continúa con la traducción de la parte derecha de la regla con el esquema *guardtrans* y finaliza con la instrucción de retorno `Return`.

ruletrans:Rule→PBAMCode
ruletrans(f $t_1 \ldots t_n := e$) ::=
unifytrans(t_1, Load 1+lv_f)
⋮
unifytrans(t_n, Load n+lv_f)
guardtrans(e)
`Return`

4.4.4 Compilación de expresiones (*exprtrans*)

Cada regla del esquema de traducción *exprtrans* define el código generado para cada tipo de expresión: variables, constructoras, aplicaciones, funciones de order superior, funciones de selección (*if then* y *if then else*), primitivas (conjunción, disyunción, igualdad, desigualdad, operaciones aritméticas ($OP_\oplus$) y la negación), la constructora *let-in* y finalmente las funciones relacionadas con la explotación de paralelismo (*letpar-in*, la conjunción y la disyunción paralelas).

- Variable.
  La traducción de una variable consiste en la instrucción `Load` que almacena el valor correspondiente en la pila de datos.

- Constructora o aplicación de función.
  En este caso los argumentos, que se extraen de la pila de datos por las primeras instrucciones generadas por este esquema, se traducen primero.

  - Para la constructora se genera un nodo de constructora en el *heap* y se almacena su dirección en la pila de datos.
  - En el caso de una función, si todos los argumentos están disponibles se genera una instrucción `Call` o `Proceed`, dependiendo si se aplica la optimización de recursión de cola o no. Si no todos los argumentos están disponibles, se genera un nodo de aplicación mediante la instrucción `StoreApp`.

- Aplicación de orden superior.
  Se traduce la expresión y a continuación sus argumentos, generando finalmente una instrucción `Apply`, que se encarga de la aplicación de la función que se encuentra en la cima de la pila de datos a los argumentos bajo ella.

- Funciones de selección.
  Se traducen generando el código del test y a continuación un salto condicional a la instrucción de producción de fallo de cómputo. La condición de salto se realiza sobre el valor lógico que se encuentra en la pila de datos después de la evaluación del test.

- Funciones secuenciales lógicas.
  Se traduce el primer argumento de la función y el resultado de la evaluación que se encuentra en la cima del la pila de datos se consulta para comprobar si, en caso de resultado definitorio, se puede omitir la evaluación del segundo argumento.

- Igualdad.
  Se traducen sus dos argumentos y a continuación se genera la llamada al preludio de la igualdad (esta llamada se puede realizar con o sin optimización de recursión de cola, esto es, con las instrucciones `Proceed` o `Call`, respectivamente).

La desigualdad se traduce como la negación de la igualdad de sus dos argumentos.

- Funciones aritméticas binarias.
  Se traduce la función aritmética correspondiente, que toma sus argumentos de la cima de la pila de datos.

- Negación.
  Se traduce la expresión seguida de la instrucción `Not`, que reemplaza el valor en la cima de la pila de datos por el valor negado.

- Constructora *let-in*.
  Se traduce cada una de las partes derecha de las asignaciones de las variables locales.

  - Si la parte izquierda es una variable se genera la instrucción `UnifyVar`. Con ello se hace apuntar cada variable local al valor en la pila de datos.
  - Si la parte izquierda no es una variable, se aplica el esquema de traducción general para la unificación con entorno vacío ε.

  Finalmente, se aplica el esquema de traducción para la expresión fuera del entorno local.

- Constructora paralela *letpar-in*.
  Se genera un salto al punto de activación de paralelismo, que consta de la instrucción es `Par`, que denota el punto de activación de paralelismo de las n expresiones, seguida de la instrucción `Wait`, que espera hasta la terminación de todas las expresiones paralelas. Aparecen después n instrucciones `WaitTab` para cada expresión paralela, definiendo los puntos de entrada L_{F_i} al programa para la i-ésima expresión.

  Se genera el esquema *strictpartrans* para cada asignación de las variables locales, seguido de la versión secuencial *let-in* de la constructora paralela *letpar-in*, a la que se accede cuando la estrategia de reparto de trabajo decide transferir el control a la parte secuencial. Después se genera la instrucción Jump con salto a la sección de unificación. A continuación viene el esquema de traducción para cada variable local correspondiente a cada expresión paralela. Finalmente, la traducción de la expresión viene dada por el esquema *exprtrans*.

- Funciones lógicas no estrictas conjunción y disyunción.
  Sus esquemas son similares correspondiente a *let-par*, salvo en que la parte de la unificación y la última traducción *exprtrans* no aparecen puesto que dichas operaciones las realizan las instrucciones `DisjWait` y `ConjWait`.

exprtrans:Expr→ $\mathcal{N}$ →PBAMCode
exprtrans(X_i, tl) ::=
    `Load` i
exprtrans(c $e_1 \ldots e_n$, tl) ::=
    exprtrans(e_1,0)
    ⋮
    exprtrans(e_n,0)
    `StoreConstr` c n
exprtrans(f $e_1 \ldots e_n$, tl) ::=
    exprtrans(e_1,0)
    ⋮
    exprtrans(e_n,0)
    $\begin{cases} \texttt{Proceed} & L_f \text{ n } si \text{ aridad}_f = n, \text{tl} = 1 \\ \texttt{Call} & L_f \text{ n } si \text{ aridad}_f = n, \text{tl} = 0 \\ \texttt{StoreApp} & L_f \text{ n } si \text{ aridad}_f > n \end{cases}$
exprtrans(e $e_1 \ldots e_n$, tl) ::=
    exprtrans(e, 0)
    exprtrans(e_1,0)
    ⋮
    exprtrans(e_n,0)
    `Apply` n tl
exprtrans(if e_1 then e_2, tl) ::=
    exprtrans(e_1, 0)
    `JumpIfFalse` L_{Fail}
    exprtrans(e_2, tl)
exprtrans(if e_1 then e_2 else e_3, tl) ::=
    exprtrans(e_1, 0)
    `JumpIfFalse` l_{false}
    exprtrans(e_2, tl)
    `Jump` l_{cont}
l_{false}: *exprtrans*(e_3, tl)
l_{cont}: ...

exprtrans(e_1,e_2, tl) ::=
  exprtrans(e_1, 0)
  `JumpIfFalse` l_{false}
  exprtrans(e_2, tl)
  `Jump` l_{cont}
l_{false} : `StoreConst` false
l_{cont}: ...
exprtrans(e_1;e_2,tl) ::=
  exprtrans(e_1, 0)
  `JumpIfTrue` l_{true}
  exprtrans(e_2, tl)
  `Jump` l_{cont}
l_{true}: `StoreConst` true
l_{cont}: ...
exprtrans(e_1=e_2, tl) ::=
  exprtrans(e_1, 0)
  exprtrans(e_2, 0)
  $\begin{cases} \texttt{Proceed} & L_{Eq}\ 2 \ \ si\ \ tl = 1 \\ \texttt{Call} & L_{Eq}\ 2 \ \ si\ \ tl = 0 \end{cases}$
exprtrans(e_1 ~=e_2,tl) ::=
  exprtrans(~(e_1=e_2),tl)
exprtrans($e_1 \oplus e_2$, tl) ::= ($\oplus \in \{+, -, *, /, \%, >, <, \leq, \geq\}$)
  exprtrans(e_1, 0)
  exprtrans(e_2, 0)
  $OP_\oplus$
exprtrans(~e, tl) ::=
  exprtrans(e, 0)
  `Not`
exprtrans(let t_1=e_1,...,t_n=e_n in e, tl) ::=
  exprtrans(e_1,0)
  $\begin{cases} \texttt{UnifyVar i} & si\ t_1 \in \text{Var} \\ \textit{unifytrans}(t_1, \varepsilon) & si\ t_1 \notin \text{Var} \end{cases}$
  ⋮
  exprtrans(e_n,0)
  $\begin{cases} \texttt{UnifyVar i} & si\ t_n \in \text{Var} \\ \textit{unifytrans}(t_n, \varepsilon) & si\ t_n \notin \text{Var} \end{cases}$
  exprtrans(e,1)
exprtrans(letpar t_1=e_1,...,t_n=e_n in e, tl) ::=

```
                Jump l_fork
l_1:            strictpartrans(e_1, t_1, 1, i_1)
                :% i_k=min{0,j | X_j ∈ var(t_k)}(1 ≤ k ≤ n)
l_n:            strictpartrans(e_n, t_n, n, i_n)
l_sequential:   exprtrans(let t_1=e_1,...,t_n=e_n in e, tl)
                Jump l_unify
l_fork:         Par             n l_sequential
                Wait            n
                WaitTab         L_F1 1
                ⋮
                WaitTab         L_Fn n
l_unify :       unifytrans(t_1, Load i_1) % i_1 > 0
                ⋮
                unifytrans(t_n, Load i_n) % i_n > 0
                exprtrans(e,1)
exprtrans(e_1 && ... && e_n, tl) ::=
                Jump l_fork
l_1:            partrans(e_1, 1)
                ⋮
l_n:            partrans(e_n, n)
l_sequential :exprtrans(e_1, ..., e_n, tl)
                Jump l_cont
l_fork:         Par             n l_sequential
                ConjWait        n
                WaitTab         L_F1 1
                ⋮
                WaitTab         L_Fn n
l_cont:         ...
exprtrans(e_1 || ... || e_n, tl) ::=
                Jump l_fork
l_1:            partrans(e_1, 1)
                ⋮
l_n:            partrans(e_n, n)
l_sequential:   exprtrans(e_1; ...; e_n, tl)
                Jump l_cont
l_fork:         Par             n l_sequential
```

```
        DisjWait     n
        WaitTab      L_F1 1
        ⋮
        WaitTab      L_Fn n
l_cont: ...
```

4.4.5 Compilación de la unificación (*unifytrans*)

Hay dos casos:

- Unificación de una variable.
  En este caso la única instrucción necesaria es la que se pasa a través del parámetro loadinstr.
- Constructora.
  En este caso aparece en primer lugar la instrucción en el parámetro loadinstr seguida de la instrucción `UnifyConstr` y del esquema *unifytrans*, que se aplica recursivamente a cada argumento de la constructora.

unifytrans:Term→({Load}×$\mathcal{N}$ ∪ ε) →PBAMCode
unifytrans(X_i, loadinstr) ::=
        loadinstr
unifytrans(c $t_1 \ldots t_n$, loadinstr) ::=
        loadinstr
        `UnifyConstr` c n nlv
        unifytrans(t_1, Load nlv)
        ⋮
        unifytrans(t_n, Load nlv+n-1)

4.4.6 Compilación de la igualdad (*equality_prelude*)

equality_prelude ::=

```
L_Eq:   Load        1
        Load        2
        CheckEq     l_eqconstr
        Return
l_eqconstr: UnifyVar    1
        Load        1
```

```
            CheckEq       l_eqconstr
            JumpIfFalse   l_eqfail
            JumpIfData    l_eqconstr 2
            StoreConst    true
            Return
l_eqfail:   StoreConst    false
            Return
```

4.4.7 Compilación de las *PEUs* (*partrans* y *strictpartrans*)

Los esquemas *strictpartrans* y *partrans* corresponden a las traducciones de las *PEUs* de las funciones estrictas y no estrictas, respectivamente.

partrans:Expr→ $\mathcal{N}$ →PBAMCode
partrans(e, n) ::=

```
            TryMeElse     l_fail
            exprtrans(e, 0)
            SendResult    n
l_fail:     SendFail      n
```

strictpartrans:Expr→Term→ $\mathcal{N}^2$ →PBAMCode
strictpartrans(e, t, n, i) ::=

```
            TryMeElse     l_fail
            exprtrans(e, 0)
             unifytrans(t, ε)   } % i=0
             StoreConst true    }
            SendResult    n
l_fail:     SendFail      n
```

4.4.8 Compilación del guarda (*guardtrans*)

La traducción consiste en primer lugar en la instrucción `InitGuardVars` o la instrucción `DynamicCut` según sea respectivamente el caso de optimización de corte dinámico o sin ella, seguido por la traducción de la expresión dada por *exprtrans*. Si la expresíon es un bicondicional se inicializan las variables del guarda con `InitGuardVArs`, seguido de la traducción del test con el esquema *exprtrans* y un salto condicional a la dirección de fallo. A continuación, la instrucción de corte dinámico `DynamicCut` comprueba si se

puede descartar el punto de elección y por último aparece la traducción de la expresión *then* del condicional.

guardtrans:Expr→PBAMCode
guardtrans(e) ::=
    `InitGuardVarsgv` n % Sin optimización de corte dinámico
    exprtrans(e, 1)
guardtrans(e) ::=
    `DynamicCut` % Con optimización de corte dinámico
    exprtrans(e,1)
guardtrans(if e_1 then e_2) ::=
    `InitGuardVarsgv` n
    exprtrans(e_1, 0)
    `JumpIfFalse` L_{Fail}
    `DynamicCut`
    exprtrans(e_2, 1)

4.4.9 Compilación de primitivas aritméticas y de comparación (*ho_prelude*)

La traducción consiste en las instrucciones `Load` para el almacenamiento de los argumentos en la pila de datos seguida por la instrucción de aplicación de función.

ho_prelude ::=

$l_{\oplus}$:	`Load`	1 % $\oplus \in \{+, -, *, /, \%, >, <, \leq, \geq\}$
	`Load`	2
	$OP_{\oplus}$	
	`Return`	

4.4.10 Compilación de la impresión (*print_prelude*)

print_prelude ::=

l_{ploop}:	`PrintChar`	44 % ','
$l_{pconstr}$:	`Print`	$l_{pconstr}$ l_{plist} l_{pccont}
	`JumpIfFalse`	L_{Stop}
l_{pccont}:	`JumpIfData`	l_{ploop} 1
	`PrintChar`	41 % ')'

```
            StoreConst    true
            Return
l_plloop:   PrintChar     44 % ','
l_plist:    Print         l_pconstr l_plist l_plcont1
            JumpIfFalse   L_Stop
l_plcont1:  Load          1
            JumpIfEoL     l_plcont2
            JumpIfList    l_plloop
            PrintChar     124 % '|'
            Print         l_pconstr l_plist l_plcont2
            JumpIfFalse   L_Stop
l_plcont2:  PrintChar     93 % ']'
            StoreConst    true
            Return
L_Print:    InitPrint
            Print         l_pconstr l_plist l_ploopvar
            JumpIfFalse   L_Stop
l_ploopvar: PrintVar      l_pexit
            Print         l_pconstr l_plist l_ploopvar
            JumpIfTrue    l_ploopvar
l_pexit:    More
            JumpIfFalse   L_Stop
L_Fail:     Fail
```

4.4.11 Ejemplo de compilación

Como ejemplo mostramos la compilación mostramos de la versión paralela del programa Fibonacci, con la expresión objetivo fib(10)[16].

```
fib X := if
           (X>1)
         then
           letpar Y1 = fib (X-1),
                  Y2 = fib (X-2)
           in Y1+Y2
         else 1.
```

[16] En el listado se utilizan itálicas para denotar las etiquetas.

La compilación de este programa produce el código que se reproduce a continuación. La dirección de comienzo de cómputo corresponde a la etiqueta L_{Start}.

```
                Idle
Stop            Stop
PLoop           PrintChar       44
PConstr         Print           PConstr PList PCCont
                JumpIfFalse     Stop
PCCont          JumpIfData      PLoop
                PrintChar       41
                StoreConst      true
                Return
PLLoop          PrintChar       44
PList           Print           PConstr PList PLCont1
                JumpIfFalse     Stop
PLCont1         JumpIfEoL       PLCont2
                JumpIfList      PLLoop
                PrintChar       124
                Print           PConstr PList PLCont2
                JumpIfFalse     Stop
PLCont2         PrintChar       93
                StoreConst      true
                Return
Print           InitPrint
                Print           PConstr PList PLoopVar
                JumpIfFalse     Stop
PLoopVar        PrintVar        PExit
                Print           PConstr PList PLoopVar
                JumpIfTrue      PLoopVar
PExit           More
                JumpIfFalse     Stop
Fail            Fail
Eq              Load            1
                Load            2
                CheckEq         EqStruct
                Return
EqStruct        InitGuardVars   1 1
                UnifyVar        1
```

```
                Load          1
                CheckEq       EqStruct
                JumpIfFalse   EqFail
                JumpIfData    EqStruct
                StoreConst    true
                Return
EqFail          StoreConst    false
                Return
Add             Load          1
                Load          2
                Add
                Return
Subtract        Load          1
                Load          2
                Subtract
                Return
Multiply        Load          1
                Load          2
                Multiply
                Return
Divide          Load          1
                Load          2
                Divide
                Return
Module          Load          1
                Load          2
                Module
                Return
GreaterThan     Load          1
                Load          2
                GreaterThan
                Return
GreaterOrEqual  Load          1
                Load          2
                GreaterOrEqual
                Return
LessThan        Load          1
                Load          2
                LessThan
```

```
                Return
LessOrEqual     Load        1
                Load        2
                LessOrEqual
                Return
And             Load        1
                Load        2
                And
                Return
Or              Load        1
                Load        2
                Or
                Return
Xor             Load        1
                Load        2
                Xor
                Return
Not             Load        1
                Not
                Return
fib             InitGuardVars 1 2
                Load        3
                StoreNum    1
                GreaterThan
                JumpIfFalse Label.1
                Jump        Label.2
Label.6         TryMeElse   Label.5
                Load        3           % X
                StoreNum    1           % 1
                Subtract                % X-1
                Call        fib 1 2 2 1 % fib(X-1)
                SendResult  1           % Y1=fib(X-1)
Label.5         SendFail    1
Label.8         TryMeElse   Label.7
                Load        3           % X
                StoreNum    2           % 2
                Subtract                % X-2
                Call        fib 1 2 2 1 % fib(X-2)
                SendResult  2           % Y2=fib(X-2)
```

```
Label.7     SendFail    2
Label.3     Load        3
            StoreNum    1
            Subtract
            Call        fib 1 2 2 1
            UnifyVar    1
            Load        3
            StoreNum    2
            Subtract
            Call        fib 1 2 2 1
            UnifyVar    2
            Load        1
            Load        2
            Add
            Jump        Label.4
Label.2     Par         2 Label.3
            Wait        2 3
            WaitTab     Label.6 2 1
            WaitTab     Label.8 2 2
Label.4     Load        1           % Y1
            Load        2           % Y2
            Add                     % Y1+Y2
            Return      3
Label.1     StoreNum    1
Label.0     Return      3
L_Start     InitBam     0 1 5 Print Stop
            StoreNum    10          % 10
            Call        fib 1 2 2 1 % fib(10)
            Jump        Print
```

Sumario

En este capítulo se ha desarrollado una máquina abstracta paralela basada en pilas sobre un modelo de memoria compartida que retiene las optimizaciones de las implementaciones secuenciales lógicas, funcionales y lógico-funcionales, fundamentalmente en la desasignación de memoria durante el cómputo hacia atrás. En particular, esta máquina incorpora las optimiza-

ciones de recursión de cola y corte dinámico, que mejoran el rendimiento de la máquina tanto en presencia de constructoras secuenciales como paralelas. Se ha presentado el repertorio de instrucciones de la máquina, que es semejante al de la WAM en cuanto a la gestión del no determinismo y similar al de las máquinas de reducción de grafos en cuanto a las aplicaciones parciales y la gestión de la evaluación de funciones de primer orden y orden superior[17].

[17]En el apéndice D se puede consultar su especificación, en la que se describe la promoción de la ejecución paralela de las diferentes operaciones (desasignación de cómputos y envío de notificaciones de descarte), no sólo en el cómputo hacia adelante sino también en el cómputo hacia atrás.

Capítulo 5

Implementación y resultados

En este capítulo se presenta una implementación de la máquina abstracta paralela sobre un multiprocesador simulado de memoria compartida. Se estudian los componentes del sistema y se plantean diferentes alternativas de diseño, que afectan al protocolo de arbitraje de bus y a la política de coherencia de la memoria caché. Se utiliza el lenguaje de descripción *hardware* VHDL para validar el comportamiento funcional de la máquina y para obtener medidas de rendimiento del sistema.

5.1 Requerimientos de la máquina PEBAM

Durante la ejecución de un programa se producen las referencias[1] entre las áreas de datos de los trabajadores que se muestran en la figura 5.1. A continuación se describe el tipo de acceso que permite cada área.

- *Pila de ejecución.*
  - Acceso local. Se produce durante la unificación, la búsqueda de trabajo local y el *backtracking* local.
  - Acceso remoto. Se produce durante la unificación en la evaluación de una expresión remota, en la búsqueda de trabajo remoto, en la adquisición de los argumentos para la evaluación de una expresión remota, en la finalización de una expresión remota y en el *backtracking* paralelo.
- *Heap.*

[1] Estas referencias tienen el formato que se muestra en la figura 5.3.

  - Acceso local. Se produce durante la unificación.
  - Acceso remoto. Se produce durante la unificación local (cuando han intervenido trabajadores remotos en el cómputo previo) o remota.

- *Pila de datos.*
  - Acceso local. Se produce durante la carga de argumentos de llamada a una función, consulta de argumentos y funciones y depósito del resultado.

- *Trail.*
  - Acceso local. Se produce durante el *backtracking* local y la petición remota del descarte de un cómputo.

- *Código.*
  - Acceso local.

De la figura 5.1 se deducen las áreas de datos que deben alojarse en una zona común a los trabajadores y las que son locales a los trabajadores. Sólo las pilas de ejecución y los *heaps* son comunes puesto que aceptan referencias de trabajadores remotos. En la figura 5.2 se muestra la ubicación de las áreas de datos de los trabajadores en la zona común y las zonas locales. Esta división muestra los requerimientos de la máquina paralela abstracta que debe satisfacer el sistema de ejecución. En la siguiente sección se estudia la correspondencia de los trabajadores y sus áreas de datos con los elementos del sistema multiprocesador.

5.2 Sistema de ejecución paralela

Para la implementación de la máquina PEBAM utilizamos un sistema multiprocesador de memoria compartida en el que varios elementos de proceso están conectados a una memoria común a través de un único bus. En la figura 5.4 se muestra el esquema de bloques del sistema con n elementos de proceso y m módulos de memoria.

Cada elemento de proceso dispone además de memoria local. En ese sistema se realiza la correspondencia de trabajadores con elementos de proceso, de áreas de datos locales con memorias locales y de áreas de datos comunes con los módulos de memoria compartida. A continuación se estudiará cada uno de los componentes del sistema multiprocesador.

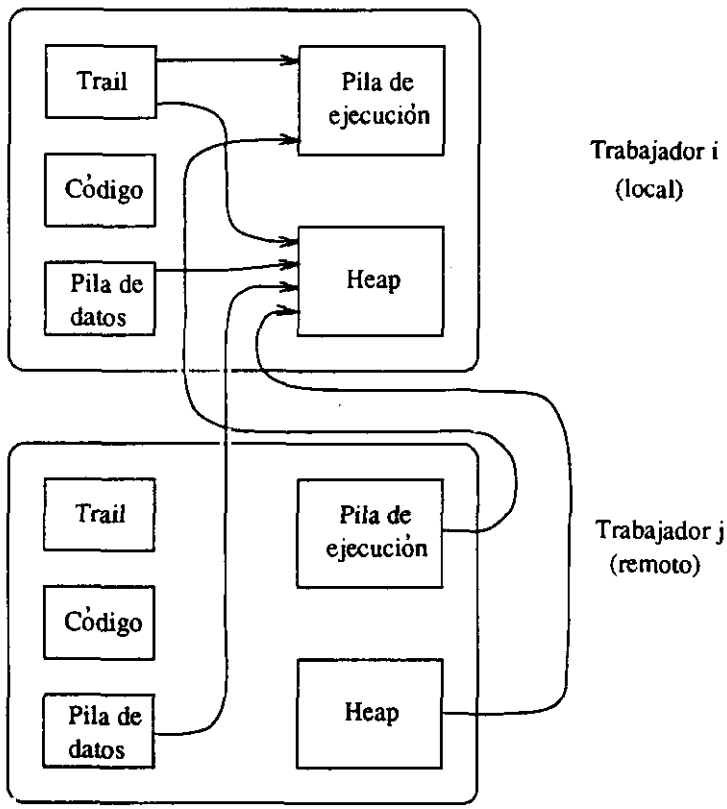


Figura 5.1: Referencias entre las áreas de datos de los trabajadores.

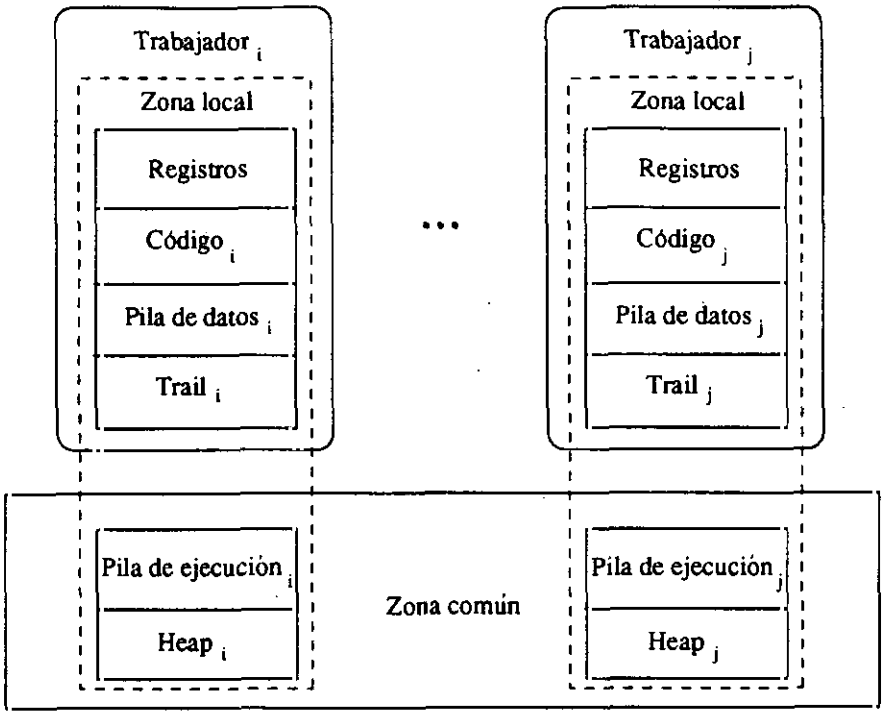


Figura 5.2: Áreas de datos.

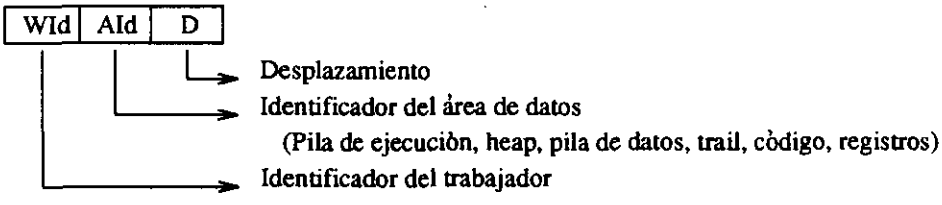


Figura 5.3: Formato de las direcciones.

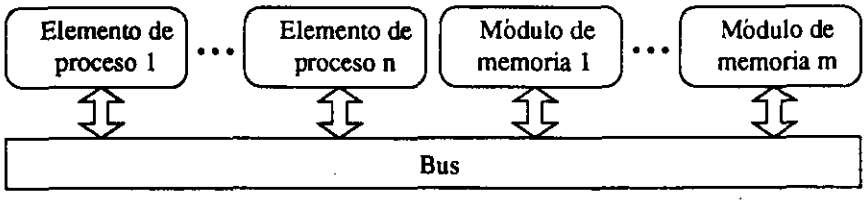


Figura 5.4: Sistema multiprocesador de memoria compartida.

5.2.1 Bus del sistema

El bus del sistema garantiza la correcta comunicación entre los diferentes dispositivos (módulos de memoria y elementos de proceso). Para ello dispone de un controlador, un árbitro (o simplemente controlador del bus) y las líneas de comunicación: control, datos y direcciones. Estudiamos dos configuraciones diferentes:

- Bus con arbitraje por sondeo. El controlador sondea secuencialmente los dispositivos conectados al bus para detectar las demandas de acceso. Son necesarias las líneas que se muestran en la figura 5.5. En esta figura se muestran el controlador y árbitro del bus, las líneas de control y el bus, que se compone del bus de datos, direcciones y resto de líneas de control. Hay una línea de concesión del bus por cada dispositivo (BGT_i)[2] y una línea de ocupación del bus (SACK). Para esta configuración consideraremos el protocolo de arbitraje LRU.

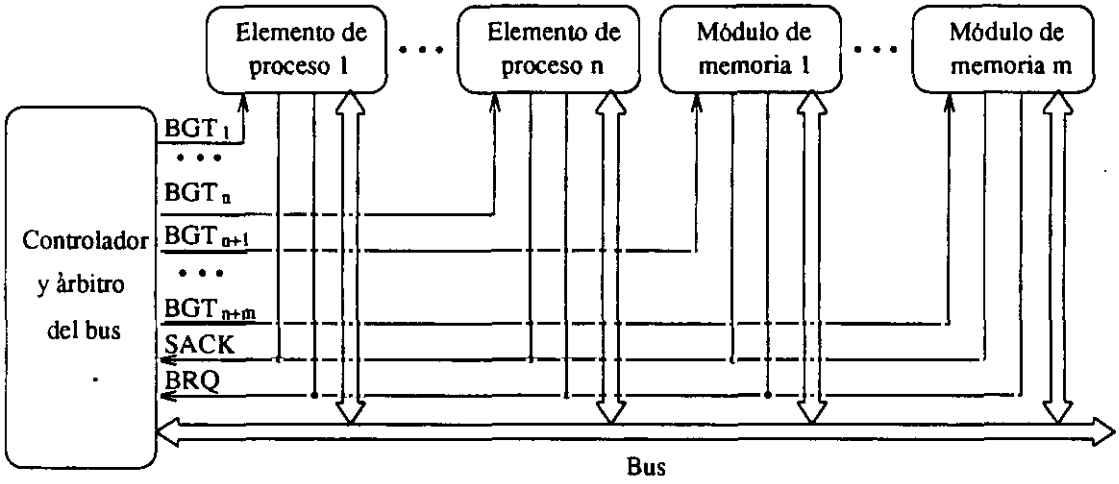


Figura 5.5: Bus con arbitraje por sondeo.

- Bus con arbitraje por petición independiente. En este caso es necesario un par de líneas de concesión y petición para cada elemento de proceso, como se muestra en la figura 5.6. Esta implementación necesita más líneas de control que la anterior, pero admite una detección paralela

[2]Este conjunto se puede simplificar para que $log_2(n+m)$ líneas codifiquen el identificativo del dispositivo al que se concede el uso del bus. En tal caso, estas líneas se conectan a cada elemento de proceso de la red.

del dispositivo demandante. Para esta configuración emplearemos el protocolo de arbitraje FCFS (*First-Come, First-Served*: primero-que-llega, primero-que-se-sirve).

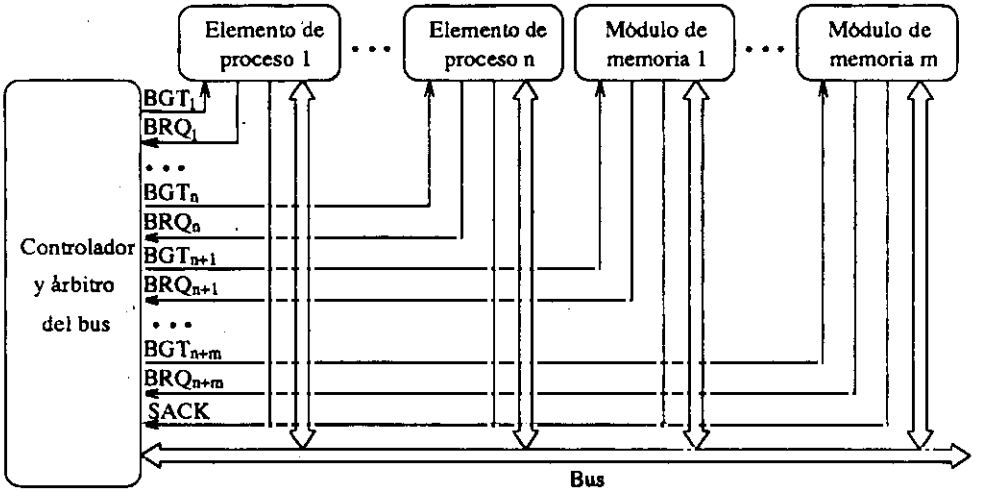


Figura 5.6: Bus con arbitraje por petición independiente.

5.2.2 Elementos de proceso

En la figura 5.7 se muestran los componentes de cada elemento de proceso, i.e., el procesador, la memoria local y la memoria caché. Los trabajadores que se ejecutan en el procesador acceden a las áreas de datos locales emplazadas en la memoria local y a las áreas comunes a través de la caché. Ésta contiene copias consistentes de trozos de las áreas comunes situadas en memoria compartida.

Memoria caché

La memoria caché de cada elemento de proceso consta del controlador de caché, el directorio de bloques, la memoria y el vigilante del bus, como se muestra en la figura 5.8. Cuando un procesador efectúa una petición de lectura o escritura (*p_read* o *p_write*) a su controlador de caché, se examina el directorio para determinar si la posición de memoria correspondiente se encuentra en caché. El controlador envía un mandato al bus para conseguir el bloque o para informar si se ha producido una actualización local, de

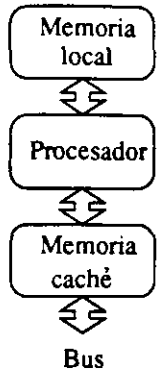


Figura 5.7: Elemento de proceso.

manera que el vigilante de cada elemento de proceso pueda inspeccionar el bus para atenderlos.

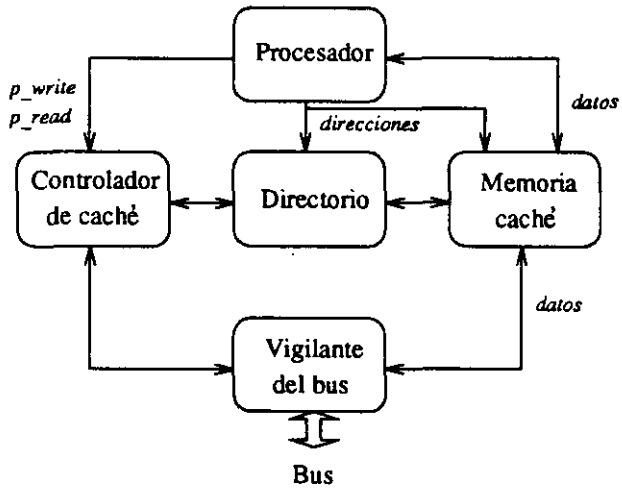


Figura 5.8: Memoria caché.

5.2.3 Memoria compartida

Cada módulo de memoria compartida consta de la memoria en sí y un controlador, como se muestra en la figura 5.9. El controlador de memoria gestiona los mandatos que circulan por el bus referidos a su módulo.

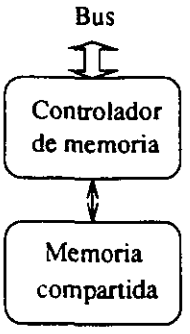


Figura 5.9: Módulo de memoria compartida.

5.2.4 Protocolos de coherencia caché

La eficiencia de un protocolo de mantenimiento de la coherencia de la memoria caché de un multiprocesador viene determinada por el comportamiento del programa paralelo. Consideraremos dos implementaciones representativas de caché: la invalidación y la actualización en escritura para estudiar en la próxima sección su adecuación a diferentes programas.

- Protocolo de Goodman [5] de invalidación en escritura.
  Cada vez que se actualiza un bloque se invalidan sus copias en las otras cachés, por lo que este protocolo es eficaz cuando la localidad temporal de los datos es baja. En este protocolo, una copia de un bloque en memoria caché puede estar en uno de los siguientes estados:

  - *Invalid*: Copia inconsistente con la memoria.
  - *Valid*: Copia consistente con la memoria.
  - *Reserved*: Copia sobre la que se ha escrito una vez y es consistente sólo con la memoria.
  - *Dirty*: Copia modificada más de una vez y única válida en el sistema.

  La coherencia del sistema se asegura por las transiciones de estado provocadas por los mandatos generados por un elemento de proceso (*p_read* y *p_write*, lectura y escritura de una posición de memoria, respectivamente), que se muestran en la figura 5.10, y en las que se emiten los siguientes mandatos de consistencia:

  - Mandatos generados por la caché:

  * *read(b)*: lectura de un bloque b realizada sobre el bus.
  * *write(b)*: escritura de un bloque b realizada sobre el bus.
- Mandatos procedentes del bus común:
  * *inv_write(b)*: invalida las copias del bloque b.
  * *inv_read(b)*: *read(b)* e invalida el resto de copias.

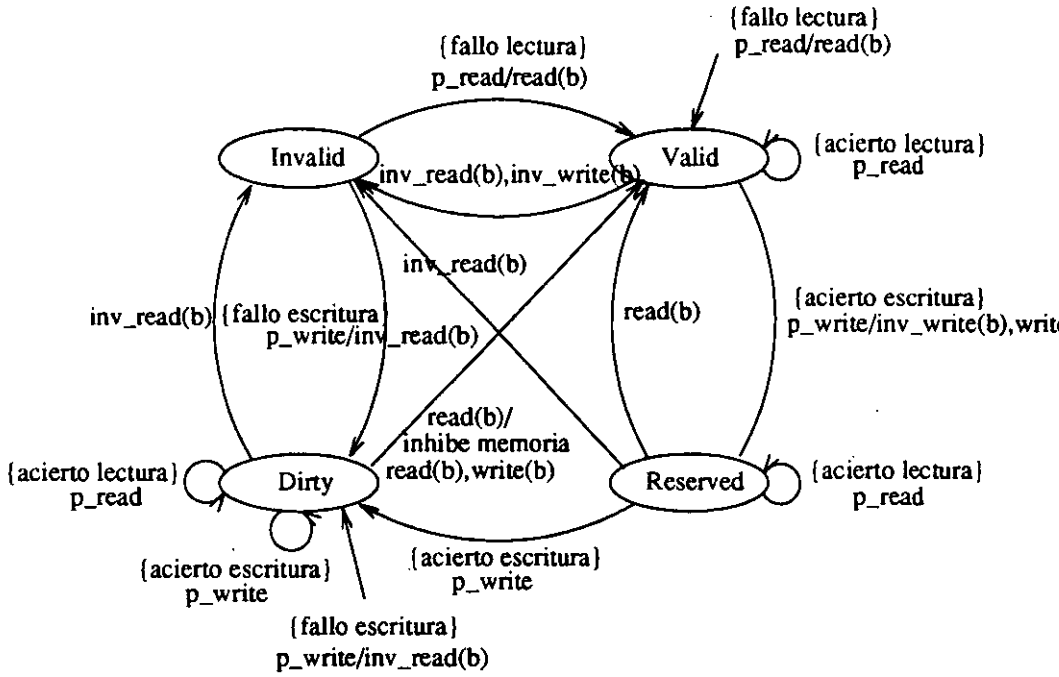


Figura 5.10: Diagrama de transición de estados del protocolo de Goodman.

La memoria compartida se actualiza cuando se reemplaza un bloque en estado *dirty* (política de actualización de memoria de retroescritura o *copy-back*).

- Protocolo Firefly [5] de actualización en escritura.
  Cada vez que se actualiza un bloque se actualizan sus copias en las otras cachés, por lo que este protocolo es eficaz cuando la localidad temporal de los datos es alta. Este protocolo sólo distingue los tres estados siguientes para una copia de un bloque:
  - *Exclusive-valid.* Copia única consistente con memoria.

- *Shared.* Copia consistente posiblemente no única.
- *Dirty.* Copia única; la copia en memoria no es consistente.

Las transiciones de estado posibles según este protocolo se describen en el diagrama de la figura 5.11.

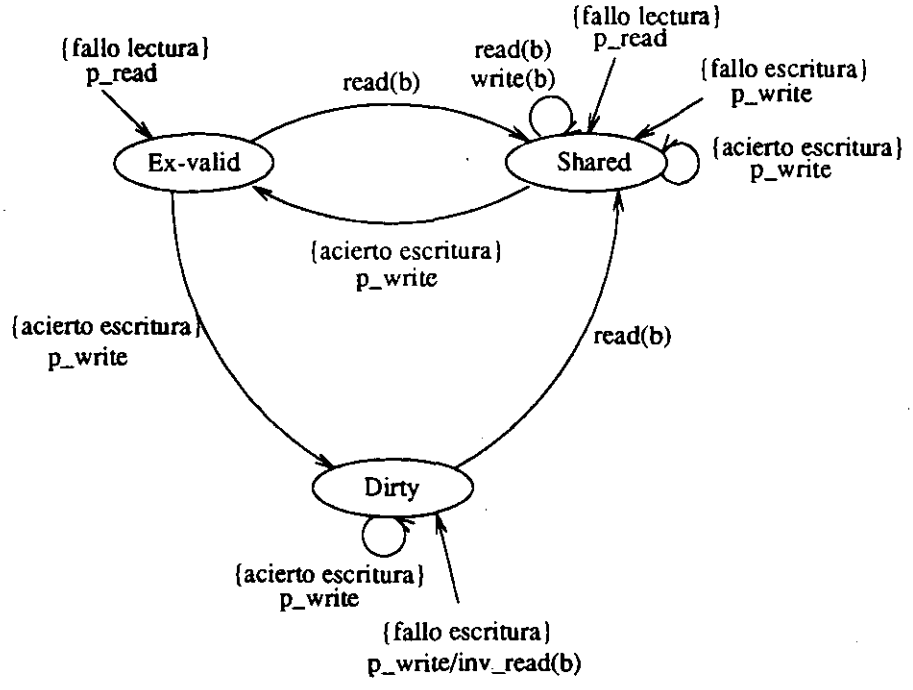


Figura 5.11: Diagrama de transición de estados del protocolo Firefly.

Este protocolo usa un política de actualización *copy-back* para los bloques privados y *write-through* para los bloques compartidos. Se utiliza una línea compartida por los elementos de proceso para los bloques compartidos.

5.3 Simulación del sistema multiprocesador

Realizamos la simulación del sistema multiprocesador utilizando el lenguaje de descripción *hardware* VHDL, que dispone de un modelo temporal que permite incorporar los retardos en la simulación para obtener medidas de rendimiento del sistema.

5.3.1 El lenguaje de descripción *hardware* VHDL

VHDL[3] es un lenguaje de descripción *hardware*, desarrollado con el apoyo del departamento de defensa de EEUU para uniformizar la descripción de las especificaciones de diseño de circuitos [85, 13] que se ha convertido en el estándar IEEE 1076-1987 revisado en 1992.

VHDL permite simultanear la especificación de comportamiento de un sistema y su descripción estructural, permitiendo su simulación para obtener medidas de rendimiento antes de su implementación, evitando el retardo y coste del prototipado *hardware*.

Un programa VHDL es un conjunto de procesos concurrentes que se comunican por medio de señales siguiendo un modelo guiado por eventos. El comportamiento de cada proceso se describe por medio de un algoritmo secuencial que utiliza instrucciones clásicas de los lenguajes imperativos, principalmente de Ada. Para controlar la comunicación entre procesos se dispone de las instrucciones `WAIT` y la asignación de señal (<=), cuya sintaxis general es:

señal <= *expresión* `AFTER` *expresión temporal*;

La *expresión* se evalúa y el resultado se planifica para que sea el valor actual de *señal* después del retardo indicado por *expresión temporal*. La instrucción `WAIT` permite la suspensión de un proceso en ejecución y la expresión de las condiciones para su reactivación.

La sintaxis general de una instrucción `WAIT` es la siguiente:

`WAIT ON` *lista de sensibilización* `UNTIL` *condición* `FOR` *expresión temporal*;

La condición lógica *condición* se evalúa cada vez que sucede un evento en una señal que se encuentre en la *lista de sensibilización*. Si el resultado de la evaluación de la condición es falso, el proceso permanece suspendido, en otro caso se reactiva después del intervalo de tiempo denotado por *expresión temporal*. El tiempo del sistema sólo avanza cuando se ejecuta una instrucción `WAIT`.

La sintaxis general de un proceso es la siguiente:

[3]VHDL es acrónimo de *VHSIC Hardware Description Language*, donde VHSIC es una abreviatura de *Very High Speed Integrated Circuits*.

```
nombre : PROCESS
             región declarativa
         BEGIN
             instrucciones secuenciales
         END PROCESS;
```

Hay una región declarativa delimitada por las palabras reservadas `PROCESS` y `BEGIN` donde se pueden declarar tipos de datos, subprogramas y variables.

La simulación de un programa VHDL consiste en dos fases. En la primera fase, la inicialización, se inicializa el valor de todas las señales y se pone a cero el tiempo de simulación. En la segunda fase, el ciclo de simulación, se ejecuta el algoritmo secuencial de cada proceso hasta que se suspenden mediante la instrucción `WAIT`. En cada ciclo de simulación se actualizan los valores de las señales y se evalúan las condiciones de suspensión. La expresión temporal de la instrucción `WAIT` marca el avance de tiempo cuando se reactiva un proceso. La simulación se completa cuando no se producen más cambios en las señales.

5.3.2 Simulación VHDL del sistema multiprocesador

En esta sección describimos en VHDL las diferentes alternativas de diseño, especificando las entidades, las arquitecturas de comportamiento y la interconexión de los componentes del sistema multiprocesador con protocolo de petición independiente. Esta descripción también es válida para la especificación del protocolo de petición por sondeo obviando las líneas innecesarias.

Bus del sistema

Las líneas de control, datos y direcciones están declaradas con las señales, que se describen en el apartado de la interconexión de componentes, y se conectan a los siguientes puertos (véase el fragmento VHDL que se muestra en la figura 5.12[4]).

- **Bus.**
  Se conecta con el bus de datos, de direcciones y las líneas de control.

[4]Los identificadores que comienzan por `t_` representan los tipos de las señales correspondientes (normalmente, tipos de datos enumerados o compuestos de posibles valores asociados a señales); los que comienzan por `v_`, variables; los que comienzan por `c_`, constantes y los que comienzan por `l_`, etiquetas. Las palabras reservadas `IN`, `OUT` e `INOUT` denotan respectivamente a los puertos que son de entrada, salida y entrada/salida. Los puntos suspensivos denotan código que se omite por claridad.

Su tipo denota un registro cuyas componentes detallan estas líneas y buses.

- `BRQs`.
  Se conecta con el conjunto de líneas de petición de uso del bus. Su tipo denota un vector cuyas componentes representan los valores lógicos de cada línea.

- `BGTs`.
  Se conecta con el conjunto de líneas de concesión. Su tipo denota un vector cuyas componentes representan los valores lógicos de cada línea.

- `SACK`.
  Se conecta con la línea de ocupación del bus. Su tipo representa los valores lógicos de la línea.

La unidad de control detecta una petición de uso del bus cuando se activa al menos una de las líneas `BRQs`. Cuando la línea de ocupación del bus `SACK` está inactiva, se selecciona un dispositivo con el procedimiento `select_asker`, que implementa el protocolo de arbitraje. La petición se sirve mediante el procedimiento `attend_request`, que activa la línea `BGT` correspondiente. Cuando el vigilante del bus del dispositivo demandante comprueba la activación de su línea `BGT`, activa la línea `SACK` y toma uso del bus. Al liberar el bus, el dispositivo desactiva la línea `SACK`.

Elemento de proceso

En la arquitectura de comportamiento del elemento de proceso, que se muestra en la figura 5.13, se especifica el algoritmo secuencial que ejecutan los trabajadores, que incluye una fase de inicialización a la que sigue el bucle de control principal, que comienza por la etiqueta `l_control_loop:`, y que consiste en:

- la condición de salida, que se basa en el contenido del registro de modo de ejecución *rm*,

- la búsqueda de la siguiente instrucción y

- una selección de casos para identificar y ejecutar la instrucción.

La instrucción `WAIT` que sigue al bucle principal asegura la parada del proceso VHDL.

Se han añadido procedimientos adicionales para:

```
ENTITY control IS
  PORT(Bus: INOUT t_BusLines;
       BRQs: IN t_BRQLines;
       BGTs: OUT t_BGTLines;
       SACK: IN t_SACKLine);
END control;

ARCHITECTURE behaviour OF control IS
...
    BEGIN
    l_control:
    LOOP
      IF no_pending_requests THEN
        EXIT l_control;
      END IF;
      select_asker(v_PriorityArray, v_Target);
      attend_request(v_Target);
    END LOOP l_control;
    WAIT;
  END PROCESS;
END behaviour;
```

Figura 5.12: Entidad y arquitectura del controlador del bus.

- Medida de parámetros de rendimiento.
  El procedimiento `printStatistics` recoge las estadísticas de los parámetros de rendimiento del sistema.

- Depuración.
  El procedimiento `printDebugInfo` da cuenta del estado de los trabajadores y el contenido de sus áreas de datos a diferentes niveles de detalle según se requiera.

- Análisis *post-mortem*.
  Los procedimientos `printTraceStart`, `printTraceStop`, `printTraceFork`, `printTraceJoin`, `printTraceStartTask` y `printTraceFinishTask` proporcionan la información necesaria para el análisis con VisAndOr [27] e IDRA [58].

Módulos de memoria compartida

La arquitectura de comportamiento de los módulos de memoria, que se muestra en la figura 5.14, incluye una fase de inicialización (`InitMemory`) a la que sigue el bucle de control principal, que comienza por la etiqueta `l_memory_loop:`. El primer paso de este bucle es la identificación de una petición de un elemento de proceso mediante `search_request`. En el segundo paso se comprueba si la petición atañe al módulo de memoria. Si es así, en el último paso se atiende la petición.

En el procedimiento `attend_request` se especifica el retardo correspondiente al tiempo de acceso a memoria mediante la instrucción `WAIT FOR v_Delay;`. Se han definido diferentes retardos para el acceso a las distintas áreas de memoria, e.g., memoria compartida, memoria local con caché y registros. Estos retardos marcan el avance de tiempo durante la simulación del sistema.

Interconexión de los componentes del sistema

Finalmente, se presenta la arquitectura estructural de interconexión de los componentes del sistema. Mediante esta arquitectura se describe la conexión de los puertos de los componentes con las líneas `BusLines`, `SACKLine`, `BRQLines` y `BGTLines` a través de la conexión `=>`. Con la construcción `FOR ... GENERATE` se declaran tantos componentes módulo de memoria y elemento de proceso como indican respectivamente las constantes `c_memory_modules`

```
ENTITY process_element IS
  GENERIC(Id: t_Id);
  PORT(Bus: INOUT t_BusLines;
       BRQ: OUT t_BRQLine;
       BGT: IN t_BGTLine;
       SACK: OUT t_SACKLine);
END process_element;

ARCHITECTURE behaviour OF process_element IS
...
      BEGIN
        printTraceStart;
        initWorker;
        l_control_loop:
        LOOP
          read(RM, v_Data);
          IF v_Data.value = cod(stop) THEN
            EXIT l_control_loop;
          END IF;
          printDebugInfo;
          fetch(v_OpCode);
          CASE v_OpCode IS
            WHEN InitBam =>
              .
              .
              .
          END CASE;
        END LOOP l_control_loop;
        printTraceStop;
        printStatistics(v_Debug);
        WAIT;
      END PROCESS;
...
END behaviour;
```

Figura 5.13: Entidad y arquitectura del elemento de proceso.

```
ENTITY memory_module IS
  GENERIC(Id: t_Id);
  PORT(Bus: INOUT t_BusLines;
       BRQ: OUT t_BRQLine;
       BGT: IN t_BGTLine;
       SACK: OUT t_SACKLine);
END memory_module;

ARCHITECTURE behaviour OF memory_module IS
...
      BEGIN
        InitMemory;
        l_memory_loop:
        LOOP
          WAIT ON Bus;
          search_request(v_Addr);
          IF own(v_Addr) THEN
            attend_request;
          END IF;
        END LOOP l_memory;
        WAIT;
      END PROCESS;
END behaviour;
```

Figura 5.14: Entidad y arquitectura del módulo de memoria compartida.

y c_process_elements. Cada puerto de petición o concesión de los dispositivos se conecta a la línea correspondiente del conjunto BGTLines. Se asigna un identificador a cada dispositivo a través del puerto GENERIC, al que se conecta el valor del array constante c_Ids, cuyos elementos son los identificadores.

```
ARCHITECTURE structural OF shared_memory_system IS

COMPONENT control
  PORT(Bus: INOUT t_BusLines;
       BRQs: IN t_BRQLines;
       BGTs: OUT t_BGTLines;
       SACK: IN t_SACKLine);
END COMPONENT;

COMPONENT process_element
  GENERIC(Id: t_Id);
  PORT(Bus: INOUT t_BusLines;
       BRQ: OUT t_BRQLine;
       BGT: IN t_BGTLine;
       SACK: OUT t_SACKLine);
END COMPONENT;

COMPONENT memory_module
  GENERIC(Id: t_Id);
  PORT(Bus: INOUT t_BusLines;
       BRQ: OUT t_BRQLine;
       BGT: IN t_BGTLine;
       SACK: OUT t_SACKLine);
END COMPONENT;

SIGNAL BusLines: t_BusLines;
SIGNAL SACKLine: t_SACKLine;
SIGNAL BRQLines: t_BRQLines;
SIGNAL BGTLines: t_BGTLines;

BEGIN
 controlunit:
   control
```

```
    PORT MAP(Bus  => BusLines,
             BRQs => BRQLines,
             BGTs => BGTLines,
             SACK => SACKLine);
memory_modules:
  FOR i IN 1 TO c_memory_modules GENERATE
   module_i:
    memory_module
     GENERIC MAP(Id => c_Ids(i))
     PORT MAP(Bus  => BusLines,
              BRQ  => BRQLines(i),
              BGT  => BGTLines(i),
              SACK => SACKLine);
process_elements:
  FOR i IN 1 TO c_process_elements GENERATE
   element_i:
    process_element
     GENERIC MAP(Id => c_Ids(i+c_memory_modules))
     PORT    MAP(Bus  => BusLines,
                 BRQ  => BRQLines(i+c_memory_modules),
                 BGT  => BGTLines(i+c_memory_modules),
                 SACK => SACKLine);
  END GENERATE processors;
END structural;
```

5.3.3 Medidas de rendimiento

Parámetros de rendimiento

Para la evaluación del sistema paralelo consideraremos los siguientes parámetros de rendimiento:

- Tasa de aciertos en memoria caché (*hits rate*).
  Es la relación entre el número de accesos a memoria resueltos en caché y el número total. Este parámetro mide la eficiencia de la política de coherencia caché para manejar datos compartidos.

- Retardo relativo debido al uso compartido del bus.
  Es la relación entre el tiempo de inactividad de un trabajador debido a la ocupación del bus y el tiempo total. Este parámetro da una medida

del cuello de botella que se produce por el uso compartido del bus y está determinada por la arquitectura y el arbitraje del bus.

- Ganancia de velocidad (*speed-up*).
  Es el cociente entre el tiempo de cómputo en la evaluación secuencial y paralela. Da una medida del rendimiento global del sistema. El paralelismo de los programas y los parámetros anteriores determinan la ganancia de velocidad.

Además de los parámetros anteriores estudiaremos:

- **Efecto de la estrategia de paralelización sobre la ganancia de velocidad**

  Se compara la ganancia de velocidad de los programas compilados con las tres estrategias presentadas en el capítulo 2.

- **Efecto de la planificación sobre la carga de trabajo**

  Se evalúa la repercusión de la política de planificación, descrita en el capítulo 4, sobre el reparto de trabajo en los elementos de proceso.

Programas de prueba

Las medidas de rendimiento se han realizado sobre los siguientes programas, que pueden encontrarse en el apéndice B.

- Problema de las reinas. Es la versión funcional del ejemplo clásico de la colocación en posiciones seguras de n reinas en un tablero de ajedrez de $n \times n$. Sus componentes paralelos son claramente independientes.

- Ordenación *Quicksort*. Es un programa usado frecuentemente en la medida de rendimiento de sistemas declarativos paralelos. Su componente secuencial limita la ganancia de velocidad.

- Números de Fibonacci. En este programa la granularidad de las expresiones decrece según avanza el cómputo. Si no se limita la planificación paralela de expresiones de bajo peso computacional, la ganancia de velocidad decrece. La versión que se ha utilizado permite la planificación paralela a partir de $fib(10)$.

- Torres de Hanoi. Este es un problema similar al anterior en cuanto a la granularidad y se permite la planificación paralela a partir de $hanoi(10)$.

Medidas obtenidas de la simulación del sistema paralelo

Estudiamos el rendimiento de sistemas de hasta 48 elementos de proceso. Las medidas mostradadas en las gráficas relativas al retardo debido al uso compartido del bus y a los aciertos en memoria caché se refieren a la media de estos parámetros de todos los elementos de proceso que intervienen.

- Porcentaje de aciertos en memoria caché.

  La figura 5.15 muestra el porcentaje de aciertos en memoria caché para el sistema con arbitraje por petición independiente y protocolo Firefly. A medida que aumenta el número de elementos de proceso decrece este parámetro debido a dos factores: la búsqueda de trabajo en pilas remotas y la espera a la consulta o actualización de un marcador de paralelismo remoto. En los programas *Reinas* y *Quicksort* se producen más accesos a memoria compartida debido al acceso a las listas calculadas en el punto de reunión. Sin embargo, estos accesos se resuelven principalmente en memoria caché, como se puede comparar con las figuras 5.16 y 5.17.

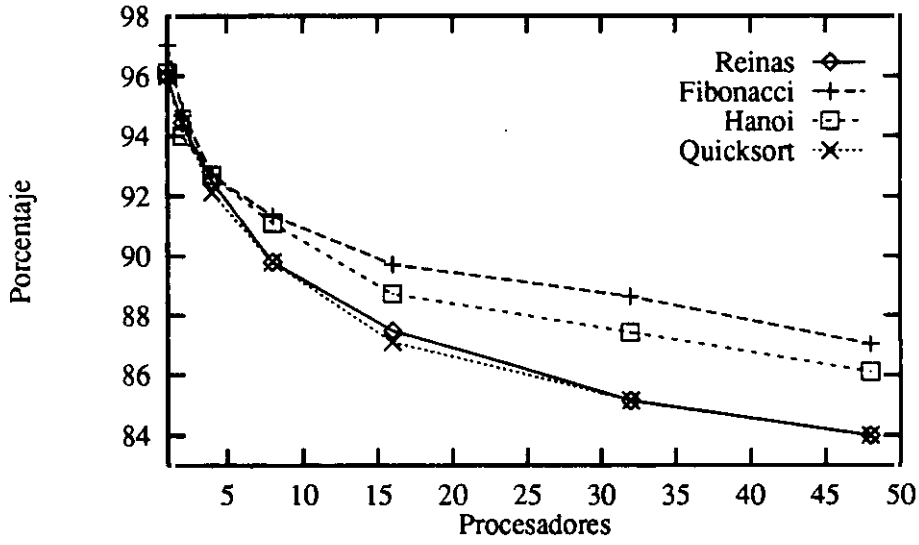


Figura 5.15: Porcentaje de aciertos en memoria caché.

- Tasa de retardo debido al uso compartido del bus.

  En la figura 5.16 se observa que al aumentar el número de elementos de proceso en el sistema anterior aumenta el porcentaje de retardo. Esto

es debido a que el tráfico del bus aumenta fundamentalmente por el acceso de los elementos de proceso remotos a las pilas de datos durante la búsqueda de trabajo. Este aumento coincide con la disminución de aciertos en memoria caché como se puede comparar con la gráfica de la figura 5.15. Las causas que explican la divergencia entre las curvas del porcentaje de retardo para cada programa coinciden con las expuestas en el apartado anterior para explicar la disminución de aciertos en caché.

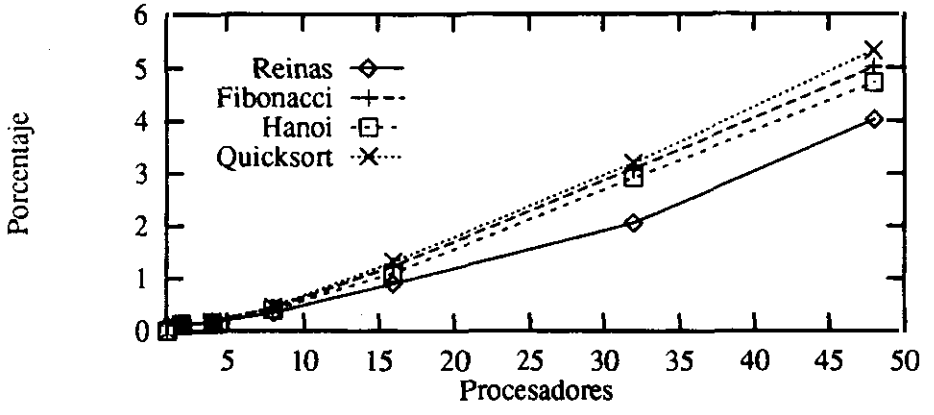


Figura 5.16: Tasa de retardo debido al uso compartido del bus.

- Ganancia de velocidad.

  La figura 5.17 muestra la ganancia de velocidad del sistema anterior para cuatro programas ejemplo. De ellos, el programa de las reinas tiene el mayor grado de independencia como muestra una curva de ganancia de velocidad casi lineal para el rango considerado de elementos de proceso. Esto demuestra que los cómputos se realizan casi completamente sobre la caché de cada elemento de proceso. Los programas *Hanoi* y *Fibonacci* presentan menor rendimiento debido al acceso más frecuente a los marcadores de paralelismo remotos. La granularidad de las expresiones paralelas de estos programas es significativamente menor que la de los demás. Las medidas corresponden a una versión de estos programas con control de granularidad, donde no se planifican en paralelo las expresiones de bajo peso computacional. El programa *Quicksort* tiene un componente secuencial importante que limita la ganancia de velocidad como predice la ley de Amdahl [3]. Debido a ello, se observa un decrecimiento de la ganancia de velocidad cuando

aumenta el número de elementos de proceso.

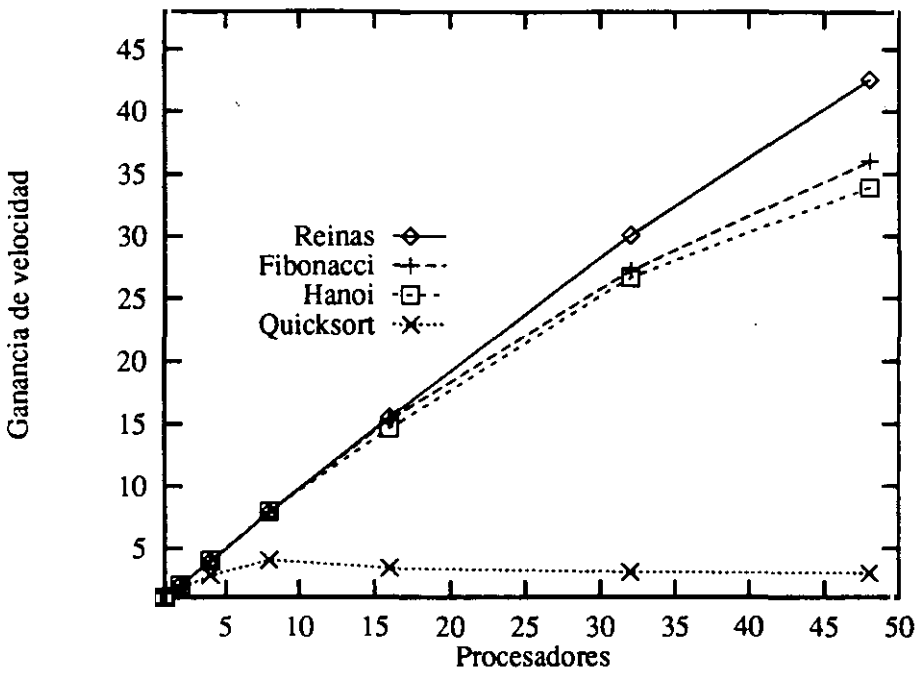


Figura 5.17: Implementación de sondeo con protocolo de coherencia Goodman.

Se observa que el decrecimiento de este parámetro es proporcional al crecimiento del retardo debido al uso compartido del bus, como se puede comparar con la figura 5.16.

La figura 5.18 muestra la relación entre la ganancia de velocidad del sistema con el protocolo Goodman y el Firefly. El rendimiento de éste último se observa ligeramente mejor que el del primero. Ello se debe al retardo que se produce cuando varios trabajadores esperan a la consulta de un marcador de paralelismo. Según el primer protocolo, cuando se modifica un marcador de paralelismo, se envía un mandato de actualización a todas las copias en caché. Esta operación se realiza más rápidamente que la situación homóloga en el protocolo Goodman, en donde primero se produce un fallo de lectura seguido de la lectura de la copia consistente. Este hecho es más apreciable en los programas *Quicksort* y *Reinas* en los que el resultado calculado por trabajadores hijos se consulta después del punto de reunión.

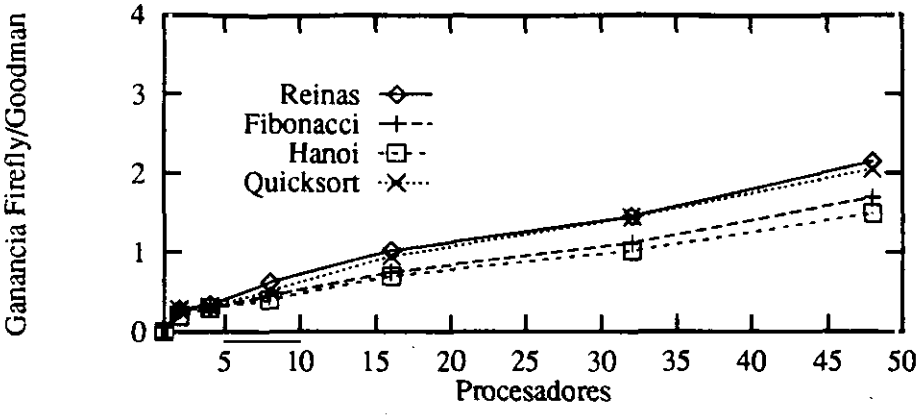


Figura 5.18: Protocolo Goodman *vs.* Firefly.

- Efecto de la estrategia de paralelización sobre la ganancia de velocidad.

  Se han tomado medidas de los programas de prueba, considerados en este capítulo y en el 2, compilados con las estrategias de paralelismo incondicional (*up*), básico (*gp*) y máximo (*mp*), así como de los programas con anotación manual de paralelismo (*mp*).

  En las tablas 5.1 y 5.2 se muestran los resultados obtenidos para cada programa. En cada fila se muestra la ganancia de velocidad para las estrategias *up*, *mp*, *gp* y *hp*, respectivamente. En las columnas se muestra la ganancia de velocidad para sistemas de 2 a 48 elementos de proceso.

  La ganancia de velocidad en los programas *Números de Fibonacci*, *Árboles balanceados* y *Derivación simbólica* es igual para las diferentes estrategias puesto que se obtiene la misma versión paralela en la compilación, que coincide con la paralelización manual. Esto ocurre también en los programas *Árboles balanceados* y *Hanoi*, pero en éstos, la paralelización manual es más eficiente puesto que no se obtiene toda la información de independencia en la compilación. En el resto de los programas se observa la sobrecarga debida a la evaluación de las condiciones de basicidad e independencia. En general se comprueba que la estrategia *gp* es más eficiente que la estrategia *mp*, ya que ésta genera expresiones con condiciones de independencia, mientras que aquélla lo hace sólo con condiciones de basicidad. Ambas identifican el mismo grado de paralelismo a diferente coste, como se puede observar en las tablas.

Programa	Estrategia	Procesadores					
		2	4	8	16	32	48
Números de Fibonacci	*up*	1.97	3.96	7.86	15.32	27.20	36.01
	mp	1.97	3.96	7.86	15.32	27.20	36.01
	gp	1.97	3.96	7.86	15.32	27.20	36.01
	hp	1.97	3.96	7.86	15.32	27.20	36.01
Aplanamiento de listas	*up*	0.99	0.99	0.98	0.98	0.97	0.95
	mp	1.49	2.25	4.35	6.49	12.47	13.85
	gp	1.49	2.27	4.40	6.55	12.69	14.10
	hp	1.51	2.30	4.49	6.87	13.12	15.04
Árboles balanceados	*up*	1.49	2.60	3.98	5.43	6.96	7.71
	mp	1.49	2.60	3.98	5.43	6.96	7.71
	gp	1.49	2.60	3.98	5.43	6.96	7.71
	hp	1.49	2.60	3.98	5.43	6.96	7.71
Sub-árboles	*up*	0.99	0.98	0.98	0.98	0.97	0.97
	mp	1.52	1.90	2.88	3.84	5.21	7.48
	gp	1.52	1.90	2.88	3.84	5.21	7.48
	hp	1.58	1.99	3.01	4.43	6.15	8.57

Tabla 5.1: Comparación de las estrategias de paralelización. I parte.

Programa	Estrategia	Procesadores					
		2	4	8	16	32	48
Torres de Hanoi	*up*	1.99	3.96	7.92	14.58	26.61	33.88
	mp	1.99	3.96	7.92	14.58	26.61	33.88
	gp	1.99	3.96	7.92	14.58	26.61	33.88
	hp	1.99	3.98	7.94	14.62	26.66	33.94
Derivación simbólica	*up*	1.97	3.87	7.75	13.12	19.98	26.50
	mp	1.97	3.87	7.75	13.12	19.98	26.50
	gp	1.97	3.87	7.75	13.12	19.98	26.50
	hp	1.97	3.87	7.75	13.12	19.98	26.50
Producto vector-matriz	*up*	0.99	0.98	0.97	0.97	0.96	0.94
	mp	1.94	3.24	6.86	10.98	16.81	20.93
	gp	1.94	3.25	6.90	11.22	16.85	21.41
	hp	1.95	3.70	7.53	12.94	18.73	25.32
Problema de las reinas	*up*	0.99	0.99	0.98	0.97	0.97	0.96
	mp	1.96	3.90	7.80	15.11	29.56	41.79
	gp	1.96	3.92	7.83	15.15	29.62	41.98
	hp	1.97	3.93	7.85	15.49	30.06	42.54

Tabla 5.2: Comparación de las estrategias de paralelización. II parte.

Se observa que las medidas de ganancia de velocidad obtenidas en este apartado difieren de las del capítulo 2. Este hecho es consecuencia de que el simulador presentado en dicho capítulo no tiene en cuenta el costo asociado a la explotación de paralelismo.

- Efecto de la planificación sobre la carga de trabajo.

  En este apartado se observa el comportamiento de la política de planificación distribuida que se presentó en el capítulo 4 y su efecto sobre la carga de trabajo.

  Para evaluar el impacto de la política de planificación sobre el rendimiento del sistema hemos utilizado la herramienta de visualización de trazas de ejecución VisAndOr [27][5]. Esta herramiente requiere como entrada un archivo de la traza de la ejecución de un programa. En este archivo se describe el tipo de paralelismo explotado (en nuestro caso, paralelismo conjuntivo), el inicio de un punto de activación de paralelismo, el inicio del cómputo efectivo de una expresión, su finalización y el instante en que sucede el punto de reunión. Los procedimientos `printTraceStart`, `printTraceStop`, `printTraceFork`, `printTraceJoin`, `printTraceStartTask` y `printTraceFinishTask` se encargan de la anotación de estos instantes de tiempo[6]. El gráfico que se puede visualizar con esta herramienta a partir de un archivo de traza representa la ejecución de un programa. El tiempo se representa en el eje de ordenadas avanzando hacia abajo. Un segmento de cómputo se representa con una línea vertical, que se anota con un identificativo del elemento de proceso que realiza el cómputo. Las líneas verticales punteadas representan retardos debidos a la planificación concurrente. Una planificación de una expresión concurrente se representa con una línea punteada horizontal. Las evaluaciones en paralelo de expresiones se pueden observar como líneas verticales cuya proyección en el eje de ordenadas se superpone. El gráfico representado es, pues, similar al árbol de evaluación abstracta. Las gráficas están escaladas para que cubran la totalidad de la figura, por lo que el tiempo de evaluación de las distintas gráficas es diferente.

[5]Con esta herramienta se puede visualizar tanto paralelismo conjuntivo como disyuntivo, en la línea de [55], un visualizador de paralelismo disyuntivo.

[6]Nótese que las medidas que realizan estos procedimientos no alteran la simulación del sistema, puesto que no intervienen lecturas de memoria sino medidas de tiempo de simulación.

En la figura 5.19 se puede observar para un caso simple el efecto de la planificación de expresiones sobre el rendimiento del sistema. En ella se pueden observar las trazas de la resolución del programa de los números de Fibonacci para tres trabajadores. Cada expresión planificada en paralelo se resuelve por un elemento de proceso distinto. En el lado izquierdo de la figura se encuentra la traza correspondiente a la planificación remota de expresiones de menor granularidad, mientras el lado derecho corresponde a una planificación remota de expresiones de mayor granularidad. La carga de trabajo en este último caso es más equitativa para los elementos de proceso de la red, en donde se produce el 36 % del cómputo efectivo en paralelo, frente al 22 % del otro caso. Para un mayor número de elementos de proceso este hecho redunda en una clara mejora del rendimiento.

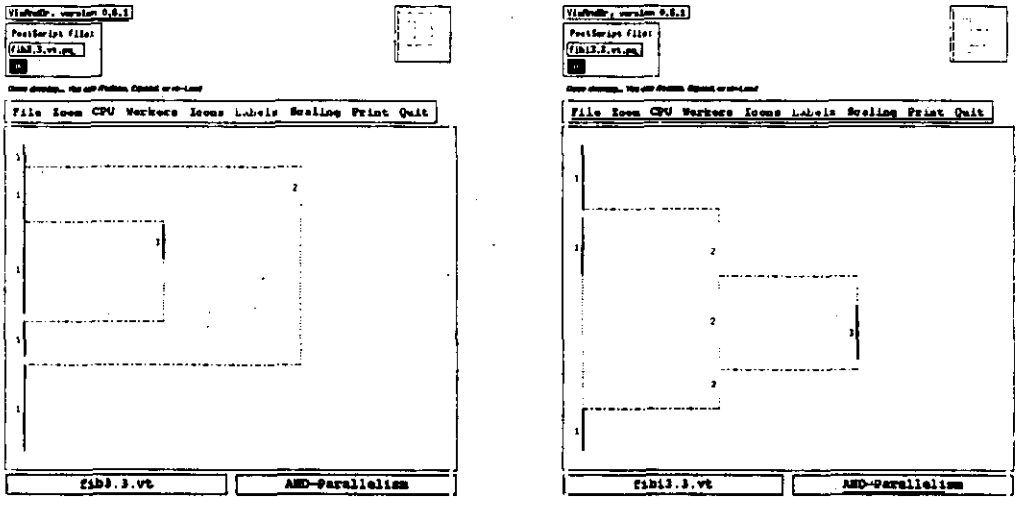


Figura 5.19: Efecto de la política de planificación en la carga de trabajo.

La condición de precedencia impone una restricción sobre la planificación que puede observarse en la figura 5.20. La figura muestra la evaluación de Fibonacci en una red de cinco elementos de proceso. En la gráfica de la izquierda se muestra la ejecución del programa limitada a cinco trabajadores, mientras que en la de la derecha no hay limitación. En la gráfica de la izquierda se observa que el reparto de trabajo

por trabajador se realiza prácticamente en el inicio de la ejecución del programa. Este reparto se hace relegando las expresiones de mayor granularidad a trabajadores remotos. Si las expresiones de mayor granularidad se reservan para ejecución local, no se produce ganancia de velocidad. Ello es debido a la condición de precedencia, que impide que los trabajadores remotos puedan evaluar nuevas expresiones del trabajador padre. Esto implica un aumento de ganancia de velocidad por debajo del esperado cuando aumenta el número de elementos de proceso. Sin embargo, si no se limita el número de trabajadores, ocurre la situación de la gráfica de la derecha, que implica una mejora de la ganancia de velocidad. En este caso se aprovecha el máximo paralelismo de la aplicación a costa de un mayor consumo y fragmentación de memoria debido al mayor número de trabajadores necesarios. Para reducir este efecto se puede plantear la adición pilas compartidas al sistema con una organización similar a los *meshed stack* [79].

Sumario

En este capítulo se ha desarrollado una implementación de la PEBAM sobre un multiprocesador simulado de memoria compartida considerando diferentes alternativas de diseño. De las medidas obtenidas de la simulación se ha concluido lo siguiente:

- El protocolo de actualización en escritura es más eficiente para programas como *Quicksort* y *Reinas* en los que se produce un acceso frecuente a los datos actualizados.

- Hay principalmente dos factores que hacen disminuir el porcentaje de aciertos en caché: el procedimiento de búsqueda de trabajo en pilas remotas, el acceso a marcadores remotos de paralelismo y el acceso a datos comunes actualizados.

- La implementación del bus por petición independiente es más eficiente que la de petición por sondeo, comprobándose que el aumento del tráfico del bus es proporcional a los fallos de referencia en memoria caché.

- Se consigue una ganancia de velocidad casi lineal en el rango considerado de elementos de proceso para algunos programas como *Reinas* y

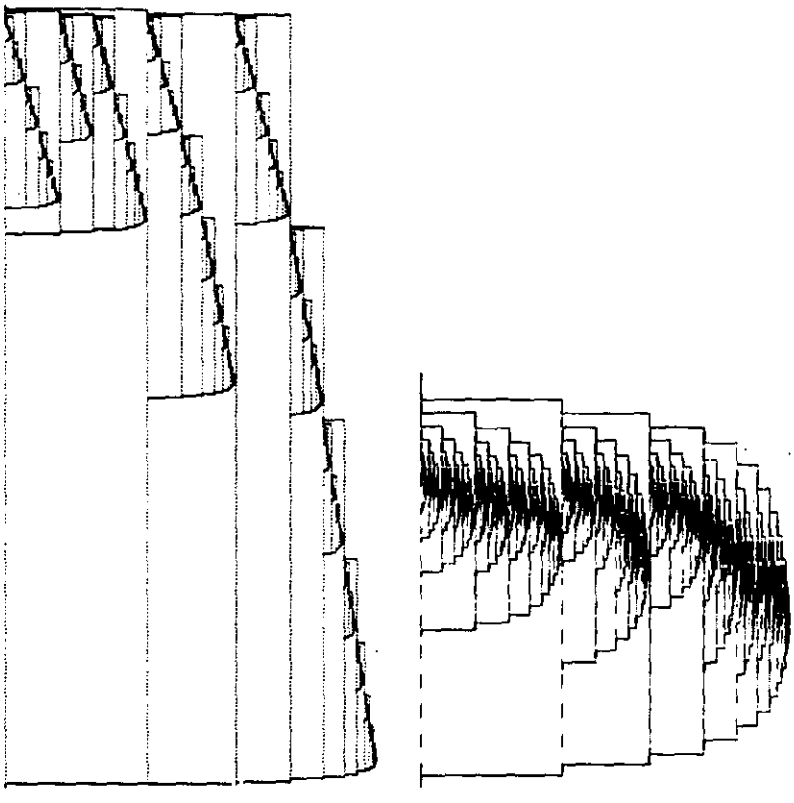


Figura 5.20: Efecto de la condición de precedencia en la planificación.

para otros como *Quicksort* se ve limitada por el componente secuencial del problema.

- La estrategia *gp* es más eficiente en la mayoría de los programas que *mp* puesto que se evitan comprobaciones de independencia en tiempo de ejecución.

- La política de planificación de trabajo, debido a la condición de precedencia, provoca un mayor consumo y fragmentación de memoria durante el cómputo hacia adelante, pero mejora significativamente la desasignación durante el cómputo hacia atrás si se compara con una implementación basada en grafos. No obstante, este problema se podría reducir en futuras implementaciones incorporando pilas compartidas.

- La incorporación de la información de granularidad es determinante para conseguir un buen rendimiento en términos de gestión de memoria y balance de carga.

Estas medidas se han realizado sobre sistemas de hasta 48 elementos de proceso. Considerar un número superior de elementos de proceso obliga a considerar otros tipos de arquitectura para disminuir el retardo debido al uso compartido del bus. Así, se podrían considerar redes de interconexión para aumentar el número de caminos entre elementos de proceso y módulos de memoria. Sin embargo, debido a la intensa difusión de mandatos de consistencia de la caché, el tiempo de latencia debido a la red puede resultar inaceptable. Por ello, sería necesario emplear un esquema de directorios en el diseño de la caché. También sería interesante considerar arquitecturas de bus/caché jerárquicas en donde se reduce el tráfico de la red gracias a protocolos jerárquicos de coherencia de caché, que no conllevan la complejidad de los esquemas de directorios.

Sería interesante proponer otras estrategias de planificación, centralizadas o híbridas, que consideren la granularidad de las tareas y el balance de carga.

Se podrían estudiar y comparar otras alternativas de diseño como caché de instrucciones, diferentes políticas de reemplazamiento y diferentes arquitecturas, como por ejemplo, memoria distribuida, memoria híbrida y redes de interconexión. Es interesante relacionar el efecto del reemplazamiento de las políticas de coherencia de la caché paralela, el protocolo de bus, las redes de interconexión, el tipo de memoria y otros, con el paralelismo identificado a alto nivel, de forma que se ajusten los parámetros de diseño del sistema paralelo al tipo de problemas que se resuelven. En particular, sería muy interesante la integración del paralelismo conjuntivo y el paralelismo disyuntivo sobre una arquitectura combinada de memoria compartida y distribuida. Esta combinación es interesante puesto que la explotación del paralelismo conjuntivo se adecúa perfectamente a su explotación en memoria compartida, debido al acceso frecuente a estructuras compartidas, y el disyuntivo a su explotación en memoria compartida, debido a la alta granularidad e independencia de las tareas, lo cual evita en gran medida las comunicaciones entre tareas.

Conclusiones y principales aportaciones

Conclusiones

El sistema de identificación y explotación de paralelismo desarrollado permite extraer las siguientes conclusiones.

De la medida de rendimiento del procedimiento de compilación que con una buena información de independencia en tiempo de compilación, las estrategias consiguen una buena identificación de paralelismo a bajo costo. Esta identificación es similar en la mayoría de los casos a la versión paralela manual. Sin embargo, con poca información de independencia, la estrategia *gp* puede identificar el mismo paralelismo que la estrategia *mp* a menor costo en determinados programas, aunque para otros no lo consigue. Se concluye, pues, que el rendimiento de las estrategias depende en gran medida del programa y de la información de independencia disponible.

De la medida de rendimiento de los niveles de análisis que, para determinados programas el nivel más simple consigue igual información de independencia que el resto. Sin embargo, este nivel no consigue la información útil para la mejora del componente secuencial que consiguen otros. Con los niveles de análisis $\mathcal{D}_1$ y $\mathcal{D}_1$ se infiere en muchos casos toda la información de independencia en un tiempo de análisis intermedio entre el nivel más simple y el más complejo. Con ellos se consigue además información de tipos para la mejora del componente secuencial de los programas.

Del diseño de la máquina abstracta paralela de memoria compartida se ha puesto de manifiesto cómo es posible extender las implementaciones eficientes basadas en pilas a una implementación de una máquina paralela lógico-funcional, reteniendo las optimizaciones de la programación lógica, funcional y lógico-funcional. En particular, esta máquina incorpora las optimizaciones de recursión de cola y corte dinámico, que mejoran el rendimiento

de la máquina tanto en presencia de constructoras secuenciales como paralelas. Se ha desarrollado el repertorio de instrucciones de la máquina, que es semejante al de la WAM en cuanto a la gestión del no determinismo y similar al de las máquinas de reducción de grafos en cuanto a las aplicaciones parciales y la gestión de la evaluación de funciones de primer orden y orden superior. Además, esta máquina incorpora otras optimizaciones como son la recursión de cola y el corte dinámico, que mejoran el rendimiento de la máquina tanto en presencia de constructoras secuenciales como paralelas.

Finalmente, de la simulación VHDL de la PEBAM sobre un multiprocesador simulado de memoria compartida se ha concluido que los protocolos de actualización de escritura redundan en mejor rendimiento que los de invalidación para ciertos programas. Se han identificado los factores que degradan el rendimiento de la caché y los programas con ganancia de velocidad casi lineal en el rango de elementos de proceso considerado y con el límite de ganancia predicho por la ley de Amdahl. La estrategia *gp* es más eficiente que *mp* para los programas en los que el paralelismo se identifica en tiempo de ejecución, debido a la evaluación de las condiciones de independencia. Además, la política de planificación permite retener la eficiencia de las implementaciones basadas en pilas, pero provocando un mayor consumo y fragmentación de memoria que podría evitarse con pilas compartidas. Se ha observado que la incorporación de información de granularidad en esta política es determinante de un buen rendimiento del sistema.

Principales aportaciones

En este trabajo se ha propuesto:

- Una extensión del modelo de ejecución secuencial a un modelo de ejecución paralela para la versión de estrechamiento impaciente de Babel. Este modelo permite la evaluación paralela de funciones estrictas y predefinidas no estrictas.

- Un método automático de identificación de paralelismo. Este método incluye tres estrategias diferentes de identificación que aprovechan la información de independencia y granularidad proporcionada por análisis de programas o anotación manual.

- Una técnica de análisis de programas para inferir información de independencia. Esta técnica, basada en interpretación abstracta, desa-

rrolla tres niveles de análisis para obtener tres grados de información de independencia.

- Una máquina abstracta paralela de memoria compartida (PEBAM). La máquina utiliza las técnicas eficientes de pilas que retiene las optimizaciones presentes en las máquinas abstractas secuenciales lógicas y lógico-funcionales y se ha desarrollado la compilación de programas paralelos en instrucciones máquina.

- Una implementación de la PEBAM sobre un multiprocesador simulado de memoria compartida.
  Se ha realizado la validación funcional del sistema con diferentes alternativas de diseño y su medida de rendimiento.

Sugerencias para trabajos futuros

- Extensión del sistema para permitir otras estrategias de cómputo, en particular, la evaluación perezosa.

- Desarrollo de una implementación más rápida del analizador con técnicas de ejecución simbólica como la que desarrollamos en [139] en el ámbito de la programación lógica.

- Comparación equitativa de la eficiencia de sistemas lógicos *vs.* lógico-funcionales con respecto a la evaluación de programas homólogos. Esto se podría llevar a cabo con una simulación de alto nivel, esto es, considerando como parámetro de medida de rendimiento el paso de resolución para los programas lógicos y de estrechamiento para los lógico-funcionales.

Apéndice A

Sintaxis y semántica de Babel

En este apéndice se presentan la sintaxis y semántica operacional de Babel[1].

Babel es un lenguaje lógico-funcional con una disciplina de constructoras[2] y un sistema de tipos polimórficos. La sintaxis de Babel es netamente funcional y su semántica operacional se basa en estrechamiento de grafos (*graph narrowing*). Consideramos la versión de orden superior y estrechamiento impaciente en profundidad y de izquierda a derecha.

A.1 Sintaxis de Babel

Comenzaremos identificando los conjuntos disjuntos de símbolos que se definen en Babel.

- Variables de datos: $X, Y, \ldots \in Var$. Son variables lógicas. Como en Prolog, se escriben con mayúsculas para diferenciarlas de los demás símbolos del lenguaje.

- Constructoras de datos: $c, d, \ldots \in DC$. Juegan el papel de los funtores de Prolog, cada una tiene asociada una aridad i, y distinguimos $DC^0 \cup \ldots \cup DC^i \cup \ldots \cup DC^n = DC$, donde DC^i es el conjunto de los símbolos de constructoras de datos de aridad i. Las constructoras de aridad cero se llaman alternativamente constantes.

[1] Para consultar la semántica declarativa, las equivalencias entre semánticas y las pruebas de corrección y completud, véanse [115, 116, 119].

[2] La disciplina de constructoras significa que se diferencian los símbolos de constructora de datos y los símbolos de función, permitiendo el encaje de símbolos de constructoras con símbolos de funciones sin interpretar.

- Símbolos de función: $f, g, \ldots \in FS$. Al igual que las constructoras de datos, cada uno tiene asociado una aridad i, y distinguimos $FS^0 \cup \ldots \cup FS^i \cup \ldots \cup FS^n = FS$, donde FS^i es el conjunto de los símbolos de funciones de aridad i.

Un término de datos $t \in Term$ [3] puede ser una variable o una aplicación de una constructora n-aria con n argumentos que a su vez son términos[4].

$t ::= X$ % variable
$\mid c\ t_1 \ldots t_n$ % constructora de datos n-aria c

Una expresión $e \in Exp$ tiene la forma:

$e ::=$	t	% término
$\mid$	$c\ e_1 \ldots e_n$	% $c \in DC^m\ (0 \leq n \leq m), e_i \in Exp$
$\mid$	$f\ e_1 \ldots e_n$	% $f \in FS^m\ (0 \leq n \leq m), e_i \in Exp$
$\mid$	$b \rightarrow e$	% expresión guardada (si b entonces e, si no indefinido)
$\mid$	$b \rightarrow e_1 \Box e_2$	% expresión condicional (si b entonces e_1, si no e_2)

Además de las funciones guarda y condicional, el lenguaje incorpora las funciones l'ogicas primitivas conjunción (,), disyunción (;) y negación ($\neg$), definidas como sigue:

$(false, X)$	$:= false.$	$(true; X)$	$:= true.$	$\neg false$	$:= true.$
$(X, false)$	$:= false.$	$(X; true)$	$:= true.$	$\neg true$	$:= false.$
$(true, true)$	$:= true.$	$(false; false)$	$:= false.$		

El lenguaje también incluye igualdad débil, definida por las siguientes reglas:

$(c = c) := true.$ % $c \in DC^0$
$(c\ X_1 \ldots X_n = c\ Y_1 \ldots, Y_n) := (X_1 = Y_1), \ldots, (X_n = Y_n).$
% $c \in DC^n, n > 0$
$(c\ X_1 \ldots X_n = d\ Y_1 \ldots Y_m) := false.$% $c \in DC^n, d \in DC^m, c \neq don \neq m$

[3] De aquí en adelante abreviaremos término de datos como término.

[4] En lo que sigue, utilizaremos la notación currificada para permitir la expresión de aplicaciones parciales para símbolos de constructora y de función, y la expresión de orden superior para símbolos de función. Asumimos que la aplicación es asociativa por la izquierda, escribiendo paréntesis cuando sea necesario o para aclarar la expresión.

Un programa Babel consiste en una secuencia finita de definiciones de función que puede ser consultado con una expresión objetivo. Las funciones Babel son funciones en el sentido matemático, i.e., a lo sumo existe un resultado para cada tupla de argumentos básicos[5], lo que se asegura por las condiciones sobre las funciones que se presentan más adelante (ver [116] para más detalles) . Cada función f es una secuencia finita y enumerable de reglas de definición de función. Una regla de definición de función tiene la forma[6] :

$$\underbrace{f\ t_1 \ldots t_n}_{\text{parte izquierda } (lhs)} \quad := \quad \underbrace{\underbrace{\{b \rightarrow\}}_{\text{guarda opcional}} \quad \underbrace{e}_{\text{cuerpo}}}_{\text{parte derecha } (rhs)} .$$

donde f es el nombre de la función, t_i son términos de datos, b una expresión l'ogica y e una expresión arbitraria.

Las reglas de función deben cumplir las siguientes condiciones:

1. *Patrones de datos:* $t_i \in Term$.

2. *Linealidad por la izquierda:* $f\ t_1 \ldots t_n$ no contiene variables repetidas.

3. *Restricciones sobre las variables libres:* cualquier variable que sólo aparezca en la *rhs* se llama *libre* (se permite que las variables libres aparezcan en el guarda, pero no en el cuerpo).

4. *No ambigüedad:* Dadas dos reglas cualesquiera de una definición de una función f: $f\ t_1 \ldots t_n := \{b_1 \rightarrow\}e_1$ y $f\ s_1 \ldots s_n := \{b_2 \rightarrow\}e_2$, no pueden devolver diferentes valores con los mismos argumentos básicos de llamada. Esta condición sintáctica se asegura imponiendo que se cumpla al menos uno de los siguientes casos:

   - *No superposición*: Las dos *lhs* no son unificables.
   - *Fusión de cuerpos*: Las dos *lhs* tienen un u.m.g. σ tal que $e_1\sigma$ y $e_2\sigma$ son sintácticamente idénticas.
   - *Incompatibilidad de guardas*: Las dos *lhs* tienen un u.m.g. σ tal que $(b_1\sigma, b_2\sigma)$ no se puede satisfacer, i.e., no se puede reducir al valor booleano *true*.

[5]Esta restricción de confluencia no impide obtener diferentes *sustituciones* de éxito como en el caso de la programación lógica como veremos más adelante.

[6]En la notación utilizamos corchetes ({ }) para denotar una posible aparición sintáctica.

Babel dispone de tipos *polimórficos*, i.e., tipos con apariciones de variables de tipo. El símbolo $\rightarrow$ (aplicación entre conjuntos) es asociativo por la derecha. Se entiende que una constructora de datos c de aridad n se declara como:

$$c : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$$

donde τ y τ_i $(1 \leq i \leq n)$ son tipos de primer orden. Una función f de aridad n se declara como:

$$f : \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$$

donde τ es un tipo de primer orden, y τ_i $(1 \leq i \leq n)$ puede ser un tipo de orden superior. Las declaraciones de los tipos de función se realizan con la palabra reservada *fun* y las declaraciones de tipos de datos con *datatype*. Los identificadores que comienzan por mayúscula se utilizan para variables y para variables de tipo.

Existen varios tipos predefinidos:
datatype $bool := true \mid false.$
datatype $nat := 0 \mid (suc\ nat).$
datatype $list\ A := nil \mid (cons\ A\ (list\ A)).$
fun $\neg : bool \rightarrow bool.$
fun , $: bool \rightarrow bool \rightarrow bool.$
fun ; $: bool \rightarrow bool \rightarrow bool.$
fun $= : A \rightarrow A \rightarrow bool.$

Finalmente, describimos la notación alternativa para los predicados lógicos, que se basa en la sintaxis de Prolog.

$p\ t_1 \ldots t_n := true.$	$p(t_1, \ldots, t_n).$
$p\ t_1 \ldots t_n := b \rightarrow true.$	$p(t_1, \ldots, t_n)\ :-\ b.$

Donde b es la expresión l'ogica que representa el cuerpo de la cláusula Prolog. Nótese que el sentido de la negación en Babel es más preciso que en Prolog, puesto que también se admiten las siguientes reglas:

$p\ t_1 \ldots t_n := false.$	$\neg p(t_1, \ldots, t_n).$
$p\ t_1 \ldots t_n := b \rightarrow false.$	$\neg p(t_1, \ldots, t_n)\ :-\ b.$

A.2 Semántica operacional de Babel

Una expresión $e \in Exp$ que no contenga variables de orden superior puede interpretarse en un programa Babel aplicando el mecanismo de estrechamiento[7]. La aplicación de estrechamiento a una expresión consiste en calcular la mínima sustitución que haga reducible la expresión, finalmente reduciéndola.

[7] Propuesto por primera vez por [133] en el marco funcional.

Esta mínima sustitución se calcula por unificación con la *lhs* de una regla. Un paso de estrechamiento consiste en el cálculo de esta unificación y en la reescritura de la expresión utilizando la *rhs* de la regla. La *respuesta computada* (*outcome*) de un paso de estrechamiento es la expresión reescrita junto a la sustitución derivada de la unificación. Consideramos la versión impaciente de estrechamiento en profundidad y de izquierda a derecha (*leftmost innermost narrowing*).

Más formalmente, sean:

- $e \equiv f\ e_1 \ldots e_n$ una expresión
- $f\ t_1 \ldots t_n := e'$ una variante de una regla que no comparte variables con e, y
- $\sigma \circ \lambda$ [8] el *unificador más general* (u.m.g.) de e y $f\ t_1 \ldots t_n$, i.e., la mínima sustitución tal que $e\sigma$ es sintácticamente idéntica a $(f\ t_1 \ldots t_n)\lambda$,

e puede estrecharse en un paso a $e'' \equiv e'\lambda$ con sustitución de éxito σ (denotado por $e \Rightarrow_\sigma e''$). Finalmente, siendo $\varphi \in FS \cup DC$, si para algún i $(0 \leq i \leq n)$ $e_i \Rightarrow_\sigma e_i'$, entonces $\varphi(e_0, \ldots, e_i, \ldots, e_n) \Rightarrow_\sigma \varphi(e_0\sigma, \ldots, e_i', \ldots, e_n\sigma)$.

Para incorporar las funciones primitivas, el mecanismo de estrechamiento se extiende con las siguientes *reglas de estrechamiento.*

$(false, b) \Longrightarrow_\varepsilon false$ $\quad$ $(true; b) \Longrightarrow_\varepsilon true$
$(true, b) \Longrightarrow_\varepsilon X$ $\quad$ $(false; b) \Longrightarrow_\varepsilon b$
$(true, true) \Longrightarrow_\varepsilon true$ $\quad$ $(false; false) \Longrightarrow_\varepsilon false$
$(true \to e_1 \square e_2) \Longrightarrow_\varepsilon e_1$ $\quad$ $(false \to e_1 \square e_2) \Longrightarrow_\varepsilon e_2$
$(true \to b) \Longrightarrow_\varepsilon b$

donde ε denota la sustitución vacía $(\varepsilon(X) = X, \forall X \in Var)$.

La relación de un paso de estrechamiento $\Longrightarrow_\sigma \subseteq Exp \times Exp$ donde $\sigma : Var \to Exp$ se extiende canónicamente a expresiones arbitrarias, definida inductivamente como:

- $$\frac{e_i \Longrightarrow_\sigma e_i'}{\varphi\ e_1 \ldots e_i \ldots e_n \Longrightarrow_\sigma \varphi\ e_1\sigma \ldots e_i' \ldots e_n\sigma}$$
- $$\frac{b \Longrightarrow_\sigma b'}{(b \to e) \Longrightarrow_\sigma (b' \to e\sigma)}$$
- $$\frac{b \Longrightarrow_\sigma b'}{(b \to e_1 \square e_2) \Longrightarrow_\sigma (b' \to e_1\sigma \square e_2\sigma)}$$

[8] Como es usual, las sustituciones se aplican en notación postfija. Por otra parte, $e(\sigma \circ \lambda) = (e\sigma)\lambda$.

La secuencia de estrechamiento de una expresión e se denota como el cierre transitivo y reflexivo $\stackrel{*}{\Longrightarrow}_\sigma$ de $\Longrightarrow_{\sigma'}$ con la composición de sustituciones.

El resultado del estrechamiento puede dar lugar a uno de los siguientes casos:

- *Éxito:* $e \stackrel{*}{\Longrightarrow}_\sigma t$, $t \in Term$ con *sustitución de respuesta* σ.
- *Fallo:* $e \stackrel{*}{\Longrightarrow}_\sigma e'$, $e' \notin Term$ y $\Longrightarrow_\sigma$ no se puede volver a aplicar a e'.
- *No terminación.*

La secuencia de pasos $e \Rightarrow_{\sigma_1} \ldots \Rightarrow_{\sigma_m} e'$ se llama *cómputo* de la expresión e. Si e' es un término, se ha hallado una *solución* o *respuesta computada* del cómputo, que se compone del resultado e' y de la *respuesta* $\sigma = \sigma_1 \circ \ldots \circ \sigma_m$. Si e no se puede estrechar más y no es un término o aplicación parcial, entonces el cómputo *falla*. En este caso, se prueban consecutivamente todas las reglas de definición de función aplicables a e.

Hay ciertos términos predefinidos especiales que determinan el resultado de las expresiones que tienen como símbolo raíz una función primitiva lógica. Son los términos *true* y *false*, que denominamos valores *definitorios* de la disyunción y la conjunción, respectivamente. Si el resultado de un operando e_i de la función lógica es un valor definitorio, la solución de la expresión está definida descartando las soluciones de los operandos a la derecha de e_i.

Apéndice B

Programas de prueba

En este apéndice se presentan los programas de prueba que se han utilizado para el análisis del procedimiento de identificación de paralelismo descrito en el capítulo 2, así como para el análisis de independencia basado en interpretación abstracta descrito en el capítulo 3 y para el análisis de rendimiento del sistema de explotación de paralelismo del capítulo 5.

B.1 Sintaxis

La sintaxis de los programas se ha desarrollado para la implementación del sistema paralelo. Así, en los listados que se proporcionan, el símbolo % denota un comentario en una línea, mientras que los símbolos /* y */ agrupan un comentario que puede contener más de una línea.

La declaración de tipos se realiza con los hechos `list_of_constants`, `list_of_constructors` y `list_of_functions`, en los que se declaran respectivamente las constantes, constructoras y funciones de cada programa.

La declaración de modos de los argumentos se realiza con el hecho `mode`. Tiene como argumento un término de datos compuesto por un funtor, que representa el símbolo de una función f, y con argumentos que representan los modos asociados a los argumentos de f. Los modos pueden ser `g` o `d`, que denotan respectivamente un modo *ground* (argumento de entrada o básico) o *top* (desconocido, i.e., puede ser un argumento de entrada o de salida).

Una condición de independencia se representa por la conjunción $(,/2)$ de condiciones simples de independencia, representadas por $g(X)$ cuando se trata de una condición de basicidad, y por $i(X,Y)$ cuando se trata de una condición de independencia.

Una expresión condicional secuencial se expresa con las constructoras infijas ==> (*if*) y # (*else*).

Las *EGEs* se construyen con:

`par(`*List*`)`
`cpar(`*List*`)`
`seq(`*List*`)`
`if(`*BoolExpr*`,`*ThenExpr*`,`*ElseExpr*`)`

donde *List* es la lista con sintaxis Prolog en la que se definen las variables auxiliares de cada unidad evaluable, *BoolExpr* es una expresión lógica y *ThenExpr* y *ElseExpr* son expresiones del mismo tipo correspondientes respectivamente a la rama *then* y a la rama *else* del condicional.

Las *EGEs* se encuentran en el cuerpo de la constructora funcional *let-in*, cuya sintaxis es:

`let` *EGE* `in` *Expr*

donde *Expr* es la expresión que contiene las variables auxiliares definidas en *EGE*.

B.2 Listado de los programas de prueba

Se proporcionan los siguientes programas:

- Árboles balanceados.
- Sumador binario.
- Derivación simbólica.
- Números de Fibonacci.
- Aplanamiento de listas.
- Función de Ackermann.
- Torres de Hanoi.
- Problema de las reinas.
- Ordenación *quicksort*.
- Subárboles.
- Producto vector matriz.

Para cada uno de ellos se proporcionan varias versiones:

- Versión secuencial.
- Versión paralela compilada con la estrategia de paralelismo incondicional con y sin anotación de modos.
- Versión paralela compilada con la estrategia de paralelismo máximo con y sin anotación de modos.
- Versión paralela compilada con la estrategia de paralelismo básico con y sin anotación de modos.
- Versión paralela realizada manualmente.

A continuación se presentan los listados de los programas en sus diferentes versiones.

% Árboles balanceados. Versión secuencial

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := eqlength(N, T1), eqlength(N, T2).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).

goal :- eval(balanced(both(both(leaf(a),
                                leaf(b),
                                leaf(c)),
                           leaf(a),
                           both(leaf(a),
                                leaf(b),
                                leaf(c))))).
/*goal :- eval(balanced(both(leaf(a),
                           leaf(b),
                           leaf(c)))).*/
% goal :- eval(balanced(leaf(a))).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).
```

% Árboles balanceados. Estrategia *gp* con anotación de modos

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := and(g(N), eqlength(N, T1), eqlength(N, T2)).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).

goal :- eval(balanced(both(both(leaf(a),
                                leaf(b),
                                leaf(c)),
                           leaf(a),
                           both(leaf(a),
                                leaf(b),
                                leaf(c))))).
% mode(balanced(g)).
```

% Árboles balanceados. Estrategia *gp*

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := and((g(N),g(T1)), eqlength(N, T1), eqlength(N, T2)).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).

goal :- eval(balanced(both(both(leaf(a),
                                leaf(b),
                                leaf(c)),
                           leaf(a),
                           both(leaf(a),
                                leaf(b),
                                leaf(c))))).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).
```

% Árboles balanceados. Paralelización manual

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
```

```
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := and(g(N), eqlength(N, T1), eqlength(N, T2)).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).

goal :- eval(balanced(both(both(leaf(a),
                                leaf(b),
                                leaf(c)),
                           leaf(a),
                           both(leaf(a),
                                leaf(b),
                                leaf(c))))).
```

% Árboles balanceados. Estrategia *mp* con anotación de modos

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := and(g(N), eqlength(N, T1), eqlength(N, T2)).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).

goal :- eval(balanced(both(both(leaf(a),
                                leaf(b),
                                leaf(c)),
                           leaf(a),
                           both(leaf(a),
                                leaf(b),
                                leaf(c))))).
% mode(balanced(g)).
```

% Árboles balanceados. Estrategia *mp*

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := and((g(N),i(T1,T2)), eqlength(N, T1), eqlength(N, T2)).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).
```

```
goal :- gen(T,8),eval(balanced(T)).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).

gen(leaf(a), 0).
gen(both(T1, leaf(a), T2), N) :- N1 is N - 1, gen(T1, N1), gen(T2, N1).
```

% Árboles balanceados. Estrategia *up* con anotación de modos

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := and(g(N), eqlength(N, T1), eqlength(N, T2)).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).

goal :- eval(balanced(both(both(leaf(a),
                                leaf(b),
                                leaf(c)),
                           leaf(a),
                           both(leaf(a),
                                leaf(b),
                                leaf(c))))).
% mode(balanced(g)).
```

% Árboles balanceados. Estrategia *up*

```
eqlength(zero, leaf(X)) := true.
eqlength(suc(N), leaf(X)) := false.
eqlength(suc(N), left(T, X)) := eqlength(N, T).
eqlength(zero, left(T, X)) := false.
eqlength(suc(N), rigth(T, X)) := eqlength(N, T).
eqlength(zero, rigth(T, X)) := false.
eqlength(suc(N), both(T1, X, T2)) := and(false, eqlength(N, T1), eqlength(N, T2)).
eqlength(zero, both(T1, X, T2)) := false.

balanced(T) := eqlength(N, T), write(N).

goal :- eval(balanced(both(both(leaf(a),
                                leaf(b),
                                leaf(c)),
                           leaf(a),
                           both(leaf(a),
                                leaf(b),
                                leaf(c))))).
```

```
/*goal :- eval(balanced(both(leaf(a),
                           leaf(b),
                           leaf(c)))).*/
% goal :- eval(balanced(leaf(a))).

list_of_constants([a, b, c, zero]).
list_of_constructors([leaf, both, left, rigth, suc]).
list_of_functions([balanced, eqlength]).
```

% Sumador binario. Versión secuencial

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
adder(X, Y, Ci) := out(xor(Ci, xor(X, Y)), or(and(X, Y), and(Ci, xor(X, Y)))).

list_of_constants([]).
list_of_constructors([out]).
list_of_functions([and, or, xor, adder]).

goal :- eval(adder(1,1,1)).
```

% Sumador binario. Estrategia *gp* con anotación de modos

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
adder(X, Y, Ci) :=
        let
                par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                     seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                          A5 &= and(Ci, A4),
                          A6 &= or(A3, A5)])])
        in out(A2, A6).

list_of_constants([]).
list_of_constructors([out]).
list_of_functions([and, or, xor, adder]).
```

```
goal :- eval(adder(1,1,1)).

% mode(adder(g,g,g)).
```

% Sumador binario. Estrategia *gp*

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
/*
adder(X, Y, Ci) :=
        let if(i(X,Y),
               if((i(X,Ci),i(Y,Ci)),
%                  true2,
                   par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                        seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                             A5 &= and(Ci, A4),
                             A6 &= or(A3, A5)])]),
%                   false2),
                   seq([A1 &= xor(X, Y),
                        A2 &= xor(Ci, A1),
                        seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                             A5 &= and(Ci, A4),
                             A6 &= or(A3, A5)])])),
%                false1)
               seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1), A3 &= and(X, Y),
                    A4 &= xor(X, Y), A5 &= and(Ci, A4), A6 &= or(A3, A5)]))
        in out(A2, A6).
*/

adder(X, Y, Ci) :=
        let if(g(X),
               if(g(Y),
%                  a,
                   par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                        seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                             A5 &= and(Ci, A4),
                             A6 &= or(A3, A5)])]),
                   if(g(Ci),
%                      a,
                   par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                        seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                             A5 &= and(Ci, A4),
                             A6 &= or(A3, A5)])]),
%                      a)),
                   par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
```

```
                    seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                         A5 &= and(Ci, A4),
                         A6 &= or(A3, A5)])])),
            if(g(Y),
               if(g(Ci),
%                 a,
                  par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                       seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                            A5 &= and(Ci, A4),
                            A6 &= or(A3, A5)])]),
%                 a),
                  par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                       seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                            A5 &= and(Ci, A4),
                            A6 &= or(A3, A5)])])),
               if(g(Ci),
%                 a,
                  par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                       seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                            A5 &= and(Ci, A4),
                            A6 &= or(A3, A5)])]),
%                 b)))
            seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1), A3 &= and(X, Y),
                 A4 &= xor(X, Y), A5 &= and(Ci, A4), A6 &= or(A3, A5)])))
        in out(A2, A6).

list_of_constants([]).
list_of_constructors([out]).
list_of_functions([and, or, xor, adder]).

goal :- eval(adder(1,1,1)).
```

% Sumador binario. Paralelización manual

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
adder(X, Y, Ci) :=
        let
                par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                     seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                          A5 &= and(Ci, A4),
                          A6 &= or(A3, A5)])])
        in out(A2, A6).

list_of_constants([]).
list_of_constructors([out]).
```

```
list_of_functions([and, or, xor, adder]).

goal :- eval(adder(1,1,1)).

% mode(adder(g,g,g)).
```

% Sumador binario. Estrategia *mp* con anotación de modos

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
adder(X, Y, Ci) :=
        let
                par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                     seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                          A5 &= and(Ci, A4),
                          A6 &= or(A3, A5)])])
        in out(A2, A6).

list_of_constants([]).
list_of_constructors([out]).
list_of_functions([and, or, xor, adder]).

goal :- eval(adder(1,1,1)).

% mode(adder(g,g,g)).
```

% Sumador binario. Estrategia *mp*

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
adder(X, Y, Ci) :=
        let if(i(X,Y),
               if((i(X,Ci),i(Y,Ci)),
%                  true2,
                   par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                        seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
```

```
                              A5 &= and(Ci, A4),
                              A6 &= or(A3, A5)])]),
%                    false2),
                     seq([A1 &= xor(X, Y),
                          A2 &= xor(Ci, A1),
                          seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                               A5 &= and(Ci, A4),
                               A6 &= or(A3, A5)])])),
%                 false1)
                  seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1), A3 &= and(X, Y),
                       A4 &= xor(X, Y), A5 &= and(Ci, A4), A6 &= or(A3, A5)]))
        in out(A2, A6).

list_of_constants([]).
list_of_constructors([out]).
list_of_functions([and, or, xor, adder]).

goal :- eval(adder(1,1,1)).
```

% Sumador binario. Estrategia *up* con anotación de modos

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
adder(X, Y, Ci) :=
        let
                  par([seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1)]),
                       seq([par([A3 &= and(X, Y), A4 &= xor(X, Y)]),
                            A5 &= and(Ci, A4),
                            A6 &= or(A3, A5)])])
        in out(A2, A6).

list_of_constants([]).
list_of_constructors([out]).
list_of_functions([and, or, xor, adder]).

goal :- eval(adder(1,1,1)).

% mode(adder(g,g,g)).
```

% Sumador binario. Estrategia *up*

```
xor(0,0) := 0.
xor(0,1) := 1.
xor(1,0) := 1.
xor(1,1) := 0.
or(0,0) := 0.
```

```
or(0,1) := 1.
or(1,0) := 1.
or(1,1) := 1.
and(0,0) := 0.
and(0,1) := 0.
and(1,0) := 0.
and(1,1) := 1.
adder(X, Y, Ci) :=
        let seq([A1 &= xor(X, Y), A2 &= xor(Ci, A1), A3 &= and(X, Y),
                 A4 &= xor(X, Y), A5 &= and(Ci, A4), A6 &= or(A3, A5)])
        in out(A2, A6).

list_of_constants([]).
list_of_constructors([out]).
list_of_functions([and, or, xor, adder]).

goal :- eval(adder(1,1,1)).
```

% Derivación simbólica. Versión secuencial

```
derive(add(U,V)) := add(derive(U), derive(V)).
derive(sub(U,V)) := sub(derive(U), derive(V)).
derive(mul(U,V)) := add(mul(U, derive(V)), mul(V, derive(U))).
derive(div(U,V)) := div(sub(mul(U, derive(V)), mul(V, derive(U))), mul(V, V)).
derive(pot(U,c(N))) := mul(mul(c(N), pot(U, c(N-1))), derive(U)).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).

goal :- eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))).
```

% Derivación simbólica. Estrategia *gp* con anotación de modos

```
derive(add(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
                    in add(A1, A2).
derive(sub(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
                    in sub(A1, A2).
derive(mul(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
                    in add(mul(U, A1), mul(V, A2)).
derive(div(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
                    in div(sub(mul(U, A1), mul(V, A2)), mul(V, V)).
derive(pot(U,c(N))) := let par([A1 &= N - 1, A2 &= derive(U)])
                       in mul(mul(c(N), pot(U, c(A1))), A2).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).
```

```
goal :- eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))).
% mode(derive(g)).
```

% Derivación simbólica. Estrategia *gp*

```
derive(add(U,V)) := let cpar(g(U), [A1 &= derive(U), A2 &= derive(V)])
                    in add(A1, A2).
derive(sub(U,V)) := let cpar(g(U), [A1 &= derive(U), A2 &= derive(V)])
                    in sub(A1, A2).
derive(mul(U,V)) := let cpar(g(U), [A1 &= derive(V), A2 &= derive(U)])
                    in add(mul(U, A1), mul(V, A2)).
derive(div(U,V)) := let cpar(g(U), [A1 &= derive(V), A2 &= derive(U)])
                    in div(sub(mul(U, A1), mul(V, A2)), mul(V, V)).
derive(pot(U,c(N))) := let cpar(g(N), [A1 &= N - 1, A2 &= derive(U)])
                       in mul(mul(c(N), pot(U, c(A1))), A2).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).

goal :- eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))).
```

% Derivación simbólica. Paralelización manual

```
derive(add(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
                    in add(A1, A2).
derive(sub(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
                    in sub(A1, A2).
derive(mul(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
                    in add(mul(U, A1), mul(V, A2)).
derive(div(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
                    in div(sub(mul(U, A1), mul(V, A2)), mul(V, V)).
derive(pot(U,c(N))) := let par([A1 &= N - 1, A2 &= derive(U)])
                       in mul(mul(c(N), pot(U, c(A1))), A2).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).

goal :- eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))).
% mode(derive(g)).
```

% Derivación simbólica. Estrategia *mp* con anotación de modos

```
derive(add(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
                    in add(A1, A2).
derive(sub(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
```

```
                    in sub(A1, A2).
derive(mul(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
                    in add(mul(U, A1), mul(V, A2)).
derive(div(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
                    in div(sub(mul(U, A1), mul(V, A2)), mul(V, V)).
derive(pot(U,c(N))) := let par([A1 &= N - 1, A2 &= derive(U)])
                       in mul(mul(c(N), pot(U, c(A1))), A2).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).

goal :- eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))).
% mode(derive(g)).
```

% Derivación simbólica. Estrategia *mp*

```
derive(add(U,V)) := let cpar(i(U,V), [A1 &= derive(U), A2 &= derive(V)])
                    in add(A1, A2).
derive(sub(U,V)) := let cpar(i(U,V), [A1 &= derive(U), A2 &= derive(V)])
                    in sub(A1, A2).
derive(mul(U,V)) := let cpar(i(U,V), [A1 &= derive(V), A2 &= derive(U)])
                    in add(mul(U, A1), mul(V, A2)).
derive(div(U,V)) := let cpar(i(U,V), [A1 &= derive(V), A2 &= derive(U)])
                    in div(sub(mul(U, A1), mul(V, A2)), mul(V, V)).
derive(pot(U,c(N))) := let cpar(i(N,U), [A1 &= N - 1, A2 &= derive(U)])
                       in mul(mul(c(N), pot(U, c(A1))), A2).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).

% goal :- tell(out), eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))), told.

goal :- eval(derive(X) &= div(c(1), x)), nl, write(X).

gen(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2))))), 0).
gen(add(U, V), N) :- N1 is N - 1, gen(U, N1), gen(V, N1).
```

% Derivación simbólica. Estrategia *up* con anotación de modos

```
derive(add(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
                    in add(A1, A2).
derive(sub(U,V)) := let par([A1 &= derive(U), A2 &= derive(V)])
                    in sub(A1, A2).
derive(mul(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
                    in add(mul(U, A1), mul(V, A2)).
derive(div(U,V)) := let par([A1 &= derive(V), A2 &= derive(U)])
```

```
                    in div(sub(mul(U, A1), mul(V, A2)), mul(V, V)).
derive(pot(U,c(N))) := let par([A1 &= N - 1, A2 &= derive(U)])
                       in mul(mul(c(N), pot(U, c(A1))), A2).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).

goal :- eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))).
% mode(derive(g)).
```

% Derivación simbólica. Estrategia *up*

```
derive(add(U,V)) := let seq([A1 &= derive(U), A2 &= derive(V)]) in add(A1, A2).
derive(sub(U,V)) := let seq([A1 &= derive(U), A2 &= derive(V)]) in sub(A1, A2).
derive(mul(U,V)) := let seq([A1 &= derive(V), A2 &= derive(U)])
                    in add(mul(U, A1), mul(V, A2)).
derive(div(U,V)) := let seq([A1 &= derive(V), A2 &= derive(U)])
                    in div(sub(mul(U, A1), mul(V, A2)), mul(V, V)).
derive(pot(U,c(N))) := let seq([A1 &= N - 1, A2 &= derive(U)])
                       in mul(mul(c(N), pot(U, c(A1))), A2).
derive(exp(U)) := mul(exp(U), derive(U)).
derive(log(U)) := div(derive(U), U).
derive(x) := c(1).
derive(c(X)) := c(0).

list_of_constants([x]).
list_of_constructors([add, sub, mul, div, pot, exp, log, c]).
list_of_functions([derive]).

goal :- eval(derive(log(div(mul(x, sub(x, c(1))), pot(exp(x), c(2)))))).
```

% Números de Fibonacci. Versión secuencial

```
fib(0) := 1.
fib(1) := 1.
fib(N) := fib(N-1) + fib(N-2).

list_of_constants([a,b,c,d]).
list_of_constructors([]).
list_of_functions([fib]).

goal :- eval(fib(5)).
```

% Números de Fibonacci. Estrategia *gp* con anotación de modos

```
fib(0) := 1.
fib(1) := 1.
fib(N) := let par([A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
```

```
list_of_functions([fib]).

goal :- eval(fib(3)).
% mode(fib(g)).
```

% Números de Fibonacci. Estrategia *gp*

```
fib(0) := 1.
fib(1) := 1.
fib(N) := let cpar(g(N), [A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([fib]).

goal :- eval(fib(5)).
```

% Números de Fibonacci. Paralelización manual

```
fib(0) := 1.
fib(1) := 1.
fib(N) := let par([A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([fib]).

goal :- eval(fib(5)).
```

% Números de Fibonacci. Estrategia *mp* con anotación de modos

```
fib(0) := 1.
fib(1) := 1.
fib(N) := let par([A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([fib]).

goal :- eval(fib(3)).
% mode(fib(g)).
```

% Números de Fibonacci. Estrategia *mp*

```
fib(0) := 1.
fib(1) := 1.
fib(N) := let cpar(i(N, N), [A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([fib]).

goal :- eval(fib(3)).
fib(0) := 1.
fib(1) := 1.
fib(N) := let par([A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.
```

```
list_of_constants([]).
list_of_constructors([]).
list_of_functions([fib]).
```

% Números de Fibonacci. Estrategia *up* con anotación de modos

```
fib(0) := 1.
fib(1) := 1.
fib(N) := let par([A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([fib]).

goal :- eval(fib(3)).
% mode(fib(g)).
```

% Números de Fibonacci. Estrategia *up*

```
fib(0) := 1.
fib(1) := 1.
fib(N) := let seq([A1 &= fib(N-1), A2 &= fib(N-2)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([fib]).

goal :- eval(fib(3)).
```

% Aplanamiento de listas. Estrategia *gp* con anotación de modos

```
flatten([H|T]) := let cpar(g(H), [A1 &= flatten(H), A2 &= flatten(T)])
                  in append(A1, A2).
flatten([]) := [].
flatten(X) := [X].

append([], L) := L.
append([X|Xs], L) := [X | append(Xs, L)].

list_of_constants([nil]).
list_of_constructors([cons]).
list_of_functions([flatten, append]).

goal :- gen(6, L), eval(flatten(L)).

gen(0, [1]).
gen(N, [L1,L2]) :- N1 is N - 1, gen(N1, L1),gen(N1, L2).
% mode(flatten(d)).
```

% Aplanamiento de listas. Estrategia *gp*

```
flatten([H|T]) := let cpar(g(H), [A1 &= flatten(H), A2 &= flatten(T)])
                  in append(A1, A2).
flatten([]) := [].
flatten(X) := [X].
```

```
append([], L) := L.
append([X|Xs], L) := [X | append(Xs, L)].

list_of_constants([nil]).
list_of_constructors([cons]).
list_of_functions([flatten, append]).

goal :- gen(6, L), eval(flatten(L)).

gen(0, [1]).
gen(N, [L1,L2]) :- N1 is N - 1, gen(N1, L1),gen(N1, L2).
```

% Aplanamiento de listas. Paralelización manual

```
flatten([H|T]) := let par([A1 &= flatten(H), A2 &= flatten(T)])
                  in append(A1, A2).
flatten([]) := [].
flatten(X) := [X].

append([], L) := L.
append([X|Xs], L) := [X | append(Xs, L)].

list_of_constants([nil]).
list_of_constructors([cons]).
list_of_functions([flatten, append]).

goal :- gen(6, L), eval(flatten(L)).

gen(0, [1]).
gen(N, [L1,L2]) :- N1 is N - 1, gen(N1, L1),gen(N1, L2).
```

% Aplanamiento de listas. Estrategia *mp* con anotación de modos

```
flatten([H|T]) := let cpar(i(H, T), [A1 &= flatten(H), A2 &= flatten(T)])
                  in append(A1, A2).
flatten([]) := [].
flatten(X) := [X].

append([], L) := L.
append([X|Xs], L) := [X | append(Xs, L)].

list_of_constants([nil]).
list_of_constructors([cons]).
list_of_functions([flatten, append]).

goal :- gen(6, L), eval(flatten(L)).

gen(0, [1]).
gen(N, [L1,L2]) :- N1 is N - 1, gen(N1, L1),gen(N1, L2).
% mode(flatten(d)).
```

% Aplanamiento de listas. Estrategia *mp*

```
flatten([H|T]) := let cpar(i(H, T), [A1 &= flatten(H), A2 &= flatten(T)])
                  in append(A1, A2).
```

```
flatten([]) := [].
flatten(X) := [X].

append([], L) := L.
append([X|Xs], L) := [X | append(Xs, L)].

list_of_constants([nil]).
list_of_constructors([cons]).
list_of_functions([flatten, append]).

goal :- gen(6, L), eval(flatten(L)).

gen(0, [1]).
gen(N, [L1,L2]) :- N1 is N - 1, gen(N1, L1),gen(N1, L2).
```

% Aplanamiento de listas. Estrategia *up* con anotación de modos

```
flatten([H|T]) := let seq([A1 &= flatten(H), A2 &= flatten(T)])
                  in append(A1, A2).
flatten([]) := [].
flatten(X) := [X].

append([], L) := L.
append([X|Xs], L) := [X | append(Xs, L)].

list_of_constants([nil]).
list_of_constructors([cons]).
list_of_functions([flatten, append]).

goal :- gen(6, L), eval(flatten(L)).

gen(0, [1]).
gen(N, [L1,L2]) :- N1 is N - 1, gen(N1, L1),gen(N1, L2).
% mode(flatten(d)).
```

% Aplanamiento de listas. Estrategia *up*

```
flatten([H|T]) := append(flatten(H), flatten(T)).
flatten([]) := [].
flatten(X) := [X].

append([], L) := L.
append([X|Xs], L) := [X | append(Xs, L)].

list_of_constants([nil]).
list_of_constructors([cons]).
list_of_functions([flatten, append]).

goal :- gen(6, L), eval(flatten(L)).

gen(0, [1]).
gen(N, [L1,L2]) :- N1 is N - 1, gen(N1, L1),gen(N1, L2).
```

% Inversión de listas. Versión secuencial

```
gen(N) := N &= 0 ==> []
```

```
                    # [N|gen(N-1)].

list_of_constructors([cons]).
list_of_constants([nil]).
list_of_functions([gen]).
```

% Inversión de listas. Paralelización manual

```
gen(N) := N &= 0 ==> []
                    # [N|gen(N-1)].

list_of_constructors([cons]).
list_of_constants([nil]).
list_of_functions([gen]).
```

% Función de Ackermann. Versión secuencial

```
ack(z, X1, s(X1)) := true.
ack(s(X1), z, X2) := ack(X1, s(z), X2) => true.
ack(s(X1), s(X2), X3) := ack(s(X1), X2, X4), ack(X1, X4, X3) => true.

list_of_constants([z]).
list_of_constructors([s]).
list_of_functions([ack]).

goal :- eval(ack(13, X, Y)).
```

% Torres de Hanoi. Versión secuencial

```
hanoi(N,S,H,D) := N &= 1 ==> leaf(10*S+D)
                           # node(hanoi(N-1,S,D,H), 10*S+D, hanoi(N-1,H,S,D)).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).
```

% Torres de Hanoi. Estrategia *gp* con anotación de modos

```
hanoi(N,S,H,D) := (N &= 1) ==> leaf(10*S+D)
                             # let par([A1 &= hanoi(N-1,S,D,H),
                                        A2 &= 10*S+D,
                                        A3 &= hanoi(N-1,H,S,D)])
                               in node(A1, A2, A3).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).

% mode(hanoi(g,g,g,g)).
```

% Torres de Hanoi. Estrategia *gp*

```
hanoi(N,S,H,D) := (N &= 1) ==> leaf(10*S+D)
                            # let if((g(S),g(D)),
                                     if((g(N),g(H)),
                                        par([A4 &= hanoi(N-1,S,D,H),
                                             A5 &= 10*S+D,
                                             A6 &= hanoi(N-1,H,S,D)]),
                                        par([seq([A4 &= hanoi(N-1,S,D,H),
                                             A6 &= hanoi(N-1,H,S,D)]),
                                             A5 &= 10*S+D])),
                                     seq([A4 &= hanoi(N-1,S,D,H),
                                          A5 &= 10*S+D,
                                          A6 &= hanoi(N-1,H,S,D)]))
                              in node(A4, A5, A6).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).
```

% Torres de Hanoi. Paralelización manual

```
hanoi(N,S,H,D) := (N &= 1) ==> leaf(10*S+D)
                            # let par([A1 &= hanoi(N-1,S,D,H),
                                       A2 &= 10*S+D,
                                       A3 &= hanoi(N-1,H,S,D)])
                              in node(A1, A2, A3).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).

% mode(hanoi(g,g,g,g)).
```

% Torres de Hanoi. Estrategia *mp* con anotación de modos

```
hanoi(N,S,H,D) := (N &= 1) ==> leaf(10*S+D)
                            # let par([A1 &= hanoi(N-1,S,D,H),
                                       A2 &= 10*S+D,
                                       A3 &= hanoi(N-1,H,S,D)])
                              in node(A1, A2, A3).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).

% mode(hanoi(g,g,g,g)).
```

% Torres de Hanoi. Estrategia *mp*

```
hanoi(N,S,H,D) := (N &= 1) ==> leaf(10*S+D)
                    # let if((g(S),g(D)),
```

```
                        if((g(N),g(H)),
                           par([A4 &= hanoi(N-1,S,D,H),
                                A5 &= 10*S+D,
                                A6 &= hanoi(N-1,H,S,D)]),
                           par([seq([A4 &= hanoi(N-1,S,D,H),
                                     A6 &= hanoi(N-1,H,S,D)]),
                                A5 &= 10*S+D])),
                        seq([A4 &= hanoi(N-1,S,D,H),
                             A5 &= 10*S+D,
                             A6 &= hanoi(N-1,H,S,D)]))
                  in node(A4, A5, A6).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).
```

% Torres de Hanoi. Estrategia *up* con anotación de modos

```
hanoi(N,S,H,D) := (N &= 1) ==> leaf(10*S+D)
                          # let par([A1 &= hanoi(N-1,S,D,H),
                                     A2 &= 10*S+D,
                                     A3 &= hanoi(N-1,H,S,D)])
                            in node(A1, A2, A3).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).

% mode(hanoi(g,g,g,g)).
```

% Torres de Hanoi. Estrategia *up*

```
hanoi(N,S,H,D) := (N &= 1) ==> leaf(10*S+D)
                  # let seq([A1 &= hanoi(N-1,S,D,H),
                             A2 &= 10*S+D,
                             A3 &= hanoi(N-1,H,S,D)])
                    in node(A1, A2, A3))).

list_of_constants([]).
list_of_constructors([node, leaf]).
list_of_functions([hanoi]).

goal :- eval(hanoi(3,1,1,1)).
```

% Problema de las reinas. Versión secuencial

```
queens(S,N,Pos) := N &= 0 ==> [Pos]
                       # iter(S,N,Pos,1).
iter(S,N,B,NPos) := NPos > S ==> []
                          # safe(B,1,NPos)
                            ==> append(queens(S,N-1,[NPos|B],
                                       iter(S,N,B,NPos+1)
```

```
                                    # iter(S,N,B,NPos+1).
safe([],D,Q) := true.
safe([H|T],D,Q) := (H=\=Q,Q-H=\=D,H-Q=\=D) ==> safe(T,D+1,Q)
                                           # false.
append([],L) := L.
append([H|T],L) := [H|append(T,L)].

list_of_constants([]).
list_of_constructors([]).
list_of_functions([queens,iter,append]).

goal :- eval(queens(11,11,[])).
```

% Problema de las reinas. Paralelización manual

```
queens(S,N,Pos) := N &= 0 ==> [Pos]
                          # iter(S,N,Pos,1).
iter(S,N,B,NPos) := NPos > S ==> []
                              # safe(B,1,NPos) ==> let par([A1 &= queens(S,N-1,[NPos|B],
                                                            A2 &= iter(S,N,B,NPos+1)])
                                                    in append(A1,A2)
                                                 # iter(S,N,B,NPos+1).
safe([],D,Q) := true.
safe([H|T],D,Q) := (H=\=Q,Q-H=\=D,H-Q=\=D) ==> safe(T,D+1,Q)
                                           # false.
append([],L) := L.
append([H|T],L) := [H|append(T,L)].

list_of_constants([]).
list_of_constructors([]).
list_of_functions([queens,iter,append]).

% mode(queens(g,g,g)).
goal :- eval(queens(11,11,[])).
```

% Ordenación *quicksort*. Versión secuencial

```
quicksort([]) := [].
quicksort([X|Ys]) := append(quicksort(lsplit(X,Ys)),quicksort(rsplit(X,Ys))).
lsplit(X,[Y,Z]) := Y <= X ==> [Y|lsplit(X,Z)]
                          # lsplit(X,Z).
lsplit(X,[]) := [].
rsplit(X,[Y,Z]) := Y <= X ==> rsplit(X,Z)
                          # [Y|rsplit(X,Z)].
rsplit(X,[]) := [].
append([],L) := L.
append([H|T],L) := [H|append(T,L)].

list_of_constants([]).
list_of_constructors([]).
list_of_functions([quicksort,lsplit,rsplit,append,generate7000]).

goal :- eval(quicksort(generate7000)).
```

% Ordenación *quicksort*. Paralelización manual

```
quicksort([]) := [].
quicksort([X|Ys]) := let par([A1 &= quicksort(lsplit(X,Ys),
                              A2 &= quicksort(rsplit(X,Ys))])
                     in append(A1,A2).
lsplit(X,[Y,Z]) := Y <= X ==> [Y|lsplit(X,Z)]
                           # lsplit(X,Z).
lsplit(X,[]) := [].
rsplit(X,[Y,Z]) := Y <= X ==> rsplit(X,Z)
                           # [Y|rsplit(X,Z)].
rsplit(X,[]) := [].
append([],L) := L.
append([H|T],L) := [H|append(T,L)].

list_of_constants([]).
list_of_constructors([]).
list_of_functions([quicksort,lsplit,rsplit,append,generate7000]).

% mode(quicksort(g)).
goal :- eval(quicksort(generate7000)).
```

% Subárboles. Versión secuencial

```
subtree(node(L, R), T) := subtree(L, T) ; x(R, T, L).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.

x(R, T, L) := subtree(R, T) ; T &= node(L, R).

list_of_constants([a,b,c,d]).
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).
```

% Subárboles. Estrategia *gp* con anotación de modos

```
subtree(node(L, R), T) := or((g(L), g(T)), subtree(L, T), x(R, T, L)).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.

x(R, T, L) := or((g(R), g(T)), subtree(R, T), T &= node(L, R)).

list_of_constants([a,b,c,d]).
```

```
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).

% mode(subtree(d)).
```

% Subárboles. Estrategia *gp*

```
subtree(node(L, R), T) := or((g(L), g(T)), subtree(L, T), x(R, T, L)).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.

x(R, T, L) := or((g(R), g(T)), subtree(R, T), T &= node(L, R)).

list_of_constants([a,b,c,d]).
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).
```

% Subárboles. Paralelización manual

```
subtree(node(L, R), T) := or(true, subtree(L, T), x(R, T, L)).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.

x(R, T, L) := or(true, subtree(R, T), T &= node(L, R)).

list_of_constants([a,b,c,d]).
```

```
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).
```

% Subárboles. Estrategia *mp* con anotación de modos

```
subtree(node(L, R), T) := or((g(L), g(T)), subtree(L, T), x(R, T, L)).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.

x(R, T, L) := or((g(R), g(T)), subtree(R, T), T &= node(L, R)).

list_of_constants([a,b,c,d]).
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).

% mode(subtree(d)).
```

% Subárboles. Estrategia *mp*

```
subtree(node(L, R), T) := or((g(L), g(T)), subtree(L, T), x(R, T, L)).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.

x(R, T, L) := or((g(R), g(T)), subtree(R, T), T &= node(L, R)).

list_of_constants([a,b,c,d]).
```

```
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).

gen(leaf(a), 0).
gen(node(T1, T2), N) :- N1 is N - 1, gen(T1, N1), gen(T2, N1).
```

% Subárboles. Estrategia *up* con anotación de modos

```
subtree(node(L, R), T) := or(false, subtree(L, T), x(R, T, L)).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.

x(R, T, L) := or(false, subtree(R, T), T &= node(L, R)).

list_of_constants([a,b,c,d]).
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).

% mode(subtree(d)).
```

% Subárboles. Estrategia *up*

```
subtree(node(L, R), T) := or(false, subtree(L, T), x(R, T, L)).
subtree(leaf(X), leaf(Y)) := X &= Y.
subtree(leaf(X), node(L, R)) := false.
```

```
x(R, T, L) := or(false, subtree(R, T), T &= node(L, R)).

list_of_constants([a,b,c,d]).
list_of_constructors([leaf, node]).
list_of_functions([subtree, x]).

goal :- eval(subtree(node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))),
                     node(
                          node(
                               leaf(a),
                               leaf(b)),
                          node(
                               leaf(c),
                               leaf(d))))).
```

% Producto vector-matriz. Versión secuencial

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) := [vmul(V, C)|vmatrix(V, Cs)].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) := V*C + vmul(Vs, Cs).

list_of_constants([]).
list_of_constructors([]).
list_of_functions([vmatrix, vmul]).

goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
```

% Producto vector-matriz. Estrategia *up* con anotación de modos

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) :=
        let par([A1 &= vmul(V, C), A2 &= vmatrix(V, Cs)])
        in [A1|A2].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) :=
        let par([A1 &= V*C, A2 &= vmul(Vs, Cs)])
        in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([vmatrix, vmul]).

% goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
goal :- eval(vmatrix([1,2], [[1,2], [1,2]])).

% mode(vmatrix(g,g)).
```

% Producto vector-matriz. Estrategia *gp*

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) :=
        let cpar((g(V),g(C)), [A1 &= vmul(V, C), A2 &= vmatrix(V, Cs)])
        in [A1|A2].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) :=
        let cpar((g(V),g(C)),
                 [A1 &= V*C, A2 &= vmul(Vs, Cs)])
        in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([vmatrix, vmul]).

% goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
goal :- eval(vmatrix([1,2], [[1,2], [1,2]])).
```

% Producto vector-matriz. Paralelización manual

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) :=
        let par([A1 &= vmul(V, C), A2 &= vmatrix(V, Cs)])
        in [A1|A2].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) :=
        let par([A1 &= V*C, A2 &= vmul(Vs, Cs)])
        in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([vmatrix, vmul]).

% goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
goal :- eval(vmatrix([1,2], [[1,2], [1,2]])).

% mode(vmatrix(g,g)).
```

% Producto vector-matriz. Estrategia *mp* con anotación de modos

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) :=
        let par([A1 &= vmul(V, C), A2 &= vmatrix(V, Cs)])
        in [A1|A2].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) :=
        let par([A1 &= V*C, A2 &= vmul(Vs, Cs)])
        in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([vmatrix, vmul]).

% goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
goal :- eval(vmatrix([1,2], [[1,2], [1,2]])).

% mode(vmatrix(g,g)).
```

% Producto vector-matriz. Estrategia *mp*

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) :=
        let cpar((g(V),i(C,Cs)), [A1 &= vmul(V, C), A2 &= vmatrix(V, Cs)])
        in [A1|A2].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) :=
        let cpar((i(V,Vs),i(V,Cs),i(C,Vs),i(C,Cs)),
                 [A1 &= V*C, A2 &= vmul(Vs, Cs)])
        in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([vmatrix, vmul]).

% goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
% goal :- eval(vmatrix([1,2], [[1,2], [1,2]])).
goal :- gen(V, M, 10), eval(vmatrix(V,M)).

gen(V, M, N) :- gen_v(V, N), gen_m(M, V, N).
gen_v([], 0).
gen_v([N|R], N) :- N1 is N - 1, gen_v(R, N1).
gen_m([], V, 0).
gen_m([V|R], V, N) :- N1 is N - 1, gen_m(R, V, N1).
```

% Producto vector-matriz. Estrategia *up* con anotación de modos

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) :=
        let par([A1 &= vmul(V, C), A2 &= vmatrix(V, Cs)])
        in [A1|A2].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) :=
        let par([A1 &= V*C, A2 &= vmul(Vs, Cs)])
        in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
list_of_functions([vmatrix, vmul]).

% goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
goal :- eval(vmatrix([1,2], [[1,2], [1,2]])).

% mode(vmatrix(g,g)).
```

% Producto vector-matriz. Estrategia *up*

```
vmatrix(V, []) := [].
vmatrix(V, [C|Cs]) := let seq([A1 &= vmul(V, C), A2 &= vmatrix(V, Cs)])
                      in [A1|A2].
vmul([], []) := 0.
vmul([V|Vs], [C|Cs]) := let seq([A1 &= V*C, A2 &= vmul(Vs, Cs)]) in A1 + A2.

list_of_constants([]).
list_of_constructors([]).
```

```
list_of_functions([vmatrix, vmul]).

% goal :- eval(vmatrix([1,2,3], [[1,2,3], [1,2,3], [1,2,3]])).
goal :- eval(vmatrix([1,2], [[1,2], [1,2]])).
```

Apéndice C

Especificación formal de $\mathcal{S}$

En este apéndice se presenta la especificación formal de la traducción $\mathcal{S}$, que tiene como objetivo reemplazar las reglas que contengan funciones no estrictas en reglas que contengan funciones no estrictas a lo sumo en la posición raíz. En el proceso de transformación aparecen nuevas funciones intermedias, cuyas reglas de definición contienen en la parte derecha las llamadas a funciones no estrictas, que se sustituyen en las reglas originales.

En lo que sigue:

- *BabelProgram* es el conjunto de definiciones de función de un programa.
- *Rule* es el conjunto de tuplas de reglas[1] de un programa.
- FS^n es el conjunto de símbolos de función aridad n para un determinado programa.
- FS es el conjunto de símbolos de función de cualquier aridad para un determinado programa. Es decir, $FS = \bigcup_i FS^i$.
- La constructora *let in* (en cursiva) forma parte del lenguaje de especificación, que no debe confundirse con la constructora let in del metalenguaje.
- *Exp* es el conjunto de expresiones asociado a un programa tal como se define en el apéndice A.
- $\mathcal{N}$ es el conjunto de números naturales.

[1] Es necesario dotar de un orden a las reglas de definición de función para mantener una semántica operacional no ambigua. Por otra parte, $BabelProgram \equiv \mathcal{P}(Rule)$.

progtrans:BabelProgram→BabelProgram
progtrans($\{f_i \mid 1 \leq i \leq n, f_i \in FS\}$) ::=
$\quad$ *functrans*(f_1) ∪ ... ∪ *functrans*(f_n)

functrans:Rule→ $\mathcal{P}$(Rule)
functrans(<f $t_{1,1} \ldots t_{1,n}$:= e_1, ..., f $t_{m,1} \ldots t_{m,n}$:= e_m >) ::=
$\quad$ *let* <te_1, F_1 > = *rootexprtrans*(e_1)
$\quad\quad$ ⋮
$\quad\quad$ <te_m, F_m > = *rootexprtrans*(e_m) *in*
$\quad$ {<f $t_{1,1} \ldots t_{1,n}$:= te_1 ,
$\quad\quad$ ⋮
$\quad\quad$ f $t_{m,1} \ldots t_{m,n}$:= te_m >} ∪
$\quad$ F_1 ∪ ... ∪ F_m

rootexprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
rootexprtrans(e_1, e_2) ::=
$\quad$ *let* <te_1, F_1 > = *exprtrans*(e_1)
$\quad\quad$ <te_2, F_2 > = *exprtrans*(e_2) *in*
$\quad$ <(te_1, te_2), F_1∪ F_2 >

rootexprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
rootexprtrans(e_1; e_2) ::=
$\quad$ *let* <te_1, F_1 > = *exprtrans*(e_1)
$\quad\quad$ <te_2, F_2 > = *exprtrans*(e_2) *in*
$\quad$ <(te_1; te_2), F_1∪ F_2 >

rootexprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
rootexprtrans(b → e) ::=
$\quad$ *let* <tb, F_1 > = *exprtrans*(b)
$\quad\quad$ <te, F_2 > = *exprtrans*(e) *in*
$\quad$ <tb → te, F_1∪ F_2 >

rootexprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
rootexprtrans(b → e_1 □ e_2) ::=

let <tb, F_1 > = *exprtrans*(b)
<te_1, F_2 > = *exprtrans*(e_1)
<te_2, F_3 > = *exprtrans*(e_2) *in*
<tb → te_1 □ te_2, F_1∪ F_2∪ F_3 >

rootexprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
rootexprtrans(f e_1 ... e_n) ::= %% f estricta
let <te_1, F_1 > = *exprtrans*(e_1)
⋮
<te_n, F_n > = *exprtrans*(e_n) *in*
<f te_1 ... te_n, F_1 ∪ ... ∪ F_n >

rootexprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
rootexprtrans(X) ::=
exprtrans(X)

rootexprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
rootexprtrans(c e_1 ... e_n) ::=
exprtrans(c e_1 ... e_n)

exprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
exprtrans(X) ::=
<X, ∅ >

exprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
exprtrans(c e_1 ... e_n) ::=
let <te_1, F_1 > = *exprtrans*(e_1)
⋮
<te_n, F_n > = *exprtrans*(e_n) *in*
<c te_1 ... te_n, F_1 ∪ ... ∪ F_n >

exprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
exprtrans((e_1 $t_{1,1}$ ... $t_{1,n}$), (e_2 $t_{2,1}$ ... $t_{2,m}$)) ::=

let f = *newfunction*(n+m)
<te$_1$, F$_1$ > = *exprtrans*(e$_1$ t$_{1,1}$ ... t$_{1,n}$)
<te$_2$, F$_2$ > = *exprtrans*(e$_2$ t$_{2,1}$ ... t$_{2,m}$) *in*
<(f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$),
{f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$:= te$_1$, te$_2$} ∪ F$_1$∪ F$_2$ >

exprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
exprtrans((e$_1$ t$_{1,1}$ ... t$_{1,n}$); (e$_2$ t$_{2,1}$ ... t$_{2,m}$)) ::=
let f = *newfunction*(n+m)
<te$_1$, F$_1$ > = *exprtrans*(e$_1$ t$_{1,1}$ ... t$_{1,n}$)
<te$_2$, F$_2$ > = *exprtrans*(e$_2$ t$_{2,1}$ ... t$_{2,m}$) *in*
<(f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$),
{f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$:= te$_1$; te$_2$} ∪ F$_1$∪ F$_2$ >

exprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
exprtrans((b t$_{1,1}$ ... t$_{1,n}$) → (e t$_{2,1}$ ... t$_{2,m}$)) ::=
let f = *newfunction*(n+m)
<tb, F$_1$ > = *exprtrans*(b t$_{1,1}$ ... t$_{1,n}$)
<te, F$_2$ > = *exprtrans*(e t$_{2,1}$ ... t$_{2,m}$) *in*
<(f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$),
{f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$:= tb → te} ∪ F$_1$∪ F$_2$ >

exprtrans:Exp→Exp ×$\mathcal{P}$(Rule)
exprtrans((b t$_{1,1}$ ... t$_{1,n}$) → (e$_1$ t$_{2,1}$ ... t$_{2,m}$) □ (e$_2$ t$_{3,1}$ ... t$_{3,m}$)) ::=
let f = *newfunction*(n+m+1)
<tb, F$_1$ > = *exprtrans*(b t$_{1,1}$ ... t$_{1,n}$)
<te$_1$, F$_2$ > = *exprtrans*(e$_1$ t$_{2,1}$ ... t$_{2,m}$)
<te$_2$, F$_3$ > = *exprtrans*(e$_2$ t$_{3,1}$ ... t$_{3,m}$) *in*
<(f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$, t$_{3,1}$, ...,t$_{3,l}$),
{f t$_{1,1}$, ...,t$_{1,n}$, t$_{2,1}$, ...,t$_{2,m}$, t$_{3,1}$, ...,t$_{3,l}$:= tb → te$_1$ □ te$_2$} ∪
F$_1$∪ F$_2$∪ F$_3$ >

newfunction:$\mathcal{N}$ → FS^{n+m}∪ f$_i$
newfunction(n) ::=
f$_i$ | f$_i$ nuevo símbolo de función

C.1 Esquema $\mathcal{S}^*$

Este esquema incluye la transformación de las predefinidas no estrictas secuenciales en predefinidas no estrictas paralelas. Sólo presentamos estos casos, que sustituyen a los correspondientes del esquema anterior[2].

rootexprtrans:Exp→Exp $\times\mathcal{P}$(Rule)
rootexprtrans(e_1, e_2) ::=
let <te_1, F_1 > = *exprtrans*(e_1)
<te_2, F_2 > = *exprtrans*(e_2) *in*
<(and $\chi(te_1, te_2)$ te_1 te_2), $F_1 \cup F_2$ >

rootexprtrans:Exp→Exp $\times\mathcal{P}$(Rule)
rootexprtrans(e_1; e_2) ::=
let <te_1, F_1 > = *exprtrans*(e_1)
<te_2, F_2 > = *exprtrans*(e_2) *in*
<(or $\chi(te_1, te_2)$ te_1 te_2), $F_1 \cup F_2$ >

exprtrans:Exp→Exp $\times\mathcal{P}$(Rule)
exprtrans((e_1 $t_{1,1} \ldots t_{1,n}$), (e_2 $t_{2,1} \ldots t_{2,m}$)) ::=
let f = *newfunction*(n+m)
<te_1, F_1 > = *exprtrans*(e_1 $t_{1,1} \ldots t_{1,n}$)
<te_2, F_2 > = *exprtrans*(e_2 $t_{2,1} \ldots t_{2,m}$) *in*
<(f $t_{1,1}, \ldots, t_{1,n}, t_{2,1}, \ldots, t_{2,m}$),
{f $t_{1,1}, \ldots, t_{1,n}, t_{2,1}, \ldots, t_{2,m}$:= (and $\chi(te_1, te_2)$ te_1 te_2)} $\cup$ $F_1 \cup F_2$ >

exprtrans:Exp→Exp $\times\mathcal{P}$(Rule)
exprtrans((e_1 $t_{1,1} \ldots t_{1,n}$); (e_2 $t_{2,1} \ldots t_{2,m}$)) ::=
let f = *newfunction*(n+m)
<te_1, F_1 > = *exprtrans*(e_1 $t_{1,1} \ldots t_{1,n}$)
<te_2, F_2 > = *exprtrans*(e_2 $t_{2,1} \ldots t_{2,m}$) *in*
<(f $t_{1,1}, \ldots, t_{1,n}, t_{2,1}, \ldots, t_{2,m}$),
{f $t_{1,1}, \ldots, t_{1,n}, t_{2,1}, \ldots, t_{2,m}$:= (or $\chi(te_1, te_2)$ te_1 te_2)} $\cup$ $F_1 \cup F_2$ >

[2]En lo que sigue, $\chi(e_1, e_2)$ corresponde a la conjunción de condiciones simples de independencia entre e_1 y e_2. Esta función se calcula con el procedimiento de trazado de arcos entre *PEUs* que se describe en la definición del *CDG*.

exprtrans:Exp→Exp $\times\mathcal{P}$(Rule)
exprtrans((b $t_{1,1} \ldots t_{1,n}$) → (e $t_{2,1} \ldots t_{2,m}$)) ::=
let f = *newfunction*(n+m)
<tb, F_1 > = *exprtrans*(b $t_{1,1} \ldots t_{1,n}$)
<te, F_2 > = *exprtrans*(e $t_{2,1} \ldots t_{2,m}$) *in*
<(f $t_{1,1}, \ldots, t_{1,n}, t_{2,1}, \ldots, t_{2,m}$),
{f $t_{1,1}, \ldots, t_{1,n}, t_{2,1}, \ldots, t_{2,m}$:= (gf χ(tb, te) tb te) $\cup$ $F_1 \cup F_2$ >

Apéndice D

Especificación de la máquina abstracta paralela

En este apéndice se presenta la especificación de las operaciones relacionadas con la explotación de paralelismo de la máquina abstracta paralela.

Se ha elegido un estilo de presentación imperativo en lugar del estilo funcional más habitual, que adolece de claridad de presentación debido a su expresión tan compacta, en la que es necesaria la transmisión de parámetros que no se modifican o consultan.

Comenzamos describiendo el lenguaje de especificación que utilizamos para presentar a continuación la especificación de las operaciones relacionadas con la explotación de paralelismo.

D.1 El lenguaje de especificación

El lenguaje de especificación incorpora los siguientes criterios:

- La indentación determina las líneas que pertenecen a un cuerpo determinado, en lugar de utilizar los delimitadores del tipo *begin-end*.
- El condicional se expresa en la forma usual (*if-then*).
- Se asume que las operaciones se ejecutan de forma secuencial, aunque también se aplica la ejecución paralela.
- Se utilizan las asignaciones secuenciales ($\leftarrow$) y comparaciones primitivas (e.g. $=$, $<$, etc).

- El acceso a componentes de estructuras o a elementos de arrays se denota respectivamente con "." y "[ ]".
- Los paréntesis agrupan argumentos en llamadas a funciones o procedimientos (e.g., *kill(pf, 1, pf.#slots)*) y se usan también para referenciar objetos (e.g., *(itm.fpf).#slots* referencia el contenido del campo *#slots* en el marcador de paralelismo anotado en el campo *fpf* del marcador de evaluación remota actual, el cual está referenciado por el registro *itm*).
- Se admiten efectos colaterales, i.e., sincronización y gestión de regiones críticas.
- La especificación reúne descripciones precisas con otras informales, que se distinguen respectivamente con los signos + y ∘. Con esta distinción nos centramos en los puntos más importantes y mejoramos la comprensión de los algoritmos mostrados. En particular, se omiten las especificaciones que no son relevantes para la presentación del comportamiento paralelo y que se pueden consultar en artículos o informes técnicos, o que se especifican de una forma mecánica sencilla (e.g., *continue with forward running mode*).
- Cada fragmento de la especificación tiene una descripción, rodeado por un rectángulo, y su nombre formal, en cursiva.

D.2 Operaciones relacionadas con la explotación de paralelismo

Utilizamos los siguientes identificadores para denotar los punteros a las estructuras de datos:

- *pf*: Marcador activo de paralelismo. Denota el marcador de paralelismo al que pertenece la expresión paralela actual. En general no se corresponde con el último marcador de paralelismo en la pila.
- *itm*: Marcador de evaluación remota en la cima de la pila.
- *ltm*: Marcador de evaluación local en la cima de la pila.
- *wm*: Marcador de reunión en la cima de la pila.
- *b*: Punto de elección en la cima de la pila.

Además de estos punteros, utilizamos los punteros usuales h (cima del grafo), t (cima del *trail*), e (último entorno en la pila) y d (cima de la pila de datos).

Asociamos a cada trabajador varios campos para el control de paralelismo:

- *#kill*: Contiene el número de marcadores de evaluación remota que se deben eliminar.

- *#ack*: Contiene el número de mensajes de reconocimiento en respuesta a notificaciones de descarte. Es necesario para la sincronización entre el trabajador local y el remoto.

Estos campos, junto a *lock* en cada marcador de paralelismo se acceden en exclusión mutua.

A continuación se presenta la especificación correspondiente a cada una de las operaciones relacionadas con la explotación de paralelismo.

D.2.1 Consecuencia calculada con éxito

En este apartado se presenta la especificación de la consecuencia de la evaluación de una expresión paralela calculada por un trabajador local o remoto, que se compone de los siguientes pasos:

- El procedimiento presentado comienza por la consulta del campo *kill* para comprobar si el cómputo actual tiene vigencia. Si hay alguna notificación de descarte, el procedimiento de descarte (*kill*) se activa. En caso contrario, el resultado se actualiza en el registro correspondiente, que se conoce por el campo *slot#* en el marcador de evaluación remota o marcador de evaluación local superior según sea el caso.

- Si hay algún punto de elección almacenado después del marcador correspodiente, entonces el puntero b es mayor que el puntero *ltm* para una expresión evaluada localmente, o mayor que el puntero *itm* para una expresión evaluada por un trabajador remoto. En este caso el campo *state* se actualiza con *alt*, porque se puede encontrar otra solución para el cómputo actual, en caso contrario se actualiza con *noalt*.

- El campo *definitory* se actualiza con *true* si el resultado calculado es un argumento definitorio y en caso contrario con *false*.

- Si se ha calculado un valor definitorio, las expresiones paralelas a su derecha se descartan y se espera su mensaje de reconocimiento.
- Si todas las expresiones a su izquierda se han calculado, el marcador de paralelismo activo se puede cerrar con un marcador de reunión y reanudar el cómputo hacia adelante.
- Si el cómputo actual no proporciona un resultado definitorio, pero es el último resultado por calcular a la izquierda de un resultado definitorio, el marcador de paralelismo también se cierra.
- Si el trabajador local ha calculado un resultado definitorio pero no todas las expresiones a la izquierda se han calculado, entonces debe permanecer a la espera.

No hay más tareas que se puedan ejecutar localmente debido a la estrategia de búsqueda de trabajo que se ha empleado (véase su descripción en un apartado posterior).

Especificación de respuesta computada por un trabajador local
locally computed successful outcome

```
+ Wait until one of the following holds
        + #kill > 0
        + the parcall frame can be accessed
+ If the first case holds then
        + turn to kill running mode
+ else
        + lock parcall frame
        + pf.slot[ltm.slot#].result ← (top of the data stack)
        + pf.slot[ltm.slot#].state ← if(b > ltm, alt, noalt)
        + If definitory(pf.function, (top of data stack))
             or pf.#slots = ltm.slot# then
             + pf.slot[ltm.slot#].definitory ← true
             + kill(pf, 1+ltm.slot#, pf.#slots)
             + wait
             + If all the tasks to the left of ltm.slot#
                  have state ∈ {alt, noalt} then
                  + push a wait marker
                  + pf.bm ← outside
```

+ ltm ← pf.ltm
+ pf ← pf.fpf
+ unlock parcall frame
∘ continue with forward running mode

+ else
+ unlock parcall frame
+ get idle

+ else
+ pf.slot[ltm.slot#].definitory ← false
+ If exists a definitory task t_d to the right of ltm.slot#
so that the tasks ltm.slot# + 1 ... t_d - 1
have been computed then
+ kill(pf, t_d + 1, pf.#slots)
+ wait
+ push a wait marker
+ pf.bm ← outside
+ ltm ← pf.ltm
+ pf ← pf.fpf
+ unlock parcall frame
∘ continue with forward running mode
+ else
+ look for local work

Especificación de respuesta computada por un trabajador remoto
remotely computed successful outcome

+ Wait until one of the following holds
+ #kill > 0
+ the parcall frame can be accessed
+ If the first case holds then
+ turn to kill running mode
+ else
+ lock father parcall frame itm.fpf
+ (itm.fpf).slot[itm.slot#].result ← (top of the data stack)
+ (itm.fpf).slot[itm.slot#].state ← if(b > itm, alt, noalt)
+ If definitory((itm.fpf).function, (top of data stack))
or (itm.fpf).#slots = itm.slot# then
+ (itm.fpf).slot[itm.slot#].definitory ← true

+ kill(itm.fpf, 1+itm.slot#, (itm.fpf).#slots)
+ wait
+ If all the tasks to the left of itm.slot#
  have state ∈ {alt, noalt} then
  + push (top of the data stack) onto father's data stack
  + push a wait marker onto father's run-time stack
  + (itm.fpf).bm ← outside
  + wait
  + (itm.procid).ltm ← (itm.fpf).ltm
  + (itm.procid).pf ← (itm.fpf).fpf
  + unlock father parcall frame itm.fpf
  ∘ continue with forward running mode at father processor
  + look for work
  + else
    + unlock father parcall frame itm.fpf
    + look for work

+ else

  + (itm.fpf).slot[itm.slot#].definitory ← false
  + If exists a definitory task t_d to the right of itm.slot#
    so that the tasks itm.slot# + 1 ... t_d - 1
    have been computed then
    + kill(itm.fpf, t_d + 1, (itm.fpf).#slots)
    + wait
    + push a wait marker onto father's run-time stack
    + (itm.fpf).bm ← outside
    + (itm.procid).ltm ← (itm.fpf).ltm
    + (itm.procid).pf ← ((itm.procid).pf).fpf
    + unlock father parcall frame itm.fpf
    ∘ continue with forward running mode
      at father processor
  + else
    + unlock father parcall frame itm.fpf
    + look for work

D.2.2 Fallo

En este apartado presentamos las especificaciones correspondientes a la operación de fallo, comenzando por el control que determina el estado en

el que sucede el fallo y continuando con la descripción de cada una de las operaciones posibles de fallo.

El estado de cómputo se determina realizando la siguiente serie ordenada de consultas:

- Si el último punto de elección está por debajo (es anterior) del último marcador de evaluación remota, significa que no hay más alternativas para el cómputo actual en la pila. Esto implica que el fallo se debe notificar al trabajador remoto correspondiente.
- Si el último punto de elección está por debajo del último marcador de evaluación local, entonces el fallo corresponde a una expresión calculada localmente por lo que el fallo afecta al trabajador local.
- Si el último punto de elección está por debajo del último marcador de reunión, entonces se buscan alternativas en el último marcador de paralelismo (no activo), que se encuentra en estado *outside*.

Especificación del control de fallo
failure

```
+ Case
        b < itm:
        + remote failure
        b < ltm:
        + local failure
        b < wm:
        + outside failure
        otherwise:
        + sequential failure
```

Los fallos local y remoto pueden ocurrir en los estados *inside* y *outside* del marcador de paralelismo, distinción de casos que se muestra en los siguientes fragmentos.

Especificación de fallo local
local failure

```
+ If (itm.procid).(itm.fpf).bm = inside then
```

+ remote inside failure
+ else
+ remote outside failure

Especificación de fallo remoto
remote failure

+ If procid.bm = inside then
+ local inside failure
+ else
+ local outside failure

Cuando el fallo ocurre en estado *inside*, después de la verificación usual de las notificaciones de descarte, el campo *state* en el correspondiente registro se actualiza con el valor *failed*. Después se descartan las restantes expresiones cuya evaluación haya comenzado y se espera a los mensajes de reconocimiento.

Si el fallo se refiere a un cómputo remoto, entonces se recupera el estado del trabajador actual del marcador de evaluación remota superior y se comienza la búsqueda de nuevo trabajo.

Si el fallo se refiere a un cómputo local, entonces se comienza el cómputo (secuencial) hacia atrás si se han calculado todas las expresiones a la izquierda con resultados no definitorios.

En caso contrario, el trabajador queda en estado de espera puesto que no es útil evaluar más expresiones locales.

Nótese que cuando llega una notificación de descarte al trabajador remoto que ha calculado fallo, los efectos de la operación de descarte son los mismos que los efectos del fallo calculado, por lo que lo único que resta es decrementar el número de notificaciones de descarte, decrementar el campo de reconocimiento *#ack* del trabajador padre y restaurar el puntero *itm*. La misma situación ocurre en el fallo remoto en estado *outside*.

Especificación de fallo local en modo *inside*
local inside failure

+ Wait until one of the following holds
+ #kill > 0
+ the parcall frame can be accessed

+ If the first case holds then
    + turn to kill running mode
+ else
    + lock parcall frame
    + pf.slot[ltm.slot#].state ← failed
    + If all the tasks to the left of ltm.slot# have been computed then
        + kill(pf, 1, pf.#slots)
        + wait
        + turn to backward running mode
    + else
        + kill(pf, ltm.slot# + 1, pf.#slots)
        + wait
        + unlock parcall frame
        + get idle

Especificación de fallo remoto en estado *inside*
remote inside failure

+ Wait until one of the following holds
    + #kill > 0
    + the parcall frame can be accessed
+ If the first case holds then
    + (itm.fpf).slot[itm.slot#].state ← killed
    + #kill ← #kill - 1
    + (itm.procid).#ack ← (itm.procid).#ack - 1
    + *restore state itm*
    + kill
+ else
    + lock father parcall frame (itm.fpf)
    + (itm.fpf).slot[itm.slot#].state ← failed
    + If all the tasks to the left of itm.slot# have been computed then
        + kill(itm.fpf, 1, (itm.fpf).#slots)
        + wait
        + turn father to backward running mode
    + else
        + (itm.fpf).slot[itm.slot#].state ← killed
        + kill(itm.fpf, itm.slot# + 1, (itm.fpf).#slots)
        + wait

+ unlock father parcall frame (itm.fpf)
+ restore state itm
+ look for work

Cuando el fallo afecta a un marcador de paralelismo en estado *outside* se pide una nueva alternativa a una expresión adecuada (calculada y anotada con alternativas pendientes). Esta expresión es la que se encuentra más a la derecha y con alternativas pendientes tal que no haya expresiones con resultado definitorio a su izquierda.

Si no se encuentra una expresión con tales características, se ejecuta el procedimiento de descarte sobre todas las expresiones. Después se espera hasta que se reciba el reconocimiento de los procedimientos de descarte. Si la expresión se ha calculado localmente, se efectúa la operación de fallo local (secuencial). En otro caso, se pide una nueva alternativa al trabajador remoto.

Cuando este trabajador proporciona el nuevo resultado pueden ocurrir tres casos:

- El resultado es fallo. En este caso, el procedimiento *local outside failure* se aplica recursivamente.
- El resultado es un valor definitorio. Esto corresponde al cierre del marcador de paralelismo y la continuación del cómputo hacia adelante.
- El resultado no es definitorio. El cómputo hacia adelante continúa en este caso.

En la búsqueda de una expresión adecuada para conseguir una nueva alternativa pueden ocurrir dos situaciones:

- No se encuentra ninguna expresión adecuada. Esto implica que todas las expresiones se deben descartar. Las tareas que se han calculado en trabajadores remotos se anotan en el campo *state* como *killed*. Al resto se le envía la notificación de descarte y se realiza una espera hasta el reconocimiento de la notificación.
- Se encuentra una expresión adecuada. En este caso se pide una alternativa a la expresión en cuestión y, si ha sido calculada por el mismo trabajador remoto, éste cambia a cómputo hacia atrás. En caso contrario continúa con búsqueda de trabajo.

Especificación de fallo local en modo *outside*
local outside failure

+ pf ← wm.pf
+ Wait until one of the following holds
    + #kill > 0
    + the parcall frame can be accessed
+ If the first case holds then
    + turn to kill running mode
+ else
    + lock parcall frame
    + t_d ← leftmost definitory task(pf, pf.#slots)
    + t_a ← rightmost alternative task(pf, t_d)
    + If $t_a = 0$ then
        + kill(pf, t_a + 1, pf.#slots)
        + wm ← pf.wm
        + wait
        + turn to backward running mode
    + else
        + kill(pf, t_a + 1, pf.#slots)
        + If t_a was locally computed then
            + wm ← pf.wm
            + wait
            + turn to backward running mode
        + else
            + pf.slot[t_a].state ← running
            + pf.slot[t_a].procid.redo ← on
            + wait
            + Wait until pf.slot[t_a].state ≠ running
            + If pf.slot[t_a].state = failed then
                + local outside failure
            + else
                + If definitory(pf.function, pf.slot[t_a].result) then
                    + push a wait marker
                    + pf.bm ← outside
                    + ltm ← pf.ltm
                    + pf ← pf.fpf
                    + unlock parcall frame
                    o continue with forward running mode

+ else
  ∘ continue with forward running mode

Especificación de fallo remoto en modo *outside*
remote outside failure

+ Wait until one of the following holds
  + #kill > 0
  + the parcall frame can be accessed
+ If the first case holds then
  + (itm.fpf).slot[itm.slot#].state ← killed
  + #kill ← #kill - 1
  + (itm.procid).#ack ← (itm.procid).#ack - 1
  + restore state itm
  + kill
+ else
  + lock father parcall frame (itm.fpf)
  + (itm.fpf).slot[itm.slot#].state ← failed
  + t_d ← leftmost definitory task(itm.fpf, itm.slot# - 1)
  + t_a ← rightmost alternative task(itm.fpf, t_d)
  + If $t_a = 0$ then
    + (itm.fpf).slot[itm.slot#].state ← killed
    + kill(itm.fpf, 1, (itm.fpf).#slots)
    + wait
    + unlock father parcall frame itm.fpf
    + turn father to backward running mode
    + restore state itm
    + look for work
  + else
    + (itm.fpf).slot[itm.slot#].state ← running
    + kill(itm.fpf, t_a + 1, (itm.fpf).#slots)
    + (itm.fpf).slot[t_a].procid.redo ← on
    ∘ wait
    + unlock father parcall frame itm.fpf
    + restore state itm
    + If (itm.fpf).slot[t_a].procid = local procid then
      + turn to backward running mode
    + else

+ look for work

Finalmente, se presenta una especificación típica de fallo secuencial, en la que se distinguen dos casos:

- No hay puntos de elección pendientes (el puntero *b* referencia a la dirección *bottom*). Por lo tanto, se produce fallo general.
- Hay al menos un punto de elección pendiente ($b > bottom$). Se procede a la recuperación del estado del trabajador en ese punto mediante la restauración del grafo, de la pila de datos, del *trail* y de la pila, desvinculando las variables anotadas en el *trail*.

Especificación del fallo secuencial
sequential failure

+ If b = bottom then
  - o notify about the general failure computed
+ else
  - + h ← b.h
  - + d ← b.d
  - + unwind(b.t)
  - + t ← b.t
  - + b ← b.b

D.2.3 Procedimiento de descarte (*kill*)

El primer fragmento de la especificación relacionado con el procedimiento de descarte es el que define el cambio al modo de descarte (*kill*). Se corresponde a la situación en que un trabajador remoto solicita el procedimiento de descarte a otro. Mediante este procedimiento, el trabajador cambia su estado a *kill* alterando el contenido del campo *rm* (*running mode* - modo de ejecución). Después se realizan tantos procedimientos de descarte sobre marcadores de evaluación remota como indica el campo *#kill*. Las variables anotadas en el *trail* se desvinculan y se espera el reconocimiento de los procedimientos de descarte remoto. Cuando han finalizado todos los procedimientos de descarte se envía notificación al trabajador que corresponde al último marcador de evaluación remota. Finalmente, se recupera el estado

del trabajador previo al marcador de evaluación remota y se empieza una nueva búsqueda de trabajo.

Nótese que el procedimiento de descarte es una acción paralela. De hecho, la notificación de *kill* a un conjunto de expresiones de un marcador de paralelismo representa un punto de activación de paralelismo. El punto de reunión se alcanza cuando se ha recibido la notificación de reconocimiento de cada una de las expresiones.

Especificación del procedimiento de descarte
kill

```
+ rm ← kill
+ While #kill > 0
        + While pf > itm
                + kill(pf, 1, pf.#slots)
                + pf ← pf.fpf
        + itm ← itm.itm
        + #kill ← #kill - 1
+ unwind(itm.t)
+ Repeat
+ until #ack = 0
+ (itm.procid).#ack ← (itm.procid).#ack - 1
+ restore state itm
+ rm ← looking
+ look for work
```

La notificación de descarte consiste en primer lugar en el envío de las notificaciones a trabajadores remotos, seguido de la ejecución del procedimiento de descarte en las expresiones locales, anticipando así el máximo trabajo posible.

Especificación del envío de notificaciones de descarte
kill(p, i, j)

```
+ kill remote(p, i, j)
+ kill local(p, i, j)
```

El siguiente fragmento muestra la especificación del envío de un grupo de notificaciones de descarte correspondientes a las expresiones de un marcador de paralelismo p evaluadas en trabajadores remotos, desde la expresión i-ésima a la j-ésima. Las expresiones susceptibles de recibir tal notificación (mediante el incremento del campo *#kill*) son las que ya han sido evaluadas ($state \in \{alt, noalt\}$) o que se están evaluando ($state = running$). Para mantener la sincronización necesaria, se incrementa el número de notificaciones de reconocimiento esperadas.

Especificación de envío de notificaciones a trabajadores remotos

kill remote(p, i, j)

```
+ For s = i to j
      + If p.slot[s].state ∈ {running, alt, noalt}
           and p.slot[s[.procid ≠ local procid then
           + (p.slot[s].procid).#kill ← (p.slot[s].procid).#kill + 1
           + #ack ← #ack + 1
```

A continuación se presenta el fragmento correspondiente a las notificaciones de descarte locales.

La primera notificación de descarte corresponde al marcador de evaluación local superior. En este caso, en lugar de restaurar el estado de cada punto de elección como en el caso secuencial, basta con restaurar el estado almacenado en el marcador de evaluación local. Se utiliza el flag *kill* para indicar cuándo se esta ejecutando una operación de *kill* local y asegurar que la operación afecta sólo al marcador de evaluación local superior. Con *kill local slots(p, i, j, l)* se produce el procedimiento de descarte sobre las expresiones i-ésima a j-ésima del marcador de paralelismo p, donde l representa la expresión más a la derecha que ha sido descartada.

Si no se descarta ninguna expresión local, no se restaura el estado puesto que las expresiones involucradas se han evaluado en trabajadores remotos.

Especificación del envío local de notificaciones

kill local(p, i, j)

```
+ If kill = on then
     + kill local slots(p, i, j, l)
+ else
```

+ kill ← on
+ kill local slots(p, i, j, l)
+ If l > 1 and l < top then
    + unwind((p.slot[l].xtm).t)
    + restore state ltm (p.slot[l].xtm)
    + pf ← p
+ wait
+ kill ← off

Finalmente, presentamos el fragmento *kill local slots*. Se ejecuta un procedimiento de descarte para cada hijo de la expresión correspondiente que haya creado al menos un marcador de paralelismo. Con este procedimiento se calcula la tarea más a la derecha que se ha descartado.

Especificación de *kill* de expresiones locales
kill local slots(p, i, j)

+ l ← top
+ For s = j downto i
    + If p.slot[s].state ∈ {running, alt, noalt} and p.slot[s].procid = local procid then
        + l ← s
        + If (p.slot[s].xtm).spf ≠ bottom then
            + kill((p.slot[s].xtm).spf, 1, (p.slot[s].xtm).spf.#slots)

D.2.4 Planificación de trabajo

En el procedimiento de planificación de trabajo distinguimos tres casos.

El primero corresponde a un trabajador local que busca trabajo. La estrategia en este caso es bastante simple: el procesador local busca de izquierda a derecha en su marcador de paralelismo activo expresiones preparadas (*ready*). Si no se encuentra trabajo, el trabajador espera hasta que terminen las expresiones remotas.

Especificación de la búsqueda local de trabajo
look for local work

+ t_d ← leftmost definitory task
+ For i = ltm.slot# + 1 to t_d - 1

+ If pf.slot[i].state = ready then
    + push a local task marker
    + ic ← pf.slot[i].code
    + pf.slot[i].xtm ← ltm
    + pf.slot[i].procid ← local process identifier
    + pf.slot[i].state ← running
    + unlock parcall frame
    ∘ turn to forward computation

El segundo caso se refiere a trabajadores que buscan trabajo y no han efectuado previamente ninguno. Son trabajadores con pilas vacías que empiezan la búsqueda en el trabajador seminal. El procedimiento se basa en buscar trabajo a partir del primer marcador de paralelismo que se haya creado en el trabajador seminal (este marcador de paralelismo está anotado en el campo *bopf*).

Finalmente, el tercer caso corresponde a un trabajador que ha evaluado alguna expresión remota. En este caso se busca trabajo a partir de la expresión a la derecha de la última expresión remota evaluada (se identifica a partir del último marcador de evaluación remota).

Estos dos últimos casos se encuentran descritos en el siguiente fragmento.

Especificación del procedimiento de búsqueda de trabajo
look for work

+ If e = bottom then
    + Repeat
        + If 1.bopf ≠ bottom then
            + look for work(1, bopf, 1)
+ else
    look for work(itm.procid, itm.fpf, itm.slot# + 1)

La especificación previa hace uso del paso general de búsqueda de trabajo que se presenta a continuación. El procedimiento consiste en buscar una expresión disponible de izquierda a derecha a partir de la expresión que se proporciona y hasta la expresión definitoria más a la izquierda si existe, o hasta la expresión más a la derecha en caso contrario. Si en la exploración del marcador de paralelismo se encuentra una expresión *ready*, se elige para su

evaluación. Si se encuentra una expresión *running* se explora recursivamente su marcador de paralelismo descendiente si lo hay.

El procedimiento comienza por la consulta del campo *#kill*. Si hay notificaciones de descarte se ejecuta el procedimiento de descarte, en caso contrario se bloquea el marcador de paralelismo y se comienza la búsqueda.

Especificación del procedimiento de búsqueda de trabajo

look for work(id, pf, slot) [1]

```
+ While #kill = 0 and pf ≠ bottom
        + Wait until one of the following holds
                + #kill > 0
                + the parcall frame can be accessed
        + If the first case holds then
                + turn to kill running mode
        + else
                + lock parcall frame pf
                + t_d ← leftmost definitory task for id, pf
                + For i = slot to t_d - 1
                        + If pf.slot[i].state = ready then
                            + push an input task marker for slot i
                            + e ← id.e
                            + ic ← pf.slot[i].code
                            + pf.slot[i].xtm ← itm
                            + pf.slot[i].procid ← local proc id
                            + pf.slot[i].state ← running
                            + unlock parcall frame pf
                            o turn to forward computation
                        + If pf.slot[i].state = running then
                            + vpf ← (slot[i].xtm).spf
                            + vid ← if(id ≠ local proc id,
                                        (slot[i].xtm).procid,
                                        local proc id)
                            + If vpf ≠ pf then
                                + unlock parcall frame pf
                                + look for work(vid, vpf, 1)
                + unlock parcall frame pf
```

[1] Este procedimiento, como el de la vida real, no asegura que la búsqueda tenga éxito.

D.2.5 Operaciones misceláneas

Para finalizar la especificación, ofrecemos varias operaciones que se han referenciado en los fragmentos anteriores.

La especificación de los cambios de estado del trabajador se presenta a continuación.

Especificación del cambio de estado a cómputo hacia adelante
turn to forward running mode

+ rm ← forward
+ narrowing

Especificación del cambio de estado a cómputo hacia atrás
turn to backward running mode

+ rm ← backward
+ failure

Especificación del cambio de estado a *kill*
turn to kill running mode

+ rm ← kill
+ kill

Especificación del cambio de estado a espera
turn to idle running mode

+ rm ← idle
+ idle

A continuación describimos la especificación correspondiente a los sucesos que pueden ocurrir en estado de inactividad.

Especificación del estado de inactividad
idle

```
+ Repeat
    + Case
          #kill > 0:
          + kill
          redo = on:
          + failure
          rm = forward:
          ∘ narrowing
          rm = backward:
          + failure
+ until true
```

La operación de espera que se especifica a continuación consiste en consultar el campo *#ack* hasta que alcanza el valor 0, que se corresponde a que ninguna notificación de reconocimiento queda pendiente.

Especificación de la espera por notificaciones de reconocimiento
wait

```
+ Repeat
+ until #ack = 0
```

La función que calcula si una posición de argumento es definitoria se especifica como sigue.

Especificación de la función definitoria
definitory(function, value)

```
+ Case
      function = conjunction:
      + return value = false
      function = disjunction:
      + return value = true
      otherwise:
      + return false
```

Finalmente, presentamos las especificaciones de las operaciones relacionadas con la recuperación del estado.

Especificación de la restauración de estado a partir de un *LTM*
restore state ltm(ltm)

+ t ← ltm.t
+ h ← ltm.h
+ b ← ltm.b
+ d ← ltm.d
+ e ← ltm.e
+ ltm ← ltm.ltm

Especificación de la restauración de estado a partir de un *ITM*
restore state itm(itm)

+ t ← itm.t
+ h ← itm.h
+ b ← itm.b
+ d ← itm.d
+ e ← itm.e
+ pf ← itm.pf
+ ltm ← itm.ltm
+ wm ← itm.wm
+ itm ← itm.itm

La especificación de la desvinculación de variables del *trail* se produce de la manera habitual como recoge la siguiente especificación:

Especificación de la desvinculación de variables del *trail*
unwind(a)

+ for i = a to t
+ (trail.i) ← (trail.i)
+ t ← a

Publicaciones

En este apartado se referencian las publicaciones que contienen las distintas partes de este trabajo y las que han contribuido a su desarrollo en otros aspectos, como los trabajos que hemos realizado en programación lógica, las arquitecturas de memoria distribuida y la explotación de la combinación de paralelismo conjuntivo y disyuntivo.

Los temas tratados en esta tesis aparecen publicados en conferencias nacionales e internacionales en los siguientes artículos:

- F. Sáenz and J. J. Ruz. Parallelism Identification in BABEL Functional Logic Programs. In *7th Conference on Logic Programming (GULP'92)*, Milan, Italy, June 1992.

- F. Sáenz and J. J. Ruz. Análisis de Paralelismo And en Programas BABEL. In *Primer Congreso Nacional de Programación Declarativa (PRODE'92)*, Madrid, Spain, September 1992.

- F. Sáenz and J. J. Ruz. Análisis de Independencia de Programas Babel. In *Segundo Congreso Nacional de Programación Declarativa (PRODE'93)*, Blanes, Spain, September 1993.

- F. Sáenz, W. Hans, J. J. Ruz, and S. Winkler. A Babel Parallel System: VHDL Modelling for Performance Measurement. In *1994 Joint Conference on Declarative Programming (GULP-PRODE'94)*, pages 238–252, Peñíscola, Spain, September 1994.

- F. Sáenz, W. Hans, J. J. Ruz, and S. Winkler. Shared Memory System for Babel: A VHDL Specification. In M.V. Hermenegildo and Jaan Penjam, editors, *Programming Language Implementation and Logic Programming (PLILP'94)*, volume 844 of *Lecture Notes in Computer Science*, pages 461–462, Madrid, Spain, September 1994. Springer-Verlag.

- F. Sáenz, J. J. Ruz, W. Hans, and S. Winkler. A Stack-based Machine for Parallel Execution of Babel Programs. In *First International Symposium on Parallel Symbolic Computation (PASCO'94)*, pages 336–345, Linz, Austria, September 1994.

Además, se han publicado informes técnicos en el Departamento de Informática y Automática de la UCM y en el Institut für Informatik de la Universidad de Aquisgrán, que se referencian a continuación.

- W. Hans, J.J. Ruz, F. Sáenz, and S. Winkler. A VHDL Specification of a Shared Memory System for Babel. Technical Report 93–19, Aachener Informatik Berichte, 1993.

- F. Sáenz and J. J. Ruz. Abstract Interpretation of a Functional Logic Language for Parallelism Identification. Technical report DIA 94/7, Departamento de Informática y Automática, Universidad Complutense de Madrid, June 1994.

- F. Sáenz, J. J. Ruz, W. Hans, and S. Winkler. Parallel System for a Functional-Logic Language. Technical report DIA 94/8, Departamento de Informática y Automática, Universidad Complutense de Madrid, July 1994.

- F. Sáenz and J. J. Ruz. Testing And-parallelism in Functional-Logic Languages. Technical report DIA 94/9, Departamento de Informática y Automática, Universidad Complutense de Madrid, July 1994.

- F. Sáenz and J. J. Ruz. Implementación eficiente de interpretación abstracta de programas lógicos basada en ejecución simbólica. Technical report DIA 94/14, Departamento de Informática y Automática, Universidad Complutense de Madrid, November 1994.

- F. Sáenz and J. J. Ruz. Parallelism Identification in Functional-Logic Programs. Technical report DIA 95/1, Departamento de Informática y Automática, Universidad Complutense de Madrid, January 1995.

Finalmente, se referencian algunos trabajos que hemos realizado relativos a la programación lógica y lógico-funcional que han contribuido al desarrollo de esta tesis.

- J.J. Ruz and F. Sáenz. Simulación de una máquina abstracta de programación lógica. *Revista de la Asociación Española de Informática y Automática*, XXII(1):55–63, January 1989.

- F. Sáenz. Simulación de una máquina secuencial de programación lógica. Research report, Departamento de Informática y Automática, Universidad Complutense de Madrid, May 1989.

- F. Sáenz. Compilador Prolog para la máquina abstracta de Warren. Research report, Departamento de Informática y Automática, Universidad Complutense de Madrid, June 1990.

- J.J. Ruz, L. Araujo, and F. Sáenz. And-Parallel Execution of Prolog on a Distributed Architecture. In *ISMM International Symposium, MIMI'90*, pages 128–131, Lugano, Switzerland, June 1990.

- F. Sáenz and J. J. Ruz. Máquina Abstracta de Ejecución Simbólica para Inferencia de Modos. In *Actas de las Jornadas de Programación Declarativa (PRODE'91)*, Torremolinos (Málaga), Spain, October 1991.

- J.J. Ruz, L. Araujo, and F. Sáenz. A Transputer-Based Architecture to Exploit the Restricted And-Parallelism of Prolog. In *MELECON'91*, pages 128–131, Ljubljana, (former) Yugoslavia, June 1990. IEEE.

- W. Hans, S. Winkler, and F. Sáenz. An Expression-Or-Parallel Implementation for a Functional Logic Language. In *1995 Joint Conference on Declarative Programming (GULP-PRODE'95)*, September 1995. To appear.

Referencias

[1] S. Abramsky and C. Hankin (editors). *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[2] H. Aït-Kaci. The WAM: A (Real) Tutorial. Technical Report 5, Paris Research Laboratory, January 1990.

[3] G. Amdahl. The validity of the single processor approach to achieving large scale capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–577, 1967.

[4] L. Araujo and J.J. Ruz. PDP: Prolog Distributed Processor for Independent AND/OR Parallel Execution of Prolog. In *International Conference on Logic Programming (ICLP'94)*, pages 142–156. The MIT Press, 1994.

[5] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[6] Arity Corporation, Domino Drive, Concord, MA 01742. *The Arity/Prolog Programming Language*, 1986.

[7] G. Båge and G. Lindstrom. Combinator Evaluation of Functional Programs With Logical Variables. *Journal of Lisp and Symbolic Computation*, 3(3):289–320, September 1990.

[8] G.P. Balboni, P.G. Bosco, C. Cecchi, R. Melen, C. Moiso, and G. Sofi. *Parallel Computers. Object-Oriented, Functional, Logic*, chapter 7: Implementation of a Parallel Logic + Functional Language. John Wiley & Sons, 1990.

[9] R. Barbuti, M. Bellia, G. Levi, and M. Martelli. LEAF: a Language which Integrates Logic, Equations and Functions. In *Logic Programming: Functions, Relations and Equations*, pages 201–238. Prentice Hall, 1986.

[10] H.P. Barendregt. *The Lambda Calculus Its Syntax and Semantics.* North Holland, 1981.

[11] J. Barklund. *Parallel Unification.* PhD thesis, Computer Science Department, Uppsala University, Uppsala, 1990.

[12] M. Bellia and G. Levi. The relation between Logic and Functional Languages: A Survey. *The Journal of Logic Programming*, 3:217–236, 1986.

[13] J. M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference.* Kluwer Academic Publisher, 1992.

[14] BIM & Katholieke Universiteit Leuven, Belgium. *Prolog by BIM - Reference Manual.*

[15] B. Bjerner and S. Holmstrom. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, pages 157–165, 1989.

[16] P. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi. Logic and Functional Programming on Distributed Memory Architectures. In *Seventh International Conference on Logic Programming (ICLP'90)*, pages 325–339, Jerusalem, Israel, June 1990.

[17] P.G. Bosco, C. Cecchi, and C. Moiso. An Extension of WAM for K-LEAF: a WAM-based Compilation of Conditional Narrowing. In *Sixth International Conference on Logic Programming*, pages 318–333. MIT Press, June 1989.

[18] P.G. Bosco and E. Giovannetti. IDEAL: An Ideal Deductive Applicative Language. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 89–94. IEEE Computer Society Press, 1986.

[19] P.G. Bosco, E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. A Complete Characterization of K-LEAF, a Logic Language with

Partial Functions. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 318–327. IEEE Computer Society Press, 1987.

[20] P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-Resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[21] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit, Leuven, Belgium, October 1987.

[22] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *The Journal of Logic Programming*, 10:91–124, 1991.

[23] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference on Logic Programming*, pages 192–204. MIT Press, August 1988.

[24] M. Bruynooghe, G. Janssens, A. Callebaut, and B. Damoen. Abstract Interpretation: Towards the Global Optimisation of Prolog Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 192–204. IEEE Computer Society Press, September 1987.

[25] G.L. Burn. *Languages for Parallel Architectures: Design, Semantics, Implementation Models*, chapter 3: Deriving a Parallel Evaluation Model for Lazy Functional Languages Using Abstract Interpretation. John Wiley & Sons, 1989.

[26] A. Calderwood and P. Szeridi. Scheduling Or-parallelism in Aurora - the Manchester scheduler. In *Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[27] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming (ICLP)*, 1993.

[28] J.H. Chang and A.M. Despain. Semi-intelligent Backtracking of Prolog based on Static Data Dependency Analysis. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 10–21. IEEE Computer Society Press, 1985.

[29] T. Chen, I.V. Ramakrishnan, and R. Ramesh. Multistage Indexing Algorithms for Speeding Prolog Execution. In *Joint International Conference and Symposium on Logic Programming*, 1992.

[30] P.H. Cheong. Compiling Lazy Narrowing into Prolog. Technical Report 25, LIENS, 1990.

[31] V. Cholvi and J.M. Bernabéu. Estudio del modelo de memoria compartida distribuida. *Revista de la Asociación Española de Informática y Automática*, XXVII(4):3–14, December 1994.

[32] A. Church. *The calculi of lambda conversion.* Princeton University Press, 1941.

[33] A. Ciepielewski, S. Haridi, and B. Hausman. Initial Evaluation of a Virtual Machine for Or-Parallel Execution of Logic Programs. In *IFIP-TC10 Working Conference on Fifth Generation Computer Architecture*, 1985.

[34] K.L. Clark and S. Gregory. A first-order theory of data and programs. *Information Processing*, pages 939–944, 1977.

[35] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[36] C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation of Concurrent Logic Languages. In *1990 North American Conference on Logic Programming*, October 1990.

[37] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. Un Systeme de Communication Homme-Machine en Francais. Technical report, Groupe de Recherce en Intelligence Artificielle, Université d'Aix-Marseille, 1973.

[38] J. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publisher, 1987.

[39] M.M. Corsini and G. Filè. A Complete Framework for the Abstract Interpretation of Logic Programs: Theory and Application. Technical Report 3/88, Mathematics Department of Padova University, 1988.

[40] A. Cortesi and G. Filè. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In P. Hudak and N.D. Jones, editors, *Proceedings ACM-PEPM, SIGPLAN Notices 26 (9)*, 1991.

[41] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoint. *Conf. Rec. 4th ACM Symposium on Prin. of Programming Languages*, pages 238–252, 1977.

[42] H.B. Curry and R. Feys. *Combinatory logic*, volume 1. North Holland, 1958.

[43] J. Darlington, A. J. Field, and H. Pull. The Unification of Functional and Logic Languages. In *Logic Programming: Functions, Relations and Equations*, pages 37–70. Prentice Hall, 1986.

[44] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. In *Proceedings of Logic Programming Symposium*, Salt Lake City, September 1986.

[45] S. K. Debray and D. S. Warren. Detection and Optimization of Functional Computations in Prolog. In *Third International Conference on Logic Programming*, pages 490–504, London, July 1986.

[46] S. K. Debray and D. S. Warren. Static Analysis of Parallel Logic Programs. In *Fifth International Conference on Logic Programming*, Salt Lake City, August 1988.

[47] S.K. Debray and Nai-Wei Lin. Automatic Complexity Analysis of Logic Programs. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 599–613. The MIT Press, 1991.

[48] S.K. Debray, Nai-Wei Lin, and Manuel Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conference on Programming Language Design and Implementation.* ACM Press, 1990.

[49] S.K. Debray and N.W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[50] D. DeGroot. Restricted And-parallelism. *Proceedings of the International Conference of Fifth Generation Computer Systems, ICOT*, 1984.

[51] D. DeGroot and G. Lindstrom. *Logic Programming: Functions, Relations and Equations.* Prentice Hall, 1986.

[52] Department of Artificial Intelligence, University of Edinburgh. *User's Guide to DECsystem-10 Prolog*, 1978.

[53] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

[54] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *4th. IEEE Symposium on Logic Programming*, pages 264–272, September 1987.

[55] T. Disz and E. Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Proceedings of the Fourth Symposium on Logic Programming*, September 1987.

[56] T. DongXing, E. Pontelli, G. Gupta, and M. Carro. Last Parallel Call Optimization and Fast Backtracking in And-parallel Logic Programming Systems. In *ICLP'94 Workshop on Parallel and Data-Parallel Execution of Declarative Languages*, S. Margherita, Italy, June 1994.

[57] D.Wolz. Design of a Compiler for Lazy Pattern Driven Narrowing. In *Recent Trends in Data Type Specification*, volume 534 of *Lecture Notes in Computer Science*, pages 362–379. Springer-Verlag, 1990.

[58] M. J. Fernández, M. Carro, and M. V. Hermenegildo. IDRA (IDeal Resource Allocation): A Tool for Computing Ideal Speedups. In *ICLP WS on Parallel and Data Parallel Execution of Logic Programs*, Uppsala University, CS Department, Box 311, S–751 05 Uppsala, Sweden, June 1994.

[59] A. J. Field and P.G. Harrison. *Functional Programming.* Addison Wesley, 1988.

[60] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP Programs Using Abstract Interpretation. *New Generation Computing*, 6:159–186, 1988.

[61] C.L. Hankin, G.L. Burn, and S.L. Peyton Jones. A Safe Approach to Parallel Combinator Reduction. In B. Robinet and R. Wilhelm, editors, *ESOP'86*, volume 413 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, March 1986.

[62] W. Hans and S. Winkler. Abstract Interpretation of Functional Logic Languages. Technical Report Aachener Informatik Berichte 92–43, Lehrstuhl für Informatik II, 1992.

[63] W. Hans, S. Winkler, and F. Sáenz. An Expression-Or-Parallel Implementation for a Functional Logic Language. In *1995 Joint Conference on Declarative Programming (GULP-PRODE'95)*, September 1995. To appear.

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

[65] R. Harper, D.B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, 1986.

[66] P. Hartel, Hugh Glaser, and John Wild. Compilation of functional languages using flow graph analysis. Technical Report CSTR 91-03, Dept. of Electronics and Computer Science, University of Southampton, 1991.

[67] P. Henderson. *Functional Programming: Applications and its Implementation.* Prentice Hall, 1980.

[68] M. V. Hermenegildo and K. Greene. &-Prolog and its Performance. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[69] M.V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[70] M.V. Hermenegildo and M. Carro. Experimenting with Independent And-Parallel Prolog using Standard Prolog. In *Actas de las Jornadas de Programación Declarativa (PRODE'91)*, Torremolinos (Málaga), Spain, October 1991.

[71] M.V. Hermenegildo and R.I. Nasr. Efficient Management of Backtracking in AND-Parallelism. In *3rd. International Conference on Logic Programming*, 1986.

[72] M.V. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[73] M.V. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 237–252. The MIT Press, 1990.

[74] M.V. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *The Journal of Logic Programming*, 1995.

[75] A. Herranz-Nieva and J. Mariño-Carballo. Specialized Compilation of Lazy Functional Logic Programs. In *Segundo Congreso Nacional de Programación Declarativa (PRODE'93)*, Blanes, Spain, September 1993.

[76] T. Hickey and J. Cohen. Automating Program Analysis. *Journal of the ACM*, 35(1):185–220, January 1988.

[77] T. Hickey and Sh. Mudambi. Global Compilation of Prolog. *The Journal of Logic Programming*, 7, 1989.

[78] L. Hirschman, W.C. Hopkins, and R.C. Smith. Or-Parallel Speed-Up in Natural Language Processing: A Case Study. In *Fifth International Conference and Symposium on Logic Programming*. The MIT Press, 1988.

[79] G. Hogen and R. Loogen. A New Stack Technique for the Management of Runtime Structures in Distributed Environments. Technical Report 93–3, Aachener Informatik Berichte, 1993.

[80] G. Hoogen and R. Loogen. PASTEL: A Parallel Stack-based Implementation of Eager Functional Programs with Lazy Data Structures. In *4th Workshop on the Parallel Implementation of Functional Languages*, Aachen, Germany, September 1992.

[81] P. Hudak. Conception, Evolution and Application of Functional Programming Languages. *ACM Computer Surveys*, 21(3):359–411, September 1989.

[82] P. Hudak and P. Wadler. Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, 27(5), 1992.

[83] G. Huet and J.J. Levy. Computations in Nonambiguous Linear Rewriting Systems. Technical Report 359, INRIA, Le Chesnay, France, 1979.

[84] G. Huet and D.C. Open. *Equations and Rewrite Rules: A Survey*, chapter Formal Language Theory: Perspective and Open Problems, pages 349–405. Academic Press, 1980.

[85] Institute of Electrical and Electronic Engineers, Inc. *IEEE Standard VHDL Language Reference Manual*, March 1988.

[86] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted AND Parallelism. *European Symposium on Programming*, pages 284–297, 1988.

[87] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In Jean-Louis Lassez, editor, *International Conference on Logic Programming (ICLP'87)*. The MIT Press, 1987.

[88] N. Jones and H. Søndergaard. A Semantics-based Framework for the Abstract Interpretation of Prolog. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 6, pages 124–142. Ellis Horwood ltd, 1987.

[89] P. Kacsuk. OR-parallel Prolog on Distributed Memory Systems. In *PARLE*, volume 817 of *Lecture Notes in Computer Science*, pages 453–463. Springer-Verlag, 1994.

[90] G. Kildall. A Unified Approach to Global Program Optimization. In *Symposium on Principles of Programming Languages*, pages 194–206. ACM, January 1973.

[91] R.A. Kowalski. Predicate Logic as a Programming Language. In *IFIP 74*, pages 569–574, 1974.

[92] H. Kuchen and W. Hans. An AND-Parallel Implementation of the Functional Logic Language BABEL. In *Workshop on the Integration of Functional and Logic Programming*, Granada, Spain, September 1990.

[93] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez Artalejo. Graph-based Implementation of a Functional Logic Language. In *ESOP*, volume 432 of *Lecture Notes in Computer Science*, pages 271–290. Springer-Verlag, 1990.

[94] H. Kuchen, R. Loogen, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Graph Narrowing to Implement a Functional Logic Language. Technical Report DIA 92/4, Departamento de Informática y Automática, UCM, 28040 Madrid, Spain, November 1992.

[95] H. Kuchen, J.J. Moreno-Navarro, and M.V. Hermenegildo. Independent AND-Parallel Implementation of Narrowing. In *Programming Language Implementation and Logic Programming: Proceedings of the 4th International Symposium, PLILP '92*, pages 24–38, Leuven, Belgium, 1992.

[96] P.J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6:308–327, 1964.

[97] J.-L. Lassez, M.J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625, Los Altos, 1988. Morgan Kaufmann Publishers Inc.

[98] G. Lindstrom. Functional Programming and the Logical Variable. In *Symposium on Principles of Programming Languages*, pages 266–280, New Orleans, January 1985. ACM.

[99] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[100] R. Loogen. Design of a parallel programmable graph reduction machine with distributed memory. Technical Report Aachener Informatik Berichte 87–11, Lehrstuhl für Informatik II, 1987.

[101] R. Loogen. Stack-based Implementation of Narrowing. In *CCPSD, Tapsoft*, volume 494 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[102] R. Loogen and Stephan Winkler. Dynamic Detection of Determinism in Functional Logic Languages. In *1991 Symposium on Programming Language Implementation and Logic Programming (PLILP)*, volume 528 of *Lecture Notes in Computer Science*, pages 335–346. Springer-Verlag, 1991.

[103] P. López and M.V. Hermenegildo. Towards Dynamic Term Size Computation via Program Transformation. In *Segundo Congreso Nacional de Programación Declarativa (PRODE'93)*, pages 73–93, Blanes, Spain, September 1993.

[104] P. López, M.V. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *First International Symposium on Parallel Symbolic Computation (PASCO'94)*, Linz, Austria, September 1994.

[105] H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 205–214. IEEE Computer Society Press, 1987.

[106] A. Marien and B. Demoen. On the Management of Choicepoint and Environment Frames in the WAM. In *North American Conference on Logic Programming*. MIT Press, October 1989.

[107] K. Marriot and H. Søndergaard. Bottom-up Dataflow Analysis of Normal Logic Programs. In *Fifth International Conference on Logic Programming*. Mit Press, August 1988.

[108] K. Marriot and H. Søndergaard. Semantics-based Dataflow Analysis of Logic Programs. *Information Processing*, pages 601–606, April 1989.

[109] J. McCarthy. Recursive Functions and Symbolic Expressions. *Communications of the ACM*, 3:184–195, 1988.

[110] C. S. Mellish. Abstract Interpretation of Prolog Programs. In *Conf. on Logic Programming, no. 225, in Lecture Notes in Computer Science*, pages 463–475. Springer-Verlag, July 1986.

[111] C.S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Department of Artificial Intelligence, University of Edimburgh, 1981.

[112] C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *The Journal of Logic Programming*, 1:43–66, 1985.

[113] D. Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[114] D. Miller and G. Nadathur. Higher Order Programming. In *1986 International Conference on Logic Programming (ICLP)*, volume 225 of *Lecture Notes in Computer Science*, pages 448–462. Springer-Verlag, 1986.

[115] J.J. Moreno. *Diseño, Semántica, e Implementación de BABEL: Un lenguaje que integra la Programación Funcional y Lógica.* PhD thesis, Facultad de Informática, UPM, Madrid, July 1989.

[116] J.J. Moreno and M. Rodríguez-Artalejo. BABEL: a Functional and Logic Programming Language Based on a Constructor Discipline and Narrowing. In *Conference on Algebraic and Logic Programming*, volume 343 of *Lecture Notes in Computer Science*, pages 223–232. Springer-Verlag, 1988.

[117] J.J. Moreno-Navarro. Expressivity of Functional-logic Languages and their Implementation. Tutorials 1994 Joint Conference GULP-PRODE'94, Universidad Politécnica de Valencia, Spain, September 1994.

[118] J.J. Moreno-Navarro, H. Kuchen, and J. Mariño-Carballo. Efficient Lazy Narrowing using Demandedness Analysis. In *1993 Symposium on Programming Language Implementation and Logic Programming (PLILP'93)*, volume 714 of *Lecture Notes in Computer Science*, pages 167–183. Springer-Verlag, 1993.

[119] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: the Language Babel. *The Journal of Logic Programming*, 12:191–223, 1992.

[120] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of Shared Data Structures for Compile-Time Garbage Collection in Logic Programs. In *Seventh International Conference on Logic Programming (ICLP-90)*, Jerusalem, Israel, June 1990.

[121] K. Muthukumar and M.V. Hermenegildo. Determination of Variable Dependence Information Through Abstract Interpretation. In *North American Conference on Logic Programming*. MIT Press, October 1989.

[122] K. Muthukumar and M.V. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. *Proceedings of the 1990 International Conference on Logic Programming*, 1990.

[123] K. Muthukumar and M.V. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *The Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

[124] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edimburgh, 1981.

[125] A. Mycroft and F. Nielson. The theory and practice of transforming call-by-need into call-by-value. In *Proceedings of the 4th International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*. Springer–Verlag, 1980.

[126] A. Mycroft and F. Nielson. Strong Abstract Interpretation using Power Domains. In *ICALP'83*, volume 154 of *Lecture Notes in Computer Science*. Springer–Verlag, 1983.

[127] U. Nilsson. *Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs*. PhD thesis, Linköping University, S-581 83 Linköping, Sweden, 1992.

[128] M.J. O'Donnell. *Equational Logic as a Programming Language*. The MIT Press, 1985.

[129] R.A. O'Keefe. Finite Fixed Point Problems. In *Fourth International Conference on Logic Programming*, volume 2, pages 729–743. MIT Press, May 1987.

[130] A. Pozo-Prieto and J.J. Moreno-Navarro. Independent Subexpressions Parallelism with Delayed Synchronization for Functional Logic Languages. In *First International Symposium on Parallel Symbolic Computation (PASCO'94)*, Linz, Austria, September 1994.

[131] F.A. Rabhi and G.A. Manson. Using Complexity Functions to Control Parallelism in Functional Programs. Technical Report CS-90-1, Department of Computer Science, University of Sheffield, England, 1990.

[132] M. Ratcliffe and J. Syre. The Static Analysis of Logic Programs. Technical report ca–11, European Computer Industry Research Centre, 1985.

[133] U.S. Reddy. Narrowing as the Operational Semantics of Functional Programs. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 138–151. IEEE Computer Society Press, July 1985.

[134] G. Robinson and L. Wos. Paramodulation and Theorem Proving in First-Order Theories with Equality. *Machine Intelligence*, pages 135–150, 1969.

[135] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 1(12):23–41, 1965.

[136] M. Rosendhal. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156, 1989.

[137] P. L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California, Berkeley, 1990.

[138] P. Van Roy, B. Demoen, and Y.D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and Determinism. In *TAPSOFT'87*, volume 250 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, September 1987.

[139] F. Sáenz and J. J. Ruz. Máquina Abstracta de Ejecución Simbólica para Inferencia de Modos. In *Actas de las Jornadas de Programación Declarativa (PRODE'91)*, Torremolinos (Málaga), Spain, October 1991.

[140] M. Sassin. Design of an Abstract Shared Memory Machine for AND-Parallel BABEL with Dependency Information. In *Workshop on the Integration of Functional and Logic Programming*, Granada, Spain, September 1990.

[141] A.V.S. Sastry and L.M. Patnaik. OR-Parallel Evaluation of Logic Programs on a Multi-Ring Dataflow Machine. *New Generation Computing*, 10:23–53, 1991.

[142] D.S. Scott. Domains for Denotational Semantics. In *Proceedings ICALP'82*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613. Springer–Verlag, 1982.

[143] Sequent Computer System, Inc. *BALANCE 8000 Guide to Parallel Programming*, 1985.

[144] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computer Surveys*, 21:413–510, September 1989.

[145] K. Shen and M. V. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *1991 International Logic Programming Symposium*. MIT Press, October 1991.

[146] K. Shen and D.H.D. Warren. A Simulation Study of the Argonne Model for Or-Parallel Execution of Prolog. In *Proceedings of the Fourth Symposium on Logic Programming*, September 1987.

[147] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, 4(21):622–642, 1965.

[148] H. Søndergaard. An Application of Abstract Interpretation of Logic Programs: Occur Check Reduction. In *Fourth International Conference on Logic Programming*, pages 769–787. MIT Press, May 1987.

[149] L. Sterling and E. Shapiro. *The Art of Prolog*. the MIT Press, 1986.

[150] C. W. Strevens. The Transputer. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 292–302, Boston, Massachussetts, 1985. IEEE Computer Society.

[151] J. Tan and I-P. Ling. Compiling Dataflow Analysis of Logic Programs. In *ACM SIGPLAN'92*, 1992.

[152] A. Taylor. Removal of Dereferencing and Trailing in Prolog Compilation. In *Sixth International Conference on Logic Programming*, pages 48–60. MIT Press, June 1989.

[153] E. Tick. Compile-Time Granularity Analysis for Parallel Logic Programming Languages. *New Generation Computing*, 7:325–337, 1990.

[154] D. Turner. Miranda: a Non-strict Functional Language with Polymorphic Types. In *Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, Nancy, France, 1985. Springer-Verlag.

[155] M.H. van Emden and R. Kowalski. Logic Programming with Equations. *The Journal of Logic Programming*, 4:256–288, 1987.

[156] P. Vataja and E. Ukkonen. Finding Temporary Terms in Prolog Programs. *Proceedings of the International Conference of Fifth Generation Computer Systems, ICOT*, pages 275–282, 1984.

[157] A.R. Verden. *And-Parallel Implementation of Prolog on Distributed Memory Machines.* PhD thesis, Dept. of Electronics and Computer Science, University of Southampton, Southampton S09 5NH, August 1991.

[158] J. von Neumann. First draft of a report on the EDVAC. Technical report, University of Pennsylvania, 1945.
The report that got von Neumann's name associated with the serial, stored-program, general purpose, digital architecture upon which 99.99% of all computers today are based.

[159] P. Wadler. Strictness Analysis aids Time Analysis. In *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 1989.

[160] A. Waern. An Implementation Technique for the Abstract Interpretation of Prolog. In *Fifth International Conference on Logic Programming*, pages 700–710, August 1988.

[161] D. H. D. Warren. An Abstract Prolog Instruction set. Technical Note 309, SRI International, 1983.

[162] D. H. D. Warren. The SRI Model for Or-parallel Execution of Prolog. In *4th. IEEE Symposium on Logic Programming*, pages 92–102, September 1987.

[163] R. Warren, M.V. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International*

Conference and Symposium on Logic Programming, pages 684–699. MIT Press, August 1988.

[164] H. Westpahl, P. Robert, J. Chassin, and J. Syre. The PEPsys Model: Combining Backtracking, AND- and OR-parallelism. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 436–448. IEEE Computer Society Press, 1987.