MULTIPATH:
Un sistema para la programación lógica

Tesis Doctoral

Septiembre 1996

Presentada por:

Jordi Tubella Murgadas

Dirigida por:

Antonio González Colás

Universitat Politècnica de Catalunya

Departament d'Arquitectura de Computadors

Tesis doctoral presentada por Jordi Tubella Murgadas
para obtener el grado de Doctor en Informática
por la Universitat Politècnica de Catalunya

MULTIPATH:
Un sistema para la programación lógica

Jordi Tubella Murgadas

Tribunal

Para mi esposa, Marta,
y mis padres, José y Josefina,
por el ánimo y apoyo
que me han proporcionado
para la realización de este trabajo

AGRADECIMIENTOS

Quisiera agradecer a mi director de tesis Antonio González por todo el apoyo y la orientación que me ha prestado durante el desarrollo de este trabajo. Para él van mis mejores agradecimientos.

También destaco a todas aquellas personas que me han ayudado en la confección del software necesario para llevar adelante las ideas presentadas en esta tesis. En especial, agradezco a Eduard Elias todo el esfuerzo que ha dedicado en la programación del intérprete paralelo de Multipath. Mi gratitud también va dirigida para Miguel Angel García por la ayuda propocionada en las primeras etapas de desarrollo de software para el grupo de investigación sobre programación declarativa.

Quisiera mencionar a todos los miembros del Departament d'Arquitectura de Computadors que, en algún momento, me han ayudado con sus comentarios y sugerencias a mejorar algunos puntos de este trabajo. Personalizo en mi compañero de despacho, David López, que es quien, seguramente, más ha tenido que soportarme durante todo este tiempo.

Mis agradecimientos van también para Peter Kacsuk y Gonzalo Escalada, que me han proporcionado puntos de vista interesantes sobre el tema de programación lógica y arquitecturas orientadas a este paradigma.

Por último, agradezco al CEPBA las facilidades dadas para el uso de sus recursos y al Ministerio de Educación por su financiación a través de la Comisión Asesora Científica y Técnica mediante los contratos TIC 1036/91 y TIC429/95.

CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE ALGORITMOS

Capítulo 6

PREFACIO

En esta tesis se presenta un nuevo sistema orientado a la ejecución de programas escritos en Prolog, denominado Multipath. El trabajo de investigación se concentra en la gestión del indeterminismo que surge durante la ejecución de los programas. Las principales contribuciones que aporta son las siguientes:

- Definición del Modelo de Ejecución de Multipath

  La novedad con respecto al modelo convencional de ejecución de Prolog estriba en la realización de una **exploración parcial en anchura** del árbol de búsqueda asociado a un programa. Esta forma de recorrido permite disminuir la penalización en que incurre la tradicional exploración en profundidad ante la presencia de objetivos indeterministas. Además, exhibe un tipo de paralelismo intrínseco en estos programas, basado en el paralelismo de datos y que, en este contexto, se ha denominado **paralelismo de caminos**. La determinación del atributo de determinismo/indeterminismo de los objetivos de que consta un programa se consigue mediante un análisis global del programa basado en técnicas de **interpretación abstracta**.

  Aprovechando este análisis global, Multipath también presenta técnicas que permiten la **reducción del árbol de búsqueda** mediante la localización de cláusulas candidatas en la resolución de un objetivo que no forman parte de una solución del programa.

- Definición del Modelo Arquitectónico de Multipath

  Se definen todos los elementos arquitectónicos que forman la denominada **Máquina Abstracta de Multipath**: motores, memorias, estructuras de datos, registros, instrucciones y comandos. Los puntos específicos que presentan mayor innovación hacen referencia a la gestión de las **variables Prolog** y de los **objetivos explorados en anchura**. El modelo arquitectónico posee la generalidad de poder ser realizado en un computador de propósito general, tanto en un entorno de ejecución secuencial

como paralelo.

La evaluación del rendimiento de Multipath se ha realizado comparando su ejecución con la ejecución convencional a partir de la Máquina Abstracta de Warren. Se han desarrollado y comparado ambos modelos en un **sistema secuencial** y en un **sistema multiprocesador con memoria compartida**.

El contenido de los capítulos en que se describe este trabajo es el siguiente.

En el capítulo 1 se realiza una introducción encaminada a enmarcar el contexto general en el que se circunscribe este trabajo, con especial hincapié en remarcar las motivaciones y objetivos que se plantean.

En el capítulo 2 se resumen los aspectos básicos que forman el modelo de ejecución convencional de Prolog y su modelo arquitectónico basado en la WAM, y se repasan algunos sistemas concretos orientados a la ejecución de aplicaciones escritas en este lenguaje.

La descripción básica e intuitiva de todas las ideas que se aplican en Multipath se encuentra en el capítulo 3. Estas aportaciones serán definidas con mayor detalle en los siguientes capítulos.

En este sentido, en el capítulo 4, se presenta la descripción formal del Modelo de Ejecución de Multipath, que está dividido en dos fases denominadas interpretación abstracta e interpretación real.

El capitulo 5 está dedicado a describir el Modelo Arquitectónico de Multipath en su fase de interpretación abstracta, y el capítulo 6 se concentra en definir la fase de interpretación real de dicho modelo arquitectónico.

En el capítulo 7 se analiza el comportamiento de Multipath en una ejecución secuencial realizada en la estación de trabajo Digital Alpha Station 5/166.

En el capítulo 8, el análisis de Multipath se realiza a partir de mapear el modelo arquitectónico en el sistema multiprocesador con memoria compartida SGI Power Challenge XL.

Por último, se resumen en el capítulo 9 las principales conclusiones y las líneas de trabajo abiertas en este trabajo.

Los apéndices A y B muestran, respectivamente, el código fuente de los programas benchmark analizados y el código intermedio de Multipath para uno de ellos.

1

INTRODUCCIÓN

En este trabajo de investigación se presenta un *sistema orientado a la programación lógica*, denominado *Multipath*. El concepto de sistema es tan amplio que sirve para referenciar una gran diversidad de objetos, máquinas o seres. En una de sus acepciones, un sistema es un dispositivo formado por una serie de elementos interrelacionados que permiten realizar una función determinada. La creación de nuevos sistemas va enfocada principalmente hacia la posibilidad de proporcionar una nueva funcionalidad o hacia una mejora de las prestaciones de sistemas ya existentes.

En el campo de la Informática aparece también el concepto de sistema. En este contexto, un sistema informático corresponde a la agrupación de elementos hardware y software con la característica de poseer una elevada potencialidad en ser utilizado para la resolución de problemas de carácter general. En términos generales, el hardware proporciona la potencialidad de ejecutar acciones e interaccionar con el exterior mientras que el software perfila la utilización que se va a hacer del sistema informático.

Es bastante común introducir un sistema informático como una "caja negra" que es capaz de obtener unos resultados a partir de una cierta interpretación de la información de entrada. La información de entrada que es capaz de interpretar define la utilización del sistema. Puede utilizarse un sistema como una máquina específica que realiza una cierta función: editor de textos, gestor de contabilidad, consola de videojuegos, etc. Puede utilizarse un sistema para generar aplicaciones que resuelven problemas de cualquier tipo (ingeniería del software), para resolver automáticamente problemas (sistemas expertos), etc. El grado de complejidad de la funcionalidad depende de la potencia del hardware y del software que posea el sistema.

Por otro lado, el estudio de un sistema puede realizarse desde múltiples niveles de abstracción según el detalle de precisión hasta el que se desee conocer. Un sistema puede estudiarse desde el punto de vista de una simple definición de su finalidad hasta un punto de vista más complejo referente a la descripción detallada de cada uno de los elementos que lo forman.

Esta introducción está enfocada a describir de forma general los diferentes lenguajes de programación utilizados en sistemas informáticos orientados a la resolución de problemas, así como los diferentes niveles de descripción de la interpretación del lenguaje de un sistema. Finalmente, se detallan las motivaciones y objetivos planteados en el diseño del sistema Multipath que se presenta en esta tesis.

1.1 Lenguajes de programación

Es de desear que la entrada de información a un sistema pueda realizarse de forma cómoda y rápida, y que la respuesta a la interpretación de esa entrada pueda obtenerse en un tiempo mínimo establecido como límite o, en su defecto, en el menor tiempo posible. En el contexto de la utilización de un sistema para la solución (automática o no) de problemas, la información que es capaz de interpretar se identifica mediante los lenguajes de programación.

Por otra parte, la resolución de un problema puede realizarse mediante la ejecución de uno o varios algoritmos, que tienen en cuenta el entorno en el que está definido el problema. El lenguaje de programación debe poseer la semántica adecuada para poder introducir esta información en el sistema informático. Además, el sistema será más eficiente cuanto mayor sea su capacidad de reaprovechamiento de la información proporcionada para resolver problemas posteriores.

1.1.1 Programas

Un programa corresponde a una instanciación concreta de las posibilidades que ofrece un len-

guaje de programación respecto a la capacidad de representar el problema que se quiere resolver, el entorno en el que está definido el problema y los algoritmos que hay que ejecutar para resolver el problema:

$$Programa = Problema + Entorno + Algoritmos$$

Esta definición responde a una generalización de la famosa ecuación de Wirth [136]: *Programa = Algoritmo + Estructuras de Datos*, cuando se considera que el objetivo final en la ejecución de un algoritmo sobre unas estructuras de datos es la resolución de un cierto problema. Por otra parte, el concepto de entorno engloba no tan sólo las estructuras de datos del programa sino también todo el conocimiento (expresado en la forma que establezca el lenguaje, por ejemplo, mediante reglas o funciones) que se puede utilizar en la resolución del problema. Por último, no hay que olvidar que un mismo problema puede resolverse de varias formas o, en otras palabras, pueden existir distintos algoritmos para la resolución de un problema. En este sentido, un programa puede ser capaz de expresar la posibilidad de escoger el algoritmo más adecuado para cada problema.

La utilización de un sistema informático en el desarrollo de programas para la resolución de aplicaciones se estudia dentro del campo de la Ingeniería del Software. La secuencia de pasos que hay que realizar en el desarrollo de programas responde a un proceso iterativo de especificación, desarrollo, prueba y optimización [60]. Si el grado de expresividad del lenguaje aumenta, la responsabilidad de llevar a cabo este proceso pasa del programador al sistema.

En este sentido, los lenguajes de programación se clasifican según su capacidad de descripción de los componentes de un programa. De forma más general, los lenguajes de programación se clasifican en imperativos y declarativos (figura 1.1).

1.1.2 Programación imperativa

Este tipo de lenguajes son capaces de expresar el entorno de un problema y un algoritmo para su resolución. Se suele denominar a este tipo de lenguajes como lenguajes orientados a la descripción de soluciones de problemas.

La utilización de sistemas orientados a la programación imperativa radica en la posibilidad de ejecutar, en el menor tiempo posible, algoritmos que, presumiblemente, conducirán a la resolución de problemas. En estos lenguajes, la capacidad de especificar un algoritmo, comprobar si conduce a la solución de un problema y optimizarlo son tareas en el desarrollo de un programa que son responsabilidad del programador.

PARADIGMAS DE PROGRAMACIÓN

Imperativa

Programa ≡ *Algoritmo* + *Estructuras de Datos*

- La ejecución de un algoritmo sobre unas estructuras de datos conduce a la resolución de un problema

Declarativa

Programa ≡ *Problema*

- Aprendizaje del conocimiento (Entorno).
- Resolución automática del problema por el sistema (Algoritmos implícitos).
- Tendencia a ampliar el dominio de problemas

Figura 1.1: *Paradigmas de programación imperativa y declarativa.*

El entorno de un problema se circunscribe a la especificación de las estructuras de datos que intervienen en el problema. Dependiendo del lenguaje, la especificación de las acciones que pueden aplicarse sobre las estructuras de datos están incluidas en la propia definición de las estructuras de datos (objetos), o bien, están incluidas en la definición del algoritmo. Habitualmente se utilizan procedimientos y funciones que ayudan a elevar el nivel de estructuración y expresividad del programa.

En la descripción del algoritmo de resolución de un problema normalmente se utiliza un control de flujo secuencial a la hora de especificar las distintas acciones de que consta un algoritmo. Una característica común es la posibilidad de utilizar estructuras de control (*for*, *repeat*, *while*, *do*) para describir acciones iterativas.

En algunos casos es posible expresar paralelismo en un programa de forma explícita, con el inconveniente que hay que bajar en el nivel semántico del lenguaje para tener en cuenta la arquitectura en concreto que posee el sistema a la hora, por ejemplo, de ubicar las estructuras de datos en memoria, de sincronizar las operaciones en paralelo, etc.

Hay que tener en cuenta que este tipo de lenguajes son los más utilizados en la actualidad en el desarrollo de programas para la resolución de problemas de carácter general. Los campos de investigación actuales están constituidos principalmente por la optimización en la traducción

del algoritmo al lenguaje máquina determinado por la arquitectura del sistema o por la posibilidad de paralelizar de forma automática algoritmos secuenciales para aplicaciones básicamente numéricas.

No obstante, existen una serie de posibilidades a la hora de expresar algoritmos que normalmente no están permitidas en este tipo de lenguajes: capacidad de expresar la ejecución de acciones en función de la disponibilidad de los datos (data-flow), capacidad de expresar indeterminismo en un algoritmo, etc.

1.1.3 Programación declarativa

La característica diferenciadora en este tipo de programación consiste en el incremento de la capacidad de expresividad que poseen los lenguajes [85]. De forma genérica, con este nuevo potencial se consigue desplazar los componentes que pueden ser descritos en un programa hacia la posibilidad de expresar el problema en el mismo programa.

En otras palabras, es posible la descripción de problemas y el entorno del problema, y se pretende que el algoritmo de resolución del problema pueda ser calculado de forma automática por el sistema que soporta este tipo de lenguajes. De forma análoga a los lenguajes imperativos o de especificación de soluciones, los lenguajes declarativos suelen denominarse lenguajes orientados a la descripción de problemas.

Este tipo de lenguajes poseen una semántica basada en la teoría matemática, lo que facilita el incremento en la expresividad y abstracción del problema, de forma independiente a la arquitectura del sistema. Al mismo tiempo, esta base matemática les permite aprovechar las ventajas de transparencia referencial y soportar métodos formales potentes de transformación de programas [93].

La consecuencia principal que provoca el incremento de expresividad en el lenguaje es el aumento del desnivel semántico entre el lenguaje soportado por el sistema y el lenguaje máquina de la arquitectura del sistema. Esto significa que usualmente una aplicación escrita en un lenguaje declarativo se ejecuta de forma más lenta que la misma aplicación escrita en un lenguaje imperativo. Esta misma situación sucede cuando una aplicación está escrita en un lenguaje imperativo o en el lenguaje máquina de la arquitectura del sistema. Como conclusión, es indispensable investigar mecanismos automáticos de transformación y de optimización de programas escritos en este tipo de lenguajes, así como también identificar los elementos del nivel de lenguaje máquina que pueden introducirse en la arquitectura del sistema en función del coste y rendimiento que se obtiene.

En contrapartida, los lenguajes declarativos facilitan las tareas a realizar en el desarrollo de aplicaciones ya que el nivel de abstracción del programa permite reducir el coste de construcción de un programa y automatizar los procesos de comprobación y optimización [75].

La utilización actual de los lenguajes declarativos no abarca la resolución de problemas de carácter general. Están especialmente orientados hacia el desarrollo rápido de prototipos de aplicaciones en campos como cálculo simbólico, inteligencia artificial y sistemas basados en el conocimiento.

Dentro de los lenguajes declarativos actuales se encuentran tres grandes tipos de lenguajes: lenguajes lógicos, funcionales y relacionales.

Los lenguajes lógicos tienen su base en la posibilidad de utilizar la teoría existente en el campo de la lógica matemática para describir el conocimiento de un problema y utilizar sus mecanismos de inferencia y refutación como modelo básico de ejecución ([96] y [134]). Los lenguajes más conocidos de este paradigma de programación son Prolog [107], Gödel [52] y Mercury [102], entre otros.

Los lenguajes funcionales tienen su modelo de programación basado en la evaluación de expresiones [6]. Expresiones que se construyen a partir de la aplicación de funciones sobre otras expresiones; y funciones que, a su vez, se representan como otras expresiones. De forma general, el modelo de ejecución debe decidir el momento en que se realiza la aplicación de las funciones y la evaluación de las expresiones. El lenguaje emblemático de este paradigma de programación es LISP [78], junto con multitud de dialectos que han aparecido posteriormente: SCHEME [103], Common LISP [106], MultiLISP [45], etc. Lenguajes funcionales más recientes son Miranda [125] y Haskell [55], entre otros. En [54] puede obtenerse una completa recopilación de las características principales de los lenguajes funcionales.

Los lenguajes relacionales permiten acceder a la información contenida en bases de datos a partir de la descripción de las propiedades que deben satisfacer [83]. Puede observarse que estos lenguajes están más cerca de la especificación de problemas ya que los mecanismos de control para acceder a la información están implícitamente definidos en el sistema. El lenguaje relacional más típico es SQL.

Actualmente van apareciendo nuevas propuestas de definiciones de lenguajes. La característica común es el incremento en la expresividad del lenguaje.

En este sentido, se encuentra el paradigma de la programación lógica con restricciones (CLP) [59]. En este nuevo paradigma, se añade el concepto de dominio de una variable y de

restricciones que debe satisfacer una variable. De cara al programador, esto permite modificar algoritmos del tipo generar todas las posibilidades y comprobar si corresponde a una solución por algoritmos que únicamente deben encargarse de establecer las restricciones que deben cumplir las variables, y dejar al sistema la parte del algoritmo que trata de encontrar los valores de las variables que cumplen las restricciones. Esta parte del algoritmo no es responsabilidad del programador y está optimizada por el sistema.

Otra posibilidad es la definición de nuevos lenguajes que integran el paradigma de la programación funcional y lógica [46]. En este caso, se trata de combinar las ventajas que ofrecen los lenguajes lógicos (variables lógicas, unificación, deducción de inferencias) con las ventajas que ofrecen los lenguajes funcionales (evaluación de funciones anidadas, declaración de tipos, programación de orden superior, evaluación perezosa de expresiones). Ejemplos de lenguajes que aplican esta idea son λProlog [82], Babel [84] y Escher [74], entre otros.

1.2 Algoritmos indeterministas

Sea proporcionado por el programador u obtenido de forma automática por el sistema, el establecimiento de un algoritmo constituye el primer paso para la resolución de un problema. En el apartado anterior se ha comentado que pueden existir varios algoritmos para resolver un mismo problema. A su vez, esta situación también se puede encontrar en el propio algoritmo, cuando una o varias acciones que lo componen pueden completarse con éxito de formas distintas. En estos casos, nos encontramos ante algoritmos indeterministas.

Un algoritmo determinista es aquel en el que únicamente existe una única forma de realizar satisfactoriamente cada una de las acciones que lo constituyen (una solución por acción). Por el contrario, en un algoritmo indeterminista pueden existir varias formas de resolver algunas de las acciones que lo componen. Posiblemente, sólo algunas de estas soluciones contribuyen a encontrar una solución global del problema, pero la determinación de la validez de las soluciones parciales no es posible realizarla hasta conocer más información en algún momento posterior de la ejecución del algoritmo.

El cálculo de aquellas soluciones parciales que, posteriormente, son descartadas para encontrar una solución del problema incrementa el tiempo de ejecución de un programa. En este sentido, una de las características primordiales que debe exhibir un algoritmo es un alto grado de determinismo [23]. Sin embargo, no siempre es posible obtener un algoritmo determinista, sea porque es imposible encontrarlo o porque el coste global de desarrollar y ejecutar un algoritmo indeterminista es menor.

En este contexto surgen dos campos de investigación. Por un lado, la reducción del grado de indeterminismo de un algoritmo y, por otra parte, la gestión eficiente del indetermismo remanente en un algoritmo.

1.2.1 Reducción del indeterminismo

Las técnicas de reducción del indeterminismo se pueden agrupar en dos tipos: aumento de la expresividad del lenguaje y disminución automática del grado de indeterminismo de un algoritmo.

Con respecto al primer tipo, elevando la expresividad del lenguaje (y la complejidad del sistema) puede facilitarse al programador la generación de algoritmos más deterministas. Los ya mencionados lenguajes lógicos con restricciones (CLP) se encuadran en este tipo de reducción del indeterminismo. Los lenguajes CLP más representativos son los que permiten definir restricciones sobre el dominio de los números reales (clp(R) [58]), sobre dominios de elementos finitos (clp(FD) [34]) y sobre el dominio de los booleanos (clp(B) [22]).

Con referencia a la segunda posibilidad de reducción del indeterminismo, la realización de una etapa de análisis global en la compilación de un programa posibilita la transformación automática de un algoritmo en otro más determinista. Este análisis permite eliminar aquellos cálculos que no contribuyen a una solución del problema, o a insertar código para que en tiempo de ejecución pueda determinarse lo antes posible aquellos cálculos que no es necesario realizar. Como ejemplo cabe citar al sistema Andorra-I [98], que reordena los objetivos parciales de un algoritmo de manera que se ejecutan en primer lugar aquellos objetivos deterministas que pueden disminuir el grado de indeterminismo de otros objetivos del algoritmo.

1.2.2 Gestión del indeterminismo

La forma más usual de representar el indeterminismo remanente en un algoritmo es mediante un árbol de búsqueda, en el que cada nodo posee tantas ramas hijas como alternativas existen para poder resolver una cierta acción. Existen dos formas básicas de recorrer estos árboles [91]: en profundidad o en anchura (figura 1.2).

En un recorrido en profundidad (*depth-first search*) se elige una alternativa en un punto en que exista indeterminismo, por lo que únicamente existe un único camino activo durante la ejecución de un programa, y las restantes alternativas son exploradas posteriormente. En una exploración en anchura (*breadth-first search*) se permite recorrer simultáneamente varias alternativas indeterministas.

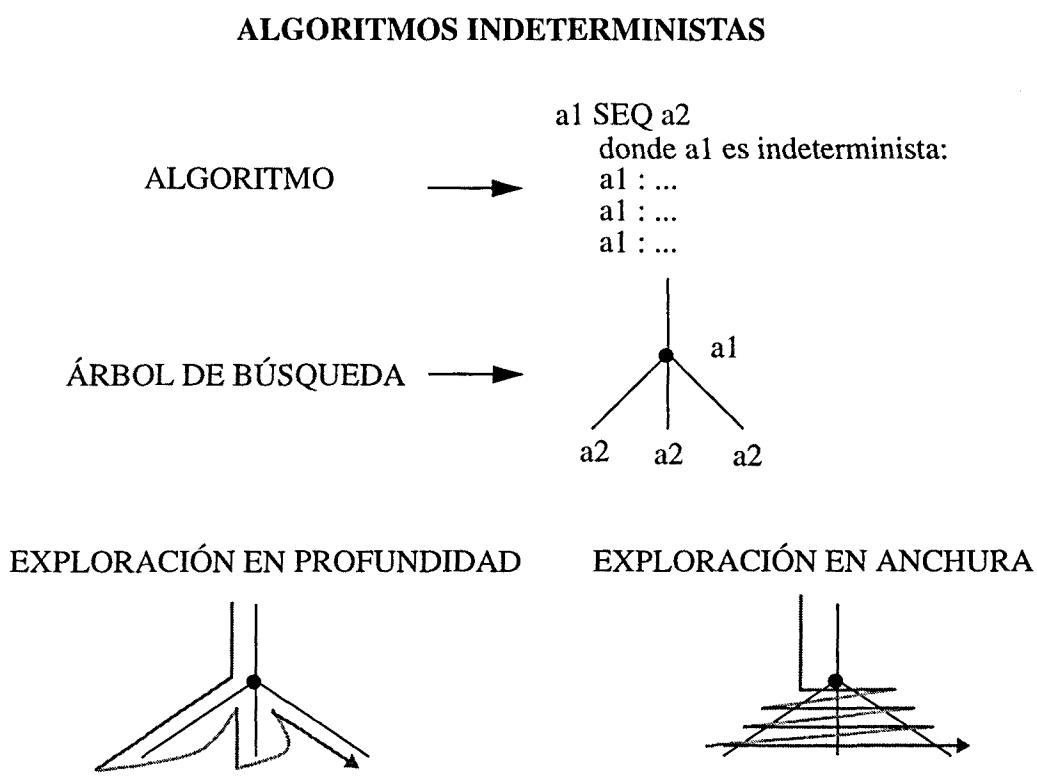


Figura 1.2: *Esquematización de la exploración en profundidad y en anchura de árboles de búsqueda.*

Tradicionalmente, los sistemas que admiten la posibilidad de expresar indeterminismo en el lenguaje efectúan un recorrido en profundidad del árbol de búsqueda, por la sencillez que supone tener visible un único camino de dicho árbol durante la ejecución. Sin embargo, en el momento en que finaliza una acción indeterminista con varias soluciones, el recorrido en profundidad repite las acciones siguientes para cada solución, mientras que un recorrido en anchura permite ejecutar una única vez todas aquellas operaciones que no dependen de datos concretos a cada solución.

Otras formas de recorrer un árbol de búsqueda son: *depth-first iterative-deepening* [66], que también se basa en una exploración ciega del árbol como las dos anteriores; y *best-first* [31] y *branch-and-bound* [68], que están basadas en la aplicación de heurísticas orientadas a encontrar la solución.

1.3 Descripción de un sistema

La descripción de un sistema puede realizarse desde varios niveles de abstracción. En esta

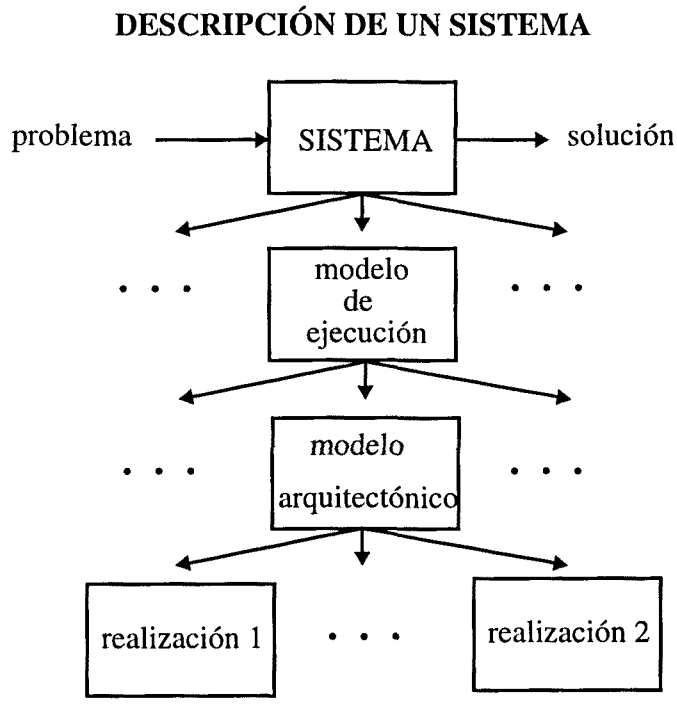


Figura 1.3: *Descripción de un sistema con tres niveles de abstracción: modelo de ejecución, modelo arquitectónico y realización física.*

sección se introducen tres niveles distintos de descripción (figura 1.3), que serán utilizados en los capítulos posteriores. En primer lugar, el denominado *modelo de ejecución*, que tiene como objetivo identificar los elementos que forman el estado de la ejecución del sistema y las operaciones necesarias para interpretar la semántica que posee el lenguaje. En segundo lugar, el *modelo arquitectónico* define una representación del estado de la ejecución y la forma de realizar las operaciones basadas en esa representación. En último lugar, la descripción de un sistema corresponde a la identificación de los elementos hardware y software que componen la *realización concreta* de dicho sistema.

1.3.1 Modelos de Ejecución

El modelo de ejecución de un lenguaje de programación establece el mecanismo de control y las operaciones necesarias con el objetivo de conseguir el resultado que se desea obtener de un programa y que viene determinado por la semántica que posee el lenguaje. Dentro de las funciones del modelo de ejecución tienen cabida todas las transformaciones y optimizaciones que pueden aplicarse a un programa que pueden realizarse de forma independiente a la arquitectura del sistema. El criterio que rige en la definición de un modelo de ejecución es la pre-

servación del significado de un programa y la minimización de las operaciones necesarias para su ejecución.

En el contexto de los lenguajes imperativos, o lenguajes orientados a la descripción de soluciones de problemas, la semántica del propio lenguaje especifica de forma determinista la ejecución de un programa. Por tanto, en los sistemas orientados a lenguajes imperativos, el modelo de ejecución únicamente describe las transformaciones y optimizaciones que el compilador puede realizar a un programa.

En el otro extremo, en un contexto de lenguajes orientados a la descripción de problemas, el sistema debe definir la estrategia a seguir para intentar resolver el problema. En este caso, el modelo de ejecución se comporta como un sistema experto que trata de encontrar la solución de un problema mediante algoritmos predefinidos u obtenidos a través de otros sistemas, o mediante la capacidad de aprendizaje que posea el sistema.

En el entorno de la programación lógica y, en concreto, el lenguaje Prolog, la ejecución de un programa se comporta como un demostrador automático de teoremas [73]. A partir de un conjunto de axiomas (cláusulas), se pretende demostrar si un teorema (pregunta del programa) es cierto o no, y para qué instanciaciones de las variables se cumple en caso de que sea cierto. La demostración de un teorema se realiza mediante un procedimiento de refutación que admite una serie de posibilidades en la regla de inferencia, en la regla de selección (tanto de objetivos como de cláusulas) y en la regla de búsqueda utilizadas. El modelo convencional de ejecución de Prolog describe de forma determinista este procedimiento de refutación.

Existen diferentes maneras de poder definir un modelo de ejecución: mediante la descripción de su comportamiento de forma operacional, axiomática o denotacional [65]. La definición de los modelos de ejecución que se realiza en este trabajo se basa en la descripción axiomática de una serie de objetos abstractos que representan el estado de la ejecución en un momento determinado y de las operaciones que los actualizan durante la ejecución de un programa. Se pretende incluir en el modelo de ejecución únicamente aquellos aspectos que son independientes de la arquitectura del sistema que ejecute este modelo. De esta forma, el modelo de ejecución es lo suficientemente flexible para ser utilizado en diferentes arquitecturas.

1.3.2 Modelos arquitectónicos

Una vez definido un modelo de ejecución, en el siguiente paso de refinamiento se establece la forma en que se llevan a cabo cada una de las fases que lo componen. En concreto, debe definirse la representación del estado de la ejecución y la realización de las operaciones carac-

terísticas de cada una de las fases.

Es preciso tener en cuenta las ventajas que proporciona la descripción de un modelo arquitectónico frente a la definición de una realización concreta. En un modelo arquitectónico se definen aquellos elementos arquitectónicos básicos que permiten realizar completamente el modelo de ejecución, sin acabar de refinar aquellos aspectos que tienen la posibilidad de realizarse de forma diferente según las características hardware de la arquitectura específica en la que al final se ejecute el programa.

La generalidad de un modelo arquitectónico aumenta la flexibilidad con que se puede portar un sistema a diferentes máquinas. Con esta idea, con las mínimas modificaciones se puede hacer frente a posibles cambios en la plataforma hardware. Estos cambios pueden ser originados por la posibilidad de pasar de una arquitectura secuencial a una paralela, o por avances tecnológicos que permiten incrementar la potencia de algunos elementos del computador e introducir ciertas operaciones realizadas por software en el hardware del sistema. No hay que olvidar también la posibilidad de emular el modelo arquitectónico mediante una interpretación software.

Es importante resaltar que la definición de un modelo arquitectónico es una manera de describir la ejecución de un programa a un nivel de abstracción más bajo que el realizado en el modelo de ejecución, donde probablemente se han tenido que tomar decisiones respecto a la representación y realización de sus operaciones, que pueden favorecer el uso de un cierto tipo de arquitecturas específicas respecto a otras. Uno de los objetivos en este trabajo es la definición de un modelo arquitectónico que pueda ser lo más general posible.

1.3.3 Realizaciones de un sistema

En el último nivel de descripción de un sistema, se especifican todas aquellas decisiones que no han sido determinadas en el modelo arquitectónico y que dependen de la realización concreta del sistema.

Por ejemplo, puede optarse por una realización secuencial de un modelo arquitectónico que admite intrínsecamente una ejecución paralela. O en caso de una realización paralela, también se podría optar por incrementar la granularidad de los procesos que intervienen, con lo que debería describirse la técnica de distribución de trabajo a elementos de proceso. Otra decisión importante es la determinación de dónde termina el componente hardware y, por tanto, donde empieza el componente software.

Si se opta por una realización completamente hardware significa que el modelo arquitec-

tónico coincide con el lenguaje máquina del sistema, tanto en la representación de los elementos que forman el estado de la ejecución como de las operaciones intrínsecas del modelo, que corresponderían a las instrucciones del lenguaje máquina.

En un punto intermedio, cuando se opta por una realización software de alguna de las fases de ejecución, el lenguaje que se utiliza para la descripción del estado de la ejecución y la realización de las operaciones debe estar ya previamente soportado por el sistema. La principal ventaja de una realización software radica en la flexibilidad de portar el modelo de ejecución a diferentes sistemas, siempre que soporten el lenguaje utilizado. Sin embargo, la eficiencia del sistema en cuanto a tiempo de ejecución no es la óptima. Nótese que otra gran ventaja es la posibilidad de investigar aquellos elementos arquitectónicos que pueden agregarse al lenguaje máquina original en función del compromiso rendimiento/coste que ofrezcan.

En resumen, tanto el modelo de ejecución como el modelo arquitectónico proporcionan las ideas motrices que caracterizan al sistema. Dichos modelos pueden tener varias realizaciones físicas que deben especificarse en este último nivel de descripción de un sistema.

1.4 Motivación y objetivos de la tesis

Actualmente, los sistemas informáticos están evolucionando continuamente por los incrementos tecnológicos que permiten la fabricación de componentes hardware más eficientes y que a su vez permiten la posibilidad de potenciar el componente software.

Desde un punto de vista personal, el primer aspecto a tener en cuenta en un trabajo de investigación es la funcionalidad hacia la que está dirigido. La tendencia que se observa es el incremento de la funcionalidad de los sistemas informáticos. A largo plazo, esto puede conducir al diseño de sistemas orientados a automatizar la resolución de problemas.

Por otra parte, se ha comentado previamente que en la resolución de un problema pueden utilizarse uno o varios algoritmos que, a su vez, contienen acciones que pueden solucionarse de distintas formas. Este indeterminismo, presente en gran cantidad de situaciones, obliga a tomar decisiones en ciertos momentos de la ejecución del programa cuya responsabilidad debe ser tomada por el sistema.

Resumiendo los dos párrafos anteriores, la motivación principal que ha originado este trabajo es la mejora de la gestión del indeterminismo que pueden poseer los algoritmos, en un contexto de sistemas orientados hacia la resolución automática de problemas.

La determinación del lenguaje del sistema condiciona la forma concreta de gestionar el

indeterminismo. Como punto de vista personal, el lenguaje a que estarán orientados los futuros sistemas debe tener un alto grado de abstracción y de expresividad y, al mismo tiempo, de flexibilidad. El incremento de la expresividad debe conducir a dotar al lenguaje de los mecanismos adecuados para facilitar la descripción de problemas, entornos y algoritmos. El incremento de la abstracción del lenguaje puede permitir la aplicación de potentes mecanismos de transformación y optimización de programas. Al mismo tiempo, facilita la portabilidad de las aplicaciones a otros sistemas con arquitecturas diferentes. Por último, el incremento en la flexibilidad del lenguaje tendría en cuenta que no hay un único algoritmo de resolución de un problema. Es necesario poder utilizar todo el conocimiento posible que se sepa de un problema (entorno y algoritmos) para poder determinar la manera más eficiente de obtener la solución.

A un nivel muy genérico, un programa puede ser únicamente la descripción de un problema, si el sistema posee la capacidad de memorización de entornos y algoritmos, o bien, puede obtenerlos de otros sistemas. En cualquier caso, el sistema ha de ser capaz de aprender entornos y algoritmos. El lenguaje debe permitir expresar de forma declarativa el conocimiento del entorno: objetos, reglas, transformaciones, optimizaciones dependientes de la arquitectura, etc. Por otra parte, el lenguaje también debe poder expresar todos aquellos aspectos que pueden aparecer en la descripción de un algoritmo: ejecución secuencial, paralela, conducida por los datos, coroutining, indeterminismo, etc.

Sin embargo, el objetivo final en este trabajo no es el diseño de un nuevo lenguaje de programación. Ante la enorme proliferación de nuevos lenguajes, el establecimiento de otro lenguaje requiere un esfuerzo que puede quedar en el anonimato si no está respaldado desde un principio por la comunidad científica. El lenguaje hacia el que está orientado el sistema Multipath es *Prolog*. Este lenguaje permite expresar indeterminismo en los programas y es un ejemplo de lenguaje consolidado que tiende hacia la descripción de problemas.

La idea clave que subyace en este trabajo es demostrar que *un recorrido en anchura de todas o parte de las diferentes posibilidades que establece un algoritmo indeterminista es más eficiente que un recorrido en profundidad,* que tradicionalmente se ha venido realizando en los sistemas informáticos orientados a la ejecución del lenguaje Prolog. También se pretende investigar su implementación eficiente en diferentes tipos de plataformas.

La definición de Multipath se realiza mediante la descripción de su modelo de ejecución, su modelo arquitectónico y finalmente dos realizaciones concretas: una secuencial y otra paralela. La evaluación del rendimiento de Multipath se efectúa comparando ambas realizaciones con la ejecución convencional de Prolog.

2

SISTEMAS ORIENTADOS A PROLOG

En este capítulo se resumen las características más relevantes de los sistemas que están orientados a la ejecución del lenguaje Prolog, el mismo lenguaje que interpreta el sistema Multipath presentado en esta tesis.

En primer lugar se presentan los conceptos básicos fundamentados en la teoría de la lógica matemática necesarios para comprender la semántica que posee el lenguaje Prolog. Posteriormente se describe el mecanismo de ejecución convencional de un programa. En la siguiente sección se definen los elementos arquitectónicos utilizados en su ejecución y, a continuación, las realizaciones concretas de diferentes sistemas representativos ya existentes o propuestos en la literatura.

La ejecución standard de Prolog responde a un control secuencial de las diferentes acciones que lo definen. Sin embargo, la semántica del lenguaje ofrece diversas fuentes de paralelismo implícito que se han explotado en gran cantidad de sistemas. En este capítulo también se resumen brevemente estas fuentes de paralelismo y se citan los principales sistemas que lo explotan.

2.1 Lenguaje Prolog

Prolog es el lenguaje emblemático de la programación lógica. En su forma declarativa pura, un programa Prolog permite especificar un cierto tipo de problemas y el entorno que le rodea. El tipo de problemas que especifica agrupa a aquéllos que pueden clasificarse como demostraciones de teoremas. La ejecución de un programa intenta resolver de forma automática el problema y, por tanto, en este contexto declarativo, la ejecución de un programa intenta realizar la demostración automática de un teorema. Por otra parte, puede utilizarse el mecanismo implícito de demostración de teoremas para programar algoritmos que resuelvan aplicaciones de carácter general. En esta situación, parte del algoritmo de resolución del problema viene especificado en el programa y la parte restante es inherente en la ejecución del programa.

Esta dualidad en la interpretación de un programa permite realizar la especificación de la semántica del lenguaje Prolog desde dos puntos de vista: teniendo en cuenta la visión puramente declarativa que exhibe un programa, entendido como un demostrador de teoremas, o bien, mediante una visión imperativa de un programa, cuando este programa corresponde a una plasmación de un algoritmo que pretende resolver un problema de carácter general.

2.1.1 Semántica declarativa

El lenguaje permite describir un teorema y una serie de axiomas basados en la lógica de predicados de primer orden, que se especifican mediante cláusulas de Horn [73]. Recordando los tres componentes que forman un programa, descritos en el capítulo introductorio, puede asociarse el problema al teorema que se pretende demostrar, el entorno a los axiomas (representan el conocimiento sobre el entorno del teorema, en este caso, las reglas de transformación que se pueden aplicar para demostrar el teorema) y el algoritmo del programa es implícito. La ejecución de un programa consiste en la demostración de la validez del teorema a partir de los axiomas indicados en el mismo programa. Esta visión del lenguaje se conoce con el nombre de semántica declarativa.

La interpretación de un axioma, tal como establece la lógica de predicados de primer orden, corresponde a una indicación de los predicados que tienen que satisfacerse para poder deducir que otro predicado es correcto, teniendo en cuenta la universalidad y existencialidad de las variables que aparecen en los predicados. Por ejemplo, dado el siguiente axioma representado en forma clausal, y según admite la sintaxis de Prolog:

a(X) :- b(X), c(Y), d(X,Y).

significa que "para todo valor de X, tal que existe un valor Y, tal que los predicados b(X), c(Y) y d(X,Y) (antecedentes) son ciertos, entonces el predicado a(X) (consecuente) también es cierto", es decir:

$$\forall\ X\ \exists\ Y\ (b(X),\ c(Y),\ d(X,Y)) \Rightarrow a(X)$$

La interpretación declarativa del teorema de un programa, por ejemplo:

$$?\text{-}\ a(Z).$$

es "existe algún valor para Z tal que el predicado a(Z) es cierto".

La ejecución de un programa tiene como objetivo la demostración automática del teorema, es decir, intenta contestar al significado del teorema cuestionado en forma de pregunta. El mecanismo implícito de demostración de teoremas, también denominado procedimiento de decisión en el contexto de la lógica matemática, se basa en una refutación del teorema, y se cita a continuación.

El punto de partida inicial consiste en utilizar la demostración mediante la reducción al absurdo. Inicialmente, se considera que todos los axiomas son ciertos. Suponiendo que el teorema que se pretende demostrar es cierto, se añade la negación del teorema en el conjunto de axiomas. El proceso debe intentar la realización de inferencias lógicas a partir del conjunto de axiomas que forman el programa. Si durante este proceso se llega a una contradicción, la única posibilidad es que la negación del teorema que se había introducido inicialmente es falsa y, por tanto, el teorema queda demostrado.

La regla de inferencia que se utiliza se denomina *resolución SLD*, o resolución *L*ineal para cláusulas *D*efinidas y con la existencia de una regla de *S*elección, y fue descrita por primera vez por Kowalski [67].

Las *cláusulas definidas* corresponden a aquellos axiomas que contienen únicamente un predicado en el consecuente y además todos los predicados (antecedentes y consecuente) están expresados en forma positiva.

En el campo del cálculo proposicional la resolución es un tipo de inferencia que permite hacer deducciones de la siguiente forma:

$$(A,\ B \Rightarrow X),\ (X,\ C \Rightarrow D) \vdash (A,\ B,\ C \Rightarrow D)$$

en la que, a partir de dos axiomas en los que aparece la misma proposición en un consecuente y en un antecedente, se deduce un nuevo axioma que se denomina resolvente.

En el campo de la lógica de predicados de primer orden es necesario comprobar si los dos predicados que se eliminan de los dos axiomas iniciales coinciden. Si es posible obtener una igualdad entre ambos predicados, la resolución debe calcular el denominado *unificador más general* (*mgu*) [95]. Este unificador es un tipo particular de substitución (θ) que contiene los vínculos mínimos, de las variables (X_i) que aparecen en los argumentos de los predicados, a aquellos términos (t_i) que permiten su igualdad:

$$mgu \equiv \theta = \{\ldots, X_i/t_i, \ldots\}$$

Al resolvente que se obtiene en la resolución se le aplica el mgu obtenido, es decir, se substituyen todas las variables del resolvente que poseen un vínculo en *mgu* por el término correspondiente.

La *resolución lineal* significa que se establece un orden para aplicar las resoluciones. Inicialmente, un axioma debe ser el teorema negado y el otro debe corresponder a cualquier axioma del programa. Las siguiente resoluciones deben partir del anterior resolvente calculado y otro axioma del programa.

La *regla de selección* establece el predicado que se trata de resolver en cada resolución. En esta visión declarativa de un programa, la refutación del teorema es independiente de la regla de selección que se aplique.

La aplicación de una secuencia (finita o infinita) de resoluciones SLD, y teniendo en cuenta que en cada una de ellas se puede utilizar cualquier axioma para resolver un predicado, genera un árbol que representa todas las posibles demostraciones del teorema de partida. Este árbol se denomina árbol SLD. Cada nodo del árbol SLD tiene tantas ramas hijas como axiomas pueden utilizarse para la resolución del predicado que indica la regla de selección. Cada hoja del árbol corresponde a dos posibilidades: el resolvente está vacío, que corresponde al caso de llegar a una contradicción y, por tanto, el teorema está demostrado; o bien, contiene un resolvente no vacío sin posibilidad de poder inferir un nuevo resolvente. Este tipo de caminos fracasan en la demostración del teorema.

Para acabar de especificar de forma determinista el procedimiento de refutación implícito de los programas Prolog, hace falta especificar la regla de selección, que determina diferentes árboles SLD, y una *regla de búsqueda* que indica el orden de recorrido de un árbol SLD. La regla de selección que se utiliza en la ejecución standard de Prolog escoge el objetivo de más a la izquierda de cada nuevo resolvente. La regla de búsqueda establece que debe realizarse un recorrido en profundidad del árbol, teniendo en cuenta que las ramas hijas de un nodo se

ordenan según el orden textual de escritura de los axiomas que se utilizan en la resolución.

Este procedimiento de decisión es sólido (si se encuentra una solución es correcta) pero no es completo (pueden no encontrarse todas las soluciones debido a ramas infinitas) [73].

Como punto final, es importante recordar que este algoritmo utilizado para la demostración de un teorema no es necesario que sea conocido por el programador.

2.1.2 Semántica imperativa

En esta visión, un programa está compuesto de una pregunta y de una serie de procedimientos. La pregunta, que corresponde al concepto de teorema utilizado en el apartado anterior, está formada por uno o más predicados. Cada predicado se interpreta como un objetivo que se pretende satisfacer. Por otra parte, cada procedimiento engloba a todas aquellas cláusulas (o axiomas, según se interpretaban en la visión declarativa) que poseen el mismo nombre de predicado en su cabecera y aridad.

La ejecución de un programa tiene un secuenciamiento implícito cuyo orden corresponde a ejecutar los objetivos de la pregunta de izquierda a derecha. Cada objetivo genera la activación del procedimiento que posee el mismo nombre de predicado y aridad. El resultado de esta llamada equivale a una indicación lógica acerca de si el predicado puede satisfacerse o no.

Para satisfacer un objetivo pueden utilizarse cualquiera de las cláusulas del procedimiento pero en un orden determinista, que corresponde al orden textual de escritura de las cláusulas. En primer lugar, se realiza una operación propia del lenguaje Prolog, denominada *unificación*, que establece si el objetivo que se pretende satisfacer y la cabecera de una cláusula coinciden. En caso de coincidencia, la unificación calcula los mínimos vínculos de las variables lógicas que aparecen en los literales que permiten su igualdad. Las variables Prolog son variables de asignación única. Si la unificación tiene éxito, deben satisfacerse todos los objetivos del cuerpo de la cláusula, en el orden textual de escritura, para que el objetivo inicial se haya satisfecho.

En el momento que una unificación fracasa, se produce otra operación típica de los programas Prolog, que se denominada *backtracking*, y que consiste en intentar resatisfacer el objetivo más reciente que posee cláusulas alternativas pendientes.

La ejecución de un programa finaliza en el momento en que se satisface la pregunta inicial del programa o cuando no existe ninguna otra cláusula alternativa para resatisfacer un predicado. En caso de satisfacerse la pregunta inicial existe la posibilidad de pedir otras formas de satisfacer esta pregunta.

Esta visión de un programa Prolog equivale a una descripción imperativa y determinista de la ejecución del programa. El programador debe conocer esta semántica para poder utilizar el lenguaje en la especificación de algoritmos para la resolución de problemas de carácter general. Con esta idea, se amplian las posibilidades del lenguaje mediante el establecimiento de predicados predefinidos, que no responden a la visión puramente lógica, pero que son totalmente necesarios para la especificación de una gran cantidad de algoritmos. Dentro de estos predicados no lógicos se encuentran predicados que permiten modificar el control de flujo implícito que se ha descrito anteriormente (operador de corte "!"), predicados aritméticos (is), predicados con efectos laterales (read, write), de inserción y eliminación dinámica de cláusulas (assert, retract), etc.

En otras palabras, en esta visión se considera al lenguaje Prolog como un lenguaje imperativo al que se añaden dos operaciones propias que lo caracterizan: la unificación y el backtracking, donde las cláusulas de un procedimiento indican la secuencia de acciones de que consta el algoritmo que resuelve dicho procedimiento o acción.

2.2 Ventajas e inconvenientes de Prolog

El modelo de programación del lenguaje Prolog ofrece una serie de ventajas e inconvenientes que se tratan de resumir seguidamente.

La principal ventaja radica en que la ejecución de programas Prolog constituye un procedimiento automático de demostración de teoremas. Sin embargo, este mecanismo implícito de ejecución es fijo y no admite ninguna posibilidad de modificación por parte del usuario salvo por el operador de corte, que permite recortar el árbol de búsqueda al evitar realizar la unificación de objetivos con sus cláusulas alternativas pendientes. La rigidez del mecanismo de ejecución se pone de manifiesto en la regla de inferencia (resolución SLD), de selección (satisfacción de objetivos de izquierda a derecha) y de búsqueda (profundidad), que son totalmente deterministas. En este aspecto, sería aconsejable dotar al sistema de mayor flexibilidad y poder optar por la alternativa que mejor se adapte a cada situación particular.

Prolog permite la especificación de indeterminismo en un algoritmo. Si existen diferentes maneras de resolver una acción, cada una de ellas se puede representar mediante una cláusula. En tiempo de ejecución, la gestión del indeterminismo se realiza principalmente mediante la operación implícita de backtracking. Sin embargo, aun permitiendo la especificación de indeterminismo en un algoritmo, la ejecución de un programa es totalmente determinista ya que el orden de exploración del árbol de búsqueda es fijo.

Prolog también facilita la operación de comparación de objetos (pattern matching), liberando al programador de la tarea de tener que definir esta operación para cada uno de los objetos diferentes que se pretenda comparar. En tiempo de ejecución, esta tarea se realiza mediante la operación de unificación.

La existencia de variables de asignación única hace posible que se cumpla la propiedad de transparencia referencial durante la ejecución de un programa. En este contexto, significa que la satisfacción de un objetivo siempre tendrá los mismos resultados, en cuanto a instanciación de variables, para unos mismos valores de entrada de las variables. Por otra parte, la restricción de asignación única para las variables ocasiona que el mecanismo básico de especificación de algoritmos esté basado en la recursividad de procedimientos. En este sentido, acciones iterativas dentro de un procedimiento no están permitidas.

Prolog no es un lenguaje tipificado, cuando desde el punto de vista de un programador de aplicaciones se puede saber perfectamente los objetos que intervienen en un problema y en el algoritmo que lo resuelve. Por otra parte, los objetos que intervienen en un programa son creados dinámicamente, con el consiguiente perjuicio en cuanto a rendimiento al no poder crear estos objetos de forma estática.

En resumen, el lenguaje Prolog tiene aspectos positivos que facilitan el proceso de desarrollo de software, pero requiere un considerable esfuerzo a nivel de investigación de nuevos mecanismos de ejecución. Estos nuevos modelos de ejecución deben ser capaces de aprovechar las ventajas que el incremento del nivel de abstracción del lenguaje proporcionan para poder realizar transformaciones y optimizaciones del programa. Al mismo tiempo, se deben definir los elementos arquitectónicos oportunos con el objetivo final de realizar ejecuciones más eficientes de programas Prolog.

2.3 Modelo de ejecución standard de Prolog

En esta sección se describe el comportamiento que sigue la ejecución convencional de un programa Prolog. Esta descripción se realiza de forma axiomática y tiene en cuenta únicamente los aspectos que forman parte del lenguaje Prolog puro. A su vez, es consistente con la definición del modelo de ejecución propuesto en el standard de Prolog ISO/IEC 13211-1 1995 [57].

En cierto sentido, esta descripción es otra forma de entender un programa Prolog en su visión imperativa, ampliándola con aquellos elementos que caracterizan el estado de la ejecución y las operaciones que lo modifican. Es importante recordar que la definición de este modelo de ejecución es independiente de cualquier arquitectura de un sistema informático que lo realice.

2.3.1 Estado de la ejecución

El estado de la ejecución (EJ) de un programa en un instante concreto se identifica por cuatro elementos:

$$EJ \equiv \{BD,\ OP,\ EV,\ AP\}$$

cuya definición y funcionalidad es la siguiente:

Base de Datos (BD). Es un conjunto de cláusulas (cl_i), tanto reglas como hechos, que representan el conocimiento del programa. A partir de estas cláusulas se debe intentar resolver la pregunta inicial del programa. Las cláusulas están agrupadas en procedimientos, de forma que cada procedimiento contiene todas las cláusulas que poseen el mismo nombre de predicado y aridad en su cabecera (*p/n*).

Objetivos Pendientes (OP). Es una lista ordenada de predicados que representan los objetivos que faltan por satisfacer para resolver el programa.

Entorno de Vínculos (EV). Es un conjunto de elementos Variable/Término. Mantiene la información de todos los vínculos de variables que se han realizado desde el inicio del programa hasta el momento actual de la ejecución.

Alternativas Pendientes (AP). Es una pila de elementos {OP, EV}. Cada uno de estos elementos de AP agrupa los elementos OP y EV del estado de la ejecución en un momento anterior del programa, indicando que existen otras alternativas para intentar resolver el programa a la que se está actualmente probando.

Inicialmente, OP contiene los objetivos que forman la pregunta inicial del programa; BD está formada por todos los procedimientos del programa; EV está vacío, y AP es una pila vacía.

$$EJ_{inicial} \equiv \{BD \equiv \{\ldots,\ \{p/n,[cl_1,\ldots,cl_i]\},\ldots\},\ OP \equiv [pregunta],\ EV \equiv \emptyset,\ AP \equiv pila()\}$$

Se obtiene una solución al programa cuando OP se convierte en una lista vacía. Los vínculos de variables que caracterizan la solución se encuentran en EV. Pueden existir más soluciones si AP no está vacía. En el momento en que se encuentra una solución al programa y AP está vacía se han obtenido todas las soluciones posibles del programa.

$$EJ_{solución} \equiv \{BD,\ OP \equiv \emptyset,\ EV,\ AP\}$$

2.3.2 Operaciones

La ejecución de un programa consiste en la continua realización de intentos de inferencia. La

figura 2.1 muestra las operaciones básicas que forman el modelo de ejecución. A continuación se describen todas estas operaciones en función de cómo actualizan el estado de la ejecución. También se puede representar gráficamente la ejecución de un programa mediante un árbol OR. Un árbol OR es equivalente a un árbol SLD con la diferencia que los nodos corresponden únicamente a objetivos con más de una cláusula alternativa para intentar su satisfacción. Junto con la descripción de las operaciones se muestra la manera en que se construye y recorre este árbol OR durante la ejecución.

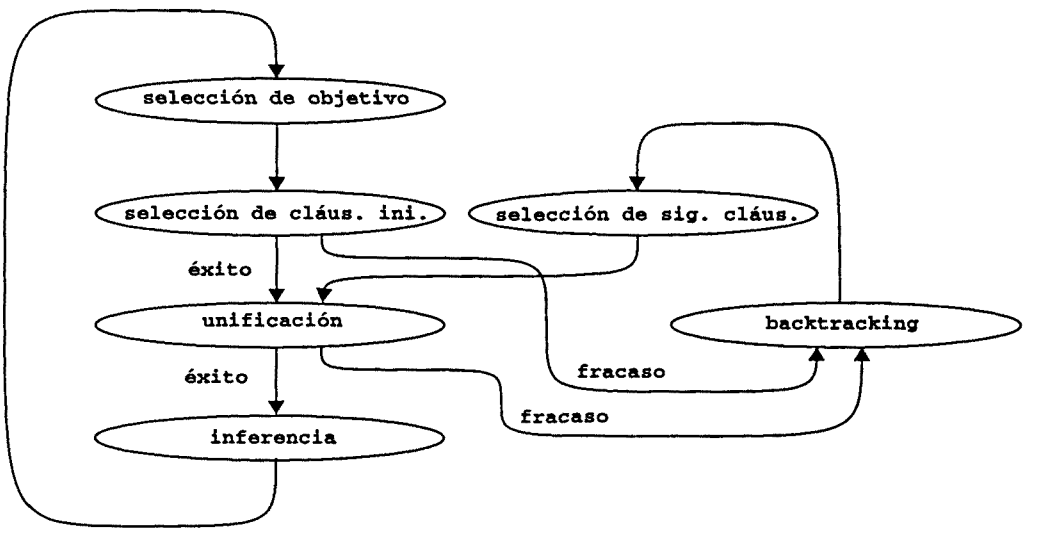


Figura 2.1: *Bucle básico de las operaciones del modelo convencional de ejecución de Prolog.*

Selección de Objetivo. Escoge un objetivo de OP para intentar su resolución. La regla de selección del objetivo es fija y escoge el primer objetivo empezando por la izquierda.

Selección de Cláusula Inicial. Escoge una cláusula existente en BD para proceder a la resolución del objetivo escogido anteriormente. En caso de que no exista ninguna cláusula (fracaso) pasa a realizarse el proceso de backtracking.

En caso de éxito, la ejecución convencional escoge, de entre todas las cláusulas que forman parte del procedimiento asociado al objetivo, la primera según el orden textual en que se han escrito. En caso que el procedimiento contenga más de una cláusula candidata a resolver el objetivo, se añade un nuevo elemento a la pila AP formado por el OP y EV actuales. Este punto de la ejecución, denominado también

punto de selección (cp), corresponde a un nodo del árbol OR que posee tantas ramas hijas como cláusulas tiene el procedimiento.

Unificación. Realiza el proceso de unificación entre los dos predicados formados por el objetivo elegido de OP y la cabecera de la cláusula elegida de DB. Este proceso de unificación se realiza para todos los argumentos del predicado. El resultado de esta operación indica si la unificación tiene éxito o fracasa. La salida en caso de éxito contiene el mgu, formado por los vínculos que permiten que los dos predicados coincidan. Si la unificación fracasa pasa a realizarse el proceso de backtracking.

Inferencia. Se realiza una inferencia cuando se ha podido completar una resolución SLD con éxito. Para poder satisfacer el objetivo que acaba de unificar con la cabecera de una cláusula, se deben resolver también los objetivos que forman el cuerpo de la cláusula. Por ello, se substituye el objetivo elegido de OP por el conjunto de objetivos del cuerpo de la cláusula elegida de BD. Por otra parte, todas los vínculos presentes en el mgu calculados durante la unificación se añaden a EV.

Backtracking. Se produce cuando el objetivo elegido de OP no puede resolverse con ninguna cláusula. Este punto de la ejecución corresponde a una hoja del árbol de búsqueda OR. En este caso, la ejecución intenta satisfacer los objetivos pendientes de OP presentes en la cima de AP. Para ello, restaura los elementos OP y EV del estado de la ejecución a partir de los elementos correspondientes existentes en la cima de la pila AP. Esta cima está asociada en al árbol OR al nodo más joven con ramas alternativas pendientes.

Selección de Siguiente Cláusula. Escoge la siguiente cláusula para resolver con el objetivo de más a la izquierda de OP. Al igual que en la selección de la cláusula inicial se sigue el orden textual de escritura de las cláusulas. En caso que no exista ninguna otra cláusula alternativa, el elemento de la cima de AP se elimina.

A modo de ejemplo, el pequeño programa Prolog de la figura 2.2a sirve para mostrar la generación de un árbol OR durante la ejecución de un programa y el valor de los elementos que forman el estado de la ejecución.

Este programa utiliza únicamente los aspectos declarativos puros que ofrece el lenguaje. El objetivo del programa es encontrar todos los abuelos de *tom*. La Base de Datos (BD) está formada por el procedimiento *grandparent/2*, que establece la regla que determina la relación abuelo-nieto y el procedimiento *parent/2*, que establece los hechos de la relación padre-hijo. En la figura 2.2b se muestra el árbol OR correspondiente a la ejecución de todo el programa

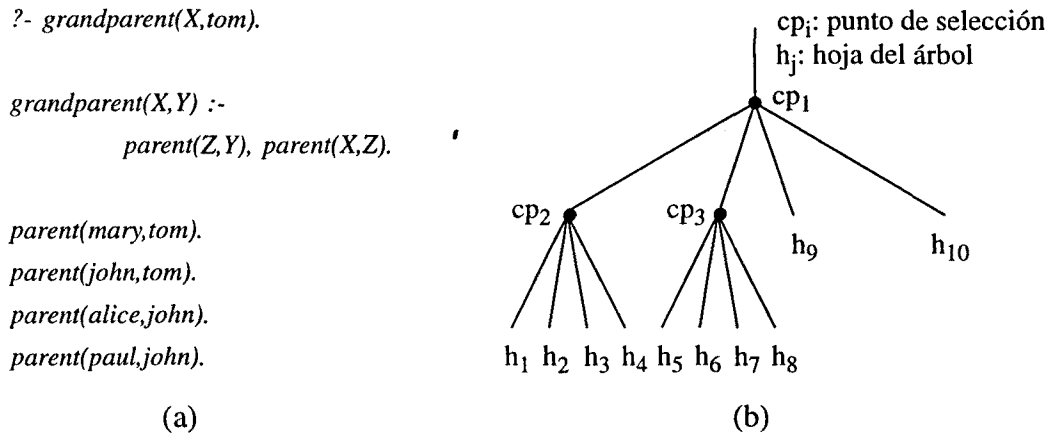


Figura 2.2: *Ejemplo de árbol OR obtenido con el modelo de ejecución convencional de Prolog:*
(a) Código Prolog.
(b) Árbol OR asociado.

(las dos soluciones del programa generan los vínculos *X/alice* y *X/paul*). Por otra parte, las operaciones más representativas y el estado de la ejecución desde el inicio del programa hasta encontrar la primera solución (hoja h_7) se reflejan en la figura 2.3. Obsérvese que las variables han sido renombradas ya que cada variable Prolog es visible únicamente en el ámbito de la cláusula en que aparece.

2.4 Modelo arquitectónico convencional de Prolog

Hay que remontarse a la propia concepción del lenguaje Prolog por el grupo liderado por Alain Colmerauer y Robert Kowalski, al principio de la década de los 70 [24], para describir los primeros sistemas orientados a la ejecución de programas Prolog. El primer sistema se basaba en una única fase de interpretación del programa. Este intérprete fue desarrollado por Philippe Roussel en 1972 y escrito en Algol-W. El primer sistema que se basó en dos fases para ejecutar un programa (una primera de compilación del programa Prolog a una representación equivalente, pero de más bajo nivel, y una segunda fase de interpretación realizada mediante software) fue conocido como Prolog DEC-10 [130] y desarrollado por David H. D. Warren en 1977.

Sin embargo, el auge en la realización de sistemas orientados a Prolog tuvo lugar en 1983, propiciado por el propio David H. D. Warren, a partir de la definición de un modelo arquitectónico cuya amplia divulgación generalizó la posibilidad de realizar el modelo de ejecución de Prolog de forma más eficiente (comparado con una interpretación del código Prolog)

ESTADO INICIAL:

$EJ_0 \equiv \{ OP \equiv [grandparent(X_1,tom)],$
$EV \equiv \emptyset,$
$AP \equiv pila() \}$

unificación$(grandparent(X_1,tom), parent(X_2,Y_2))$: *ÉXITO* $mgu \equiv \{X_2/X_1, Y_2/tom\}$; ***inferencia*** *1:*

$EJ_1 \equiv \{ OP \equiv [parent(Z_2,Y_2), parent(X_2,Z_2)],$
$EV \equiv \{X_2/X_1, Y_2/tom\},$
$AP \equiv pila() \}$

selección cláusula inicial *(creación* cp_1*):*

$AP \equiv pila(cp_1:\{OP \equiv [parent(Z_2,Y_2), parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom\}\})$

unificación$(parent(Z_2,tom), parent(mary,tom))$: *ÉXITO* $mgu \equiv \{Z_2/mary\}$; ***inferencia*** *2:*

$EJ_2 \equiv \{ OP \equiv [parent(X_2,Z_2)],$
$EV \equiv \{X_2/X_1, Y_2/tom, Z_2/mary\},$
$AP \equiv pila(cp_1:\{OP \equiv [parent(Z_2,Y_2), parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom\}\}) \}$

selección cláusula inicial *(creación* cp_2*):*

$AP \equiv pila(cp_1:\{OP \equiv [parent(Z_2,Y_2), parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom\}\},$
$cp_2:\{OP \equiv [parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom, Z_2/mary\}\})$

unificación$(parent(X_1,mary), parent(mary,tom))$: *FRACASO* (h_1); ***backtracking*** cp_2

unificación$(parent(X_1,mary), parent(john,tom))$: *FRACASO* (h_2); ***backtracking*** cp_2

unificación$(parent(X_1,mary), parent(alice,john))$: *FRACASO* (h_3); ***backtracking*** cp_2

unificación$(parent(X_1,mary), parent(paul,john))$: *FRACASO* (h_4); ***backtracking*** cp_2, *eliminación* cp_2

unificación$(parent(Z_1,tom), parent(john,tom))$: *ÉXITO* $mgu \equiv \{Z_2/john\}$; ***inferencia*** *3:*

$EJ_3 \equiv \{ OP \equiv [parent(X_2,Z_2)],$
$EV \equiv \{X_2/X_1, Y_2/tom, Z_2/john\},$
$AP \equiv pila(cp_1:\{OP \equiv [parent(Z_2,Y_2), parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom\}\}) \}$

selección cláusula inicial *(creación* cp_3*):*

$AP \equiv pila(cp_1:\{OP \equiv [parent(Z_2,Y_2), parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom\}\}$
$cp_3:\{OP \equiv [parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom, Z_2/john\}\})$

unificación$(parent(X_1,john), parent(mary,tom))$: *FRACASO* (h_5); ***backtracking*** *cp3*

unificación$(parent(X_1,john), parent(john,tom))$: *FRACASO* (h_6); ***backtracking*** *cp3*

unificación$(parent(X_1,john), parent(alice,john))$: *ÉXITO* $mgu \equiv \{X_1/alice\}$; ***inferencia*** *4:*

$EJ_4 \equiv \{ OP \equiv [],$
$EV \equiv \{X_1/alice\},$
$AP \equiv pila(cp_1:\{OP \equiv [parent(Z_2,Y_2), parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom\}\}$
$cp_3:\{OP \equiv [parent(X_2,Z_2)], EV \equiv \{X_2/X_1, Y_2/tom, Z_2/john\}\})$

SOLUCIÓN (h_7): $X_1/alice$

Figura 2.3: *Operaciones más representativas y el estado de la ejecución desde el inicio del programa grandparent hasta encontrar la primera solución.*

en cualquier sistema convencional secuencial basado en la arquitectura von Neumann. Este modelo arquitectónico es conocido como WAM (Máquina Abstracta de Warren) [131]. A continuación se resumen sus principales aspectos, suponiendo que el lector posee un cierto grado de conocimiento de este modelo arquitectónico. Una explicación detallada de la WAM puede encontrarse en [1]. El principal objetivo de esta descripción es poner de relieve las diferencias más importantes entre el modelo arquitectónico que realiza la ejecución convencional de Prolog y el modelo arquitectónico de Multipath, que será descrito en el capítulo 6.

2.4.1 Máquina Abstracta de Warren (WAM)

La arquitectura de la mayoría de los sistemas actuales orientados a Prolog se basa principalmente en los elementos arquitectónicos que presenta la WAM. En la figura 2.4 se muestra un esquema de los registros, zonas de memoria y estructuras de datos que forman la WAM. Las características concretas de cada sistema pueden considerarse como diferentes optimizaciones que se han propuesto para mejorar aspectos puntuales de la ejecución. La definición inicial de la WAM considera una realización mínima del modelo de ejecución convencional de Prolog en dos fases.

La primera fase consiste en la traducción del programa original en Prolog, que constituye la denominada Base de Datos (BD) del modelo de ejecución, a otra representación equivalente, que utiliza los elementos arquitectónicos propuestos en la WAM. La segunda fase consiste en la interpretación de este programa equivalente. Esta interpretación puede realizarse de diversas formas a su vez, que pueden clasificarse en interpretación soportada directamente por el hardware del sistema o soportada por software (emulación). En la siguiente sección, dedicada a la descripción de sistemas Prolog secuenciales, se citarán ejemplos de cada tipo.

2.4.1.1 Representación del estado de la ejecución

Los elementos del estado de la ejecución definidos por el modelo convencional

$$EJ \equiv \{BD,\ OP,\ EV,\ AP\}$$

pueden caracterizarse por los siguientes valores en un momento determinado de la ejecución, tal como se ha descrito en el apartado 2.3.1:

$$BD \equiv \{\ldots,\ \{p/n,\ [cl_1,\ldots,cl_i]\},\ \ldots\}$$
$$OP \equiv [gl_{111},\ gl_{112},\ldots,\ gl_{11k},\ gl_{12},\ \ldots,\ gl_{1j},\ gl_2,\ \ldots,\ gl_i]$$
$$EV \equiv \{\ldots,\ \{variable/término\},\ \ldots\}$$
$$AP \equiv pila(\ldots,\ \{OP_a,\ EV_a\},\ \ldots)$$

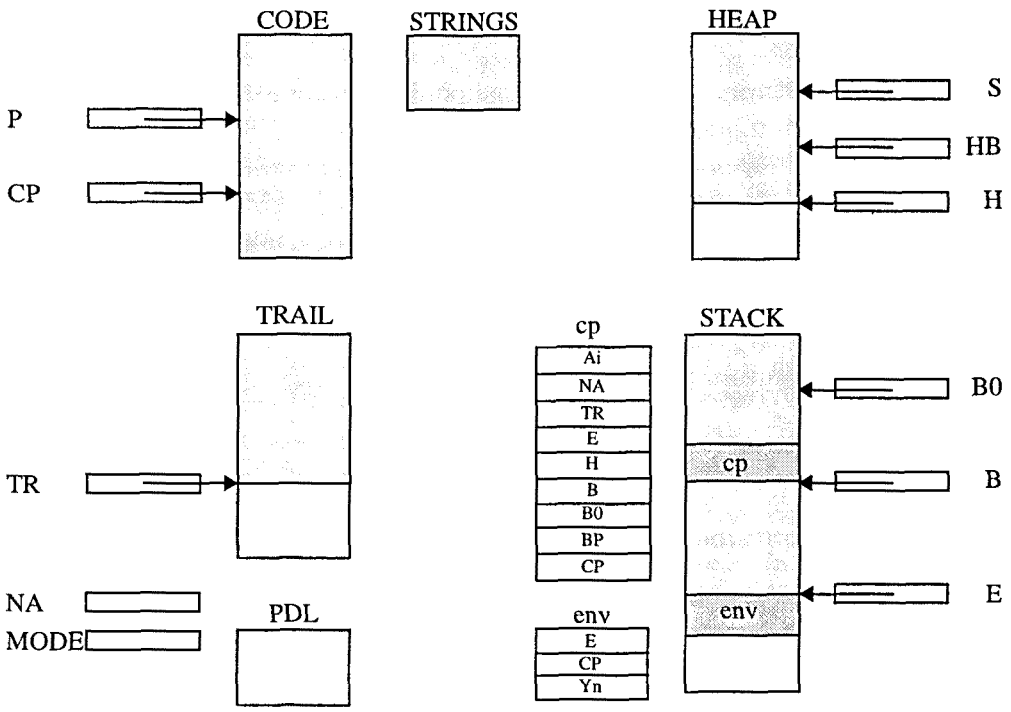


Figura 2.4: *Elementos arquitectónicos de la WAM.*

La **Base de Datos** (BD) contiene todos los procedimientos de que consta un programa. Se almacena en una zona de ***memoria*** denominada ***CODE***, y se comporta como una base de datos activa, es decir, que las instrucciones que representan a los procedimientos dirigen el control de flujo de la ejecución del programa. Cada procedimiento (p/n) se identifica por la dirección donde está ubicado en la memoria de código. La lista de cláusulas se identifica mediante ***instrucciones TRY, RETRY*** y ***TRUST***, cuyo argumento especifica la dirección de la cláusula asociada:

```
p/n:    TRY (p/n/cl1)
        ...
        RETRY (p/n/clj)
        ...
        TRUST (p/n/cli)
```

Cada cláusula (p/n/cl) está formada por instrucciones que realizan la unificación de la cabecera e instrucciones que activan la ejecución de los procedimientos asociados a los objetivos del cuerpo de la cláusula.

```
p/n/cl:    unifica(cabecera)
```

activa(gl1)

...

activa(glj)

La cabecera de una cláusula se especifica mediante ***instrucciones*** de unificación, denominadas ***GET***. Existen diferentes instrucciones GET según el tipo de datos del argumento de la cabecera. A su vez, para unificar los elementos de un término complejo (lista o estructura) se utilizan ***instrucciones UNIFY***. La especificación de todos los argumentos de la cabecera se realiza en el mismo orden de escritura de dichos argumentos en el programa Prolog.

Los predicados (u objetivos) del cuerpo de una cláusula se representan mediante ***instrucciones PUT*** y ***CALL***. Estas instrucciones tienen como fin la activación del procedimiento asociado al predicado, es decir, el procedimiento que posee el mismo nombre de predicado y aridad. El soporte utilizado para pasar los argumentos de un predicado a su procedimiento asociado corresponde a los ***registros Xi***. Las instrucciones PUT almacenan los términos Prolog correspondientes a los argumentos, y establecen que estos términos sean accesibles desde los registros Xi. La instrucción CALL indica la dirección del procedimiento asociado al predicado.

Los **Objetivos Pendientes** (OP) vienen representados a través de una lista en el modelo de ejecución. Esta lista de objetivos no existe explícitamente en la WAM, sino que se identifica a través de un ***registro P***, que apunta a la dirección de la memoria CODE que contiene el primer objetivo de la lista; los objetivos siguientes hasta finalizar los objetivos del cuerpo de la última cláusula que se ha resuelto se obtienen por secuenciamiento implícito en la posiciones de memoria siguientes; un ***registro CP*** apunta al primer objetivo de la continuación (en este caso, gl_{12}); y posteriores objetivos de continuación se pueden acceder a través de la dirección almacenada en una ***estructura de datos*** denominada ***entorno*** (en forma abreviada, ***env***). Estas estructuras de datos se encuentran almacenadas, de forma encadenada, en una zona de ***memoria*** denominada ***STACK***. Un ***registro E*** apunta al entorno más joven.

El **Entorno de Vínculos** (EV) contiene los vínculos asociados a todas las variables visibles en un momento de la ejecución. Las variables de un programa, creadas todas dinámicamente, se identifican por la dirección donde están almacenadas en memoria. A su vez, las variables pueden estar almacenadas en dos zonas de memoria diferentes: en los entornos de la memoria STACK y en otra zona de ***memoria*** denominada ***HEAP***. La primera posición libre del HEAP es apuntada por el ***registro H***. Los entornos contienen las variables permanentes (variables que son accedidas en más de un objetivo del cuerpo de la cláusula, o en la cabecera y en un objetivo que no es el primero del cuerpo; que se simbolizan como Yn) y el HEAP contiene las variables temporales (las que no son permanentes) y las inseguras (un caso particular de variables per-

manentes que es descrito posteriormente). El hecho de dividir las variables en dos tipos (permanentes y temporales) radica en la forma de acceso. Las variables temporales pueden ser accedidas a través de los registros Xi, mientras que las variables permanentes deben accederse mediante indirecciones a memoria, en concreto, a través de los entornos del STACK (los registros Xi pueden perder su contenido tras una llamada a un procedimiento).

Los vínculos de las variables tienen 4 posibles tipos de datos: variable libre, en la cual el contenido de la celda de memoria apunta a la misma dirección de memoria; constante atómica, cuyos códigos ASCII se almacenan en la ***memoria STRINGS***; constante numérica, que se almacena en la propia celda de la variable; y términos estructurados, cuyos elementos se almacenan en la memoria HEAP.

Las **Alternativas Pendientes** (AP) se almacenan en una pila, y cada una de ellas corresponde a un elemento {OP, EV}. En el modelo arquitectónico que proporciona la WAM, estos elementos se representan mediante una ***estructura de datos*** denominada ***punto de selección*** (en forma abreviada, ***cp***), que se almacenan en la memoria STACK. Todos los cp's están encadenados entre si, siendo la cima accesible a través de un ***registro B***. A continuación se resume la representación concreta de estos elementos. OP se obtiene a partir de los siguientes campos presentes en un cp: los campos Ai, que contienen referencias a los argumentos del objetivo de más a la izquierda de OP (el número de argumentos es NA); y los campos CP y E, que contienen el valor de los registros respectivos en el momento de la creación del cp, a partir de los cuales se accede a las direcciones de continuación de OP. Por otra parte, el EV existente en la creación del cp no se almacena de forma completa. Únicamente se almacena los punteros a la cima de cada una de las dos zonas de memoria que contienen las variables de un EV, junto con información para poder restaurar el estado existente en ese punto. La forma de restaurar el estado consiste en deshacer todas los vínculos obtenidos desde el momento de la creación del cp hasta el momento en que se inicia la operación de backtracking. Para ello se utiliza una zona de ***memoria***, gestionada en forma de pila y denominada ***TRAIL***, en la cual se almacenan las direcciones de variables condicionales (creadas anteriormente a un punto de selección) en el momento que se obtiene su vínculo. El ***registro TR*** apunta a la cima de esta memoria.

2.4.1.2 Realización de las operaciones

Las operaciones que forman el modelo de ejecución convencional se han descrito en la sección anterior. En este apartado se describe su realización teniendo en cuenta el modelo arquitectónico que proporciona la WAM.

La **Selección de Objetivo** consiste en ejecutar las instrucciones PUT apuntadas por el

registro P, que crean los términos Prolog correspondientes a cada argumento y permiten su acceso a través de los registros Xi. Por otra parte, la instrucción CALL determina la dirección del procedimiento asociado al objetivo, que corresponde a la dirección de la lista de cláusulas susceptibles de unificar.

La **Selección de Cláusula Inicial** se realiza mediante la ejecución de la instrucción TRY. Esta instrucción tiene como argumento la dirección de inicio del código asociado a la primera cláusula del procedimiento. Por otra parte, esta instrucción es la responsable de crear un nuevo punto de selección en la memoria STACK. Este punto de selección no contiene todas las cláusulas del procedimiento asociado al objetivo. En la WAM se realiza una operación de optimización, denominada *indexación*, consistente en guardar en un punto de selección únicamente aquellas cláusulas de un procedimiento que pueden unificar con el tipo de datos que posee el primer argumento del objetivo elegido. Esta optimización puede catalogarse como una técnica de disminución del grado de indeterminismo de un programa.

La operación de **Unificación** se realiza mediante la ejecución de las instrucciones GET específicas para unificar los argumentos del objetivo con los argumentos de la cabecera de la cláusula. El algoritmo de unificación general entre dos términos Prolog [77] utiliza otra zona de ***memoria***, denominada ***PDL***, cuando se unifican dos términos estructurados. Esta memoria se gestiona en forma de pila y contiene los elementos de los dos términos estructurados que faltan por unificar.

La creación de las estructuras de datos denominadas entornos tiene lugar antes de la unificación de la cabecera, y su eliminación se produce antes de ejecutar el último objetivo de la cláusula (*last call optimization* o LCO). Esta optimización tiene la finalidad de reducir el espacio de memoria necesario para gestionar los procedimientos. Sin embargo, estas estructuras de datos podrían ser accedidas durante la ejecución del último objetivo. Las variables inseguras son aquellas que permanecen sin vincular en un entorno antes de eliminarlo. Cuando sucede esta situación, dichas variables se copian al HEAP.

La operación de **Inferencia** almacena las vínculos obtenidos durante la unificación en EV, elimina el objetivo de OP, e introduce en su lugar los objetivos del cuerpo de la cláusula. En la WAM, los vínculos se almacenan incrementalmente mientras se realiza la unificación. La resolución del objetivo se consigue en el momento que el registro P apunta al primer objetivo del cuerpo de la cláusula, mientras que la ejecución de la instrucción CALL modifica el registro CP para que pase a apuntar al segundo objetivo existente en el OP inicial.

La operación de **Backtracking** debe restaurar el estado de la ejecución existente en el

momento de crear el punto de selección más joven, que es apuntado por el registro B. Para restaurar OP, se actualizan los registros Xi, CP y E a partir de los campos respectivos del punto de selección. Para restaurar EV, se actualizan las cimas de las dos zonas de memoria que contienen las variables del EV (registros H y E), y se deshacen todos los vínculos realizados desde el momento de crear el punto de selección (guardado en el campo TR) hasta que se ha producido el fracaso en una unificación (especificado por el contenido actual del registro TR). Las direcciones de estas variables se encuentran en la memoria TRAIL.

La **Selección de Siguiente Cláusula** se realiza consultando la dirección de inicio de la lista de cláusulas pendientes de unificar, que está almacenada en el campo BP del punto de selección más joven. Si esta lista de cláusulas pendientes contiene la instrucción RETRY significa que existe más de una cláusula pendiente. En este caso, se modifica la dirección de la lista de cláusulas pendientes en el punto de selección, y se obtiene la dirección de la siguiente cláusula a unificar (contenida en la propia instrucción). Cuando la cláusula que se selecciona es la última del procedimiento, indicado por una instrucción TRUST, se elimina el punto de selección de la memoria STACK.

2.5 Realizaciones secuenciales de Prolog

Desde el desarrollo de la WAM en 1983 han aparecido una gran cantidad de sistemas orientados a la ejecución de Prolog. Una completa guía de la mayor parte de estos sistemas puede encontrarse en la Prolog Resource Guide [63], que es actualizada dinámicamente, y en el estupendo trabajo de recopilación de Peter van Roy [129]. En este apartado se describen brevemente aquellos sistemas que han tenido una difusión más extensa, incluidos los que no están basados estrictamente en la WAM. Esta descripción se divide en dos categorías: realizaciones software y realizaciones hardware.

2.5.1 Realizaciones software

Las realizaciones software incluyen a aquellos sistemas en los cuales se realiza una interpretación de Prolog o bien están basados en la WAM. En este último caso, su interpretación se realiza emulando sus instrucciones en otro lenguaje ya soportado por el sistema (normalmente C) o traduciendo las instrucciones WAM al lenguaje máquina del procesador que posee el sistema (compilador a código nativo).

MProlog. Fue el primer sistema Prolog comercial. Comenzó a desarrollarse en Hungría en 1978 impulsado principalmente por Péter Szeredi [8] y completado en 1982 en

SZKI, un centro de investigación húngaro [39]. Este sistema posee un compilador a código máquina nativo, soporta declaraciones de modos (indica si en el momento de iniciar un procedimiento sus argumentos deben corresponder a variable libres o a términos instanciados) y realiza recolección de basura en las memorias CODE y STRINGS pero no de las restantes pilas.

IF/Prolog y **SNI-Prolog**. La primera versión de IF/Prolog fue un intérprete desarrollado en 1983. No existen publicaciones de esta realización. La versión basada en un compilador fue divulgada en 1985. Posteriormente, sirvió de base para una versión más completa, denominada SNI-Prolog, conforme con el standard ISO de Prolog [57]. Ambos sistemas permiten la utilización de los lenguajes Prolog y C, y realizan la recolección de basura en todas las zonas de memoria de la WAM.

MU-Prolog y **NU-Prolog**. Ambos sistemas fueron desarrollados en la Universidad de Melbourne [86]. NU-Prolog está basado en un intérprete mientras que MU-Prolog realiza una emulación de las instrucciones WAM en C. La principal característica de estos sistemas es la introducción de nuevos predicados en la WAM: negación lógica, cuantificadores, if-then-else, desigualdad, etc. Por otro lado, NU-Prolog ha servido de base para experimentos relacionados con paralelismo [87], bases de datos [94] y entornos de programación [88].

Quintus Prolog. Es el primer sistema ampliamente difundido orientado al lenguaje Prolog. La primera versión data de 1985 y su promotor más destacado fue David H. D. Warren. Se basa en un eficiente compilador que genera código muy compacto. Además permite llamadas entre Prolog y C o viceversa, y posee un amplio conjunto de librerías.

SEPIA y **ECLiPSe**. Estos sistemas han sido desarrollados por ECRC en Alemania, un centro de investigación con realizaciones software y hardware de Prolog tanto secuenciales como paralelas. SEPIA salió a la luz pública en 1988 [79] mientras que su más reciente sistema, ECLiPSe, incorpora la capacidad de retrasar la evaluación de objetivos hasta que las variables de sus argumentos poseen unos tipos de datos predeterminados (*delaying* de objetivos) [81]. Sin embargo, la característica más notable de este sistema es la presencia de un compilador incremental que además es el más rápido de todos los sistemas existentes.

SB-Prolog y **XSB**. SB-Prolog es un emulador basado en la WAM desarrollado en SUNY (Universidad del Estado de Nueva York) por David S. Warren. Surgió en 1986 y

se ha convertido en bastante popular debido a ser un software de dominio público, que sirvió de base para muchos [30] trabajos de investigación. XSB es la versión más reciente de SB-Prolog.

SICStus Prolog. Desarrollado en el instituto sueco SICS constituye tal vez el sistema más popular de todos los mencionados: es robusto, barato, rápido, compatible con el standard de Prolog y ha sido portado a numerosas máquinas [16]. La primera versión de 1986 era un sistema que emulaba WAM con el intérprete escrito en C. Las versiones actuales ya se basan en compiladores a código nativo.

Aquarius. Enmarcado dentro del proyecto Aquarius de la Universidad de California (Berkeley). Constituye el sistema más rápido de todos los existentes para la ejecución del código objeto generado [128]. El compilador posee dos etapas: la primera realiza un análisis global del programa para determinar información sobre los modos y tipos de entrada de los argumentos de cada uno de los objetivos, y genera código de una máquina abstracta denominada BAM (modificación de la WAM); la segunda etapa realiza una expansión de cada una de las instrucciones BAM a código nativo.

2.5.2 Realizaciones hardware

Dentro de las realizaciones hardware se incluyen aquellos sistemas en los cuales existe un diseño específico del procesador encargado de interpretar las instrucciones WAM. A continuación se citan los sistemas más relevantes:

PSI y **CHI**. El proyecto FGCS (Fifth Generation Computer Systems) [40] desarrollado en el instituto japonés de ICOT promocionó el diseño y construcción de un gran número de máquinas Prolog tanto secuenciales como paralelas. Se construyeron dos series de máquinas paralelas: PSI y CHI. En concreto se realizaron tres máquinas PSI (Personal Sequential Inference) todas ellas microprogramadas horizontalmente, con palabras de 40 bits, 8 de ellos para almacenar la etiqueta del tipo de datos: PSI-I, desarrollada antes que la WAM (1983) [113], con frecuencia de reloj de 5 MHz; PSI-II construida en 1986 y a 6.45 Mhz [89]; y PSI-III (1990) a 15 Mhz. Las dos últimas máquinas ya están basadas en la WAM. Con respecto a las máquinas CHI, se realizaron dos versiones [44], aunque no fueron tan populares como las PSI.

IPP. Integrated Prolog Processor desarrollado en Hitachi. Interpreta las instrucciones WAM mediante microinstrucciones con un tiempo de ciclo de 23 ns. [70]. El coste hardware adicional para el soporte de Prolog se estimó en un 3%.

KCM. El trabajo a nivel de arquitectura realizado en ECRC culminó en la máquina KCM (Knowledge Crunching Machine) en 1988 [90]. KCM es un coprocesador dedicado a Prolog con palabras de 64 bits (32 para la etiqueta) y frecuencia de reloj a 12,5 MHz., que incrementa en un 60% el rendimiento del procesador.

Pegasus. Proyecto desarrollado en Mitsubishi en el periodo 1987-90 en el cual se realizaron 3 chips. El último de ellos, a una frecuencia de 10 Mhz, alcanza el mismo rendimiento que KCM [138].

Aquarius. El proyecto Aquarius en Berkeley es el que ha propiciado un mayor número de diseños hardware secuenciales y paralelos. Entre los secuenciales cabe citar: PLM (1983-85) [128], VLSI-PLM (1985-89) [104] y VLSI-BAM (1988-91) [53]. Este último es el que proporciona un mayor rendimiento: el soporte a Prolog supone un 10,6% del área del chip y proporciona un incremento de velocidad del 70%.

2.6 Paralelismo en Prolog

En esta sección se pretende dar una descripción general de las diversas fuentes de paralelismo que se pueden explotar en la ejecución de un programa Prolog. Es importante destacar que estas fuentes de paralelismo son implícitas, teniendo en cuenta la semántica declarativa que posee el lenguaje Prolog [18]. Hay tres tipos básicos de paralelismo explotados en los programas lógicos: paralelismo O, paralelismo Y y paralelismo de unificación.

2.6.1 Paralelismo O

Cuando el procedimiento asociado al objetivo elegido de OP posee varias cláusulas candidatas a realizar la unificación, la ejecución puede proseguir en paralelo intentando resolver, en cada uno de los procesos paralelos, los objetivos pendientes en OP con una de las cláusulas candidatas a unificar.

Conceptualmente, la idea básica del paralelismo O es la más simple de los tres tipos. Una de las ventajas principales es la independencia de los procesos paralelos. El espacio de búsqueda puede ser dividido en subárboles totalmente disjuntos entre ellos. Esta característica ha hecho del paralelismo O uno de los más utilizados en la realización de sistemas paralelos orientados a la programación lógica.

Sin embargo, este modelo paralelo no está exento de problemas. Debe establecerse un mecanismo que gestione los posibles vínculos múltiples que una misma variable puede poseer.

Durante la exploración paralela del árbol de búsqueda, se está recorriendo de forma concurrente más de un camino del árbol, que corresponden a diferentes satisfacciones de la pregunta inicial. Estos caminos pueden instanciar la misma variable a diferentes pero independientes vínculos. El mecanismo de gestión debe definir una forma de representación de los múltiples vínculos y una forma de reconocer el ámbito de cada vínculo [29]. Estos diferentes vínculos no contradicen la propiedad de asignación única que poseen las variables Prolog, ya que cada vínculo afecta únicamente a un subárbol de forma independiente al resto del árbol de búsqueda.

Los mecanismos de gestión del conflicto de instanciación múltiple más eficientes son: Binding Arrays [133] y Hash Windows [9]. Una completa recopilación de todas las técnicas puede encontrarse en [20].

2.6.2 Paralelismo Y

La idea básica de este tipo implícito de paralelismo surge de la propia semántica declarativa que posee la resolución SLD. Para resolver un objetivo se requiere unificarlo con la cabecera de una cláusula y satisfacer los objetivos del cuerpo de la cláusula. El paralelismo Y aparece cuando la satisfacción de los objetivos del cuerpo de una cláusula se realiza de forma paralela.

Sin embargo, esta forma de paralelismo también presenta sus inconvenientes. En este caso, tras la ejecución paralela de los objetivos de la cláusula deben hacerse coherentes los vínculos de las variables compartidas en estos objetivos. Las diferentes formas de tratar este problema clasifican el paralelismo Y en cuatro tipos:

Paralelismo Y no restringido. Este tipo es el más costoso y, a su vez, el más simple. Todos los objetivos se ejecutan en paralelo. Una vez finalizada la ejecución paralela, existe un nivel adicional de unificación, denominado *join*, que soluciona los conflictos que puedan presentar las variables compartidas. En la práctica, la técnica de join se ha mostrado como ineficiente por su alta complejidad [25].

Paralelismo Y restringido o independiente. En este caso, se evita la ejecución paralela de los objetivos que poseen variables compartidas. Existen distintos métodos para detectar la independencia de los objetivos que se diferencian en el momento que realizan la detección: en tiempo de compilación [17], en tiempo de ejecución [72] [25], o una forma híbrida [32].

Paralelismo Y determinista. En este caso, se posibilita la ejecución paralela de aquellos objetivos con variables compartidas que sean deterministas, es decir, que únicamente puedan tener una solución. Aunque los objetivos sean dependientes, no debe reali-

zarse la unificación adicional ya que únicamente se debe comprobar si el vínculo es coherente con el que se haya obtenido durante la ejecución de otros objetivos [98]. Esta comprobación se realiza en el mismo momento de vincular una variable.

Paralelismo Y segmentado o de cadena. Corresponde al modelo de proceso productor/ consumidor. Mientras un objetivo del cuerpo de una cláusula está generando un elemento, otro objetivo del cuerpo de la cláusula puede estar consumiendo un elemento anterior. Este tipo de paralelismo puede ser explotado en una familia de lenguajes lógicos, denominada *Commited Choice Languages* [100], que fuerza a tener una única solución en los procedimientos indeterministas. Parlog [21], Concurrent Prolog [99] y Guarded Horn Clause [126] son sus tres lenguajes más representativos.

2.6.3 Paralelismo de unificación

Corresponde a la posibilidad de unificar en paralelo los argumentos de un objetivo con los argumentos de la cabecera de la cláusula durante el intento de realizar una resolución SLD.

Este tipo de paralelismo es de granularidad muy fina y se ha realizado en relativamente pocos sistemas: Parallel Unification Processor (PUP) [19] y Parallel Unification Machine (PUM) [101], como más representativos.

2.7 Realizaciones paralelas

En este apartado se citan brevemente los sistemas más representativos que explotan el paralelismo O, el paralelismo Y, o una combinación de ambos.

PepSys y **ElipSys.** Ambos sistemas han sido realizados en ECRC. PepSys fue el pionero y explota paralelismo O y paralelismo Y independiente [135]. El problema de vínculos múltiples que aparece en el paralelismo O se soluciona con la técnica de Hash Windows y el problema de realizar el producto cruzado de todas las soluciones coherentes que aparecen al explotar el paralelismo Y se soluciona con las denominadas *join cells*. ElipSys fue su sucesor y sus características básicas fueron la renuncia de la gestión de la técnica de Hash Windows por la de Binding Arrays y la incorporación de restricciones en el lenguaje, basado en CHIP [35]. Elipsys constituyó la base del proyecto europeo EP2025, que fructificó en la máquina EDS [47]. La investigación llevada a cabo de forma previa a la definición de Multipath se encuadra en este contexto y, en concreto, se han desarrollado técnicas de planifica-

ción de trabajo para la gestión del paralelismo O [41][117][118][119].

AURORA. Sistema basado en el modelo SRI [132], que explota paralelismo O con utilización de la técnica de Binding Arrays [76]. Una de las principales cuestiones a tratar en los sistemas con paralelismo O es la planificación de la carga en los distintos procesadores. Es conocido el sistema Aurora por los distintos planificadores de trabajo que han sido desarrollados: Manchester [15], Bristol [7], Wavefront [12] y Argonne [14]. Tanto en un entorno secuencial como en un entorno paralelo, Aurora se comporta mejor que PepSys [112].

MUSE. Sistema con explotación de paralelismo O [4]. El modelo paralelo se caracteriza por realizar una copia de todas las pilas que constituyen el estado de la ejecución cuando un procesador decide coger trabajo existente en otro procesador. Se ha realizado en varios sistemas multiprocesadores comerciales con speed-ups prácticamente iguales a los de Aurora [5].

α-PROLOG. Desarrollado por Manuel V. Hermenegildo, explota el paralelismo Y independiente, tanto en su versión estricta (dos objetivos no pueden compartir variables para ejecutarse en paralelo) o no estricta (aunque compartan variables, pueden ejecutarse en paralelo si el árbol de búsqueda no varía con respecto a su ejecución secuencial). Este sistema está basado en el modelo RAP-WAM [49] y la determinación de los objetivos candidatos a ejecutarse en paralelo puede ser realizada por el compilador o mediante anotaciones del programador. El modelo se realizó en un multiprocesador Sequent Symmetry con 12 procesadores [50].

ANDORRA-I. El proyecto que ha dado lugar al sistema Andorra-I está liderado por David H. D. Warren y está principalmente influenciado por el sistema Aurora. Andorra-I es un sistema paralelo de programación lógica que explota paralelismo Y determinista junto con paralelismo O. El compilador realiza un análisis global del programa [98] en el cual se opta por ejecutar en primer lugar los objetivos deterministas (sólo pueden generar una solución y nunca se realiza backtracking sobre estos objetivos) y, en caso de no existir este tipo de objetivos, se selecciona un objetivo no determinista y se exploran todas sus alternativas en paralelo. El compilador debe realizar la tarea de decidir los objetivos que son candidatos a ejecutarse en primer lugar y aquellos que no pueden ejecutarse hasta que no estén suficientemente instanciados sus argumentos. Se ha realizado en un Sequent Symmetry con 12 procesadores [137].

3

SISTEMA MULTIPATH

En este capítulo se presenta el sistema informático Multipath, sistema que está orientado a la ejecución de programas lógicos escritos en lenguaje Prolog. El sistema Multipath exhibe un mecanismo de ejecución que difiere de la ejecución convencional de programas Prolog, fundamentando su estrategia básica en la gestión eficiente del indeterminismo que puede estar presente en un programa.

En esta introducción del sistema Multipath se define el concepto de indeterminismo, la forma en que el lenguaje Prolog es capaz de expresar este indeterminismo y cómo se gestiona en el modelo de ejecución convencional de Prolog. Posteriormente, se presenta informalmente la gestión del indeterminismo realizada en el sistema Multipath. Esta gestión se divide lógicamente en dos funciones: por un lado, intentar reducir el indeterminismo que exhibe un programa y, por otra parte, recorrer de forma más eficiente las alternativas indeterministas remanentes en el programa. Durante la presentación del sistema Multipath se hace especial relevancia en las funciones concretas que se introducen y las innovaciones que aportan respecto a los trabajos presentados en la literatura directamente relacionados.

Después de esta presentación básica del sistema, se enumeran las características del sistema utilizando los tres niveles de descripción de un sistema comentados en el capítulo introductorio: modelo de ejecución, modelo arquitectónico y realización concreta de un sistema. La definición completa del sistema en cada uno de estos niveles se realiza en los capítulos posteriores: el Modelo de Ejecución de Multipath (MEM) en el capítulo 4; el Modelo Arquitectónico de Multipath (MAM) en los capítulos 5 y 6, una realización secuencial de Multipath (SMAM) en el capítulo 7 y una realización paralela (PMAM) en el capítulo 8.

3.1 Indeterminismo en Prolog

La motivación principal en el diseño de Multipath hace referencia a la capacidad de gestionar de forma más eficiente el indeterminismo que permite establecer el lenguaje Prolog. Este **indeterminismo** surge en el momento en que se declara más de una cláusula candidata a satisfacer un objetivo del programa (ver figura 3.1).

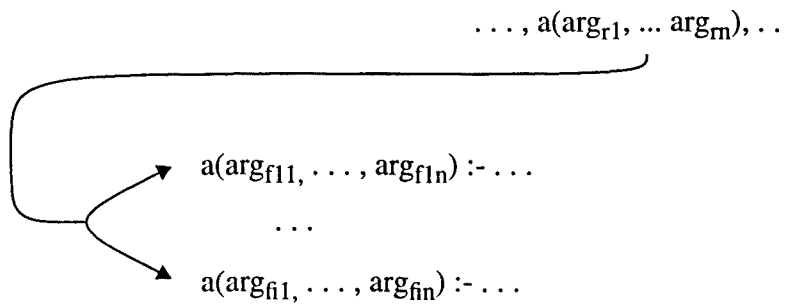


Figura 3.1: *Indeterminismo en Prolog al poder intentar unificar un objetivo con más de una cláusula.*

Nótese que el intento de satisfacer un objetivo con una cláusula que posteriormente fracasa en la unificación de algún argumento o en la satisfacción de algún objetivo del cuerpo de la cláusula constituye una penalización en cuanto a tiempo de ejecución: se consume un cierto tiempo en la ejecución de unas acciones que no contribuyen al cálculo de la solución o soluciones que tiene el programa. Por ello, es importante poder identificar lo antes posible las cláusulas que no sirven para satisfacer un objetivo. Este campo de trabajo está enfocado a **reducir el indeterminismo** que presenta un programa. Touati y Despain mostraron que alrededor del 40% de las operaciones relacionadas con la gestión del indeterminismo pueden ser eliminadas a través de optimizaciones [116].

Por otra parte, la exploración de un árbol de búsqueda obtenido por la presencia de indeterminismo en un programa puede efectuarse de dos formas completamente antagónicas [71]: en **profundidad** y en **anchura.**

A continuación se resumen las características básicas del modelo de ejecución convencional de Prolog en los aspectos referentes a disminución del indeterminismo y de recorrido de un árbol de búsqueda. El propósito es introducir los mecanismos utilizados en el modelo convencional de Prolog en la gestión de indeterminismo para, posteriormente, en la sección 3.2 presentar los mecanismos propuestos en Multipath.

3.1.1 Reducción del indeterminismo

En primer lugar, el modelo de ejecución convencional de Prolog [57] no considera ningún mecanismo para reducir el indeterminismo: las cláusulas candidatas a unificar con un objetivo son todas aquellas que tienen el mismo nombre de predicado y aridad.

Sin embargo, la Máquina Abstracta de Warren, considerada como el modelo base en las realizaciones del modelo de ejecución convencional de Prolog, incluye una función de **indexación** por procedimiento. Esta función de indexación permite, en función del tipo de datos y valor del primer argumento de un objetivo, descartar cláusulas a considerar en la unificación de dicho objetivo ya que fracasarían posteriormente durante la unificación de los argumentos.

3.1.2 Exploración en profundidad

La característica básica de la exploración en profundidad consiste en que existe únicamente un camino activo del árbol búsqueda durante la ejecución del programa. Este camino activo se mantiene hasta el momento en que se llega a una hoja del árbol, que corresponde a encontrar una solución del programa o a un fracaso de una unificación. En este último caso, o bien por desear obtener otra solución del programa, la ejecución continúa con la exploración de otro único camino del árbol de búsqueda tras realizar la operación de backtracking. Este es el recorrido que realiza la ejecución convencional de Prolog, conocido también como exploración del primero en profundidad (*depth-first search*).

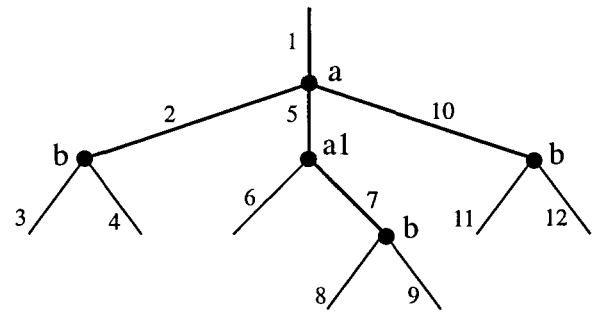
Durante la ejecución de un programa es posible que alguno de sus objetivos posea más de una solución (o manera de satisfacerlo). La **solución de un objetivo** se obtiene cuando su unificación con la cabecera de una cláusula tiene éxito y los objetivos del cuerpo de la cláusula se han satisfecho. Este concepto de solución está asociado a cada uno de los objetivos del programa, incluyendo al objetivo inicial del programa (?-), cuya solución corresponde a encon-

```
?- a,b.

a.          a1 :- fail.
a :- a1.    a1.
a.

b.
b.
```

(a)



(b)

Exploración en profundidad:
El orden numeración de las ramas corresponde a este tipo de exploración.
Exploración en anchura a nivel de puntos de selección:
1, 2, 5, 10, 3, 4, 6, 7, 11, 12, 8, 9.

(c)

Figura 3.2: *Orden de un recorrido en profundidad o en anchura de las ramas de un árbol de búsqueda:*
(a) Programa base, donde se muestran únicamente los aspectos relacionados con el indeterminismo.
(b) Árbol de búsqueda OR asociado.
(c) Orden de una exploración en profundidad y en anchura a nivel de puntos de selección.

trar una forma de satisfacer el programa.

Una exploración en profundidad ocasiona la repetición de la ejecución de instrucciones de control y de datos comunes en los caminos asociados a las soluciones de un objetivo mientras se ejecuta la continuación de dicho objetivo. La **continuación de un objetivo** incluye a todos los siguientes objetivos que se requiere satisfacer después de solucionarlo para encontrar una solución del programa.

La pregunta del programa de la figura 3.2a consta de los objetivos *a/0* y *b/0*. La continuación del objetivo *a/0* es una lista de objetivos formada, en este caso, por el objetivo *b/0*. Obsérvese en la figura 3.2b como la exploración en profundidad requiere forzosamente repetir la satisfacción del objetivo *b/0* para cada una de las soluciones del objetivo *a/0*.

3.1.3 Exploración en anchura

En este tipo de exploración se permite recorrer de forma simultánea un número de caminos del árbol de búsqueda mayor que uno. Una exploración total en anchura no requiere la operación

de backtracking, pero introduce una nueva operación que permite la creación y gestión dinámica de nuevos caminos del árbol de búsqueda.

La exploración en anchura puede realizarse de varias formas según el momento durante el recorrido de un camino del árbol en que se decide pasar a recorrer otro camino diferente. Se puede clasificar la exploración en anchura según este momento en que se cambia de camino en el árbol de búsqueda: (i) después de finalizar cada una de las operaciones de que consta una rama; (ii) en el momento de llegar a un punto de selección; o (iii) en el momento de encontrar una solución de un objetivo interno del programa.

En un contexto teórico de recorrido de árboles, la forma más típica de exploración en anchura consiste en cambiar de camino en cada nodo o punto de selección del árbol (ver figura 3.2c), de forma que hasta que no ha finalizado la ejecución de todas las ramas de un mismo nivel del árbol no se pasa al siguiente nivel. Esta forma de exploración corresponde con la segunda posibilidad comentada en el párrafo anterior y también se denomina exploración del primero en anchura (*breadth-first search*).

El modelo convencional de Prolog no permite ningún tipo de exploración en anchura, por lo que debe ser el programador quien la gestione de forma explícita mediante predicados predefinidos (del tipo *bagof/3*), que permiten calcular todas las soluciones de un objetivo. Una vez obtenidas todas las soluciones, el programador debe tratar de forma explícita cada una de estas soluciones. Esta gestión del indeterminismo presenta los siguientes inconvenientes: se reduce la visión declarativa del programa, al involucrar al programador en tareas de control de la ejecución; complica la gestión individual de las soluciones; y puede provocar problemas de falta de memoria al obligar a calcular todas las soluciones de objetivos indeterministas.

3.2 Gestión del indeterminismo en Multipath

La gestión del indeterminismo que se efectúa en Multipath intenta reducir el indeterminismo que presenta un programa y realizar un recorrido del indeterminismo remanente de forma más eficiente que en el modelo de ejecución convencional de Prolog. A continuación se describen las ideas propuestas en cada una de estas dos funciones.

3.2.1 Reducción del indeterminismo

La disminución del indeterminismo se consigue al reducir el número de cláusulas candidatas a realizar la unificación de un objetivo del programa. La característica común que engloba a las cláusulas que son eliminadas como candidatas a la unificación consiste en que no conducen a

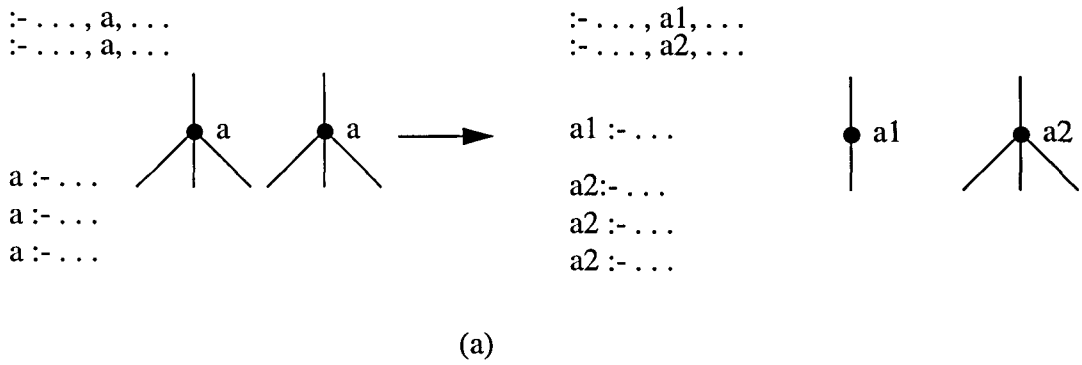
encontrar una solución del programa ya que, posteriormente, en algún punto del recorrido del mismo camino del árbol de búsqueda se produce un fracaso. Existen tres momentos en que se pueden reducir las cláusulas candidatas a unificar con un objetivo del programa: (i) antes del tiempo de ejecución (de forma estática); (ii) en tiempo de ejecución (de forma dinámica), en el momento de seleccionar un objetivo para intentar su resolución; y (iii) también en tiempo de ejecución, mientras se están resolviendo los objetivos del cuerpo de una cláusula y existen otras cláusulas alternativas para probar.

La identificación de estas cláusulas que no deben ser consideradas candidatas en la unificación de un objetivo se basa en la realización de un análisis global del comportamiento del programa. El proceso que realiza este análisis se denomina interpretación abstracta y será descrito con detalle en el capítulo 4, a nivel de modelo de ejecución, y en el capítulo 5, a nivel de modelo arquitectónico. Su idea fundamental es calcular información característica de la ejecución del programa de forma que es posible: (i) determinar de forma estática las cláusulas de un procedimiento que fracasarían en la ejecución; (ii) determinar las condiciones, a evaluar en tiempo de ejecución en el momento de activar un objetivo, que disminuyen el número de cláusulas candidatas en la resolución de dicho objetivo; o (iii) localizar objetivos en el cuerpo de una cláusula que, una vez satisfechos, aseguran que el resto de cláusulas del mismo punto de selección fracasarían en la resolución de su objetivo asociado. La información característica a que se hace referencia anteriormente consiste en calcular el tipo de datos de los términos Prolog utilizados como argumentos, tanto en los objetivos como en las cláusulas, y en calcular las condiciones de éxito de cada cláusula en función de los términos Prolog involucrados.

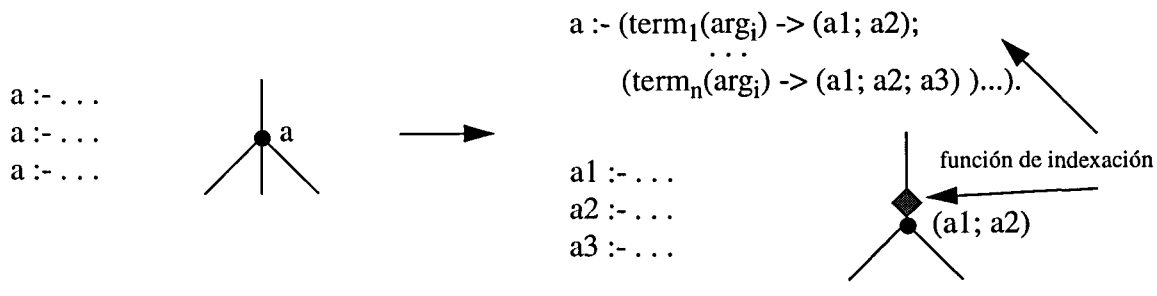
La reducción del indeterminismo que realiza el sistema Multipath es una aplicación particular dentro del campo de transformaciones automáticas del código fuente de un programa. A continuación se describe con mayor detalle las tres posibilidades de reducción de indeterminismo.

3.2.1.1 Determinación de puntos de entrada a procedimientos.

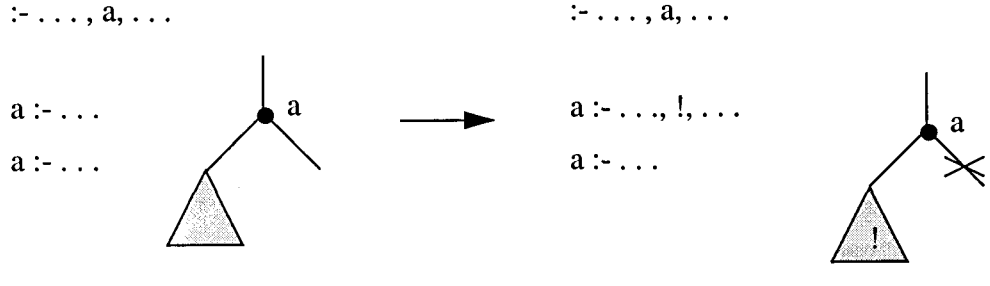
En primer lugar, se define la **activación de un objetivo** del programa como la acción de empezar el intento de satisfacer dicho objetivo mediante alguna de las cláusulas del programa que son consideradas candidatas. Por otra parte, una de las tareas de la interpretación abstracta es detectar, para cada activación de un objetivo, cláusulas de un procedimiento que de forma segura provocarían un fracaso en tiempo de ejecución, o interpretación real, que es la nomenclatura utilizada en el sistema Multipath. Estas cláusulas no son consideradas como candidatas en el momento de iniciar la activación de un objetivo.



(a)



(b)



(c)

Figura 3.3: *Tres posibilidades de reducción del indeterminismo en Multipath:*

(a) Determinación de puntos de entrada a procedimientos.

(b) Establecimiento de una función de indexación.

(c) Introducción de operadores de poda.

Por todo ello, la activación de un objetivo en Multipath no está ligada con el procedimiento compuesto por todas las cláusulas con el mismo nombre y aridad sino con los denominados **puntos de entrada** a un procedimiento (ver figura 3.3a). Un punto de entrada contiene únicamente aquellas cláusulas de un procedimiento que la interpretación abstracta considera como candidatas para resolver los objetivos asociados a dicho punto de entrada. Un procedimiento puede disponer de distintos puntos de entrada y, a su vez, pueden existir varios objetivos del programa, con el mismo nombre de predicado y aridad, asociados al mismo punto de entrada.

Como ejemplo genérico de transformación de un programa Prolog en el contexto de detección de puntos de entrada a un procedimiento, considérese los dos objetivos siguientes y su procedimiento asociado:

```
:- ..., g(t1, ..., tn), ...
:- ..., g(v1, ..., vn), ...

g(a11, ..., a1n) :- ...
g(a21, ..., a2n) :- ...
g(a31, ..., a3n) :- ...
```

Supóngase que el análisis global del programa detecta que todas las posibles activaciones del primer objetivo tienen únicamente la primera cláusula del procedimiento como candidata a su satisfacción (se descarta el resto de cláusulas como candidatas), mientras que el segundo objetivo mantiene todas las cláusulas como candidatas. En este caso, los dos objetivos y el procedimiento serán transformados de forma que se generan dos puntos de entrada para el procedimiento, y cada objetivo está asociado a uno de estos puntos de entrada:

```
..., g_ep1(t1, ..., tn), ...
..., g_ep2(v1, ..., vn), ...

g_ep1(A1, ..., An) :- g_cl1(A1, ..., An).
g_ep2(A1, ..., An) :- g_cl1(A1, ..., An); g_cl2(A1, ..., An); g_cl3(A1, ..., An).

g_cl1(a11, ..., a1n) :- ...
g_cl2(a21, ..., a2n) :- ...
g_cl3(a31, ..., a3n) :- ...
```

3.2.1.2 Indexación de puntos de entrada.

Es posible disminuir el número de cláusulas candidatas a satisfacer un objetivo mediante la evaluación de una función en el momento de activar dicho objetivo que restringe, en función

de los términos Prolog asociados a los argumentos del objetivo, la lista de cláusulas candidatas a resolverlo. En el caso general, la selección de cláusulas se consigue mediante la evaluación de un grafo de decisión que contiene tres tipos de tests: unificaciones, comparaciones de tipos de datos y comparaciones aritméticas [64].

La opción utilizada en Multipath consiste en establecer una función de **indexación** en la que, a partir del término que posee uno de los argumentos del objetivo en el momento de activarse, se determina la lista de cláusulas candidatas para resolverlo (ver figura 3.3b). Más en concreto, esta función permite discriminar según el tipo de datos de un argumento. Además, cuando el término es una constante se permite discriminar según su valor, y cuando el término corresponde a una estructura también se permite discriminar según el valor del functor. Por otra parte, el argumento que se indexa es aquél que permite eliminar un mayor número de cláusulas candidatas a realizar la unificación.

El próximo ejemplo muestra la transformación que sufre un punto de entrada por el establecimiento de una función de indexación. Considérese el siguiente punto de entrada.

$g(a_{11}, ..., a_{1n})$:- ...
$g(a_{21}, ..., a_{2n})$:- ...
$g(a_{31}, ..., a_{3n})$:- ...

En caso de localizar un argumento que elimina cláusulas candidatas según el término que posea en el momento de la activación, el código transformado puede representarse de la siguiente manera:

```
g(A1, ..., An) :-   ( Aj = term1 -> g_cl1(A1, ..., An);
                    ( Aj = term2 -> g_cl2(A1, ..., An);
                    ( Aj = term3 -> (g_cl1(A1, ..., An); g_cl2(A1, ..., An); g_cl3(A1, ..., An)) ))).
g_cl1(a11, ..., a1n) :- ...
g_cl2(a21, ..., a2n) :- ...
g_cl3(a31, ..., a3n) :- ...
```

3.2.1.3 Inserción automática de instrucciones de poda.

Esta forma de reducción del indeterminismo está asociada a la posibilidad que existe de determinar con mayor precisión el dominio de términos de los argumentos de un objetivo en su activación por el hecho de haber finalizado con éxito la unificación de la cabecera o la satisfacción de algún objetivo del cuerpo de una cláusula perteneciente al punto de entrada. Téngase en cuenta que la información referente a tipos de datos y condiciones de éxito de una cláusula puede ser más precisa conforme se avanza en el árbol de búsqueda.

Si el análisis global determina que, después de unificar la cabecera o satisfacer un subobjetivo del cuerpo de una cláusula, los tipos de datos y términos de los argumentos en el momento de activar un objetivo permiten garantizar las condiciones que aseguran que el resto de cláusulas candidatas fracasarían, se introduce de forma automática un **operador de poda** del árbol de búsqueda (ver figura 3.3c). Hay que tener en cuenta que, cuando la cláusula corresponde a un hecho, el operador que se inserta tiene la misma semántica que el operador de corte (!) de Prolog. No obstante, cuando la cláusula corresponde a una regla, el operador de poda elimina únicamente las cláusulas alternativas candidatas a unificar que quedan por probar del punto de entrada, y no elimina las cláusulas que pueden quedar pendientes en los puntos de entrada de los objetivos precedentes en el cuerpo de la cláusula.

El siguiente ejemplo muestra la transformación del programa por la inserción de un operador de corte tras la unificación de la cabecera de una cláusula. Considérese la siguiente lista de cláusulas.

$g(a_{11}, ..., a_{1n})$.
$g(a_{21}, ..., a_{2n})$:- ...

Si la unificación con éxito de la cabecera de la primera cláusula asegura que los términos que poseen los argumentos del objetivo a resolver son incompatibles con las condiciones de éxito de la segunda cláusula, la interpretación abstracta transforma dicho código en:

$g(a_{11}, ..., a_{1n})$:- !.
$g(a_{21}, ..., a_{2n})$:- ...

3.2.2 Exploración del árbol de búsqueda

En este apartado se define el recorrido del árbol de búsqueda que se propone en Multipath. Esta descripción pretende ser el punto de partida inicial en la descripción de la gestión del indeterminismo remanente en un programa.

En Multipath, el recorrido del árbol de búsqueda se realiza mediante la denominada **exploración parcial en anchura** a nivel de objetivos [122]. Esta regla de búsqueda se aplica para cada uno de los objetivos a satisfacer durante la ejecución, y está dividida en dos fases, correspondientes al cálculo de soluciones del objetivo y a la ejecución de la continuación del objetivo.

En la primera fase se **calcula secuencialmente un cierto número de soluciones del objetivo**. Este número puede variar desde una solución hasta todas las soluciones que posea el objetivo. En este sentido, cuando se activa un objetivo el recorrido empieza por la primera alternativa y continúa recorriendo este camino hasta encontrar una solución del objetivo. En el

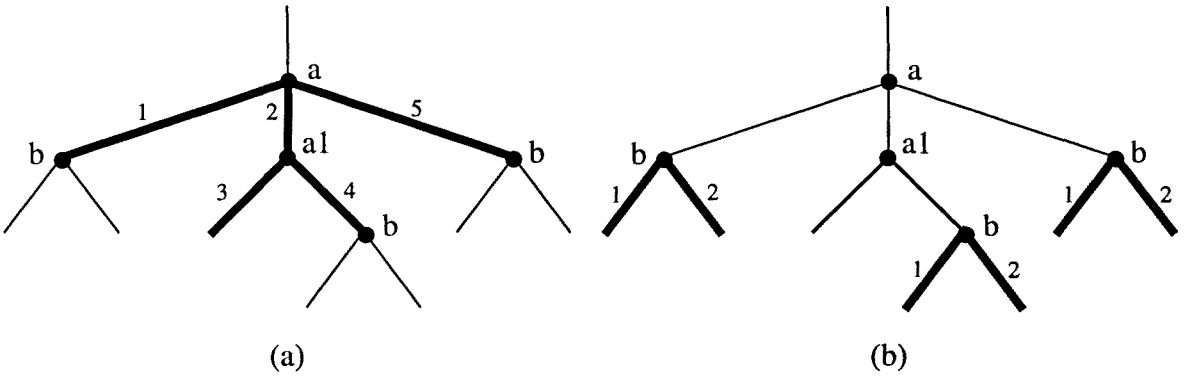


Figura 3.4: *Exploración parcial en anchura de un objetivo:*

(a) Cálculo de todas las soluciones del objetivo a/0 (cada rama indica su orden de recorrido).

(b) Ejecución de la continuación del objetivo a/0 (cada rama indica su orden de recorrido).

momento de encontrar la primera solución, existe la posibilidad de continuar el recorrido del mismo camino del árbol de búsqueda o de tratar de explorar otro camino del árbol que permita obtener una nueva solución del objetivo. Si se opta por la primera alternativa, el recorrido coincide con una exploración en profundidad. Si se opta por la segunda alternativa, se está recorriendo en anchura el árbol de búsqueda ya que, a partir de este momento, es visible más de un camino. La decisión de continuar o finalizar el cálculo de nuevas soluciones viene determinada por factores de rendimiento, que son transparentes en esta definición de la regla de búsqueda. Si se produce un fracaso durante el recorrido, se aplica el mismo criterio que decide si continúa el intento de explorar nuevas soluciones potenciales o se da por finalizada esta fase de cálculo de soluciones.

Dado el mismo programa de la figura 3.2a, obsérvese en la figura 3.4a un ejemplo de esta fase de la exploración parcial en anchura aplicada al objetivo *a/0*, en la que se calculan todas las soluciones existentes de dicho objetivo *a/0*.

La segunda fase de la exploración parcial en anchura de un objetivo consiste en el **recorrido simultáneo de los caminos asociados a las soluciones encontradas previamente** a dicho objetivo. La simultaneidad en este parte del recorrido es posible gracias a que todas las soluciones comparten el mismo flujo de control, establecido por la continuación del objetivo, y por la independencia existente entre los caminos del árbol de búsqueda.

Para el mismo ejemplo referenciado en la fase anterior, la figura 3.4b muestra el orden

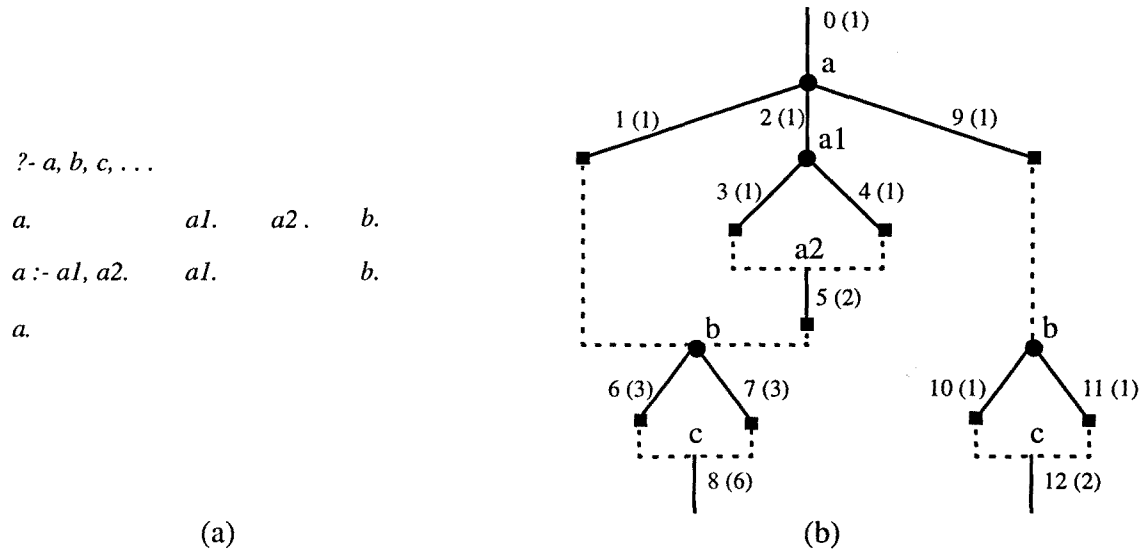


Figura 3.5: *Exploración parcial en anchura a nivel de objetivos realizada en Multipath:*
(a) Programa base.
(b) Árbol Multipath (cada rama indica el orden de recorrido y, entre paréntesis, el número de caminos visibles).

en el recorrido efectuado en esta fase. Téngase en cuenta que la continuación del objetivo *a/0* está formada únicamente por el objetivo *b/0*, que posee dos cláusulas alternativas. En este sentido, se recorre en primer lugar los caminos asociados a la primera cláusula de forma simultánea en las tres soluciones calculadas anteriormente. Tras llegar a las hojas del árbol, el recorrido continúa explorando los caminos asociados a la segunda cláusula de *b/0*.

La visualización del recorrido de la exploración parcial en anchura mediante un árbol OR no es la más apropiada para apreciar la simultaneidad en la recorrido de varios caminos. Por ello, se propone la representación del árbol de búsqueda de un programa mediante el denominado árbol Multipath. Respecto a un árbol OR, en un árbol Multipath se introduce un nuevo nodo que corresponde a una solución de un objetivo, y se contempla la posibilidad de reunir varios caminos solución en una única rama, simbolizando el flujo de control común a las soluciones. Debido a que en cada punto del árbol de búsqueda Multipath se está recorriendo más de un camino del árbol OR tradicional, existe la posibilidad de especificar el número de caminos recorridos en los puntos más característicos de las ramas. Téngase en cuenta que el número de caminos normalmente se incrementa tras la reunión de soluciones, y disminuye en una misma rama a medida que las unificaciones van fracasando de forma local en algunos caminos.

La figura 3.5b muestra el árbol de búsqueda Multipath de un posible recorrido parcial en anchura, contemplando la anidación y serialización existente entre los objetivos presentes en el programa de la figura 3.5a. Obsérvese que se calculan tres de las cuatro soluciones del objetivo *a/0* antes de recorrer el subárbol asociado a su continuación. La cuarta y última solución del objetivo *a/0* no se obtiene hasta que haya finalizado completamente el recorrido del subárbol anterior. Nótese también que pueden existir distintos árboles Multipath equivalentes a un mismo árbol OR, según el número de soluciones obtenidas en los objetivos presentes.

3.2.2.1 Ventajas e inconvenientes de la exploración parcial en anchura

En este momento se pueden apreciar las **ventajas** que ofrece este tipo de búsqueda [121]. Un único flujo de control para recorrer más de un camino ofrece la posibilidad de **ejecutar una única vez** en todos los caminos explorados simultáneamente aquellas **operaciones que comparten los datos**. A su vez, todas las **operaciones relacionadas con el control** de la ejecución se ejecutan también una única vez, a diferencia de lo que sucede en una exploración en profundidad, donde son repetidas durante el recorrido de cada uno de los caminos.

Además, se introduce una fuente de paralelismo inherente al modelo, basado en el paralelismo de datos y que, en este contexto de ejecución de programas lógicos, hemos denominado **paralelismo de caminos** [123]. El paralelismo de caminos corresponde a la posibilidad de ejecutar en paralelo operaciones sobre datos distintos que deben realizarse en cada uno de los caminos que se recorre de forma simultánea.

Obsérvese la importancia de poder determinar si las variables que se gestionan en un programa tienen un vínculo común en todos los caminos en que son visibles o poseen vínculos locales a cada camino. Si el número de variables con vínculo común es alto, se reducirá en mayor medida el número total de operaciones a efectuar durante el recorrido del árbol. No obstante, la existencia de variables con vínculos locales ofrece una capacidad intrínseca de ejecutar en paralelo las operaciones requeridas durante el recorrido del árbol.

Los **inconvenientes** de la exploración parcial en anchura hacen referencia a la mayor **complejidad en la gestión del recorrido** y en la **pérdida de localidad en el acceso a la información**. El incremento en la complejidad viene marcado por la necesidad de introducir nuevos elementos, respecto a los necesarios en una búsqueda en profundidad, para poder diferenciar y acceder a la información local referente a cada uno de los caminos visibles. La pérdida de localidad se pone de manifiesto porque la información que se requiere acceder en un mismo intervalo de tiempo está más distribuida que en una exploración en profundidad. Esta pérdida de localidad se acrecienta conforme se hace visible un número de caminos simultáneos más

elevado.

3.2.2.2 Clasificación de los objetivos

Obsérvese que la regla de búsqueda definida en Multipath ofrece indeterminismo en la ejecución de un programa, a diferencia de la regla determinista correspondiente a la exploración en profundidad, utilizada en los sistemas convencionales orientados a Prolog. En Multipath, la ejecución de un programa calcula un número variable de soluciones para cada uno de los objetivos de que consta. Si se calcula una solución por objetivo, Multipath realiza una exploración del primero en profundidad (*depth-first search*). Si se calculan todas las soluciones existentes para cada objetivo, Multipath realiza una exploración del primero en anchura (*breadth-first search*).

La idea básica en la definición de esta nueva regla de búsqueda es poder combinar las ventajas que ofrecen tanto el recorrido en profundidad como el recorrido en anchura, y evitar en la mayor medida posible sus inconvenientes. Debe tenerse en cuenta que un objetivo Prolog puede tener 0, 1 o más de una solución y, por tanto, existen objetivos en los cuales no se podrán aplicar las ventajas que ofrece la exploración en anchura. En este sentido, es importante tipificar los objetivos Prolog de cara a identificar aquellos objetivos en los cuales se pueden explotar las ventajas de la exploración en anchura y aquéllos en que no. Para estos últimos, es recomendable poder establecer lo antes posible que el número de soluciones que se deben obtener en cada activación es una. Para los primeros, es preciso proponer una estrategia que limite el número de soluciones a calcular entre una y el número total.

Se han determinado tres tipos de objetivos en los que se pueden aplicar las ventajas de la exploración en anchura, que son denominados:

- Objetivos indeterministas,
- Objetivos semideterministas, y
- Objetivos con punto de indexación

Los **objetivos indeterministas** son aquellos que poseen más de una solución. Este tipo de objetivos constituye la base mediante la cual se obtienen las ventajas de la exploración en anchura. La reunión en un único flujo de control de sus soluciones permite que las instrucciones de control y de datos comunes se ejecuten una única vez durante la exploración del subárbol correspondiente a la continuación del objetivo indeterminista.

Ejemplo: Dado el siguiente objetivo y punto de entrada:

```
?- ..., a(X), ...
a(1).
a(2).
a(3).
```

Si la variable *X* corresponde a una variable libre en la activación del objetivo *a(X)*, se puede observar que el objetivo tiene 3 soluciones, caracterizadas por los vínculos: *X/1*, *X/2* y *X/3*. La exploración en anchura de este objetivo permite explorar simultáneamente la continuación de este objetivo indeterminista para las 3 soluciones que posee.

Los **objetivos semideterministas** corresponden a aquellos objetivos que poseen una solución como máximo (son deterministas), pero que no es calculada mediante la misma cláusula para todos los caminos de entrada del objetivo. Denominamos *caminos de entrada* de un objetivo a los caminos visibles en su activación. Nótese que desde el momento en que ya se ha explorado en anchura un objetivo del programa, la ejecución puede estar recorriendo simultáneamente más de un camino del árbol de búsqueda. A partir de este momento, es aconsejable la exploración en anchura de un objetivo semideterminista para poder obtener la solución, si existe, en el mayor número posible de caminos de entrada al objetivo. En un recorrido en profundidad sólo se obtendría la solución para un subconjunto de los caminos de entrada (los asociados a la primera cláusula que tuviera éxito). El resto de soluciones se calcularía tras la operación de backtracking.

Ejemplo: Dado el siguiente objetivo y punto de entrada:

```
?- ..., a(X,Y), ...
a(X,[X|_]) :- !.
a(X,[Y|L]) :- a(X,L).
```

Se puede observar que el objetivo *a(X,Y)* es determinista (tiene una única solución), pero la solución puede obtenerse mediante la primera cláusula (si el primer argumento unifica con la cabeza de la lista del segundo argumento) o con la segunda cláusula (cuando no se cumple la anterior condición). En tiempo de ejecución, puede activarse este objetivo semideterminista simultáneamente para varios caminos de entrada. La exploración en anchura permite explorar la continuación del objetivo para todos los caminos que correspondan a soluciones de este objetivo.

Existen otros objetivos deterministas en los cuales se puede asegurar que, de forma independiente a los caminos de entrada, la cláusula que provoca la solución siempre será la misma.

Esto sucede siempre que el término o términos Prolog implicados en el éxito o fracaso de una cláusula corresponden a datos comunes a todos los caminos de entrada.

Los **objetivos con punto de indexación** son aquellos objetivos (indeterministas o deterministas) en los que se aplica una función de indexación y, a su salida, se obtiene más de un grupo de caminos asociado a un tipo de datos y valor distinto para el argumento indexado. Obsérvese que la técnica de indexación causa la división de todos los caminos de entrada en la activación de un objetivo en varios grupos de caminos. En cada *grupo de caminos* el argumento indexado posee el mismo tipo de datos y valor, y existe un flujo de control distinto para encontrar la solución. El momento en que se obtiene más de un grupo de caminos se denomina *punto de indexación*. La búsqueda en anchura en un punto de indexación permite calcular la solución, si existe, para todos los caminos de entrada. La búsqueda en profundidad se caracterizaría por encontrar la solución en el primer grupo de caminos del punto de indexación.

Ejemplo: Considérese el siguiente objetivo y punto de entrada:

```
?- ..., a(X,...), ...
a(1,...) :- ...
a(1,...) :- ...
a([Y],...) :- ...
```

Si no se puede determinar el tipo de datos que posee la variable *X* en cada activación del objetivo *a(X, ...)*, la interpretación abstracta inserta una función de indexación en el punto de entrada que permite, dada una activación concreta del objetivo, explorar únicamente las dos primeras cláusulas cuando la variable *X* corresponda a un término constante; la última cláusula cuando sea una lista, o todas las cláusulas cuando sea una variable libre. Por el hecho de estar recorriendo simultáneamente varios caminos en el momento de activar el objetivo puede llegarse a un punto de indexación. La exploración en anchura de todos los grupos de caminos, cada una de ellos con el mismo término Prolog en el primer argumento, permite calcular todas las soluciones del objetivo antes de proseguir con su continuación.

3.2.2.3 Determinación del número de soluciones de un objetivo

Una vez tipificados los objetivos en los cuales es recomendable calcular más de una solución, se debe especificar cuándo y cómo se localizan estos objetivos.

En primer lugar, es importante darse cuenta que el momento más tarde en que puede optarse por la exploración en anchura de un objetivo (calcular más de una solución) corresponde

al momento de encontrar la primera solución de dicho objetivo. A partir de este momento, la exploración en profundidad y en anchura difieren. Por otra parte, en la tipificación de los dos primeros casos del apartado anterior se han utilizado condiciones que no pueden conocerse en el momento de encontrar la primera solución de un objetivo. No se sabe si el objetivo puede tener más soluciones (el objetivo sería indeterminista) o bien si los caminos que han encontrado la primera solución no tienen ninguna otra solución en el resto de cláusulas (el objetivo sería semideterminista).

En consecuencia, la responsabilidad de determinar el número de soluciones de un objetivo debe recaer en el programador o bien debe ser el sistema que calcule de forma automática el número más adecuado según un compromiso coste/eficiencia. El sistema Multipath está diseñado para esta última opción: **determinación automática del número de soluciones de cada objetivo.** El cálculo de este número está basado en los siguientes criterios:

- Realización de un análisis global del determinismo del programa, aplicando técnicas de interpretación abstracta, que obtiene un atributo para cada objetivo del programa, tipificándolo según se ha establecido en el subapartado 3.2.2.2.
- Cálculo final del número concreto de soluciones en el momento de activar un objetivo y en el momento de encontrar cada una de las soluciones que tenga.

Con respecto al primer criterio, la interpretación abstracta del programa realiza una estimación de las condiciones que aseguran el éxito de las cláusulas pertenecientes a los puntos de entrada de todos objetivos, y determina un **atributo de indeterminismo** para cada objetivo del programa. Los valores que puede tomar este atributo son los siguientes:

NONDET. Se garantiza que el objetivo va a tener más de una solución. Se entiende como solución del objetivo que existe una cláusula que satisface completamente dicho objetivo (su cabecera unifica con éxito y los objetivos de su cuerpo también se satisfacen completamente). Durante la ejecución del programa (fase de interpretación real) se recomienda calcular más de una solución de este tipo de objetivos, es decir, explorarlo en anchura.

SEMIDET. El objetivo va a tener una solución como máximo pero la cláusula que lo satisface depende de los valores locales a cada camino que poseen algunos argumentos. En tiempo de ejecución se recomienda una exploración en anchura de estos objetivos.

OTHER. Se utiliza para objetivos que no cumplen alguna de las condiciones ante-

riores, incluyendo la posibilidad que la interpretación abstracta no sea capaz de evaluar correctamente este atributo de indeterminismo. Durante la ejecución, se recomienda obtener una única solución de dicho objetivo, es decir, explorarlo en profundidad.

Debido a que esta información no está contemplada en la sintaxis del lenguaje Prolog, será representada en los programas mediante una directiva

:- attr(cláusula, num_objetivo, atributo).

La identificación del objetivo se realiza mediante los dos primeros argumentos: *cláusula* especifica la cláusula que contiene el objetivo y *num_objetivo* especifica el número de orden del objetivo dentro del cuerpo de la cláusula. El tercer argumento especifica el atributo y corresponde a uno de los tres valores descritos anteriormente. El valor por defecto de este atributo es OTHER. Multipath contempla la posibilidad que el programador pueda realizar anotaciones en el programa para poder forzar el tipo de exploración que desee, aparte del que pueda ser establecido por la interpretación abstracta del programa.

El segundo criterio para determinar el número final de soluciones tiene en cuenta los factores que pueden limitar el beneficio de una exploración en anchura. En este sentido, se define un número máximo de caminos visibles en un momento de la ejecución, denominado **MAX_CAMINOS**. En un principio, el número de soluciones que se calcularán para objetivos NONDET es el total de soluciones que tenga. Este número puede limitarse en la activación del objetivo: teniendo en cuenta el número de caminos de entrada en la activación, se calcula 1 solución si el número total de caminos visibles en el momento de encontrar la segunda solución, suponiendo que se obtiene solución para todos los caminos de entrada, excediese MAX_CAMINOS; y también puede limitarse en el momento de encontrar cada solución: si el número de caminos visibles que implicaría la exploración en anchura de una nueva alternativa excede MAX_CAMINOS, se da por finalizado el cálculo de soluciones.

El número de soluciones a calcular en objetivos SEMIDET siempre será el máximo ya que nunca se incrementa el número total de caminos visibles. Téngase en cuenta que el objetivo tiene como máximo una solución en cada camino de entrada. Obsérvese que si el número de caminos de entrada es uno, no es necesario establecer el número de soluciones ya que la exploración en profundidad y en anchura coinciden.

El número de soluciones para objetivos OTHER es 1, es decir, siempre se exploran en profundidad.

Existe otra situación en la que se realiza la exploración en anchura, que es detectada dinámicamente: en todos aquellos objetivos con punto de indexación. Cuando sucede esta situación se permite siempre calcular todas las soluciones. Al igual que en los objetivos SEMIDET, el inicio del cálculo de una nueva solución no incrementa nunca el número total de caminos visibles.

3.3 Trabajos relacionados

La comparación con otros trabajos directamente relacionados sigue el mismo orden con el que se ha presentado la gestión del indeterminismo en Multipath: reducción y gestión eficiente del indeterminismo remanente. Es importante mencionar que las técnicas presentadas en Multipath sobre reducción de indeterminismo son una aplicación directa de la información calculada por el análisis global del programa, cuyo objetivo primordial es encontrar el atributo de indeterminismo de cada objetivo. Las técnicas de reducción del indeterminismo se han presentado antes que la gestión del indeterminismo remanente, que es el principal aspecto de investigación, por seguir un orden más lógico en la descripción del comportamiento de Multipath

En referencia a la tarea de **disminuir el indeterminismo** que exhibe un programa, se han propuesto técnicas, también basadas en la *interpretación abstracta* [26], tendentes hacia el establecimiento de una función de indexación que se pueda aplicar sobre el argumento que es capaz de eliminar un mayor número de cláusulas. Andorra-I [98] es un ejemplo de estos sistemas. En este sistema, el análisis de determinismo de un programa está principalmente orientado a localizar los objetivos deterministas, con el fin de ejecutarlos antes que los objetivos indeterministas. Esta idea significa un cambio en la regla de selección de los objetivos de un programa, que no se ha realizado en Multipath. El análisis en Andorra-I se realiza únicamente a partir de las cabeceras de las cláusulas y de predicados predefinidos internos. En Multipath, el análisis global tiene en cuenta todos los objetivos presentes en el cuerpo de una cláusula.

Aunque la determinación de los puntos de entrada se puede conseguir directamente a partir de la información necesaria para calcular la función de indexación, no se conocen trabajos relacionados explícitamente en este aspecto.

La disminución de indeterminismo durante la ejecución de los objetivos del cuerpo de una cláusula está relacionada con los trabajos sobre *backtracking inteligente* [28]. En estos trabajos se evita el intento de resatisfacer aquellos subobjetivos que no contribuyen a encontrar una solución del objetivo que los engloba. El backtracking inteligente se caracteriza por la cantidad de trabajo a realizar de forma dinámica y de forma estática. Los primeros trabajos

[92] se basaban en detectar las condiciones para eliminar backtrackings a puntos de selección innecesarios en tiempo de ejecución. Posteriormente, se han propuesto técnicas basadas en un análisis de dependencias de los datos [17] que realizan, en tiempo de compilación, la determinación de los puntos de selección a descartar durante el backtracking. En este caso, se prima por la disminución de la penalización en tiempo de ejecución, aunque no se consiga la reducción máxima de indeterminismo, por la necesidad de estimar el caso peor [69]. En Multipath, también se aboga por efectuar un análisis global antes del tiempo de ejecución.

Mercury es un ejemplo de definición de un nuevo lenguaje lógico que incorpora declaración de atributos de determinismo [102]. Mediante la información que proporciona el programador se simplifica el análisis estático de determinismo del programa. En este caso, el análisis se orienta a evitar la creación de puntos de selección en aquellos objetivos que tienen una solución como máximo. En Multipath, se acepta el lenguaje Prolog standard, que no incorpora declaraciones del grado de determinismo o indeterminismo de los objetivos.

Relacionado con la reducción del indeterminismo se encuentra el paradigma de la programación lógica con restricciones (CLP). Su idea básica es la de proporcionar nuevas reglas de inferencia a la utilizada en Prolog que permiten establecer las propiedades que deben cumplir las variables e instanciarlas incrementalmente conforme se dispone de la información de que dependen. Con este nuevo paradigma aumenta el grado de determinismo de un programa, es decir, se reduce el árbol de búsqueda. Multipath ejecuta el lenguaje Prolog standard, pero es importante destacar que las ideas presentadas pueden aplicarse a cualquier lenguaje de programación que permita expresar indeterminismo, incluidos los lenguajes pertenecientes a la CLP.

En referencia a la tarea de **recorrer** de forma más eficiente **el árbol de búsqueda** de un programa, los trabajos más relacionados con las ideas que presenta el sistema Multipath son las siguientes.

A nivel teórico, Escalada [38] define los denominados multisistemas como un mecanismo de resolución que permite tener múltiples vínculos en las variables Prolog y la combinación de estos vínculos entre varias variables. Multipath tiene su propia gestión de variables con múltiples vínculos obtenidos en la exploración parcial en anchura y, además, se complementa con un modelo arquitectónico y realizaciones concretas del sistema que permiten medir su rendimiento.

El sistema DAP Prolog propuesto por Kacsuk y Bale en 1988 [61] presenta un modo de operación basado en conjuntos que extiende la realización convencional de Prolog con el fin de soportar conjuntos de datos y permitir la explotación de paralelismo en la gestión de los diferentes elementos de un conjunto. El concepto de conjuntos se asemeja a las soluciones que

pueden obtenerse mediante una exploración en anchura como la que realiza el sistema Multipath. Sin embargo, los sistemas DAP Prolog y Multipath presentan diferencias importantes. En primer lugar, DAP extiende la semántica de Prolog con los denominados conjuntos mientras que en Multipath un programa Prolog no requiere ninguna modificación para poder ser ejecutado. En segundo lugar, el paralelismo en DAP está basado únicamente en los hechos de un programa mientras que en Multipath se basa en cualquier tipo de procedimientos (tanto hechos como reglas) que pueden resolver objetivos indeterministas. DAP Prolog constituyó únicamente un diseño de modelo de ejecución sin ningún prototipo construido que permita establecer una estimación de su rendimiento.

Kanada propone la realización de transformaciones en los programas Prolog y la aplicación de técnicas de vectorización que permiten obtener todas las soluciones de objetivos no deterministas en vectores de variables para, posteriormente, procesar de forma vectorial el acceso a este tipo de variables [62]. Esta propuesta ha sido únicamente probada para un benchmark (problema de las 8 reinas) en un procesador vectorial Hitachi S-810.

El trabajo más similar a Multipath fue propuesto por Smith en 1993, en la misma fecha en que apareció la primera publicación de Multipath, con la presentación de un lenguaje de explotación de paralelismo de datos, denominado Multilog [108]. Este lenguaje corresponde básicamente a Prolog junto con anotaciones de objetivos indeterministas, permitiendo el cálculo de un conjunto de soluciones de estos objetivos y la posterior ejecución paralela de los objetivos siguientes, basándose en la regla de inferencia que ha denominado Multi-SLD [111]. Las diferencias principales están relacionadas con la gestión de los objetivos a explorar en anchura y de las variables Prolog. En Multipath, la recomendación de los objetivos a explorar en anchura se determina automáticamente mientras que en Multilog son declarados mediante anotaciones del programador. Por lo que respecta a la gestión de las variables, en Multipath se realizan técnicas más agresivas de análisis global del programa que permiten incrementar el número de variables con un único vínculo común en los caminos recorridos simultáneamente y, por tanto, incrementar el beneficio que supone una exploración en anchura en cuanto a reducción de operaciones que comparten los mismos datos. Multilog se ha realizado en un sistema secuencial, y los resultados obtenidos serán comparados con las realizaciones del sistema Multipath.

3.4 Descripción del sistema Multipath

En la sección 3.2 se han presentado las ideas principales que se proponen en el sistema Multipath. La descripción detallada de Multipath sigue un enfoque vertical, empezando por la semántica del lenguaje Prolog tal como es interpretado en Multipath, la definición del modelo

de ejecución, la definición del modelo arquitectónico y, en último lugar, dos realizaciones concretas de Multipath. En los siguientes apartados se resumen las características principales de cada visión del sistema. Posteriormente, el capítulo 4 está dedicado a describir completamente el modelo de ejecución; los capítulos 5 y 6 están dedicados a describir el modelo arquitectónico; y, finalmente, los capítulos 7 y 8 corresponden a las dos realizaciones de Multipath.

3.4.1 Semántica de Prolog en Multipath

Normalmente, un programador en lenguaje Prolog considera la visión imperativa que posee el lenguaje para poder expresar algoritmos de resolución de problemas. Aunque el lenguaje que es capaz de interpretar el sistema Multipath es el lenguaje Prolog, hay que tener en cuenta una única diferencia que tiene la semántica imperativa de un programa Prolog en el sistema Multipath respecto al definido en el standard propuesto por el ISO/IEC 13211-1.

En concreto, si existe más de una manera de satisfacer un objetivo parcial del programa, el orden de satisfacción de los objetivos posteriores es indeterminista y, como resultado, **el orden de obtención de las soluciones de un programa** también **es indeterminista**. Desde el punto de vista del programador, esta restricción se puede justificar en el sentido que si un problema tiene varias soluciones, es independiente el orden en que se calcule una (o todas las soluciones) del programa mientras el sistema sea capaz de obtenerla (u obtenerlas) en el menor tiempo posible.

Nótese que, desde el punto de vista de la semántica declarativa, un programa Prolog en Multipath continúa comportándose como un demostrador automático de problemas. El mecanismo de refutación es el mismo que se ha descrito en el capítulo 2, y la única diferencia reside en el cambio de la regla de búsqueda, que pasa de ser una exploración en profundidad a una exploración parcial en anchura.

3.4.2 Modelo de Ejecución de Multipath

Un modelo de ejecución describe las fases que permiten interpretar la semántica del lenguaje. Además, en cada una de las fases se indican los elementos del estado de la ejecución y las operaciones que lo modifican. En esta visión del sistema es transparente la forma concreta de representar la información y de realizar las operaciones.

El Modelo de Ejecución de Multipath, denominado MEM, consta de dos fases: una interpretación abstracta y una interpretación real del programa.

La funcionalidad de la interpretación abstracta es transformar el código Prolog original e

insertar información complementaria para su posterior ejecución. Las características concretas son:

- Determinación del atributo de indeterminismo de cada objetivo.
- Determinación de los puntos de entrada de cada procedimiento.
- Inserción de funciones de indexación.
- Inserción de instrucciones de poda.

La interpretación real es la responsable de aplicar el procedimiento de decisión convencional del lenguaje Prolog. Sus características concretas son:

- Regla de búsqueda basada en una exploración parcial en anchura.
- Determinación del número concreto de soluciones a calcular en cada objetivo en el momento de su activación y en el momento de encontrar cada solución.
- Criterio óptimo para determinar el número de vínculos de las variables. Cada variable posee un vínculo común en todos los caminos recorridos simultáneamente o vínculos locales a cada camino.
- Posibilidad de explotar el denominado paralelismo de caminos. Las instrucciones que operan con variables que poseen vínculos locales en cada camino se pueden realizar en paralelo sin ningún tipo de dependencias entre ellas.

3.4.3 Modelo Arquitectónico de Multipath

Un modelo arquitectónico define la representación de los elementos que forman el estado de la ejecución y la forma de realizar las operaciones presentes en el modelo de ejecución en que se basa.

El Modelo Arquitectónico de Multipath, denominado MAM, está basado en el Modelo de Ejecución de Multipath. En concreto, MAM integra todos los aspectos de MEM excepto el criterio óptimo de determinación del tipo de las variables. En concreto, el tipo de una variable es decidido en el momento de su creación y no puede modificarse en adelante. La interpretación abstracta calcula información relevante para tomar esta decisión.

Los aspectos básicos de MAM en la interpretación abstracta son:

- Definición del dominio abstracto utilizado en el análisis de tipos de datos y de determinismo de un programa.

- Obtención de la información característica de cada objetivo mediante evaluación de funciones.

Los aspectos básicos de la MAM en la interpretación real son:

- Definición de motores como elementos responsables de explotar el paralelismo inherente en el modelo de ejecución.
- Gestión de múltiples entornos de vínculos mediante copia de entornos.

3.4.4 Realizaciones de Multipath

Se ha realizado el Modelo Arquitectónico de Multipath en una estación de trabajo secuencial y en un sistema multiprocesador con memoria compartida. En cada una de estas realizaciones (SMS y PMS, respectivamente) se describen únicamente los aspectos relacionados con el mapeo de la fase de interpretación real del modelo arquitectónico en la plataforma hardware correspondiente. El análisis del comportamiento de Multipath se obtiene mediante su comparación con el modelo convencional establecido por la Máquina Abstracta de Warren.

3.5 Resumen y contribuciones

En este capítulo se han presentado las ideas que incorpora Multipath en el aspecto de la gestión del indeterminismo existente en los programas Prolog.

La contribución más importante es la definición de una nueva regla de búsqueda del árbol que refleja el indeterminismo de un programa, denominada exploración parcial en anchura. El número de soluciones que se calculan en cada objetivo de un programa se determina a partir de un atributo de determinismo de los objetivos. Este atributo de determinismo se obtiene a través de un análisis global del programa basado en técnicas de interpretación abstracta.

La información que es necesaria calcular en el análisis de determinismo se aprovecha para reducir en lo posible el árbol de búsqueda de un programa. Se han presentado tres técnicas de reducción del indeterminismo que se caracterizan por el momento en que se eliminan cláusulas candidatas a satisfacer un objetivo: de forma estática; en la activación del objetivo; y durante la ejecución del cuerpo de una cláusula.

4

MODELO DE EJECUCIÓN DE MULTIPATH

El objetivo que subyace en la definición del **Modelo de Ejecución de Multipath (MEM)** [120] es establecer los aspectos básicos de la ejecución, o interpretación, de programas Prolog. En este sentido, el modelo de ejecución describe las transformaciones y optimizaciones que pueden realizarse a un programa Prolog sin perder su semántica inicial, así como los elementos del estado de la ejecución y las operaciones que lo modifican durante su interpretación. En el modelo de ejecución quedan sin definir aquellos aspectos relacionados con la representación de las estructuras de datos que forman el estado de la ejecución o de la realización de las operaciones que lo modifican, que son dependientes de la arquitectura específica del sistema en que se ejecute. Estos aspectos son descritos en los dos siguientes capítulos.

Las características principales del modelo de ejecución se centran en la capacidad de disminuir el árbol de búsqueda que se genera en el modelo convencional y la posibilidad de realizar con mayor eficiencia el recorrido del nuevo árbol de búsqueda. Es importante resaltar que las decisiones realizadas en Multipath se realizan automáticamente de forma transparente al usuario. De todas formas, el programador tiene la posibilidad de proporcionar información

referente al comportamiento del programa que ayude al proceso de ejecución.

En primer lugar, MEM establece dos fases en la interpretación de un programa. La primera fase corresponde a una **interpretación abstracta** del programa, cuya misión es analizar de forma global el comportamiento del programa con el fin de transformar el programa original y calcular información relevante para la posterior fase de **interpretación real** del programa. Cada ejecución de la fase de interpretación real se realiza a partir de la información proporcionada por la interpretación abstracta, por lo que esta fase de interpretación abstracta debe ejecutarse una única vez para cada programa Prolog.

Este capítulo está dividido en dos secciones básicas, dedicadas a desglosar el comportamiento de la interpretación abstracta y de la interpretación real, respectivamente.

4.1 Interpretación abstracta

La interpretación abstracta fue propuesta por Cousot [26] como una herramienta que generaliza la mayoría de las técnicas utilizadas para realizar un análisis global de un programa. El punto de partida consiste en determinar el dominio abstracto de información que se desea conocer mediante el análisis global. El objetivo de la interpretación abstracta consiste en obtener la máxima y más específica información posible del dominio abstracto elegido, basándose en la modificación de ese dominio abstracto según la semántica que posee el lenguaje [27].

De forma consecuente, cuanto más completo y específico sea el dominio de información elegido mayor será la complejidad de la realización de la interpretación abstracta. En el caso extremo, es decir, cuando se quiere conocer cómo se comporta de forma exacta un programa, la interpretación abstracta correspondería a la ejecución real de dicho programa. Por otra parte, la interpretación abstracta tiene la ventaja de permitir realizar optimizaciones y transformaciones en el programa analizado.

En el contexto de la programación lógica, la interpretación abstracta se ha utilizado en varios sistemas para obtener información diversa, básicamente enfocada hacia el análisis de tipos de datos [51], de determinismo [48] y de ayuda a la gestión de corrutinas [86].

En el sistema Multipath, la fase de interpretación abstracta realiza un análisis de determinismo del programa basado en un dominio abstracto de las condiciones de éxito de las cláusulas. Estas condiciones de éxito se establecen a partir de los tipos de datos y valores de los términos Prolog que intervienen en las cláusulas. El resultado obtenido se encuadra en tres grandes acciones genéricas:

- La transformación del programa Prolog inicial en otro programa semánticamente equivalente con un menor número de operaciones a efectuar durante su interpretación real.
- La inserción de información no presente en el programa Prolog inicial que es necesaria durante la fase de interpretación real.
- La representación del programa transformado y la información adicional en el formato adecuado para la interpretación real.

Aparte de describir las acciones genéricas que realiza la fase de interpretación abstracta, es necesario profundizar en las tareas concretas que se pretende obtener con su realización. Recuérdese que el objetivo principal en el diseño de Multipath es la gestión eficiente del indeterminismo presente en un programa.

La acción genérica de transformación del programa está encaminada a disminuir el grado de indeterminismo que exhibe el programa Prolog inicial. En concreto, consta de tres tareas que aplican las tres técnicas descritas en el capítulo anterior para disminuir el indeterminismo en Multipath. Dichas técnicas pueden considerarse como transformaciones a realizar a nivel del propio lenguaje Prolog ya que el resultado de aplicar cada una de ellas puede describirse mediante otro programa Prolog, utilizando la misma sintaxis convencional que posee este lenguaje.

La acción descrita como inserción de información en el programa está encaminada hacia la posibilidad de optimizar el recorrido del árbol de búsqueda que posee un programa. En concreto, para cada objetivo del programa se obtiene un atributo de indeterminismo con el fin de ayudar en la fase de interpretación real a determinar el número de soluciones que deben calcularse en la exploración parcial en anchura del objetivo. Esta tarea no se incluye como una transformación del programa ya que esta información no está contemplada dentro de la sintaxis convencional del lenguaje Prolog:

La última acción consistente en realizar la traducción o compilación del programa para que pueda ser ejecutado durante la fase de interpretación real depende básicamente del modelo arquitectónico propuesto y será descrita en el capítulo 5.

En los dos apartados siguientes se definen los elementos que forman el estado de la interpretación abstracta y las operaciones que lo modifican.

4.1.1 Estado de la ejecución durante la interpretación abstracta

El estado de la ejecución durante la fase de interpretación abstracta está constituido por el programa Prolog transformado a medida que se realiza el análisis global (PRG), la información adicional para la fase de interpretación real (INFO), y la información temporal que se calcula durante el análisis de determinismo del programa, que se identifica mediante unas funciones denominadas FUNCT:

$$EJ_{IA} \equiv \{PRG,\ INFO,\ FUNCT\}$$

En los subapartados siguientes se describen estos elementos.

4.1.1.1 Programa Prolog (PRG)

El programa Prolog viene caracterizado por un conjunto de puntos de entrada a procedimientos.

$$PRG \equiv \{ep_1,\ \ldots,\ ep_i\}$$

Cada punto de entrada (EP) está formado opcionalmente por una lista de cláusulas o por una función de indexación. La función de indexación, si existe, selecciona un argumento e indica para diferentes términos Prolog una lista de cláusulas candidatas. El orden de las cláusulas, establecido por su aparición textual en el programa, indica la secuencialidad a la hora de utilizarlas para intentar resolver los objetivos asociados a dicho punto de entrada.

$$EP \equiv \{\ [cl_1,\ \ldots,\ cl_j] \mid \{arg,\ \{\{term_1,\ [cl_{11},\ \ldots,\ cl_{1j}]\},\ \ldots,\ \{term_n,\ [cl_{n1},\ \ldots,\ cl_{nj}]\}\ \}\ \}$$

Cada una de las cláusulas del programa (CL) está formada por una lista de uno o más predicados. El primer predicado corresponde a la cabecera de la cláusula y el resto de predicados corresponden a los objetivos del cuerpo de la cláusula. En caso que la cláusula sea un hecho, únicamente contiene el predicado asociado a su cabecera.

$$CL \equiv [hd,\ gl_1,\ \ldots,\ gl_k]$$

La cabecera de una cláusula (HD) está formada por sus argumentos, mientras que cada uno de los predicados del cuerpo de una cláusula, también llamados objetivos (GL), contiene, aparte de sus argumentos, el punto de entrada a que está asociado.

$$HD \equiv [arg_1,\ \ldots,\ arg_{hd}]$$
$$GL \equiv \{ep,\ [arg_1,\ \ldots,\ arg_{gl}]\}$$

Cada argumento (ARG) corresponde a un término Prolog.

4.1.1.2 Información adicional (INFO)

Este elemento del estado de la ejecución está compuesto por una función ATTR, que especifica el atributo de indeterminismo de cada objetivo del programa.

$$ATTR(gl) \rightarrow atributo$$

El dominio de valores de esta función es el siguiente:

$$atributo = \{\ NONDET \mid SEMIDET \mid OTHER\ \}$$

4.1.1.3 Funciones caracterizadoras (FUNCT)

La posibilidad de realizar las tareas que se pretenden conseguir mediante la interpretación abstracta se basa en obtener información característica del grado de determinismo en la ejecución del programa. Las funciones FUNCT permiten la realización de estas tareas ya que con ellas se obtiene información del comportamiento dinámico del programa. La información que proporcionan consiste, básicamente, en la tipificación del término Prolog asociado a cada uno de los argumentos y variables existentes en el programa, y en la determinación del número de soluciones que tienen los objetivos.

En referencia a la descripción general de las funciones FUNCT, es importante clarificar hasta qué nivel se van a definir estas funciones. La definición de una función consiste en establecer el *dominio* de valores de los parámetros de entrada y del resultado, y la descripción del cuerpo de la función, por ejemplo mediante una *ecuación* [65]. El dominio de los posibles valores del resultado de estas funciones establece el nivel con que se realiza el análisis global. Los resultados de estas funciones deben ser lo suficientemente detallados para poder realizar los objetivos propuestos. Es conveniente introducir en este momento que las funciones pertenecientes a FUNCT son dependientes entre si: en el cuerpo de una función puede utilizarse el resultado de otra función. En la definición de cada una de estas funciones aparecen constantes pertenecientes al dominio de salida de las funciones y operadores que permiten transformar los resultados que proporcionan las funciones de que dependen.

Por otra parte, la definición de un Modelo Arquitectónico concreto para la interpretación real del programa puede introducir modificaciones en el dominio de salida de las funciones y en los operadores del cuerpo de las funciones. Por ello, a nivel de Modelo de Ejecución, las funciones englobadas en FUNCT se describen mediante una indicación textual de sus parámetros de entrada y la finalidad que se pretende obtener con cada una de ellas. Lo importante en esta descripción es comprender qué deben calcular estas funciones, sin definir en forma ecuacional

el cuerpo de la función ni los valores concretos de los resultados que deben evaluarse. También se indica, para cada función, las dependencias existentes respecto a otras funciones pertenecientes a FUNCT que requiere evaluar previamente. La figura 4.1 muestra la notación que se utilizará para describir estas funciones: indicación textual de los parámetros de entrada y del resultado, e indicación de las funciones de que depende. Estas últimas funciones se modificarán mediante operadores *g* y, a su vez, sus parámetros *i(x)* se calculan según los parámetros de entrada que posea la función inicial.

dominio de la entrada y del resultado de la función

$f(x) \rightarrow y$	$f \in$ FUNCT x : descripción textual de los parámetros de entrada y: descripción textual del resultado

descripción del cuerpo de la función

$f(x) = g(\ h(\ i(x)\ )\ )$	$g \in$ OPERADORES $h \in$ FUNCT i(x) : parámetros de entrada de las funciones h

Figura 4.1: *Definición de las funciones f ∈ FUNCT según el Modelo de Ejecución de Multipath*

La definición completa de cada una de estas funciones, con el dominio concreto de salida y los operadores existentes en el cuerpo de la función, se efectúa en el capítulo siguiente dedicado al Modelo Arquitectónico de Multipath. Por otra parte, la forma de representar y evaluar estas funciones es una cuestión arquitectónica que también está descrita en el próximo capítulo. A continuación, se describen a nivel de Modelo de Ejecución todas las funciones que pertenecen a FUNCT.

Función **NSOL_EP(ep) → nsol**

Se define esta función en primer lugar como punto de partida en el **análisis global de determinismo** de un programa.

Indica el número de soluciones (*nsol*) que tiene cualquier activación de un punto de entrada del programa (*ep*). El número de soluciones corresponde al número de cláusulas que pueden satisfacer los objetivos asociados a dicho punto de entrada. En adelante, estos objetivos se denominarán objetivos padre. En caso que el número de soluciones dependa de los argumentos de los objetivos padre, también se indican las condiciones que deben cumplir estos

argumentos para tener un número concreto de soluciones.

Para poder evaluar esta función se necesita conocer las condiciones que garantizan el éxito de las cláusulas pertenecientes al punto de entrada (función SUCC_CL). Por ello, el cuerpo de la función NSOL_EP para un punto de entrada (*ep*) se establece a partir del resultado de transformar, mediante un operador *g*, el resultado de la función *SUCC_CL*, correspondiente a todas las cláusulas del punto de entrada (*cl(ep)*):

$$NSOL\_EP(ep) =$$
$$g(\ SUCC\_CL(cl(ep))\ )$$

Función **SUCC_CL(cl) → succ**

Establece las condiciones (*succ*) que permiten que una cláusula (*cl*) satisfaga los objetivos padre del punto de entrada al que pertenece dicha cláusula.

Para poder establecer las condiciones de éxito de una cláusula se precisa determinar las condiciones que aseguran el éxito de la unificación de los objetivos padre del punto de entrada, referenciados como *gl(ep(cl))*, con la cabecera de la cláusula y también se precisa conocer las condiciones que aseguran que los objetivos del cuerpo de la cláusula pueden satisfacerse. Respecto a la unificación de la cabecera, en este momento se introduce la necesidad de realizar un **análisis global de tipos de datos** de los términos asociados a los argumentos con el fin de poder precisar el resultado de la unificación.

En el cuerpo de esta función aparecen operadores *g* que determinan: (i) si la unificación del dominio de términos de los argumentos formales de los objetivos padre (función *IN_GL_ARG*) con los términos de los argumentos reales de la cabecera, referenciados como *arg(hd(cl))*, puede tener éxito, y (ii) si los objetivos del cuerpo de la cláusula pueden tener éxito (función *NSOL_EP*):

$$SUCC\_CL(cl) =$$
$$g(\ arg(hd(cl)),\ IN\_GL\_ARG(gl(ep(cl)),arg),\ NSOL\_EP(ep(cl))\ )$$

La posible existencia de objetivos no declarativos en el cuerpo de la cláusula (predicados predefinidos o built-in) tiene un tratamiento especial. En lugar de considerar las funciones NSOL_EP asociadas a estos objetivos se tiene en cuenta la semántica concreta del predicado predefinido con el fin de poder evaluar de forma más exacta las condiciones de éxito de la cláusula.

Función **IN_GL_ARG(gl,arg) → dterm**

Esta función determina el dominio de términos (*dterm*) que posee un argumento (*arg*) en la activación de un objetivo (*gl*) del cuerpo de una cláusula.

La finalidad de calcular un dominio de términos en esta función, y en el resto de funciones FUNCT que evalúan a *dterm*, estriba en restringir todos los posibles términos que puede tener un argumento o una variable en tiempo de ejecución (universo de Herbrand) a un subconjunto de dicho universo de Herbrand (el dominio de términos asociado a cada función), de forma que nunca podrá obtenerse un término que no pertenezca al dominio de términos calculado.

La evaluación de esta función depende de la definición estática de los términos utilizados en los argumentos del objetivo, referenciados como *arg(gl)*, y en caso de aparecer variables en dichos argumentos, también depende del dominio de términos que poseen las variables involucradas antes de realizar la activación del objetivo (función *IN_GL_VAR*).

$$IN\_GL\_ARG(gl,arg) =$$
$$g(\ arg(gl),\ IN\_GL\_VAR(gl,var)\ )$$

Función **IN_GL_VAR(gl,var) → dterm**

Esta función indica el dominio de términos (*dterm*) que posee una variable (*var*) en la activación de un objetivo (*gl*) perteneciente al cuerpo de una cláusula donde es visible la variable.

Cabe diferenciar que la evaluación de esta función varía si el objetivo (gl) es el primero del cuerpo de la cláusula o corresponde a cualquiera de los restantes. En caso de ser el primero, esta función depende de la definición estática de los argumentos de la cabecera, *arg(hd(cl(gl)))*, y del dominio de términos que poseen los argumentos antes de iniciar la ejecución de la cláusula (función *IN_EP_ARG*). En caso de ser cualquiera de los objetivos siguientes, esta función depende del dominio de términos que posea la misma variable en el objetivo anterior (función *IN_GL_VAR*), de la definición estática de los argumentos de dicho objetivo anterior, *arg(ant(gl))*, y de las posibles modificaciones que pueden haberse obtenido por la satisfacción de dicho objetivo anterior (función *OUT_EP_ARG*).

$$IN\_GL\_VAR(gl,var) =$$
$$= g(\ arg(hd(cl(gl))),\ IN\_EP\_ARG(ep(cl(gl)),arg)\ )$$
$$= g(\ arg(ant(gl)),\ IN\_GL\_VAR(ant(gl),var),\ OUT\_EP\_ARG(ep(ant(gl)),arg)\ )$$

Función **IN_EP_ARG(ep, arg) → dterm**

Esta función determina el dominio de términos (*dterm*) que posee un argumento (*arg*) en el momento de iniciar la ejecución de un punto de entrada (*ep*) y, por tanto, la ejecución de cualquiera de las cláusulas que pertenezcan al punto de entrada.

La evaluación de esta función consiste en determinar la reunión (operador *g*) del dominio de términos asociado al mismo argumento de todos los objetivos padre del punto de entrada (función *IN_GL_ARG*).

$$IN\_EP\_ARG(ep,arg) =$$
$$g(\ IN\_GL\_ARG(gl(ep),arg)\ )$$

Función **OUT_EP_ARG(ep,arg) → dterm**

Esta función determina el dominio de términos (*dterm*) de un argumento (*arg*) en una solución cualquiera de los objetivos padre de un punto de entrada (*ep*).

La evaluación de esta función, dado un argumento concreto, consiste en la reunión del dominio de términos del mismo argumento en todas las cláusulas pertenecientes al punto de entrada (funciones *OUT_CL_ARG*)

$$OUT\_EP\_ARG(ep,arg) =$$
$$g(\ OUT\_CL\_ARG(cl(ep),arg)\ )$$

Función **OUT_CL_ARG(cl,arg) → dterm**

Esta función indica el dominio de términos (*dterm*) que posee un argumento (*arg*) en la solución, establecida por una cláusula (*cl*), de los objetivos padre del punto de entrada al que pertenece la cláusula.

Para evaluar esta función se precisa comprobar los argumentos de la cabecera, *arg(hd(cl))*. Cuando se utiliza una variable en un argumento, esta función depende del dominio de términos que posea la variable en una solución del último objetivo del cuerpo de la cláusula (función *OUT_CL_VAR*).

$$OUT\_CL\_ARG(cl,arg) =$$
$$g(\ arg(hd(cl)),\ OUT\_CL\_VAR(cl,var)\ )$$

Función **OUT_CL_VAR(cl,var) → dterm**

Esta función indica el dominio de términos (*dterm*) que posee una variable (*var*) visible dentro de una cláusula (*cl*) después de haber satisfecho el último objetivo de la cláusula, es decir, en una solución del último objetivo.

La definición de esta función es análoga a la de la función IN_GL_VAR establecida para un hipotético objetivo existente después del último objetivo de la cláusula.

$$OUT\_CL\_VAR(cl,var) =$$
$$= g(\ arg(last(gl)),\ IN\_GL\_VAR(last(gl),var),\ OUT\_EP\_ARG(ep(last(gl)),arg)\ )$$

Función **INEP_GL_ARG(gl,arg) → dterm**

Esta función determina el dominio de términos (*dterm*) de un argumento (*arg*) en el momento de iniciar la ejecución de la cláusula a la que pertenece el objetivo (*gl*). Existe una función en FUNCT que ya determina el mismo tipo de información (IN_EP_ARG). La diferencia radica en que la función INEP_CL_ARG es capaz de afinar más el dominio de términos por el hecho de haber resuelto con éxito los objetivos anteriores a gl (en caso de que gl sea el primer objetivo del cuerpo, afina el dominio de términos por el hecho de que la unificación de la cabecera de la cláusula se ha realizado con éxito). Esta función es la que proporciona información que permite la inserción de operadores de poda en el cuerpo de una cláusula.

El cuerpo de esta función depende de los argumentos de la cabecera de la cláusula, *arg(hd(cl(gl)))*, del dominio de términos del mismo argumento en el momento de iniciar la ejecución de un punto de entrada (función *IN_EP_ARG*) y del dominio de términos de las variables que aparecen en la cabecera durante la activación del objetivo *gl* (función *IN_GL_VAR*).

$$INEP\_GL\_ARG(gl,arg) =$$
$$g(\ arg(hd(cl(gl))),\ IN\_EP\_ARG(ep(cl(gl)),arg),\ IN\_GL\_VAR(gl,var)\ )$$

Función **INEP_CL_ARG(gl,arg) → dterm**

Esta función determina la misma información que la función INEP_GL_ARG, es decir, el dominio de términos (*dterm*) de un argumento (*arg*) en el momento de iniciar la ejecución de la cláusula (*cl*), pero teniendo en cuenta que este dominio de términos se afina por el hecho de haber satisfecho completamente todos los objetivos de dicha cláusula.

La definición de esta función es análoga a la función INEP_GL_ARG, con la diferencia que el dominio de términos asociado a las variables de la cabecera viene indicado por la función *OUT_CL_VAR*:

$$INEP\_CL\_ARG(gl,arg) =$$
$$g(\ arg(hd(cl)),\ IN\_EP\_ARG(ep(cl),arg),\ OUT\_CL\_VAR(cl,var)\ )$$

4.1.2 Operaciones de la interpretación abstracta.

Una vez descritos los elementos que forman el estado de la ejecución durante la interpretación abstracta se pasa a describir las operaciones que lo procesan. En la figura 4.2 se muestra la secuencialidad existente entre estas operaciones y, a continuación, se describe la funcionalidad de dichas operaciones.

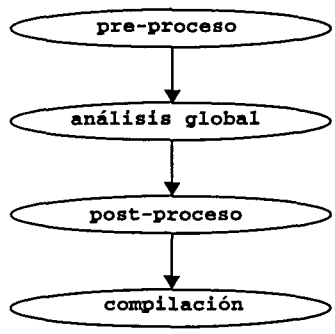


Figura 4.2: *Operaciones de la fase de interpretación abstracta.*

4.1.2.1 Pre-proceso

Permite inicializar los elementos del estado de la ejecución: PRG, INFO e FUNCT. PRG se inicializa a partir del programa Prolog fuente con las siguientes características: existe un único punto de entrada en cada procedimiento, y cada punto de entrada no tiene establecida la función de indexación. INFO no se inicializa ya que su valor se calcula una vez evaluadas las funciones FUNCT. Por su parte, estas funciones FUNCT se inicializan según la forma de representarlas, que es una decisión a establecer en el Modelo Arquitectónico de Multipath.

La inicialización de FUNCT permite codificar el programa original de forma que la ejecución de estas funciones va a proporcionar la información dinámica referente al determinismo del programa. En este sentido, la codificación del cuerpo de las funciones puede considerarse

una técnica que aplica las ideas propuestas mediante la *compilación abstracta* [10], tendentes a reducir la penalización por la consulta del programa Prolog original durante el análisis global.

4.1.2.2 Análisis global

La realización del análisis global del programa puede describirse como la evaluación de todas las funciones FUNCT que caracterizan el programa fuente. El algoritmo concreto utilizado para evaluar dichas funciones es una cuestión transparente a nivel de Modelo de Ejecución (lo importante es definir el resultado que calculan, no cómo obtenerlo) y, por tanto, se determina en el Modelo Arquitectónico.

Aparte de conseguir la evaluación de las funciones, en esta operación se debe realizar una tarea adicional. Esta tarea se puede considerar como un efecto lateral que provoca la evaluación de la función SUCC_CL y es la responsable de determinar los puntos de entrada del programa. En concreto, si la evaluación de las condiciones de éxito de una cláusula para un cierto objetivo de entrada establece que la cláusula va a fracasar siempre, se elimina del punto de entrada el objetivo que provoca el fracaso y se crea un nuevo punto de entrada que tiene asociado únicamente dicho objetivo, donde se ha eliminado la cláusula que fracasa de la lista de cláusulas candidatas a resolverlo.

4.1.2.3 Post-proceso.

Permite acabar de realizar las tareas que se pretenden obtener mediante la interpretación abstracta, una vez finalizada la caracterización dinámica del programa. En concreto, determina la conveniencia de insertar instrucciones de corte en el cuerpo de una cláusula, la conveniencia de realizar la función de indexación durante la interpretación abstracta y el atributo de indeterminismo asociado a cada objetivo del programa.

La inserción de operadores de corte en el cuerpo de la cláusula se realiza si el dominio de términos en el momento de iniciar la ejecución de un punto de entrada, restringido por el hecho de haber realizado con éxito la unificación de la cabecera o haber satisfecho un objetivo del cuerpo permite asegurar que el resto de cláusulas van a fracasar. El dominio de términos restringido lo determinan las funciones INEP_CL_ARG o INEP_GL_ARG. Las condiciones de éxito del resto de las cláusulas están establecidas por la función SUCC_CL.

La determinación de si es conveniente realizar indexación en un punto de entrada se realiza a partir del dominio de términos que posee cada argumento a la hora de activar un punto de entrada (función IN_EP_ARG) y de las condiciones de éxito de las cláusulas que lo

integran (función SUCC_CL). Para determinar si se realiza la indexación se debe comprobar si existe algún argumento que, dado el dominio de términos que posee, permite evitar la utilización de alguna cláusula durante la unificación. Para identificar el argumento concreto, se debe detectar el que permite eliminar, en promedio según las distintas salidas de la indexación, un mayor número de cláusulas.

La determinación del atributo de indeterminismo de cada objetivo del programa se realiza a partir de la función NSOL_EP. Según el número de soluciones que tiene un punto de entrada y las condiciones que deben cumplir los argumentos para obtenerlas, se calcula un mismo atributo de indeterminismo para todos los objetivos padre de dicho punto de entrada.

4.1.2.4 Compilación.

MEM define que el programa Prolog transformado (PRG) y la información obtenida (INFO) forma la entrada de la posterior fase de interpretación real. Esta operación realiza la función de interface para traducir el programa Prolog y la información adicional en el formato adecuado para su consulta en la fase de interpretación real.

4.2 Interpretación real

La segunda fase del Modelo de Ejecución de Multipath consiste en la interpretación real del programa. Esta fase se realiza a partir de la salida obtenida en la interpretación abstracta y se caracteriza por recorrer el árbol de búsqueda mediante la exploración parcial en anchura definida en el capítulo anterior.

El número concreto de soluciones a calcular para cada objetivo se determina dinámicamente durante la activación del objetivo y en la obtención de cada una de sus soluciones. El criterio se establece en función de un límite máximo de caminos visibles durante la ejecución. Este límite es un **parámetro** de la fase de interpretación abstracta, denominado **MAX_CAMINOS.**

Del mismo modo que en el capítulo 2 se realizó la descripción axiomática del modelo de ejecución convencional de Prolog, a continuación se describe de forma también axiomática la fase de interpretación real del modelo de ejecución de Multipath. En esta descripción, se identifican los elementos que forman parte del estado de la ejecución y las distintas operaciones que lo modifican. A lo largo de esta descripción se utiliza, siempre que sea posible, la misma nomenclatura que la realizada en el modelo convencional.

4.2.1 Estado de la ejecución durante la interpretación real

El estado de la ejecución en la fase de interpretación real viene determinado por los siguientes elementos:

$$EJ_{IR} \equiv \{BD,\ OAA,\ OPA,\ EAV,\ AP,\ COA\}$$

cuya definición y funcionalidad se describe a continuación.

4.2.1.1 Base de Datos (BD)

Contiene toda la información que caracteriza el conocimiento de un programa, obtenida tras la etapa de compilación de la fase de interpretación abstracta.

La notación que describe el contenido de BD sigue el mismo formato utilizado para describir el elemento PRG de la fase de interpretación abstracta al que se le añade la información referente al atributo de indeterminismo (función ATTR) en cada objetivo del programa:

$$BD \equiv \{ep_1,\ \ldots,\ ep_i\}$$

$$EP \equiv \{\ [cl_1,\ \ldots,\ cl_j]\ |\ \{arg,\ \{\{term_1,\ [cl_{11},\ \ldots,\ cl_{1j}]\},\ \ldots,\ \{term_n,\ [cl_{n1},\ \ldots,\ cl_{nj}]\}\ \}\ \}$$

$$CL \equiv [hd,\ gl_1,\ \ldots,\ gl_k]$$

$$HD \equiv [arg_1,\ \ldots,\ arg_{hd}]$$

$$GL \equiv \{ep,\ [arg_1,\ \ldots,\ arg_{gl}]\}$$

$$ATTR(gl) = \{NONDET,\ SEMIDET,\ OTHER\}$$

4.2.1.2 Objetivo en anchura actual (OAA)

Se define un **objetivo en anchura** (o también, objetivo explorado en anchura) como aquel objetivo del programa para el cual se decide intentar calcular más de una solución después de su activación. Si se decide calcular una única solución en la activación de un objetivo, se trata de un **objetivo en profundidad** (o explorado en profundidad). El elemento OAA del estado de la ejecución especifica el objetivo en anchura más joven que se está satisfaciendo en un momento dado.

Un objetivo en anchura se identifica completamente en tiempo de ejecución mediante la continuación del objetivo (OP). La continuación de un objetivo contiene la lista de objetivos pendientes de resolver una vez el objetivo se ha solucionado. Por ello, OAA es descrito como

$$OAA \equiv [gl_1, \ldots, gl_{oaa}]$$

Nótese que el último objetivo del cuerpo de una cláusula que satisface un objetivo en anchura y este mismo objetivo comparten la misma continuación. Sin embargo, no es necesario diferenciar los dos objetivos en tiempo de ejecución ya que el objetivo interno nunca va a ser explorado en anchura. Esto es debido a que su exploración en anchura y en profundidad coincide y, en estos casos, se opta siempre por explorar el objetivo más joven en profundidad.

4.2.1.3 Objetivos pendientes actuales (OPA)

Especifica la lista de objetivos que faltan por satisfacer para, a su vez, satisfacer el objetivo en anchura actual OAA.

$$OPA \equiv [gl_1, \ldots, gl_{opa}]$$

4.2.1.4 Entornos Actuales de Vínculos (EAV)

Contiene todos los vínculos de variables, obtenidos desde el inicio del programa, que son visibles en cada uno de los caminos actuales, o caminos recorridos simultáneamente en un momento de la ejecución. EAV, al igual que todos los entornos de vínculos que aparecen en el modelo de ejecución, está compuesto de tantos **entornos locales** (EVL) como caminos actuales se están recorriendo (NCA), más un **entorno global** (EVG) a todos los caminos. Los entornos locales almacenan los vínculos que son visibles a un único camino del árbol de búsqueda y el entorno global almacena los vínculos que son comunes a todos los caminos:

$$EAV \equiv \{\{EVL_1, \ldots, \{\ldots, variable/term, \ldots\}, \ldots, EVL_{NCA}\}, EVG\}$$

La decisión de dividir el entorno de vínculos en estos dos componentes es crucial de cara a poder obtener un buen rendimiento del sistema. Precisamente, las ventajas de la exploración en anchura radican en la disminución de las operaciones de control y de datos comunes. La cantidad de variables que tienen un vínculo común en todos los caminos recorridos en un momento de la ejecución es lo suficientemente elevada como para tratarlas de forma diferente al resto de variables. En un primer prototipo de Multipath se comprobó experimentalmente la pérdida de rendimiento si se consideraba que todas las variables tenían únicamente vínculos locales.

4.2.1.5 Alternativas Pendientes (AP)

Es una lista ordenada de elementos de dos tipos distintos, denominados **puntos de selección**

y **puntos de indexación.** El criterio de ordenación establece que el primer elemento de la lista es el último introducido.

Un punto de selección (cp) contiene elementos OA, OP y EV. Cada uno de estos elementos coincide con los elementos OAA, OPA, EAV del estado de la ejecución en un momento anterior del programa en el que se determina que existen otras alternativas para intentar resolver un objetivo.

Un punto de indexación (ip) especifica, además de un OA y un OP, un conjunto conteniendo grupos de los caminos de entrada (EV_i), cada uno de ellos con un término Prolog distinto ($term_i$) en el argumento que se ha realizado la indexación. Cada uno de estos grupos de caminos requiere un flujo de control distinto para resolver un cierto objetivo.

$$AP \equiv [\ldots, \{cp, OA, OP, EV\}, \ldots, \{ip, OA, OP, \{\ldots, \{term_i, EV_i\}, \ldots\}\}, \ldots]$$

4.2.1.6 Conjunto de Objetivos en anchura (COA)

Es un conjunto que contiene elementos {OA, EV, OAS}. Cada uno de estos elementos proporciona información sobre objetivos en anchura (OA) con posibilidad de encontrar más soluciones. EV contiene los vínculos realizados en caminos que son solución del objetivo. La exploración de estos caminos está suspendida en espera de poder conseguir un mayor número de soluciones. OAS identifica al siguiente objetivo en anchura que tiene que resolverse una vez la ejecución prosiga con la continuación de este objetivo.

$$COA \equiv \{\ldots, \{OA, EV, OAS\}, \ldots\}$$

4.2.1.7 Estado inicial y final de la ejecución

Inicialmente, el estado de la ejecución determina que BD contiene todos los puntos de entrada determinados en la interpretación abstracta; OAA corresponde al programa (su continuación es la lista vacía); OPA está formado por los objetivos de la pregunta inicial en el mismo orden en que aparecen en el programa; EAV contiene un entorno local vacío y el entorno global también vacío, que corresponde a explorar un único camino del árbol de búsqueda; AP es una lista vacía; y COA es un conjunto vacío:

$$EJ_{IR,ini} \equiv \{\ BD,\ OAA \equiv [],\ OPA \equiv [pregunta],\ EAV \equiv \{\{\emptyset\}, \emptyset\},\ AP \equiv [],\ COA \equiv \emptyset\}$$

Se obtienen soluciones del programa cuando no existe ningún objetivo pendiente en OPA para satisfacer el objetivo en anchura OAA que identifica el programa. En esta situación, el

número de soluciones del programa viene determinado por el número de entornos locales existentes en EAV. Cada solución se identifica por los vínculos visibles en su entorno local y en el entorno global. No existe posibilidad de encontrar más soluciones cuando AP es una lista vacía y COA es un conjunto vacío:

$$EJ_{IR,sol} \equiv \{ OAA \equiv [], OPA \equiv \emptyset \}$$

4.2.2 Operaciones de la interpretación real

El Modelo de Ejecución de Multipath intenta continuamente realizar inferencias hasta encontrar una o todas las soluciones del programa, según se indique en la pregunta inicial. En la figura 4.3 se muestra el diagrama de flujo de las operaciones de Multipath, que son descritas a continuación. A lo largo de esta descripción, también se indica la representación de la ejecución mediante el denominado árbol de búsqueda Multipath y el orden en recorrerlo.

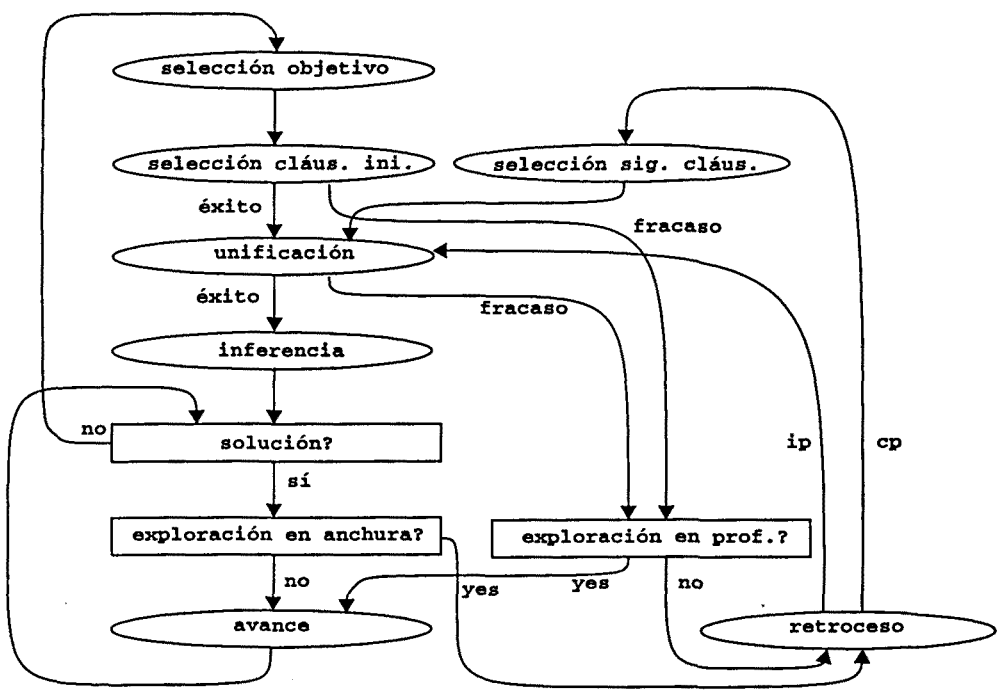


Figura 4.3: *Operaciones que conforman el bucle básico de la fase de interpretación real.*

4.2.2.1 Selección de Objetivo

Es idéntica a la del modelo convencional: se selecciona el primer objetivo de OPA. Suponiendo que OPA está formado por los siguientes objetivos OPA = [gl_1, gl_2, ..., gl_i], esta acción se

simboliza como:

$$objetivo \equiv gl_1$$

Este comportamiento corresponde a un secuenciamiento implícito en la ejecución de los objetivos según el mismo orden de aparición textual en el cuerpo de las cláusulas de un programa. La siguiente operación a realizar consiste en la *Selección de Cláusula Inicial.*

4.2.2.2 Selección de Cláusula Inicial

Este momento de la ejecución corresponde a la activación de un objetivo. Su función es escoger una cláusula para intentar la resolución del objetivo escogido anteriormente. La secuencia de acciones que se realiza en esta operación se detalla seguidamente.

En primer lugar, consulta el punto de entrada asociado al objetivo gl_1 y comprueba si se debe realizar indexación.

En el supuesto que se realice la indexación, se obtiene el término que posee el argumento a indexar en cada uno de los caminos actuales (estos caminos de entrada al objetivo se identifican mediante cada uno de los entornos locales de EAV). Según el término obtenido, se descomponen todos los caminos de entrada al objetivo en varios grupos: un grupo corresponde a los caminos que fracasan tras la indexación, es decir, el término no aparece en la lista de términos permitidos en el punto de entrada consultado; y cada uno del resto de grupos engloba a todos los caminos de entrada con el mismo término para el argumento que se indexa. A continuación, se eliminan todos los entornos locales de EAV correspondientes a los caminos que fracasan. Si no existe ningún grupo de caminos que haya pasado con éxito la indexación, la ejecución continúa con la comprobación de la condición de *Exploración en Profundidad?.* Si el número de grupos que no fracasan es mayor que uno, se selecciona un grupo de caminos y el resto de grupos forman un punto de indexación que se añade a AP. Obsérvese que el orden de recorrido de los grupos no afecta al modelo de ejecución. El nuevo EAV estará formado por el EV correspondiente al grupo de caminos seleccionado:

$$AP \leftarrow [\{ip, OAA, OPA, \{\{term_1, EV_1\}, \ldots, \{term_n, EV_n\}\}\} \mid AP]$$
$$EAV \leftarrow EV_0$$

La representación de un punto de indexación en el árbol de búsqueda se realiza con un nodo que posee tantas ramas hijas como grupos de caminos que no fracasan. Además, una raya horizontal abarca todas las ramas hijas (sirve para distinguirlo de un punto de selección).

Una vez determinados los caminos actuales, es preciso decidir la cláusula con la que intentar resolver el objetivo seleccionado en la operación anterior. Se selecciona la primera cláusula de la lista correspondiente, tras la consulta del punto de entrada en la BD. Esta cláusula se simboliza mediante una cabecera (hd) y los objetivos del cuerpo de la cláusula (gl_{bi}):

$$cláusula \equiv [hd, gl_{b1}, \ldots, gl_{bj}]$$

En caso de no existir ninguna cláusula candidata a unificar se produce un fracaso y el modelo de ejecución prosigue con la evaluación de la condición *Exploración en Profundidad?*. Si existe más de una cláusula candidata para realizar la unificación se crea un punto de selección en AP:

$$AP \leftarrow [\{cp, OAA, OPA, EAV\} \mid AP]$$

Un punto de selección se representa mediante un nodo con tantas ramas hijas como cláusulas candidatas existan para realizar la unificación.

En este punto de la ejecución se determina **si el objetivo va a ser explorado en profundidad o en anchura**. Esta decisión se toma en función del atributo de indeterminismo que posee el objetivo, del número de caminos actuales y del parámetro MAX_CAMINOS. El criterio que determina si un objetivo se explora en anchura comprueba que se cumpla uno de los tres casos siguientes:

- Tras la indexación de un objetivo se obtiene más de un grupo de caminos de entrada al objetivo.
- El atributo de indeterminismo es SEMIDET y el número de caminos actuales es mayor que uno.
- El atributo es NONDET y el número de caminos disponibles, en función del límite MAX_CAMINOS, es mayor o igual que el número de caminos actuales.

Estas tres condiciones reflejan los tres tipos de objetivos en que se aplican las ventajas que conlleva la exploración en anchura. Es recomendable recorrer en anchura todos los grupos de caminos en un objetivo con punto de indexación. La segunda condición evita la exploración en anchura de un objetivo semideterminista cuando se explora un único camino. La tercera condición evita la exploración en anchura de un objetivo indeterminista cuando no existen suficientes caminos disponibles para encontrar un mínimo de dos soluciones para todos los caminos de entrada.

En caso de permitir la exploración en anchura del objetivo, se inserta un nuevo elemento

en COA, donde OA es el objetivo elegido, EV no contiene ninguna solución y OAS corresponde al objetivo en anchura actual OAA. A continuación, se modifican OAA y OPA:

$$COA \leftarrow COA \cup \{gl_1, \{\{\emptyset\}, \emptyset\}, OAA\}$$
$$OAA \leftarrow [gl_2, \ldots, gl_i \mid OAA]$$
$$OPA \leftarrow [gl_1]$$

En el árbol de búsqueda, el inicio de la exploración de un objetivo en anchura se asocia al primer punto de indexación o de selección que crea el objetivo, cuyo nodo se representa de forma triangular.

La siguiente operación del modelo de ejecución es la *Unificación.*

4.2.2.3 Unificación

Esta operación realiza la unificación de los argumentos del objetivo y de la cabecera de la cláusula elegidos en las dos acciones anteriores. Como resultado de esta operación se indica, para cada camino en que finaliza con éxito la unificación, el unificador más general (mgu). Téngase en cuenta que el mgu es un entorno de vínculos y, por tanto, se descompone en entornos locales (θ_i) y en un entorno global (θ):

$$unifica(gl_1, hd) \equiv mgu = \{\{\theta_1, \ldots, \theta_{NCA}\}, \theta\}$$

El orden de las unificaciones de los argumentos es transparente a nivel de modelo de ejecución. No obstante, no se contempla la simultaneidad entre estas unificaciones, es decir, la posibilidad de explotar paralelismo de unificación. Inicialmente, el mgu contiene tantos entornos locales vacíos como caminos actuales.

Durante la unificación de un argumento se instancian los vínculos existentes en EAV y en el mgu sobre las variables que aparecen en los dos términos Prolog. Si en el proceso de instanciación no se obtienen vínculos que son locales a caminos del árbol de búsqueda, se realiza una única **unificación global** para todos los caminos actuales. Cuando una variable no tiene vínculos locales ni un vínculo global (la variable es libre en todos los caminos) también se realiza una unificación global.

En el momento en que, en alguno de los dos términos a unificar, se obtiene un vínculo local, la unificación pasa a realizarse de forma local para cada uno de los caminos actuales. Estas **unificaciones locales** pueden efectuarse simultáneamente, y constituyen un nuevo tipo de paralelismo implícito en el modelo de ejecución que se denomina **paralelismo de caminos.**

Cada una de estas unificaciones locales es independiente del resto.

Los entornos locales de EAV y de mgu correspondientes a los caminos que han fracasado en la unificación de un argumento son eliminados. Téngase en cuenta que el número de caminos actuales siempre coincide con el número de entornos locales.

El modelo de ejecución define un **criterio óptimo para obtener el mínimo número de variables que poseen vínculos locales**. Este criterio óptimo se basa en: (i) retrasar el paso de vínculos globales a vínculos locales hasta el momento en que se referencia una variable que tiene vínculos visibles en otros caminos solución del mismo objetivo en anchura; y (ii) permitir el paso de vínculos locales iguales a un único vínculo global. Este criterio se aplica en tres operaciones del modelo de ejecución: en la *unificación*, en el *retroceso* y en el *avance*. A continuación se describe la aplicación de este criterio durante la unificación.

- Los vínculos obtenidos en las unificaciones locales se almacenan en los entornos locales del mgu (θ_i).
- Si los vínculos obtenidos en la unificación global corresponden a variables visibles en otros caminos solución del mismo objetivo en anchura actual, se almacenan en θ_i; en caso contrario, se almacenan en el entorno global del mgu (θ).

Si la unificación fracasa en todos los caminos actuales pasa a comprobarse la condición de *Exploración en Profundidad?*. Si la unificación tiene éxito, como mínimo en un camino, pasa a realizarse la operación de *Inferencia*.

4.2.2.4 Inferencia

Se producen tantas inferencias como caminos actuales tienen éxito tras la unificación. Se añaden los vínculos locales θ_i obtenidos durante la unificación en cada EVL_i. El entorno global EVG también se actualiza con los vínculos globales θ que se hayan obtenido.

Por otra parte, se substituye el objetivo seleccionado de OPA por los objetivos del cuerpo de la cláusula elegida para intentar su satisfacción:

$$EAV \leftarrow \{\{EVL_1 \cup \theta_1, \ldots, EVL_{NCA} \cup \theta_{NCA}\}, EVG \cup \theta\}$$
$$OPA \leftarrow [gl_{b1}, \ldots, gl_{bj}, gl_2, \ldots, gl_i]$$

4.2.2.5 Solución?

En el momento que OPA no contiene ningún objetivo significa que se ha encontrado una solu-

ción de OAA. Este objetivo puede ser el programa o bien un objetivo parcial del programa explorado en anchura.

$$solución \equiv (OPA = [])$$

Cuando se encuentra una solución, los caminos actuales se añaden a los caminos solución en el elemento de COA correspondiente al objetivo en anchura explorado actualmente (OAA). Obsérvese que la reunión de los caminos solución ya existentes con los caminos actuales implica ampliar el número de entornos locales solución y aplicar la unión de los dos entornos globales:

$$COA(OA=OAA).EV \leftarrow COA(OA=OAA).EV + EAV$$
$$+(EVL_1, EVL_2) = \{EVL_{11}, ..., EVL_{1i}, EVL_{21}, ..., EVL_{2j}\}$$
$$+(EVG_1, EVG_2) = EVG_1 \cup EVG_2$$

La representación de una solución en el árbol de búsqueda se realiza mediante un nodo de forma cuadrangular. La ejecución prosigue con la comprobación de si continúa o no la *exploración en anchura* del objetivo.

4.2.2.6 Exploración en Anchura?

Esta función comprueba las condiciones requeridas para intentar el cálculo de una nueva solución del objetivo en anchura que se acaba de solucionar. Las condiciones que se evalúan en esta operación determinan una **exploración en anchura** denominada **perezosa y parcial**.

La exploración perezosa en anchura establece que se permite el cálculo de una nueva solución si:

- La primera alternativa de AP está dentro del ámbito del objetivo que se acaba de solucionar, es decir, la alternativa debe ser más joven que el punto de indexación o de selección que inició el objetivo en anchura.

Existe otra posibilidad más avariciosa que puede explotar en mayor grado las ventajas que supone la exploración en anchura. La exploración avariciosa en anchura se podría realizar cuando la alternativa más joven puede conducir a intentar satisfacer el mismo objetivo del cual se ha encontrado una solución, incluyendo la posibilidad que se intente iniciar otra vez la activación u operación de *selección de objetivo* del mismo objetivo en anchura que se ha solucionado.

La exploración en anchura parcial significa que no se intenta encontrar ciegamente todas las soluciones del objetivo en anchura actual. Esta restricción depende del parámetro

MAX_CAMINOS, y su objetivo es la adaptación dinámica de la exploración en función de la penalización que pueda causar el cálculo de un número excesivo de soluciones. El modelo de ejecución deja abierta la posibilidad que este parámetro MAX_CAMINOS pueda ser calculado automáticamente según el comportamiento de cada programa a ejecutar. En concreto, la exploración parcial en anchura establece que se permite el cálculo de una nueva solución si:

- La alternativa más joven de AP es un punto de selección de un objetivo con atributo NONDET y el número total de caminos visibles que se obtendría al explorar en anchura dicha alternativa no excede MAX_CAMINOS.
- La alternativa más joven de AP pertenece a un objetivo SEMIDET o existe alguna alternativa en AP que corresponde a un punto de indexación. En estos casos nunca se necesita incrementar el número total de caminos activos y, por tanto, siempre se permite su exploración en anchura.

En caso que se cumpla alguna de estas dos condiciones, la ejecución continúa con la exploración en anchura de la alternativa de AP elegida (operación denominada *Retroceso*). En caso que no se cumpla ninguna de las condiciones, la ejecución continúa con una exploración en profundidad (operación denominada *Avance*).

4.2.2.7 Avance

Esta operación se realiza cuando se determina que, después de obtener una solución de un objetivo en anchura, la ejecución debe proseguir con un recorrido en profundidad, intentando resolver la continuación de este objetivo.

Su realización consiste en actualizar EAV, OPA y OAA de la siguiente forma. Se establecen como Entornos Actuales de Vínculos (EAV) todos los entornos de vínculos asociados a los caminos solución del objetivo en anchura actual; se establece como objetivo en anchura actual (OAA) el objetivo siguiente, indicado en COA mediante el elemento OAS; y se establece como objetivos pendientes actuales (OPA) la lista que resulta de eliminar a los objetivos pendientes del OAA anterior, los objetivos pendientes del actual:

$$EAV \leftarrow COA(OA=OOA).EV$$

$$OPA \leftarrow COA(OA=OOA).OA.OP - COA(OA=OOA).OAS.OP$$

$$OAA \leftarrow COA(OA=OOA).OAS$$

En referencia al criterio de clasificación de vínculos en locales o globales, introducido en la operación de unificación, se aplica la siguiente acción:

- Si todos los vínculos locales de una misma variable son iguales en todos los caminos actuales, se eliminan de los entornos locales y se introduce el vínculo en el entorno global.

La acción de avance se representa en el árbol de búsqueda mediante líneas discontinuas que agrupan las soluciones obtenidas del objetivo en un único flujo de control.

4.2.2.8 Exploración en profundidad?

Esta función se ejecuta cuando fracasa la unificación de un objetivo con la cabecera de una cláusula y determina si la ejecución debe avanzar (exploración en profundidad) o debe retroceder en el árbol de búsqueda.

La posibilidad de continuar con una exploración en profundidad es debida a la existencia de objetivos en anchura con soluciones ya encontradas y que no poseen alternativas pendientes que permitan continuar su exploración en anchura. Esta función comprueba esta posibilidad y, en caso de encontrar un objetivo de este tipo, la ejecución prosigue con la acción de avance. En caso contrario, se retrocede en el árbol de búsqueda.

4.2.2.9 Retroceso

Esta acción se ejecuta por dos causas: (i) tras un fracaso en la unificación y haber determinado que la ejecución debe retroceder en el árbol, o (ii) tras encontrar una solución de un objetivo de COA y permitir continuar la ejecución en anchura de este objetivo.

En referencia al criterio de clasificación de vínculos en locales o globales, introducido en la operación de unificación, se aplica la siguiente acción:

- Todos los vínculos globales del objetivo en anchura actual (OAA), almacenados en COA, que no aparecen en el EVG de la alternativa de AP que se explorará, pasan a ser vínculos locales. Se eliminan del entorno global de los caminos solución y se introducen en cada entorno local.

Posteriormente, se restauran OAA y OPA a partir del elemento de AP hacia el que se debe retroceder. El comportamiento posterior depende del tipo de alternativa de este elemento de AP.

Si la alternativa corresponde a un punto de indexación, se escoge el siguiente grupo de caminos, es decir, se restaura EAV a partir del EV_i de la siguiente alternativa del punto de indexación. Si el grupo de caminos elegido es el último, se elimina el punto de indexación de

AP. En función del término Prolog del grupo de caminos, se consulta en BD el punto de entrada asociado al primer objetivo de OPA y se obtiene la lista de cláusulas candidatas a la unificación. Si existe más de una cláusula se introduce un punto de selección en AP. Posteriormente, se obtiene la primera cláusula y la ejecución continúa con la *Unificación* del objetivo almacenado en el punto de indexación y la cabecera de esta cláusula.

Si la alternativa corresponde a un punto de selección, se restaura EAV a partir del EV del punto de selección y la ejecución prosigue con la operación denominada *Selección de Siguiente Cláusula.*

Los elementos pertenecientes a COA pueden ser eliminados siempre que no contengan caminos solución (el número de entornos locales es 0) y no exista la posibilidad de encontrar nuevas soluciones. Estas condiciones se detectan en las operaciones de avance y de retroceso.

4.2.2.10 Selección de Siguiente Cláusula

Selecciona la siguiente cláusula para intentar la unificación con el objetivo de más a la izquierda de OPA. En caso de ser la última cláusula, se elimina el punto de selección de AP.

4.3 Ejemplo

En esta sección se muestra un ejemplo que representa las acciones que se realizan durante las fases de interpretación abstracta e interpretación real para el mismo programa *grandparent* utilizado en la figura 2.2 como ejemplo de la ejecución realizada por el modelo convencional de Prolog.

```
?- grandparent(X,tom).

grandparent(X,Y) :- parent(Z,Y), parent(X,Z).

parent(mary,tom).
parent(john,tom).
parent(alice,john).
parent(paul,john).
```

En primer lugar, la interpretación abstracta transforma el programa e inserta información adicional. La salida de esta fase se representa en el formato adecuado para su posterior ejecución. A nivel de modelo de ejecución, el resultado de la interpretación abstracta se representa con el mismo lenguaje Prolog junto con directivas que proporcionan la información referente al

atributo de indeterminismo. A continuación, se muestra el resultado para el programa *grandparent*:

```
main_ep1 :- main_cl1.
main_cl1 :- grandparent_ep1(X,tom).

grandparent_ep1(A1,A2) :- grandparent_cl1(A1,A2).
:- attr(grandparent_cl1/2, 1, NONDET).
:- attr(grandparent_cl1/2, 2, NONDET).
grandparent_cl1(X,Y) :- parent_ep1(Z,Y), parent_ep2(X,Z).

parent_ep1(A1,A2) :- parent_cl1(A1,A2); parent_cl2(A1,A2).

parent_ep2(A1,A2) :-
                    (A2 = tom -> (parent_cl1(A1,A2), parent_cl2(A1,A2));
                    (A2 = john -> (parent_cl3(A1,A2), parent_cl3(A1,A2)) )).

parent_cl1(mary,tom).
parent_cl2(john,tom).
parent_cl3(alice,john).
parent_cl4(paul,john).
```

Las principales características a comentar son:

- Los dos objetivos dentro de la cláusula *grandparent_cl1/2* son indeterministas: cada uno de estos objetivos posee más de una solución, al tener el primer argumento no instanciado y el segundo argumento instanciado a una constante que tiene éxito en dos cláusulas.
- Se han obtenido dos puntos de entrada en el procedimiento *parent/2*. El punto de entrada *parent_ep1/2* está asociado al primer objetivo indeterminista y considera que el primer argumento corresponde a una variable libre y el segundo argumento está instanciado a la constante *tom*. En este punto de entrada se restringe la lista de cláusulas candidatas a unificar, conteniendo únicamente las cláusulas *parent_cl1/2* y *parent_cl2/2*. El punto de entrada *parent_ep2/2* está asociado al segundo objetivo indeterminista y considera que el primer argumento corresponde a una variable libre y el segundo argumento a una constante. En este caso se define una función de indexación que, aplicada sobre el segundo argumento, indica la lista de cláusulas candidatas según la constante específica que contenga dicho argumento.

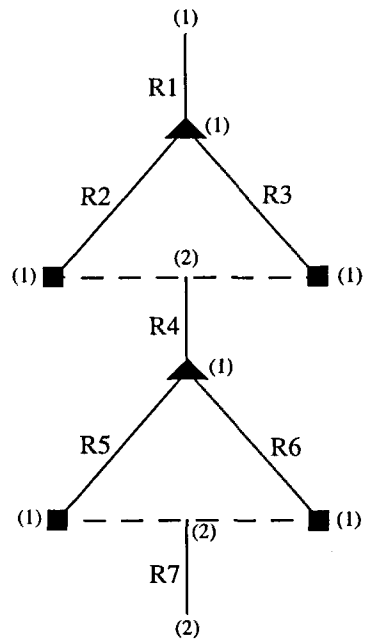


Figura 4.4: *Árbol Multipath representativo de la interpretación real del programa ejemplo grandparent.*

En segundo lugar, la fase de interpretación real del programa es la encargada de realizar la ejecución del programa. Esta ejecución se simboliza mediante el árbol Multipath de la figura 4.4. Este árbol se obtiene con cualquier valor del parámetro MAX_CAMINOS mayor que 1. En el árbol se incluye también el orden de recorrido de las ramas (Ri) y el número de caminos actuales (NCA) en varios puntos significativos. Este número de caminos se indica encerrado entre paréntesis.

Las principales características a comentar son:

- El final de la rama R1 se produce en el momento en que se selecciona una cláusula para el primer objetivo indeterminista y se crea un punto de selección.
- Se obtienen secuencialmente las dos soluciones de este objetivo en las ramas R2 y R3, respectivamente.
- La rama R4 empieza con dos caminos actuales. Posteriormente, en la selección de cláusula inicial para unificar el segundo objetivo indeterminista fracasa un camino (el que posee el vínculo *X/mary*). Esta rama finaliza creando un punto de selección al tener dos cláusulas candidatas a realizar la unificación.

- Se obtienen las dos soluciones del segundo objetivo indeterminista en las ramas R5 y R6.
- La rama R7 finaliza por haber satisfecho el programa con dos soluciones (*X/alice*; *X/paul*).

4.4 Resumen y contribuciones

En este capítulo se ha presentado el Modelo de Ejecución de Multipath (MEM). Este modelo constituye un primer nivel de descripción del sistema en el que se establecen los elementos de la ejecución y las operaciones que lo actualizan en una notación axiomática. En este nivel de abstracción, lo importante es determinar el flujo de control del algoritmo de interpretación de la semántica del lenguaje, sin entrar en cuestiones de representación y realización concretas de los elementos y operaciones.

Las contribuciones que se aportan en MEM son la definición de:

- Un análisis de determinismo del programa, a partir de una compilación abstracta del programa, con el objetivo de reducir el árbol de búsqueda de su ejecución y de identificar los objetivos en que es recomendable el cálculo de más de una solución.
- Una exploración perezosa y parcial en anchura de los objetivos, que reduce el número de operaciones a realizar en la ejecución respecto la exploración convencional en profundidad.
- La posibilidad de explotar una nueva fuente de paralelismo inherente en los programas Prolog, que se ha denominado paralelismo de caminos.
- Un criterio óptimo para la determinación del menor número de variables con vínculos locales en los caminos recorridos simultáneamente.