UNIVERSIDAD POLITECNICA DE MADRID

FACULTAD DE INFORMATICA

CONTROL DE GRANULARIDAD EN LA EJECUCION PARALELA DE PROGRAMAS LOGICOS MEDIANTE TECNICAS DE ANALISIS Y TRANSFORMACION

TESIS DOCTORAL

PEDRO LOPEZ GARCIA

JUNIO 2.000

Tesis Doctoral

Control de Granularidad en la Ejecución Paralela de Programas Lógicos Mediante Técnicas de Análisis y Transformación

presentada en la
Facultad de Informática
Universidad Politécnica de Madrid
como parte de los requisitos para el título de
Doctor en Informática

Autor: Pedro López García
Licenciado en Informática
Universidad Politécnica de Madrid – UPM

Director: Manuel V. de Hermenegildo y Salinas
Catedrático de Universidad

Madrid, junio, 2.000

A mi familia

Agradecimientos

Quisiera dar las gracias a mi director de tesis, Manuel Hermenegildo, sin cuya ayuda este trabajo no hubiera sido posible, ya que me ha proporcionado todos los medios necesarios para realizarlo. Le doy las gracias además por haberme dedicado su tiempo, por sus consejos, su amistad, y sobre todo, por haberme brindado la oportunidad de poder trabajar en investigación.

Gracias también a los demás miembros del grupo CLIP (Computación Lógica, Implementación y Paralelismo), por haberme prestado su amistad y ayuda siempre que la he necesitado, haberme resuelto innumerables dudas y por sus valiosos comentarios. Especialmente gracias a Francisco Bueno, por su trabajo realizado en la integración en el sistema `ciaopp` de las herramientas implementadas.

También quisiera expresar mi agradecimiento a Saumya Debray y Nai-Wei Lin por haber realizado la implementación del sistema CASLOG y tenerla disponible, haber colaborado en muchos de los trabajos realizados y haber sido coautores de algunas publicaciones a las que ha dado lugar la presente tesis.

Mi agradecimiento también a mi familia y a todas las personas que de alguna manera, con su apoyo moral, me han ayudado a concluir esta tesis.

Resumen

Los lenguajes de programación lógica ofrecen un excelente marco para la aplicación de técnicas de paralelización automática. Por otra parte, existen resultados teóricos que pueden asegurar cuándo los programas paralelizados son correctos, es decir, obtienen los mismos resultados que los correspondientes secuenciales, y cuándo la ejecución de los primeros no tarda más tiempo que la de los segundos. Sin embargo, dichos resultados suponen un entorno ideal de ejecución paralela y no tienen en cuenta que en la práctica existen una serie de costes asociados con la ejecución paralela de tareas, como, por ejemplo, creación y gestión de tareas, posible migración de tareas a procesadores remotos, costes de comunicación, etc. Dichos costes pueden dar lugar a que la ejecución de los programas paralelos sea más lenta que la de los secuenciales, o, al menos, limitar la ganancia debida al paralelismo introducido. En esta tesis hemos desarrollado completamente (e integrado en un sistema avanzado de manipulación de programas que realiza análisis y optimización de los mismos) un sistema automático de control de granularidad para programas lógicos que estima eficientemente la granularidad de las tareas (e.d., el trabajo necesario para su ejecución completa) y la usa para limitar el paralelismo, de forma que se controla el efecto de los costes mencionados anteriormente. El sistema está basado en un esquema de análisis y transformación de programas, en el cual se realiza tanto trabajo como sea posible en tiempo de compilación para evitar añadir nuevos costes a la ejecución de los programas.

En la realización de la transformación de programas se persigue el objetivo de minimizar el trabajo adicional hecho en tiempo de ejecución, para lo que hemos propuesto una serie de técnicas, mientras que en el análisis el objetivo es obtener la información necesaria para esta fase de transformación, lo que nos ha llevado a desarrollar varios tipos de análisis capaces de inferir informaciones tales como cotas del coste de procedimientos, qué llamadas a procedimientos no fallarán, etc. El coste adicional en tiempo de ejecución asociado con la técnica que proponemos es usualmente bastante pequeño. Además realizamos un análisis estático del coste asociado con el proceso de control de granularidad, de forma que se pueda decidir su conveniencia. Finalmente mostramos que las ganancias en tiempo de ejecución obtenidas mediante la incorporación de control de granularidad son bastante buenas, especialmente para los sistemas cuyos costes asociados con la ejecución paralela varian de medios a grandes.

Finalmente, es interesante resaltar que muchas de las técnicas que hemos desarrollado tienen otras aplicaciones importantes, además del mencionado control de granularidad, entre otras: eliminación de paralelismo especulativo, detección de errores de programación, varias aplicaciones relacionadas con la transformación de programas, por ejemplo, reordenación de objetivos, o ayuda a los sistemas de transformación de programas a elegir las transformaciones óptimas, eliminación de recursividad, selección de diferentes algoritmos o reglas de control cuya eficiencia puede depender de los tamaños de los datos de entrada, depuración de eficiencia (optimización) de programas y la optimización de consultas a bases de datos deductivas. Los resultados experimentales realizados con un prototipo del sistema han mostrado que el control de granularidad puede contribuir sustancialmente a la optimización de la ejecución en paralelo de programas, especialmente en sistemas distribuidos, y que parece bastante factible un enfoque automático del mismo.

Índice general

Capítulo 1

Introducción

En el ámbito de la Inteligencia Artificial (IA) la programación lógica [Kow74, vEK76, Kow80, Col87, RN95] tiene una larga tradición de conveniencia y efectividad en la tarea de implementación de aplicaciones típicas de la IA (sistemas expertos, bases de conocimiento) generalmente complejas y con alto componente simbólico. Históricamente, la lógica se ha erigido, ya desde los primeros tiempos, en substituta ideal del lenguaje natural (ambiguo y necesitado de contexto) en la expresión y formalización del razonamiento humano. La lógica de predicados de primer orden ofrece generalidad, gran poder de expresión en muchas áreas, precisión, flexibilidad y, desde que Robinson propone la regla de Resolución [Rob65], un método de deducción efectivo que le dota de una capacidad computacional equivalente a la de una máquina de Turing, es decir, que es capaz de expresar e implementar todas las funciones computables. Aunque la lógica pura presenta ciertas limitaciones en algunos casos (p.e., en presencia de información imprecisa) los sistemas de programación lógicos prácticos unen a las ventajas de la lógica (cuando son usados de manera "pura") las de permitir expresar de manera precisa y eficiente la heurística y la información parcial, implementar lógicas de orden superior, y representar otros esquemas (p.e. marcos, redes semánticas, sistemas basados en reglas, bases de datos deductivas, etc...) y métodos de inferencia (p.e. inducción, aprendizaje, etc...), como es el caso de la programación lógica inductiva [Qui90, Mug90].

Por su característica de estar fundados en la lógica, los lenguajes de programación lógica (Prolog, en particular) son candidatos ideales para la manipulación semántica de programas. Es decir, presentan una semántica *declarativa* que permite probar condiciones de corrección de técnicas de manipulación tales como el análisis y la transformación de programas [PP94]. Ello permite, a su vez, definir un gran número de optimizaciones en tales programas que garanticen su mayor eficiencia, y cuya corrección se deriva del mismo hecho de estar basadas en las técnicas, formalmente correctas, anteriores. Una de estas optimizaciones es la paralelización automática. La paralelización automática de programas, y la explotación de paralelismo en general, está dando frutos especialmente alentadores en lo que respecta a los lenguajes *declarativos*. No sólo entre paralelismo y programación lógica, sino también entre paralelismo e Inteligencia Artificial existe una sinergia especial. Ello es debido tanto a la presencia de paralelismo en los sistemas inteligentes naturales, como a sus prestaciones en cuanto a velocidad/economía. Por una parte, en algunos casos se está produciendo un acercamiento de los sistemas inteligentes a la forma en que la naturaleza realiza las tareas que se están simulando, muchas de las cuales se llevan a cabo mediante procesos paralelos de algún tipo. Por otra, la mayoría de las aplicaciones de interés en IA son grandes y complejas, con costosos procesos de búsqueda e inferencia, y con mezclas de proceso simbólico, numérico y bases de datos, que con frecuencia se aproximan o superan los límites de las capacidades de los sistemas actuales. El paralelismo es una opción que puede ser muy útil a la hora de incrementar la capacidad y efectividad de los sistemas actuales en este sentido.

El paralelismo existente en un programa lógico puede ser, básicamente, de dos tipos [Con83]: *conjuntivo* y *disyuntivo*. El paralelismo disyuntivo es el resultante de la ejecución simultánea, realizada por procesadores diferentes, de cada uno de los posibles caminos en el árbol de resolución. Por tanto, equivale a la ejecución paralela de las distintas cláusulas en que se puede expandir un objetivo determinado. El paralelismo disyuntivo es característico de los problemas no determinísticos, es decir, problemas de búsqueda. Conceptualmente este tipo de paralelismo es bastante sencillo, gracias a lo cual su implementación sobre programas lógicos está prácticamente resuelta [LH85, War87b, Lus88, Kal87, Sze89, War87a, AK90, CH83, Hau90].

El paralelismo conjuntivo [DeG84, Kal87, BSY88, CDD85, Her86, Lin88, WR87, WW88, BR86, Con83, Fag87, Hua85, LK88, PK88, Kow79] se corresponde con la ejecución en paralelo de diferentes objetivos en el cuerpo de una cláusula. Por contraste con el disyuntivo, se presenta tanto en problemas determinísticos como no determinísticos, lo que amplía considerablemente su campo de acción. Sin embargo, su implementación es una tarea particularmente delicada debido a las interrelaciones entre los objetivos que se desean ejecutar en paralelo.

Para paralelizar los programas lógicos se pueden explotar los tipos de paralelismo mencionados anteriormente [Kal87]. Por otra parte, existen resultados teóricos — véase por ejemplo [CC94, HR95] — que pueden asegurar que los programas paralelizados son correctos, es decir, obtienen los mismos resultados que los correspondientes secuenciales, y que la ejecución de los primeros no tarda más tiempo que la de los segundos. Sin embargo, dichos resultados suponen un entorno ideal de ejecución paralela y no tienen en cuenta que en la práctica existen una serie de costes asociados con la ejecución paralela de tareas, como por ejemplo, creación y gestión de tareas, posible migración de tareas a procesadores remotos, costes de comunicación, etc. Por ello, si la "granularidad" de las tareas paralelas, es decir, el trabajo necesario para su ejecución completa, es muy "pequeña", puede ocurrir que dichos costes superen el beneficio obtenido en su ejecución paralela. El concepto de granularidad "pequeña" es relativo, ya que depende del sistema concreto en el cual se explote el paralelismo, y está relacionado con los costes asociados con la ejecución paralela de tareas. Por tanto, el objetivo del control de granularidad es cambiar la ejecución paralela a secuencial o viceversa en base a condiciones relacionadas con tamaños de grano de las tareas y costes de ejecución paralela.

En sistemas en donde los costes de ejecución paralela son pequeños, por ejemplo en pequeños sistemas multiprocesadores con memoria compartida, el control de granularidad no es esencial, pero puede suponer importantes mejoras. En cambio, en otros sistemas en donde dichos costes son considerables, tales como sistemas multiprocesadores con memoria distribuida o colecciones de máquinas en red, la paralelización automática de programas no se puede realizar razonablemente sin control de granularidad.

1.1. Estado del arte

El control de granularidad se ha estudiado en el contexto de la programación tradicional [KL88, MG89], programación funcional [Hue93, HLA94] y en el de la programación lógica [Kap88, DLH90, ZTD$^+$92, DL93, LGHD94, LGHD96]. La realización de un análisis preciso en tiempo de compilación es difícil, dado que la mayor parte de la información necesaria, como por ejemplo el tamaño de los datos de entrada, se conoce sólo en tiempo de ejecución. Una estrategia útil es hacer tanto trabajo como sea posible en tiempo de compilación y posponer las decisiones finales al momento de la ejecución. Esto se consigue generando funciones de coste que estiman, en tiempo de compilación, el coste de tareas a partir del tamaño de los datos de entrada. Dichas funciones se pueden evaluar posteriormente en tiempo de ejecución, cuando se conocen los tamaños de los datos de entrada, y en base a la comparación entre los costes de una ejecución paralela y secuencial, determinar el tipo de ejecución.

El citado esquema se ha propuesto para programación lógica [DLH90] y para programación funcional [RM90]. Sin embargo, el tema central del trabajo presentado en [DLH90] era realmente la estimación de cotas superiores del tiempo de ejecución de procedimientos, dejando como trabajo futuro la determinación de cómo dicha información habría de utilizarse. Uno de los desafíos planteados en esta tesis es el tapar dicho agujero y ofrecer soluciones a los muchos problemas que aparecen en la realización de control de granularidad cuando se dispone de información sobre cotas inferiores y superiores del coste de procedimientos y para un modelo general de ejecución. Dichos problemas incluyen, por una parte, la estimación del coste de tareas, del coste asociado a su ejecución paralela y el coste del propio control de granularidad. Por otra parte, existe el problema de determinar, dada la información anterior, técnicas de control de granularidad eficientes en tiempo de compilación y ejecución. En el presente trabajo proponemos soluciones para todos los problemas mencionados anteriormente.

Un enfoque alternativo consiste en determinar sólo el coste relativo de objetivos [ZTD$^+$92]. Este enfoque puede ser útil para la optimización de gestores de tareas en tiempo de ejecución que funcionen por planificación bajo demanda, pero puede no ser tan efectivo en cuanto a reducir los costes de creación de tareas. Otros esquemas de control de granularidad se basan en el uso de información obtenida, no mediante análisis en tiempo de compilación, sino mediante la obtención de "perfiles" de la ejecución real de los programas [Sar89].

En [GH85] se estudian los combinadores secuenciales ("serial combinators") con tamaño de grano razonable, pero no se discute el análisis en tiempo de compilación necesario para estimar la cantidad de trabajo que realizará una llamada. Los combinadores secuenciales expresan información de bajo nivel (p.e. evaluar funciones en paralelo, esperar a que unas variables tomen valor, etc.), mediante pseudo-funciones en un lenguaje funcional. No se pretende que los usuarios los utilicen directamente, sino que sean insertados automáticamente en los programas por paralelizadores automáticos.

El enfoque que damos en la tesis al problema del control de granularidad se basa en la solución inicialmente esbozada en [DLH90], consistente en calcular funciones de coste y realizar una transformación del programa en tiempo de compilación en base a dichas funciones, de forma que el programa transformado realiza un control de granularidad automático en tiempo de ejecución. Tomando este enfoque como punto de partida hemos propuesto un modelo general de sistema de control de granularidad capaz de utilizar información de cotas inferiores y/o superiores de coste de procedimientos, y hemos particularizado dicho modelo al caso de la programación lógica y a los tipos de paralelismo conjuntivo y disyuntivo [LGHD94, LGHD96].

En el contexto de la programación lógica, la mayor parte del trabajo realizado sobre estimación de coste en tiempo de compilación se había centrado en la obtención de cotas superiores. Sin embargo, en muchas aplicaciones es interesante trabajar con cotas inferiores [DLGHL97], como por ejemplo en la realización de control de granularidad en sistemas paralelos distribuidos. Por ello hemos desarrollado un análisis que obtiene funciones que devuelven cotas inferiores del coste de procedimientos en función de los tamaños de los datos de entrada, trabajo que hemos realizado en el contexto del proyecto ESPRIT PARFORCE (1.992 – 1.995) y publicado en [DLGHL97]. Posteriormente a nuestro trabajo se ha realizado uno parecido [KSB97], pero su objetivo no es la determinación de funciones de coste, sino de umbrales de tamaños de datos para los cuales se puede asegurar que el coste de un procedimiento será mayor que un determinado tamaño de grano. Por lo tanto, nuestro trabajo es más general y con un abanico de aplicaciones más amplio.

En el análisis de cotas inferiores, la información acerca de qué llamadas no fallarán es de vital importancia, ya que de no disponer de la misma habría que asumir que una determinada llamada podría fallar al no haber ninguna cláusula cuya cabeza unifique con ella, en cuyo caso el coste de la llamada sería nulo. Por tanto ha de desarrollarse un análisis que proporcione esta información si se quieren obtener cotas inferiores no triviales [DLGH97]. En esta tesis hemos abordado este desafío desarrollando un análisis de "no-fallo" capaz de inferir qué (llamadas a) predicados no fallarán [DLGH97]. El enfoque a este problema que damos en la tesis está inspirado en algunos trabajos realizados sobre tipos regulares [DZ92], algoritmos de resolución de ecuaciones e inecuaciones sobre el Universo de Herbrand [CL89, Kun87, LM87, LMM88, LMM91], y algoritmos de resolución de restricciones numéricas [Pug92, PW93], de forma que hemos conseguido un algoritmo capaz de tratar un amplio número de casos y cuya corrección hemos probado utilizando trabajos teóricos ya realizados. Existe un trabajo relacionado con el análisis de no-fallo, el denominado "análisis de cardinalidad" que infiere cotas inferiores y superiores del número de soluciones producidas por una llamada [BCMH94]. Sin embargo, dicho análisis parece más indicado para el análisis de determinismo que para determinar qué objetivos no fallarán.

Por otra parte, en lo que a precisión se refiere, el análisis de coste realizado en [DL93] obtiene cotas muy conservadoras para algunos programas de la clase denominada "divide y vencerás" en los que hay dependencias en los tamaños de los datos de salida de los procedimientos "divide". Dado que los programas de este tipo se usan con frecuencia en la práctica, el análisis de este tipo de programas era otro desafío [DLGH95]. Por ello hemos propuesto una mejora del tratamiento de predicados del tipo "divide y vencerás" para ambos tipos de análisis de coste (cotas superiores e inferiores) [DLGHL97].

La mayor parte del trabajo sobre control de granularidad para programas lógicos se ha centrado en proporcionar soluciones para sistemas de paralelismo conjuntivo. Era necesario por tanto extender el enfoque basado en análisis de coste y transformación de programas al caso del paralelismo disyuntivo, tarea que abordamos en [LGHD94, LGHD96].

En [AR94, AR97] se presenta un sistema paralelo distribuido que incorpora un sencillo control de granularidad para el caso del paralelismo disyuntivo, pero no se basa en análisis de costes, sino en la antigüedad de los puntos de elección creados.

Existen algunos trabajos sobre el control de granularidad y análisis de coste de programas lógicos concurrentes [GT94, GT95, Gal97] que se basan, al igual que el modelo que nosotros proponemos, en el enfoque esbozado originalmente en [DLH90], consistente en calcular funciones de coste y realizar una transformación del programa en tiempo de compilación en base a dichas funciones, de forma que el programa transformado realiza un control de granularidad automático en tiempo de ejecución. Sin embargo, estos trabajos no describen el sistema de control de granularidad de forma tan general como lo hacemos nosotros, sino que se centran exclusivamente en lenguajes lógicos concurrentes, y en sistemas multiprocesador de memoria compartida, con lo que las soluciones propuestas son bastante particulares. Nosotros proponemos un modelo general de control de granularidad que posteriormente particularizamos a programas lógicos que explotan paralelismo conjuntivo y disyuntivo, y nos centramos principalmente en este tipo de programas (no en programas concurrentes). Por tanto, aunque los dos trabajos están relacionados y tienen puntos comunes, también resuelven problemas concretos totalmente distintos, o los mismos problemas de forma diferente.

Otra diferencia es que nosotros utilizamos técnicas de transformación de programas para generar código eficiente que realiza control de granularidad en tiempo de ejecución (como por ejemplo, transformaciones para realizar el cálculo de tamaños de datos de entrada), mientras que en los otros trabajos no se realiza ninguna optimización de este tipo (se centran principalmente en diferentes análisis de programas lógicos concurrentes). Además, nuestro trabajo es el primero y único existente que describe (con la generalidad que nosotros lo hacemos) e implementa un sistema completo y totalmente automático de control de granularidad para programas lógicos. De hecho, el sistema completo de control de granularidad que proponemos lo describimos por primera vez en [LGHD94], y la primera versión de la técnica para calculo de tamaños la publicamos en [LGH93].

Otro trabajo posterior sobre control de granularidad es [SCK98], pero está basado en el uso de una métrica llamada "distancia" y su objetivo es limitar el paralelismo de un programa guardando una distancia en el tiempo entre los puntos en los que se realizan paralelizaciones, limitando de esta forma el número de tareas paralelas creadas en el sistema.

1.2. Objetivos de la tesis

La tesis pretende dar respuesta a los desafíos mencionados anteriormente. El principal objetivo de la tesis es, por tanto, el desarrollo de técnicas eficientes de control de granularidad para justificar aún más la importancia del mismo como un método adicional a considerar en la optimización de la ejecución paralela de programas lógicos y extender el campo de aplicabilidad del mismo a sistemas para los que los esquemas existentes de control de granularidad no suponen ninguna optimización. Pretendemos desarrollar completamente un sistema automático de control de granularidad para programas lógicos y evaluar dicho sistema de control de granularidad en diferentes sistemas paralelos. Perseguimos el objetivo de que las técnicas mencionadas anteriormente realicen un control de granularidad que sea más eficiente y preciso que los realizados hasta el momento y sin incurrir en demasiada carga de trabajo, ya que la mayor parte de éste se realizará en la fase de compilación, dejando para la fase de ejecución sólo lo estrictamente necesario.

Para lograr los objetivos mencionados, la tesis se centra en el desarrollo de un sistema de control de granularidad basado en el esquema de análisis y transformación de programas similar al esbozado en [DLH90]. En la realización de la transformación de programas se persigue el objetivo de minimizar el trabajo adicional hecho en tiempo de ejecución [LGH95], mientras que en el análisis, el objetivo es obtener la información necesaria para esta fase de transformación.

Es de destacar que algunas de las técnicas que hemos desarrollado para su aplicación al control de granularidad (que es el principal objetivo de esta tesis) tienen además otras aplicaciones importantes, por ejemplo, la información de no-fallo es muy útil para la eliminación de paralelismo especulativo, detección de errores de programación y transformación de programas. Por este motivo, y porque una vez que empezamos a trabajar en el análisis de no-fallo, los resultados obtenidos eran bastante prometedores, hubo un periodo en el que nos centramos exclusivamente en el análisis de no-fallo (y posteriormente en el análisis de cotas inferiores del coste de procediminetos), desviándonos del objetivo general de conseguir un sistema completo de control de granularidad. Es por ello que el lector podrán notar este mayor esfuerzo que hemos dedicado a estos análisis (especialmente al de no-fallo) en relación con otras partes del sistema completo de control de granularidad, como por ejemplo, la fase de transformación de programas o la determinación de condiciones para decidir entre ejecución paralela y secuencial.

También nos planteamos como objetivo el estudio de la eficiencia y precisión de las técnicas desarrolladas.

Finalmente, otro de los objetivos de la tesis es desarrollar técnicas que sean lo suficientemente generales para que puedan aplicarse a otros lenguajes de programación.

1.3. Principales contribuciones de la tesis

A continuación enumeramos las contribuciones principales de esta tesis. Dado que hemos realizado algunas partes del trabajo en colaboración con otros investigadores, mencionamos los mismos. Asimismo comentamos las publicaciones a las que han dado lugar los trabajos realizados (por brevedad no mencionamos los numerosos informes de proyectos ESPRIT y CICYT, ni los informes técnicos que también hemos escrito sobre nuestro trabajo):

1. Hemos propuesto un modelo general de sistema de control de granularidad capaz de utilizar información de cotas inferiores y/o superiores de coste de procedimientos, y hemos particularizado dicho modelo al caso de la programación lógica y a los tipos de paralelismo conjuntivo y disyuntivo. El modelo se basa en el esquema de análisis y transformación de programas. Este trabajo se ha realizado en colaboración con el Dr. Saumya Debray de la Universidad de Arizona y ha dado lugar a dos publicaciones. La primera de ellas en las actas del "First International Symposium on Parallel Symbolic Computation, PASCO'94" [LGHD94]. La segunda en el "Journal of Symbolic Computation" (año 1.966) [LGHD96], la cual es una versión mejorada y ampliada de la anterior.

2. Hemos implementado completamente (e integrado en un sistema real de precompilación/optimización de programas: `ciaopp` [HBPLG99, HBC^{+}99]), un sistema automático de control de granularidad para programas lógicos siguiendo el modelo comentado anteriormente.

3. Para desarrollar el sistema completo de control de granularidad propuesto hemos resuelto dos problemas fundamentales: a) Determinar costes de tareas y "overheads" y b) Controlar el paralelismo usando dicha información. Hemos descrito las soluciones a estos problemas de una forma lo suficiente

general para que los resultados obtenidos puedan aplicarse a otros sistemas (no necesariamente de programación lógica) y a diferentes modelos de ejecución.

4. En cuanto al problema de determinar costes de tareas y “overheads”:

   a) Hemos completado la estimación de cotas superiores de coste que realiza el sistema CASLOG [DL93]. Para lo cual hemos integrado CASLOG en el sistema `ciaopp` de forma que la información sobre tipos, modos de llamada, y métricas de tamaño que necesita CASLOG la suministran los analizadores de `ciaopp`, consiguiendo de esta forma que el análisis de coste sea totalmente automático.

   b) Para conseguir lo anterior hemos adaptado e integrado en `ciaopp` (en colaboración con el Dr. Francisco Bueno de la Universidad Politécnica de Madrid) una implementación del análisis de tipos regulares de Gallagher [GdW94]. Hemos añadido además la posibilidad de utilizar tipos paramétricos y de realizar simplificaciones de las definiciones de tipos dadas en forma de predicados o reglas de tipos regulares indistintamente (también es posible realizar la reescritura de tipos en ambos sentidos).

   c) Hemos desarrollado un análisis que obtiene funciones que devuelven cotas inferiores del coste de procedimientos en función de los tamaños de los datos de entrada. Hemos realizado una implementación de dicho análisis de coste y la hemos integrado en el sistema `ciaopp`. Hemos realizado experimentos que muestran que el análisis desarrollado es preciso y eficiente. Este trabajo también se ha realizado en colaboración con el Dr. Saumya Debray de la Universidad de Arizona y con el Dr. Nai-Wei Lin de la “National Chung Cheng University” en Taiwan, y ha dado lugar a dos publicaciones. La primera en el "International Static Analysis Symposium, SAS'94" [DLGHL94], que además fue una charla invitada dada por el Dr. Saumya Debray, en la que además de describir nuestro trabajo inicial sobre estimación de cotas inferiores también discutíamos la estimación de cotas superiores y diferentes problemas asociados con la estimación de costes en general. La otra publicación [DLGHL97] en las actas del "International Logic Programming Symposium (ILPS'97)" del año 1.997.

d) Hemos propuesto una mejora del tratamiento de predicados del tipo "divide y vencerás" para ambos tipos de análisis de coste (cotas superiores e inferiores), la cual se ha publicado en el artículo mencionado en el punto anterior.

e) Hemos desarrollado un análisis de "no-fallo" capaz de inferir qué (llamadas a) predicados no fallarán. Hemos implementado dicho análisis e integrado en el sistema `ciaopp`. Hemos mostrado que el análisis es bastante preciso y eficiente (y más preciso que los realizados hasta el momento). Este trabajo también se ha realizado en colaboración con el Dr. Saumya Debray y ha dado lugar a una publicación [DLGH97] en las actas de "International Conference on Logic Programming" del año 1.997.

5. Respecto al problema de controlar el paralelismo usando la información de costes y "overheads":

   a) Hemos propuesto condiciones (eficientes) para decidir entre las ejecuciones paralela y secuencial para un modelo de ejecución genérico. Dichas condiciones se basan en información sobre cotas inferiores y superiores de la granularidad de tareas. Este trabajo se ha publicado en los artículos mencionados en el punto 1 [LGHD94, LGHD96].

   b) Hemos propuesto técnicas de transformación de programas para realizar eficientemente el control de granularidad en tiempo de ejecución, como por ejemplo una técnica para el cálculo dinámico de tamaños de datos "sobre la marcha". Una versión preliminar de esta técnica se ha publicado en las actas del Segundo Congreso Nacional de Programación Declarativa (ProDe'93) [LGH93] y una versión mejorada y ampliada se ha publicado en las actas de "International Conference on Logic Programming" del año 1.995 [LGH95].

6. Hemos estudiado la eficiencia y precisión de las técnicas desarrolladas.

1.4. Estructura de la tesis

La estructura del resto del presente trabajo es la siguiente: en el capítulo 2 describimos el modelo de sistema de control de granularidad que proponemos. Comentamos los muchos problemas que surgen en la realización del control de granularidad (algunos de ellos más sutiles de lo que puedan parecer a primera vista) y proporcionamos soluciones a dichos problemas de forma general. Finalmente mostramos resultados experimentales de las técnicas de control de granularidad desarrolladas.

En el capítulo 3 describimos el análisis de cotas inferiores del coste de procedimientos desarrollado, y la mejora del tratamiento de predicados del tipo "divide y vencerás" que comentamos en la sección anterior. También comentamos la implementación que hemos hecho de dicho análisis y mostramos resultados experimentales.

En el capítulo 4 describimos el análisis de no-fallo desarrollado (necesario para el análisis de cotas inferiores mencionado en el punto anterior), comentamos su implementación y también mostramos resultados experimentales.

En el capítulo 5 describimos detalladamente una de las técnicas de transformación de programas para realizar eficientemente el control de granularidad: la técnica para transformar programas de forma que calculen los tamaños de algunos datos "sobre la marcha" (durante la ejecución del propio programa y sin modificación de su flujo de control). También mostramos resultados experimentales de la ganancia obtenida mediante esta optimización.

Finalmente, en el capítulo 6 comentamos las principales conclusiones que sacamos del trabajo realizado y algunas líneas a seguir en trabajos futuros.

Capítulo 2

Metodología para el control de granularidad

Como se menciona en la introducción, es posible explotar diferentes tipos de paralelismo (conjuntivo y disyuntivo) en los programas lógicos, de forma que se preserve la corrección (es decir, los programas paralelizados obtienen los mismos resultados que los correspondientes secuenciales), y la eficiencia (la ejecución de los primeros no tarda más tiempo que la de los segundos). Sin embargo, dichos resultados suponen un entorno ideal de ejecución paralela y no tienen en cuenta que en la práctica existen una serie de costes asociados con la ejecución paralela de tareas, como por ejemplo, creación y gestión de tareas, posible migración de tareas a procesadores remotos, costes de comunicación asociados, etc. Dichos costes pueden dar lugar a que la ejecución de los programas paralelos sea más lenta que los secuenciales, o al menos, limitar la ganancia debida al paralelismo introducido. En este capítulo describimos una metodología que estima eficientemente la granularidad de las tareas y la usa para limitar el paralelismo, de forma que se controla el efecto de los costes mencionados anteriormente. El coste en tiempo de ejecución asociado con nuestro enfoque es usualmente bastante pequeño, ya que se realiza tanto trabajo como sea posible en tiempo de compilación. Además realizamos un análisis estático del coste en tiempo de ejecución asociado con el proceso de control de granularidad, de forma que se pueda decidir su conveniencia. Finalmente mostramos que las ganancias en tiempo de ejecución obtenidas mediante la incorporación de control de granularidad son bastante buenas, especialmente para los sistemas con costes asociados a la ejecución paralela que varian desde medios a grandes.

Hasta la fecha, no tenemos conocimiento de otros trabajos que describan un sistema completo de control de granularidad para programas lógicos, discuta los muchos problemas que surgen (algunos de ellos más sutiles de lo que puedan parecer a primera vista) y que proporcione soluciones a dichos problemas con la generalidad con la que nosotros presentamos nuestro trabajo.

Por brevedad, y porque estamos más interesados en tiempos de ejecución, cuando comparamos la ejecución paralela con la secuencial, no discutimos en detalle los diferentes tipos de costes que aparecen en la ejecución paralela, los cuales pueden incluir no sólo costes relacionados con tiempos, sino también, otros costes, como por ejemplo, los asociados con el consumo de memoria. Sin embargo, creemos que se puede aplicar un tratamiento similar al que proponemos al análisis y control de los costes asociados con el manejo de la memoria.

2.1. Un modelo general

En esta sección comenzamos discutiendo los problemas básicos que necesitan abordarse en el enfoque que damos al control de granularidad, en términos de un modelo de ejecución genérico. En las siguientes secciones particularizaremos al caso de los programas lógicos.

2.1.1. Derivación de condiciones suficientes

En primer lugar discutimos como se pueden derivar condiciones para decidir entre las ejecuciones paralela y secuencial. Consideramos un modelo de ejecución genérico: sea $t = t_1, \ldots, t_n$ una tarea que se puede descomponer en las subtareas $t_1, \ldots, t_n$, las cuales son candidatas para su ejecución paralela, Ts representa el coste (tiempo de ejecución) de la ejecución secuencial de t, y T_i representa el coste de la ejecución de la subtarea t_i.

Puede haber varias formas de ejecutar t en paralelo, dependiendo de diferentes alternativas de planificación de tareas ("scheduling"), balance de carga, etc., cada una de ellas con su propio coste (tiempo de ejecución). Nuestro objetivo es derivar condiciones para decidir entre las ejecuciones paralela y secuencial, que sean independientes de los algoritmos de planificación de tareas, y poder expresarlas en términos de un modelo de ejecución genérico. Para simplificar la discusión, asumiremos que Tp representa de alguna forma los costes de todas las posibles ejecuciones en paralelo de t (los cuales dependerán de las diferentes políticas de planificación de tareas). Más concretamente, $Tp \leq Ts$ debería de entenderse como "Ts es más grande o igual que cualquier valor posible para Tp".

En una primera aproximación, asumimos que los puntos de paralelización de t son fijos. También asumimos, por simplicidad, y sin pérdida de generalidad, que ningún test — tales como, quizás, "tests de independencia" [CC94, HR95] — salvo los relacionados con el control de granularidad son necesarios.

Por tanto, el propósito del control de granularidad será determinar, en base a algunas condiciones, si las t_i's han de ejecutarse en paralelo o secuencialmente. Al hacer esto, el objetivo es mejorar la relación entre los tiempos de las ejecuciones paralela y secuencial. Es interesante asegurar que $Tp \leq Ts$. En general, esta condición no se puede determinar antes de ejecutar t, mientras que el control de granularidad, intuitivamente, debería de realizarse antes. Por tanto, nos vemos forzados a usar aproximaciones. En este punto, una alternativa clara es abandonar el objetivo de asegurar estrictamente que $Tp \leq Ts$ y usar alguna heurística que se comporte bien en término medio. Por otra parte, no es fácil encontrar dicha heurística y, además, obviamente, el asegurar que la ejecución paralela no tardará más tiempo que la secuencial, tiene su importancia en la práctica. Esto nos sugiere una solución alternativa: evaluar una condición más simple, para la cual podamos probar que se asegure que $Tp \leq Ts$. Dicha condición puede basarse en el cálculo de una cota superior para Tp y una cota inferior para Ts. El asegurar que $Tp \leq Ts$ corresponde al caso en el cual la acción realizada cuando la condición no se cumple es ejecutar secuencialmente, e.d. corresponde a una filosofía en la que las tareas se ejecutan secuencialmente a menos que se demuestre que la ejecución paralela es más rápida. Esto último es útil cuando se "paraleliza un programa secuencial". Este enfoque se discute en la sección siguiente. El caso opuesto de "secuencializar un programa paralelo", en donde el objetivo es detectar cuándo se cumple la condición opuesta $Ts \leq Tp$, se considera en la sección 2.1.1.

Paralelización de un programa secuencial

Para derivar una condición suficiente de forma que se cumpla la desigualdad $Tp \leq Ts$, calculamos cotas superiores para su parte izquierda y cotas inferiores para su parte derecha, e.d. una condición suficiente para $Tp \leq Ts$ es $Tp^u \leq Ts^l$, en donde Tp^u denota una cota superior de Tp y Ts^l una cota inferior de Ts. A lo largo de la discusión usaremos los superíndices l y u para denotar cotas inferiores y superiores respectivamente.

Sea $t = t_1, \ldots, t_n$ una tarea que se puede descomponer en las subtareas $t_1, \ldots, t_n$, las cuales son candidatas para su ejecución paralela. Supongamos que hay p procesadores libres en el sistema en el instante en el que la tarea t va a ejecutarse. Supongamos además que $p \geq 2$ (si sólo hay un procesador, entonces la ejecución se realiza secuencialmente) y que m es el menor entero que es más grande o igual que n/p, e.d. $m = \lceil \frac{n}{p} \rceil$. Suponiendo que cuando un procesador se queda libre y hay una tarea esperando a ser ejecutada, dicho procesador la "coge", podemos decir que cualquier procesador ejecutará como máximo m tareas. Tenemos que $Tp^u = Spaw^u + C^u$, en donde $Spaw^u$ es una cota superior del coste de la creación de n subtareas paralelas, y C^u es una cota superior del tiempo de la propia ejecución de t. $Spaw^u$ dependerá del sistema particular en el cual la tarea t vaya a ejecutarse. Puede que sea una constante, o una función de varios parámetros, tales como tamaños de los datos de entrada, número de argumentos de entrada, número de tareas, etc. y se puede determinar experimentalmente. Consideraremos ahora cómo puede calcularse C^u. Sea C_i^u una cota superior del coste de la subtarea t_i, y supongamos que los costes $C_1^u, \ldots, C_n^u$ están ordenados en orden no creciente. Una forma posible de calcular C^u sería la siguiente:

$$C^u = \sum_{i=1}^{m} C_i^u$$

en donde $m = \lceil \frac{n}{p} \rceil$ (obviamente también se podría utilizar la expresión $C^u = m\ C_1^u$, la cual es menos precisa, pero en algunos casos podría ser más fácil de calcular).

Cada C_i^u puede considerarse como la suma de dos componentes:

$$C_i^u = Sched_i^u + T_i^u$$

en donde $Sched_i^u$ denota el tiempo transcurrido desde que el procesador que ejecuta la tarea t_i se queda libre hasta que empieza a ejecutar la tarea t_i propiamente dicha, e.d. el coste de la preparación, planificación, coste de comunicación, etc.[1] T_i^u denota el tiempo empleado por la ejecución de t_i sin tener en cuenta los

[1] Nótese que en algunos sistemas paralelos, como por ejemplo &-Prolog [HG91], $Sched_i^u$ puede que sea cero en algunos casos, ya que no existe ningún coste asociado con la preparación de una tarea paralela si ésta se ejecuta por el mismo procesador que la creó.

costes asociados a la ejecución paralela mencionados anteriormente. Suponemos que está garantizado que las tareas $t_1, \ldots, t_n$ no fallarán. También asumimos que Ts^l se puede calcular de la forma siguiente: $Ts^l = Ts^l_1 + \cdots + Ts^l_n$, en donde Ts^l_i es una cota inferior del coste de la ejecución (secuencial) de la subtarea t_i.

El siguiente lema resume la discusión anterior:

Lema 2.1.1 *Si* $Spaw^u + \sum_{i=1}^{m} C_i^u \leq Ts^l_1 + \cdots + Ts^l_n$, *entonces* $Tp \leq Ts$.

Demostración Trivial. ∎

Dado que a menudo, en tiempo de ejecución, han de calcularse (total o parcialmente) cotas de los costes de ejecución, lo mismo habrá que hacer con las condiciones anteriores. Sería deseable que esta evaluación se realizase de la forma más eficiente posible. Claramente, existe un compromiso entre el coste de evaluación de dicha condición suficiente y su precisión. Una condición suficiente que conlleva una evaluación más simple que la del lema 2.1.1 es la que damos a continuación, la cual se basa en una serie de asunciones razonables.

Supongamos que puede asegurarse que las tareas $t_1, \ldots, t_n$ no tardarán más tiempo que en su ejecución secuencial, sin tener en cuenta el tiempo necesario para lanzarlas en paralelo, ni todos los costes asociados con la ejecución paralela[2] y que los costes $Sched^u_1, \ldots, Sched^u_n$ están ordenados en orden no creciente. Sea $Thres^u$ un umbral calculado mediante la siguiente expresión: $Thres^u = Spaw^u + \sum_{i=1}^{m} Sched^u_i$. Supongamos además que hay p procesadores libres en el sistema en el instante en el que las tareas $t_1, \ldots, t_n$ van a ejecutarse. Supongamos además que $p \geq 2$ (si sólo hay un procesador, entonces la ejecución se realiza secuencialmente) y que $m = \lceil \frac{n}{p} \rceil$.

[2]Esto puede asegurarse para algunas plataformas de ejecución, por ejemplo si las tareas son “independientes”. Sin embargo, en algunos casos, si las tareas son “dependientes”, pueden tardar más de lo que lo harían en la ejecución secuencial.

Teorema 2.1.2 *Si existe un subconjunto I con $m+1$ elementos del conjunto de índices $\{1, 2, \ldots, n\}$, es decir $I \subseteq \{1, 2, \ldots, n\}$, tal que para todo $i \in I$ se cumple que $Thres^u \leq Ts_i^l$ (en donde Ts_i^l denota una cota inferior del coste de la ejecución secuencial de la tarea t_i) entonces $Tp \leq Ts$.*

Demostración Sin pérdida de generalidad, supongamos que los costes $Ts_1, \ldots, Ts_n$ están ordenados en orden no creciente (en donde Ts_i denota el coste de la ejecución secuencial de la tarea t_i). Consideremos la siguiente condición:

$$T_p^u \leq Ts_1 + \cdots + Ts_m + Ts_{m+1} + \cdots + Ts_n \tag{2.1}$$

en donde $T_p^u = Thres^u + Ts_1 + \cdots + Ts_m$, y $Ts = Ts_1 + \cdots + Ts_m + Ts_{m+1} + \cdots + Ts_n$. Si esta condición se cumple entonces $Tp \leq Ts$, puesto que su parte izquierda es una cota superior de Tp. Simplificando la condición 2.1 obtenemos:

$$Thres^u \leq Ts_{m+1} + \cdots + Ts_n \tag{2.2}$$

Si el subconjunto I tiene $m+1$ elementos (índices), necesariamente existirá al menos un índice $j \in I$ tal que $m+1 \leq j \leq n$. Si $Thres^u \leq Ts_j^l$ entonces $Thres^u \leq Ts_j$, y por tanto se cumple la condición 2.2. ∎

Tratamos ahora un caso ligeramente más complejo en el cual, además, consideramos otros costes, incluyendo el coste del propio control de granularidad: supongamos ahora que la ejecución de t_i tarda T_i unidades de tiempo, tal que $T_i = T_{s_i} + W_i$, en donde W_i es el coste de algún trabajo "extra" debido a la propia ejecución paralela (por ejemplo el coste de acceder a referencias remotas) o del control de granularidad, o de ambos. Sean l ($1 \leq l \leq n$) las tareas para las cuales sabemos que $W_i \neq 0$ (o equivalentemente, $T_i > T_{s_i}$). Supongamos que los costes $W_1^u, \ldots, W_l^u$ están ordenados en orden no creciente, y sea $r = min(l, m)$. Entonces, podemos calcular un nuevo umbral, $Thres_w^u$, sumando W ($Thres_w^u = Thres^u + W$) al umbral anterior ($Thres^u$). W puede calcularse de la forma siguiente: $W = \sum_{i=1}^{r} W_i^u$.

Teorema 2.1.3 *Si existe un subconjunto I con $m+1$ elementos del conjunto de índices $\{1, 2, \ldots, n\}$, es decir $I \subseteq \{1, 2, \ldots, n\}$, tal que para todo $i \in I$ se cumple que $Thres^u_w \leq Ts^l_i$ (en donde Ts^l_i denota una cota inferior del coste de la ejecución secuencial de la tarea t_i) entonces $Tp \leq Ts$.*

Demostración La demostración es similar a la del teorema 2.1.2. Dado que $Thres^u + W + Ts_1 + \cdots + Ts_m$, es además una cota superior de Tp, podemos argumentar lo mismo en esta demostración reemplazando la condición 2.1 por $Thres^u + W + Ts_1 + \cdots + Ts_m \leq Ts_1 + \cdots + Ts_m + Ts_{m+1} + \cdots + Ts_n$ ∎

Supongamos ahora que no podemos asegurar que para todo i, $1 \leq i \leq n$, t_i no fallará. Supongamos que t_k es la tarea más a la izquierda para la cual puede asegurarse que no fallará, para algún $1 \leq k \leq n$. Podemos modificar ligeramente el lema 2.1.1 y el teorema 2.1.2 de la forma que describimos a continuación.

El lema 2.1.1 se puede reescribir de la forma siguiente:

Lema 2.1.4 *Si $Spaw^u + \sum_{i=1}^{m} C^u_i \leq Ts^l_1 + \cdots + Ts^l_k$, entonces $Tp \leq Ts$.*

Demostración Trivial. ∎

La única diferencia es que consideramos $Ts^l_1 + \cdots + Ts^l_k$ en la parte derecha de la respectiva inecuación en lugar de $Ts^l_1 + \cdots + Ts^l_n$.

Es posible reescribir los teoremas 2.1.2 y 2.1.3 suponiendo que las tareas que tienen los m costes más grandes están entre $t_1, \ldots, t_k$ (es decir, entre las tareas que con seguridad sabemos que no fallarán). Las demostraciones son similares.

Teorema 2.1.5 *Si existe un subconjunto I con $m+1$ elementos del conjunto de índices $\{1, 2, \ldots, n\}$, es decir $I \subseteq \{1, 2, \ldots, n\}$, tal que para todo $i \in I$ se cumple que $Thres^u \leq Ts^l_i$ (en donde Ts^l_i denota una cota inferior del coste de la ejecución secuencial de la tarea t_i) e $i \leq k$ (es decir las tareas que tienen los m costes más grandes están entre $t_1, \ldots, t_k$), entonces $Tp \leq Ts$.*

Este teorema también se puede formular imponiendo la condición $Thres_w^u \leq Ts_i^l$ en lugar de $Thres^u \leq Ts_i^l$ (la demostración es similar).

Secuencialización de un programa paralelo

Supongamos ahora que queremos detectar cuándo se cumple que $Ts \leq Tp$, porque tenemos un programa paralelo y queremos sacar provecho de realizarle algunas secuencializaciones. En este caso podemos calcular Ts^u y Tp^l. Sea T_i^l una cota inferior del tiempo de ejecución de t_i. Tp^l puede determinarse de varias formas:

1. Si $n \leq p$ entonces: $Tp^l = Spaw^l + \max(T_1^l, \ldots, T_n^l)$. Nótese que en el mejor caso cada tarea se ejecutará en un procesador diferente, pero en la práctica podría haber casos en los que esto no se consiguiera debido a una mala planificación de tareas.

2. Si $n > p$ entonces: $Tp^l = Spaw^l + \lceil \frac{n}{p} \rceil \min(T_1^l, \ldots, T_n^l)$. En este caso elegimos el mínimo porque sabemos que en el mejor caso habrá al menos un procesador que ejecutará al menos $\lceil \frac{n}{p} \rceil$ tareas.

3. $Tp^l = Spaw^l + \sum_{i=1}^{m} T_i^l$, en donde $m = \lceil \frac{n}{p} \rceil$ y los costes $T_1^l, \ldots, T_n^l$ están ordenados en orden no decreciente.

4. $Tp^l = Spaw^l + \frac{Ts_1^l + \cdots + Ts_n^l}{p}$

La determinación de T_i^l dependerá, por supuesto, de la forma en que se vaya a ejecutar t_i. Si la ejecución se va a realizar en paralelo sin control de granularidad, con control de granularidad, o secuencialmente, calculamos Tp_i^l, T_i^l, o Ts_i^l respectivamente. La determinación de Tp_i^l y T_i^l se discute en la sección 2.7.

Por supuesto, buscamos las mejores cotas, es decir, Tp^l lo más grande posible y Tp^u lo más pequeño posible. Por ello, podemos elegir el máximo de las diferentes posibilidades para calcular Tp^l. En general, si existen n alternativas diferentes $x_1, \ldots, x_n$ para calcular Tp^l (Tp^u, respectivamente) elegiremos $Tp^l = \text{máx}(x_1, \ldots, x_n)$ ($Tp^u = \text{mín}(x_1, \ldots, x_n)$, respectivamente).

2.1.2. Comparación del control de granularidad en tiempo de ejecución con el realizado en tiempo de compilación

La evaluación de las condiciones suficientes propuestas en las secciones anteriores, en principio pueden realizarse totalmente en tiempo de ejecución, totalmente en tiempo de compilación, o parcialmente en cada uno de ellos. Por ejemplo, puede que sea posible determinar en tiempo de compilación si la condición expresada en el teorema 2.1.2 será siempre cierta cuando se evalúe en tiempo de ejecución. Sea C^l una cota inferior del coste de cada t_i, $1 \leq i \leq n$, entonces si $Thres^u \leq (n-m)C^l$, la condición del teorema se cumple, ya que $(n-m)C^l$ es una cota inferior de $T_{s_{m+1}} + \cdots + T_{s_n}$. Claramente, en este caso no es necesario realizar ningún control de granularidad y las tareas se pueden ejecutar siempre en paralelo. El caso opuesto, en donde se puede determinar estáticamente que es mejor ejecutarlas secuencialmente, también es posible. Por tanto, desde el punto de vista del control de granularidad las partes de un programa se pueden clasificar como *paralelas* (todas las paralelizaciones realizadas son incondicionales), *secuenciales* (no hay tareas paralelas), y *las que realizan control de granularidad* (se realizan tests en tiempo de ejecución basados en información de granularidad para decidir entre ejecución paralela o secuencial). Ya sea en tiempo de compilación o en tiempo de ejecución, hemos de abordar dos problemas básicos para realizar control de granularidad: cómo se calculan las cotas de los tiempos de ejecución y de los costes asociados a la ejecución paralela, los cuales son los parámetros de las condiciones suficientes (*análisis de coste y de "overhead"*) y cómo se utilizan condiciones suficientes para controlar paralelismo (*control de granularidad*). Éstos son los temas tratados en las siguientes secciones. En nuestro enfoque, ambos temas implican en general técnicas en tiempo de compilación y ejecución.

Análisis del coste de las tareas

Dado que el *coste de una tarea* no es computable en general en tiempo de compilación, no vemos forzados a recurrir a aproximaciones y, posiblemente, a realizar algún trabajo en tiempo de ejecución. De hecho, tal y como se dice en [DLH90], dado que el trabajo realizado por una llamada a un procedimiento recursivo a menudo depende del tamaño de sus datos de entrada, dicho trabajo no puede estimarse en general en tiempo de compilación de ninguna forma razonable y para dichas llamadas es necesario realizar algún trabajo en tiempo de ejecución. Nuestro enfoque básico es el siguiente: dada una llamada p, se calcula una expresión $\Phi_p(n)$ tal que:

- es relativamente fácil de evaluar, y
- aproxima $\mathtt{Cost}_p(n)$, en donde $\mathtt{Cost}_p(n)$ denota el coste de la ejecución de p para unos datos de entrada de tamaño n.

La idea es que $\Phi_p(n)$ se determina en tiempo de compilación, y luego se evalúa en tiempo de ejecución, cuando se conozcan los tamaños de los datos de entrada, obteniéndose de esta forma una estimación del coste de la llamada. Es de destacar que el compilador simplificará la evaluación de $\Phi_p(n)$ lo máximo posible. En muchos casos, será posible simplificar la función de coste (o, siendo más precisos, el test que se realizará) hasta el punto de que sea posible determinar estáticamente un umbral para el tamaño de uno de los argumentos de entrada. En este caso, en tiempo de ejecución, se compara dicho tamaño de entrada con el umbral ya calculado, y por tanto no se necesita evaluar la función. Esta simplificación se discute en la sección 2.5.1. Si después de dicha simplificación, la expresión es costosa desde el punto de vista de su evaluación, el compilador puede decidir calcular una aproximación segura que tenga un menor coste de evaluación. También es de destacar que además se tiene en cuenta el coste de la evaluación de tests, y, en general, el de realizar el control de granularidad, tal y como se describe en la sección 2.6. En lo que sigue nos referiremos a las expresiones $\Phi_p(n)$ calculadas en tiempo de compilación como *funciones de coste*.

Tal y como mencionamos en la sección 2.1 la aproximación de la condición utilizada para decidir entre paralelización y secuencialización puede basarse, bien en alguna heurística, o bien en una aproximación *segura* (e.d. una cota superior o inferior). Para el último enfoque ya mostramos condiciones suficientes para realizar una ejecución paralela preservando la eficiencia. Debido a estos resultados, en general requeriremos que $\Phi_p(n)$ sea no sólo una simple aproximación, sino que además sea una cota del coste real de ejecución. Afortunadamente, y como ya se ha mencionado, se ha realizado mucho trabajo sobre el análisis de la complejidad (respecto a tiempo de ejecución) de los programas [Met88, Wad88, Ros89, BH89, Sar89, ZZ89, FSZ91]. Los métodos más directamente aplicables son los presentados en [DL93] y [DLGHL97], los cuales estiman estáticamente las funciones de coste para los predicados en un programa lógico. Los dos enfoques tienen mucho en común pero difieren en el modo en el que se realiza la aproximación. En el primero de ellos se calculan cotas superiores de los costes de las tareas, es decir $(\forall n)\mathtt{Cost}_p(n) \leq \Phi_p(n)$, mientras que en el segundo, el cual se describe en el capítulo 3, se realiza la aproximación opuesta: $(\forall n)\mathtt{Cost}_p(n) \geq \Phi_p(n)$.

Ejemplo 2.1.1 Considérese el procedimiento `q/2` definido de la forma siguiente:

```
q([],[]).
q([H|T],[X|Y]):- X is H + 1, q(T,Y).
```

en donde el primer argumento es un argumento de entrada. Supongamos que la unidad de coste utilizada es el número de pasos de resolución. En una primera aproximación, y por simplicidad, suponemos que el coste de un paso de resolución (e.d., una llamada a un procedimiento) es el mismo que el del predicado predefinido `is/2`. Con estas suposiciones, la función de coste de `q/2` es $\texttt{Cost}_q(n) = 2\ n + 1$, en donde n es el tamaño (longitud) de la lista de entrada (primer argumento). □

Análisis de costes asociados a la paralelización (overheads)

En cuanto a la determinación de los costes (overheads) que aparecen con los otros costes en las condiciones suficientes de la sección 2.1.1, tal y como allí mencionábamos, ésta es una tarea más o menos trivial en sistemas en donde tales costes pueden considerase constantes. Sin embargo, es frecuente que dichos costes tengan, además de una componente constante, otros componentes, los cuales pueden ser una función de varios parámetros, tales como tamaño de los datos de entrada, número de argumentos de entrada, número de tareas, número de procesadores activos en el sistema, tipo de procesador, etc., en cuyo caso se necesita alguna evaluación en tiempo de ejecución. Por ejemplo, en un sistema distribuido, el coste de lanzamiento de una tarea es a menudo proporcional al tamaño de los datos, ya que en algunos modelos se envía un cierre completo (una llamada y sus argumentos) al procesador remoto. Por tanto, la evaluación de los "overheads" también implica en general la generación en tiempo de compilación de una función de coste, la cual se evaluará en tiempo de ejecución cuando se conocen los parámetros (tales como tamaño de datos del ejemplo anterior).

Realización del control de granularidad

Suponiendo que existen técnicas, como las descritas anteriormente en términos generales, para determinar costes de las tareas y "overheads", el resto del trabajo del control de granularidad consiste en determinar un modo para calcular dichos costes y controlar así la creación de tareas utilizando dicha información.

De nuevo, adoptamos el enfoque de hacer en tiempo de compilación tanto trabajo como sea posible, y proponemos realizar una transformación del programa de forma que los cálculos de las funciones de coste y las decisiones sobre qué objetivos se ejecutan en paralelo estén codificadas en el propio programa de la forma más eficiente posible.

La idea es posponer los cálculos y decisiones reales a tiempo de ejecución cuando se conocen los parámetros que no se conocían en tiempo de compilación, tales como tamaño de datos, o carga de los procesadores. En particular, los programas transformados realizarán las tareas siguientes: calcular los tamaños de los datos de entrada; usar dichos tamaños para evaluar las funciones de coste; estimar los overheads debidos al lanzamiento en paralelo de tareas y al scheduling; decidir si las tareas se ejecutan en paralelo o secuencialmente; decidir si se debe de continuar o no haciendo control de granularidad, etc.

2.2. Análisis de coste de programas lógicos

Ahora discutiremos el problema del análisis de coste en el contexto de los programas lógicos. Distinguiremos entre los casos de paralelismo conjuntivo y disyuntivo.

2.2.1. Análisis de coste para el paralelismo conjuntivo

En el paralelismo conjuntivo a nivel de objetivo las unidades que se paralelizan son los objetivos. Hemos desarrollado un análisis de cotas inferiores de coste (el cual además incluye un análisis de no fallo). Dicho análisis se describe detalladamente en el capítulo 3 (y además está publicado en el artículo [DLGHL97]), por lo que referimos al lector a dicho capítulo.

En el análisis de cotas inferiores de coste, la información acerca de qué llamadas no fallarán es de vital importancia, ya que de no disponer de la misma habría que asumir que una determinada llamada puede fallar al realizarse la unificación con la cabeza de una cláusula, en cuyo caso el coste de la llamada es nulo. Por ello nos vimos obligados a desarrollar un análisis que proporcionase dicha información de no-fallo, el cual se describe en detalle en el capítulo 4.

Un análisis de cotas superiores (no triviales) de objetivos se describe en [DL93]. Es muy similar y más "simple" que el de cotas inferiores. Una de las causas de esta simplicidad es que no utilizamos ningún análisis de no-fallo (como se realiza en el cálculo de cotas inferiores), y asumimos que cada literal del cuerpo de una cláusula no fallará. Como en la páctica es más frecuente que un literal no falle, las cotas superiores calculadas bajo esta suposición, en general, son no triviales y se aproximan razonablemente al coste real. Además, también suponemos que se ejecutan todas las cláusulas de un predicado (independientemente de si se requieren o no todas las soluciones). Esto hace que en el caso de que se requieran todas las soluciones, las cotas superiores calculadas sea no triviales. En contrapartida, cuando sólo se requiere una solución, en algunos casos esta suposición puede dar lugar a cotas superiores muy "grandes".

2.2.2. Análisis de coste para el paralelismo disyuntivo

El caso del paralelismo disyuntivo es similar al del paralelismo conjuntivo salvo que aquí las unidades que se paralelizan son ramas de computación en lugar de objetivos. Sin embargo, los análisis de coste de las secciones previas pueden adaptarse teniendo en cuenta simplemente la "continuación" de los puntos de elección en consideración. A modo de ejemplo, consideremos una cláusula $h :- \ \ldots, L, L_1, \ldots, L_n.$. Supongamos que el predicado del literal L es p, y que la definición del predicado p tiene "c" cláusulas "elegibles" $\{Cl_1, \ldots, Cl_c\}$, en donde $Cl_i = h_i :- \ b_i$. En la ejecución paralela disyuntiva del literal L, las "c" alternativas (cada una de ellas corresponde a una cláusula del predicado p) y sus continuaciones (el resto de los L_i, $1 \leq i \leq n$, y los otros objetivos desde L_{n+1} hasta L_k que aparecen después de ellos en el resolvente en el momento en el que L es el literal más a la izquierda) se ejecutan en paralelo. Denotemos como $Cost_{cl_i}(x)$ y $Cost_{L_i}(x)$ el coste de la cláusula Cl_i y del literal L_i respectivamente, entonces el coste de la alternativa correspondiente a la cláusula Cl_i, denotada por $Cost_{ch_i}$ se puede calcular de la forma siguiente: en el caso de que estemos calculando cotas inferiores tenemos que $Cost^l_{ch_i}(x) = Cost^l_{cl_i}(x) + \sum_{j=1}^{m} Cost^l_{L_j}(x)$, si está garantizado que la cláusula Cl_i no fallará y m es el literal más a la izquierda para el que podemos garantizar que no fallará; o, alternativamente, $Cost^l_{ch_i}(x) = Cost^l_{cl_i}(x)$, si no se puede garantizar que la cláusula Cl_i no fallará. Por otra parte, en el cálculo de cotas superiores tenemos que $Cost^u_{ch_i}(x) = Cost^u_{cl_i}(x) + \sum_{j=1}^{k} Cost^u_{L_j}(x)$.

La determinación de los literales desde L_{n+1} hasta L_k, las continuaciones de las cláusulas consideradas, no se pueden obtener directamente a partir del grafo de llamadas en el caso de que existan predicados recursivos que incorporen la optimización de última llamada (o recursividad de cola). El problema es que mientras que las llamadas que no son recursivas por la cola que aparecen en el cuerpo de un procedimiento vuelven al "llamador", debido a la optimización de última llamada, una llamada recursiva por la cola puede que no vuelva a su "llamador", sino al procedimiento antecesor más próximo que hizo una llamada no recursiva por la cola. Por tanto, mientras que para llamadas no recursivas por la cola, la transferencia del control del llamador al llamado y que luego es devuelto en sentido contrario, es evidente a partir del grafo de llamadas del programa, no ocurre lo mismo para el caso de llamadas recursivas por la cola. Para abordar este problema, dado un programa, construimos una gramática libre de contexto de la forma siguiente: para cada cláusula del programa

$$p(\bar{t}) :- \ Guard \mid q_1(\bar{t}_1), \ldots, q_n(\bar{t}_n)$$

la gramática contiene una producción

$$p \longrightarrow q_1 \ \mathtt{L}_1 \ q_2 \ \mathtt{L}_2 \ \ldots \ \mathtt{L}_{n-1} \ q_n$$

en donde los $\mathtt{L}_i$, que son etiquetas correspondientes a las continuaciones de procedimientos, son los símbolos terminales de la gramática. A continuación calculamos los conjuntos de CONTINUACIÓN (FOLLOW sets) para esta gramática [ASU86]: para cualquier predicado p, FOLLOW(p) da el conjunto de continuaciones posibles para p.

2.3. Control de granularidad en programación lógica para el paralelismo conjuntivo

En esta sección utilizaremos un ejemplo para explicar intuitivamente en qué consiste la transformación básica de un programa que se realiza en nuestro enfoque, ya que una presentación formal haría, innecesariamente, la exposición más compleja.[3]

[3]Aunque el presentar la técnica propuesta en términos de una transformación fuente-fuente es conveniente por claridad y porque es una técnica de implementación viable, obviamente,

Ejemplo 2.3.1 Considérese el predicado q/2 definido en el ejemplo 2.1.1, y el predicado r/2 definido de la forma siguiente:

```
r([], []).
r([X|RX], [X2|RX1]) :- X1 is X * 2, X2 is X1 + 7, r(RX, RX1).
```

y el objetivo paralelo: `..., q(X, Y) & r(X, Z), ...`, en el que los literales `q(X, Y)` y `r(X, Z)` se ejecutan en paralelo, como indica la conectiva & (conjunción paralela) [HG91].

Suponiendo que el primer argumento de r/2 es un argumento de entrada, las funciones de coste de q/2 y r/2 son $\mathtt{Cost}_q(n) = 2\ n + 1$ y $\mathtt{Cost}_r(n) = 3\ n + 1$ respectivamente. Supongamos que hay un número de procesadores $p \geq 2$. De acuerdo con el teorema 2.1.2, el objetivo anterior puede transformarse de forma segura en:

```
..., length(X, LX),
     Cost_q is LX * 2 + 1,
     Cost_r is LX * 3 + 1,
     (Cost_q > 15, Cost_r > 15 ->
             q(X, Y) & r(X, Z)
             ;
             q(X, Y), r(X, Z)), ...
```

en donde se supone un valor para el umbral ($Thres^u$) de 15 unidades de cómputo, las variables `Cost_q` y `Cost_r` denotan el coste de la ejecución (secuencial) de los objetivos `q(X, Y)` y `r(X, Z)` respectivamente, y `LX` denota la longitud de la lista `X`. □

las transformaciones pueden realizarse también a un nivel más bajo, para reducir aún más los "overheads" en tiempo de ejecución.

2.4. Control de granularidad en programación lógica para el paralelismo disyuntivo

Consideremos el cuerpo $\ldots, L, L_1, \ldots, L_n$. de una cláusula del ejemplo de la sección 2.2.2. Dicho cuerpo se puede transformar para realizar control de granularidad de la forma siguiente: $\ldots, (cond \rightarrow L' \ ; \ L), L_1, \ldots, L_n$. En donde L' es la versión paralela de L, y se crea reemplazando el nombre del predicado de L (p) por otro, por ejemplo p', tal que p' es la versión paralela de p, y se obtiene a partir de p reemplazando el nombre del predicado p por p' en todas las cláusulas de p. p' se declara entonces como "paralelo" mediante la directiva apropiada. Si *cond* se cumple, entonces el literal L' (versión paralela de L) se ejecuta, en otro caso se ejecuta L.

Un problema que presenta el uso de una directiva para expresar paralelismo a nivel de predicado es que o bien todas o ninguna de las cláusulas se ejecutan en paralelo. Dado que puede haber diferencias de costes entre diferentes cláusulas, esto puede acarrear un balance de carga peor, por tanto, una opción mejor puede ser el uso de una declarativa que nos permita especificar grupos (clusters) de cláusulas tales que las cláusulas de un grupo se ejecutan secuencialmente, y grupos diferentes se ejecutan en paralelo. De esta forma, se pueden tener varias versiones paralelas de un predicado, y cada una de ellas se ejecuta si se cumple una condición determinada. El siguiente ejemplo ilustra esta propuesta, en donde una llamada a `p` en `...,p, q, r.` y el predicado `p` se transforman de la forma siguiente:

```
..., (cond_1 -> p1 ; cond_2 -> p2; p), q, r.

p:- q1, q2, q3.  p1:- q1, q2, q3 //   p2:- q1, q2, q3 //
p:- r1, r2.      p1:- r1, r2 //       p2:- r1, r2.
p:- s1, s2.      p1:- s1, s2.         p2:- s1, s2.
p.               p1.                  p2.
```

En este ejemplo, la directiva `//` declara tres grupos para el predicado `p1`: el primero y el segundo están compuestos de las cláusulas primera y segunda respectivamente, y el tercer grupo compuesto por las cláusulas tercera y cuarta (estas dos cláusulas se ejecutan, o exploran, secuencialmente). Además, para el predicado `p2` tenemos dos grupos: el primero compuesto por la primera cláusula y el segundo compuesto por las cláusulas segunda, tercera y cuarta.

2.5. Reducción del coste del control de granularidad

Inevitablemente, las transformaciones propuestas introducen nuevos costes en la ejecución. Por tanto, sería deseable reducir estos costes lo máximo posible. Con esta finalidad proponemos optimizaciones, las cuales incluyen simplificación de tests, cálculo eficiente del tamaño de términos, y parar el control de granularidad cuando podamos determinar que un objetivo no producirá tareas candidatas para su ejecución paralela, en cuyo caso se ejecuta una versión que no realiza control de granularidad.

Antes de comenzar la discusión de las optimizaciones que proponemos, necesitamos definir algunos términos. En primer lugar, recordamos la noción de "tamaño" de un término. Se pueden utilizar varias métricas para determinar el "tamaño" de un término, p.e., número de nodos en un término (term-size), profundidad del término (term-depth), longitud de lista (list-length), valor entero (integer-value), etc. [DL93]. La métrica apropiada para una situación determinada, generalmente se puede determinar examinando las operaciones que realiza el programa. Sea $|\cdot|_m : \mathcal{H} \to \mathcal{N}_\perp$ una función que hace corresponder a los términos totalmente instanciados (e.d. "ground") su tamaño de acuerdo con alguna métrica específica m, en donde $\mathcal{H}$ denota el Universo de Herbrand, e.d. el conjunto de términos "ground" del lenguaje, y $\mathcal{N}_\perp$ denota el conjunto de los números naturales aumentado con el símbolo especial $\perp$, que significa "no definido". Algunos ejemplos de dichas funciones son "list_length", que asigna a las listas "ground" sus longitudes, y a todos los demás términos "ground" el símbolo $\perp$; "term_size", que asigna a cada término "ground" el número de constantes y símbolos de función que aparecen en él; "term_depth", que asigna a cada término "ground" la profundidad de su representación en árbol; etc. Por tanto, $|[\mathtt{a},\mathtt{b}]|_{\mathrm{list\_length}} = 2$, en cambio $|f(a)|_{\mathrm{list\_length}} = \perp$. Extendemos la definición de $|\cdot|_m$ a tuplas de términos de la manera obvia, definiendo la función $Siz_m : \mathcal{H}^n \mapsto \mathcal{N}_\perp{}^n$, tal que $Siz_m((t_1, \ldots, t_n)) = (|t_1|_m, \ldots, |t_n|_m)$. Supongamos que I y I' denotan dos tuplas de términos, Φ un conjunto de sustituciones y θ una sustitución. Supongamos además que definimos el conjunto de estados correspondientes a un punto determinado de una cláusula, como aquellos estados cuyo objetivo que está más a la izquierda en el resolvente corresponde al literal que hay después de ese punto de programa. También definimos el conjunto de sustituciones en ese punto de la cláusula de manera análoga.

Definición 2.5.1 [Función *comp*] Dado un estado s_1 correspondiente a un punto de una cláusula p_1, la sustitución actual θ correspondiente a ese estado, y otro punto de cláusula p_2, definimos $comp(\theta, p_2)$ como el conjunto de sustituciones en el punto p_2 que corresponden a estados que están en la misma derivación que s_1. ∎

Definición 2.5.2 [Tamaños directamente calculables] Consideremos un conjunto Φ de sustituciones en el punto de la cláusula p_1 y otro punto de cláusula p_2. $Siz_m(I')$ es directamente computable en p_2 a partir de $Siz_m(I)$ con respecto a Φ si existe una función (computable) ψ tal que para todo θ, θ', $\theta \in \Phi$, y $\theta' \in comp(\theta, p_2)$, $Siz_m(I\theta)$ está definido y $Siz_m(I'\theta') = \psi(Siz_m(I\theta))$. ∎

Definición 2.5.3 [Equivalencia de expresiones] Dos expresiones E y E' son equivalentes con respecto al conjunto de sustituciones Φ si para todo $\theta \in \Phi$ $E\theta$ devuelve el mismo valor que $E'\theta$ al evaluarse. ∎

2.5.1. Simplificación de tests

Informalmente, podemos ver la simplificación de tests de la forma siguiente: el punto de partida es una expresión que es una función del tamaño de un conjunto de términos. Intentamos encontrar una expresión que sea equivalente a ella pero que sea una función que dependa de un conjunto menor de términos. Además, aplicamos simplificaciones aritméticas estándar a esta expresión. Dado que esta nueva expresión tendrá menos variables, la simplificación será más fácil y la expresión simplificada correspondiente será menos costosa de calcular. Describimos a continuación la noción de simplificación de expresiones formalmente. Consideremos el conjunto de sustituciones Φ' en el punto de cláusula p_2, justo antes de la ejecución del objetivo t. Supongamos que tenemos una expresión $E(Siz_m(I'))$ para evaluar en p_2. El objetivo es encontrar un punto de programa p_1 y una tupla de términos I tal que $Siz_m(I')$ es directamente computable en p_2 a partir de $Siz_m(I)$ con respecto a Φ con la función ψ, en donde Φ es el conjunto de sustituciones en el punto de cláusula p_1 y, o bien $p_1 = p_2$ o p_1 precede a p_2 y $E(Siz_m(I'))$ aparece después de p_1. Tenemos que $E(\psi(Siz_m(I))$ es equivalente a $E(Siz_m(I'))$ con respecto a Φ'. Entonces podemos calcular una expresión E' que es equivalente a $E(\psi(Siz_m(I))$ (mediante simplificaciones) con respecto a Φ' y cuyo coste de evaluación es menor que el de $E(\psi(Siz_m(I))$. Con el siguiente ejemplo ilustramos este tipo de optimización.

Ejemplo 2.5.1 Consideremos el objetivo `..., q(X,Y) & r(X, Z), ...` en el ejemplo 2.3.1. En este ejemplo $I = I' = (X)$; $Siz(I')$ es directamente computable a partir de $Siz(I)$ con respecto a Φ con ψ, en donde ψ es la función identidad. $Siz(I\theta)$ está definida para todo θ en Φ, ya que X está ligada a una lista "ground". Por tanto, tenemos que para todo $\theta \in \Phi$ y para todo $\theta' \in comp(\theta, p_2)$, $Siz(I'\theta') = \psi(Siz(I\theta))$. $E(Siz(I)) \equiv max(2\ Siz(X) + 1, 3\ Siz(X) + 1) + 15 \leq 2\ Siz(X) + 1 + 3\ Siz(X) + 1$. Calculemos ahora E'. Tenemos que para todo $\theta \in \Phi$, $max(2\ Siz(X) + 1, 3\ Siz(X) + 1) \equiv 3\ Siz(X) + 1$, por tanto tenemos $3\ Siz(X) + 1 + 15 \leq 2\ Siz(X) + 1 + 3\ Siz(X) + 1$ la cual puede simplificarse a $15 \leq 2\ Siz(X) + 1$ y posteriormente a $7 \leq Siz(X)$ que es E'. Usando esta expresión conseguimos un programa transformado más eficiente que el del ejemplo 2.3.1:

```
 ..., length(X, LX),
      (LX > 7 ->
           q(X, Y) & r(X, Z)
           ;
           q(X, Y), r(X, Z)), ...
```

□

En algunos casos, la simplificación de tests evita el tener que evaluar funciones de coste, de modo que los tamaños de términos se comparan directamente con algún umbral. Supongamos que tenemos un test de la forma $Cost_p(n) > G$ en donde G es un número y $Cost_p(n)$ una función monótona de coste que depende de una variable, para algún predicado p. En este caso, podemos encontrar un valor k tal que $Cost_p(k) \leq G$ y $Cost_p(k+1) > G$, de modo que la expresión $Cost_p(n) > G$ se puede simplificar a $n > k$.

2.5.2. Interrupción del control de granularidad

Una optimización importante encaminada a reducir el coste del control de granularidad se basa en detectar cuándo se cumple recursivamente una invariante sobre la condición para realizar paralelización/secuencialización y ejecutar en esos casos una versión del predicado que no realiza control de granularidad y que ejecuta de la forma apropiada que corresponde a la invariante.

Ejemplo 2.5.2 Consideremos el predicado `qsort/2` definido de la forma siguiente:

```
qsort([], []).
qsort([First|L1], L2) :- partition(First, L1, Ls, Lg),
                         (qsort(Ls, Ls2) & qsort(Lg, Lg2)),
                         append(Ls2, [First|Lg2], L2).
```

La transformación siguiente realizará control de granularidad basado en la condición dada en el teorema 2.1.2 y la detección de una invariante (los tests se han simplificado ya — omitimos los detalles — de forma que los tamaños de los datos de entrada se comparan directamente con un umbral):

```
g_qsort([], []).
g_qsort([First|L1], L2) :-
  partition(First, L1, Ls, Lg),
  length(Ls, SLs), length(Lg, SLg),
  (SLs > 20 -> (SLg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2);
                            g_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))
            ;  (SLg > 20 -> s_qsort(Ls, Ls2) , g_qsort(Lg, Lg2);
                            s_qsort(Ls, Ls2) , s_qsort(Lg, Lg2))),
  append(Ls2, [First|Lg2], L2).


s_qsort([], []).
s_qsort([First|L1], L2) :-
    partition(First, L1, Ls, Lg),
    s_qsort(Ls, Ls2), s_qsort(Lg, Lg2),
    append(Ls2, [First|Lg2], L2).
```

Nótese que si el tamaño del dato de entrada es menor que el umbral (20 unidades de cómputo en este caso[4]) entonces se ejecuta una versión (secuencial) que no realiza control de granularidad. Esto se basa en la detección de una invariante recursiva: en subsecuentes recursiones este objetivo no producirá tareas con tamaños mayores o iguales que el umbral, y por tanto, para todas ellas, la ejecución debe realizarse secuencialmente y obviamente no se necesita realizar ningún control de granularidad. En [GH91] se describen técnicas para detectar tales invariantes. □

2.5.3. Reducción del coste del cálculo de tamaños

El enfoque estándar dado al problema del cálculo de tamaños consiste en recorrer explícitamente los términos, utilizando predicados predefinidos como `length`/2. Sin embargo, dicho cálculo puede además realizarse de otras formas que potencialmente reducen el coste en tiempo de ejecución:

1. En caso de que los tamaños de datos de entrada a los subobjetivos en el cuerpo que son candidatos para ejecución paralela, sean directamente computables a partir de los de la cabeza de la cláusula (como ocurre por ejemplo en el programa "Fibonacci" clásico – véase el ejemplo 2.7.1) dichos tamaños pueden calcularse realizando una operación aritmética. Los tamaños de los datos de entrada los pueden suministrar las cabezas de las cláusulas mediante el uso de argumentos adicionales.

2. En otro caso el cálculo de tamaños de términos puede simplificarse mediante la transformación de ciertos procedimientos para que calculen los tamaños de los términos "sobre la marcha" [LGH95]. Esta técnica de transformación se describe detalladamente en el capítulo 5.

[4]Este umbral de determina experimentalmente, tomando el valor medio resultante de varias ejecuciones.

3. En los casos en donde los tamaños de los términos se comparan directamente con un umbral no es necesario recorrer totalmente los términos, sino hasta el punto en el cual se alcanza el umbral.

2.6. Estimación del coste del control de granularidad

Como resultado de las simplificaciones propuestas en las secciones anteriores, se pueden generar tres versiones diferentes para un predicado: secuencial, paralela sin control de granularidad, y paralela con control de granularidad. En esta sección abordamos el problema de cómo hacer la selección entre éstas. Podemos ver este problema de forma análoga al del problema original de decidir entre ejecución paralela y secuencial, ya tratado en la sección 2.1, pero en este caso tenemos el problema adicional de decidir si se realiza o no control de granularidad. Sean T_s, T_p, y T_g los tiempos de ejecución de las versiones secuencial, paralela, y con control de granularidad respectivamente del predicado correspondiente a una llamada determinada. El problema original tratado en la sección 2.1 puede verse cómo determinar $min(T_s, T_p, T_g)$. De nuevo, esto no se puede calcular antes de la ejecución de los objetivos y otra vez más nos vemos forzados a calcular una aproximación basada en heurísticas o en condiciones suficientes. Como hicimos anteriormente adoptamos el último enfoque, e.d. utilizar condiciones suficientes, las cuales intentaremos calcular en principio para cada una de los seis posibles casos: $T_g \leq T_s$, $T_p \leq T_s$, $T_p \leq T_g$, $T_s \leq T_g$, $T_s \leq T_p$ y $T_g \leq T_p$. Ya que sólo podemos aproximar estas condiciones, un problema importante es la decisión tomada cuando no podamos probar que se cumple alguna de estas condiciones. Una posible solución es tener una relación de orden predeterminado, la cual se usa a menos que se pueda demostrar que se cumple otra relación. Esto corresponde a los dos casos de "secuencialización por defecto" o "paralelización por defecto" que ya estudiamos en la sección 2.1, y en donde solamente considerábamos una condición. Por ejemplo, una ordenación por defecto podría ser: $T_g \leq T_s \leq T_p$, la cual expresa la suposición de que el tiempo de ejecución óptimo se consigue cuando la ejecución se realiza en paralelo con control de granularidad a menos que se pueda demostrar lo contrario. Además, los objetivos se ejecutan secuencialmente a menos que se pueda demostrar que la ejecución paralela tardará menos tiempo. Si forzamos que se cumpla la condición de "sin-pérdida" (no-slow-down), e.d. requerimos que no se exceda el tiempo de ejecución secuencial, entonces, en todas las relaciones de orden predeterminadas tenemos que tener que $T_s \leq T_g$ y $T_s \leq T_p$.

Nótese que estas relaciones de orden predeterminadas pueden ser parciales en cuyo caso, tendremos que aplicar alguna heurística en algún punto. Podemos determinar entonces el orden entre dos costes T_1 y T_2 de la forma siguiente: Si T_1 y T_2 están relacionados en la relación de orden predeterminada, entonces calculamos una condición suficiente para demostrar el orden opuesto; en otro caso, si se cumple alguna condición suficiente para probar algunas de las relaciones $T_1 \leq T_2$ o $T_2 \leq T_1$ entonces elegimos la que corresponda; en otro caso podemos determinar el orden mediante alguna heurística. Una buena heurística puede ser utilizar la media entre las cotas inferiores y superiores que ya hemos calculado, o tomar la media de los costes calculados para las diferentes cláusulas de un predicado.

2.7. Determinación de T_p y T_g para una llamada

El problema de la determinación de una cota para T_s lo hemos tratado ya en las secciones previas, en donde asumíamos simplemente que T_p era el mismo que T_s, tomando como su aproximación la cota opuesta a la usada para T_s. Ahora tratamos el problema de la determinación más precisa de T_p y también el de la determinación de T_g. Por brevedad, presentaremos las técnicas mediante un ejemplo.

Ejemplo 2.7.1 Consideremos una versión transformada `gfib/2` del predicado `fib/2` la cual realiza control de granularidad en tiempo de ejecución:

```
gfib(0, 0).
gfib(1, 1).
gfib(N, F):- N1 is N - 1, N2 is N - 2,
            (N > 15 -> gfib(N1, F1) & gfib(N2, F2)
                     ; sfib(N1,F1), sfib(N2,F2)),  F is F1 + F2.

sfib(0, 0).
sfib(1, 1).
sfib(N, F):- N > 1, N1 is N - 1, N2 is N - 2,
             sfib(N1, F1), sfib(N2, F2),
             F is F1+F2.
```

□

2.7.1. Coste de la ejecución paralela sin control de granularidad: T_p

Cotas superiores

En general es difícil dar una cota superior no trivial del coste de la ejecución paralela de un conjunto de tareas, ya que es difícil de predecir el número de procesadores libres que estarán disponibles en tiempo de ejecución. Nótese que una cota superior trivial se puede calcular en algunos casos suponiendo que todos los objetivos que potencialmente se pueden ejecutar en paralelo se crean como tareas separadas pero todas ellas se ejecutan en un mismo procesador. Consideremos el predicado `fib/2` definido en el ejemplo 2.7.1. Denotemos como Is el tamaño del dato de entrada (primer argumento) y como $Tp(Is)$ el coste de la ejecución paralela sin control de granularidad de una llamada al predicado `fib`/2 para una entrada de tamaño Is. Podemos plantear la siguiente ecuación en diferencias para la cláusula recursiva de `fib`/2: $Tp^u(Is) = C_b^u(Is) + Spaw^u(Is) + Sched^u(Is) + Tp^u(Is-1) + Tp^u(Is-2) + C_a^u(Is)$ para $Is > 1$, en donde $C_b(Is)$ y $C_a(Is)$ representan los costes de la ejecución secuencial de los literales antes y después de la llamada respectivamente, es decir, $C_b(Is)$ representa el coste de `N1 is N-1,N2 is N-2` y $C_a(Is)$ el coste de `F is F1+F2`. La solución a esta ecuación en diferencias da el coste de una llamada a `fib`/2 para un dato de entrada de tamaño Is. De los casos base obtenemos las siguientes condiciones iniciales para las ecuaciones: $Tp^u(0) = 1$ y $Tp^u(1) = 1$.

Cotas inferiores

Una cota inferior trivial — teniendo en cuenta información de no-fallo, tal y como se discute en el capítulo 3 y en [DLGHL97] — puede calcularse de la forma siguiente: $Tp^l(Is) = \frac{W_p^l(Is)}{p}$, en donde W_p^l representa el trabajo realizado por la ejecución paralela sin control de granularidad de una llamada al predicado `fib`/2 para una entrada de tamaño Is, y puede calcularse resolviendo la siguiente ecuación en diferencias: $W_p^l(Is) = C_b^l(Is) + Spaw^l(Is) + Sched^l(Is) + W_p^l(Is-1) + W_p^l(Is-2) + C_a^l(Is)$ para $Is > 1$, con las condiciones iniciales: $W_p^l(0) = 1$ y $W_p^l(1) = 1$.

Como una alternativa, otro valor para $Tp^l(Is)$ puede obtenerse resolviendo la siguiente ecuación en diferencias: $Tp^l(Is) = C_b^l(Is) + Spaw^l(Is) + Sched^l(Is) + Tp^l(Is-1) + C_a^l(Is)$ para $Is > 1$, con las condiciones iniciales: $Tp^l(0) = 1$ y $Tp^l(1) = 1$. En este caso, consideramos un número infinito de procesadores. Ya que en cada "bifurcación" (fork) hay dos ramas, se elige la más larga de ellas ($Tp^u(Is-1)$).

2.7.2. Coste de la ejecución con control de granularidad: T_g

Cotas superiores

Podemos plantear la siguiente ecuación en diferencias para la cláusula recursiva de `fib/2`: $T_g^u(Is) = C_b^u(Is) + Test^u(Is) + Spaw^u(Is) + Sched^u(Is) + T_g^u(Is - 1) + T_g^u(Is-2) + C_a^u(Is)$ para $Is > 15$. Asumimos que todos los objetivos potencialmente paralelos se crean como tareas separadas pero todas ellas se ejecutan en un mismo procesador, como hemos supuesto en la sección 2.7.1.

Para una llamada con $Is = 15$ no hay coste (overhead) asociado con la ejecución paralela ya que la ejecución se realiza secuencialmente, de modo que obtenemos las siguientes condiciones iniciales: $T_g^u(15) = Test^u(15) + Ts^u(15)$; y $T_g^u(Is) = Ts^u(15)$ para $Is \leq 15$, en donde $Ts^u(15)$ denota una cota superior del tiempo de ejecución secuencial de una llamada a `fib/2` con un dato de entrada de tamaño 15.

Cotas inferiores

Podemos calcular una cota inferior trivial (teniendo en cuenta información de no-fallo) de la forma siguiente: $T_g^l(Is) = \frac{W_g^l(Is)}{g}$, en donde W_g^l representa el trabajo realizado por la ejecución con control de granularidad de una llamada a `fib/2` para un dato de entrada de tamaño Is, el cual lo podemos calcular resolviendo la siguiente ecuación en diferencias: $W_g^l(Is) = C_b^l(Is) + Test^l(Is) + Spaw^l(Is) + Sched^l(Is) + W_g^l(Is-1) + W_g^l(Is-2) + C_a^l(Is)$ para $Is > 15$, con las condiciones iniciales: $W_g^l(15) = Test^l(15) + Ts^l(15)$, y $W_g^l(Is) = Ts^l(15)$ para $Is \leq 15$, en donde $Ts^l(15)$ denota una cota inferior del tiempo de la ejecución secuencial de una llamada a `fib/2` con un dato de entrada de tamaño 15.

Otro valor para $T_g^l(Is)$ puede obtenerse resolviendo la siguiente ecuación en diferencias: $T_g^l(Is) = C_b^l(Is) + Test^l(Is) + Spaw^l(Is) + Sched^l(Is) + T_g^l(Is-1) + C_a^l(Is)$ para $Is > 15$, con las condiciones iniciales: $T_g^l(15) = Test^l(15) + Ts^l(15)$, y $T_g^l(Is) = Ts^l(15)$ para $Is \leq 15$.

Cuadro 2.1: Resultados experimentales para programas en el sistema &-Prolog.

programas	**seq**	**ngc**	**gc**	**gct**	**gcts**	**gctss**
fib(19)	1.839	0.729	1.169	0.819	0.819	0.549
(**O**=m)		1	-60 %	-12 %	-12 %	+24 %
fib(19)	1.839	0.970	1.389	1.009	1.009	0.639
(**O**=5)		1	-43 %	-4.0 %	-4.0 %	+34 %
hanoi(13)	6.309	2.509	2.829	2.419	2.399	2.399
(**O**=m)		1	-12.8 %	+3.6 %	+4.4 %	+4.4 %
hanoi(13)	6.309	2.690	2.839	2.439	2.419	2.419
(**O**=5)		1	-5.5 %	+9.3 %	+10.1 %	+10.1 %
unb_matrix	2.099	1.009	1.339	1.259	0.870	0.870
(**O**=m)		1	-32.71 %	-24.78 %	+13.78 %	+13.78 %
unb_matrix	2.099	1.039	1.349	1.269	0.870	0.870
(**O**=5)		1	-29.84 %	-22.14 %	+16.27 %	+16.27 %
qsort(1000)	3.670	1.399	1.790	1.759	1.659	1.409
(**O**=m)		1	-28 %	-20 %	-19 %	-0.0 %
qsort(1000)	3.670	1.819	2.009	1.939	1.649	1.429
(**O**=5)		1	-11 %	-6.6 %	+9.3 %	+21 %

2.8. Resultados experimentales

Hemos desarrollado un prototipo de un sistema control de granularidad basado en las ideas descritas. El prototipo actual presenta dos inconvenientes: sólo

Cuadro 2.2: Resultados experimentales para programas en el sistema Muse.

programas	**seq**	**ngc**	**gctss**	**opt**	$\mathbf{e_1}$	$\mathbf{e_2}$
queens(8)	17.019	2.090	1.759	1.702	+15.84 %	+86.83 %
domino(12)	37.049	4.459	4.139	3.705	+7.18 %	+42.43 %
series	22.429	7.360	4.860	2.243	+33.97 %	+48.86 %
farmer	17.929	2.170	2.149	1.793	+0.97 %	+5.57 %

considera el caso de paralelismo independiente conjuntivo a nivel de objetivo, y utiliza un analizador de tipos cuya precisión no es suficiente en algunos casos. A pesar de esto, es capaz de realizar un control de granularidad totalmente automático en tres de los cuatro programas de prueba utilizados (fib, hanoi, y qsort). Los resultados se muestran en la tabla 2.1. Para el otro programa de prueba (unb_matrix) y para el caso de paralelismo disyuntivo hemos tenido que anotar a mano los programas siguiendo los algoritmos presentados y suponiendo que contábamos con la tecnología actual de inferencia de tipos más precisa que la incorporada actualmente al prototipo. Estos resultados se muestran en las tablas 2.1 y 2.2. Creemos que completando la implementación para el caso del paralelismo disyuntivo e incorporando uno de los análisis de tipos existentes más precisos, es posible también realizar un control de granularidad completamente automático para este caso, y que los resultados obtenidos por el sistema pueden suponer una mejora importante en los tiempos de ejecución.

En primer lugar hemos probado el sistema de control de granularidad con el sistema &-Prolog [HG91], un sistema paralelo para Prolog, en una máquina multiprocesador Sequent Symmetry utilizando 4 procesadores. Los resultados del control de granularidad se muestran en la tabla 2.1 (los tiempos de ejecución vienen dados en segundos). Hemos probado cuatro programas representativos y utilizado dos niveles de costes de creación y lanzamiento de tareas en paralelo (**O**): mínimo (**m**), que representa el coste del sistema de memoria compartida &-Prolog (el cual es muy pequeño – unos pocos microsegundos), y un coste (el sistema &-Prolog permite añadir costes de creación de tareas arbitrarios en tiempo de ejecución mediante un "switch") de 5 milisegundos (**5**), el cual se puede considerar representativo de un sistema de memoria jerárquica o una implementación eficiente de un multicomputador con una red de interconexión muy rápida. El programa `unb_matrix` realiza la multiplicación de dos matrices, una de 4×2 y la otra de 2×1000. Los resultados vienen dados para varios niveles de optimización del proceso de control de granularidad: control de granularidad sin ninguna optimización (`gc`), incorporando simplificación de tests (`gct`), incorporando la interrupción del control de granularidad (`gcts`), e incorporando el cálculo de tamaños "sobre la marcha" (`gctss`). Además se muestran los resultados para la ejecución secuencial (`seq`) y la ejecución paralela sin control de granularidad (`ngc`) para poder compararlas. Las ganancias obtenidas se han calculado respecto a `ngc`. La importancia de las optimizaciones propuestas se pone de manifiesto por el hecho de que las ganancias se van incrementando a medida que se van incorporando optimizaciones. Además, y exceptuando el caso de `qsort` en un sistema con un coste muy bajo de creación de tareas paralelas, las versiones totalmente optimizadas son bastante más eficientes que las que no realizan ningún control de granularidad. Es de destacar que las situaciones que se han estudiado corresponden a máquinas de memoria compartida, y que tienen un coste pequeño de creación de tareas en paralelo, es decir, en condiciones en donde el control de granularidad es menos ventajoso. Por tanto, estos resultados pueden verse como cotas inferiores de la mejora potencial debida al control de granularidad. Obviamente, en sistemas con mayores costes de creación de tareas, tales como sistemas distribuidos, las ganancias pueden ser mucho mayores.

Respecto al paralelismo disyuntivo, en la tabla 2.2 se presentan resultados del control de granularidad (los tiempos de ejecución vienen dados en segundos) para programas de prueba ejecutados en el sistema Muse [AK90] utilizando 10 agentes, y una máquina multiprocesador Sequent Symmetry con 10 procesadores. **queens(8)** calcula todas las soluciones del problema de las 8 reinas. **domino(12)** calcula todas las secuencias legales de 12 dominós. **series** calcula una serie cuya expresión es una disyunción de series. **farmer** es un puzle de ECRC. Los resultados corresponden a las versiones totalmente optimizadas, las cuales realizan control de granularidad (`gctss`), la ejecución secuencial (`seq`) y la ejecución paralela sin control de granularidad (`ngc`) de forma que se puedan comparar. **opt** es una cota inferior del tiempo de ejecución óptimo, e.d. $\mathbf{opt} = \frac{\mathbf{seq}}{10}$. $\mathbf{e_1} = \frac{\mathbf{ngc}-\mathbf{gctss}}{\mathbf{ngc}} \times 100$, y $\mathbf{e_2} = \frac{\mathbf{ngc}-\mathbf{gctss}}{\mathbf{ngc}-\mathbf{opt}} \times 100$ indica el porcentaje de tiempo ahorrado con respecto a la ejecución paralela sin control de granularidad y el tiempo de la ejecución paralela ideal respectivamente, cuando se realiza control de granularidad. Nótese que algunos programas no presentan el suficiente paralelismo inherente como para alcanzar este tiempo ideal de ejecución incluso cuando no haya ningún tipo de coste asociado con la ejecución paralela. La razón por la cual utilizamos estas dos métricas es que el sistema Muse ejecutaba muy eficientemente los programas de prueba seleccionados. Esto se debe a que el "scheduler" de Muse realiza un control implícito de paralelismo dependiendo de la carga del sistema. Por tanto, los beneficios potenciales de la aplicación de nuestro control de granularidad eran muy limitados. Estas métricas nos permiten llegar a la conclusión de que los resultados son de hecho muy buenos, ya que logran una parte significativa de los beneficios potenciales. Nótese además que las situaciones estudiadas corresponden a máquinas pequeñas de memoria compartida, y que por tanto, como en el caso del paralelismo conjuntivo, los resultados pueden verse como cotas inferiores de la mejora potencial debida al control de granularidad.

2.9. Conclusiones del capítulo

Hemos desarrollado completamente (e integrado en el sistema `ciaopp` [HBPLG99, HBC+99]) un sistema automático de control de granularidad para programas lógicos basado en el esquema de análisis y transformación de programas, en el cual se realiza tanto trabajo como sea posible en tiempo de compilación.

Hemos discutido los muchos problemas que surgen en la realización del control de granularidad (para los casos en los que se dispone de información de cotas inferiores y superiores de coste) y dado soluciones a los mismos con la suficiente generalidad para que los resultados obtenidos puedan aplicarse a otros sistemas, no necesariamente de programación lógica, y para diferentes modelos de ejecución. Creemos por tanto que los resultados presentados son de interés para su posible aplicación a otros lenguajes de programación paralelos.

Además hemos hecho una evaluación de las técnicas de control de granularidad para los casos de paralelismo conjuntivo y disyuntivo en los sistemas &-Prolog y Muse respectivamente, y hemos obtenido resultados esperanzadores.

De los resultados experimentales se desprende que no es esencial obtener el mejor tamaño de grano para un problema en concreto, sino que existe una cierta holgura en cuanto a la precisión con la que se calcula el mismo. Esto nos sugiere que los compiladores podrían realizar un control de granularidad automático que sea útil en la práctica.

Podemos concluir que el análisis/control de granularidad es una técnica particularmente prometedora ya que tiene el potencial de hacer posible la explotación automática de arquitecturas paralelas, tales como estaciones de trabajo en una red de área local (posiblemente de alta velocidad).

Capítulo 3

Análisis de cotas inferiores del coste de procedimientos

Generalmente se reconoce que la información sobre los costes de tiempos de ejecución pueden ser útiles para una gran variedad de aplicaciones, entre las cuales se incluyen la transformación de programas, el control de granularidad durante la ejecución paralela [LGHD96, DLH90, ZTD$^+$92, HLA94, RM90, Kap88], y la optimización de consultas en bases de datos deductivas [DL90]. En el contexto de la programación lógica, la mayor parte del trabajo realizado hasta la fecha sobre estimación de coste en tiempo de compilación se ha centrado en la obtención de cotas superiores [DL93]. Sin embargo, en muchas aplicaciones es interesante trabajar con cotas inferiores [DLGHL97], como por ejemplo en la realización de control de granularidad en sistemas paralelos distribuidos.

A modo de ejemplo, consideremos una implementación de Prolog con memoria distribuida. Supongamos que el trabajo necesario para lanzar un objetivo en un procesador remoto es de 1.000 instrucciones, y que somos capaces de inferir que una llamada a un procedimiento no realizará más de 5.000 instrucciones. Esto nos sugiere que puede que valga la pena ejecutar esta llamada en un procesador remoto, pero no nos asegura que no se producirá una pérdida de eficiencia con respecto a la ejecución secuencial (la llamada podría terminar después de realizar sólo un número pequeño de instrucciones). Por otra parte, si sabemos que una llamada realizará al menos 5.000 instrucciones, podemos asegurar que merece la pena ejecutar la tarea en un procesador remoto. Por lo tanto, aunque es mejor tener información sobre cotas superiores que no disponer de ninguna, las cotas inferiores pueden ser mucho más útiles que las superiores.

En el análisis de cotas inferiores, la información acerca de qué llamadas no fallarán es de vital importancia, ya que de no disponer de la misma habría que asumir que una determinada llamada puede fallar al realizarse la unificación con la cabeza de una cláusula, en cuyo caso el coste de la llamada es nulo. Por ello nos vemos obligados a desarrollar un análisis que proporcione dicha información, el cual se describe en detalle en el capítulo 4. En éste nos centramos en la estimación de cotas inferiores del coste de procedimientos, y sus contribuciones principales son las siguientes:

1. Mostramos cómo se puede utilizar la información de no-fallo para inferir cotas inferiores no triviales del coste de llamadas a procedimientos.

2. Mostramos cómo la información sobre el número de soluciones puede mejorar la estimación de las cotas inferiores de coste en el caso de que se requieran todas las soluciones de una llamada.

3. Mostramos cómo se pueden obtener cotas inferiores de coste para una clase simple, pero común, de predicados del tipo "divide y vencerás".

4. Comentamos una implementación que hemos realizado del análisis de coste, la cual hemos integrado en el sistema `ciaopp` [HBPLG99, HBC$^+$99].

5. Finalmente mostramos (en la sección 3.4), que las estimaciones sobre cotas

inferiores hechas por dicha implementación son bastante buenas, especialmente si consideramos que se trata de una herramienta automática.

3.1. Análisis de cotas inferiores de coste cuando se requiere sólo una solución

Si para cualquier computación solamente se requiere una solución, únicamente es necesario saber si una computación generará como mínimo una solución, e.d., no fallará. Asumiendo que disponemos de dicha información, por ejemplo usando la técnica mencionada en la sección previa, el análisis de coste puede realizarse de la forma siguiente:

1. En primer lugar determinamos los tamaños relativos de ligaduras de variables en diferentes puntos de una cláusula calculando cotas inferiores de los tamaños de los argumentos de salida en forma de funciones que dependen de los parámetros de entrada. Esto puede realizarse resolviendo (o estimando cotas inferiores de) las ecuaciones en diferencias resultantes: el enfoque es muy similar al discutido en [DL93], la única diferencia estriba en que mientras que [DL93] calculaba cotas superiores de los tamaños de argumentos utilizando la función *max* sobre los tamaños de los datos de salida de diferentes cláusulas en un cluster, nosotros usamos la función *min* sobre las cláusulas para estimar cotas inferiores de los tamaños de los argumentos.

2. Una cota inferior del coste computacional de una clausula puede entonces expresarse como una función del tamaño del dato de entrada, en términos de los costes de los literales del cuerpo de esa cláusula.

   Consideremos una cláusula $C \equiv$ '$H :- B_1, \ldots, B_m$'. Sea n la r–tupla que representa los tamaños de los r argumentos de entrada de la cabeza de la cláusula, y sean $\phi_1(n), \ldots, \phi_m(n)$ (cotas inferiores de) los tamaños de los argumentos de entrada de los literales del cuerpo $B_1, \ldots, B_m$ respectivamente. Supongamos que el coste de la unificación con la cabeza y los tests para esta cláusula es al menos $h(n)$, y sea $Cost_{B_i}(x)$ una cota inferior del coste del literal del cuerpo B_i. Entonces, si B_k es el literal más a la derecha del cuerpo para el cual podemos asegurar que no fallará, una cota inferior del coste $Cost_C(n)$ de la cláusula C para un dato de entrada de tamaño n es:

$$h(n) + \sum_{i=1}^{k} Cost_{B_i}(\phi_i(n)) \ \leq \ Cost_C(n).$$

Si la cláusula C corresponde a un predicado que no falla, entonces tomamos $k = m$.

3. Una cota inferior del coste $Cost_p(n)$ de un predicado p para un dato de entrada de tamaño n viene entonces dada por:

$$\text{mín}\{Cost_C(n) \mid C \in cl(p)\} \ \leq \ Cost_p(n)$$

donde $cl(p)$ denota el conjunto de las cláusulas que definen p.

Como se discute en [DL93], la recursividad se trata expresando el coste de los objetivos recursivos simbólicamente como una función del tamaño de los datos de entrada. A partir de aquí, podemos obtener un conjunto de ecuaciones en diferencias que podemos resolver (o aproximar) para obtener una cota inferior del coste de un predicado en términos del tamaño de los datos de entrada.

Dado un predicado definido por m cláusulas $C_1, \ldots, C_m$, podemos mejorar la precisión de este análisis teniendo en cuenta que la cláusula C_i se ejecutará sólo si las cláusulas $C_1, \ldots, C_{i-1}$ fallan. Para un tamaño de entrada n, sea $\delta_i(n)$ la mínima cantidad de trabajo necesario para determinar que las cláusulas $C_1, \ldots, C_{i-1}$ no producirán ninguna solución y que la cláusula C_i debe de ejecutarse: la función δ_i

tiene que tener en cuenta obviamente el tipo y coste del esquema de indexación usado en la implementación subyacente. En este caso, la cota inferior para p puede mejorarse de la forma siguiente:

$$\text{mín}\{Cost_{C_i}(n) + \delta_i(n) \mid 1 \leq i \leq m\} \;\leq\; Cost_p(n).$$

También podemos tener en cuenta el operador de poda (corte), de modo que las cláusulas que están después de la primera, la cual denominamos C_i, que justo antes del corte tiene una secuencia de literales que no fallan, se ignoran, y la cota inferior del coste del predicado es entonces el mínimo de los costes de las cláusulas que preceden a la cláusula C_i y esta misma cláusula.

3.2. Análisis de cotas inferiores de coste cuando se requieren todas las soluciones

En muchas aplicaciones es razonable suponer que se requieren todas las soluciones. Por ejemplo, en una implementación con memoria distribuida de un sistema de programación lógica, el coste de mandar o recibir un mensaje puede ser lo suficientemente grande como para que tenga sentido que una computación remota calcule todas las soluciones de una consulta y las devuelva en un solo mensaje en lugar de enviar un gran número de mensajes, cada uno de ellos conteniendo una única solución. En tales casos, si disponemos de cotas inferiores del número de soluciones, podemos mejorar bastante la estimación del coste computacional de un objetivo—de hecho, como en el ejemplo de un sistema de memoria distribuida, en algunos casos, el número de soluciones puede ser en si misma una métrica razonable de coste.

Si obtenemos cotas inferiores del número de soluciones que pueden generar los literales de una cláusula (este problema está resuelto en [DLGHL97]), podemos usar esta información para mejorar las estimaciones de cotas inferiores para el caso en el que se requieren todas las soluciones de un predicado. Consideremos una cláusula '$p(\bar{x}) :- B_1, \ldots, B_n$' en donde B_k es el literal más a la derecha para el que podemos garantizar que no fallará. Sea n el tamaño del argumento de entrada para la cabeza de la cláusula, y sean $\phi_1(n), \ldots, \phi_m(n)$ (cotas inferiores de) los tamaños de los argumentos de entrada para los literales del cuerpo $B_1, \ldots, B_m$ respectivamente. Supongamos que el coste de la unificación con la cabeza y de los tests para esta cláusula es al menos $h(n)$, y sea $Cost_{B_i}(x)$ una cota inferior del coste del literal del cuerpo B_i. Consideremos ahora el literal B_j, en donde $1 \leq j \leq k+1$, e.d., está garantizado que todos los predecesores de B_j no fallarán. El número de veces que se ejecutará B_j viene dado por el número total de soluciones generadas por sus predecesores, e.d., los literales $B_1, \ldots, B_{j-1}$. Denotemos este número como N_j: podemos estimar N_j usando los algoritmos descritos en [DLGHL97]. Supongamos que el coste de unificación con la cabeza y los tests de esta cláusula es al menos $h(n)$, y sea $Cost_{B_i}(x)$ una cota inferior del coste del literal B_j del cuerpo de la cláusula. Podemos calcular entonces una cota inferior del coste de la cláusula para el caso en el que se requieren todas las soluciones de la forma siguiente:

$$h(n) + \sum_{i=1}^{k}(N_i \times Cost_{B_i}(\phi_i(n))) \ \leq \ Cost_C(n).$$

3.3. Estimación de costes en programas del tipo divide-y-vencerás

Un inconveniente importante de la técnica de estimación de coste que hemos presentado es su pérdida de precisión cuando se trata con programas del tipo divide-y-vencerás en los que los tamaños de los argumentos de salida de los predicados "divide" son dependientes.

Por ejemplo, en el típico programa “quicksort” (véase el ejemplo 3.3.1), dado que alguno de los dos argumentos (no los dos al mismo tiempo) de salida del predicado “partition” puede ser la lista vacía, el enfoque general calcula cotas inferiores asumiendo que ambos argumentos pueden ser la lista vacía *simultáneamente*, y por tanto infraestima significativamente el coste del programa. De alguna forma, la razón de esta falta de precisión es que el enfoque descrito hasta el momento es esencialmente un análisis independiente de atributos [JM81]. Sin embargo, incluso en el caso en el que dispongamos de un análisis relacional de atributos en el que se infieran relaciones entre los tamaños de diferentes argumentos de salida de un predicado, no es tan obvio como podemos, sistemáticamente, utilizar esta información para mejorar nuestras estimaciones de cotas inferiores de coste. Por ejemplo, para el programa “quicksort”, si la lista de entrada tiene longitud n, entonces las dos listas de salida del predicado “partition” tienen longitudes m y $n - m - 1$ para algún m, $0 \leq m < n$. La ecuación de coste resultante para la cláusula recursiva tiene la forma siguiente

$$C(n) = C(m) + C(n - m - 1) + \ldots \qquad (0 \leq m \leq n - 1)$$

Para determinar la solución de esta ecuación que es una cota inferior correspondiente al peor caso, necesitamos determinar el valor de m que minimiza la función $C(n)$. Hacer esto automáticamente, incluso cuando no sabemos cómo es $C(n)$, no parece trivial. Como una solución pragmática, argumentamos que puede ser posible obtener resultados útiles simplemente identificando y tratando clases comunes de programas del tipo divide-y-vencerás de forma especial. En muchos de estos programas, la suma de los tamaños de los datos de entrada a los predicados “divide” del cuerpo de la cláusula es igual al tamaño del dato de entrada de la cabeza de la cláusula menos alguna constante. Esta relación de tamaño se puede derivar en algunos casos mediante la técnica presentada en [DL93]. Sin embargo, esto no es posible en otros casos, ya que en esta técnica se trata cada argumento de salida como una función que depende solamente de los datos de entrada, independientemente de los tamaños de otros argumentos, y, como resultado, se pierden las relaciones entre los tamaños de los diferentes argumentos de salida (considérese por ejemplo el predicado `partition/4` definido en el ejemplo 3.3.1). Aunque el análisis puede tratar estos casos, puede perder precisión. Una posible solución para mejorar la precisión es usar uno de los enfoques recientemente propuestos para inferir relaciones de tamaño para esta clase de programas [BK96, GDL95].

Suponiendo que disponemos de las relaciones de tamaño mencionadas para estos programas, en la fase de análisis de coste obtenemos una expresión de la forma:

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k) + g(n)$ para $n > 0$, en donde k es un valor arbitrario tal que $0 \leq k \leq n-1$, C es una constante y $g(n)$ es una función.

en donde $y(n)$ denota el coste del predicado de tipo divide-y-vencerás para un dato de entrada de tamaño n y $g(n)$ es el coste de la parte del cuerpo de la cláusula que no contiene ninguna llamada al predicado de tipo divide-y-vencerás.

Para cada computación particular, obtenemos una sucesión de valores para k. Cada sucesión de valores para k obtiene una valor para $y(n)$.

A continuación discutimos cómo podemos calcular cotas inferiores/superiores para expresiones tales como $\mathtt{Cost}_{\mathtt{qsort}}(n)$ del ejemplo 3.3.1. Consideremos la expresión:

> $y(0) = C$,
> $y(n) = y(n-1-k) + y(k)$ para $n > 0$, en donde k es un valor arbitrario tal que $0 \leq k \leq n-1$ y C es una constante.

Un *árbol de cómputo* para dicha expresión es un árbol en el que cada nodo no-terminal se etiqueta con $y(n)$, $n > 0$, y tiene dos hijos $y(n-1-k)$ y $y(k)$

(parte izquierda y derecha respectivamente), en donde k es un valor arbitrario tal que $0 \leq k \leq n-1$. Los nodos terminales se etiquetan $y(0)$ y no tienen hijos. Supongamos que construimos un árbol para $y(n)$ haciendo un recorrido en profundidad. En cada nodo no terminal elegimos (arbitrariamente) un valor para k tal que $0 \leq k \leq n-1$. Decimos que la *sucesión de cómputo* del árbol es la sucesión de valores que hemos elegido para k en orden cronológico.

Lema 3.3.1 *Cualquier árbol de cómputo correspondiente a la expresión:*

> $y(0) = C$,
>
> $y(n) = y(n-1-k) + y(k)$ *para* $n > 0$, *en donde* k *es un valor arbitrario tal que* $0 \leq k \leq n-1$ *y* C *es una constante,*

tiene $n+1$ *nodos terminales y* n *nodos no terminales.*

Demostración Por inducción sobre n. Para $n = 0$ el teorema se cumple trivialmente. Supongamos que el teorema se cumple para todo m tal que $0 \leq m \leq n$, entonces, podemos probar que para todo m tal que $0 \leq m \leq n+1$ el teorema también se cumple razonando de la siguiente forma: tenemos que $y(n+1) = y(n-k) + y(k)$, en donde k es un valor arbitrario tal que $0 \leq k \leq n$. Dado que $0 \leq k \leq n$, también tenemos que $0 \leq n-k \leq n$, y, por la hipótesis de inducción, el número de nodos terminales en cualquier árbol de cómputo de $y(n-k)$ ($y(k)$, respectivamente) es $n-k+1$ ($k+1$, respectivamente). El número de nodos terminales en cualquier árbol de cómputo de $y(n+1)$ es la suma del número de nodos terminales de los hijos del nodo etiquetado con $y(n+1)$, e.d. $(n-k+1)+(k+1) = n+2$. Además, el número de nodos no terminales de cualquier árbol de cómputo de $y(n-k)$ ($y(k)$ respectivamente) es $n-k$ (k respectivamente). El número de nodos no terminales de cualquier árbol de cómputo de $y(n+1)$ es la suma del número de nodos no terminales de los hijos del nodo etiquetado con $y(n+1)$ más uno (el propio nodo $y(n+1)$, ya que es no terminal) e.d. $1+(n-k)+k = n+1$. ∎

Teorema 3.3.2 *Para cualquier árbol de cómputo correspondiente a la expresión:*

> $y(0) = C$,
>
> $y(n) = y(n-1-k)+y(k)$ *para* $n > 0$*, en donde* k *es un valor arbitrario tal que* $0 \leq k \leq n-1$ *y* C *es una constante,*

se cumple que $y(n) = (n+1) \times C$.

Demostración Según el lema 3.3.1, cualquier árbol de cómputo tiene $n+1$ nodos terminales con $y(0)$ y la evaluación de cada uno de estos nodos terminales es C. ∎

Teorema 3.3.3 *Dada la expresión:*

> $y(0) = C$,
>
> $y(n) = y(n-1-k) + y(k) + g(k)$ *para* $n > 0$*, en donde* k *es un valor arbitrario tal que* $0 \leq k \leq n-1$*,* C *es una constante y* $g(k)$ *una función,*

para cualquier árbol de cómputo correspondiente a ella, se cumple que $y(n) = (n+1) \times C + \sum_{i=1}^{n} g(k_i)$*, en donde* $\{k_i\}_{i=1}^{n}$ *es la sucesión de cómputo del árbol.*

Demostración Según el lema 3.3.1, cualquier árbol de cómputo tiene $n+1$ nodos terminales y n nodos no terminales. La evaluación de cada nodo terminal devuelve el valor C y cada vez que se evalúa un nodo no terminal i, se suma $g(k_i)$.

∎

Para minimizar (maximizar respectivamente) $y(n)$ podemos buscar una sucesión $\{k_i\}_{i=1}^{n}$ que minimice (maximice respectivamente) $\sum_{i=1}^{n} g(k_i)$. Esto es fácil cuando $g(k)$ es una función monótona, tal y como mostramos en el siguiente corolario.

Corolario 3.3.1 *Dada la expresión:*

> $y(0) = C$,
>
> $y(n) = y(n-1-k) + y(k) + g(k)$ *para* $n > 0$, *en donde* k *es un valor arbitrario tal que* $0 \leq k \leq n-1$, C *es una constante y* $g(k)$ *una función monótona creciente,*

Entonces, la sucesión $\{k_i\}_{i=1}^{n}$, *en donde* $k_i = 0$ *(*$k_i = n-1$ *respectivamente) para todo* $1 \leq i \leq n$ *da el mínimo (máximo respectivamente) valor para* $y(n)$ *para todos los árboles de cómputo.*

Demostración Se deduce del teorema 3.3.3 y del hecho de que $g(k)$ es una función monótona creciente. ∎

Del corolario 3.3.1 se deduce que la solución de la ecuación en diferencias (obtenida reemplazando k por 0):

> $y(0) = C$,
>
> $y(n) = y(n-1) + y(0) + g(0)$ para $n > 0$,

e.d. $(n+1) \times C + n \times g(0)$ es el mínimo de $y(n)$, y la solución de la ecuación en diferencias:

> $y(0) = C$,
>
> $y(n) = y(0) + y(n-1) + g(n-1)$ para $n > 0$,

e.d. $(n+1) \times C + n \times g(n-1)$ es el máximo de $y(n)$.

Nótese que podemos reemplazar $g(k)$ por cualquier cota inferior/superior de ella para calcular una cota inferior/superior de $y(n)$. También podemos tomar cualquier cota inferior/superior de $g(k_i)$. Por ejemplo, si $g(k)$ es una función monótona creciente entonces $g(k_i) \leq g(n-1)$ y $g(k_i) \geq g(0)$ para $1 \leq i \leq n$, por tanto, $y(n) \leq (n+1) \times C + n \times g(n-1)$ y $y(n) \geq (n+1) \times C + n \times g(0)$.

Supongamos ahora que la función g depende de n y de k:

Corolario 3.3.2 *Dada la expresión:*

$y(0) = C,$

$y(n) = y(n-1-k) + y(k) + g(n,k)$ *para* $n > 0$, *en donde* k *es un valor arbitrario tal que* $0 \leq k \leq n-1$, C *es una constante y* $g(n,k)$ *una función.*

Entonces, la solución a la ecuación en diferencias:

$f(0) = C,$

$f(n) = f(n-1) + C + L$ *para* $n > 0$,

en donde L *es una cota inferior/superior de* $g(n,k)$, *es una cota inferior/superior de* $y(n)$ *para todo* $n \geq 0$ *y para cualquier árbol de cómputo correspondiente a* $y(n)$. *En particular, si* $g(n,k)$ *es una función monótona creciente, entonces* $L \equiv g(1,0)$ *(*$L \equiv g(n, n-1)$ *respectivamente) es una cota inferior (superior respectivamente) de* $g(n,k)$.

Ejemplo 3.3.1 Veamos cómo podemos mejorar el análisis de cotas inferiores de coste, utilizando el enfoque descrito para programas del tipo divide-y-vencerás. En primer lugar consideraremos el análisis sin la incorporación de la optimización, y luego lo compararemos con el resultado obtenido cuando usamos la optimización.

Consideremos el predicado `qsort/2` definido de la forma siguiente:

```
qsort([], []).
qsort([First|L1], L2) :-
     partition(L1, First, Ls, Lg),
     qsort(Ls, Ls2), qsort(Lg, Lg2),
     append(Ls2, [First|Lg2], L2).
```

```
partition([], F, [], []).
partition([X|Y], F, [X|Y1], Y2) :-
        X =< F,
        partition(Y, F, Y1, Y2).
partition([X|Y], F, Y1, [X|Y2]) :-
        X > F,
        partition(Y, F, Y1, Y2).


append([], L, L).
append([H|L], L1, [H|R]) :- append(L, L1, R).
```

Sea $\texttt{Cost}_{\texttt{p}}(n)$ el coste (número de pasos de resolución) de una llamada al predicado `p` con un dato de entrada de tamaño n (en este ejemplo, la métrica de tamaño utilizada para todos los predicados es la longitud de lista [DL93]). La estimación de funciones de coste se realiza de forma ascendente ("bottom-up") como sigue:

La ecuación en diferencias obtenida para `append/3` es:

$$\texttt{Cost}_{\texttt{append}}(0, m) = 1$$
$$\texttt{Cost}_{\texttt{append}}(n, m) = 1 + \texttt{Cost}_{\texttt{append}}(n-1, m).$$

en donde la primera expresión expresa el coste de la unificación con la cabeza, y $\texttt{Cost}_{\texttt{append}}(n, m)$ es el coste de una llamada a `append/3` con listas de entrada de longitudes n y m (primer y segundo argumento, respectivamente). La solución de esta ecuación es: $\texttt{Cost}_{\texttt{append}}(n, m) = n + 1$. Dado que esta función depende sólo de n, usamos la función $\texttt{Cost}_{\texttt{append}}(n)$ en su lugar.

La ecuación en diferencias para `partition/4` es:

$$\texttt{Cost}_{\texttt{partition}}(0) = 1$$
$$\texttt{Cost}_{\texttt{partition}}(n) = 1 + \texttt{Cost}_{\texttt{partition}}(n-1).$$

en donde la primera expresión expresa el coste de la unificación con la cabeza, y $\texttt{Cost}_{\texttt{partition}}(n)$ da el coste de una llamada a `partition/4` con una lista de entrada (primer argumento) de longitud n. La solución de esta ecuación es: $\texttt{Cost}_{\texttt{partition}}(n) = n + 1$.

Para `qsort/2`, tenemos:

$$\texttt{Cost}_{\texttt{qsort}}(0) = 1$$
$$\texttt{Cost}_{\texttt{qsort}}(n) = 1 + \texttt{Cost}_{\texttt{partition}}(n-1) + 2 \times \texttt{Cost}_{\texttt{qsort}}(0) + \texttt{Cost}_{\texttt{append}}(0)$$

en donde la primera expresión expresa el coste de la unificación con la cabeza. En este caso la cota inferior calculada para el tamaño del dato de entrada de las llamadas a `qsort` y `append` es 0. Por tanto, la función de coste para `qsort/2` viene dada por:

$$\texttt{Cost}_{\texttt{qsort}}(0) = 1,$$
$$\texttt{Cost}_{\texttt{qsort}}(n) = n + 4, \text{ para } n > 0.$$

Usamos ahora la técnica descrita para programas del tipo divide-y-vencerás. Supongamos que utilizamos las expresiones:

$$\texttt{Cost}_{\texttt{qsort}}(0) = 1,$$
$$\begin{aligned}\texttt{Cost}_{\texttt{qsort}}(n) = {} & 1 + \texttt{Cost}_{\texttt{partition}}(n-1) + \texttt{Cost}_{\texttt{qsort}}(k) \\ & + \texttt{Cost}_{\texttt{qsort}}(n-1-k) + \texttt{Cost}_{\texttt{append}}(k), \text{ para } 0 \leq k \leq n-1 \text{ y } n > 0.\end{aligned}$$

Reemplazando valores obtenemos:

$$\begin{aligned}\texttt{Cost}_{\texttt{qsort}}(n) = {} & n + k + 2 + \texttt{Cost}_{\texttt{qsort}}(k) + \texttt{Cost}_{\texttt{qsort}}(n-1-k), \\ & \text{para } 0 \leq k \leq n-1.\end{aligned}$$

Según el corolario 3.3.2, dando a n y k el mínimo valor posible, e.d. 1 y 0 respectivamente, tenemos que $n+k+2 \geq 3$, y por tanto reemplazamos $n+k+2$ por 3 para obtener así una cota inferior de la anterior expresión, la cual da lugar a:

$$\mathtt{Cost}_{\mathtt{qsort}}(n) = 3 + \mathtt{Cost}_{\mathtt{qsort}}(k) + \mathtt{Cost}_{\mathtt{qsort}}(n-1-k), \text{ para } 0 \leq k \leq n-1.$$

que es equivalente a la ecuación en diferencias:

$$\mathtt{Cost}_{\mathtt{qsort}}(n) = 3 + 1 + \mathtt{Cost}_{\mathtt{qsort}}(n-1), \text{ para } n > 0.$$

La solución de esta ecuación es $\mathtt{Cost}_{\mathtt{qsort}}(n) = 4n + 1$, que claramente es una mejora de la cota inferior obtenida anteriormente. □

Podemos generalizar fácilmente los resultados anteriores para tratar programas recursivos múltiples del tipo divide-y-vencerás y programas en donde la suma de los tamaños de los datos de entrada de los predicados "divide" del cuerpo de la cláusula es igual al tamaño del dato de entrada de la cabeza de la cláusula menos alguna constante, la cual no tiene que ser 1 necesariamente.

3.4. Implementación

Hemos implementado un prototipo de sistema que calcula cotas inferiores del coste de procedimientos y tamaños de datos modificando el código fuente de CASLOG [DL93] el cual hemos integrado también en el sistema `ciaopp` [HBPLG99, HBC$^+$99].

El análisis es totalmente automático, y sólo necesita información de tipos referente al punto de entrada del programa. El resto de información, tipos, modos y medidas de tamaño la infiere el sistema automáticamente. En la tabla 3.1 se muestran algunos resultados sobre precisión y eficiencia del analizador de cotas inferiores. La segunda columna de la tabla muestra la función de coste (la cual depende del tamaño de los argumentos de entrada) inferidas por el análisis. T_{tms} es el tiempo requerido por los análisis de tipos, modos, y métrica de tamaños (SPARCstation 10, 55MHz, 64Mbytes de memoria), T_{nf} es el tiempo requerido por el análisis de no-fallo, y T_s y T_{ca} son los tiempos requeridos por los análisis de tamaño de datos y coste respectivamente. **Total** es el tiempo total de análisis ($Total = T_{tms} + T_{nf} + T_s + T_{ca}$). Todos los tiempos vienen dados en milisegundos.

Programa	**Función de Coste**	T_{tms}	T_{nf}	T_s	T_{ca}	**Total**
fibonacci	$\lambda x,1{,}447 \times 1{,}618^x + 0{,}552 \times (-0{,}618)^x - 1$	90	10	20	20	140
hanoi	$\lambda x.x2^x + 2^{x-1} - 2$	430	30	60	60	580
qsort	$\lambda x,4x + 1$	420	50	70	50	590
nreverse	$\lambda x,0{,}5x^2 + 1{,}5x + 1$	220	20	30	30	300
mmatrix	$\lambda\langle x, y\rangle,2xy + 2x + 1$	350	90	90	90	620
deriv	$\lambda x.x$	1010	80	170	120	1,380
addpoly	$\lambda\langle x, y\rangle x + 1$	220	70	40	30	360
append	$\lambda x.x + 1$	100	20	10	10	140
partition	$\lambda x.x + 1$	175	30	30	20	255
substitute	$\lambda\langle x, y, z\rangle.x$	70	50	110	100	330
intersection	$\lambda\langle x, y\rangle.x + 1$	150	130	20	30	260
difference	$\lambda\langle x, y\rangle.x + 1$	140	90	20	40	290

Cuadro 3.1: Precisión y eficiencia del análisis de cotas inferiores.

3.5. Aplicación a la paralelización automática

Hemos conectado el analizador de cotas inferiores de coste (véase la sección 3.4) con el sistema de control de granularidad descrito en el capítulo 2 (el cual está también integrado en el sistema CIAO como otra etapa, e incluye un anotador que transforma los programas para realizar control de granularidad). El resultado es un paralelizador de programas completo (basado en información de cotas inferiores de coste). Dado que nuestro objetivo en esta sección es simplemente estudiar la utilidad de las cotas inferiores estimadas, hemos elegido una estrategia simple para el control de granularidad: los objetivos se ejecutan en paralelo siempre que se pueda estimar que sus tamaños de grano son mayores que un umbral fijado, el cual es una constante para todos los programas. Además, las versiones de los programas que realizan control de granularidad son transformaciones fuente-fuente las cuales añaden tests a las versiones originales. En el capítulo 2 hacemos una discusión sobre estrategias más avanzadas que incluyen umbrales variables (los cuales dependen de parámetros tales como costes de comunicación de datos, número de procesadores, carga del sistema, etc.), transformaciones de más bajo nivel, y reagrupación de objetivos para aumentar la granularidad.

programas	**seq**	**ngc**	**gclb(175)**	**gclb(959)**
mmatrix(100)	52.389	74.760 (0.70)	29.040 (1.80)	27.981 (1.87)
mmatrix(50)	6.469	5.978 (1.08)	3.378 (1.92)	3.758 (1.72)
fib(19)	0.757	1.458 (0.52)	0.128 (5.93)	0.103 (7.32)
hanoi(13)	1.442	1.464 (0.98)	0.677 (2.13)	0.619 (2.33)
qsort(1000)	0.475	0.414 (1.15)	0.230 (2.06)	0.314 (1.51)
qsort(3000)	4.142	2.423 (1.71)	1.094 (3.79)	1.575 (2.63)

Cuadro 3.2: Resultados de control de granularidad para programas en ECL^{i}PS^e.

Hemos realizado algunos experimentos preliminares en los cuales se han paralelizado automáticamente unos programas de prueba en dos casos: con la opción correspondiente a la inclusión de control de granularidad activada y desactivada. Los programas resultantes se han ejecutado en el sistema ECL^{i}PS^e utilizando 10 agentes, y ejecutando en una máquina SUN SPARC 2000 SERVER con 10 procesadores. Hemos elegido este sistema, el cual implementa el paralelismo conjuntivo encima del disyuntivo, porque tiene unos costes asociados con la ejecución paralela considerablemente mayores que otros sistemas que implementa el paralelismo conjuntivo de forma nativa (como por ejemplo el sistema &-Prolog utilizado en el sistema CIAO). Como resultado, este sistema nos proporciona un reto interesante – demostró que era muy difícil obtener aceleraciones con paralelizadores previos que explotaban el paralelismo conjuntivo.

Los resultados obtenidos se muestran en la tabla 3.2. Los tiempos de ejecución se muestran en segundos. Los resultados se refieren a la ejecución secuencial (**seq**), ejecución paralela sin control de granularidad (**ngc**), y las versiones que realizan control de granularidad (**gclb(175)** y **gclb(959)**). Las dos cifras que aparecen corresponden a dos valores diferentes para el umbral, e ilustran que los resultados presentan una baja variabilidad respecto a este parámetro, la cual ya habíamos observado y comentado en el capítulo 2.

Los resultados experimentales parecen prometedores, en el sentido de que el control de granularidad mejora las aceleraciones en la práctica, en una situación bastante desafiante. En sistemas con mayores costes de ejecución paralela, tales como sistemas distribuidos, los beneficios pueden ser mucho mayores, aunque puede que sea difícil obtener aceleraciones en algunos casos (es decir, que si estos costes son muy grandes, el resultado del sistema de control de granularidad puede ser en muchos casos un programa secuencial). En cualquier caso, creemos que es posible mejorar estos resultados significativamente utilizando estrategias de control más sofisticadas, como a las que nos referíamos antes.

3.6. Conclusiones del capítulo

Hemos desarrollado un análisis que obtiene funciones que devuelven cotas inferiores del coste de procedimientos en función de los tamaños de los datos de entrada. Hemos propuesto una mejora del tratamiento de predicados del tipo "divide y vencerás". Hemos realizado una implementación de dicho análisis de coste y la hemos integrado en el sistema `ciaopp`. Finalmente hemos realizado experimentos que muestran que el análisis desarrollado es preciso y eficiente.

A pesar de que los nombres puedan sugerir cierta similitud, nuestro trabajo es bastante diferente al realizado por Basin y Ganzinger sobre análisis de complejidad basado en resolución ordenada [BG96]. Ellos consideran la resolución basada en un orden total y bien fundado sobre átomos totalmente instanciados ("ground"), y la usan para examinar la complejidad de determinar, dado un conjunto N de cláusulas de Horn y una cláusula de Horn totalmente instanciada C, si se cumple que $N \models C$. Nuestro trabajo, en contraste, se basa en una formulación operacional de la ejecución de un programa lógico que no se restringe a consultas "ground" (o, a programas de Horn, puesto que es fácil tratar con características tales como cortes y negación por fallo). Dado que los aspectos operacionales de la ejecución de un programa se modelan de forma más precisa con nuestra técnica, los resultados obtenidos son también más precisos.

La información sobre los costes de tiempos de ejecución, además de su aplicación al control de granularidad (que es la principal motivación de esta tesis) puede ser útil para una gran variedad de aplicaciones, entre las cuales se incluyen ayudar a los sistemas de transformación de programas a elegir las transformaciones óptimas, elección entre distintos algoritmos, depuración de eficiencia (optimización) de programas y la optimización de consultas en bases de datos deductivas.

Capítulo 4

Análisis de no-fallo

En este capítulo describimos un método (estático) en el que, a partir de la información sobre modos y una aproximación segura (por arriba) de tipos de llamada, detectamos procedimientos y llamadas que no fallarán (e.d., que producirán como mínimo una solución, o no terminarán). La técnica se basa en una noción muy intuitiva: la de un conjunto de tests que "cubren" el tipo de un conjunto de variables (e.d. para cualquier elemento del tipo al menos uno de los tests tendrá éxito). Mostramos que el problema de determinar un "recubrimiento" es en general indecidible, y mostramos además resultados sobre decibilidad y complejidad para los sistemas con restricciones aritméticas lineales y sobre el dominio de Herbrand. También mostramos resultados para determinar "recubrimientos" que son precisos y eficientes en la práctica. En base a esta información, mostramos cómo se pueden identificar procedimientos y llamadas que no fallarán en tiempo de ejecución.

Entre las aplicaciones de la información de no-fallo, podemos citar la detección de errores de programación, transformación de programas, optimización de la ejecución paralela, la eliminación de paralelismo especulativo y la estimación de cotas inferiores del coste computacional de objetivos. Estas últimas pueden utilizarse para realizar control de granularidad.

Finalmente, describimos una implementación de nuestro método y mostramos que los resultados experimentales obtenidos son mejores que los de otros enfoques propuestos anteriormente.

4.1. Motivación

Existen dos importantes motivaciones para considerar análisis que, en tiempo de compilación, identifiquen procedimientos y llamadas que no fallan en programas lógicos. La primera es que, usualmente, es muy útil identificar programas que se comportan mal. Por ejemplo, en lenguajes tipados estáticamente, se espera que los componentes de los programas se usen de una forma consistente con sus tipos, y por ello se realizan comprobaciones en tiempo de compilación para detectar desviaciones del comportamiento esperado. A pesar de que esto no elimina todos los errores de programación, hace que sea más simple identificar y localizar ciertos tipos de errores de programación comunes. De la misma forma, en programas lógicos, el comportamiento esperado suele ser que un predicado tenga éxito y produzca una o más soluciones. Sin embargo, en la mayoría de los sistemas de programación lógica, la única comprobación que se suele hacer es más bien simple— aunque útil—y consiste en la detección de variables que aparecen sólo una vez en una cláusula.

La segunda razón es que la información de no-fallo puede usarse en numerosas transformaciones y optimizaciones de programas. Por ejemplo, es deseable ejecutar objetivos que pueden fallar lo antes posible (principio conocido como "first-fail"); y en sistemas paralelos, la información de no-fallo se puede utilizar para eliminar el paralelismo especulativo y para estimar cotas inferiores del coste de procedimientos [DLGHL94, DLGHL97], las cuales a su vez se pueden utilizar para realizar control de granularidad de tareas paralelas [LGHD96].

El problema existente con intentos simples para inferir no-fallo, es que en general siempre es posible que un objetivo falle porque ciertos argumentos "erróneos" provocan el fallo de la unificación con la cabeza. Una solución obvia podría ser probar y descartar tales valores de los argumentos considerando el tipo de los predicados. Sin embargo, la mayoría de los análisis de tipos existentes proporcionan aproximaciones "por arriba", en el sentido de que calculan un superconjunto del conjunto de valores para los argumentos que ocurren en tiempo de ejecución. Desafortunadamente, los intentos para solucionar este problema, por ejemplo infiriendo aproximaciones "por debajo" de los tipos de llamada a los procedimientos obtienen cotas inferiores triviales en la mayoría de los casos.

4.2. Preliminares

Para razonar sobre no-fallo, necesitamos distinguir entre operaciones de unificación que actúan como tests (y que por tanto pueden fallar) y unificaciones de "salida" que actúan como asignaciones (las cuales siempre tienen éxito). Suponemos que los programas son *modados*, e.d. para cada unificación en cada predicado, sabemos cuándo la operación actúa como un test y cuándo realiza una ligadura de salida (nótese que ésta es una noción más débil que la mayoría de las nociones convencionales, en el sentido de que no requiere que los argumentos de entrada estén totalmente instanciados (e.d. que sean "ground"). Cuando sea necesario hacer énfasis en los tests de entrada de una cláusula, escribiremos la misma con guardas (guarded form) como

$$p(x_1, \ldots, x_n) :- \mathit{input\_tests}(x_1, \ldots, x_n) \parallel \mathit{Body}.$$

Considérese el predicado definido por las cláusulas:

$$abs(X, Y) :- X \geq 0 \parallel Y = X.$$
$$abs(Y, Z) :- Y < 0 \parallel Z = -Y.$$

Supongamos que sabemos que este predicado siempre se llamará con su primer argumento instanciado a un número entero. Obviamente, para una llamada particular, uno de los dos tests '$X \geq 0$' o '$X < 0$' podrá fallar; sin embargo, si los consideramos de forma conjunta, al menos uno de los dos tendrá éxito. Esto nos muestra que no podemos examinar los tests de cada cláusula por separado sino que es necesario seleccionarlos y examinarlos conjuntamente. Al seleccionar los tests conjuntamente, tenemos que tener cuidado para no confundirnos con los nombres diferentes que pueden tomar las variables en cláusulas diferentes. Por ejemplo, en el predicado *abs* definido anteriormente, necesitamos asegurar que:

1. nos damos cuenta que la variable X de la primera cláusula y la variable Y de la segunda se refieren en realidad al mismo componente de los argumentos del predicado; y

2. no confundimos la variable Y de la primera cláusula con la variable Y de la segunda cláusula.

Estos inconvenientes pueden evitarse usualmente normalizando las cláusulas, de modo que se utilizan los nombres de variables de forma consistente y según una convención predeterminada. El enfoque que tomamos es el usual, que consiste en usar secuencias de enteros para codificar caminos en árboles ordenados: la secuencia vacía ε corresponde al nodo de la raíz del árbol, y si la secuencia π corresponde a un nodo N, entonces las secuencias $\pi 1, \ldots, \pi k$ corresponden a sus hijos $N_1, \ldots, N_k$ tomados en orden. Adoptamos la convención de que las variables en una cláusula se designan como X_π, donde π es el camino desde la raíz de la cláusula, etiquetada `:-`/2, a la ocurrencia más a la izquierda de esa variable. Para aumentar la legibilidad, en este capítulo no utilizamos los nombres según la convención descrita anteriormente a menos que sea necesario, aunque en realidad ha de suponerse que las cláusulas están normalizadas.

4.3. Tipos, tests y recubrimientos

Un tipo representa un conjunto de términos, y se puede denotar utilizando varias representaciones—por ejemplo *términos tipados* (*type terms*) y *gramáticas de términos regulares* (*regular term grammars*), como en [DZ92], o *grafos de tipo* (*type graphs*) como en [JB92]. Supongamos que $\mathsf{type}[p]$ denota el tipo de cada predicado p en un programa dado. En este capítulo, nos interesan exclusivamente los "tipos de llamada" para los predicados—en otras palabras, cuando decimos que "*un predicado p en un programa P tiene tipo* $\mathsf{type}[p]$", queremos decir que en cualquier ejecución del programa P para alguna clase de llamada de interés, siempre que haya una llamada $p(\bar{t})$ al predicado p, la tupla de argumentos $\bar{t}$ en la llamada será un elemento del conjunto denotado por $\mathsf{type}[p]$. El análisis de no-fallo que describimos se basa en *tipos regulares* [DZ92], que se especifican con *gramáticas de términos regulares* en las cuales cada *símbolo de tipo* (*type symbol*) está definido por una única *regla de tipo* (*type rule*).

Un tratamiento más detallado de estos temas puede verse en varios artículos sobre análisis de tipos, por ejemplo en [DZ92, JB92]. Por brevedad, no comentaremos más sobre ello en este trabajo. Denotamos el Universo de Herbrand (e.d., el conjunto de todos los términos "ground") como $\mathcal{H}$, y el conjunto de n–tuplas de elementos de $\mathcal{H}$ como $\mathcal{H}^n$.

Dado un conjunto (finito) de variables V, una *asignación de tipos sobre* V es una correspondencia desde V a un conjunto de tipos. Escribiremos una asignación de tipos ρ sobre un conjunto de variables $\{x_1, \dots, x_n\}$, como $(x_1 : a_1, \dots, x_n : a_n)$, donde $\rho(x_i) = a_i$, para $1 \leq i \leq n$, y a_i es una representación de tipo. Alternativamente, también utilizaremos la notación $\bar{x} : \bar{T}$, donde $\bar{x}$ es la tupla de variables $(x_1, \dots, x_n)$, y $\bar{T}$ es la tupla de tipos $(a_1, \dots, a_n)$. Dado un término t y una representación de tipo T, abusaremos de terminología y diremos que $t \in T$, para expresar que t pertenece al conjunto de términos denotado por

T.

Un *test primitivo* es un "átomo" cuyo predicado está predefinido, que actúa como un "test". Ejemplos de algunos de estos predicados predefinidos son la unificación, o algún predicado aritmético ($<, >, \leq, \geq, \neq$, etc.). Definimos un *test* como un test primitivo, o una conjunción $\tau_1 \wedge \tau_2$, o una disyunción $\tau_1 \vee \tau_2$, o una negación $\neg\tau_1$, donde τ_1 y τ_2 son tests.

La noción de un test que "cubre" una asignación de tipos es fundamental para nuestra técnica de detección de no-fallo:

Definición 4.3.1 Un test $S(\bar{x})$ *cubre* una asignación de tipos $\bar{x} : \bar{T}$, donde $\bar{x}$ y $\bar{T}$ son, respectivamente, una tupla de variables y tipos no vacíos, si para todo $\bar{t} \in \bar{T}$ se cumple que $\bar{x} = \bar{t} \models S(\bar{x})$. ∎

Consideremos un predicado p definido por n cláusulas, con tests de entrada $\tau_1, \ldots, \tau_n$:

$$\begin{array}{l} p(\bar{x}) :- \ \tau_1(\bar{x}) \ [\!] \ Body_1. \\ \vdots \\ p(\bar{x}) :- \ \tau_n(\bar{x}) \ [\!] \ Body_n. \end{array}$$

Nos referiremos al test $\tau(\bar{x}) \equiv \tau_1(\bar{x}) \vee \cdots \vee \tau_n(\bar{x})$ como al *test de entrada de* p. Supongamos que el predicado p tiene tipo $\mathsf{type}[p]$: por simplicidad, a veces abusamos de terminología y decimos que el predicado p cubre el tipo $\mathsf{type}[p]$ si el test de entrada $\tau(\bar{x})$ de p cubre la asignación de tipos $\bar{x} : \mathsf{type}[p]$.

Definimos la relación "llama" entre los predicados de un programa de la forma siguiente: p llama q, denotado $p \leadsto q$, si y sólo si un literal con símbolo de predicado q aparece en el cuerpo de una cláusula que define p, y usamos $\leadsto^\star$ para denotar el cierre reflexivo transitivo de $\leadsto$. La importancia de la noción de "recubrimiento" viene expresada por el siguiente resultado:

Teorema 4.3.1 *Un predicado p de un programa no falla si para cada predicado q tal que $p \leadsto^\star q$, se cumple que q cubre* $\mathsf{type}[q]$.

Demostración Supongamos que p puede fallar, e.d., hay un objetivo $p(\bar{t})$, tal que $\bar{t} \in \mathsf{type}[p]$, que falla. Podemos demostrar fácilmente mediante inducción sobre el número de pasos de resolución, que existe un q tal que $p \leadsto^{\star} q$ y q no cubre su tipo. ∎

Nótese que el no fallo no implica el éxito: un predicado que no falla puede que nunca produzca una solución porque no termine. Como ejemplo, consideremos el siguiente predicado, definido por una sola cláusula, que no falla y que —en la mayoría de los sistemas Prolog existentes—no termina:

$p(X) :- p(X).$

Idealmente, nos gustaría disponer de un procedimiento para decidir si un test cubre una asignación de tipos dada. Desafortunadamente, esto es imposible en general, como mostramos con el resultado siguiente:

Teorema 4.3.2 *Dado un test arbitrario y una asignación de tipos, en general es indecidible determinar si el test cubre la asignación de tipos.*

Demostración La demostración es directa a partir de un resultado, dado por Matijasevič, que demuestra que es indecidible determinar la existencia de soluciones (enteras) para ecuaciones Diofantinas arbitrarias [Mat70]. Dado un polinomio arbitrario $\phi(x_1, \ldots, x_n)$, consideremos el test $\phi(x_1, \ldots, x_n) \neq 0$. Este test cubre la asignación de tipos $(x_1 : \mathtt{integer}, \ldots, x_n : \mathtt{integer})$ si y sólo si cada asignación posible de enteros a las variables $x_1, \ldots, x_n$ hace que el polinomio ϕ tome un valor distinto de cero, e.d., si y sólo si la ecuación Diofantina $\phi(x_1, \ldots, x_n) = 0$ no tiene soluciones enteras. Pero dado que el problema de determinar la existencia de raíces enteras para una ecuación Diofantina arbitraria es indecidible, deducimos que el problema de determinar si un test arbitrario cubre una asignación de tipos arbitraria también es indecidible. ∎

Por tanto, nos vemos forzados a recurrir a algoritmos correctos (pero, necesariamente, incompletos) para determinar recubrimientos. En el resto de esta sección mostramos que los problemas de recubrimiento son decidibles para la mayoría de los casos que aparecen en la práctica—en particular, para tests de igualdad y desigualdad sobre el dominio de Herbrand y para tests aritméticos lineales—y damos algoritmos para determinar recubrimientos para estos casos. El enfoque que damos al problema de determinar si un test cubre una asignación de tipos consiste en:

1. particionar el test de forma que los tests que quedan en cada partición resultante son sistemas diferentes de restricciones, y

2. aplicar un algoritmo de recubrimiento particular del sistema de restricciones correspondiente.

En este capítulo consideramos dos sistemas de restricciones comúnmente encontrados:

- tests de igualdad y desigualdad sobre variables con tipos *regulares distributivos de tupla* (*tuple-distributive regular types*) [DZ92], es decir, tipos que se pueden especificar por gramáticas de términos regulares en las que cada símbolo de tipo tiene exactamente una regla de tipo que lo define y cada regla de tipo es *determinista*; y

- tests aritméticos lineales sobre variables enteras.

4.3.1. Recubrimientos en el dominio de Herbrand

Decidibilidad y complejidad

Mientras que la determinación de recubrimientos es indecidible para tests aritméticos arbitrarios, el problema resulta decidible si nos restringimos a ecuaciones e inecuaciones sobre términos de Herbrand. Antes de discutir el algoritmo para este caso, damos un resultado sobre la complejidad del problema de recubrimiento en el dominio de Herbrand:

Teorema 4.3.3 *El problema de recubrimiento en el dominio de Herbrand es "co-NP-hard". Incluso sigue siendo "co-NP-hard" si nos restringimos sólo a tests de igualdad.*

Demostración Por reducción al problema de determinar si una fórmula proposicional en forma normal disyuntiva que contiene como mucho 3 literales en cada disyunción es una tautología ([GJ79], problema LO8). ∎

Un procedimiento de decisión

El procedimiento de decisión que presentamos aquí está inspirado por un resultado dado por Kunen [Kun87], que consiste en que el problema de determinar si un conjunto de términos es vacío es decidible para combinaciones Booleanas de (notaciones para) ciertos subconjuntos "básicos" del Universo de Herbrand de un programa. También utilizamos adaptaciones directas de algunas operaciones descritas por Dart y Zobel [DZ92].

La razón por la cual el algoritmo de recubrimiento es tan complejo es que queremos un algoritmo *completo* para tests de igualdad y desigualdad. Es posible simplificar el algoritmo considerablemente si sólo nos interesan los tests de igualdad. Antes de describir el algoritmo necesitamos presentar algunas definiciones y notaciones.

Utilizamos las nociones (que serán definidas a continuación) de *término anotado con tipos* (*type-annotated term*), y en general la de *conjunto elemental* (*elementary set*), como representaciones que denotan algunos subconjuntos de $\mathcal{H}^n$ (para algún $n \geq 1$). Dada una representación S (conjunto elemental o término anotado con tipos), $Den(S)$ se refiere al subconjunto de $\mathcal{H}^n$ denotado por S.

Definición 4.3.2 [Término anotado con tipos] Un *término anotado con tipos* es un par $M = (\bar{t}, \rho)$, donde $\bar{t}$ es una tupla términos, y ρ es una asignación de tipos $(x_1 : T_1, \ldots, x_k : T_k)$. Para hacer el texto más legible, escribiremos el tipo de x_i en M, es decir $\rho(x_i)$, como $type(x_i, M)$ o como $type(x_i, \rho)$. Además, dado un término anotado con tipos M, denotamos su tupla de términos y su asignación de tipos como $\bar{t}_M$ y ρ_M respectivamente. Un término anotado con tipos denota el conjunto de todos los términos "ground" $\theta(\bar{t})$ tal que $\theta(x) \in type(\bar{x}, \rho)$ para cada variable de $\bar{t}$. ∎

Dado un término anotado con tipos $(\bar{t}, \rho)$, la tupla de términos $\bar{t}$ puede verse como un *término tipado* (*type term*) y ρ puede considerarse como una *sustitución tipada* (*type substitution*). Esto resulta útil para utilizar un algoritmo descrito por Dart y Zobel [DZ92] para calcular la "intersección" y la "inclusión" de términos anotados con tipos, las cuales definiremos más adelante. Denotemos con $\top$ el tipo de todo el Universo Herbrand. Cuando tenemos un término anotado con tipos $(\bar{t}, \rho)$ tal que $\rho(x) = \top$ para toda variable x de $\bar{t}$, por brevedad omitimos la asignación de tipos ρ y usamos sólo la tupla de términos $\bar{t}$. Por tanto, una tupla de términos $\bar{t}$ que no tenga asociada ninguna asignación de tipos se puede ver como un término anotado con tipos que denota el conjunto de todas las instancias "ground" de $\bar{t}$.

Definición 4.3.3 [Conjunto elemental] Definimos un conjunto elemental de la forma siguiente:

- Λ es un conjunto elemental, y denota el conjunto vacío (e.d., $Den(\Lambda) = \emptyset$);
- un término anotado con tipos $(\bar{t}, \rho)$ es un conjunto elemental; y
- si A y B son conjuntos elementales tales que $Den(A), Den(B) \subseteq \mathcal{H}^n$, entonces $A \otimes B$, $A \oplus B$ y $comp(A)$ son conjuntos elementales que denotan, respectivamente, los conjuntos de (tuplas de) términos $Den(A) \cap Den(B)$, $Den(A) \cup Den(B)$, y $\mathcal{H}^n \setminus Den(A)$. ∎

Definimos las siguientes relaciones entre conjuntos elementales:

- $A \sqsubseteq B$ si y sólo si $Den(A) \subseteq Den(B)$.
- $A \sqsubset B$ si y sólo si $Den(A) \subset Den(B)$.

- $A \simeq B$ si y sólo si $Den(A) = Den(B)$.

Definición 4.3.4 [Conjunto cobásico] Un conjunto cobásico es un conjunto elemental de la forma $comp(B)$, donde B es una tupla de términos (recuérdese que una tupla de términos $\bar{t}$, es en realidad un término anotado con tipos $(\bar{t}, \rho)$ tal que $\rho(x) = \top$ para toda variable x de $\bar{t}$). ∎

Definición 4.3.5 [Minset] Un *minset* es: Λ, o un conjunto elemental de la forma $X \otimes comp(Y_1) \otimes \cdots \otimes comp(Y_p)$, para algún $p \geq 0$, donde X es una tupla de términos, $comp(Y_1), \ldots, comp(Y_p)$ son conjuntos cobásicos, y para todo i tal que $1 \leq i \leq p$, se cumple que $Y_i = X\theta_i$ y $X \not\sqsubseteq Y_i$ para alguna sustitución θ_i (e.d. $Y_i \sqsubset X$ y $X \not\sqsubseteq Y_i$). ∎

Por brevedad, y de cara a la definición de algunos algoritmos, escribimos un minset de la forma $X \otimes comp(Y_1) \otimes \cdots \otimes comp(Y_p)$ como X/C, donde $C = \{comp(Y_1), \ldots, comp(Y_p)\}$. También denotaremos la tupla de términos de un conjunto cobásico $Cob \equiv comp(Y)$ como $tt(Cob)$, es decir $tt(Cob) \equiv Y$.

Definición 4.3.6 [Instancia de un término anotado con tipos] Sean α y β dos términos anotados con tipos. Decimos que α es una instancia de β si $Den(\alpha) \subseteq Den(\beta)$ y existe una sustitución θ tal que $\bar{t}_\alpha = \bar{t}_\beta\theta$. ∎

Consideremos un predicado p definido por n cláusulas, con tests de entrada $\tau_1(\bar{x}), \ldots, \tau_n(\bar{x})$: Supongamos que el predicado p tiene tipo $\mathsf{type}[p]$. Comprobar si el test de entrada de p, $\tau(\bar{x})$, cubre la asignación de tipos $\bar{x} : \mathsf{type}[p]$ se puede reducir a comprobar si:

$$M \sqsubseteq S_1 \oplus \cdots \oplus S_n \tag{4.1}$$

donde M es un término anotado con tipos que es una representación de $\bar{x} : \mathsf{type}[p]$, y cada S_i es un minset, que es la representación de $\tau_i(\bar{x})$. A continuación describimos cómo se puede transformar $\tau_i(\bar{x})$ en un minset que denotamos S_i:

1. Supongamos que el test $\tau_i(\bar{x})$ es de la forma $E_i \wedge D_i^1 \wedge \cdots \wedge D_i^m$, donde E_i es la conjunción de todos los tests de unificación de $\tau_i(\bar{x})$ (e.d., un sistema de ecuaciones) y cada D_i^j es un test de unificación negado ("disunification test", e.d., una inecuación).

2. Sea θ_i la sustitución asociada con la *forma resuelta* ("*solved form*") de E_i (esto se puede calcular usando las técnicas de Lassez y otros [LMM88]).

3. Sea θ_i^j, para $1 \leq j \leq m$, la sustitución asociada con la forma resuelta de $E_i \wedge N_i^j$, en donde N_i^j es la negación de D_i^j.

4. $S_i = B_i \otimes \mathit{comp}(B_i^1) \otimes \cdots \otimes \mathit{comp}(B_i^m)$, donde $B_i = \bar{x}\theta_i$ y $B_i^j = \bar{x}\theta_i^j$, para $1 \leq j \leq m$.

Considerando de nuevo la condición 4.1, tenemos que:

$$M \sqsubseteq S_1 \oplus \cdots \oplus S_m \text{ si y sólo si } M \otimes \mathit{comp}(S_1) \otimes \cdots \otimes \mathit{comp}(S_m) \simeq \Lambda$$

Podemos escribir $\mathit{comp}(S_1) \otimes \cdots \otimes \mathit{comp}(S_m)$ en forma normal disyuntiva como $M_1 \oplus \cdots \oplus M_u$, donde cada M_i es un minset [1]. Dado que:

$$M \otimes (M_1 \oplus \cdots \oplus M_u) \simeq (M \otimes M_1) \oplus \cdots \oplus (M \otimes M_u)$$

tenemos que:

$$M \sqsubseteq S_1 \oplus \cdots \oplus S_m \text{ si y sólo si } M \otimes M_i \simeq \Lambda \text{ para todo } i,\ 1 \leq i \leq u$$

Por tanto, el problema fundamental es encontrar un algoritmo para comprobar si $M \otimes S \simeq \Lambda$, donde M es un término anotado con tipos y S un minset. El algoritmo que proponemos viene dado por la función booleana $\mathit{empty}(M, S)$, definida en la figura 4.1[2].

[1] Nótese que $\oplus$, $\otimes$, y *comp* constituyen un álgebra de Boole, y la operación $\otimes$ es computable para términos anotados con tipos.

[2] Usamos la representación de tipos de [DZ92], y suponemos que hay un conjunto común de reglas de tipo que describen los símbolos de tipo. Por brevedad, omitimos dicho conjunto de reglas en la descripción de los algoritmos.

$\boxed{empty(M, S):}$

Entrada: un término anotado con tipos M y un minset S ($S = A/C$, donde A es una tupla de términos, y C un conjunto de conjuntos cobásicos).

Salida: un valor booleano que denota si $M \otimes S \simeq \Lambda$.

Proceso:

1. si $S \equiv \Lambda$ entonces return(**true**), en otro caso, sea $R = intersection(M, A)$;
2. si $R \equiv \Lambda$ entonces return(**true**);
3. en otro caso, si $included(A, R)$ entonces return(**false**) en otro caso return($empty1(C, R, \emptyset)$).

Figura 4.1: Definición de la función *empty*.

En primer lugar, se realiza la "intersección" de M y la tupla de términos del minset S (suponemos que $S = A/C$). Esta intersección la implementa la función $intersection(M, A)$, la cual devuelve $M \otimes A$ (téngase en cuenta que una tupla de términos es un término anotado con tipos), y que es una adaptación directa de la función $unify(\tau_1, \tau_2, T, \Theta)$ descrita en [DZ92], la cual realiza una *unificación de tipos* donde τ_1 y τ_2 son términos tipados, Θ una sustitución tipada para las variables que aparecen en τ_1 y τ_2, y T un conjunto de reglas de tipo que definen τ_1, τ_2, y Θ. Entonces, si la intersección mencionada (a la que llamamos R) no es vacía, ni A (recuérdese que $S = A/C$) está "incluido" en R, llama a $empty1(C, R, \emptyset)$, definida en la figura 4.2, la cual comprueba si $R/C \simeq \Lambda$. Esto se hace comprobando si R está "incluido" en alguna tupla de términos de algún conjunto cobásico de C (en cuyo caso $R/C \simeq \Lambda$). Para este propósito, utiliza la función $included(R, \bar{t})$, donde $\bar{t}$ es la tupla de términos de un conjunto cobásico Co de C (e.d. $\bar{t} \equiv tt(Co)$). Dicha función es una adaptación directa de la función $subset_T(\tau_1, \tau_2)$ descrita en [DZ92], que determina si el tipo denotado por un término tipado puro (pure type term), es decir, un término Herbrand que no tiene ningún símbolo de tipo, es un subconjunto del tipo denotado por otro (e.d., $included(R, \bar{t})$ devuelve **true** si y sólo si $R \sqsubseteq \bar{t}$). Nótese que R/C se puede ver como un sistema de ecuaciones (correspondiente a R) y cero o más inecuaciones (las cuales proceden de los conjuntos cobásicos de C). Por tanto, el problema se puede ver como la determinación de si tal sistema tiene o no soluciones. En el paso 1 de la función *empty1*, se eliminan conjuntos cobásicos que son "inútiles". Decimos que un conjunto cobásico Cob es "inútil" (para determinar la insatisfacibilidad del sistema) si cuando se cumple que $R/(C - \{Cob\}) \not\simeq \Lambda$, entonces se cumple que $R/C \not\simeq \Lambda$. Cualquier conjunto cobásico inútil Cob se puede eliminar de C, ya que $R/(C - \{Cob\}) \simeq \Lambda$ si y sólo si $R/C \simeq \Lambda$ (nótese que si $R/(C - \{Cob\}) \simeq \Lambda$, entonces obviamente $R/C \simeq \Lambda$). Si la tupla de términos de un conjunto cobásico Cob de C es "disjunta" con R, entonces Cob es inútil (sin embargo, puede haber conjuntos cobásicos inútiles en C cuyas tuplas de términos no sean disjuntas con R). Si después de esta eliminación queda algún conjunto cobásico útil (e.d. no inútil), entonces vamos al paso 3. En este paso, si R no está "incluido" en ninguna de las tuplas de términos de los conjuntos cobásicos de C, entonces esto significa que R es "demasiado grande", y por tanto, se "expande" en varios términos anotados con tipos "más pequeños" (con la esperanza de que cada uno de ellos esté "incluido" en alguna tupla de términos de algún conjunto cobásico de C). Esto se hace en el paso 4, en donde se selecciona un conjunto cobásico Cob de C'', y se "expande" R utilizando la función $expansion(R, Cob)$, la cual toma como entrada un término anotado con tipos R y un conjunto cobásico Cob y devuelve un par $(R', Rest)$ (que es una "partición" de R) tal que:

- R' es un término anotado con tipos;
- $Rest$ es un conjunto de términos anotados con tipos;
- para todo $x \in vars(R')$, se cumple que $type(x, R') = \top$, o $x\theta$ es una variable,

donde $\theta = mgu(\bar{t}_{R'}, tt(Cob))$, y la función $mgu(A, B)$ devuelve un unificador de máxima generalidad (idempotente) de las tuplas de términos A y B;

- $(\cup_{X \in Rest} Den(X)) \cup Den(R') = Den(R)$; y
- para todo $X \in Rest$, $X \otimes tt(Cob) \simeq \Lambda$.

R' es una instancia de R obtenida mediante la expansión de R hasta una "profundidad de decisión". Esta profundidad permite detectar si el conjunto cobásico Cob es inútil.

Por ejemplo, supongamos que $R = ((X, Y), (X : list, Y : list))$ y $C = \{C_1, C_2\}$, donde $C_1 = comp([H|L], Z)$ y $C_2 = comp([\,], Z)$. R no está incluido en ninguna de la tuplas de términos de los conjuntos cobásicos de C, pero si expandimos R usando C_1, e.d., $(R_1, \{R_2\}) = expansion(R, C_1)$, donde $R_1 = (([H1|L1], Y1), (H1 : \top, L1 : list, Y1 : list))$ y $R_2 = (([\,], Y2), (Y2 : list))$, tenemos que R_1 y R_2 están incluidos en $tt(C_1)$ y $tt(C_2)$ respectivamente. Sin embargo, en otras situaciones, el problema no se puede resolver expandiendo R: supongamos, por ejemplo, que ahora $C = \{comp(Z, Z)\}$, en este caso, R no está incluido en (Z, Z) porque esta tupla de términos impone una restricción de igualdad en $Den(R)$ (nótese que aquí R ya está expandido hasta la "profundidad de decisión", donde las restricciones de igualdad vienen dadas por las variables "sinónimas"). En el paso 6, estas variables "sinónimas" se calculan usando la función $aliased(R, \bar{t})$, donde $\bar{t} = tt(Co)$ y Co es un conjunto cobásico. Dicha función toma una tupla de términos $\bar{t}$ y un término anotado con tipos R tal que para todo $x \in vars(R)$, $x\theta$ es una variable, donde $\theta = mgu(\bar{t}_R, \bar{t})$, o $type(x, R) = \top$, y devuelve un conjunto de variables $AlVars$ tal que $v \in AlVars$ si y sólo si $v \in vars(R)$ y existe $v' \in vars(R)$ tal que $v\theta \equiv v'\theta$. Si para algún $x \in vars(R')$, se cumple que $type(x, R') = \top$ y, o bien $x \in AVars$, o bien $x\theta'$ no es una variable, donde $\theta' = mgu(\bar{t}_{R'}, \bar{t})$, entonces podemos decir que Cob es inútil. Esto se puede demostrar usando la variable x para construir una instancia S tal que: suponiendo que existe una instancia I de R, tal que $I \otimes\ tt(C_1) \simeq \Lambda$ para todo $C_1 \in Cset$, donde $Cset = C' \cup \{CS \mid (B, A, CS) \in AL\}$, entonces, S se puede construir a partir de I de forma que $S \otimes\ tt(C_2) \simeq \Lambda$ para todo $C_2 \in \{Cob\} \cup Cset$.

La función *empty1*(C, R, AL), definida en la figura 4.2, realiza una "primera vuelta" por los conjuntos cobásicos en C. Después de esta vuelta (cuyo final se detecta en el paso 2 por la condición $C'' \equiv \emptyset$), los conjuntos cobásicos que se han determinado como inútiles se han eliminado, y el resto de ellos están en AL, el cual es un parámetro de acumulación. En el paso 7, R' y $AVars$ (además de Cob) se almacenan en este parámetro, dado que las variables "sinónimas" cuyo tipo es infinito (o que después de haberse expandido resultan ligadas a un término que contiene variables cuyo tipo es infinito) nos permiten detectar conjuntos cobásicos inútiles (los cuales se eliminan antes de que se llame a *empty2*(AL', R) — definida en la figura 4.3 — en el paso 2).

La función *empty2*(AL, R), definida en la figura 4.3, selecciona un conjunto cobásico Cob en AL y si R no está incluido en $tt(Cob)$, entonces R se expande como un conjunto de términos anotados con tipos $R_1, \ldots, R_n$ expandiendo solamente "variables decisivas". Esto asegura que cada R_i está o bien "incluido" en $tt(Cob)$ o bien es "disjunto" con $tt(Cob)$. También asegura que R no se expande de forma infinita (nótese que el tipo de dichas variables es finito).

Ejemplo 4.3.1 Consideremos el predicado `reverse/2`:

```
reverse(X,Y)  :- X = []      ‖ Y = [].
reverse(X,Y)  :- X = [X1|X2] ‖ reverse(X2,Y2), append(Y2,[X1],Y).
```

y la asignación de tipos $\rho \equiv (X : list)$, donde $list ::= [\,] \mid [\top|list]$. Sea τ el test de entrada del predicado `reverse/2`, e.d., $\tau \equiv X = [\,] \vee X = [X1|X2]$. Sea M el término anotado con tipos que es una representación de ρ, e.d., $M = ((X), (X :$

$\boxed{\textit{empty1}(C, R, AL):}$

Entrada: un término anotado con tipos R, un conjunto de conjuntos cobásicos C y, un conjunto AL de tuplas de la forma (B, AV, CS) donde:

- B es un término anotado con tipos.
- CS es un conjunto cobásico.
- Para todo $x \in vars(B)$, $x\theta$ (donde $\theta = mgu(\bar{t}_B, tt(CS))$) es una variable.
- $v \in AV$ si y sólo si $v \in vars(B)$ y existe $v' \in vars(B)$ tal que $v\theta \equiv v'\theta$. Es decir, AV es el conjunto de variables en $vars(B)$ que son "sinónimas" (aliased) con alguna otra variable en $vars(B)$ según θ.

Salida: un valor booleano que denota si $R/C_1 \simeq \Lambda$, donde $C_1 = C \cup \{Cob \mid (B, A, Cob) \in AL$, para algún B y $A\}$.

Proceso:

1. Sea $C'' = \{Cob \in C \mid \ \textit{intersection}(R, tt(Cob)) \not\simeq \Lambda\}$;
2. si $C'' \equiv \emptyset$ entonces return($\textit{empty2}(AL', R)$), donde $AL' = \{(S, AVars, Cob) \mid (S, AVars, Cob) \in AL, \textit{intersection}(R, tt(Cob)) \not\simeq \Lambda, \theta = mgu(\bar{t}_S, \bar{t}_R)$, y para todo x, y tal que $x \in AVars$ e $y \in vars(x\theta)$, $\textit{type}(y, R)$ es finito$\}$. Para esto existen algoritmos simples que comprueban si una expresión de tipo denota un conjunto finito o infinito de términos.
3. En otro caso, si $\textit{included}(R, tt(Co))$ para algún conjunto cobásico Co en C'' entonces return(**true**);
4. En otro caso, tomamos un conjunto cobásico Cob de C''.
   Sean $C' = C'' - \{Cob\}$ y $(R', Rest) = \textit{expansion}(R, Cob)$;
5. si $\textit{included}(R', tt(Cob))$ entonces return($\bigwedge_{X \in Rest} \textit{empty1}(C', X, AL)$);
6. en otro caso, sea $AVars = \textit{aliased}(R', tt(Cob))$. Si para algún $x \in vars(R')$, se cumple que $\textit{type}(x, R') = \top$ y, bien $x \in AVars$, o bien $x\theta'$ no es una variable, donde $\theta' = mgu(\bar{t}_{R'}, tt(Cob))$, entonces return($\textit{empty1}(C', R, AL)$);
7. En otro caso, sea $AL' = AL \cup \{(R', AVars, Cob)\}$;
8. return($\textit{empty1}(C', R', AL') \wedge (\bigwedge_{X \in Rest} \textit{empty1}(C', X, AL))$);

Figura 4.2: Definición de la función *empty1*.

$\boxed{empty2(AL, R):}$

1. Si $AL \equiv \emptyset$ entonces return(**false**); en otro caso, coger un elemento $A \in AL$. Supongamos que $A \equiv (B, AV, Cob)$, y sea $AL' = AL - \{A\}$ y $\theta = mgu(\bar{t}_B, \bar{t}_R)$;
2. si $included(R, tt(Cob))$ entonces return(**true**), en otro caso, para todas las variables $y \in AV$, expandir todas las variables x tal que $x \in vars(y\theta)$ (necesariamente $x \in vars(R)$ y $type(x, R)$ es finito). Sea RS el conjunto de términos anotados con tipos resultante de estas expansiones.
3. Sea $RS' = \{r \in RS \mid intersection(r, Cob) \simeq \Lambda\}$ (necesariamente para todo $s \in RS$ y $s \notin RS'$, $s \sqsubseteq tt(Cob)$);
4. si $RS' = \emptyset$ entonces return(**true**), en otro caso return($\bigwedge_{X \in RS'} empty2(AL', X)$).

Figura 4.3: Definición de la función *empty2*.

$list$)). Las representaciones mediante minsets de $X = [\,]$ y $X = [X1|X2]$ son $([\,])$ y $([X1|X2])$ respectivamente (en este ejemplo utilizamos tuplas unarias).

Tenemos que τ cubre ρ si y sólo si $((X), (X : list)) \sqsubseteq ([]) \oplus ([X1|X2])$ si y sólo si $((X), (X : list)) \otimes comp(([\,]) \oplus ([X1|X2])) \simeq \Lambda$ si y sólo si $((X), (X : list)) \otimes comp(([\,])) \otimes comp(([X1|X2])) \simeq \Lambda$. La forma normal disyuntiva de $comp(([\,])) \otimes comp(([X1|X2]))$ es $(X3) \otimes comp(([\,])) \otimes comp(([X1|X2]))$, la cual sólo tiene un minset. Ahora, tenemos que probar que $((X), (X : list)) \otimes (X3) \otimes comp(([\,])) \otimes comp(([X1|X2])) \simeq \Lambda$, e.d., si la llamada $empty(M, S)$, donde $M = (\bar{t}_M, \rho_M)$, $\bar{t}_M \equiv (X)$, $\rho_M \equiv (X : list)$, y $S \equiv (X3)/\{comp([\,]), comp([X1|X2])\}$ devuelve **true**. Esta llamada hace lo siguiente (y de hecho devuelve **true**):

1. $intersection(M, (X3))$ devuelve el término anotado con tipos $((X4), (X4 : list))$.

2. Ya que esta intersección no es "vacía" y $(X3)$, que representa el término anotado con tipos $((X3),(X3:\top))$, no está "incluida" en $((X4),(X4:\mathit{list}))$, se realiza la llamada

$$\mathit{empty1}\,(\{\mathit{comp}([\,]),\mathit{comp}([X1|X2])\},((X4),(X4:\mathit{list})),\emptyset)$$

Esta llamada devuelve **true** y hace lo siguiente:

   a) Tenemos que $((X4),(X4 : \mathit{list}))$ no está "incluido" en ninguna de las tuplas de términos de los conjuntos cobásicos en $\{\mathit{comp}([\ ]),\mathit{comp}([X1|X2])\}$. Por tanto, se selecciona un conjunto cobásico de este conjunto. Supongamos que $\mathit{comp}([X1|X2])$ es el conjunto cobásico seleccionado;

   b) $(R',\mathit{Rest}) = \mathit{expansion}(((X4),(X4 : \mathit{list})),\mathit{comp}([X1|X2]))$, donde $R' = (([X5|X6]),(X5 : \top, X6 : \mathit{list}))$, y $\mathit{Rest} = \{(([\ ]),\emptyset)\}$ ($\emptyset$ denota una asignación de tipos vacía, y $([\ ])$ no tiene variables).

   c) La llamada $\mathit{included}(R',([X1|X2]))$ devuelve **true**, y por tanto se realiza la llamada $\mathit{empty1}\,(\{\mathit{comp}([\,])\},(([\,]),\emptyset),\emptyset)$. Dicha llamada también devuelve **true**, puesto que $(([\,]),\emptyset) \sqsubseteq ([\,])$. Por tanto, la llamada inicial devuelve **true**. $\Box$

El algoritmo de recubrimiento que presentamos es completo para *tipos regulares distributivos de tupla* (*tuple-distributive regular types*):

Teorema 4.3.4 *Sea M un término anotado con tipos en el que todos los tipos son regulares distributivos de tupla, y S un minset. Entonces empty(M,S) termina, y devuelve* **true** *si y sólo si $M \otimes S \simeq \Lambda$.*

Aunque es correcto, el algoritmo no es completo para tipos regulares en general (aunque creemos que es bastante preciso en la práctica):

Teorema 4.3.5 *Sea M un término anotado con tipos donde todos los tipos son regulares, y S un minset. Entonces $empty(M, S)$ termina, y si devuelve* **true** *entonces $M \otimes S \simeq \Lambda$.*

Para cada uno de estos teoremas, la corrección se puede demostrar por inducción sobre la profundidad de recursión de las funciones *empty1* y *empty2* una vez que terminan; la demostración sobre su terminación se puede realizar de la forma estándar (dichas demostraciones se describen completamente en [DLGH96], y no las incluimos aquí por brevedad).

Una fuente de imprecisión del análisis en el caso de los tipos regulares que no son distributivos de tupla es que la función $intersection(M, A)$ descrita anteriormente calcula un superconjunto de la intersección exacta cuando tratamos con tipos regulares en general (este resultado se puede derivar de los trabajos sobre tipos de Dart y Zobel [DZ92]). Otra fuente de imprecisión es el uso de $expansion(R, Cob)$ para particionar el término anotado con tipos R en la función booleana $empty1(C, R, \emptyset)$. Dado un par $(R', Rest)$ donde R' es un término anotado con tipos, y $Rest$ es un conjunto de términos anotados con tipos, suponemos que todos los términos anotados con tipos en $Rest$ son disjuntos con la tupla de términos del conjunto cobásico Cob, pero esto no es cierto para tipos regulares en general, y, consecuentemente, se puede perder precisión. Una solución para obtener un algoritmo completo para tipos regulares en general consiste en reescribir el término anotado con tipos que representa el tipo de entrada de un predicado como la unión de términos anotados con tipos que contienen solamente tipos regulares distributivos de tupla, y aplicar entonces el algoritmo de recubrimiento descrito anteriormente para cada uno de los elementos de la unión.

4.3.2. Recubrimientos para aritmética lineal sobre enteros

En esta sección, consideramos tests aritméticos lineales sobre enteros (las ideas se pueden extender directamente a tests lineales sobre los reales, que es computacionalmente algo más simple). Sin pérdida de generalidad, suponemos que los tests están en forma normal disyuntiva, e.d., son de la forma $\Phi(\bar{x}) = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m} \phi_{ij}(\bar{x})$ donde cada uno de los tests $\phi_{ij}(\bar{x})$ es de la forma $\phi_{ij}(\bar{x}) \equiv a_0 + a_1x_1 + \cdots + a_kx_k \circledcirc 0$, con $\circledcirc \in \{=, \neq, <, \leq, >, \geq\}$. Determinar si $\Phi(\bar{x})$ cubre la asignación de tipos de un `entero` a cada variable en $\bar{x}$ consiste en determinar si $\models (\forall\bar{x})\Phi(\bar{x})$. Esto es cierto si y sólo si $(\exists\bar{x})\neg\Phi(\bar{x})$ es insatisfacible. En otras palabras, necesitamos determinar la insatisfacibilidad de

$$\neg\Phi(\bar{x}) \;=\; \bigwedge_{i=1}^{n}\bigvee_{j=1}^{m} \neg\phi_{ij}(\bar{x}) \;=\; \bigwedge_{i=1}^{n}\bigvee_{j=1}^{m} \psi_{ij}(\bar{x}),$$

donde $\psi_{ij}(\bar{x})$ se deriva de $\neg\phi_{ij}(\bar{x})$ de la forma siguiente: sea $\phi_{ij}(\bar{x}) = \sum_{i=0}^{k} a_ix_i \circledcirc 0$. Si $\circledcirc$ es un operador de comparación distinto de '=', $\psi_{ij}(\bar{x})$ es simplemente $\sum_{i=0}^{k} a_ix_i \overline{\circledcirc} 0$, donde $\overline{\circledcirc}$ es el operador complementario de $\circledcirc$, p.e., si $\circledcirc \equiv$ '>' entonces $\overline{\circledcirc} \equiv$ '$\leq$'. Si $\circledcirc \equiv$ '=', el correspondiente operador complementario es '$\neq$', pero esto se puede escribir en términos de dos tests que utilizan los operadores '>' y '<':

$$\psi_{ij}(\bar{x}) = (\textstyle\sum_{i=0}^{k} a_ix_i > 0) \;\vee\; (\sum_{i=0}^{k} a_ix_i < 0).$$

El sistema resultante, transformado en forma normal disyuntiva, define un conjunto de problemas de programación entera: la respuesta al problema original de recubrimiento es "si" si y sólo si ninguno de estos problemas de programación entera tiene solución. Puesto que de esta forma un test puede dar lugar como mucho a problemas finitos de programación entera, se deduce que el problema de recubrimiento para tests lineales sobre los enteros es decidible.

Ya que determinar si un problema de programación entera tiene solución es NP-completo [GJ79], el siguiente resultado de complejidad es inmediato:

Teorema 4.3.6 *El problema de recubrimiento para tests aritméticos lineales sobre los enteros es "co-NP-hard".*

Es de destacar que la gran mayoría de los tests aritméticos encontrados en la práctica tienden a ser bastante simples: nuestra experiencia es que es raro encontrar tests con más de dos variables. La resolución de problemas de programación entera en el caso en donde cada desigualdad tiene como mucho dos variables, e.d., es de la forma $ax + by \leq c$, puede decidirse eficientemente en tiempo polinomial examinando los bucles en un grafo construido a partir de las inecuaciones [AS79]. Los problemas de programación entera que aparecen en la práctica, en el contexto del análisis, son por tanto eficientemente decidibles.

4.3.3. Análisis de recubrimiento para tests mixtos

Sea τ el test de entrada del predicado p y ρ una asignación de tipos. Consideremos la asignación de tipos ρ escrita como un término anotado con tipos M, y τ escrito en forma normal disyuntiva, e.d., $\tau = \tau_1 \vee \cdots \vee \tau_n$, donde cada τ_i es una conjunción de tests primitivos (recuérdese que los tests primitivos son la unificación, desunificación, etc.). Consideremos el test τ_i escrito como $\tau_i^H \wedge \tau_i^A$, en donde τ_i^H y τ_i^A son una conjunción de tests primitivos de unificación y tests aritméticos respectivamente (e.d., escribimos los tests aritméticos después de los test de unificación). Consideremos también τ_i^H escrito como un minset D_i (recuérdese que D_i es la intersección de una tupla de términos y cero o más conjuntos cobásicos). Sea D la unión (disyunción) de estos minsets.

Ejemplo 4.3.2 Sea p el predicado `partition`/4 del programa quicksort. Sea τ el test $X = [] \vee (X = [H|L] \wedge H > Y) \vee (X = [H|L] \wedge H \leq Y)$ y sea ρ la asignación de tipos $(X : \mathit{intlist}, Y : \mathit{integer})$, en donde $\mathit{intlist} ::= [\,] \mid [\mathit{integer}|\mathit{intlist}]$. En este caso, tenemos que M es $((X, Y), (X : \mathit{intlist}, Y : \mathit{integer}))$. $\tau_1 \equiv X = [\,]$, $\tau_2 \equiv X = [H|L], H > Y$, y $\tau_3 = X = [H|L], H \leq Y$. τ_1 se puede escribir como $\tau_1^H \wedge \tau_1^A$, donde $\tau_1^H \equiv X = [\,]$ y $\tau_1^A \equiv \mathit{true}$. De forma similar, $\tau_2^H \equiv X = [H|L]$ y $\tau_2^A \equiv H > Y$, y $\tau_3^H \equiv X = [H|L]$ y $\tau_3^A \equiv H \leq Y$. Finalmente, $D = D_1 \oplus D_2 \oplus D_3$, donde $D_1 \equiv ([\,], Y)$, $D_2 \equiv ([H|L], Y)$ y $D_3 \equiv ([H|L], Y)$. □

Para comprobar si τ cubre ρ, primero comprobamos que D cubre M, ignorando los tests aritméticos. Si D no cubre M, entonces obviamente, el test (completo) de entrada de p, τ, no cubre M, y devolvemos fallo. En otro caso, creamos (cero o más) subproblemas de recubrimiento, cada uno de ellos conteniendo sólo tests aritméticos, de la forma siguiente:

1. Sea A el conjunto de todas las tuplas de términos y negaciones de conjuntos cobásicos que aparecen en D (nótese que la negación de un conjunto cobásico es una tupla de términos, por tanto A es un conjunto de tuplas de términos), y sea $A' = \{b \in A \mid M \otimes b \not\simeq \Lambda\}$.
2. Para cada tupla de términos b en A':
   a) Sea $I = M \otimes b$ y $\theta = mgu(\bar{t}_M, \bar{t}_I)$;
   b) Sea $\tau_b = \bigvee_{j=1}^{m} r_j$, donde $\{r_1, \ldots, r_m\} = \{t_i \mid b \sqsubseteq D_i$ para algún $1 \leq i \leq n$ y t_i es el resultado de aplicar θ a τ_i^A (esto se hace teniendo en cuenta todas las posibilidades de variables "sinónimas")$\}$. Nótese que hay un algoritmo para comprobar si $b \sqsubseteq D_i$ en [Kun87].
   c) Comprobar si τ_b cubre ρ_I (recuérdese que ρ_I se refiere a la asignación de tipos de I):
      1) Supongamos que $\tau_b = s_1 \vee \cdots \vee s_n$ y cada s_i es una conjunción de tests aritméticos primitivos. Si $\tau_b \equiv$ **true** entonces devolvemos éxito;
      2) en otro caso, si para alguna variable x que aparece en todos los s_i, $1 \leq i \leq n$, se cumple que $type(x, \rho_I)$ no es un tipo numérico, entonces devolvemos fallo;
      3) en otro caso, usamos el algoritmo descrito en la sección 4.3.2 para comprobar si τ_b cubre ρ_I.

Nota: otra forma de crear los subproblemas es: $\tau_b = \bigvee_{j=1}^{m} r_j$, donde $\{r_1, \ldots, r_m\} = \{t_i \mid I \sqsubseteq D_i$ para algún $1 \leq i \leq n$ y t_i es el resultado de aplicar θ a $\tau_i^A\}$. Este algoritmo es más preciso que el anterior, pero es más complejo ya que I es un término anotado con tipos y por tanto tenemos que usar el algoritmo de recubrimiento descrito en la sección 4.3.1 para comprobar que $I \sqsubseteq D_i$.

Teorema 4.3.7 *Si D cubre M y para cada $b \in A'$, τ_b cubre I, entonces el test de entrada de p, τ, cubre M.*

Demostración Está claro que si D cubre M, entonces la disyunción de todas la tuplas de términos de los conjuntos cobásicos en A' también cubre M. Por tanto, para cualquier tupla de términos $\bar{x}$ que sea una instancia de M, existe al menos un $b \in A'$, tal que $\bar{x}$ es una instancia de b, y todos los tests τ_i^H tal que $b \sqsubseteq D_i$, tendrán éxito para $\bar{x}$. Si τ_b cubre I, entonces al menos uno de los tests t_i en τ_b tendrá éxito para $\bar{x}$. Por tanto, según la construcción de τ_b, al menos τ_i tendrá éxito para $\bar{x}$, y podemos concluir que τ cubre M. ∎

Ejemplo 4.3.3 Consideremos el ejemplo 4.3.2. Está claro que D cubre a M, por tanto procedemos de la forma siguiente:

1. $A = \{([\,], Y), ([H|L], Y)\}$, y $A' = A$.
2. Sean $b1 = ([\,], Y)$ y $b2 = ([H|L], Y)$. Entonces $\tau_{b1} \equiv$ **true** y $\tau_{b2} \equiv H > Y \vee H \leq Y$.
3. Tenemos que **true** cubre $(([\,], Y), (Y : integer))$, y también que $H > Y \vee H \leq Y$ cubre $(([H|L], Y), (L : intlist, H : integer, Y : integer))$, por tanto τ cubre M. □

Nótese que el enfoque anterior también se puede usar para particionar un problema en un subproblema de recubrimiento en el dominio de Herbrand (tests de unificación/desunificación) y cero o más subproblemas de cualquier tipo. En este caso, utilizaríamos el algoritmo apropiado para resolver cada uno de los subproblemas de recubrimiento resultantes.

4.4. Análisis de no-fallo

4.4.1. El algoritmo de análisis

Una vez que hemos determinado qué predicados cubren sus tipos la determinación de no-fallo es una tarea simple: según el teorema 4.3.1, el análisis de no-fallo se reduce a la determinación de alcanzabilidad en el grafo de llamadas del programa. En otras palabras, un predicado p no falla si no existe ningún camino en dicho grafo desde p hacia algún predicado q que no cubre su tipo. Esta información de alcanzabilidad se puede propagar fácilmente mediante un recorrido simple del grafo de llamadas en orden topológico inverso. Ilustramos esta idea con el siguiente ejemplo.

Ejemplo 4.4.1 Consideremos el siguiente predicado tomado de un programa quicksort:

```
qs(X1,X2)  :- X1 = []   ‖ X2 = [].
qs(X1,X2)  :- X1 = [H|L]  ‖ part(H,L,Sm,Lg),
    qs(Sm,Sm1), qs(Lg,Lg1), app(Sm1,[H|Lg1],X2).
```

Supongamos que `qs`/2 tiene el modo de llamada (`in`, `out`) el tipo (`intlist`, `-`), y supongamos que ya hemos demostrado que `part`/4 y `app`/3 cubren los tipos `(int, intlist, -, -)` y `(intlist, intlist, -)` inducidos a partir de sus literales del cuerpo de la cláusula recursiva anterior. Los tests de entrada para `qs`/2 son `X1 = []` $\vee$ `X1 = [H|L]`, los cuales cubren el tipo `intlist`, lo que significa que la unificación con la cabeza no fallará para `qs`/2. Deducimos por tanto que una llamada a `qs`/2 con el primer argumento instanciado a una lista de enteros no fallará. □

4.4.2. Implementación de un prototipo

Para evaluar la eficiencia y efectividad de nuestra técnica de análisis de no-fallo, hemos implementado e integrado en el sistema `ciaopp` [HBPLG99, HBC$^+$99] un prototipo de un analizador totalmente automático. El sistema toma como entrada un programa Prolog, el cual incluye una declaración de módulo de la forma estándar. También incluye los tipos y modos de los argumentos de los predicados exportados y sus definiciones. El sistema utiliza el analizador PLAI (también integrado en `ciaopp`) para inferir la información sobre los modos de llamada utilizando el dominio "Sharing+Freeness" [MH91], y una adaptación del análisis de Gallagher para inferir los tipos de los predicados [GdW94]. Esta adaptación incluye la extensión para tratar con tipos paramétricos. Los programas resultantes anotados con informaciones de tipos y modos se analizan utilizando los algoritmos presentados para el dominio de Herbrand y tests aritméticos lineales.

El recubrimiento en el dominio de Herbrand se realiza mediante una implementación simple y directa de los análisis presentados. La comprobación de recubrimiento para tests aritméticos lineales se implementa directamente utilizando el "Omega test" [Pug92]. Este test determina si existe una solución entera de un conjunto arbitrario de igualdades y desigualdades lineales, a las que nos referimos como a un problema.

Hemos probado el prototipo, en primer lugar, con algunos programas simples de prueba estándar, y luego con programas más complejos. Estos últimos los hemos tomado de los usados en el análisis de cardinalidad de Braem y otros [BCMH94], que es el trabajo más relacionado con el nuestro de los que conocemos. En la tabla 4.1 mostramos algunos resultados relevantes de nuestras pruebas. La columna **Programa** muestra los nombres de programa, **N** el número de predicados en el programa, **F** el número de predicados que el análisis detecta que no fallan, **Cov** el número de predicados que el análisis detecta que cubren sus tipos, **C** el número de predicados que no fallan detectados por el análisis de cardinalidad mencionado anteriormente [BCMH94], $\mathbf{T}_F$ el tiempo empleado por el análisis de no-fallo (SPARCstation 10, 55MHz, 64Mbytes de memoria), $\mathbf{T}_M$ el tiempo requerido para inferir los tipos y modos, y $\mathbf{T}_T$ el tiempo total de análisis (todos los tiempos vienen dados en milisegundos). La última fila de la tabla muestra los tiempos medios de análisis por predicado.

Los resultados son bastante esperanzadores y muestran que el análisis desarrollado es bastante preciso y que también es bastante más potente que el análisis de cardinalidad [BCMH94] en cuanto a detección de no-fallo se refiere. Que nosotros sepamos, el nuestro es el único análisis de los desarrollados hasta el momento que pueden detectar no-fallo. Los resultados experimentales presentados en [BCMH94] sugieren que el análisis de cardinalidad es más apropiado para detectar determinismo que no fallo. En [BCMH94] se comenta que la información sobre éxito seguro se puede mejorar utilizando un dominio de tipos más sofisticado. Sin embargo, esta observación también es aplicable a nuestro análisis, y los tipos inferidos por nuestro sistema son similares a los usados en [BCMH94]. Además, la razón principal de la potencia de nuestro algoritmo es el uso de la noción de recubrimiento, la cual permite detectar cuándo al menos una de las cláusulas (no necesariamente la misma) que definen un predicado no fallará para todas las llamadas posibles. El análisis de cardinalidad detecta el no-fallo sólo cuando al menos una de las cláusulas (siempre la misma) que definen un predicado no fallará para todas las llamadas posibles. Los tiempos del análisis de no-fallo son bastante buenos, a pesar de que la implementación actual del sistema es bastante simple y no está optimizada (por ejemplo, la llamada al "omega test" se realiza mediante un proceso externo). Los tiempos de análisis totales son bastante aceptables, teniendo en cuenta incluso los tiempos de análisis de tipos y modos, los cuales también son útiles en otras partes del proceso de compilación.

El sistema Mercury [HSC96] permite al programador declarar que un predicado producirá como mínimo una solución, e intenta verificar esto con respecto a los términos Herbrand con tests de igualdad. Que nosotros sepamos, el compilador de Mercury no trata con restricciones de desigualdad en el dominio de Herbrand. Ni tampoco trata con tests aritméticos, excepto en el contexto del constructor "if-then-else". Como tal, es considerablemente más débil que el enfoque descrito aquí.

4.5. Aplicaciones

Existen varias aplicaciones de nuestro análisis. La primera aplicación es la de implementar control de granularidad en compiladores paralelizantes, mediante el uso de funciones que estiman unas cotas inferiores del coste de procedimientos, la cual fue la pricipal motivación por la que hemos desarrollado el análisis de no-fallo. Referimos al lector al capítulo 2 en donde se describe en detalle esta aplicación y al capítulo 3 en el que se describe en detalle la determinación de cotas inferiores del coste de procedimientos.

La segunda aplicación también tiene que ver con el paralelismo (conjuntivo), en particular con la eliminación de computaciones especulativas. Consideremos un número de objetivos en un resolvente para las cuales se ha determinado que son independientes. Como se muestra en [HR95], e ignorando los costes de paralelización (los cuales pueden tenerse en cuenta utilizando técnicas de control de granularidad), se puede garantizar que el tiempo empleado en su ejecución paralela es menor o igual que el correspondiente a su ejecución secuencial. Sin embargo, es imposible determinar que no se realizará más trabajo en esta ejecución paralela. Esto se debe a la posibilidad de que uno de los objetivos falle. Consideremos dos objetivos p y q de forma que q se ejecuta después de p en la eje-

Programa	**N**	**F (%)**	**Cov (%)**	**C**	$\mathbf{T}_F$	$\mathbf{T}_M$	$\mathbf{T}_T$
Hanoi	2	2 (100)	2 (100)	N/A	60	860	920
Deriv	1	1 (100)	1 (100)	N/A	80	940	1,020
Fib	1	1 (100)	1 (100)	N/A	20	90	110
Mmatrix	3	3 (100)	3 (100)	N/A	90	350	440
Tak	1	1 (100)	1 (100)	N/A	10	110	120
Subs	1	1 (100)	1 (100)	N/A	50	90	140
Reverse	2	2 (100)	2 (100)	N/A	10	100	110
Qsort	3	3 (100)	3 (100)	0 (0)	80	440	520
Qsort2	5	3 (60)	3 (60)	0 (0)	100	390	490
Queens	5	2 (40)	2 (40)	0 (0)	120	360	480
Gabriel	20	3 (15)	10 (50)	0 (0)	420	1,860	2,280
Read	38	8 (21)	19 (50)	8 (21)	540	12,240	12,780
Kalah	44	18 (40)	29 (65)	6 (13)	1,500	14,570	16,070
Plan	16	4 (25)	11 (68)	0 (0)	810	7,000	7,810
Credit	25	10 (40)	18 (72)	0 (0)	4,720	1,470	6,190
Pg	10	2 (20)	6 (60)	0 (0)	540	1,600	2,140
Media	–	36 %	63 %	3 %	51 (/p)	239 (/p)	291 (/p)

Cuadro 4.1: Precisión y eficiencia del análisis de no-fallo (tiempos en mS).

cución secuencial. Supongamos también que p falla (en ambas de las ejecuciones, secuencial y la correspondiente paralela). Si p y q se ejecutan en paralelo, puede que una parte de q se ejecute hasta el punto en el que p falla (la ejecución de q se interrumpirá en este punto normalmente). Aunque esto no supone ninguna pérdida (slow-down), acarrea cómputos innecesarios que quitan recursos a otros cómputos que sí son útiles (y por tanto sí se reduce la ganancia). Por tanto, la determinación de que los objetivos en una conjunción no fallarán, todos salvo el de más a la derecha (nótese que el fallo de q en el ejemplo anterior no tiene estos efectos negativos), nos permite garantizar que no se realiza computaciones especulativas.

Una tercera aplicación se engloba dentro del área general de transformación de programas, en donde la información de no-fallo se puede utilizar para determinar el orden de ejecución de los literales en una cláusula. Consideremos una cláusula

$$H :- \ B_1, p(X), B_2, q(X), B_3$$

donde B_1, B_2, B_3 son secuencias de literales, $p(X)$ produce ligaduras para X, y $q(X)$ es el objetivo más a la izquierda que tiene X como un argumento de entrada. Si conocemos que p no falla, podemos transformar esta cláusula en

$$H :- \ B_1, B_2, p(X), q(X), B_3.$$

El código resultante puede que sea más eficiente que el original si un literal de la secuencia B_2 puede fallar.

Finalmente, otra de las aplicaciones importantes del análisis de no-fallo es acelerar el proceso de desarrollo de programas asistiendo a los programadores en la detección de predicados que con seguridad no fallan. Esto puede ayudar a la detección de errores de programación en tiempo de compilación, de forma muy similar a como lo hace la comprobación de tipos en lenguages tipados, ya que en programas lógicos usualmente se espera que un predicado tenga éxito y produzca una o más soluciones. Sin embargo, en la mayoría de los sistemas de programación, se realizan muy pocas comprobaciones en tiempo de compilación. Como comentábamos antes, el sistema está actualmente integrado en el sistema `ciaopp` y se utiliza con estos propósitos (además de su uso en optimizaciones).

4.6. Conclusiones del capítulo

Hemos propuesto un método en el que a partir de modos de llamada e información de tipos (aproximaciones por arriba) se pueden detectar procedimientos y llamadas que no fallarán (e.d., producirán al menos una solución o no terminarán). La técnica se basa en una noción simple e intuitiva de un conjunto de tests que "cubren" el tipo de un conjunto de variables. Hemos propuesto algoritmos correctos y completos para determinar recubrimientos que son precisos y eficientes en la práctica. Hemos comentado aplicaciones del análisis de no-fallo, entre las que podemos citar: estimación de cotas inferiores del coste de procedimientos (que a su vez son útiles en el control de granularidad); eliminación de paralelismo especulativo; detección de errores de programación y transformación de programas. Hemos implementado (e integrado en el sistema `ciaopp`) el análisis de no-fallo propuesto y hemos mostrado que se obtienen mejores resultados que con técnicas propuestas anteriormente.

Capítulo 5

Cálculo dinámico y eficiente de tamaños de términos

Como se vio en el capítulo 2, para estimar de forma razonablemente aproximada el coste de la ejecución de una llamada recursiva, y por tanto, poder realizar un control de granularidad adecuado, es necesario conocer en tiempo de ejecución el tamaño de los términos que aparecen en los argumentos de entrada de dicha llamada. Por tamaño de término nos referimos a medidas tales como longitud de lista, profundidad, o número de nodos (constructores o "funtores") en el mismo.

Posponer el cálculo preciso de tamaños de términos a tiempo de ejecución parece inevitable en general, ya que incluso técnicas sofisticadas de compilación tales como interpretación abstracta están basadas en calcular aproximaciones de sustituciones de variables para ejecuciones genéricas que corresponden a clases generales de entradas, mientras que el tamaño es una característica bastante específica de una entrada particular. Aunque la técnica de aproximación puede ser útil en algunos casos, nuestro objetivo es abordar el problema general en el que el tamaño de los datos de entrada han de ser calculados dinámicamente en tiempo de ejecución. Por supuesto, el calcular el tamaño de los datos en tiempo de ejecución es bastante simple, pero conlleva una cantidad significativa de trabajo extra asociado. Este trabajo incluye el tener que recorrer partes significativas de los términos (a menudo los términos enteros) y el proceso de cuenta o incremento realizado durante este recorrido.

Nuestro objetivo es proponer una forma nueva y eficiente de calcular dichos tamaños. La idea esencial está basada en la observación de que los términos a menudo ya son recorridos por procedimientos que son llamados en el programa antes que otros procedimientos que necesitan el conocimiento del tamaño de dichos términos, y por tanto, sus tamaños pueden ser calculados "sobre la marcha" por los procedimientos anteriores, después de realizarles algunas transformaciones. Aunque el proceso de cuenta no se elimina, el trabajo asociado se reduce debido a que no se realiza un recorrido adicional de los términos. En este capítulo presentamos un método sistemático para determinar si un programa puede ser transformado para calcular el tamaño de un término particular en un punto del programa sin tener que realizar un recorrido adicional de dicho término. Además, si existen varias transformaciones, la técnica consigue encontrar transformaciones minimales con respecto a cierto criterio.

Aunque la principal motivación de nuestro trabajo sobre cálculo de tamaños ha sido el control de granularidad, es de destacar que la necesidad de conocer en tiempo de ejecución el tamaño de los términos a los cuales están ligadas las variables de un programa lógico se pone de manifiesto además en otras aplicaciones relacionadas con optimización de programas, como por ejemplo eliminación de recursividad y selección de diferentes algoritmos o reglas de control cuya eficiencia puede depender de tales tamaños.

En el caso de la eliminación de la recursividad, el problema consiste en que, dado que los tamaños de ciertos términos son conocidos, un predicado recursivo se puede convertir en otro no recursivo, que contiene los cuerpos de las diferentes recursiones y es más eficiente. Técnicas tales como “reform compilation” [Mil91, BM92] intentan hacer esto de forma eficiente realizando cierto preproceso en tiempo de compilación, pero dejando el cálculo del tamaño de los términos para ser realizado en tiempo de ejecución.

Los contenidos del resto del capítulo son los siguientes: en la sección 5.1 se hace una breve descripción de la técnica. La sección 5.2 introduce las representaciones básicas y la sección 5.3 presenta el concepto de transformación. La sección 5.4 introduce los conceptos de transformaciones irreducibles y óptimas, y destaca su utilidad. La sección 5.5 presenta algoritmos para encontrar transformaciones irreducibles y un ejemplo completo de uno de ellos. La sección 5.6 discute las ventajas de esta técnica y finalmente la sección 5.7 muestra algunos resultados experimentales.

5.1. Descripción del enfoque

Como mencionamos al principio del capítulo, estamos interesados en transformar algunos predicados de forma que calculen los tamaños de algunos de sus datos en tiempo de ejecución, además de realizar su computación normal. A menudo se da la situación en la que, debido a previas transformaciones u otras razones, el tamaño de ciertos datos es conocido, de forma que éste puede ser utilizado como un punto de partida en el cálculo dinámico del tamaño de los datos que se necesita determinar en un determinado punto del programa. Por tanto, estamos interesados en el problema general de transformar programas de forma que calculen el tamaño de un conjunto de términos, dado que se conoce el tamaño de otro conjunto (disjunto) de términos.

Por ejemplo, consideremos el predicado `append/3`, definido como:

```
append([], L, L).
append([H|L], L1, [H|R])  :-  append(L, L1, R).
```

Supongamos que queremos transformar dicho predicado pare que calcule la longitud de su tercer argumento. Observando el caso base podemos inferir que la longitud del término que aparece en el tercer argumento de la cabeza es igual a la del término que aparece en el segundo argumento después de cualquier computación que tenga éxito. Podemos expresar esta relación de la forma siguiente: $head[3] = head[2]$, en donde $head[i]$ denota el tamaño del término que aparece en el i-ésimo argumento de la cabeza. Por tanto, podemos transformar este caso base añadiendo dos argumentos adicionales, que representan el tamaño de los términos que aparecen en el segundo y tercer argumento respectivamente.

```
append3i2([], L, L, S, S).
```

De esta forma, si llamamos al caso base suministrando el tamaño del segundo argumento, obtendremos el del tercero. Observando la cláusula recursiva, vemos que el tamaño del tercer argumento de la cabeza es igual al tamaño del tercer argumento del primer literal del cuerpo más uno. Expresamos esta relación de tamaño de la forma siguiente: $head[3] = body_1[3] + 1$, en donde $body_j[i]$ denota el tamaño del término que aparece en el i-ésimo argumento del j-ésimo literal del cuerpo (numeramos los literales de izquierda a derecha, asignando "1" al literal que hay justo después de la cabeza).

Podemos pensar entonces en utilizar una versión transformada de este literal del cuerpo para calcular $body_1[3]$. Pero para realizar esto es necesario suministrar a la llamada el tamaño del segundo argumento de este literal del cuerpo ($body_1[2]$), de forma que se calcule $body_1[3]$ cuando la recursión termine. Dado que ya tenemos la relación de tamaño $body_1[2] = head[2]$, podemos concluir que es posible calcular el tamaño del tercer argumento de `append/3` si suministramos el tamaño del segundo argumento en la llamada. La cláusula recursiva se puede transformar trivialmente, con el conocimiento de las relaciones de tamaño anteriores.

Concluimos que el predicado `append/3` puede transformarse de forma que calcule el tamaño de su tercer argumento, dado que se le suministra el del segundo en la llamada a dicho predicado. El predicado transformado podría ser:

```
append3i2([],L,L,S,S).
append3i2([H|L],L1,[H|R],S2,S3) :- append3i2(L,L1,R,S2,Sb3),
                                   S3 is Sb3 + 1.
```

Por razones de claridad de exposición hemos utilizado esta transformación, aunque no sea ideal, dado que destruye la recursividad "por la cola". Sin embargo, el problema de realizar otra transformación equivalente a la anterior y que preserve dicha recursividad es trivial. De hecho, éstas son las transformaciones realizadas en la práctica. Nótese también que, aunque el presentar la técnica propuesta en términos de una transformación fuente-fuente es conveniente por claridad y porque es una técnica de implementación viable, obviamente, las transformaciones pueden realizarse también a un nivel más bajo, para reducir aún más los "overheads" en tiempo de ejecución.

Observando el predicado transformado anterior vemos que el problema se puede reducir a encontrar lo que denominamos un "grafo de dependencias de tamaños" para cada cláusula del predicado que se va a transformar. En la figura 5.1 mostramos los grafos de dependencia de tamaños correspondientes al ejemplo anterior. En esta figura, los grafos `G2` y `G1` corresponden al caso base y a la cláusula recursiva de `append/3` respectivamente.

Informalmente, el conjunto de grafos de dependencia de tamaños contiene la información necesaria para transformar un predicado, y se representa con lo que denominamos un *nodo de transformación*. En general, es necesario transformar más de un predicado para realizar un cálculo de tamaños determinado. En este

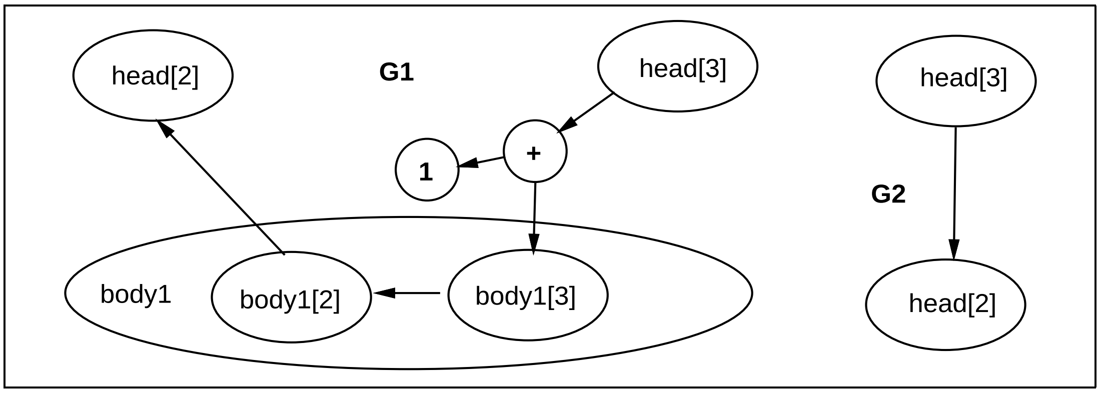

Figura 5.1: Grafos de dependencia de tamaños para el predicado `append`/3.

caso, los nodos de transformación se pueden ver como nodos en un árbol de búsqueda que se explorará con el objetivo de encontrar un conjunto de tales nodos que conduzca a una transformación del programa que calcula correctamente los tamaños deseados de los datos.

En esencia, el enfoque propuesto implica:

1. Inferir todas las posibles relaciones de tamaño entre argumentos de las cláusulas del programa que pueden verse implicadas en el cálculo de tamaños deseado. Podemos considerar solamente predicados en el componente fuertemente conexo del grafo de llamadas correspondiente al predicado que es el punto de entrada de la transformación.

2. Construir todos los posibles nodos de transformación.

3. Encontrar el conjunto de nodos de transformación a partir de estas relaciones que conduzcan a cálculos de tamaños correctos.

La inferencia estática de relaciones de tamaño entre argumentos se ha estudiado ampliamente [UV88, VS92, DLH90, GDL95, BK96]. En particular, nosotros nos referimos a las relaciones de tamaño descritas en [DLH90]. Se pueden utilizar varias métricas para determinar el "tamaño" de un término, p.ej., número de nodos en un término (term-size), profundidad del término (term-depth), longitud de lista (list-length), valor entero (integer-value), etc. [DL93]. Consideremos la función $|\cdot|_m : \mathcal{H} \rightarrow \mathcal{N}_\perp$ (definida en [DL93]), que hace corresponder a los términos "ground" su tamaño de acuerdo con alguna métrica específica m, en donde $\mathcal{H}$ denota el Universo de Herbrand, e.d. el conjunto de términos "ground" del lenguaje, y $\mathcal{N}_\perp$ denota el conjunto de los números naturales aumentado con el símbolo especial $\perp$, que significa "no definido". Algunos ejemplos de dichas funciones son "list_length", que asigna a las listas "ground" sus longitudes, y a todos los demás términos "ground" el símbolo $\perp$; "term_size", que asigna a cada término "ground" el número de constantes y símbolos de función que aparecen en él; "term_depth", que asigna a cada término "ground" la profundidad de su representación en árbol; etc. Por ejemplo, $|[\mathtt{a},\mathtt{b}]|_{\mathrm{list\_length}} = 2$, en cambio $|f(a)|_{\mathrm{list\_length}} = \perp$.

En [DL93], las relaciones de tamaño se clasifican en dos tipos: "intra-literal" e "inter-literal". El primer tipo se refiere a relaciones de tamaño entre las posiciones de argumentos en un mismo literal. Son relaciones que se cumplen entre los tamaños de los argumentos de todos los átomos en el conjunto de éxito para el predicado correspondiente al literal y son similares a las descritas en [VS92]. El segundo tipo se refiere a las relaciones entre posiciones de argumentos entre diferentes literales de una cláusula de la cabeza de la cláusula. Por ejemplo $size_3 = size_1 + size_2$ es una relación de tamaño intra-literal para el predicado `append/3` que expresa que la longitud de su tercer argumento es la suma de las longitudes de sus dos primeros argumentos. Sin embargo $head[3] = body_1[3] + 1$ es una relación de tamaño de tipo inter-literal correspondiente a la cláusula recursiva de `append/3`, y expresa que para cada sustitución que hace "ground" a los términos que aparecen en las posiciones $head[3]$ y $body_1[3]$, el tamaño del término que aparece en la posición $head[3]$ es igual al tamaño del término que aparece en la posición $body_1[3]$ más uno, e.d. $|\ \mathtt{[H|R]}\ |_{\mathrm{list\_length}} = |\ \mathtt{R}\ |_{\mathrm{list\_length}} + 1$ se cumple para cada sustitución que hace "ground" a `H` y `R`.

5.2. Transformación de procedimientos

Un *grafo de dependencia de tamaños* es un grafo dirigido y acíclico cuyos nodos pueden ser de los tipos siguientes: **a)** Una *posición* en una cláusula: $head[i]$ o $body_j[i]$, como se describe en la sección 5.1; **b)** Un *operador aritmético* binario ($+$, $-$, etc.); o **c)** Un *número* entero no negativo.

Distinguimos dos clases de arcos en el grafo:

- Arcos del tipo *Intra-literal* que son los que van desde una posición en un literal del cuerpo hasta otra posición en el mismo literal, más formalmente, desde $body_i[k]$ hasta $body_j[n]$ en donde $i = j$ y $k \neq n$. Su significado es el siguiente: el tamaño del término que aparece en la posición k-ésima del i-ésimo literal del cuerpo lo calcula una versión transformada del predicado de este literal. Para realizar dicho cálculo de tamaños, la versión mencionada requiere que en la llamada se suministre el tamaño del término que aparece en la posición n-ésima.
- Arcos de tipo *Inter-literal* son los demás (e.d. aquéllos que no son de tipo intra-literal).

Hay un arco de tipo inter-literal desde la posición x hacia otra posición y si el tamaño del término que aparece en la posición x es igual al tamaño del término que aparece en la posición y. Los nodos que son operadores aritméticos o números se utilizan para expresar relaciones aritméticas entre los tamaños de posiciones de argumentos, como se ilustra en la figura 5.1. Atendiendo al número y tipo de arcos salientes y entrantes permitidos, establecemos una clasificación de los nodos de la forma siguiente:

- Solamente están permitidos dos casos para nodos referentes a posiciones de la cabeza:
  - Nodos de tamaños de entrada, los cuales tienen uno o más arcos entrantes del tipo inter-literal.
  - Nodos de tamaños de salida, que tienen exactamente un arco saliente del tipo inter-literal y ningún arco entrante.
- Para las posiciones del cuerpo, solamente se permiten dos casos:
  - Nodos de tamaños suministrados, los cuales tienen un arco saliente del tipo inter-literal y uno o más arcos entrantes del tipo intra-literal. Estos nodos corresponden a los argumentos cuyo tamaño se suministra en la llamada a un literal transformado del cuerpo.
  - Nodos de tamaños calculados, que tienen uno o más arcos entrantes del tipo inter-literal y cero o más arcos salientes del tipo intra-literal. Estos nodos corresponden a aquellos argumentos cuyo tamaño se calcula por literales transformados del cuerpo.
- Un nodo que es un operador aritmético binario tiene dos arcos salientes del tipo inter-literal y un arco entrante del tipo inter-literal.
- Un nodo que es un número entero no negativo tiene solamente un arco entrante del tipo inter-literal y ningún arco saliente.

Consideremos el grafo de dependencias de tamaños `G1` de la figura 5.1. $head[2]$ es un nodo de tamaños de entrada, $head[3]$ un nodo de tamaños de salida, $body_1[2]$ es un nodo de tamaños suministrados y $body_1[3]$ es un nodo de tamaños calculados. Un *nodo de transformación* para un predicado $Pred$ es un par $(Label, Graphs)$, en donde $Graphs$ es un conjunto de grafos de dependencias de tamaños. Hay exactamente un grafo para cada cláusula que define un predicado. Supongamos que hay n cláusulas en la definición del predicado $Pred$. Sea G_i el grafo de dependencia de tamaños para la cláusula i, e I_i y O_i el conjunto de argumentos de entrada y salida de G_i respectivamente. Sea $I = \bigcup_{i=1}^{n} I_i$ y $O = \bigcup_{i=1}^{n} O_i$. Entonces $Label$, la etiqueta del nodo de transformación, es una tupla $(Pred, Is, Os)$, en donde $Is = \{i \mid \textit{head}[i] \in I\}$ y $Os = \{i \mid \textit{head}[i] \in O\}$. Con la etiqueta definida anteriormente podemos expresar qué predicado $Pred$ se transforma y qué tamaños de argumentos se calcularán en función de qué otros. La versión transformada de $Pred$ tendrá un argumento adicional para cada elemento $i \in Is$ (el cual quedará ligado al tamaño del término que aparece en el i-ésimo argumento de la cabeza a la llamada del predicado) y $j \in Os$ (que quedará ligada al tamaño del término que aparece en la posición j-ésima de la cabeza una vez que la llamada tenga éxito). Por ejemplo, $(\textit{append}/3, \{2\}, \{3\})$ es una etiqueta que establece que el predicado `append`/3 se transformará para calcular el tamaño de su tercer argumento, dado que a la llamada se le suministra el tamaño del segundo argumento. Esto significa que es necesario añadir dos argumentos extra al predicado transformado los cuales representarán los tamaños del segundo y tercer argumento de `append`/3.

Ejemplo 5.2.1 La figura 5.1 representa el nodo de trasformación compuesto por los grafos de dependencia de tamaños `G1` y `G2`, es decir $((\textit{append}/3, \{2\}, \{3\}), \{$`G1`, `G2`$\})$. $\square$

Requerimos que los grafos de dependencia de tamaños cumplan la condición siguiente: si hay un arco de tipo inter-literal desde un nodo de tamaños suministrados $\textit{body}_i[k]$ hacia un nodo de tamaños calculados $\textit{body}_j[n]$ entonces $j < i$. Esta condición nos asegura que los tamaños suministrados a un literal transformado solamente los calculan literales anteriores en el cuerpo. Este requisito se debe al hecho de que los tamaños suministrados deben estar "ground" en la llamada, dado que nos interesa utilizar predicados predefinidos similares a "is/2"

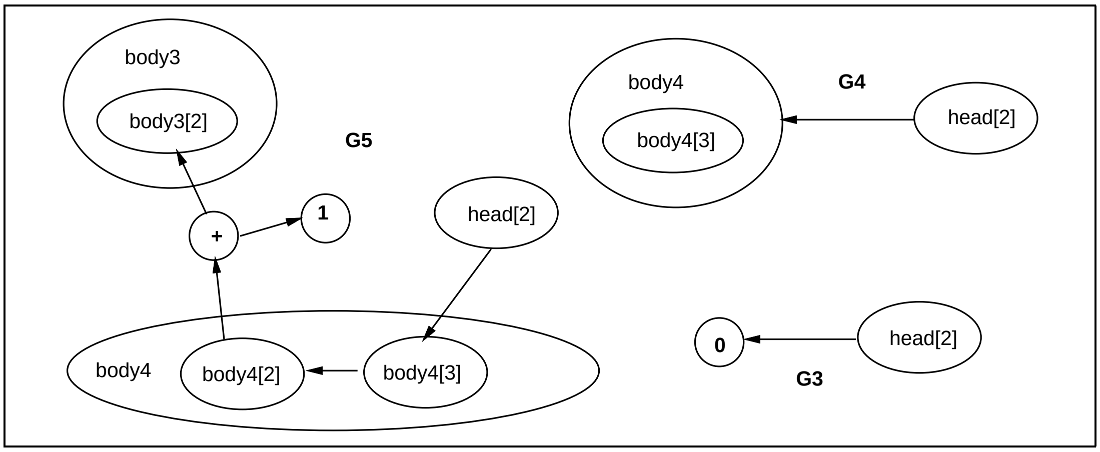

Figura 5.2: Grafos de dependencia de tamaños para el predicado `qsort`/2.

(de hecho, versiones más eficientes y especializadas) para realizar las operaciones aritméticas necesarias para calcular los tamaños, y estos predicados predefinidos requieren que todos sus argumentos excepto uno de ellos sea "ground". Esta condición puede relajarse si el lenguaje subyacente es por ejemplo un lenguaje de programación lógica con restricciones [JL87], que pueda resolver ecuaciones lineales. Sin embargo, la resolución de ecuaciones probablemente supondría un coste significativo. Por tanto, forzamos la condición mencionada por dos motivos: para que el cálculo de tamaños sea eficiente y para que los programas transformados puedan ejecutarse sin necesidad de ninguna capacidad de resolución de restricciones en el lenguaje subyacente.

En un grafo de dependencia de tamaños el conjunto de todos los nodos correspondientes al i-ésimo literal (e.d. aquellos de la forma $body_i[j]$) se denomina un *nodo de literal* $body_i$. A modo de ejemplo, consideremos el grafo de dependencia de tamaños `G1` de la figura 5.1. En dicha figura, el conjunto $\{body_1[2], body_1[3]\}$ es el nodo de literal $body_1$. También agrupamos los nodos de tamaños suministrados y calculados correspondientes a un literal determinado en los conjuntos S y C respectivamente (en el ejemplo $S = \{body_1[2]\}$ y $C = \{body_1[3]\}$). Asociamos con el nodo de literal la etiqueta $(Pred, Is, Os)$, en donde $Pred$ es el nombre del predicado y aridad del literal y $Is = \{j \mid body_i[j] \in S\}$ y $Os = \{j \mid body_i[j] \in C\}$ (en el ejemplo, la etiqueta asociada con el nodo de literal $body_1$ es $(append/3, \{2\}, \{3\}))$. La etiqueta del nodo de literal indica qué versión transformada del predicado del literal corresponde a dicho literal. Esta es la versión que realiza el cálculo de tamaños expresado por la propia etiqueta. Cuando se transforma la cláusula en la que aparece el literal, dicho literal se reemplazará por una llamada que realiza el cálculo de tamaños.

5.3. Transformación de conjuntos de procedimientos: *transformaciones*

En esta sección abordamos el problema de transformar conjuntos de procedimientos que forman parte de un grafo de llamada para que éstos realicen el cálculo de tamaños de términos. En este caso se necesita al menos un nodo de transformación para algunos de ellos, los cuales deben de cumplir unas condiciones que se explicarán a continuación.

Definición 5.3.1 [Transformación] Es un grafo compuesto por un conjunto N de nodos de transformación y un conjunto de arcos. Hay un nodo de transformación especial $E \in N$ que llamamos el *punto de entrada* de la transformación y:

1. Sea G un grafo de dependencias de tamaños cualquiera de T_1, en donde T_1 es un nodo de transformación tal que $T_1 \in N$, y sea l cualquier nodo de literal de G, entonces l tiene exactamente un arco saliente y ningún arco entrante. Este arco va desde l hasta algún nodo de transformación $T_2 \in N$ tal que la etiqueta de T_2 es igual a la etiqueta asociada con el nodo de literal l (nótese que T_1 y T_2 pueden ser el mismo nodo de transformación).

La idea intuitiva subyacente a este arco es la siguiente: supongamos que L_1 es el literal correspondiente a l en la cláusula correspondiente a G, y L_2 es la versión transformada de L_1 la cual realiza el cálculo de tamaños indicado por la etiqueta asociada con l. El arco establece que el predicado de L_1 se puede transformar de acuerdo con la información representada en T_2 obteniéndose el predicado de L_2.

2. Hay un arco desde un nodo de transformación $T_1 \in N$ hacia un nodo de transformación $T_2 \in N$ si y sólo si hay un arco desde algún nodo de literal l de T_1 hacia T_2. Intuitivamente, este arco establece qué predicado transformado correspondiente a T_1 llama al predicado transformado correspondiente a T_2.

3. Todos los nodos de transformación $T \in N$ son alcanzables desde E. ■

Definición 5.3.2 [Especificación del cálculo de tamaños] Definimos una *Especificación del Cálculo de Tamaños* como un par $(Pred, Os)$, en donde $Pred$ es el nombre y aridad del predicado que será transformado, y Os es un conjunto de números de argumentos cuyos tamaños son calculados por el predicado transformado en tiempo de ejecución. ■

Definición 5.3.3 [Transformación para una especificación de cálculo de tamaños] Una transformación para una especificación del cálculo de tamaños $(Pred, Os)$, es una transformación T tal que la etiqueta del punto de entrada de T es de la forma $(Pred, Is, Os)$. ■

Teorema 5.3.1 *Si existe una Transformación T, para una especificación de cálculo de tamaños $(Pred, Os)$, tal que la etiqueta del punto de entrada de T es $(Pred, Is, Os)$, entonces es posible transformar las cláusulas de $Pred$ de forma que se obtiene un predicado transformado $Pred'$, tal que $Pred'$ calcula los tamaños de los argumentos indicados en Os, dado que se suministran los tamaños* *de los argumentos indicados en Is, además de realizar la misma computación que realiza $Pred$.*$\square$

5.4. Transformaciones irreducibles/óptimas

Dado que puede haber muchas transformaciones posibles para una determinada especificación de cálculo de tamaños, nos interesa encontrar transformaciones que supongan el mínimo coste en tiempo de ejecución. Dicho coste es dependiente del sistema, ya que depende del coste del paso de argumentos y de las operaciones aritméticas. La reducción del mencionado coste nos sugiere el considerar transformaciones que tengan el mínimo número de nodos de transformación y que cada uno de ellos tenga el mínimo número de elementos en Is, en donde $(Pred, Is, Os)$ es la etiqueta de cualquier nodo que forma parte de la transformación. Es decir, para transformar un predicado de forma que éste calcule los tamaños de algunos de sus argumentos, quisiéramos conocer cuáles son los argumentos cuyo tamaño es estrictamente necesario para realizar este cálculo (de forma que sólo se añadan los argumentos adicionales necesarios y se añadan sólo las operaciones necesarias a los predicados transformados) y también cuál es el mínimo número de predicados que han de ser transformados. En primer lugar introduciremos el concepto de *transformación irreducible* y veremos que para determinar si es posible transformar un predicado, sólo necesitamos considerar transformaciones irreducibles. Seguidamente presentaremos algunas ideas acerca de la obtención de transformaciones irreducibles óptimas.

Definición 5.4.1 [Orden entre etiquetas] Dadas dos etiquetas, $X = (Pred, Is_x, Os)$ e $Y = (Pred, Is_y, Os)$, decimos que $X <_l Y$ si y sólo si $Is_x \subset Is_y$.
$\blacksquare$

Por ejemplo: $(append/3, \{2\}, \{3\}) <_l (append/3, \{1, 2\}, \{3\})$, sin embargo $(append/3, \{2\}, \{3\}) \not<_l (append/3, \{1\}, \{3\})$

Definición 5.4.2 [Transformación Irreducible]

Una transformación T, es *irreducible* si y sólo si:

1. La etiquetas de los nodos de transformación en T son únicas.

2. No hay dos nodos de transformación en T, etiquetados con las etiquetas X e Y respectivamente, tales que $X <_l Y$. ∎

Representamos una transformación irreducible como un par (L, T), en donde T es un conjunto de nodos de transformación y L es la etiqueta del nodo de transformación que es el punto de entrada de la transformación (recuérdese que las etiquetas de los nodos de transformación en T son únicas). El punto de entrada pertenece al conjunto T. Dado que las etiquetas de los nodos de transformación son únicas, no es necesario representar explícitamente ningún arco en la transformación irreducible (estos se pueden determinar de forma unívoca de las condiciones de la definición 5.3.3). Por tanto, omitimos todos los arcos.

Ejemplo 5.4.1 Consideremos el predicado `qsort/2` definido de la forma siguiente:

```
C1: qsort([],[]).
C2: qsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        qsort(Ls,Ls2),qsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).
```

y supongamos que queremos transformarlo para que calcule la longitud de su segundo argumento. En la figura 5.2 se muestran grafos de dependencia de tamaños

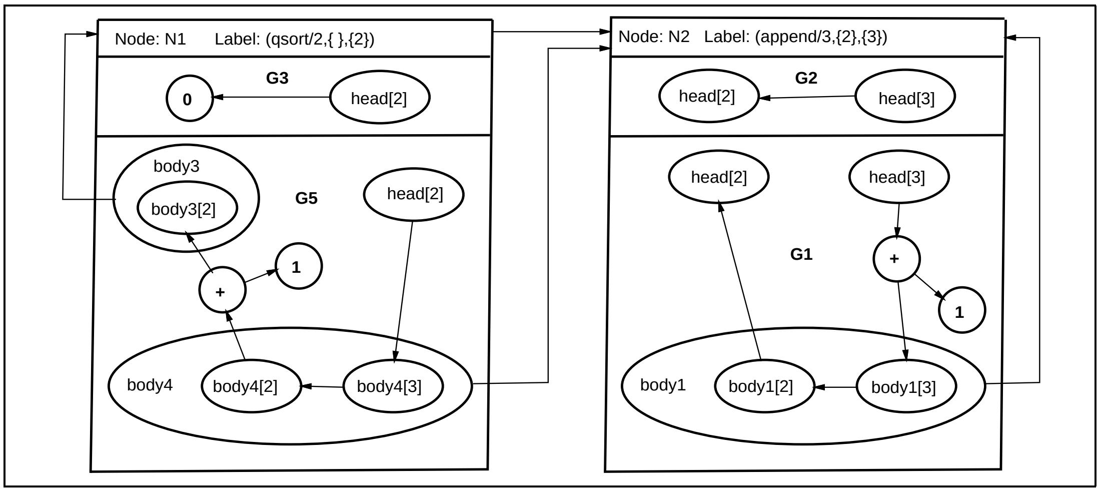

Figura 5.3: Una transformación irreducible.

correspondientes a las cláusulas del predicado `qsort`/2. En esta figura, el grafo de dependencia de tamaños `G3` corresponde al caso base (`C1`) de este predicado, y `G4` y `G5` corresponden a su cláusula recursiva (`C2`). Sea $N1$ el nodo de transformación $N1 = ((qsort/2, \emptyset, \{2\}), \{G3, G5\})$. Sea $N2$ el nodo de transformación del ejemplo 5.2.1. Entonces, el par $((qsort/2, \emptyset, \{2\}), \{N1, N2\})$ es una transformación irreducible, cuyo punto de entrada es el nodo $N1$. Esta transformación irreducible se representa en la figura 5.3. El par $((append/3, \{2\}, \{3\}), \{N2\})$ es también una transformación irreducible. $\square$

Un comentario acerca de la generación y naturaleza de nodos de transformación: esta generación se realiza a través de un análisis de modos para determinar patrones de flujo de datos [Deb89, MH92, Bru91] y un análisis del tamaño de los argumentos [DLH90]. Este análisis combinado puede inferir, en algunos casos, relaciones de tamaño de tipo intra-literal entre los argumentos de un predicado. Esta información se puede usar para generar nodos de transformación que forman parte de una transformación, pero que necesitan recorrer menos datos, ya que el cálculo de un tamaño dado puede realizarse directamente mediante una operación aritmética, en lugar de tener que ir incrementando durante la ejecución del predicado. Supongamos por ejemplo que el análisis infiere la relación de tamaño de tipo intra-literal $size_3 = size_1 + size_2$ para el predicado `append`/3 (la cual expresa que la longitud de su tercer argumento es la suma de las longitudes de sus dos primeros argumentos), y la relación de tamaño de tipo intra-literal $size_2 = size_1$ para el predicado `qsort`/2. Consideremos la cláusula `C2` del ejemplo 5.4.1. Utilizando $size_3 = size_1 + size_2$ para `append`/3 tenemos que se cumple $|\texttt{L2}|_{\text{list\_length}} = |\texttt{Ls2}|_{\text{list\_length}} + |[\texttt{First}|\texttt{Lg2}]|_{\text{list\_length}}$ para cada sustitución que hace "ground" todos los términos que aparecen en ella, y también que se cumple $|\texttt{L2}|_{\text{list\_length}} = |\texttt{Ls2}|_{\text{list\_length}} + |\texttt{Lg2}|_{\text{list\_length}} + 1$. Por tanto, podemos inferir la siguiente relación de tamaño de tipo inter-literal $head[2] = body_2[2] + body_3[2] + 1$ la cual no implica ninguna transformación del predicado `append`/3 sino solamente del predicado `qsort`/2. Además, utilizando $size_2 = size_1$ para `qsort`/2 tenemos que también se cumplen $|\texttt{Ls2}|_{\text{list\_length}} = |\texttt{Ls}|_{\text{list\_length}}$ y $|\texttt{Lg2}|_{\text{list\_length}} = |\texttt{Lg}|_{\text{list\_length}}$. Por tanto, podemos inferir otra relación de tamaño de tipo inter-literal $head[2] = body_1[3] + body_1[4] + 1$ (la cual implica la transformación del predicado `partition`/4).

Teorema 5.4.1 *Si existe una transformación T para una especificación de cálculo de tamaños X entonces existe una transformación irreducible T' para X.*□

El teorema 5.4.1 implica que sólo necesitamos encontrar transformaciones irreducibles para determinar si un procedimiento es transformable para calcular tamaños. Obviamente, las transformaciones irreducibles conducen a procedimientos transformados con (potencialmente) menos coste en tiempo de ejecución que las transformaciones de las cuales se han obtenido, pero ahora el problema consiste en determinar qué transformación irreducible tendrá menos coste, o en otras palabras, será minimal. El problema de encontrar dichas transformaciones óptimas estriba en el hecho de que necesitamos utilizar dos parámetros (número de nodos de transformación y número de argumentos necesarios) para realizar comparaciones y por ello algunas transformaciones no se pueden comparar, en el sentido de que una puede ser menor que la otra según un criterio, pero puede que lo contrario sea cierto según el otro criterio. En la práctica siempre podemos asignar costes o pesos al paso de argumentos y a las operaciones aritméticas de modo que obtengamos para cada transformación una función que da su coste en función de los tamaños de los datos de entrada. En este caso podemos comparar el coste de las transformaciones irreducibles y decidir cuales de ellas son óptimas. De la misma forma, podemos comparar el coste de las transformaciones irreducibles con el coste de realizar el cálculo de tamaños de forma estándar, e.d. utilizando predicados predefinidos tales como `length`/2, para ver así la conveniencia de realizar la transformación que calcula los tamaños de términos.

5.5. Búsqueda de transformaciones irreducibles

Puesto que el número de nodos de transformación para una especificación de cálculo de tamaños dada es finito, un posible algoritmo para encontrar transformaciones podría consistir simplemente en generar todos los posibles conjuntos de nodos de transformación y comprobar que son transformaciones irreducibles. Nótese que el número de nodos de transformación está restringido en cualquier caso por el número de relaciones de tamaño que se pueden inferir mediante análisis de tamaños [DLH90] (de hecho, si el algoritmo no encuentra ninguna transformación, ésto no significa que no exista ninguna, sino que con la información inferida en el análisis de tamaños no es posible encontrarla). Sin embargo, hay algunos enfoques más eficientes.

En la figura 5.4 proponemos un algoritmo simple, y dirigido por el objetivo (para el que luego propondremos algunas optimizaciones) que realiza una búsqueda partiendo de una especificación de cálculo de tamaños (también se puede desarrollar un algoritmo de tipo “bottom-up”). El espacio de búsqueda está descrito

Predicado: `find_trans(SCS, S, Trans)`

Entrada: una especificación de cálculo de tamaños `SCS` y la información `S` referente a relaciones de tamaño entre argumentos de las cláusulas diferentes de un programa para el predicado en `SCS`, inferida mediante análisis.

Salida: una transformación irreducible `Trans` para `SCS`.

Definición:
```
find_trans(SCS, S, Trans) ←
    generate_label(SCS, L), search([L], S, nil, T), Trans=(L, T).
```

Predicado: `generate_label(SCS, L)`

Descripción: genera una etiqueta `L` para `SCS`. Falla cuando todas las etiquetas posibles se han generado mediante vuelta atrás.

Predicado: `search(LabelList, SzRel, InTrans, OutTrans)`

Definición:
```
search(nil, SzRel, Trans, Trans).
search([Label|LabList], SzRel, InTrans, OutTrans) ←
    generate_node(Label,SzRel,[Label|LabList],InTrans,Node,LL),
    append(LL, LabList, NewLabList), Trans = [Node|InTrans],
    search(NewLabList, SzRel, Trans, OutTrans).
```

Predicado: `generate_node(Label, SzRel, LabList, InTrans, Node, LL)`

Descripción: Genera un nodo de transformación `Node` con etiqueta `Label`, utilizando la información sobre relaciones de tamaño `SzRel` de forma que se cumpla la siguiente condición: Sea S_t el conjunto de etiquetas de los nodos de transformación en la transformación actual `InTrans`. Sea S_l el conjunto de etiquetas en `LabList`. Sea S_n el conjunto de etiquetas asociadas con los nodos de literal en `Node`. Entonces, no existen dos etiquetas l_1 y l_2, $l_1 \in S_n$ y $l_2 \in (S_t \cup S_l \cup S_n)$, tal que $l_2 <_l l_1$.

Si esto no es posible, o todos los posibles nodos de transformación se han generado previamente mediante vuelta atrás, entonces falla. En otro caso, crea una lista `LL` que contiene las etiquetas en el conjunto $S_n - (S_t \cup S_l)$ y tiene éxito. Por brevedad, omitimos la generación de `Node`.

Figura 5.4: Un algoritmo para encontrar transformaciones irreducibles.

por el predicado `find_trans`/3. Nótese que para las transformaciones irreducibles generadas tenemos que comprobar todavía cuales de ellas tienen el mínimo coste en cuanto al proceso de cálculo de tamaños se refiere.

Ejemplo 5.5.1 Consideremos el predicado `qsort`/2 definido en el ejemplo 5.4.1, y supongamos que queremos transformarlo para que calcule la longitud de su segundo argumento, es decir, queremos encontrar una transformación para la especificación de cálculo de tamaños $(qsort/2, \{2\})$. Suponemos que se realiza una búsqueda en profundidad (similar a la realizada por el predicado `find_trans`/3 cuando se ejecuta en Prolog).

1. La búsqueda empieza llamando a `find_trans(SCS, S, Trans)`, en donde `SCS` $= (qsort/2, \{2\})$ y `S` constituyen la información sobre relaciones de tamaño para los predicados en el programa `quick-sort` (e.d. `qsort`/2, `partition`/4, y `append`/3).
2. Supongamos que `generate_label(SCS, L)` genera la etiqueta `L` $= (qsort/2, \emptyset, \{2\})$.
3. Entonces se hace una llamada a `search([L], S, nil, T)`. Supongamos que `generate_node(L, S, [L], nil, Node, LL)` tiene éxito generando el nodo de transformación `Node` $=$ `N1`, en donde `N1` $= ((qsort/2, \emptyset, \{2\}), \{G3, G4\})$, siendo $G3$ y $G4$ los grafos de dependencia de tamaños de la figura 5.2, y haciendo `LL` $=$ `[L1]`, en donde `L1`$=$ $(append/3, \emptyset, \{3\})$.
4. A continuación se realiza una llamada recursiva `search([L1], S, [N1], OutTrans)`. Dicha llamada puede fallar puesto que también puede hacerlo `generate_node(L1, S, [L1], [N1], Node2, LL2)`. Por tanto, se realiza una vuelta atrás y se prueba ahora el nodo:

   `generate_node(L, S, [L], nil, Node, LL)`

Supongamos que esta llamada tiene éxito y que genera el nodo de transformación `Node` = `N2`, en donde `N2` = $((qsort/2, \emptyset, \{2\}), \{G3, G5\})$, y $G3$ y $G5$ son los grafos de dependencia de tamaños de la figura 5.2, y haciendo `LL` = `[L2]`, en donde `L2`= $(append/3, \{2\}, \{3\})$.

5. Se realiza una llamada recursiva `search([L2], S, [N2], OutTrans)`. Supongamos que `generate_node(L2, S, [L2], [N2], Node3, LL3)` tiene éxito y que genera el nodo `Node3` = `N3`, en donde `N3` = $((append/3, \{2\}, \{3\}), \{$`G1`, `G2`$\})$, en donde `G1` y `G2` son los grafos de dependencia de tamaños de la figura 5.1, y haciendo `LL3` = `nil`.

6. Finalmente, se realiza una llamada recursiva `search(nil, nil, [N3, N2], OutTrans)`, la cual tiene éxito haciendo `OutTrans` = `[N3, N2]`. Por tanto `Trans` = `(L, [N3, N2])`. □

La eficiencia del algoritmo top-down anterior se puede mejorar si se utiliza cierta información durante la generación de los nodos de transformación realizada por `generate_node`/3. En particular, el conocimiento de qué etiquetas asociadas con nodos de literal del nodo de transformación generado hacen más probable el fallo del predicado `generate_node`/3 cuando intente encontrar nodos de transformación para dichas etiquetas. Esto puede podar el espacio de búsqueda considerablemente. A veces es posible detectar dichas etiquetas examinando los hechos del programa. Por ejemplo, es posible detectar que `generate_node`/3 no encontrará un nodo de transformación para $(append/3, \emptyset, \{3\})$, puesto que, examinando el hecho que aparece en la definición de `append`/3, podemos inferir que en la llamada, al menos se necesita suministrar el tamaño del segundo argumento de `append`/3. Por tanto, no se generará ningún nodo de transformación con la etiqueta $(append/3, \emptyset, \{3\})$ asociada a un nodo de literal. Hemos implementado un prototipo en Prolog siguiendo las ideas presentadas utilizando las capacidades de Prolog para realizar una búsqueda dirigida por el objetivo (top-down). Nótese que nuestro algoritmo de transformación se puede clasificar dentro del tipo "reglas + estrategias" – véase [PP94] y sus referencias bibliográficas – y por tanto, se puede describir en términos de la aplicación de ciertas reglas de "folding" y "unfolding" en un orden determinado. De hecho, lo que expresa nuestro algoritmo es una "estrategia" particular para encontrar transformaciones óptimas, en el sentido de que, si existen varias transformaciones aplicables, construye aquellas que tienen un menor coste en tiempo de ejecución, basándose en el criterio de elegir aquellas que recorren menos datos y realizan menos operaciones aritméticas. Esto es útil por razones de implementación dado que evita el tener que implementar un evaluador parcial completo. En algunos casos se pueden aplicar transformaciones similares a las que proponemos añadiendo al programa original algún código fuente que realiza el cálculo de tamaños de forma simple, y aplicar después una estrategia de transformación de propósito general (p.ej. evaluando parcialmente un predicado "length/2" en un bucle recursivo anterior). Sin embargo, la necesidad de nuestro algoritmo estriba en el hecho de que las estrategias de propósito general utilizadas en sistemas de transformación de programas son menos potentes para *esta aplicación de cálculo de tamaños en particular* que nuestro algoritmo, en el sentido de que una estrategia general no nos asegura que obtengamos una transformación en algunos casos, lo que sí que hace nuestro algoritmo, y, además, tampoco nos asegura que las transformaciones encontradas son irreducibles. Nótese, por ejemplo, que existen ciertas transformaciones que se basan en detectar que deben de conocerse los tamaños de algunos términos que se usan como punto de partida para otros cálculos de tamaños. Esto sólo se puede conseguir razonando a nivel de estrategia.

5.6. Ventajas de la transformación de predicados para calcular tamaños de términos

Como mencionamos al principio del capítulo, el enfoque que nosotros denominamos estándar para calcular tamaños de datos es introducir nuevas llamadas a predicados que los calculan explícitamente. Por ejemplo, podemos utilizar el predicado predefinido de Prolog `length`/2 para calcular longitudes de listas o usar otros predicados predefinidos similares. Sin embargo, ésto implica un coste asociado que incluye el tener que recorrer partes significativas de los términos (a menudo los términos enteros) y realizar un proceso de cuenta o incremento. Como resultado de las transformaciones que proponemos se obtienen programas que, aunque realizan el proceso de cuenta, evitan el tener que volver a recorrer los términos, ya que este recorrido ya está implícito en los recorridos realizados por predicados que ya existían en el programa. Además, el proceso de cálculo se evita siempre que sea posible.

Como una ventaja adicional podemos observar que un predicado transformado puede recorrer menos o datos más pequeños que un predicado predefinido. Consideremos, por ejemplo, el predicado `p`/2 definido de la forma siguiente:

```
p([],[]).
p([X|Y],[X,X,X|Y1]):- p(Y,Y1).
```

Si tenemos el objetivo `(p(X,Y),...)`, en el cual el primer argumento de `p`/1 está totalmente instanciado y el segundo sin instanciar, y necesitamos conocer la longitud del segundo argumento después de su ejecución, el enfoque estándar incluiría una llamada a `length`/2 de la forma siguiente:`(p(X,Y),length(Y,L),...)`. En este caso `length(Y,L)` tiene que recorrer una lista tres veces más grande que `X`. Sin embargo, si transformamos `p`/2:

```
p2o1i([],[],0).
p2o1i([X|Y],[X,X,X|Y1],S):- p2o1i(Y,Y1,Sb), S is Sb + 3.
```

y llamamos a: `p2o1i(X,Y,L)` para calcular `L`, el número de sumas realizadas es igual a la longitud de `X`, es decir, la tercera parte de las realizadas con la anterior solución. Por supuesto, la situación inversa también podría darse, pero en cualquier caso, el programa ya realizaba el recorrido del término.

Consideremos otro caso – supongamos que tenemos el objetivo:

```
q(X), append(Y,X,Z), append(W,X,K)
```

donde `X`, `Y` y `W` son listas totalmente instanciadas, y `Z` y `K` son variables libres, las cuales quedarán totalmente instanciadas cuando el objetivo tenga éxito. Supongamos además que estamos interesados en conocer las longitudes de `Z` y `K` después de la ejecución del objetivo. Mediante el enfoque estándar podríamos tener:

```
q(X), append(Y,X,Z), append(W,X,K), length(Z,LZ),
length(K,LK)
```

Mientras que con la técnica de transformación de predicados tendríamos:

```
q1o(X,SX), append3o2i(Y,X,Z,SX,SZ),
append3o2i(W,X,K,SX,SK)
```

en donde `q1o(X,SX)` calcula la longitud de `X` (`SX`), la cual utiliza `append3o2i`/5 para calcular las longitudes de `Z` y `K` (`SZ` y `SK`). En este caso, la suma de las longitudes de los datos recorridos, que es igual al número de operaciones realizadas para calcular estas longitudes es: $length(X) + length(Y) + length(W)$. Mientras que con el enfoque estándar (primer caso) tendríamos: $length(Z) + length(K)$, pero como: $length(Z) = length(X) + length(Y)$, y $length(K) = length(X) + length(W)$ tendríamos: $2 * length(X) + length(Y) + length(W)$

Puede que pensemos que una mejor solución al primer enfoque podría ser

```
q(X), append(Y,X,Z), append(W,X,K), length(X,SX),
length(Y,SY), length(W,SW), SZ is SX + SY, SK is SX + SW
```

pero en este caso es necesario analizar el programa para inferir que la longitud del tercer argumento de `append`/3 es la suma de sus dos primeros. Esto puede ser fácil en algunos casos, como por ejemplo para `append`/3, pero en otros puede ser difícil o imposible. Tal es el caso en el que hay listas que no sólo dependen de las longitudes de otras listas, sino también de sus contenidos. En cualquier caso, nuestra técnica también podría obtener ventaja de esta clase de optimizaciones, en el caso de que se detecten.

Por otra parte, en algunos casos el cálculo de tamaños mediante transformación de programas puede ser más caro que con el enfoque estándar. Estos casos pueden aparecer cuando se realiza vuelta atrás – si un predicado que ha sido transformado para realizar cálculo de tamaños falla y realiza vuelta atrás con mucha frecuencia puede que sea mejor calcular los tamaños de los términos a posteriori una sola vez utilizando el enfoque estándar. Además, es posible que se realicen transformaciones de programas que realizan cálculos redundantes de tamaños de términos.

5.7. Resultados experimentales

Hemos realizado una serie de experimentos, utilizando SICStus PROLOG corriendo en una estación de trabajo SUN IPC, para medir la ganancia obtenida mediante la técnica de transformación de programas, con respecto a lo que nosotros denominamos el enfoque estándar para calcular tamaños de términos. En teoría esta ganancia puede ser del 100 %. Para medir la ganancia en la práctica hemos elegido algunos programas de prueba los cuales pensamos que son representativos de los casos peores o típicos, para tener así una cierta idea de la ganancia que se

programa	$\mathbf{T_{wsc}}$	$\mathbf{T_{st}}$	$\mathbf{T_{pt}}$	$\mathbf{T_{st} - T_{wsc}}$	$\mathbf{T_{pt} - T_{wsc}}$	**ganancia**
c/2	202.90	405.69	277.99	202.79	75.09	63.0 %
qsort/2	1218.00	1495.00	1343.90	277.00	125.90	55.3 %
q/2	52.59	90.20	61.69	37.61	9.10	76.7 %
deriv/2	119.00	3349.00	239.00	3110.00	120.00	92.9 %

Cuadro 5.1: Tiempos de ejecución (ms) para programas de prueba.

puede obtener en la práctica. En la tabla 5.1 se muestran los tiempos de ejecución obtenidos en los experimentos con estos programas de prueba. $\mathbf{T_{wsc}}$ representa el tiempo de ejecución del programa de prueba sin el cálculo de tamaños de términos. $\mathbf{T_{st}}$ es el tiempo de ejecución de este cálculo de tamaños mediante el enfoque estándar. $\mathbf{T_{pt}}$ representa el tiempo de ejecución mediante la técnica de transformación de programas. $\mathbf{T_{st} - T_{wsc}}$ y $\mathbf{T_{pt} - T_{wsc}}$ son los costes debidos al cálculo de tamaños con los enfoques estándar y de transformación de predicados respectivamente. En la última columna se muestra la ganancia obtenida en el proceso de cálculo de tamaños utilizando la técnica de transformación de programas, con respecto al enfoque estándar. Esta ganancia se ha calculado de acuerdo con la siguiente expresión: $ganancia = \frac{(\mathbf{T_{st}} - \mathbf{T_{wsc}}) - (\mathbf{T_2} - \mathbf{T_{wsc}})}{\mathbf{T_{st}} - \mathbf{T_{wsc}}} 100$

El primer programa de prueba seleccionado es el predicado `c/2`, que representa el caso estándar de un simple recorrido de una lista:

```
c([],[]).
c([X|Y],[X|Y1]) :- c(Y,Y1).
```

Se supone que dicho predicado se llama con el primer argumento instanciado, y el segundo sin instanciar, es decir, con una variable libre, y que se desea calcular la longitud del término al que queda instanciado dicho argumento una vez que el objetivo `c(X, Y)` tiene éxito. Para realizar este cálculo se ha utilizado la siguiente transformación, la cual preserva la recursión "por la cola" mediante el uso de un parámetro de acumulación:

```
trc([],[],S,S).
trc([X|Y],[X|Y1],S1,S) :- S2 is S1 + 1, trc(Y,Y1,S2,S).
```

Para medir el tiempo de ejecución con el enfoque estándar se ha utilizado la siguiente definición de `length`/3, la cual utiliza el predicado predefinido `is`/2, de forma que este tiempo de ejecución sea comparable al de `trc`/4:

```
length([], I, I).
length([_|L], I0, I) :- I1 is I0 + 1, length(L, I1, I).
```

Es de destacar que, aunque se podría utilizar otra definición de `length`/3 más eficiente, mediante el uso de otros predicados predefinidos más eficientes que `is`/2, siempre se puede hacer lo propio en la técnica de transformación de predicados, de modo que se utilicen predicados predefinidos especiales. En este experimento, $\mathbf{T_{wsc}}$ es el tiempo de ejecución del objetivo `c(X, Y)`, $\mathbf{T_{st}}$ es el de `c(X, Y), length(Y, 0, L)` y $\mathbf{T_{pt}}$ el de `trc(X, Y, 0, L)`.

El segundo programa de prueba es el predicado `qsort`/2, en el que se calculan las longitudes de las dos listas de salida de `partition`/4 . Aquí, $\mathbf{T_{wsc}}$ es el tiempo de ejecución del objetivo `qsort(X, Y)`, $\mathbf{T_{st}}$ el de `lqsort(X, Y)`, en donde `lqsort`/2 se define de la forma:

```
lqsort([],[]).
lqsort([First|L1],L2) :-
        partition(First,L1,Ls,Lg),
        length(Ls,0,Lssize),length(Lg,0,Lgsize),
        lqsort(Ls,Ls2), lqsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).
```

y $\mathbf{T_{pt}}$ es el tiempo de ejecución del objetivo `trqsort(X, Y)`, en donde `trqsort`/2 se define:

```
trqsort([],[]).
trqsort([First|L1],L2) :-
        trpartition(First,L1,Ls,Lg,0,Lssize,0,Lgsize),
        trqsort(Ls,Ls2), trqsort(Lg,Lg2),
        append(Ls2,[First|Lg2],L2).


trpartition(F,[],[],[],S1,S1,S2,S2).
trpartition(F,[X|Y],Y1,[X|Y2],ISize1,OSize1,ISize2,OSize2):-
        X > F, !,
        ISize3 is ISize2 + 1,
        trpartition(F,Y,Y1,Y2,ISize1,OSize1,ISize3,OSize2).
trpartition(F,[X|Y],[X|Y1],Y2,ISize1,OSize1,ISize2,OSize2) :-
        ISize3 is ISize1 + 1,
        trpartition(F,Y,Y1,Y2,ISize3,OSize1,ISize2,OSize2).
```

Este cálculo de tamaños es útil cuando se quiere transformar el predicado `qsort`/2 para que realice control de granularidad (véase el capítulo 2).

El tercer programa de prueba es el predicado `q`/2 definido de la forma siguiente:

```
q([], []).
q([X|Y], [X,X|Y1]) :- X > 7, !, q(Y, Y1).
q([X|Y], [X,X,X|Y1]) :- X =< 7, q(Y, Y1).
```

En este caso $\mathbf{T_{wsc}}$ es el tiempo de ejecución del objetivo `q(X, Y)`, $\mathbf{T_{st}}$ el de `q(X, Y), length(Y, 0, L)` y $\mathbf{T_{pt}}$ corresponde a `trq(X, Y, 0, L)`, en donde `trq`/2 se define:

```
trq([], [], S, S).
trq([X|Y], [X,X|Y1], S1, S) :-
    X > 7, !,
    S2 is S1 + 2,
    trq(Y, Y1, S2, S).
trq([X|Y], [X,X,X|Y1], S1, S) :-
    X =< 7,
    S2 is S1 + 3,
    trq(Y, Y1, S2, S).
```

Estos tiempos se ha tomado en ejecuciones con listas de entrada de distinta longitud. La ganancia obtenida es constante en cada uno de ellos, es decir, no depende de la longitud de la lista de entrada.

Finalmente, el cuarto programa de prueba es el predicado `deriv`/2, que calcula la derivada simbólica de una expresión (por brevedad, no incluimos su definición). En este caso la ganancia obtenida es 92,9 %. Esto es debido al hecho de que el tamaño del dato de salida es mucho más grande que el de entrada (39599 frente a 4607). En este caso no es posible inferir una expresión que dé el tamaño del dato de salida en función del de entrada, de modo que no hay forma de poder evitar el recorrido del dato de salida si se utiliza el enfoque estándar.

5.8. Conclusiones del capítulo

Hemos desarrollado un método para transformar predicados de forma que éstos calculen en tiempo de ejecución los tamaños de términos. Hemos presentado una forma sistemática para determinar si se puede transformar un programa para calcular los tamaños de unos términos determinados en un punto determinado del programa sin tener que realizar un recorrido adicional de los mismos. Además, si existen varias transformaciones posibles, nuestro enfoque permite encontrar transformaciones minimales de acuerdo con algún criterio. Hemos comentado las ventajas de la utilización de dicha transformación y mostrado algunos resultados experimentales. También hemos desarrollado, implementado, e integrado en el sistema `ciaopp` un algoritmo para encontrar transformaciones irreducibles.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Hemos desarrollado completamente (e integrado en el sistema `ciaopp` [HBPLG99, HBC$^+$99]) un sistema automático de control de granularidad para programas lógicos basado en el esquema de análisis y transformación de programas, en el cual se realiza tanto trabajo como sea posible en tiempo de compilación. En la realización de la transformación de programas perseguimos el objetivo de minimizar el trabajo adicional hecho en tiempo de ejecución, por lo que hemos propuesto una serie de técnicas, mientras que en el análisis el objetivo es obtener la información necesaria para esta fase de transformación, lo que nos ha llevado a desarrollar varios tipos de análisis capaces de inferir informaciones tales como cotas (principalmente inferiores) del coste de procedimientos, qué llamadas no fallarán, etc.

Hemos discutido los muchos problemas que surgen en la realización del control de granularidad (para los casos en los que se dispone de información de cotas inferiores y superiores de coste) y dado soluciones a los mismos con la suficiente generalidad para que los resultados obtenidos puedan aplicarse a otros sistemas, no necesariamente de programación lógica y para diferentes modelos de ejecución. Creemos por tanto que los resultados presentados son de interés para trabajos similares en otros lenguajes de programación paralelos.

Que nosotros sepamos, nuestro trabajo es el primero y único existente que describe con la generalidad que nosotros lo hacemos (e implementa) un sistema completo y totalmente automático de control de granularidad para programas lógicos.

Además hemos hecho una evaluación de las técnicas de control de granularidad desarrolladas para los casos de paralelismo conjuntivo y disyuntivo en los sistemas &-Prolog y Muse respectivamente, y hemos obtenido unos resultados esperanzadores.

De los resultados experimentales se desprende que no es esencial obtener el mejor tamaño de grano para un problema en concreto, sino que existe una cierta holgura en cuanto a la precisión con la que se calcula el mismo. Esto nos sugiere que los compiladores podrían realizar un control de granularidad automático que sea útil en la práctica.

Podemos concluir que el análisis/control de granularidad es una técnica particularmente prometedora ya que tiene el potencial de hacer posible la explotación automática de arquitecturas paralelas, tales como estaciones de trabajo en una red de área local (posiblemente de alta velocidad).

Algunas técnicas que hemos desarrollado para su aplicación al control de granularidad (que es la principal motivación de esta tesis) tienen además otras aplicaciones importantes, por ejemplo, la información de no-fallo es muy útil para la eliminación de paralelismo especulativo, detección de errores de programación y transformación de programas. La técnica de transformación de programas para que calculen los tamaños de algunos datos "sobre la marcha" es útil en aplicaciones relacionadas con optimización de programas, como por ejemplo eliminación de recursividad y selección de diferentes algoritmos o reglas de control cuya eficiencia puede depender de tales tamaños. La información sobre los costes de tiempos de ejecución puede ser útil para una gran variedad de aplicaciones, entre las cuales se incluyen ayudar a los sistemas de transformación de programas a elegir las transformaciones óptimas, elección entre distintos algoritmos, depuración de eficiencia (optimización) de programas y la optimización de consultas en bases de datos deductivas. Aparte de las mencionadas aplicaciones de la información de coste, el problema del análisis de coste puede ser de interés para investigadores de otras clases de análisis estáticos de programas lógicos (modos de llamada, tipos, etc.) por dos motivos. El primero es que cualquier mejora de estos análisis supone una mejora potencial del análisis de coste. El segundo motivo es la gran variedad de algoritmos de análisis combinatorio que aparecen, especialmente cuando se trabaja con resolución de restricciones.

6.2. Trabajo futuro

Entre las líneas de trabajo futuro que pensamos realizar podemos comentar el estudio de técnicas de asignación de tareas basadas en la combinación de información de granularidad y técnicas de gestión de tareas tales como las usadas en paralelismo de datos. Combinar las técnicas desarrolladas con técnicas de evaluación parcial para crear tareas con grano grueso (p.e. mediante operaciones de "unfolding", desdoblado de bucles, etc.). También pensamos extender las técnicas desarrolladas a lenguajes (concurrentes) de programación lógica con restricciones.

Otra área de investigación que resulta interesante es el análisis de coste medio. Para un gran número de aplicaciones resulta más interesante y apropiado el conocer el coste medio de una llamada a un procedimiento, y no el coste en el peor caso. Obviamente, el dar una definición aceptable de "coste medio" requiere definir una distribución de probabilidades de los posibles datos de entrada, tarea que no parece ser trivial en modo alguno. Sin embargo, se podría pensar en técnicas de obtención de "perfiles" para estimar las mencionadas distribuciones.

Finalmente, también pensamos evaluar el prototipo de control de granularidad con programas más grandes y reales, y aplicar las herramientas desarrolladas a la versión paralela de ECL^{i}PS^e ejecutándose en una red de estaciones de trabajo, y a la versión distribuida de `Ciao`.

Bibliografía

[AK90] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[AR94] L. Araujo and J.J. Ruz. PDP: Prolog Distributed Processor for Independent_AND/OR Parallel Execution of Prolog. In *Eleventh International Conference on Logic Programming*, pages 142–156, Cambridge, MA, June 1994. MIT Press.

[AR97] L. Araujo and J.J. Ruz. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming*, 33(1):49–79, October 1997.

[AS79] B. Aspvall and Y. Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. In *Proc. 20th ACM Symposium on Foundations of Computer Science*, pages 205–217, October 1979.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BCMH94] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on Log-*

ic Programming, pages 457–471, Ithaca, NY, November 1994. MIT Press.

[BG96] D. Basin and H. Ganzinger. Complexity Analysis based on Ordered Resolution. In *11th. IEEE Symposium on Logic in Computer Science*, 1996.

[BH89] B. Bjerner and S. Holmstrom. A Compositional Approach to Time Analysis of First Order Lazy Functional Programs. In *Proc. ACM Functional Programming Languages and Computer Architecture*, pages 157–165. ACM Press, 1989.

[BK96] F. Benoy and A. King. Inferring Argument Size Relationships with CLP(R). In *Proc. 6th International Workshop on Logic Program Synthesis and Transformation*, pages 34–153. Stockholm University/Royal Intitute of Technology, 1996.

[BM92] Jonas Barklund and Håkan Millroth. Providing iteration and concurrency in logic programs through bounded quantifications. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 817–824, ICOT, Japan, 1992. Association for Computing Machinery.

[BR86] P. Borgwardt and D. Rea. Distributed Semi-Intelligent Backtracking for a Stack-Based AND-Parallel Prolog. In *Symp. on Logic Prog.*, pages 211–222, 1986.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[BSY88] P. Biswas, S. Su, and D. Yun. A Scalable Abstract Machine Model to Support Limited-OR/Restricted AND Parallelism in Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 1160–1179, Seattle,Washington, 1988.

[CC94] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[CDD85] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[CH83] A. Ciepielewski and S. Haridi. A formal model for or-parallel execution of logic programs. In Mason, editor, *IFIP 83*, 1983.

[CL89] H. Comon and P. Lescanne. Equational Problems and Disunification. *J. Symbolic Computation*, 7(3/4):371–425, 1989.

[Col87] A. Colmerauer. Opening the Prolog-III Universe. In *BYTE Magazine*, August 1987.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[Deb89] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, 1989.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DL90] S.K. Debray and N.-W. Lin. Static Estimation of Query Sizes in Horn Programs. In *Third International Conference on Database Theory*, Lecture Notes in Computer Science 470, pages 515–528, Paris, France, December 1990. Springer-Verlag.

[DL93] S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[DLGH95] S.K. Debray, P. López-García, and M. Hermenegildo. Towards Cost Analysis of Divide-and-Conquer Logic Programs. Technical Report CLIP18/95.0, Facultad Informática UPM, Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, March 1995.

[DLGH96] S.K. Debray, P. López-García, and M. Hermenegildo. Towards Precise Non-Failure Analysis for Logic Programs. Technical Report TR CLIP18/96.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte Madrid-Spain, November 1996.

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[DLGHL97] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 Interna-*

tional Logic Programming Symposium, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[DLH90] S.K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[DZ92] P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[Fag87] B. S. Fagin. *A Parallel Execution Model for Prolog*. PhD thesis, The University of California at Berkeley, November 1987. Technical Report UCB/CSD 87/380.

[FSZ91] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic Average-Case Analysis of Algorithms. *Theor. Comp. Sci.*, (79):37–109, 1991.

[Gal97] M. M. Gallardo. *Análisis de Lenguajes Lógicos Concurrentes Mediante Interpretación Abstracta*. PhD thesis, University of Málaga, November 1997.

[GDL95] R. Giacobazzi, S.K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *Journal of Logic Programming*, 25(3):191–248, 1995.

[GdW94] J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[GH85] B. Goldberg and P. Hudak. Serial Combinators: Optimal Grains of Parallelism. In *Proc. Functional Programming Languages and Computer Architecture*, number 201 in LNCS, pages 382–399. Springer-Verlag, Aug 1985.

[GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[GT94] M. M. Gallardo and J. M. Troya. Granularity Analysis of Concurrent Logic Languages based on Abstract Interpretation. In Alpuente, Barbuti, and Ramos, editors, *GULP-ProDe'94*, volume 2, pages 342–356. Universidad Politécnica de Valencia, September 1994.

[GT95] M. M. Gallardo and J. M. Troya. Studying the Cost of Logic Languages in an Abstract Interpretation Framework for Granularity Analysis. In Logic Program Synthesis and Transformation. *Pre-Proceedings of LOPSTR'95*, Utrecht, Netherlands, September 1995. Extended version as Technical Report CW 216, K.U. Leuven.

[Hau90] B. Hausman. Handling speculative work in or-parallel prolog: Evaluation results. In *North American Conference on Logic Programming*, pages 721–736, Austin, TX, October 1990.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[Her86] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, U. of Texas at Austin, August 1986.

[HG91] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[HLA94] L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[HSC96] F. Henderson, Z. Somogyi, and T. Conway. Determinism analysis in the mercury compiler. In *Proc. Australian Computer Science Conference*, Melbourne, Australia, January 1996.

[Hua85] C.H. Huang. An interpreter of restricted and-parallelism for prolog programs. MCC AI Note, 1985.

[Hue93] L. Huelsbergen. Dynamic Language Parallelization. Technical Report 1178, Computer Science Dept. Univ. of Wisconsin, September 1993.

[JB92] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

[JM81] Neil D. Jones and Steven S. Muchnick. *Program Flow Analysis: Theory and Applications*, chapter Complexity of Flow Analysis, Inductive Assertion Synthesis, and a Language due to Dijkstra, pages 380–393. Prentice-Hall, 1981.

[Kal87] L. V. Kalé. Completeness and Full Parallelism of Parallel Logic Programming Schemes. In *Fourth IEEE Symposium on Logic Programming*, pages 125–133. IEEE, 1987.

[Kap88] S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.

[KL88] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.

[Kow74] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

[Kow79] Robert A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland Inc., 1979.

[Kow80] R. A. Kowalski. Logic as a computer language. *Proc. Infotec State of the Art Conference, Software Development: Management*, June 1980.

[KSB97] A. King, K. Shen, and F. Benoy. Lower-bound Time-complexity Analysis of Logic Programs. In *1997 International Logic Programming Symposium*, pages 261–275. MIT Press, Cambridge, MA, October 1997.

[Kun87] K. Kunen. Answer Sets and Negation as Failure. In *Proc. of the Fourth International Conference on Logic Programming*, pages 219–228, Melbourne, May 1987. MIT Press.

[LGH93] P. López-García and M. Hermenegildo. Towards Dynamic Term Size Computation via Program Transformation. In *Second Spanish Conference on Declarative Programming*, pages 73–93, Blanes, Spain, September 1993. IIIA/CSIC.

[LGH95] P. López-García and M. Hermenegildo. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming*, pages 647–661, Cambridge, MA, June 1995. MIT Press, Cambridge, MA.

[LGHD94] P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[LGHD96] P. López-García, M. Hermenegildo, and S.K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[LH85] G. J. Lipovski and M. Hermenegildo. B-LOG: A Branch and Bound Methodology for the Parallel Execution of Logic Programs. In *1985 IEEE International Conference on Parallel Processing*, pages 560–568. IEEE Computer Society, August 1985.

[Lin88] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[LK88] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

[LM87] J.-L. Lassez and K. Marriott. Explicit Representation of Terms Defined by Counter Examples. *Journal of Automated Reasoning*, 1987.

[LMM88] J.-L. Lassez, M. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–626. Morgan Kaufman, 1988.

[LMM91] J.-L. Lassez, M. Maher, and K. Marriott. Elimination of Negation in Term Algebras. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science*, number 520 in Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1991.

[Lus88] E. Lusk et al. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*. Tokyo, November 1988.

[Mat70] J. V. Matijasevič. Enumerable sets are diophantine. 191:279–282, 1970. English translation in *Soviet Mathematics—Doklady*, 11 (1970), 354-357.

[Met88] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

[MG89] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32, 1989.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[Mil91] Håkan Millroth. Reforming compilation of logic programs. In V. Saraswat and K. Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 485–502, San Diego, USA, 1991. The MIT Press.

[Mug90] S.H. Muggleton. Inductive Logic Programming. In *Proc. First Conference on Algorithmic Learning Theory*, 1990.

[PK88] V. Penner and A. Klinger. AND-Parallel PROLOG on Large Scale Transputer-Systems. Internal report, Institute for Applied Mathematics at the Aachen Technical University, West Germany, 1988.

[PP94] A. Pettorossi and M. Proietti. Transformations of Logic Programs: Foundations and Techniques. *Journal of Logic Programming, Special Issue: Ten Years of Logic Programming*, 19/20, May/July 1994.

[Pug92] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[PW93] W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. Technical report, Dept. of Computer Science, Univ. of Maryland, December 1993.

[Qui90] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[RM90] F. A. Rabhi and G. A. Manson. Using Complexity Functions to Control Parallelism in Functional Programs. Res. Rep. CS-90-1, Dept. of Computer Science, Univ. of Sheffield, England, January 1990.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Rob65] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[Ros89] M. Rosendhal. Automatic Complexity Analysis. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM, New York, 1989.

[Sar89] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Pitman, London, (1989).

[SCK98] K. Shen, V.S. Costa, and A. King. Distance: a New Metric for Controlling Granularity for Parallel Execution. In Joxan Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 85–99, Cambridge, MA, June 1998. MIT Press, Cambridge, MA.

[Sze89] P. Szeredi. Performance Analysis of the Aurora Or-Parallel Prolog System. In *1989 North American Conference on Logic Programming.* MIT Press, October 1989.

[UV88] J.D. Ullman and A. Van Gelder. Efficient Tests for Top-Down Termination of Logical Rules. *Journal ACM*, 35(2):345–373, 1988.

[vEK76] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23:733–742, October 1976.

[VS92] K. Verschaetse and D. De Schreye. Derivation of Linear Size Relations by Abstract Interpretation. In *Fourth International Symposium PLILP'92, Programming Language Implementation and Logic Programming*, pages 296–310, Leuven, Belgium, August 1992. Springer Verlag.

[Wad88] P. Wadler. Strictness analysis aids time analysis. In *Proc. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 1988.

[War87a] D.H.D. Warren. OR-Parallel Execution Models of Prolog. In *Proceedings of TAPSOFT '87*, Lecture Notes in Computer Science. Springer-Verlag, March 1987.

[War87b] D.H.D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *International Symposium on Logic Programming*, pages 92–102. San Francisco, IEEE Computer Society, August 1987.

[WR87] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.

[WW88] W. Winsborough and A. Waern. Transparent And-Parallelism in the Presence of Shared Free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, Seattle,Washington, 1988.

[ZTD+92] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference* *on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.

[ZZ89] P. Zimmermann and W. Zimmermann. The Automatic Complexity Analysis of Divide-and-Conquer Programs. Res. Rep. 1149, INRIA, France, Dec. 1989.