UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Un sistema de Programación Lógica extensible y con soporte para análisis global

(An Extensible, Global Analysis Friendly Logic Programming System)

Tesis Doctoral

DANIEL CABEZA GRAS

Agosto, 2004

Tesis Doctoral

Un sistema de Programación Lógica extensible y con soporte para análisis global

presentada en la Facultad de Informática
de la Universidad Politécnica de Madrid
para la obtención del título de
Doctor en Informática

Candidato: **Daniel Cabeza Gras**
Licenciado en Informática
Universidad Politécnica de Madrid

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

Madrid, Agosto, 2004

A mis padres, Ana y Jaime

Agradecimientos

Quiero expresar aquí mi gratitud a todas las personas que de una forma u otra me han ayudado a llevar esta tesis a buen término. En primer lugar, a mi padre y a mi madre, sin los cuales no la habría ni empezado. Estoy muy agradecido a Manuel Hermenegildo, mi director de tesis, por el asesoramiento y aliento que me ha brindado todos estos años. Mi gratitud también para mis compañeros del grupo CLIP, pasados y presentes, entre los cuales mencionaré a los veteranos Francisco Bueno, Manuel Carro, Pedro López y Germán Puebla, y al reciente fichaje Elvira Albert (tenía que hacerlo `:)`). Finalmente, quiero dar las gracias a toda la gente que me ha dado su afecto en este tiempo, en particular mencionaré a Beatriz Martín del Campo, debido a lo que renunció por esta tesis.

El sistema Ciao es un esfuerzo en colaboración e incluye contribuciones de miembros de diversas instituciones, incluyendo UPM, SICS, U.Melbourne, Monash U., U.Arizona, Linköping U., NMSU, K.U.Leuven, Bristol U., y Ben-Gurion U. La documentación del sistema y las publicaciones relacionadas contienen créditos más específicos.

La investigación incluida en esta tesis ha sido parcialmente financiada por una beca de Formación de Personal Investigador del Ministerio de Educación y Ciencia español y por los proyectos de investigación españoles y de la Unión Europea "ParForCE" (ESPRIT BR 6707 / TIC93-0976-CE), "ACCLAIM" (ESPRIT BR 7195 / TIC93-0975-CE), "IPL-D" (CICYT TIC 93-0737-C02-01), "ELLA" (CICYT TIC 96-1012-C02-01), "DiSCiPl" (ESPRIT LTR 22532/CICYT TIC 97-1640-CE), "ECCOSIC" (programa de intercambio Fulbright EE.UU.-España), y "EDIPIA" (MCYT TIC 99-1151).

Sinopsis

El objetivo de esta tesis doctoral es diseñar y desarrollar un sistema de programación multi-paradigma de nueva generación, que esté basado en un núcleo de Programación Lógica y ofrezca soporte al análisis global.

El paradigma de la Programación Lógica se ha demostrado particularmente útil para el desarrollo de aplicaciones complejas, como las que aparecen en el campo de la Inteligencia Artificial, incluyendo sistemas basados en el conocimiento, agentes inteligentes, sistemas expertos, etc.

El área del análisis global de programas (generalmente basado en la interpretación abstracta) ha experimentado grandes progresos en el contexto de la Programación Lógica, usado para inferir información en tiempo de compilación sobre el comportamiento de los programas en ejecución. La información inferida se ha mostrado muy provechosa no sólo para producir código optimizado, sino también en verificación de programas y en detección y diagnosis de errores.

Sin embargo, mientras que el análisis global de programas es en este momento relativamente bien entendido desde un punto de vista teórico, se ha prestado escasa atención al desarrollo de un sistema práctico de programación lógica (que al menos proporcione una implementación completa y modular de Prolog) que incorpore dicho análisis. Esta tesis rellena este vacío proponiendo un diseño del lenguaje y una implementación que hace el análisis global práctico y escalable.

Otro objetivo del sistema desarrollado es servir como banco de trabajo experimental para desarrollar nuevas extensiones a la Programación Lógica. El sistema incluye técnicas que permiten extender el lenguaje núcleo de una manera muy flexible (y al nivel del fuente), lo que ha sido utilizado para desarrollar y probar numerosas extensiones incluyendo funciones, objetos, restricciones, registros, persistencia, ejecución distribuida, y otras.

El sistema producido (Ciao) se ha hecho libremente disponible a la comunidad y tiene actualmente un número importante de usuarios.

Resumen*

El campo de la *Inteligencia Artificial* [MMRS55, RN95] se caracteriza por sus requerimientos de procesamiento simbólico, representación del conocimiento, búsqueda en espacios de estados, y otras tareas de alto nivel de similar complejidad. El paradigma de la *Programación Lógica* [Kow74, Kow80, Col87] ha sido intensivamente aplicado desde sus orígenes en el contexto de la Inteligencia Artificial, debido a su adecuación a los requerimientos que ésta presenta, para la implementación de las aplicaciones usuales en este campo, como sistemas expertos, bases de conocimiento, agentes inteligentes, etc. Además, la relativamente reciente extensión de la Programación Lógica, llamada *Programación Lógica Basada en Restricciones* [JM94], facilita enormemente la implementación y ejecución eficiente de cierto tipo de tareas que son también muy relevantes en Inteligencia Artificial, como la planificación, optimización, etc. Sin embargo, la ventaja del mayor nivel de abstracción de la Programación Lógica con respecto al paradigma tradicional de la programación imperativa tiene como contrapartida una mayor separación con respecto al lenguaje interno del computador, acarreando en principio una menor eficiencia en términos de velocidad de ejecución y uso de memoria. Por otra parte, aunque programar a un nivel más alto ayuda enormemente al proceso de codificación, debido a la naturaleza compleja de los problemas a los que se enfrenta la Inteligencia Artificial, se hace deseable el desarrollo de técnicas para ayudar al programador en la verificación y depuración de su código. La técnica del análisis global de programas, que consiste en inferir en tiempo de compilación

*Este resumen de la Tesis Doctoral, presentada en lengua inglesa para su defensa ante un tribunal internacional, es preceptivo según la normativa de doctorado vigente en la Universidad Politécnica de Madrid

información sobre el comportamiento del programa en su ejecución, ofrece ayuda para solventar estos dos problemas. Por una parte, la información recolectada puede ser usada por el compilador del lenguaje de alto nivel para producir código optimizado, especializando la traducción de cada construcción del lenguaje a cada caso particular, y también para permitir el uso de características avanzadas del *hardware*, como el procesamiento paralelo. Por otra parte, es evidente que dicha información es vital en la verificación del programa y en la detección y diagnosis de errores.

Estas premisas nos llevaron a enfocar este trabajo en el paradigma de programación de la Programación Lógica, con el fin de hacer progresar el desarrollo en un tiempo razonable de programas eficientes y correctos. En todo caso, muchas de las ideas presentadas aquí son generales y por tanto aplicables también a otros paradigmas de programación. Además, el trabajo que realizamos para esta Tesis será igualmente útil para el desarrollo de aplicaciones que no sean específicas de la Inteligencia Artificial.

Al comienzo del trabajo de esta Tesis, existían un número importante de sistemas de Programación Lógica, desde experimentales o no convencionales [AK93, HL94] a consolidados y comerciales [Swe95, Qui86] (en general ofreciendo el lenguaje Prolog, con ligeras variaciones). Por otra parte, la investigación sobre análisis global de programas en el contexto de la Programación Lógica (generalmente basada en la interpretación abstracta [CC77]) había recibido también una gran atención desde el punto de vista teórico. Sin embargo, en nuestra opinión no se había prestado suficiente atención al desarrollo de un sistema práctico de programación lógica estándar diseñado desde el principio para ofrecer un soporte adecuado al análisis global de programas, de manera que programas reales pudieran ser analizados, optimizados y verificados de una manera precisa e integrada.

Esta situación nos llevó a trabajar en aras a la realización de este objetivo. El trabajo resultado de esta Tesis se ha aplicado al diseño e implementación del sistema "Ciao" de Programación Lógica de nueva generación. Ciao se ha desarrollado desde el principio para ser un sistema práctico, eficiente, apropiado tanto para programar a pequeña escala como para grandes proyectos, y sobretodo con soporte para el análisis, la optimización y la verificación globales. Además, con

el objetivo de que sea un banco de trabajo experimental para desarrollos futuros, se ha diseñado para ser altamente extensible y reconfigurable. En nuestra opinión, las propuestas actuales de lenguajes y sistemas de programación lógica son realmente "paquetes cerrados", en el sentido que ofrecen una solución combinada que afecta un número de cuestiones, muchas de ellas absolutamente ortogonales. Nuestra filosofía es dar siempre la posibilidad de explorar tecnologías y modelos novedosos mediante la redefinición modular del sistema o lenguaje. En ese sentido, Ciao es un sistema que puede evolucionar, basado en el diseño de un núcleo simple pero poderoso al cual se pueden añadir características de una manera modular.

Objetivos de la Tesis

El objetivo principal del trabajo presentado en esta Tesis Doctoral ha sido el diseño y desarrollo de un sistema de Programación Lógica de nueva generación, extensible, de alto nivel y con soporte para el análisis global en sus aplicaciones a la optimización de código y la verificación, de utilidad general pero enfocado especialmente a la resolución de problemas relacionados con el campo de la Inteligencia Artificial. Un objetivo relacionado ha sido el desarrollo de técnicas para el uso mejorado de la información obtenida mediante el análisis global de programas en la explotación del paralelismo en la Programación Lógica.

Estructura del trabajo

La Tesis se estructura en tres partes. Estas partes se refieren a los temas siguientes: diseño del lenguaje, implementación del sistema y aplicaciones del análisis global (paralelización de programas lógicos). A continuación se describe cada una de las partes en detalle.

Parte I: Diseño del lenguaje

Un diseño adecuado del lenguaje es fundamental para el logro de los objetivos perseguidos por esta tesis: extensibilidad, soporte al análisis global, detección mejorada de errores y verificación y depuración avanzadas de programas.

Nos hemos concentrado en el lenguaje de programación lógica más ampliamente usado, Prolog, como el estándar de facto (y actualmente oficial) para la mayoría de las implementaciones del paradigma de la programación lógica. La investigación se enfocó primeramente a estudiar las características del lenguaje que planteaban problemas al análisis global de programas, fundamentalmente para proporcionar métodos que solventen dichos problemas, pero además y como resultado se adquirió el conocimiento necesario para elaborar alternativas o mejoras a esas características para evitar los problemas encontrados.

Después el trabajo se enfocó en el diseño de un sistema de módulos para Prolog, que proporcionase los objetivos arriba planteados. Nótese que la modularidad, aparte de ser de gran ayuda al desarrollo y mantenimiento de los programas, es un requisito en la práctica para el análisis global de proyectos informáticos de tamaño medio a grande, dado que resulta poco factible, por los requerimientos en términos de tiempo de proceso y memoria usada, el análisis completo del código total del que se componen tales aplicaciones. Con un diseño apropiado del sistema de módulos y del procedimiento de compilación, se debe poder analizar y compilar un módulo por separado sin requerir el código fuente de los otros módulos, solamente su interfaz declarado. De esta manera, el análisis global se vuelve más manejable y el proceso de desarrollo se acelera. El sistema del módulos fue diseñado también para permitir fácilmente la extensión y adaptación del lenguaje en el nivel del código fuente (es decir en Prolog), haciendo posible entre otras cosas el cambio de los predicados predefinidos, extensiones a la sintaxis y extensiones semánticas, de una forma modular y estructurada. Algunas de las extensiones que hemos desarrollado usando las capacidades proporcionadas son los términos con características ("feature-terms"), la búsqueda en anchura, la programación lógica basada en restricciones sobre el dominio real o racional (que usa el potente concepto de variables con atributos), la programación orientada a objetos y la

programación funcional (incluyendo una sintaxis apropiada).

El "orden superior" es otra de las características que incrementa el poder expresivo del lenguaje. Aunque en Prolog hay tradicionalmente características rudimentarias para la meta-programación mediante el uso de los predicados `call/1` y `=../2`, que permiten emular el orden superior, nosotros hemos querido proporcionar métodos más expresivos y formales de usar el orden superior en la programación lógica estándar, modular y no tipada, y con una sintaxis más conveniente. Hemos desarrollado una implementación de orden superior para Ciao, basada en el predicado `call/N` [NP92, Nai96, MO84], que incluye "abstracciones de predicado" (la traducción a la Programación Lógica de las expresiones lambda de la Programación Funcional). Todas estas extensiones se pueden combinar obteniendo un efecto sinérgico. Por ejemplo, cuando la notación funcional se combina con las características de orden superior de Ciao, el lenguaje subsume en funcionalidad a un lenguaje funcional impaciente sin tipos (que pueden sin embargo utilizarse usando el lenguaje de aserciones y el preprocesador, mencionados más adelante).

A continuación exponemos los puntos más importantes de cada uno de los temas tratados

Análisis global de programas Prolog

El análisis global de programas lógicos basado en la interpretación abstracta es en la actualidad relativamente bien entendido desde el punto de vista de los marcos de trabajo generales y los dominios abstractos. Sin embargo, se ha prestado comparativamente poca atención a los problemas que surgen cuando se pretende analizar un dialecto completo y práctico del lenguaje Prolog, y solamente se habían propuesto hasta el momento algunas pocas soluciones a estos problemas. Las propuestas que había generalmente restringían de un modo u otro la clase de programas que podían analizarse. Este capítulo rellena este vacío considerando un dialecto completo de Prolog (esencialmente el estándar ISO), indicando los problemas que surgen en el análisis de dicho dialecto y proponiendo una combinación novedosa de soluciones conocidas y nuevas que en conjunto permiten el

análisis correcto de programas arbitrarios que usen todo el potencial del lenguaje.

En nuestra propuesta hemos tratado de complacer al mismo tiempo a dos tipos diferentes de usuario: primero, al usuario novato, que le gustaría que el análisis fuese lo más automático posible, y segundo, al usuario avanzado, que querría poder guiar al análisis en lugares difíciles para obtener resultados mejorados. Por tanto, para cada característica problemática ofrecemos tanto soluciones que no requieren intervención del usuario como otras que requieren proporcionar ayudas al proceso de análisis. Esto último requiere un interfaz claro con el analizador al nivel del lenguaje fuente, lo cual también es necesario para expresar la información recolectada por los diferentes análisis disponibles. Nuestra solución consiste en usar *anotaciones* en el programa, que no sólo son útiles como una forma de comunicación bidireccional entre el usuario y el compilador, sino que también permiten la cooperación entre las diferentes herramientas de análisis y la conexión entre los análisis y los otros módulos del compilador. Estas anotaciones (que en la actualidad llamamos *aserciones*) son construcciones sintácticas que permiten expresar propiedades de los programas.

Un problema fundamental que surge al analizar estáticamente programas Prolog es que no todo el código que en realidad va a ejecutarse está accesible directamente al análisis (debido al uso de código dinámico y/o meta-predicados). Esto puede ocurrir bien porque ciertas llamadas que ocurren en el programa se construyen dinámicamente, bien porque el código que define ciertos predicados se modifica dinámicamente. Lo cual produce los problemas siguientes:

- Cómo se calculan las substituciones abstractas de salida para las llamadas a código desconocido, llamamos a esto *problema de las substituciones de salida*.

- Cómo determinar las llamadas y las substituciones abstractas de entrada que pueden aparecer en el código desconocido, llamamos a esto *problema de llamadas extra*.

El primer problema puede solucionarse fácilmente usando substituciones abstractas de salida "supremas" con respecto a las sustituciones abstractas de entrada. Hemos definido una substitución abstracta de salida suprema con respecto a

una dada como la que recoge todas las posibilidades de modificación de la primera en una llamada.

Para análisis de programas independientes de las metas, esta solución es suficiente dado que el segundo problema no es aplicable. Pero para análisis dependientes de las metas, hemos de solucionar el segundo problema de alguna forma. La causa del problema es la imposibilidad de calcular estáticamente el sub-árbol que aparece bajo una llamada, bien porque esta llamada es desconocida (se usa `call/1`, por ejemplo), bien porque no todo el código que define el predicado es conocido (es un predicado dinámico). Por tanto, dado que en esos sub-árboles pueden darse nuevas llamadas (y nuevas substituciones de llamada), que afectarían a otras partes del programa, el análisis puede ofrecer resultados incorrectos.

Una primera solución obvia del problema de llamadas extra sería simplemente suponer que hay llamadas desconocidas y por tanto cualquier predicado del programa podrá ser invocado de cualquier manera posible. Esto significa que el análisis se puede realizar pero las substituciones de entrada de todos los predicados serán supremas, lo cual es similar a realizar un análisis independiente de las metas, que puede aún permitir algunas optimizaciones, pero posiblemente impedirá otras.

La segunda solución que proponemos es general y muy elegante, con la única desventaja de su coste en términos de tamaño del código. La idea clave es esencialmente compilar dos versiones del programa: una que es la compilación directa del programa original, y la otra que se analiza asumiendo que las únicas llamadas a cada predicado son las que aparecen explícitamente en el programa. Esta última versión podría incluir todas las optimizaciones que se pudieran realizar ignorando el efecto de las llamadas desconocidas. Y en la primera versión todavía se podrían incluir las optimizaciones posibles con un análisis independiente de las metas, o con uno dependiente de las metas con substituciones de entrada supremas. Las llamadas extra desconocidas que se pudieran producir en el código dinámico no entrarían en la versión más optimizada (que podría no estar preparada para esas llamadas) sino que irían a parar a la versión menos optimizada. Esto tendría lugar automáticamente porque en la versión menos optimizada se mantendrían los nombres originales de los predicados, mientras que en la versión más optimiza-

da los nombres de los predicados estáticos se renombrarían apropiadamente. Los términos que se construyan en tiempo de ejecución usarán los nombres originales de los predicados, de manera que se evita acceder a los predicados de la versión más optimizada desde las cláusulas incluidas dinámicamente. Por último, los puntos de entrada al código se renombran adecuadamente de modo que se accede inicialmente a las versiones optimizadas. Esta solución se puede aplicar tanto para el caso de meta-predicados como para predicados dinámicos, permitiendo optimizaciones máximas a programas dinámicos.

Como dijimos arriba, una parte importante de nuestras soluciones se basa en usar anotaciones en el programa. Estas anotaciones son aserciones sobre el programa que se incluyen como parte de su código, y que se refieren bien a puntos del programa dentro de una cláusula (por ejemplo, antes o después de la ejecución de una meta dada – el "nivel de meta"), bien a puntos que se refieren a un predicado completo (por ejemplo, a la entrada o a la salida de un predicado – el "nivel de predicado"). En cualquier nivel, las anotaciones describen propiedades de las variables que aparecen en el programa. Llamaremos a las descripciones de tales propiedades "declaraciones".

Una clase de anotaciones a nivel de predicado son las anotaciones "`entry`". Estas anotaciones expresan que en la ejecución se pueden producir llamadas a un cierto predicado con una substitución abstracta de entrada compatible con la dada. Una propiedad fundamental de estas anotaciones es que las que se refieren a un mismo predicado deben cubrir todas las llamadas que se pueden producir a este predicado desde el exterior del texto del programa/módulo (no hace falta que cubran las llamadas que ocurren explícitamente en el texto del programa/módulo). También asumiremos por ahora que estas anotaciones definen todos los puntos de entrada al programa o módulo.

Además de las más usuales anotaciones `entry`, proponemos también otra clase de anotaciones a nivel de predicado, las anotaciones "`trust`". Las declaraciones que incorporan anotaciones `trust` ponen en relación las substituciones de entrada y de salida de las llamadas a un predicado. Más concretamente, afirman que si una llamada a ese predicado tiene una substitución de entrada compatible con el patrón de entrada dado, entonces su substitución de salida será compati-

ble con el patrón de salida dado. Obsérvese que el patrón de entrada puede ser vacío, en ese caso se pueden afirmar propiedades de las substituciones de salida independientemente de las substituciones de entrada. Este tipo de anotaciones, a diferencia de las `entry`, no tiene porque ser exhaustivo con respecto a las posibles substituciones de entrada.

Uno de los usos más usuales de las anotaciones `trust` es para describir predicados que no aparecen definidos en el texto del programa, pero que se usan en el mismo, con la consiguiente mejora en los resultados del análisis. Además, para las anotaciones que describen predicados que sí aparecen en el programa, permiten también encontrar errores de programación, cuando las substituciones de salida calculadas por el análisis no son compatibles con las dadas en las anotaciones.

La anotación que proponemos a nivel de meta es la anotación pragma. Ésta se refiere al estado de las variables en una cláusula en el preciso punto en donde aparece: a continuación de la cabeza, entre dos literales, o después del último literal de la cláusula. La anotación se produce introduciendo un literal extra especial, que incluye toda la información necesaria, y puede referirse a cualquiera de las variables de la cláusula. La información se expresa usando el mismo tipo de propiedades que en las anotaciones a nivel de predicado. Este tipo de anotaciones tiene un uso y tratamiento similar al de las anotaciones `trust`.

Estos diferentes tipos de anotaciones, como veremos más adelante, nos proporcionan diferentes soluciones para analizar las diversas características del lenguaje Prolog.

Un buen número de predicados predefinidos de Prolog pueden tratarse en el análisis de una forma eficiente y precisa mediante funciones que capturen su semántica. Estas funciones proporcionan abstracciones de las substituciones de salida para cada tipo de llamada al predicado. Es interesante darse cuenta de que las funciones abstractas que describen los predicados predefinidos son muy similares en espíritu a las anotaciones `trust` lo cual no es sorprendente dado que los predicados predefinidos pueden verse como predicados Prolog cuyo código no está disponible.

Con respecto a los predicados de control, el tratamiento de `true` y `repeat` es muy sencillo: la función abstracta es la identidad (es decir, pueden ignorarse). La

abstracción de `fail` y `halt` es $\bot$ (el valor ínfimo). Para el corte (`!`) también es posible usar la función identidad (es decir, ignorarlo), puesto que únicamente se provocará que más casos que los necesarios sean tenidos en cuenta, provocando a lo más una falta de precisión. Por supuesto, existen en la literatura métodos más precisos y complejos para evitar esta falta de precisión. Finalmente, el mecanismo de las excepciones también puede incluirse en esta clase. Proponemos un método para tratarlas que se basa en el hecho de que los análisis, en general, asumen que la ejecución puede fallar en cualquier punto. Por ello, un literal de la forma `catch(Goal,Catcher,Recovery)` puede aproximarse de una forma segura mediante la disyunción `(Goal;Recovery)`. El predicado predefinido `throw` puede aproximarse tratándolo como un fallo, es decir con $\bot$.

La función correspondiente a `=` es simplemente la unificación abstracta. Versiones especializadas de ella pueden usarse para otros predicados predefinidos tales como `\=`, `functor`, `arg`, `univ` (`=..`), y `copy_term`. Otros predicados para manipular términos y listas son relativamente sencillos de implementar. Los predicados aritméticos y de chequeo de tipos básicos usualmente tienen una traducción natural en el dominio abstracto considerado, y además, son una fuente de información muy valiosa para el analizador a su salida, donde se puede asumir que no han fallado. Sin embargo, los predicados `==`, `\==`, y sus versiones aritméticas son más complejos, pero se pueden tratar usando versiones especializadas de la función de unificación abstracta.

Con respecto a los predicados de entrada/salida, la salida en general no supone ningún problema porque no instancian variables ni producen otros efectos colaterales en el programa. Por otro lado, la entrada no se puede determinar a priori, pero en general el análisis puede proseguir simplemente asumiendo substituciones de salida supremas.

Los meta-predicados son predicados que usan otros predicados como argumentos. El tratamiento de los definidos por el usuario puede reducirse a tratar los predicados predefinidos de meta-llamada que internamente usen. Predicados predefinidos en esta clase son `call`, `bagof`, `findall`, `setof`, negación por fallo, y `once`. Las llamadas a los predicados recolectores de soluciones pueden tratarse como simples meta-llamadas, dado que la mayoría de los análisis son "condensan-

tes" en el sentido que siempre consideran todas las soluciones a los predicados. La negación por fallo (`\+`) y `once` pueden definirse en términos de `call` y el corte.

Dado que el predicado al que se refiere el argumento de una meta-llamada no siempre puede determinarse en tiempo de compilación, aparecen los ya mencionados *problema de las substituciones de salida*, en el predicado donde aparece la meta-llamada, y *problema de llamadas extra*, debido a las llamadas desconocidas que pueden originarse por la meta-llamada. Ya hemos comentado soluciones para dichos problemas. En todo caso, se pueden dar soluciones más precisas dependiendo del conocimiento que se tenga en tiempo de compilación de los términos que se transformarán en metas. Podemos distinguir entre:

- Meta-llamadas completamente determinadas. La meta aparece explícitamente en el texto del programa (lo que ocurre a menudo en `bagof`, `findall`, `setof`, `\+` y `once`), o puede determinarse mediante algún análisis. El tratamiento es inmediato.

- Meta-llamadas parcialmente determinadas. El término exacto no puede determinarse, pero al menos se puede determinar, mediante análisis, el constructor principal. Tampoco revisten muchos problemas, mediante el uso de substituciones abstractas supremas.

- Meta-llamadas indeterminadas. Tienen que tratarse usando las soluciones generales. Pero también el usuario puede ayudar al análisis mediante anotaciones pragma, que darán información acerca de los posibles términos a los que estará ligado en ejecución el meta-argumento de la meta-llamada. Otra solución sería considerar que las meta-llamadas son llamadas externas, y por tanto es responsabilidad del usuario completar las anotaciones `entry` con las posibles llamadas a predicados que se hagan en los meta-predicados.

Otra clase de predicados predefinidos son los que manipulan la base de datos de Prolog. Estos predicados afectan al programa en sí porque añaden o eliminan cláusulas de éste. Los predicados a los que pueden afectar se llaman predicados dinámicos y en las modernas implementaciones de Prolog deben estar declarados

como tales. Tres son los potenciales problemas creados por este tipo de predicados predefinidos. Por una parte, aparece de nuevo el problema de las llamadas extra (desde los cuerpos de las cláusulas añadidas). Por otra, aparece también el problema de las substituciones de salida, donde se llama a predicados dinámicos. Y por último, existe el problema adicional de calcular las substituciones de salida para los predicados de manipulación de la base de datos.

Desde el punto de vista de la corrección, el problema de las llamadas extra sólo aparece con `assert`. En todo caso, no todos sus usos son igualmente dañinos, pudiéndose distinguir los siguientes tipos:

- `memo`: solo se añaden hechos que son consecuencia lógica del programa;
- `data`: sólo se añaden hechos al programa;
- `local_call`: los predicados dinámicos únicamente llaman a otros predicados dinámicos;
- `global_call`: caso general.

Los dos primeros tipos no producen el problema de las llamadas extra, al sólo añadir nuevos hechos. Si el resto de los predicados dinámicos del programa es del tipo `local_call`, entonces el análisis estático del programa es correcto excepto por las cláusulas que definen los predicados dinámicos. Las optimizaciones pueden entonces aplicarse a todo el programa excepto a esos predicados, lo cual en general no es problema porque en la mayoría de los sistemas los predicados dinámicos son interpretados, no compilados. Para el caso en el que no se puedan determinar estos casos especiales de uso de predicados de manipulación de la base de datos, la solución es la general que ya hemos visto anteriormente para resolver el problema de las llamadas extra.

Con respecto al problema de las substituciones de salida para las llamadas a predicados dinámicos, en general se puede solventar usando substituciones de salida supremas. No obstante, en el caso de predicados tipo `memo`, este problema no ocurre puesto que las substituciones de salida calculadas a partir del programa estático son correctas.

El último problema, calcular las substituciones de salida para los predicados de manipulación de la base de datos, sólo se da en `retract` y `clause`. En general, se pueden usar substituciones de salida supremas con respecto a la entrada, aunque existen soluciones más precisas en ciertos casos para predicados del tipo `memo`.

Existe también una solución completamente alternativa al problema de los predicados dinámicos, basada en el análisis global incremental. En efecto, si el programa incluye una copia del compilador y del analizador incremental (en el estado en el que se quedaría habiendo analizado el programa estático), después de cada añadido o borrado problemático de una cláusula, el analizador incremental y el optimizador serían invocados convenientemente en la parte afectada. Se ha visto que este reanálisis puede realizarse en una fracción muy pequeña del tiempo de compilación normal.

Por último, hemos estudiado el impacto de la modularidad en el análisis. Consideramos únicamente módulos que tengan un interfaz estático, dado que en otro caso para el análisis es indistinguible a que no haya módulos. La declaración de un módulo, pues, proporciona los puntos de entrada al mismo, utilizables para el análisis. Si existen anotaciones `entry`, las sustituciones que contengan podrán ser utilizadas por el análisis, en otro caso se utilizarán sustituciones supremas.

El análisis incremental se puede usar convenientemente para analizar programas modulares. De esta forma, a medida que se compila cada módulo se van obteniendo resultados más exactos del proceso de análisis, y se va pudiendo optimizar los programas de una forma más efectiva.

Pero la propuesta novedosa que hacemos aquí es considerar el interfaz de un módulo ampliado con anotaciones `trust` para los predicados exportados. Entonces, cada llamada a un predicado que no está definido en el módulo que se está analizando sino que es importado de otro módulo, se tratará bien usando sustituciones supremas, si no existen anotaciones `trust` aplicables a dicho patrón de llamada, o si existe una de tales anotaciones ésta se utilizará con ventaja. Este método puede aplicarse tanto si el usuario escribe las anotaciones, como si éstas son obtenidas de una forma automática. Y esto último porque al analizar cada módulo, los pares de patrones de entrada y de salida calculados por el análisis para cada predicado exportado se pueden escribir en el interfaz del módulo como

anotaciones `trust`. A partir de ahí, esas anotaciones podrán ser utilizadas en el análisis de otros módulos que usen los ya analizados, mejorando asimismo su información exportada. Se podrá producir así un punto fijo global de una forma distribuida.

Llegados a este punto, afirmamos que el conjunto de soluciones propuestas es el primero que permite el análisis correcto de cualquier programa que use todas las características del lenguaje Prolog, y sin necesidad de información adicional por parte del usuario, pero al mismo tiempo permitiendo que dicha información pueda proporcionarse de una forma efectiva.

Un nuevo sistema de módulos para Prolog

La modularidad es una noción básica en los lenguajes de programación modernos. A pesar de que muchas implementaciones de Prolog incluyen sistemas de módulos, creemos que estos sistemas son mejorables en un número de aspectos, como por ejemplo la adecuación a un análisis global efectivo, la posibilidad de compilación separada o la creación de ejecutables independientes. Esto nos llevó a diseñar un sistema de módulos para Prolog mejorado, que hemos concretado en el sistema de módulos de Ciao. El sistema de módulos ha sido diseñado de forma que se mantenga lo más similar posible a los sistemas de módulos de las implementaciones de Prolog más populares, pero con cambios cruciales para permitir características tales como compilación separada incremental, análisis global efectivo, depuración, especialización, etc.

En concreto, las características principales que hemos tenido en cuenta al diseñar el sistema de módulos han sido:

- *Compilación modular (separada) y eficiente.* Lo cual implica que debe ser posible compilar un módulo sin tener que compilar los módulos relacionados.
- *Extensión local de capacidades y sintaxis.* Ha de ser posible definir extensiones sintácticas y semánticas del lenguaje de una manera local, sin afectar a otros módulos.
- *Adecuación al análisis global modular.* Creemos en un uso generalizado del

análisis global de programas lógicos, no sólo en su aplicación más tradicional de optimización, sino también en las nuevas aplicaciones relacionadas con el desarrollo de programas, tales como depuración automática, validación y transformación de programas. Esto es especialmente importante para Ciao, que cuenta con una herramienta de análisis global (`ciaopp`, el preprocesador de Ciao [HPB99, HBPLG99, HPBLG03]).

- *Detección ampliada de errores.* Debe ser posible comprobar estáticamente los interfaces entre los módulos y detectar errores tales como predicados no definidos, incompatibilidades, etc.

- *Soporte para la meta-programación y el orden superior.* De forma que sea posible, sin excesiva carga al programador, la meta-programación y el orden superior que crucen límites de módulos, y con la posibilidad de detectar errores en llamadas lo suficientemente determinadas.

- *Compatibilidad con estándares de facto y oficiales.* Hasta donde sea posible (es decir, sin renunciar a otros objetivos principales). Esto es debido a que un objetivo de Ciao es que sea (gracias a un conjunto particular de librerías cargadas por defecto) un sistema Prolog estándar. Lo que significa que, al menos por defecto, el sistema de módulos será basado en predicados en vez de basado en átomos. Además el sistema no debe obligar a un tipado fuerte, dado que Prolog tradicionalmente es no tipado.

Ninguno de los sistemas de módulos usados en las implementaciones de Prolog, excepto el de Ciao, proporciona todos los objetivos arriba mencionados, e incluso algunos incluyen características claramente opuestas a esos objetivos. Algunas de las características concretas del sistema de módulos necesarias para cumplir los objetivos enunciados son:

- *La sintaxis, las opciones (*flags*), etc, deben ser locales a los módulos.* Es decir, que la sintaxis o el modo de compilación de un módulo no podrá ser modificado por otros módulos. En caso contrario, la compilación y el análisis modular son imposibles. En conclusión, directivas como `op/3` y `set_prolog_flag/2` deben ser locales al módulo.

- *Los puntos de entrada de un módulo deben estar definidos estáticamente.* Por tanto, las únicas llamadas externas al módulo permitidas han de ser a predicados exportados. El permitir la llamada a un predicado no exportado de un módulo desde otro tiene un impacto catastrófico en la precisión del análisis global, impidiendo muchas optimizaciones.

- *La cualificación de módulos es para desambiguar nombres de predicados, no para cambiar el contexto de nombrado.* Esto es necesario para la compilación separada dado que en caso contrario para compilar un módulo puede ser necesario conocer las importaciones y exportaciones de otros módulos. Por ejemplo, dada la meta `m:p` ("llamar a `p` en el módulo `m`"), con la primera opción sólo necesitamos saber si el módulo `m` exporta `p`, pero con la segunda, dado que `m` puede importar el predicado `p` de otro módulo, necesitamos también conocer esas relaciones del otro módulo.

- *El texto de un módulo no debe estar en partes sin relación o no disponibles.* Esto quiere decir que todo el código de un módulo debe estar junto o directamente accesible al compilar, de modo que el compilador pueda acceder a todo él automáticamente.

- *Las partes dinámicas deben estar aisladas lo más posible.* La modificación dinámica del código, aunque útil en algunas aplicaciones, es muy perjudicial para el análisis global de programas, como hemos visto anteriormente. Una primera idea es relegar dichos predicados a un módulo especial de biblioteca, de modo que si el analizador detecta que éste es usado en un módulo pueda tomar las medidas oportunas. Además, en nuestra experiencia los predicados dinámicos son las más de las veces usados únicamente para implementar "variables globales", y para este propósito es suficiente poder añadir hechos al programa. Esta utilidad, siempre que los predicados dinámicos que se vayan a utilizar así se declaren como tales en el código, no da excesivos problemas al análisis global. Por ello Ciao incluye un conjunto de predicados predefinidos para añadir y eliminar hechos de una clase especial de predicados dinámicos que se declaran como `data`. Además, la

implementación de dichos predicados "data" puede ser más eficiente que la de los predicados dinámicos no restringidos.

- *La mayoría de los "predicados predefinidos" debería estar en librerías que puedan ser usadas o descartadas en el contexto de un módulo dado.* Esto es un requerimiento para la extensibilidad, sirve de ayuda al analizador (tal como hemos visto arriba) y permite producir ejecutables más pequeños.

- *Las directivas no deben ser llamadas.* Tradicionalmente, las directivas eran ejecutadas por el intérprete de Prolog como llamadas al programa. Esto no tiene sentido en el contexto de la compilación separada. Afortunadamente, este tema ha sido bien resuelto en el estándar Prolog [PRO94, DEDC96], donde existe una directiva especial para proporcionar llamadas de inicialización.

- *Los meta-predicados deben declararse, al menos si son exportados, y la declaración debe reflejar el tipo de meta-información manipulada en cada argumento.* Esto es necesario de modo a poder realizar un control de errores razonable para los meta-predicados, y para poder resolver estáticamente meta-llamadas entre módulos en la mayoría de los casos.

Dadas las premisas comentadas, procedemos a presentar su concretización en el sistema de módulos de Ciao.

El fuente de un módulo Ciao está típicamente contenido en un único fichero, que coincide en nombre con el módulo (excepto quizás por una extensión opcional). En todo caso, el sistema permite la inclusión del código de otro fichero en un punto preciso del módulo mediante el uso de la declaración ISO `:- include`. Al principio del fichero que define un módulo debe haber una declaración "`:- module(...`".

Como ya hemos comentado, el sistema de módulos de Ciao es en la actualidad basado en predicados, lo cual quiere decir que los nombres de los predicados no exportados son locales al módulo, mientras que los nombres de los átomos y constructores de los datos son compartidos. Esto, aunque ofrece ventajas con respecto a hacer más conciso el interfaz, tiene la desventaja de que hay que tratar

de forma especial los meta-predicados, y puede complicar el tener tipos abstractos de datos en los módulos. El problema de los meta-predicados se resuelve en Ciao mediante declaraciones apropiadas o usando orden superior auténtico. Además, para permitir definir tipos abstractos de datos auténticos, en Ciao es posible ocultar nombres de átomos, esto es, hacer que sean locales al módulo.

Un módulo puede exportar un número de predicados, mediante declaraciones `:- export` o en la lista de exportación de la declaración del módulo. También es posible indicar que se exportan *todos* los predicados del módulo (usando '_'). Se pueden importar un conjunto de predicados explícitamente y también todos los predicados de un módulo, usando las declaraciones `:- use_module` y `:- import`. En cualquier caso sólo es posible importar de un módulo predicados que éste exporte. Es posible importar un predicado que tenga el mismo nombre y aridad que uno local, o que otro importado de otro módulo. Esto se aplica también a los predicados de los módulos implícitamente importados, que hacen el papel de los predicados predefinidos de otros sistemas de programación lógica. Un módulo `m1` puede *reexportar* otro módulo, `m2`, mediante una declaración `:- reexport`. El efecto de esto es que `m1` exporta todos los predicados de `m2` como si estuvieran definidos en `m1`. Esto permite definir módulos que *extienden* otros módulos. También es posible reexportar únicamente algunos de los predicados de otro módulo, mediante una lista en la declaración.

En Ciao es posible marcar ciertos predicados como *propiedades*. Ejemplos de propiedades son los *tipos regulares*, propiedades de instanciación (`var`, `indep`, o `ground`), propiedades computacionales (`det` o `fails`), etc. Estas propiedades, dado que en realidad son predicados, pueden importarse o exportarse usando las mismas reglas que cualquier otro predicado. Las propiedades importadas pueden ser usadas en aserciones del mismo modo que las definidas localmente.

Con respecto a las reglas de visibilidad, el conjunto de predicados visibles en un módulo son los predicados definidos en el módulo junto con los predicados importados de otros módulos. Es posible referirse a los predicados usando o no *cualificación de módulos*. Un nombre de predicado con cualificación de módulo tiene la forma *modulo*:*predicado* como en la llamada `lists:append(A,B,C)`. Denominamos el módulo por defecto de un nombre de predicado el módulo que

contiene la definición del predicado que será ejecutado cuando se use dicho nombre sin cualificación de módulo. La cualificación de módulos permite referirse a un predicado de un módulo que no es el módulo por defecto de ese nombre de predicado. Las reglas que determinan el módulo por defecto de un nombre de predicado son las siguientes: si el predicado está definido en el módulo donde aparece la llamada, entonces ese módulo es el por defecto (es decir, las definiciones locales tienen prioridad sobre las importaciones); en caso contrario, el módulo por defecto es el último módulo desde donde se ha importado el predicado en el texto del programa. En todo caso, los predicados que se importan *explícitamente* (es decir, que aparecen listados en la lista de importación de `:- use_module`) tienen prioridad sobre los que se importan *implícitamente* (es decir, importados cuando se importan todos los predicados de un módulo). Dado que se considera que los módulos implícitamente importados se importan antes que los demás, el sistema permite la redefinición de los "predicados predefinidos". Es más, combinando las llamadas con cualificación y sin cualificación se puede también *extender* los predicados predefinidos, una capacidad usada en la implementación de alguna de las librerías de Ciao. No es posible acceder a un predicado de otro módulo que no se ha importado, incluso si se usa cualificación de módulos y el predicado es exportado por el otro módulo. Tampoco está permitido definir cláusulas de predicados pertenecientes a otros módulos (excepto si el módulo que define el predicado lo declara como dinámico y lo exporta). Veremos más adelante, sin embargo, la cuestión de los predicados "multifile". Los predicados reexportados también pueden ser usados en el módulo que los reexporta, es decir, también son importados en dicho módulo. Esto, unido a que un predicado exportado explícitamente tiene prioridad sobre uno reexportado, permite reexportar un módulo pero redefiniendo ciertos predicados, usando posiblemente la definición original (cualificada) para definir el predicado modificado.

Por razones básicamente de compatibilidad hacia atrás con sistemas Prolog no modulares, hay algunas desviaciones de las reglas generales que son comunes a otros sistemas de programación lógica modulares: el módulo "`user`" y los predicados `multifile`.

Para proporcionar compatibilidad hacia atrás con código no modular, todo

código que pertenece a ficheros que no tengan declaración de módulo se asume que pertenece a un módulo especial llamado "`user`". Todos los predicados que pertenecen al módulo `user` son exportados, y se puede llamar sin restricción desde un predicado de un fichero `user` a otro definido en otro fichero `user`. Sin embargo, a diferencia de lo que ocurre en otros sistemas Prolog, las importaciones de módulos normales que aparecen en un fichero `user` no son visibles en los otros ficheros `user`. Esto permite al menos realizar compilación separada de ficheros `user`, dado que todas las llamadas a predicados en el fichero pueden determinarse estáticamente por el contenido del fichero. Los predicados definidos en ficheros `user` pueden ser visibles en módulos auténticos, pero únicamente si se importan explícitamente del módulo `user` (nombrando los predicados que se importan).

No se recomienda el uso de ficheros `user` porque, aparte de perder la separación de los nombres de predicado, su estructura hace imposible detectar muchos errores que el compilador puede detectar en los módulos reales (por ejemplo, predicados no definidos). Además, el análisis global de ficheros `user` conlleva una pérdida de precisión considerable, dado que todos los predicados son posibles puntos de entrada. Obsérvese que en general es tan sencillo y flexible usar en vez de ficheros `user` módulos normales que exporten todos los predicados (lo cual en Ciao es fácil substituyendo la lista de exportación por una variable), manteniendo al mismo tiempo muchas de las ventajas de los módulos.

Los predicados `multifile` son una característica útil (también definida en ISO-Prolog) que permite que un predicado pueda ser definido por cláusulas que pertenecen a diferentes ficheros (módulos en el caso de Ciao). Para poder acomodar esto en el sistema de módulos, en Ciao estos predicados son implementados como si perteneciesen a un módulo especial `multifile`. Sin embargo, las llamadas que ocurren en una cláusula de un predicado `multifile` se considera que son siempre a los predicados visibles en el módulo donde aparece la cláusula. Debido a esto, los predicados `multifile` no ofrecen problemas especiales al analizador global (si los considera como predicados exportados) ni al procesamiento de código en general.

El sistema de módulos que hemos descrito hasta ahora es *estático*, lo cual tiene muchas ventajas con respecto a la compilación, análisis global, optimización y de-

tección de errores. Pero en la práctica es a veces muy útil la posibilidad de cargar código dinámicamente y ejecutarlo. En Ciao esto también es posible, pero únicamente si el módulo de biblioteca correspondiente al compilador es explícitamente importado. Esta importación puede así ser detectada por las herramientas de compilación, que pueden actuar de una forma más prudente en los módulos donde se produzca. Además, cuando se generan ejecutables sólo es necesario incluir el compilador si la aplicación hace uso del módulo de biblioteca correspondiente.

Con respecto a los meta-predicados, la solución de Ciao es similar a la de otros Prolog con sistemas de módulos basados en predicados: el uso de la declaración `meta_predicate`, que especifica qué argumentos de los predicados contienen meta-datos. Sin embargo, en Ciao se distingue en la declaración el tipo de meta-datos: objetivo, cláusula, hecho, etc, lo cual es de gran ayuda a las herramientas de compilación.

Tradicionalmente, los sistemas Prolog han tenido la capacidad de cambiar la sintaxis del programa fuente mediante el uso de la directiva `op/3`. Además, muchos proporcionan capacidades de expansión de código (generalmente definiendo el predicado `term_expansion/2`). Pero estas capacidades, en su forma original, crean problemas a la compilación modular o para crear sensatamente ejecutables independientes. Su efecto era global y no se podía determinar estáticamente a qué ficheros afectaban. Además, no estaba separado el efecto de las definiciones en tiempo de compilación y en tiempo de ejecución. Para resolver estos problemas, en Ciao hemos redefinido estas capacidades de forma que son locales al módulo que las define, y además tienen un efecto bien definido en el contexto de un compilador independiente. En particular, la directiva `load_compilation_module/1` permite cargar un fichero *en el compilador*, separando de esta forma el código que se usa en tiempo de compilación del que es usado por el programa en ejecución. Además, los efectos que usualmente se conseguían usando `term_expansion/2` se pueden obtener en Ciao mediante el uso de cuatro directivas diferentes, más especializadas.

Hay otra directiva en Ciao que está relacionada con la naturaleza extensible del sistema: `new_declaration/2`. Esto permite añadir nuevas declaraciones que, aunque son ignoradas por el compilador, pueden ser usadas por otras herramientas

de proceso de código (analizadores, documentadores, etc).

La experiencia en el uso del sistema de módulos de Ciao muestra que la naturaleza local de las extensiones sintácticas y la distinción entre código para la compilación y código para la ejecución permite la sistematización de las librerías que definen extensiones al lenguaje. Estas librerías consisten usualmente en un fichero principal que contiene las declaraciones pertinentes, y que se *incluye* como parte del fichero que usa la librería, más otro módulo en el caso que se definan expansiones que contiene el código necesario en compilación (cargado en el compilador mediante una directiva `load_compilation_module`), más el código que opcionalmente se pueda usar en tiempo de ejecución, cargado por medio de declaraciones `use_module`. Las librerías que se estructuran de esta forma se llaman "paquetes" en Ciao, e incluyen capacidades como DCGs (gramáticas), programación funcional, base de datos persistente, aserciones, etc.

Un modelo de orden superior para Ciao

Proponemos un modelo de orden superior para Prolog que se aparta de otros enfoques anteriores. Nuestro objetivo era definir el orden superior en el contexto del lenguaje Prolog estándar, que no es tipado, y evitando ralentizar la ejecución del código de primer orden. Además, queríamos que el modelo facilitara el análisis global del programa.

Los principios en los que nos basamos en este modelo de orden superior han sido:

- Un dato de orden superior se diferencia sintácticamente de un dato ordinario. De esta forma, cualquier nombre del programa puede saberse fácilmente si corresponde a un predicado o a un constructor de datos. Adicionalmente, un nombre de predicado ocurre siempre con su aridad. Creemos que estas propiedades son cruciales en un sistema de módulos basado en predicados, como el actual de Ciao.

- Los datos de orden superior son como un "tipo abstracto de datos": se consideran una "caja negra" que no puede inspeccionarse o manipularse,

excepto mediante operaciones específicas definidas. Esta propiedad hace al orden superior más compatible con un análisis global efectivo.

Además, y como ocurre con todas las extensiones en Ciao, el orden superior sólo estará disponible en módulos que usen el paquete `hiord`.

La sintaxis que hemos definido para los datos de orden superior es que éstos están delimitados entre `{` y `}` Hemos definido dos formas de escribir datos de orden superior: *abstracciones de predicado* y *clausuras*. Las *clausuras* pueden considerarse "edulcorante sintáctico" para abstracciones de predicado, y pueden siempre ser escritas de esa otra forma. Las abstracciones de predicado son nuestra traslación a la programación lógica de las expresiones lambda de la programación funcional: definen predicados sin nombre que se ejecutarán en último término por una llamada de orden superior, unificando sus argumentos convenientemente. La sintaxis más general para las abstracciones de predicado es

```
{varscomp -> ''(parametros) :- cuerpo}
```

donde *varscomp* es una secuencia de variables distintas y *parametros* es la secuencia de parámetros. Cuando *varscomp* es vacía no se escribe la flecha, y cuando *parametros* es vacío no se escriben los paréntesis. Además, cuando *cuerpo* es `true` puede omitirse el texto "`:- true`". Algunos ejemplos son:

```
all_less(L1, L2) :- map(L1, {''(X,Y) :- X < Y}, L2).

same_mother(L) :- list(L, {M -> ''(S) :- child_of(S,M,_)}).

same_parents(L) :- list(L, {M,F -> ''(S) :- child_of(S,M,F)}).
```

Las variables en *varscomp* se comparten con el resto de la cláusula y en sucesivas invocaciones de la abstracción de predicado, de modo que la instanciación de dichas variables afecta a la abstracción de predicado. El resto de las variables en la abstracción de predicado es local a la misma, incluso si sus nombres coinciden con variables que aparezcan fuera de la abstracción de predicado. Por tanto, se considera que las variables que no están en *varscomp* ni en *parametros* están cuantificadas existencialmente.

Todo dato de orden superior que no tenga sintaxis de abstracción de predicado es una clausura. En una clausura, todas las ocurrencias del átomo `#` marcan un parámetro de la abstracción de predicado que representan. Todas las variables de una clausura se comparten con el resto de la cláusula, de modo que la siguiente definición de `same_parents/2` es equivalente a la de arriba:

```
same_parents(L) :- list(L, {child_of(#,_M,_F)}).
```

Los parámetros, representados por `#`, se ordenan tal y como aparecen en la clausura. Por ejemplo, la siguiente definición de `all_less/2` es equivalente a la de arriba:

```
all_less(L1, L2) :- map(L1, {# < #}, L2).
```

Si se necesita un orden diferente, debe usarse una abstracción de predicado.

Las operaciones permitidas con datos de orden superior son las siguientes:

`apply(P,`*`parametros`*`)`
: Si `P` está ligada a un dato de orden superior esta llamada ejecuta `P` instanciando sus parámetros formales a *`parametros`*. Si `P` no está ligada, la ejecución continúa retrasando la ejecución de la llamada hasta que se ligue. Si `P` está ligada a un término normal se eleva un error.

`shared_vars(P, VList)`
: Devuelve en `VList` la lista de las variables que `P` comparte con el resto del programa.

`arity(P, N)`
: Devuelve en `N` la aridad de la abstracción de predicado `P`.

Obsérvese que dada nuestra definición del orden superior, no hay forma de crear nuevos datos de orden superior en ejecución (todos los datos tienen que estar definidos en el código). Esto no es por olvido, sino que creemos que esa capacidad se necesita raramente, y además impide definitivamente un análisis global efectivo del programa. Por otra parte, usando el código dinámico de Prolog

podemos lograr añadir esta capacidad a una abstracción de predicado, cuando sea necesaria.

En nuestra propuesta, evitamos las complicaciones que proporciona la unificación de orden superior considerando las abstracciones de predicado como una caja negra que no puede ser inspeccionada. Por tanto, los datos de orden superior no pueden unificarse con términos ordinarios, y dos instancias diferentes de una abstracción de predicado son unificables si y sólo si representan una llamada directa al mismo predicado existente en el programa.

Parte II: Cuestiones de implementación

Para obtener un sistema de Programación Lógica de nueva generación práctico y usable, la implementación de las características indicadas arriba debe ser diseñada cuidadosamente. En esta parte se muestran las soluciones que hemos adoptado en la implementación del sistema Ciao. Se ha implementado un compilador modular que permite la compilación separada de módulos, y que puede producir varias clases de ejecutables, que ofrecen diversos compromisos entre el tamaño del ejecutable, el tiempo de arranque, y la portabilidad. Uno de los resultados más interesantes en el desarrollo del compilador, desde el punto de vista de la arquitectura del programa, ha sido que hemos sido capaces de abstraer en un módulo genérico de procesamiento de código fuente la mayor parte de la funcionalidad relacionada con el tratamiento incremental y modular de los programas, inclusive cuando están separados en varios ficheros y usan múltiples módulos de biblioteca tanto del usuario como del sistema. Este módulo genérico permite el desarrollo de herramientas de transformación y análisis de programas de una manera que es hasta cierto punto ortogonal a los detalles de diseño del sistema de módulos, y ha sido usado también en otras herramientas del sistema Ciao como el preprocesador `ciaopp` [HPB99, HBPLG99, HPBLG03], el documentador automático [Her99, Her00], etc. Este módulo genérico no trata el proceso de compilación como una simple transformación de un único y aislado programa fuente de Prolog a, por ejemplo, su código WAM. En lugar de ello, cada módulo se compila teniendo en cuenta su relación con los otros módulos que usa. Además,

el conjunto de módulos que compone una aplicación, junto con el conjunto de módulos de biblioteca del usuario o del sistema que ésta usa, se procesan global e incrementalmente. Como consecuencia de ello, el código objeto correspondiente y cualquier otro resultado del procesamiento se mantienen siempre actualizados con respecto al código fuente cuando se recompila el conjunto mínimo de módulos afectados por un cambio.

Parte III: Aplicaciones del análisis global

Dado que esta tesis está dedicada a desarrollar un sistema y lenguaje de Programación Lógica con soporte al análisis global, en esta parte mostramos una aplicación práctica del análisis global: la paralelización automática. De hecho, fue precisamente nuestro trabajo inicial en este último tema lo que nos llevó a enfocarnos en el diseño de un lenguaje de programación lógica con las características de las que hemos hablado. En esta parte presentamos una técnica para la paralelización automática de Prolog basada en la información recopilada por el análisis global. El análisis usado se basa en la interpretación abstracta con el dominio de Sharing+Freeness (compartición y libertad), que captura las propiedades relevantes de compartición, basicidad ("groundness") y libertad de variables. Estas propiedades permiten la detección de la independencia no-estricta de metas (NSI), una noción que asegura la eficacia de la paralelización. En esta parte, introducimos una representación pictórica novedosa para las substituciones abstractas implicadas, presentamos las condiciones suficientes para la detección de la independencia no-estricta en tiempo de compilación, basadas en la información de las substituciones abstractas, y describimos una técnica del paralelización que combina la detección en tiempo de compilación con la detección en tiempo de ejecución.

Principales aportaciones

A continuación enumeramos las principales aportaciones de esta tesis. Algunos de los resultados obtenidos han sido ya publicados y presentados en foros interna-

cionales, en cuyo caso las publicaciones relevantes se mencionan explícitamente. Además, la primera aportación mencionada ha sido hecha en colaboración con otros investigadores además del director de la tesis. Esta colaboración también se menciona explicitamente.

- Las técnicas de análisis de programas basadas en la interpretación abstracta han recibido una atención considerable, desde el punto de vista de los marcos de trabajo generales y los dominios abstractos. Sin embargo, en comparación se ha prestado poca atención a los problemas que surgen cuando se trata de analizar un dialecto completo y práctico del lenguaje Prolog, y hasta el momento solamente se habían propuesto de forma dispersa algunas soluciones a estos problemas. Nosotros hemos propuesto la primera combinación de técnicas de análisis (algunas conocidas, otras novedosas) que permite analizar de forma correcta cualquier programa que use cualquier elemento del estándar ISO de Prolog. Este trabajo ha sido realizado en colaboración con Francisco Bueno y Germán Puebla, ambos del grupo CLIP de la Universidad Politécnica de Madrid, al que también pertenece el doctorando. El trabajo fue presentado primeramente en el taller de trabajo sobre Análisis Estático asociado a la Conferencia Internacional de Programación Lógica (International Conference of Logic Programming – ICLP) de 1995 [BCHP95], y una versión mejorada se publicó en el Simposio Europeo de Programación (European Symposium on Programming – ESOP) de 1996 [BCHP96].

- Los sistemas de programación lógica estándar existentes hasta ahora tenían características que dificultaban de una forma u otra su análisis global efectivo. Además, extender dichos sistemas era a veces difícil y otras veces retorcido, haciendo que fueran poco adecuados para explorar tecnologías y modelos venideros, debido a la imposibilidad de una redefinición modular del sistema o del lenguaje. Hemos diseñado un sistema de módulos para Prolog que ayuda al análisis global, la optimización y la verificación, altamente extensible y reconfigurable, y lo hemos aplicado al sistema Ciao. El trabajo ha sido presentado en el taller de trabajo sobre paralelismo e im-

plementación de sistemas (C)LP de ICLP'99 [CH99a], en los Comentarios Electrónicos sobre Ciencia de la Computación Teórica (Electronic Notes in Theoretical Computer Science), en un número especial sobre paralelismo e implementación de sistemas (C)LP [CH00c], y finalmente se presentó una versión mejorada en la Conferencia Internacional de Computación Lógica (International Conference on Computational Logic) CL2000 [CH00a].

- Dentro del sistema Ciao, hemos desarrollado una implementación de "orden superior" para un lenguaje estándar de programación lógica modular y no tipado, con características originales, que incluye una implementación de `call/N` y *abstracciones de predicado*.

- Hemos concebido e implementado un módulo genérico de procesamiento de código fuente, modular y paramétrico, que proporciona compilación separada y tratamiento automático de dependencias, y ha sido diseñado para soportar de forma nativa análisis global, optimización y verificación. Basado en este módulo genérico de procesamiento, hemos implementado un compilador, que aparte de ofrecer las características proporcionadas por el primero, puede producir varios tipos diferentes de ejecutables, algunos de ellos basados en conceptos novedosos, como los módulos activos. Las implementaciones se han integrado en el sistema Ciao. Este trabajo ha sido presentado en el taller de trabajo de ICLP'99 sobre paralelismo e implementación de sistemas (C)LP [CH99b], y en los Comentarios Electrónicos sobre Ciencia de la Computación Teórica (Electronic Notes in Theoretical Computer Science), en un número especial sobre paralelismo e implementación de sistemas (C)LP [CH00b].

- Hemos desarrollado la primera técnica para la paralelización automática de lenguajes de programación lógica basada en la noción de independencia no-estricta (NSI). La NSI es una noción más relajada que la noción tradicional de independencia estricta (SI), que asegura de la misma forma la eficacia de la paralelización pero permite considerablemente más paralelismo que la SI. Hemos encontrado condiciones suficientes para la detección en tiempo de

compilación de la NSI, basadas en el análisis por interpretación abstracta con el dominio de Sharing+Freeness, hemos elaborado una técnica para calcular pruebas de independencia en tiempo de ejecución (para permitir más paralelismo), y hemos desarrollado un algoritmo de paralelización basado en propiedades de independencia entre pares de metas. El trabajo ha sido presentado en el Simposio Internacional de Análisis Estático (International Static Analysis Symposium) de 1994 [CH94] y en mi Trabajo Fin de Carrera [CG93].

El sistema Ciao completo, así como otras aplicaciones relacionadas, están disponibles para su descarga en el sitio WWW del grupo CLIP:

`http://www.clip.dia.upm.es/Software`

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

An Extensible, Global Analysis Friendly Logic Programming System

PhD Thesis

DANIEL CABEZA GRAS

August, 2004

PhD Thesis

An Extensible, Global Analysis Friendly Logic Programming System

presented at the Computer Science School
of the Technical University of Madrid
in partial fulfillment of the degree of
Doctor in Computer Science

PhD Candidate: **Daniel Cabeza Gras**
Licenciado en Informática
Universidad Politécnica de Madrid

Advisor: **Manuel V. Hermenegildo Salinas**
Catedrático de Universidad

Madrid, August, 2004

To my parents, Ana and Jaime

Acknowledgements

I want to express here my gratitude to all people who in one way or another have helped me in bringing this thesis to an end. In the first place, my father and mother, without whom I could not even started it. I am very grateful to Manuel Hermenegildo, my PhD advisor, for his advice and encouragement all these years. My appreciation also for my fellows at the CLIP group, past and present, among whom I will mention the old-timers Francisco Bueno, Manuel Carro, Pedro López, and Germán Puebla, and the recent acquisition Elvira Albert (I had to `:)`). Finally, I want to thank all people who have given me their affection during this time, particularly I will mention Beatriz Martín del Campo, because of what she renounced for this thesis.

The Ciao system is a collaborative effort and includes contributions from members of several institutions, including UPM, SICS, U.Melbourne, Monash U., U.Arizona, Linköping U., NMSU, K.U.Leuven, Bristol U., and Ben-Gurion U. The system documentation and related publications contain more specific credits. The Ciao abstract machine, the low-level WAM compiler, and some of the library files, are evolutions of the corresponding &-Prolog components, themselves independent evolutions of the excellent corresponding modules in SICStus versions 0.5-0.7. We thank SICS in the name of the logic programming community for allowing public distribution of those components.

The research included in this thesis was partially supported by a research scholarship of the Spanish Ministerio de Educación y Ciencia and by EU and Spanish research projects "ParForCE" (ESPRIT BR 6707 / TIC93-0976-CE), "ACCLAIM" (ESPRIT BR 7195 / TIC93-0975-CE), "IPL-D" (CICYT TIC 93-0737-C02-01), "ELLA" (CICYT TIC 96-1012-C02-01), "DiSCiPl" (ESPRIT LTR 22532/CICYT TIC 97-1640-CE), "ECCOSIC" (Fulbright U.S.-Spain Science and Technology Exchange Program), and "EDIPIA" (MCYT TIC 99-1151).

Abstract

The goal of this PhD Thesis is to design and develop a next-generation multi-paradigm programming system, which is based on a Logic Programming kernel and is amenable to global analysis.

The Logic Programming paradigm has proved particularly useful for developing complex applications, such as those which appear in Artificial Intelligence, including knowledge based systems, intelligent agents, expert systems, etc.

In the context of Logic Programming, much progress has been made in the area of global program analysis (mainly based on abstract interpretation), which is used for inferring information at compile-time on the run-time behavior of programs. The inferred information has been shown to be useful not only to produce optimized code, but also in program verification and error detection/diagnosis.

However, while global analysis of logic programs is at this point relatively well understood from a theoretical point of view, little attention has been given to developing a practical logic programming system (i.e., providing at least a complete Prolog implementation with modules) which incorporates such analysis. This thesis fills this gap by proposing a language design and system implementation that makes global analysis practical and scalable.

Another objective of the system developed is to serve as a workbench for developing new extensions to Logic Programming. The system includes techniques which allow extending the kernel language in a very flexible way (and at the source level), and this has been used to develop and test numerous extensions including functions, objects, constraints, records, persistence, distributed execution, and others.

The resulting system (Ciao) has been made available freely to the community and currently has a sizable number of users.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of *Artificial Intelligence* [MMRS55, RN95] is characterized by its requirements of symbolic processing, knowledge representation, state space searching, and other such high-level, complex tasks. The *Logic Programming* paradigm [Kow74, Kow80, Col87] has intensively been applied since its origins in the context of Artificial Intelligence due to its appropriateness for such requirements, for the implementation of typical applications in this field, such as expert systems, knowledge bases, intelligent agents, etc. In addition, the recent extension to Logic Programming, the *Constraint Logic Programming* paradigm [JM94], greatly facilitates the implementation and efficient execution of some kinds of tasks which are also highly relevant in Artificial Intelligence, such as planning, optimization, etc.

However, it is also the case that the higher the level of a programming language, the more separation it implies from the machine language, thus leading in principle to lower efficiency in terms of speed and storage. Also, although programming at a higher level greatly helps the coding process, due to the complex nature of the problems tackled in the field of Artificial Intelligence it is desirable to develop techniques for assisting the programmer in the verification/debugging of his/her code. The technique of global program analysis, consisting in inferring information at compile-time on the run-time behavior of the program, can help to fulfill both needs. Firstly, such information can be used by the compiler of the high-level language to produce optimized code, by specializing the transla-

tion of a language construct to each particular case, or by allowing the use of advanced characteristics of the hardware, as parallel processing. And secondly, the compile-time information is clearly instrumental in program verification and error detection/diagnosis.

These premises have lead us to focus this work on the programming paradigm of (Constraint) Logic Programming, in order to improve the development of correct and efficient programs in a reasonable time. Note, however, that most of the ideas presented herein are general and thus equally applicable to other programming paradigms. In addition, the work shown here will be of use also in improving the development process of applications which are not specific to Artificial Intelligence.

At the time of starting the work of this thesis, there were quite a number of Logic Programming systems, from experimental or unconventional [AK93, HL94] to consolidated and commercial [Swe95, Qui86] (mainly featuring the Prolog language, with slight variations). Research on global program analysis for Logic Programming, generally based on abstract interpretation [CC77], had also received considerable attention from the theoretical point of view. But, in our opinion, little effort had been placed in the development of a practical, standard logic programming system designed from the start to support global program analysis, in a way that real programs would be amenable to global analysis, optimization and verification in an accurate and seamless manner.

This situation directed our research towards the fulfillment of this objective. The work thus developed for this thesis has been applied to the design and implementation of the next-generation logic programming system "Ciao". Ciao has been developed from the start to be a practical, efficient system, suited both for programming in the small as well as programming in the large, and above all supportive of global analysis, optimization, and verification. Furthermore, with the objective of also being an experimental workbench for future developments, it has been designed to be highly extensible and reconfigurable. In our view, current proposals for logic programming languages and systems are actually "bundled packages", in the sense that they offer a combined solution affecting a number of issues, many of them quite orthogonal. Our philosophy is to give always the

possibility of exploring upcoming technologies and models by the modular redefinition of the system/language. In that sense, Ciao is an evolving system, based on the design of a simple but powerful core to which features are added in a modular fashion.

1.1 Thesis Objectives

The main objective of the work presented in this thesis is the design and implementation of a high-level, extensible logic programming system/language supporting global program analysis and its applications to code optimization and verification, of general use but aimed especially to the resolution of problems related to the field of Artificial Intelligence. An associated objective has been the development of techniques for the improvement of the use of global program analysis information in the exploitation of parallelism in logic programming.

1.2 Structure of the Work

This thesis is structured in three parts. Each part is devoted to the following issues: language design, system implementation, and global analysis applications (in particular parallelization of logic programs). Next, we provide an introductory description of each of these parts.

1.2.1 Part I: Language design

A suitable language design is fundamental for the achievement of the objectives pursued by this thesis: extensibility, global analysis support, and enhanced error detection and program verification/debugging. We concentrate first on studying issues related to the most commonly used logic programming language, Prolog, as the de-facto and nowadays official standard for most of the implementations of the logic programming paradigm. The research was firstly directed to study the language features which pose problems to global analysis, fundamentally to provide solutions for dealing with these features but as a result acquiring the

necessary knowledge to devise alternatives or enhancements to avoid the difficulties found. Chapter 2 shows the results of this research. Then, the work focused on the design of a module system for Prolog, which provided our above stated objectives. Note that modularity, apart from aiding in program development and maintenance, is a practical requisite for the global analysis of medium- to high-sized software projects, as the complete analysis of the whole code composing such applications results impractical due to its requirements in terms of time and memory consumption. With proper module system and compilation procedure design, one should be able to analyze and compile a single module without requiring the source code of other modules, just their stated interface. In this way, global analysis turns more manageable and the development process faster. The module system was designed also to allow the easy extension/customization of the language, at source level (i.e. in Prolog), making possible among other things the redefinition of predefined builtins, extensions to the syntax, and semantic extensions, in a modular and structured way. Some of the extensions that we have developed using the facilities provided are feature-terms, breadth-first and iterative deepening search, constraint logic programming over the real or the rational domains (using the powerful concept of attributed variables), objects, and functional programming (including appropriate syntax). The resulting system is a true multi-paradigm programming system. Chapter 3 deals with this issue of module system design. Another feature which increases the expressive power of the language is that of higher order. Although in Prolog there is traditionally rudimentary meta-programming facilities in the form of the `call/1` and `=../2` builtins, which allow emulating higher order, we wanted to provide more expressive and formal ways of using higher order in a standard, untyped, modular logic programming language, and with better syntax. We have developed an implementation of higher order for Ciao, based on the `call/N` [NP92, Nai96, MO84] construct, including "predicate abstractions" (the translation to Logic Programming of the lambda expressions of Functional Programming). In Chapter 4 we describe these higher-order facilities. All these extensions can be combined for a synergistic effect. For example, when functional notation is combined with the higher-order features of Ciao, the language subsumes in functionality an (eager,

untyped) functional language (support for types and type declaring is provided through the assertion language and preprocessor, see chapter 3).

1.2.2 Part II: Implementation Issues

To achieve a usable, practical next-generation logic programming system the implementation of the features stated above should be carefully designed. In this part the solutions we adopted in the Ciao system implementation are shown. We implemented a modular compiler that allows separate compilation of modules, and is able to produce several kinds of executables, which offer different tradeoffs between executable size, startup time, and portability. One of the most interesting results of the development of the compiler, from the software architecture point of view, has been that we have been able to abstract away into a generic code processing library much of the functionality related to the modular and incremental treatment of programs, even if they are multi-file and use multiple user and system libraries. This library allows the development of program analysis and transformation tools in a way that is to some extent orthogonal to the details of the module system design, and has been used also in other Ciao system tools such as the `ciaopp` preprocessor [HPB99, HBPLG99, HPBLG03], the automatic documenter [Her99, Her00], etc. The library does not treat the compilation process as a translation from a single, isolated Prolog source to, e.g., its WAM bytecode. Instead, a module is compiled taking into account its relationship with the modules it uses. Also, sets of modules comprising an application, together with the set of user or system libraries it uses, are processed globally and incrementally. As a result, the corresponding object code and any other output of the process is kept always updated with respect to the source, while recompiling the minimal required set of dependent modules after a change. Chapter 5 explains in detail all the aforementioned issues.

1.2.3 Part III: Global Analysis Applications

Given that this thesis is devoted to devising a logic programming system/language which supports global analysis well, we show in this part a practical application

of global analysis: automatic parallelization. In fact, it was precisely our early work on this issue that lead us to pursue the design of a logic programming language with the characteristics we have discussed. We present a technique for automatic parallelization of Prolog taking as input the information gathered by global analysis. The analysis used is based on abstract interpretation with the Sharing+Freeness domain, which captures the relevant properties of variable sharing, groundness, and freeness. These properties allow the detection of non-strict independence of goals (NSI), a notion which ensures the efficiency of the parallelization. In this part, we introduce a novel pictorial representation for the abstract substitutions involved, present sufficient conditions for compile-time detection of NSI based on the abstract substitution information, and describe a parallelization technique combining compile-time and run-time detection of NSI.

There are of course many more applications of global analysis in addition to automatic parallelization. Some recent specific research on this topic which takes advantage of the modular design of the Ciao system is described in [BdlBH$^+$01, MMNH01, MCH03, MCH04, APH04].

1.3 Main Contributions

The main contributions of this thesis are enumerated below. Some of these results have already been published and presented in international forums, in which case the relevant publications are mentioned explicitly. Also, the first mentioned contribution has been made in collaboration with other researchers in addition to the thesis supervisor. This is also explicitly mentioned below.

- Program analysis techniques based on abstract interpretation have received considerable attention, from the point of view of general frameworks and abstract domains. However, comparatively little attention has been given to the problems which arise when analysis of a full, practical dialect of the Prolog language is attempted, and only few, scattered solutions to these problems have been proposed before. We have proposed the first complete combination of analysis techniques (some known, some novel) which allow

analyzing in a correct way any program written using any of the features of ISO standard Prolog. This work has been performed in collaboration with Francisco Bueno and Germán Puebla, both from the CLIP group at the Technical University of Madrid (UPM), to which the PhD candidate also belongs. The work has been firstly presented at the Workshop on Static Analysis [BCHP95] associated to the International Conference of Logic Programming (ICLP) in 1995 and an improved version published at the European Symposium on Programming (ESOP) in 1996 [BCHP96].

- Existing standard Logic Programming systems had characteristics which made difficult in one way or another their effective global analysis. Furthermore, extending those systems was sometimes difficult and other times contorted, making them hardly adequate for exploring upcoming technologies and models, because the impossibility of a modular redefinition of the system/language. We have designed a module system for Prolog supportive of global analysis, optimization and verification, highly extensible and reconfigurable, which we have applied to the Ciao system. The work has been presented in the ICLP'99 workshop on Parallelism and Implementation of (C)LP Systems [CH99a], in Electronic Notes in Theoretical Computer Science, within a Special Issue on Parallelism and Implementation of (C)LP Systems [CH00c], and finally an improved version was presented in the International Conference on Computational Logic CL2000 [CH00a].

- Within the Ciao system, we have developed an implementation of higher-order in a standard, untyped, modular logic programming language, with original features, which includes an implementation of `call/N` and *predicate abstractions*.

- We have devised and implemented a modular, parametric code processing framework, providing separate compilation and automatic dependence handling, and designed to natively support global analysis, optimization and verification. Based on that framework, we have implemented a compiler which apart from the characteristics of the framework is able to produce

several kinds of executables, some of them based on novel concepts, such as active modules. The implementations are integrated in the Ciao system. This work has been presented in the ICLP'99 workshop on Parallelism and Implementation of (C)LP Systems [CH99b], and in Electronic Notes in Theoretical Computer Science, within a Special Issue on Parallelism and Implementation of (C)LP Systems [CH00b].

- We have developed the first technique for automatic parallelization of logic programming languages based on the notion of "non-strict" independence (NSI). NSI is a more relaxed notion than the traditional notion of "strict" independence (SI), which ensures the same efficiency properties of the parallelization but allows considerable more parallelism than SI. We have found sufficient conditions for compile-time detection of NSI based on the Sharing+Freeness abstract interpretation analysis, devised a technique for computing run-time independence checks (to allow more parallelism), and developed a parallelization algorithm based on pairwise independence properties between goals. This work was presented in the 1994 International Static Analysis Symposium [CH94], and in my Masters Thesis [CG93].

The complete Ciao system, as well as other related software can be downloaded from the CLIP Software WWW site:

`http://www.clip.dia.upm.es/Software`

1.4 Other Work not Included in the Thesis

During the time of my research which led to this thesis, other works were also carried out which are not included herein. The reason of this omission is that these works were considered somehow less related to the main objectives of the thesis, and their inclusion would make it too long and also perhaps lose some of its coherence. On the other hand most of these works have used the results of the thesis, including the design of the Ciao module system and the Ciao compiler. We list here the publications which came out from this omitted work, together with their abstracts:

- **On The Uses of Attributed Variables in Parallel and Concurrent Logic Programming Systems** [HCC95].

  Incorporating the possibility of attaching attributes to variables in a logic programming system has been shown to allow the addition of general constraint solving capabilities to it. This approach is very attractive in that by adding a few primitives any logic programming system can be turned into a generic constraint logic programming system in which constraint solving can be user defined, and at source level – an extreme example of the "glass box" approach. In this paper we propose a different and novel use for the concept of attributed variables: developing a generic parallel/concurrent (constraint) logic programming system, using the same "glass box" flavor. We argue that a system which implements attributed variables and a few additional primitives can be easily customized at source level to implement many of the languages and execution models of parallelism and concurrency currently proposed, in both shared memory and distributed systems. We illustrate this through examples and report on an implementation of our ideas.

- **Distributed Concurrent Constraint Execution in the Ciao System** [CH95a, CH96]

  This paper describes the current prototype of the distributed Ciao system. It introduces the concepts of "teams" and "active modules" (or active objects), which conveniently encapsulate different types of functionalities desirable from a distributed system, from parallelism for achieving speedup to client-server applications. It presents the user primitives available and describes their implementation. This implementation uses attributed variables and, as an example of a communication abstraction, a blackboard that follows the Linda model. The functionalities of the system are

illustrated through examples, using the implemented primitives. The implementation of most of the primitives is also described in detail.

- **Agent Programming in Ciao Prolog** [BCC+01].

  The agent programming landscape has been revealed as a natural framework for developing "intelligence" in AI. This can be seen from the extensive use of the agent concept in presenting (and developing) AI systems, the proliferation of agent theories, and the evolution of concepts such as agent societies (social intelligence) and coordination.

  For programming purposes, and in particular for AI programming, one would need a programming language/system that allows to reflect the nature of agents in the code: to map code to some abstract entities (the "agents"), to declare well-defined interfaces between such entities, their individual execution, possibly concurrent, possibly distributed, and their synchronization, and, last but not least, to program intelligence.

  It is our thesis that for the last purpose above the best suited languages are logic programming languages. It is arguably more difficult (and unnatural) to incorporate reasoning capabilities into, for example, an object oriented language than to incorporate the other capabilities mentioned above into a logic language. Our aim is, thus, to do the latter: to offer a logic language that provides the features required to program (intelligent) agents comfortably.

  The purpose of this talk, then, is not to introduce sophisticated reasoning theories or coordination languages, but to go through the (low-level, if you want) features which, in our view, provide for agent programming into a (high-level) language, based on logic, which naturally offers the capability of programming reasoning.

  The language we present is Ciao, whose relevant features are out-

lined. Most of them have been included as language-level extensions, thanks to the extensibility of Ciao. Hopefully, the Ciao approach will demonstrate how the required features can be embedded in a logic programming language in a natural way, both for the implementor and for the programmer.

- **A Generic Persistence Model for CLP Systems (and two useful implementations)** [CGC+03b, CGC+03a, CGC+04].

  This paper describes a model of persistence in (C)LP languages and two different and practically very useful ways to implement this model in current systems. The fundamental idea is that persistence is a characteristic of certain dynamic predicates (which encapsulate state). The main effect of declaring a predicate persistent is that the dynamic changes made to such predicates *persist* from one execution to the next one. After proposing a syntax for declaring persistent predicates, a simple, file-based implementation of the concept is presented and some examples shown. It is then argued that the concept developed provides the most natural way to interface with databases. Such an interface to a relational database is then developed as simply one more implementation alternative to the simple, file-based approach. The abstraction of the concept of persistence from its implementation allows developing applications which can store data alternatively on files or databases with only a few simple changes to a declaration stating the location and modality used for persistent storage. The paper presents the model, the implementation approach in both the cases of using files and relational databases, a number of optimizations of the process (using information obtained from static global analysis and goal clustering), and performance results from an implementation of these ideas.

- **Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library** [CH97, CH01].

  We discuss from a practical point of view a number of issues involved in writing distributed Internet and WWW applications using LP/CLP systems. We describe PiLLoW, a *public-domain* Internet and WWW programming library for LP/CLP systems that we have designed in order to simplify the process of writing such applications. PiLLoW provides facilities for accessing documents and code on the WWW; parsing, manipulating and generating HTML and XML structured documents and data; producing HTML forms; writing form handlers and CGI-scripts; and processing HTML/XML templates. An important contribution of PiLLoW is to model HTML/XML code (and, thus, the content of WWW pages) as terms. The PiLLoW library has been developed in the context of the Ciao Prolog system, but it has been adapted to a number of popular LP/CLP systems, supporting most of its functionality. We also describe the use of concurrency and a high-level model of client-server interaction, Ciao Prolog's *active modules*, in the context of WWW programming. We propose a solution for client-side downloading and execution of Prolog code, using generic browsers. Finally, we also provide an overview of related work on the topic.

- **Design of a Generic, Homogeneous Interface to Relational Databases** [CHGT98a].

  This document reports on the overall design of a generic, homogeneous interface between Prolog (specifically the Ciao system) and external storage, as files or relational databases. This interface provides persistence of the data managed by a Prolog program between runs of the program. It is also a very high level interface to existing relational databases which allows viewing such

databases simply as collections of Prolog predicates. This document describes informally the overall design while a more formal companion document describes in more detail the actual specification of the interface.

- **WOF Design** [CGT98].

  This document reports on the design of two formalisms used in the RadioWeb project: a declarative language (WOFs – Web Object Formatters) to map abstract descriptions of on-line data to descriptions of the logical structure of objects in a Web site, and the language WSDL (Web Structure Description Language) to capture and describe the logical structure (both inter- and intra-page) of a Web site, abstracting away from its layout and stylistic features. The WOF interpreter converts WOFs to WSDL terms by accessing the relevant data on the databases.

- **Layout and Style (LaSt) Language** [CHGT98b].

  This document reports on the preliminary design of the LaSt (Layout and Style) language used in the RadioWeb project. The LaSt language is a constraint-based language, which defines a mapping of WSDL [CGT98] terms to a concrete layout. The mapping of WSDL terms is done in two stages: the first stage maps the WSDL term to an abstract layout language, and the second stage maps the abstract layout language to a concrete layout (*e.g.*, HTML, XML+CSS, etc.). This document presents the initial design of the LaSt language and examples of its use. It also compares the LaSt approach with related technologies such as DSSS, CSS and more recent developments.

Part I

Language Design

Chapter 2

Global Analysis of Standard Prolog Programs

Abstract interpretation-based data-flow analysis of logic programs is, at this point, relatively well understood from the point of view of general frameworks and abstract domains. On the other hand, comparatively little attention has been given to the problems which arise when analysis of a full, practical dialect of the Prolog language is attempted, and only few solutions to these problems have been proposed to date. Existing proposals generally restrict in one way or another the classes of programs which can be analyzed. This chapter attempts to fill this gap by considering a full dialect of Prolog, essentially the ISO standard, pointing out the problems that may arise in the analysis of such a dialect, and proposing a novel combination of known and novel solutions that together allow the correct analysis of arbitrary programs which use the full power of the language.

2.1 Introduction

Global program analysis, generally based on abstract interpretation [CC77], is becoming a practical tool in logic program compilation, in which information about calls, answers, and substitutions at different program points is computed statically [HWD92, VD92, MH92, SCWY91, BdlBH94, Deb89b, Bru91, Deb92,

MSJ94, LV94]. Most proposals to date have concentrated on general frameworks and suitable abstract domains. On the other hand, comparatively little attention has been given to the problems which arise when analysis of a full, practical language is attempted. Such problems relate to dealing correctly with all builtins, including meta-logical, extra-logical, and dynamic predicates (whose code is modified during execution). Often, problems also arise because not all the program code is accessible to the analysis, as is the case for some builtins (meta-calls), some predicates (multifile and/or dynamic), and some programs (multifile or modular).

Implementors of the analyses obviously have to somehow deal with such problems, and some of the implemented analyses provide solutions for some problems. However, the few solutions which have been published to date [VD92, Deb89a, HWD92, MH92, LRV94] generally restrict the use of builtin predicates in one way or another (and thus the class of programs which can be analyzed).

The work presented in this chapter attempts to fill this gap. We consider the correct (data-flow) analysis of a *full* dialect of Prolog. For concreteness, we essentially follow the ISO standard [PRO94, DEDC96]. Our purpose is to review the features of the language which pose problems to global analysis and propose alternative solutions for dealing with these features. The most important objective is obviously to achieve correctness, but also as much accuracy as possible. Since arguably the main problem in static analysis is having dynamic code, which is not available at compile-time, we first propose a general solution for solving the problems associated with features such as dynamic predicates and meta-predicates, and consider other alternative solutions. The proposed alternatives are a combination of known solutions when they are useful, and novel solutions when the known ones are found lacking. The former are identified by giving references.

One of the motivations of our approach is that we would like to accommodate at the same time two types of users. First, the naive user, which would like analysis to be as transparent as possible. Second, we would also like to cater for the advanced user, which may like to guide the analysis in difficult places in order to obtain better results. Thus, for each feature, we will propose solutions that require no user input, but we will also propose solutions that allow the

user to provide input to the analysis process. This requires a clear interface to the analyzer at the program text level. Furthermore, this need also arises when expressing the information gathered by the different analyses supported. We solve this by proposing an interface, in the form of *annotations*, which is useful not only as a two-way communication between the user and the compiler, but also for the cooperation among different analysis tools and for connecting analyses with other modules of the compiler. Annotations (which we currently call *assertions*) are syntactic constructions which allow expressing properties of programs.

After some necessary preliminaries in Section 2.2, we propose several novel general solutions to deal with the analysis of dynamic programs in Section 2.3. A set of program annotations which can help in this task is proposed in Section 2.4. We then revise our and previous solutions to deal with each of the language features in Section 2.5, except for modules and multifile programs, which are discussed in Section 2.6. There, we propose a solution based on incremental analysis, and another one based on our program annotations. We conclude with Section 2.7.

We argue that the proposed set of solutions is the first one to allow the correct analysis of arbitrary programs which use the full power of the language without input from the user (while at the same time allowing such input if so desired).

2.2 Preliminaries and Notation

For simplicity we will assume that the abstract interpretation based analysis is constructed using the "Galois insertion" approach [CC77], in which an abstract domain is used which has a lattice structure, with a partial order denoted by $\sqsubseteq$, and whose top value we will refer to by $\top$, and its bottom value by $\bot$. We will refer to the least upper bound (lub) and greatest lower bound (glb) operators in the lattice by $\sqcup$ and $\sqcap$, respectively. The abstract computation proceeds using abstract counterparts of the concrete operations, the most relevant ones being unification (mgu^α) and composition ($\circ^\alpha$), which operate over abstract substitutions (α). Abstract unification is however often also expressed as a function $unify^\alpha$ which computes the abstract mgu of two concrete terms in the presence

of a given abstract substitution.

Usually, a *collecting* semantics is used which attaches one or more (abstract) substitutions to program points (such as, for example, the point just before or just after the call of a given literal — the call and success substitutions for that literal). A *goal dependent* analysis associates abstract success substitutions to specific goals, in particular to call patterns, i.e. pairs of a goal and an abstract call substitution which expresses how the goal is called. Depending on the granularity of the analysis, one or more success substitutions can be computed for different call patterns at the same program point. *Goal independent* analyses compute abstract success substitutions for generic goals, regardless of the (abstract) call substitution. In the following, we will use "substitution" and "abstract substitution" interchangeably, when it is clear from the context.

In general we will concentrate on top-down analyses, since they are at present the ones most frequently used in optimizing compilers. However, we believe the techniques proposed are equally applicable to bottom-up analyses. In the text, we consider in general goal dependent analyses, but point out solutions for goal independent analyses where appropriate (see, e.g., [GDL92, GGL94, CdlBBH94]).

The pairs of call and success patterns computed by the analysis, be it top-down or bottom-up, goal dependent or independent, will be denoted by $AOT^{\alpha}(P)$ for a given program P. A *most general goal pattern* (or simply "goal pattern," hereafter) of a predicate is a *normalized* goal for that predicate, i.e. a goal whose predicate symbol and arity are those of the predicate and whose arguments are distinct variables. In goal dependent analyses, for every call pattern of the form $(goal\_pattern, call\_substitution)$ of a program P there are one or more associated success substitutions which will be denoted hereafter by $AOT^{\alpha}(P, call\_pattern)$. The same holds for goal independent analysis, where the call pattern is simply reduced to the goal pattern. By *program* we refer to the entire program text that the compiler has access to, including any directives and annotations.

2.3 Static Analysis of Dynamic Program Text

A main problem in statically analyzing logic programs is that not all of the code that would actually be run is statically accessible to the analysis. This can occur either because the particular calls occuring at some places are dynamically constructed, or because the code defining some predicates is dynamically modified. The following problems appear:

1. How to compute success substitutions for the calls which are not known; we call this problem the *success substitution problem*, and

2. How to determine calls and call substitutions which may appear from the code which is not known; we call this problem the *extra call pattern problem*.

Consider the following program, to be analyzed with entry point `goal`. The predicate `p/2` is known to be dynamic, and may thus be modified at run-time.

```
goal:- ..., X=a, ..., p(X,Y), ...

:- dynamic p/2.
p(X,Y):- q(X,Y).

q(X,Y).
l(a,b).
```

Assume that the call pattern of the goal `p(X,Y)` computed by the analysis indicates that `X` is a ground term and `Y` a free variable. If we do not consider the possibility of run-time modifications of the code, the success pattern for `p(X,Y)` is the same as the call pattern (since predicate `q/2` does not change its arguments). Also, since no calls exist to `l/2`, its definition is dead code. Assume now that a clause "`p(X,Y):- l(X,Y).`" is asserted at run-time. The previous analysis information is not correct for two reasons. First, the success pattern of `p(X,Y)` should now indicate that `Y` may be ground (success substitution problem). Second, a call for `l/2` now occurs which has not been considered in the previous analysis (extra call pattern problem).

The first problem is easier to solve: using appropriate topmost substitutions. We call an abstract substitution α *topmost* w.r.t. a tuple (set) of variables $\vec{x}$ iff $vars(\alpha) = \vec{x}$ and for any other substitution α' such that $vars(\alpha') = \vec{x}$, $\alpha' \sqsubseteq \alpha$. An abstract substitution α referring to variables $\vec{x}$ is said to be *topmost of* another substitution α', referring to the same variables, iff $\alpha \equiv \alpha' \circ^{\alpha} \alpha''$, where α'' is the topmost substitution w.r.t. $\vec{x}$. Therefore, for a given call substitution, the topmost abstract substitution w.r.t. it is the most accurate approximation which solves the success substitution problem. This is in contrast to roughly considering $\top$ or just giving up in the analysis. Topmost substitutions are preferred, since they are usually more accurate for some domains. For example, if a variable is known to be ground in the call substitution, it will continue being ground in the success substitution.

Note that this is in fact enough for goal independent analyses, for which the second problem does not apply. However, for goal dependent analyses the second problem needs to be solved in some way. This problem is caused by the impossibility of statically computing the subtree underlying a given call, either because this call is not known (it is statically undetermined), or because not all of the code defining the predicate for that call is available. Therefore, since from these subtrees new calls (and new call patterns) can appear, which affect other parts of the program, the whole analysis may not be correct.

There is a first straightforward solution to the extra call pattern problem. It can be tackled by simply assuming that there are unknown call patterns, and thus any of the predicates in the program may be called (either from the undetermined call or from within its subtree) and in any possible way. This means that analysis may still proceed but topmost call patterns must be assumed for all predicates. This is similar to performing a goal independent analysis and it may allow some optimizations, but it will probably preclude others. However, if program multiple specialization [Win92, PH95, VD92, Pue97] is done, a non-optimized version of the program should exist (since all the predicates in the program must be prepared to receive any input value), but other optimized versions could be inferred.

We illustrate the points discussed so far using again the previous example. To solve the success substitution problem for `p/2` we can:

(a) assume a topmost substitution w.r.t. `X` and `Y`, which will indicate that nothing is known of these two variables; or

(b) assume the topmost substitution w.r.t. the call substitution, which will indicate that nothing is known of `Y`, but still `X` is known to be ground.

To solve the extra call pattern problem we can assume new call patterns with topmost substitutions for all predicates in the program, since the asserted clause is not known during analysis, as proposed above. However, we can also perform the transformation that we will propose below, which will isolate the problem to predicate `l/2`, which is the only one affected.

We propose indeed a second complete solution which is general enough and very elegant, with the only penalty of some cost in code size. The key idea is to compile essentially two versions of the program — one that is a straightforward compilation of the original program, and another that is analyzed assuming that the only possible calls to each predicate are those that appear explicitly in the program. This version will contain all the optimizations, which will be performed ignoring the effect of the undetermined calls. Still, in the other version, any optimizations possible with a goal independent analysis, or a topmost call pattern goal dependent analysis, may be introduced. Calling from undetermined calls into the more optimized version of the program (which will possibly be unprepared for the call patterns created by such calls) is avoided by making such calls call the less optimized version of the program. This will take place automatically because in the less optimized version the original predicate names are used, whereas in the more optimized version names of static predicates will be renamed in an appropriate way (we will assume for simplicity that it is by using the prefix "`opt_`"). The terms that will be built at run-time will use the names of the original predicates, thus avoiding the access to predicates in the more optimized version from the dynamically asserted clauses. Thus, correctness of a transformation such as the following is guaranteed. Assume that `call(X)` is an undetermined call. If a clause such as the first one appears in the program, the second one is added:

```
p(...) :-
        q(...),
        call(X),
        r(...).

opt_p(...) :-
        opt_q(...),
        call(X),
        opt_r(...).
```

The top-level rewrites calls which have been declared as entry points to the program so that the optimized version is accessed. Note that this also solves (if needed) the general problem of answering queries that have not been declared as entry points: they simply access the less optimized version of the program. If the top-level does also check the call patterns, then it guarantees that only the entry patterns used in the analysis will be executed. For the declared entry patterns, execution will start in the optimized program and will move to the original program to compute a resolution subtree each time an undetermined call is executed. Upon return from the undetermined call, execution will go back to the optimized program.

We shall see how this solution can be applied both to the case of meta-predicates and to that of dynamic predicates, allowing full optimizations to be performed in general to "dynamic" programs. The impact of the optimizations performed in the renamed copy of the program will depend on the time that execution stays in each of the versions. Therefore, the relative computational load of undetermined calls w.r.t. the whole program will condition the benefits of the optimizations achieved. The only drawback with this solution is that it implies keeping two full copies of the program, although only in case there are undetermined calls. This will be attenuated if using a modular system with appropriate features, as is in the case of Ciao, because only those modules in which dynamic addition of clauses is used will have to be duplicated (see Chapter 3 for details). Also, using multiple specialization, available in the Ciao preprocessor,

the unneeded parts of the duplicated code could be eliminated. In any case, to cope with situations where code space is a pressing issue, the user should be given the choice of turning this copying on and off.

2.4 Program Annotations

Annotations[1] are assertions regarding a program that are introduced as part of its code. Annotations refer to a given program point. We consider two general classes of program points: points inside a clause (such as, for example, before or after the execution of a given goal — the "goal level") and points that refer to a whole predicate (such as, for example, before entering or after exiting a predicate — the "predicate level"). For our purposes here, at all levels annotations describe properties of the variables that appear in the program. We will call the descriptions of such properties *declarations*. There are at least two ways of representing declarations: "property oriented" and "abstract domain oriented". In a property oriented annotation framework, there are declarations for each property a given variable or set of variables may have. Examples of such declarations are:

`nonvar(X)`	`X` is bound to a non-variable term
`term(X,r(Y))`	`X` is bound to term `r(Y)`
`depth(X,r/1)`	`X` is bound to a term with principal functor `r/1`

The property oriented approach presents two advantages. First, it is easily extensible, provided one defines the semantics for the new properties one wants to add. And second, it is also independent from any abstract domain for analysis. One only needs to define the semantics of each declaration, and, for each abstract domain, a translation into the corresponding abstract substitutions. For concreteness, and in order to avoid referring to any abstract domain in particular, we will use such a framework herein.

An alternative solution is to define declarations in an abstract domain oriented way. For example, for the sharing domain [JL89, MH89, MH92, JL92]:

`sharing([[X],[Y,Z]])`	shows the sharing pattern among variables X, Y, Z

[1]The annotations proposed here (and, originally, in [BCHP96] and previous papers) are a subset of what has later become the Ciao Assertion Language [PBH00b]

This is a simple enough solution but has the disadvantage that the meaning of such domains is often difficult for users to understand. Also, the interface is bound to change any time the domain changes. It has two other disadvantages. The semantics and the translation functions mentioned above have to be defined pairwise, i.e. one for each two different domains to be communicated. And, secondly, there can exist several (possibly overlapping) properties declared, one for each different domain. In the property oriented approach, additional properties that several domains might take advantage of are declared only once. In any case, both approaches are compatible via the *syntactic* scheme we propose.

2.4.1 Predicate Level: Entry Annotations

One class of predicate level annotations are `entry` annotations. They are specified using a directive-style syntax, as follows:

`:- entry` *goal_pattern* `:` *declaration*`.`

These annotations state that calls to that predicate with the given abstract call substitution may exist at execution time. For example, the following annotation states that there can be a call to predicate `p/2` in which its two arguments are ground:

```
:- entry p(X,Y) : (ground(X),ground(Y)).
```

Entry annotations and goal dependent analysis. A crucial property of `entry` annotations, which makes them useful in goal dependent analyses, is that they must be *closed with respect to outside calls*. By this we mean that no call patterns for a given predicate other than those specified by the annotations in the program may occur from outside the program text. I.e., the list of `entry` annotations for a single predicate includes all calls that may occur to that predicate in the program, apart from those which arise from the literals explicitly present in the program text. Thus, for now, we assume that they define *all* entry points, and optionally, their call patterns. Obviously this is not an issue in goal independent analyses.

Entry annotations and multiple program specialization. If analysis is multivariant it is often convenient to create different versions of a predicate (multiple specialization). This allows implementing different optimizations in each version. Each one of these versions generally receives an automatically generated unique name in the multiply specialized program. However, in order to keep the multiple specialization process transparent to the user, whenever more than one version is generated for a predicate which is a declared entry point of the program (and, thus, appears in an `entry` directive), the original name of the predicate is reserved for the version that will be called upon program query. If more than one `entry` annotation appears for a predicate and different versions are used for different annotations, it is obviously not possible to assign to all of them the original name of the predicate. There are two solutions to this. The first one is to add a front end with the exported name and run-time tests to determine the version to use. However, this implies run-time overhead. As an alternative we allow the `entry` directive to have one more argument which indicates the name to be used for the version corresponding to this entry point. For example, given:

```
:- entry mmultiply(A,B,C) : ground([A,B]) ; mmultiply_ground.
:- entry mmultiply(A,B,C) : true ; mmultiply_any.
```

if these two entries originate different versions, they would be given different names. If two or more versions such as those above are collapsed into one, this one will get the name of any of the entry points and, in order to allow calls to all the names given in the annotations, binary clauses will be added to provide the other entry points to that same version.

2.4.2 Predicate Level: Trust Pred Annotations

In addition to the more standard `entry` annotations we propose a different kind of annotations at the predicate level, which take the following form: [2]

`:- trust pred` *goal_pattern* `:` *call_props* `=>` *success_props*`.`

[2]Note that pred declarations can be expressed in terms of the more elementary calls and success declarations of [PBH00b], where pred declarations are just syntactic sugar.

Declarations in `trust` pred annotations put in relation the call and the success patterns of calls to the given predicate. These annotations can be read as follows: if a literal that corresponds to *goal_pattern* is executed and *call_props* holds for the associated call substitution, then *success_props* holds for the associated success substitution. Thus, these annotations relate abstract call and success substitutions. Note that *call_props* can be empty (i.e., `true`). In this way, properties can be stated that must always hold for the success substitution, no matter what the call substitution is. This is useful also in goal independent analyses (and in this case it is equivalent to the "omode" declaration of [HWD92]).

Let $(p(\vec{x}), \alpha)$ denote the call pattern and α' the success substitution of a given `trust` annotation of a program P. The semantics of `trust` implies that $\forall \alpha_c \ (\alpha_c \sqsubseteq \alpha \Rightarrow AOT^{\alpha}(P, (p(\vec{x}), \alpha_c)) \sqsubseteq \alpha')$. I.e., for all call substitutions approximated by that of the given call pattern, their success substitutions are approximated by that of the annotation. For this reason, the compiler will "trust" them. Note that this implies that, in contrast to `entry` annotations, several `trust` annotations are assumed to form a conjunction. This justifies their consideration of "extra" information, and thus and in contrast to `entry` annotations, the list of `trust` annotations of a program does *not* have to be closed w.r.t. all possible call patterns occurring in the program.

One of the main uses of `trust` annotations is in describing predicates that are not present in the program text. For example, the following annotations describe the behavior of the predicate `p/2` for two possible call patterns:

```
:- trust pred p(X,Y) : (ground(X),var(Y)) => (ground(X),ground(Y)).
:- trust pred p(X,Y) : (var(X),ground(Y)) => (var(X),ground(Y)).
```

This would allow performing the analysis even if the code for `p/2` is not present. In that case the corresponding success information in the annotation can be used ("trusted") as success substitution.

In addition, `trust` annotations can be used to improve possible imprecise results of the analysis. However, note that this does not save analyzing the predicate for the corresponding call pattern, since the abstract underlying subtree may contain call patterns that do not occur elsewhere in the program.

If we analyze a call pattern for which a `trust` annotation exists, two abstract success patterns will be available for it: that computed by the analysis (say α_s) and that given by the `trust` annotation (say α', for a call substitution α). As both must be correct, their intersection (which may be more accurate than any of them) must also be correct. The intersection among abstract substitutions (whose domain we have assumed has a lattice structure) is computed with the glb operator, $\sqcap$. Therefore, $AOT^{\alpha}(P,(p(\vec{x}),\alpha_c)) = \alpha_s \sqcap \alpha'$, provided that $\alpha_c \sqsubseteq \alpha$. Since $\forall\alpha_s\forall\alpha'\ (\alpha_s \sqcap \alpha' \sqsubseteq \alpha_s \wedge \alpha_s \sqcap \alpha' \sqsubseteq \alpha')$ correctness of the analysis within the `trust` semantics is guaranteed, i.e. $AOT^{\alpha}(P,(p(\vec{x}),\alpha_c)) \sqsubseteq \alpha'$ and $AOT^{\alpha}(P,(p(\vec{x}),\alpha_c)) \sqsubseteq \alpha_s$. However, if their informations are incompatible, their intersection is empty, and $\alpha_s \sqcap \alpha' = \bot$. This indicates an error (if $\alpha_s \neq \bot$ and also $\alpha' \neq \bot$), because the analysis information must be correct, and the same thing is assumed for the `trust` information. The analysis should give up and warn the user.

A similar scheme can be used to check the mutual consistency of annotations provided by the user. The result of the glb operation between inconsistent annotations will be $\bot$. Also, note that, in addition to improving the substitution at the given point, the trusted information can be used to improve previous patterns computed in the analysis. This might be done by "propagating" the information backwards in the analysis process [KL02].[3]

2.4.3 Goal Level: Pragma Annotations

Annotations at the goal level refer to the state of the variables of the clause just at the point where the annotation appears: between two literals, after the head of a clause or after the last literal of a clause.[4] We propose adding extra-literals

[3]These early observations regarding error detection have later been used to develop a full framework for program validation and debugging based on abstract interpretation [BDD+97, HPB99, PBH00a, PBH00b]

[4]Similar annotations can be used at other levels of granularity, from between head unifications to even between low level instructions, but we will limit the discussion for concreteness to goal-level program points.

which enclose all necessary information referring to a given program point in a clause. It takes the form:

`...,` $goal_1$`, trust(`*props*`),` $goal_2$`, ...`

where the information in the `trust` literal should be valid before calling $goal_2$ and also after calling $goal_1$, that is, at the success point for $goal_1$ and at the call point of $goal_2$, within that clause.

The information given by `trust` literals can refer to any of the variables in the clause. The information is expressed using the same kind of properties as in the predicate level annotations. This allows a uniform format for the declarations of properties in annotations at both the predicate and the goal level. These annotations are related to predicate level `trust` annotations in the sense that they give information that should be trusted by the compiler. Therefore, they have similar uses and a similar treatment that them.

2.5 Dealing with Concrete Features of Standard Prolog

In the previous sections we have presented a number of user annotations and already addressed a number of general issues in the practical analysis of programs. In this section we discuss different solutions for analyzing the different concrete features of the full standard Prolog language. In order to do so we have divided the complete set of builtins offered by the language into several classes.

2.5.1 Builtins as Abstract Functions

Many Prolog builtins can be dealt with efficiently and accurately during analysis by means of functions which capture their semantics. Such functions provide an (as accurate as possible) abstraction of every success substitution for any call to the corresponding builtin. This applies also to goal independent analyses, with minor modifications. It is interesting to note that the functions that describe builtin predicates are very similar in spirit to `trust` annotations. This is not

surprising, if builtins are seen as Prolog predicates for which the code is not available. Since most of the treatment of builtins is rather straightforward and specialized, the presentation is very brief, concentrating on the more interesting cases of meta-predicates and dynamic predicates. In order to avoid reference to any particular abstract domain any functions described will be given in terms of simple-minded `trust` annotations. For the reader interested in the details, the source code for the PLAI analyzer (available by ftp from `clip.dia.fi.upm.es` and currently contained in CiaoPP [HBPLG99, HPBLG03]) contains detailed functions for all Prolog builtins and for a large collection of well known abstract domains. For a description of such functions for some builtins in a different domain see e.g. [CFR92].

Control flow predicates include `true` and `repeat`, which can simply be treated as identity (i.e., they can be simply ignored). The abstraction of `fail` and `halt` is $\bot$. For cut (`!`) it is also possible to use the identity function (i.e., ignore it). This is certainly correct in that it only implies that more cases than necessary will be computed in the analysis upon predicate exit, but may result in a certain loss of accuracy (specially if *red cuts* –those which modify the meaning of a program– are used). This can be addressed by using a semantics which keeps track of sequences, rather than sets, of substitutions, as shown in [LRV94]. Finally, exception handling can also be included in this class. The methods used by the different Prolog dialects for this purpose have been unified in the Prolog standard into two builtins: `catch` and `throw`. We propose a method for dealing with this new mechanism: note that, since analysis in general assumes that execution can fail at any point, literals of the form `catch(Goal,Catcher,Recovery)` (where execution starts in `Goal` and backtracks to `Recovery` if the exception described by `Catcher` occurs) can be safely approximated by the disjunction `(Goal;Recovery)`, and simply analyzed as a meta-call. The correctness of this transformation is based on the fact that no new control paths can appear due to an exception, since those paths are a subset of the ones considered by the analysis when it assumes that any goal may fail. The builtin `throw`, which explicitly raises an exception, can then be approximated by directly mapping it to failure, i.e. $\bot$.

The function corresponding to `=` is simply abstract unification. Specialized

versions of the full abstract unification function can be used for other builtins such as `\=`, `functor`, `arg`, `univ` (`=..`), and `copy_term`. Other term- and string-manipulation builtins are relatively straightforward to implement. Arithmetic builtins and base type tests such as `is`, `>`, `@>`, `integer`, `var`, `number`, etc., usually also have a natural mapping in the abstract domain considered. In fact, their incomplete implementation in Prolog is an invaluable source of information for the analyzer upon their exit (which assumes that the predicate did not fail — failure is of course always considered as an alternative). For example, their mappings will include relations such as "`:- trust pred is(X,Y) : true => (ground(X),ground(Y)).`". On the contrary, `==`, `\==`, and their arithmetic counterparts, are somewhat more involved, and are implemented (in the same way as with the term manipulation builtins above) by using specialized versions of the abstract unification function.

Output from the program does not directly pose any problem since the related predicates do not instantiate any variables or produce any other side effects beyond modifying external streams, whose effect can only be seen during input to the program. Thus, identity can again be used in this case. On the other hand, the external input cannot be determined beforehand. The main problem happens to be the success substitution problem. In the general case, analysis can always proceed by simply assuming topmost success substitutions in the domain.

The treatment of *directives* is somewhat peculiar. The directive `dynamic` is used to declare predicates which can be modified at run-time. Dynamic predicates will be considered in detail below. The directive `multifile` specifies that the definition of a predicate is not complete in the program. Multifile predicates can therefore be treated as either dynamic or imported predicates — see Section 2.6. The directives `include` and `ensure_loaded` must specify an accessible file, which can be read in and analyzed together with the current program. The directive `initialization` specifies new (concrete) entry points to the program.

2.5.2 Meta-Predicates

Meta-predicates are predicates which use other predicates as arguments. All user-defined meta-predicates are in this class but their treatment can be reduced to the treatment of the meta-call builtins they use. Such meta-calls are literals which call one of their arguments at run-time, converting at the time of the call a term into a goal. Builtins in this class are not only `call`, but also `bagof`, `findall`, `setof`, negation by failure, and `once` (single solution). Calls to the solution gathering builtins can be treated as normal (meta-)calls since most analyzers are "collecting" in the sense that they always consider all solutions to predicates. Negation by failure (`\+`) can be defined in terms of `call` and cut, and can be dealt with by combining the treatment of cut with the treatment of meta-calls. Single solution (`once`) can be dealt with in a similar way since it is equivalent to "`once(X) :- call(X), !.`".

Since meta-call builtins convert a term into a goal, they can be difficult to deal with if it is not possible to know at compile-time the exact nature of those terms [Deb89a, HWD92]. In particular, the success substitution problem for the meta-call appears, as well as the extra call pattern problem, within the code defining the corresponding predicate, and for the possible calls which can occur from such code. Both problems can be dealt with using the techniques in Section 2.3. First, topmost call patterns can be used for all predicates in the program, and multiple specialization will create the copies of predicates as needed; second, and alternatively, the renaming transformation can also be applied. In this case meta-calls that are fully determined either by declaration or as a result of analysis, and incorporated into the program text will call the more optimized version. Analysis will have taken into account the call patterns produced by such calls since they would have been entered and analyzed as normal calls. I.e., the following transformation will take place:

```
..., trust(term(X,p(Y))), call(X),...  ⟹  ..., opt_p(Y),...
```

Meta-calls that are partially determined, such as, for example,

```
..., trust(depth(X,p/1)), call(X), ...
```

are a special case. One solution is not to rename them. In that case they will

be treated as undetermined meta-calls. Alternatively, the solution above can be used. It is necessary in this case to ensure that the optimized program will be entered upon reaching a partially determined meta-call. This can be done dynamically, using a special version of `call/1` or by providing binary predicates which transform the calls into new predicates which perform a mapping of the original terms (known from the analysis) into the renamed ones. Using this idea the example above may be transformed into a new literal and a new clause, as follows:

```
..., opt_call(X),...          opt_call(p(X)) :- opt_p(X).
```

Undetermined meta-calls will not be renamed, and thus will call the original (less optimized) code. This fulfills the correctness requirement, since these calls would not have been analyzed, and therefore cannot be allowed to call the optimized code.

More precise solutions to both problems are possible if knowledge regarding the terms to be converted is available at compile-time. Thus, we can distinguish between:

- *Completely determined* meta-calls. These are calls in which the term (functor and arguments) is given in the program text (this is often the case for example in many uses of `bagof`, `findall`, `setof`, `\+`, and `once`), or can be inferred via some kind of analysis, as proposed in [Deb89a]. In the latter case they can even be incoporated into the program text before analysis. These calls can be analyzed in a straightforward way.

- *Partially determined* meta-calls. The exact term cannot be statically found, but at least its main functor can be determined by program analysis. Then, since the predicate that will be called at run-time is known, it is sufficient for analysis to enter only this predicate using the appropriate projection of the current abstract call substitution on the variables involved in the call.

- *Undetermined* meta-calls.

The first two classes distinguish subclasses of the *fully determined* predicates of [Deb89a], where certain interesting types of programs are characterized which

allow the static determination of this generally undecidable property. Relying exclusively on program analysis, as in [Deb89a], has the disadvantage that it restricts the class of programs which can be optimized to those which are fully determined. Our previous solution solves the general case.

There are other possible solutions to the general case. The first and simplest one is to issue a warning if an undetermined meta-call is found and ask the user to provide information regarding the meta-terms. This can be easily done via program-point `trust` annotations. For example, the following annotation:

```
..., trust(( term(X,p(Y)) ; term(X,q(Z)) )), call(X), ...
```

states that the term called in the meta-call is either `p(Y)` or `q(Z)`. Note also that this is in some way similar to giving entry mode information for the `p/1` and `q/1` predicates. This suggests another solution to the problem, which has been used before in MA3 [WHD88], in Aquarius [VD92], and in previous versions of the PLAI analyzer [BdlBCH93]. The idea (cast in the terms of our discussion) is to take the position that meta-calls are *external calls*. Then, since `entry` annotations have to be closed with respect to external calls it is the user's responsibility to declare any entry points and patterns to predicates which can be "meta-called" via `entry` annotations. Accuracy of the analysis will depend on that of the information supplied by the user. These solutions have the disadvantage of putting the burden on the user — something that we would like to avoid at least for naive users. The alternatives that we have proposed have the advantage that they are completely transparent to the user.

2.5.3 Database Manipulation and Dynamic Predicates

Database manipulation builtins include `assert`, `retract`, `abolish`, and `clause`. These builtins (with the exception of `clause`) affect the program itself by adding to or removing clauses from it. Predicates that can be affected by such builtins are called dynamic predicates and must usually be declared as such in modern Prolog implementations (and this is also the case in the ISO standard).

The potential problems created by the use of the database manipulation builtins are threefold. On the one hand, the extra call pattern problem ap-

pears again since the literals in the body of the new clauses that are added dynamically can produce new and different call patterns not considered during analysis. The success substitution problem also appears for literals which call dynamic predicates ("dynamic literals"). Even if abstract success substitutions can be computed from any static definition of the predicate which may be available at compile-time, it may change during program execution. On the other hand, there exists the additional problem of computing success substitutions for the calls to the database manipulation builtins `retract` and `clause`. We call this the *database builtin success substitution problem*. Next we propose solutions to the three problems mentioned above. Note that the builtin `clause` —which can be viewed as a special case of retract— does not modify the database and thus clearly only has the third problem.

Solving the extra call pattern problem. From the correctness point of view, the extra call pattern problem only arises from the use of `assert`, but not from the use of `abolish` or `retract`. These predicates do not introduce new clauses in the program, and thus they do not introduce any new call patterns. On the other hand, it is conceivable that more accuracy could be obtained if these predicates were analyzed more precisely since removing clauses may remove call patterns which in turn could make the analysis more precise. We discuss this in the context of incremental analysis at the end of the section.

The `assert` predicate is much more problematic in this respect, since it can introduce new clauses and through them new call patterns. The problem is compounded by the fact that asserted clauses can call predicates which are not declared as dynamic, and thus the effect is not confined to dynamic predicates. In any case, and as pointed out in [Deb89a], not all uses of `assert` are equally damaging. To distinguish these uses, we propose to divide dynamic predicates into the following types:

- `memo` only facts which are logical consequences of the program itself are asserted;

- `data` only facts are asserted, or, if clauses are asserted, they are never called

(i.e., only read with `clause` or `retract`);

- `local_call` the dynamic predicate only calls other dynamic predicates;
- `global_call`.

The first two classes correspond to the *unit-assertive* and *green-assertive* predicates of [Deb89a], except that we have slightly extended the unit-assertive type by also considering in this type arbitrary predicates which are asserted/retracted but never called. Clauses used in this way can be seen as just recorded terms: simply a set of facts for the predicate symbol `:-/2`.

The advantage of `data` predicates is that they are guaranteed to produce no new call patterns and therefore they are safe with respect to the extra call pattern problem.[5] This is also the case for `memo` predicates since they only assert facts.[6] If all dynamic predicates are of the `local_call` type, then the analysis of the static program is correct except for the clauses defining the dynamic predicates themselves. Analysis can even ignore the clauses defining such predicates. Optimizations can then be performed over the program text except for those clauses, which in any case may not be such a big loss since in some systems such clauses are not compiled, but rather interpreted.

While the classification mentioned above is useful, two problems remain. The first one is how to detect that dynamic procedures are in the classes that are easy to analyze (dynamic predicates in principle need to be assumed in the `global_call` class). This can be done through analysis for certain programs, as shown in [Deb89a], but, as in the case of meta-calls, this does not offer a solution in all cases.

The general case in which `global_call` dynamic predicates appear in the program is similar to that which appeared with undetermined meta-calls. In fact, the calls that appear in the bodies of asserted clauses can be seen as undetermined meta-calls, and similar solutions apply. Additionally, the static clauses of the

[5] Ciao provides "data predicates" as a special class of dynamic predicates, see Chapter 3

[6] Note however that certain analyses, and especially cost analyses which are affected by program execution time, need to treat these predicates specially.

dynamic predicates themselves are subject to the same treatment as the rest of the program, and therefore subject to full optimization. Clearly, this solution can be combined with the previous ones when particular cases can be identified.

Solving the dynamic literal success substitution problem. For the analysis of literals which call dynamic predicates, if only `abolish` and `retract` are used in the program the abstract success substitutions of the static clauses of the dynamic predicates are a safe approximation of the run-time success substitutions. However, a loss of accuracy can occur, as the abstract success substitution for the remaining clauses (if any) may be more particular. In the presence of `assert`, a correct (but possibly inaccurate) analysis is obtained by using appropriate topmost abstract substitutions. Finally, note that in the case of `memo` predicates (and for certain properties) this problem is avoided since the success substitutions computed from the static program are correct.

Solving the database builtin success substitution problem. This problem does not affect `assert` and `abolish` since the success substitution for calls to these builtins is the same as the call substitution. On the other hand, success substitutions for `retract` (and `clause`) are more difficult to obtain. However, appropriate topmost substitutions can always be safely used. In the special case of dynamic predicates of the `memo` class, and if the term used as argument in the call to `retract` or `clause` is at least partially determined, abstract counterparts of the *static* clauses of the program can be used as approximations in order to compute a more precise success substitution.

Dynamic analysis and optimization. There is still another, quite different and interesting solution to the problem of dynamic predicates, which is based on incremental global analysis [HPMS95, PH96, HPMS00]. Note that in order to implement `assert` some systems include a copy of the full compiler at runtime. The idea would be to also include the (incremental) global analyzer and the analysis information for the program, computed for the static part of the program. The program is in principle optimized using this information but the

optimizer is also assumed to be incremental. After each non-trivial assertion or retraction (some cases may be treated specially) the incremental global analysis and optimizer are rerun and any affected parts of the program reanalyzed (and reoptimized). This has the advantage of having fully optimized code at all times, at the expense of increasing the cost of calls to database manipulation predicates and of executable size. A system along these lines has been built by our group for a parallelizing compiler [BdlBH99]. The results presented in [HPMS95, PH99b] show that such a reanalysis can be made in a very small fraction of the normal compilation time.

2.6 Program Modules

The main problem with studying the impact of modularity in analysis (and the reason we have left the issue until this section) is the lack of an agreed de-facto standard. There have been many proposals for module systems in logic programming languages (see [BLM94]). We will assume here that the module interface is *static*, i.e. each module in the program must declare the procedures it exports,[7] and imports. The `module` directive will be used for this.

As already pointed out in [HWD92], `module` directives provide the entry points for the analysis of a module for free. Thus, as far as entry points are concerned, only exported predicates need be considered. They can be analyzed using the substitutions declared in the `entry` annotations if available, and topmost otherwise. The analysis of literals which call imported predicates requires new approaches, some of which are discussed in the following paragraphs. One advantage of modules is that they help encapsulate the propagation of complex situations such as with `global_call` dynamic predicates.

[7] This is in contrast with other module systems used in some Prolog implementations that allow entering the code in modules at arbitrary points other than those declared as exported. This defeats the purpose of modules. We will not discuss such module systems since the corresponding programs in general need to be treated as non modular programs from the point of view of analysis.

Compositional Analysis. Modular analyses based on compositional semantics (such as, for example, that of [CDG93]) can be used to analyze programs split in modules. Such analyses leave the abstract substitutions for the predicates whose definitions are not available *open*, in the sense that some representation of the literals and their interaction with the abstract substitution is incorporated as a handle into the substitutions themselves. Once the corresponding module is analyzed and the (abstract) semantics of such open predicates known, substitutions can be composed via these handles. The main drawback of this interesting approach is that the result of the analysis is not definite if there are open predicates. In principle, this would force some optimizations to be delayed until the final composed semantics is known, which in general can only be done when the code for all modules is available. Therefore, although analysis can be performed for each module separately, optimizations (and thus, compilation) cannot in principle use the global information.

Incremental Analysis. When analyzing a module, each call to a predicate not declared in it is mapped to $\bot$. Each time analysis information is updated, it is applied directly to the parts of the analysis where this information may be relevant. Incremental analysis [HPMS95] is conservative: it is correct and optimal. By optimal we mean that if we put together in a single module the code for all modules (with the necessary renaming to avoid name clashes) and analyze it in the traditional way, we will obtain the same information. However, incremental analysis, in a very similar way to the previous solution, is only useful for optimization if the code for all modules is available, since the information obtained for one isolated module is only partial. On the other hand, if optimization is also made incremental, then this does present a solution to the general problem: modules are optimized as much as possible assuming no knowledge of the other modules. Optimizations will be correct with respect to the partial information available at that time. Upon module composition incremental reanalysis and reoptimization will make the composed optimized program always correct.

Note that Prolog compilers are incremental in the sense that at any point in time new clauses can be compiled into the program. Incremental analysis (aided

by incremental optimization) allows the combination of full interactive program development with full global analysis based optimization.

Trust-Enhanced Module Interface. In a module system where imported predicates have to be declared in the module importing them, such a module can only be compiled if all the module interfaces for the predicates it imports are defined, even if the actual code is not yet available. Note that the same happens for most languages with modules (e.g., Modula). When such languages have some kind of global analysis (e.g., type checking) the module interface also includes suitable declarations. We propose to augment the module interface definition so that it may include `trust` annotations for the exported predicates. Each call to a predicate not defined in the module being analyzed but exported by some module interface is in principle mapped to appropriate topmost substitutions. But if in the module interface there are one or more `trust` annotations applicable to the call pattern, such annotations will be used instead. Any call to a predicate not defined in that module and not present in any of the module interfaces can be safely mapped to $\bot$ during analysis (this corresponds to mapping program errors to failure – note that error can also be treated alternatively as a first class element in the analysis). The advantages are that we do not need the code for other modules and also that we can perform optimizations using the (inaccurate) analysis information obtained in this way.

Analysis using the trust-enhanced interface is correct, but it may be suboptimal. This can only be avoided if the programmer provides the most accurate `trust` annotations. The disadvantage of this method is that it requires the trust-enhanced interface for each module. However, the process of generating these `trust` annotations can be automated. Whenever the module is analyzed, the call/success patterns for each exported predicate in the module which are obtained by the analysis are written out in the module interface as `trust` annotations. From there, they will be seen by other modules during their analysis and will improve their exported information. A global fixpoint can be reached in a distributed way even if different modules are being developed by different programmers at different times and running the analysis only locally, provided that,

as required by the module system, the module interfaces (but not necessarily the code) are always made visible to other modules.

In practice it may be useful to use a combination of incremental analysis and the trust-enhanced module interface. The trust-enhanced interface can be used during the development phase to compile modules independently. Once the actual code for all modules is available, incremental analysis can be used to analyze modules by loading them one after another. In this way we obtain the best accuracy.

Multifile predicates (those defined over more than one file or module) also need to be treated in a special way. They can be easily identified due to the `multifile` declaration. They are similar to `dynamic` predicates (and also imported predicates) in that if we analyze a file independently of others, part of the code of a predicate is missing. We can treat such predicates as dynamic predicates and assume topmost substitutions as their abstract success substitutions unless there is a `trust` annotation for them. When the whole program composed of several files is compiled, we can again use incremental analysis. At that point, clauses for predicates are added to the analysis using *incremental addition* [HPMS95] (regardless of whether these clauses belong to different files and/or modules).

A case also worth discussing is that of libraries. Usually utility libraries provide predicates with an intended use. The automatic generation of `trust` annotations after analysis can be used for each library to provide information regarding the exported predicates. This is done for different uses and the generated annotations are stored in the library interface. With this scheme it is not necessary to analyze a library predicate when it is used in different programs. Instead, it is only analyzed once, and the information stored in the `trust` annotation is used from then on. If new uses of the library predicates arise for a given program, the library code can be reanalyzed and recompiled, keeping track of these new uses for future compilations. An alternative approach is to perform a goal independent analysis of the library, coupled with a goal dependent analysis for the particular call patterns used thereafter [CdlBBH94]. Following these ideas, several algorithms for modular analysis and optimization have been proposed recently [PH00, BdlBH$^+$01, PCH$^+$04] and implemented in the context of the Ciao

and HAL [GDMS02] systems. Chapter 3 discusses the issue of modularity further.

2.7 Chapter Conclusions

We have studied several ways in which optimizations based on static analysis can be guaranteed correct for programs which use the full power of Prolog, including modules. We have also introduced several types of program annotations that can be used to both increase the accuracy and efficiency of the analysis and to express its results. The proposed techniques offer different trade-offs between accuracy, analysis cost, and user involvement. We argue that the presented combination of known and novel techniques offers a comprehensive solution for the correct analysis and optimization of arbitrary programs using the full power of the language.

Chapter 3

A New Module System for Prolog

It is now widely accepted that separating programs into modules has proven very useful in program development and maintenance. While many Prolog implementations include useful module systems, we feel that these systems can be improved in a number of ways, such as, for example, being more amenable to effective global analysis and allowing separate compilation or sensible creation of standalone executables. We discuss a number of issues related to the design of such an improved module system for Prolog. Based on this, we present the choices made in the Ciao module system, which has been designed to meet a number of objectives: allowing separate compilation, extensibility in features and in syntax, amenability to modular global analysis, etc.

3.1 Introduction

Modularity is a basic notion in modern computer languages. Modules allow dividing programs into several parts, which have their own independent name spaces and a clear interface with the rest of the program. Experience has shown that there are at least two important advantages to such program modularization. The first one is that being able to look at parts of a program in a more or less

isolated way allows a divide-and-conquer approach to program development and maintenance. For example, it allows a programmer to develop or update a module at a time or several programmers to work on different modules in parallel. The second advantage is in efficiency: tools which process programs can be more efficient if they can work on a single module at a time. For example, after a change to a program module the compiler needs to recompile only that module (and perhaps a few related modules). Another example is a program verifier which is applied to one module at a time and does its job assuming some properties of other modules.[1]

The topic of modules and logic programming has received considerable attention (see, for example, [O'K85, Che87, WC87, GM86, Mil89, MP89, BLM94]). By the time we were designing the module system for Ciao, many popular Prolog systems such as Quintus [Qui86] and SICStus [CW94] included module systems which have proved quite useful in practice.[2] However, these practical module systems also have a series of shortcomings, specially with respect to supporting effectively separate program compilation, debugging, and optimization.

Our objective is to discuss from a practical point of view a number of issues related to the design of an improved module system for Prolog and, based on this, to present the choices made in the module system of Ciao Prolog [BCC+97]. Ciao Prolog is a next-generation logic programming system which, among other features, has been designed with modular incremental compilation, global analysis, debugging, and specialization in mind. The module system has been designed to stay as similar as possible to the module systems of the most popular Prolog implementations, but with a number of crucial changes that achieve the previously mentioned design objectives. We believe that it would not be difficult to incorporate these changes in the ISO-Prolog module standard or in other module systems.

The rest of the chapter proceeds as follows: Section 3.2 discusses the objec-

[1] Modularity is also one of the fundamental principles behind object-oriented programming.

[2] Surprisingly, though, it is also true that a number of Prolog systems do not have any module system at all.

tives of the desired module system and Section 3.3 discusses some of the issues involved in meeting these objectives. Section 3.4 then describes in detail the Ciao Prolog module system. Within this section, Subsection 3.4.5 discusses some enhancements to standard Prolog syntax extension facilities. Finally, Section 3.5 introduces the notion of packages, a flexible mechanism for implementing modular language extensions, which emerges naturally from the module system design. An example of a package is provided which illustrates some of the advantages of our module design.

3.2 Objectives in the Design of the Ciao Module System

We start by stating the main objectives that we have had in mind during the design of the Ciao module system:

- *Allowing modular (separate) and efficient compilation.* This means that it should be possible to compile (or, in general, process) a module without having to compile the code of the related modules. This allows for example having pre-compiled (pre-processed, in general) system or user-defined libraries. It also facilitates the incremental and parallel development of large software projects.

- *Local extensibility, in features and in syntax.* This means that it should be possible to define syntactic and semantic extensions of the language in a local way, i.e., so that they affect only modules using those extensions. This is very important in the context of Ciao, since one of its objectives is to serve as a workbench for new extensions to logic programming.

- *Amenability to modular global analysis.* We foresee a much larger role for global analysis of logic programs, not only in the more traditional application of optimization [WHD88, VD92, Tay91, BdlBH99], but also in new applications related to program development, such as automated debugging, validation, and program transformation [BCHP96, CLMV96,

BDD+97, HC97, HPB99]. This is specially important in Ciao because the program development environment already includes a global analysis tool (`ciaopp`, the Ciao preprocessor [HPB99, HBPLG99, HPBLG03]) which performs these tasks and which in our experience to date has shown to be an invaluable help in program development and maintenance.

- *Amenability to error detection.* This means that it should be possible to check statically the interfaces between the modules and detect errors such as undefined predicates, incompatible arities and types, etc.

- *Support for meta-programming and higher-order.* This means that it should be possible to do meta- and higher-order programming across modules without too much burden on the programmer. Also, in combination with the previous point, it should be possible to detect errors (such as calls to undefined predicates) on sufficiently determined meta-calls.

- *Compatibility with official and de-facto standards.* To the extent possible (i.e., without giving up other major objectives to fulfill this one) the module system should be compatible with those of popular Prolog systems (e.g., Quintus/SICStus) and official standards, such as the core ISO-Prolog standard [PRO94, DEDC96] and the ISO-Prolog module standard [PRO00]. This is because it is also a design objective of Ciao that it be (thanks to a particular set of libraries which is loaded by default) a standard Prolog system. This is in contrast to systems like Mercury [SHC96] or Goedel [HL94] which are more radical departures from Prolog. This means that the module system will be (at least by default) *predicate-based* rather than *atom-based* (as in XSB [SSW93] and BIM-Prolog [BIM89]), i.e., it will provide separation of predicate symbols, but not of atom names.[3] Also, the module system should not force the language to become strongly typed, since traditional

[3]However, we recognize the virtues of an atom-based system, and have made provisions in the design for separation of atom names. Also, we would applaud a switch to an atom-based module system if agreed by the community.

Prologs are untyped.[4]

3.3 Discussion of the Main Issues Involved

None of the module systems used by current Prolog implementations fulfill all of the above stated objectives, and some include characteristics which are in clear opposition to such objectives. Thus, we set out to develop an improved design. We start by discussing a number of desirable characteristics of the module system in order to fulfill our objectives. Amenability to global analysis and being able to deal with the core ISO-Prolog standard features were discussed at length in Chapter 2, where many novel solutions to the problems involved were proposed. However, the emphasis there was not on modular analysis. Herein, we will choose from some of the solutions proposed in that chapter and provide further solutions for the issues that are more specific to modular analysis and to separate compilation.[5]

- *Syntax, flags, etc. should be local to modules.* The syntax or mode of compilation of a module should not be modified by unrelated modules, since otherwise separate compilation and modular analysis would be impossible. Also, it should be possible to use different syntactic extensions (such as operator declarations or term expansions) in different modules without them interacting. I.e., it should be possible to use the same operator in different modules with different precedences and meanings. In most current module systems for Prolog this does not hold because syntactic extensions and compilation parameters (e.g., Prolog flags) are global. As a result, a module can be compiled in radically different ways depending on the operators, expansions, Prolog flags, etc. set by previously loaded modules or simply typed at the top level. Also, using a syntactic extension in a module prevents

[4]Note however, that this does not prevent having voluntary type declarations or more general assertions, as is indeed done in Ciao [PBH97, PBH00b].

[5]We concentrate here on the design on the module system. The issue of how this module system is applied to modular analysis is addressed in more detail in [PH00, BdlBH$^+$01, PCH$^+$04].

the use of, e.g., the involved operators in other modules in a different way, making the development of optional language extensions very complicated. In conclusion, we feel that directives such as `op/3` and `set_prolog_flag/2` must be local to a module.

- *The entry points of a module should be statically defined.* Thus, the only external calls allowed from other modules should be to exported predicates. Note that modules contain code which is usually related in some way to that of other modules. A good design for a modular program should produce a set of modules such that each module can be understood independently of the rest of the program and such that the communication (dependencies) among the different modules is as reduced as possible. By a *strict* module system we refer to one in which a module can only communicate with other modules via its *interface* (this interface usually contains data such as the names of the *exported* predicates). Other modules can only use predicates which are among the ones exported by the considered module. Therefore, predicates which are not exported are not visible outside the module. Many current module systems for Prolog are not strict and allow calling a procedure of a module even if it is not exported by the module. This clearly defeats the purpose of the module system and, in addition, has a catastrophic impact on the precision of global analysis, precluding many program optimizations. Thus, we feel that the module system should be strict.

- *Module qualification is for disambiguating predicate names, not for changing naming context.* In the first case, a literal `m:p` ("call `p` in module `m`") is used to refer to the predicate `p` imported from module `m`, when `p` is defined locally as well. Thus, the compiler only needs to check the exports of module `m`. But when module qualification changes naming context, a literal `m:p` is interpreted as if a literal `p` appears in module `m`, and since module `m` can import predicate `p` from another module, and that module from another, and so forth, the interfaces of all those modules would have to be followed. This precludes separate compilation (processing) since to compile (process)

a module it may be necessary to know the imports/exports of all other modules. Furthermore, in some situations changing naming context could invalidate the strictness of the module system.

- *Module text should not be in unavailable or unrelated parts.* This means that all parts of a module should be within the module itself or directly accessible at the time of compilation, i.e., the compiler must be able to automatically and independently access the complete source of the module being processed. Note that this is not the case with the classical *`user` files* used in non-modular Prolog systems, since code used by a `user` file may be in a different `user` file which has no explicit relation with the first one: there is no usage declaration that allows relating them.

- *Dynamic parts should be isolated as much as possible.* Dynamic code modification, such as arbitrary runtime clause addition (by the use of `assert`-like predicates), while very useful in some applications, has the disadvantage that it adds new entry points to predicates which are not "visible" at compile-time and are thus very detrimental to global analysis (as we have seen in Chapter 2). Our approach is to relegate such predicates to a library module, which has to be loaded explicitly.[6] In that way, only the modules using those functionalities have to be specially handled, and the fact that such predicates are used can be determined statically. Also, in our experience, dynamic predicates are very often used only to implement "global variables", and for this purpose a facility for adding facts to the program suffices. This simpler functionality, provided that this kind of dynamic predicates are declared as such explicitly in the source, poses no big problems to modular global analysis. To this end, Ciao provides a set of builtins for adding and deleting facts to a special class of dynamic predicates, called "data predicates" (`asserta_fact/1`, `retract_fact/1`, etc),

[6]Note, however, that in Ciao, to preserve compatibility for older programs, a special case is implemented: if no library modules are explicitly loaded, then all the modules containing the ISO predicates are loaded by default.

which are declared as "`:- data ...`" (similar kinds of dynamic predicates are mentioned in [Deb89a]). Furthermore, the implementation of such data predicates is typically much more efficient than that of the normal dynamic predicates, due to their restricted nature.

- *Most "built-ins" should be in libraries which can be loaded and/or unloaded from the context of a given module.* This is a requirement related to extensibility and also to more specific needs such as those mentioned in the previous point, where it was argued that program modification "built-ins" should be relegated to a library. The idea is to have a core language with very few predefined predicates (if any) and which should be a (hopefully pure) subset of ISO-Prolog. This makes it possible to develop alternative languages defining, for example, alternative I/O predicates, and to use them in a given module while others perhaps use full ISO-Prolog. It also makes it easier to produce small executables.

- *Directives should not be queries.* Traditionally, directives (clauses starting with "`:-`") were executed by the Prolog interpreter as queries. While this makes some sense in an interpretative environment, where program compilation, load (linking), and startup are simultaneous, it does not in other environments (and, specially, in the context of separate compilation) in which program compilation, linking, and startup occur at separate times. For example, some of the directives used traditionally are meant as instructions for the compiler while, e.g., others are used as initialization goals. Fortunately, this is well clarified in the current ISO standard [PRO94, DEDC96], where declarations are clearly separated from initialization goals.

- *Meta-predicates should be declared, at least if they are exported, and the declaration must reflect the type of meta-information handled in each argument.* This is needed in order to be able to perform a reasonable amount of error checking for meta-predicates and also to be able to statically resolve meta-calls across modules in most cases.

3.4 The Ciao Module System

Given the premises of previous sections, we now proceed to present their concretization in the Ciao module system.

3.4.1 General Issues

Defining Modules: The source of a Ciao module is typically contained in a single file, whose name must coincide with the name of the module, except that it may have an optional extension (`.pl`, etc). Nevertheless, the system allows inclusion of source from another file at a precise point in the module, by using the ISO-Prolog [PRO94, DEDC96] `:- include` declaration. Such included files must be present at the time of processing the module and can for all purposes be considered as an integral part of the module text. The fact that the file contains a module (as opposed to, e.g., being a `user` file –see below) is flagged by the presence of a "`:- module(...`" declaration at the beginning of the file.

For the reasons mentioned in Section 3.2, and in order to follow most logic programming system implementations, the Ciao module system is currently predicate-based (but only by default, see below). This means that non-exported predicate names are local to a module, but all functor and atom names in data are shared. We have found that this choice does provide the needed capabilities most of the time, without imposing too much burden on the user or on the implementation. The advantage of this, other than compatibility, and probably the reason why this option has been chosen traditionally, is that it is more concise for typical Prolog programs in which many atoms and functors are shared (and would thus have to be exported in an atom-based system). On the other hand, it forces having to deal specially with meta-programming, since in that case functors can become predicate names and vice-versa. It can also complicate having truly abstract data types in modules. The meta-predicate problem is solved in Ciao through suitable declarations (see Section 3.4.4) or by using true higher-order (see Chapter 4). Also, in order to allow defining truly abstract data types in Ciao it is possible to *hide* atom names, i.e., make them local to a module, by

means of "`:- hide ...`" declarations. Thus, in contrast to predicate names, which are local unless explicitly exported, functor and atom names are exported by default unless a `:- hide` declaration is used.[7]

Imports, Exports, and Reexports: A number of predicates in the module can be *exported*, i.e., made available outside the module, via explicit `:- export` declarations or in an export list in the `:- module(...` declaration. It is also possible to state that *all* predicates in the module are exported (by using '_').

It is possible to *import* a number of individual predicates or also all predicates from another module, by using `:- use_module` and `:- import` declarations (the second one to be used when the imported module is going to be dynamically loaded). In any case it is only possible to import from a module predicates that it exports. It is possible to import a predicate which has the same name/arity as a local predicate. It is also possible to import several predicates with the same name from different modules. This applies also to predicates belonging to *implicitly-imported modules*, which play the role of the built-ins in other logic programming systems. In Ciao there are really no "built-ins": all system predicates are (at least conceptually) defined in libraries which have to be loaded for these predicates to be accessible to the module. However, for compatibility with ISO, a set of these libraries implementing the standard set of ISO builtins is loaded by default.

A module `m1` can *reexport* another module, `m2`, via a `:- reexport` declaration. The effect of this is that `m1` exports all predicates of `m2` as if they had been defined in `m1` in the same way as they are defined in `m2`. This allows implementing modules which *extend* other modules (or, in object-oriented terms, classes which inherit from other classes [PH99a]). It is also possible to reexport only some selected predicates of another module, by providing an explicit list in the `:- reexport` declaration.

In Ciao it is possible to mark certain predicates as being *properties*. Examples of properties are *regular types*, *instantiation properties* (such as `var`, `indep`, or

[7]This feature of being able to hide functor and atom names is not yet implemented in the currently distributed version of Ciao.

`ground`), *computational properties* (such as `det` or `fails`), etc. Such properties, since they are actually predicates, can be exported or imported using the same rules as any other predicate. Imported properties can be used in assertions (declarations stating certain characteristics of the program, such as, e.g., preconditions and postconditions) in the same way as locally defined ones. This allows defining, e.g., the abstract data types mentioned above. This is discussed in more detail in the descriptions of the Ciao assertion language [BCC+97, PBH97, PBH00b] and the Ciao preprocessor [HPB99, HBPLG99, HPBLG03, BLGPH04].

Visibility Rules: Regarding *visibility*, the set of predicates which are visible in a module are the predicates defined in that module plus the predicates imported from other modules. It is possible to refer to predicates with or without *module qualification*. A module-qualified predicate name has the form *module*:*predicate* as in the call `lists:append(A,B,C)`. We call the *default module* for a given predicate name the module which contains the definition of the predicate which will be called when using the predicate name without module qualification, i.e., when calling `append(A,B,C)` instead of `lists:append(A,B,C)`. Module qualification makes it possible to refer to a predicate from a module which is not the default for that predicate name.

We now state the rules used to determine the default module of a given predicate name. If the predicate is defined in the module in which the call occurs, then this module is the default module. I.e., local definitions have priority over imported definitions. Otherwise, the default module is the *last module* from which the predicate is imported in the module text. Also, predicates which are explicitly imported (i.e. listed in the importation list of a `:- use_module`) have priority over those which are imported implicitly (i.e. imported when importing all predicates of a module). As implicitly-imported modules are considered to be imported first, the system allows the redefinition of "builtins". By combining implicit and explicit calls it is also possible not only to redefine builtins, but also to *extend* them, a feature used in the implementation of many Ciao libraries. It is not possible to access predicates which are not imported from a module, even if module qualification is used and even if the module exports them. It is also

not possible to define clauses of predicates belonging to other modules. This is only allowed if the predicate is defined as dynamic and exported by the module in which it is defined (but see later multifile predicates).

Reexported predicates can also be used in the module reexporting them, i.e., they are also imported into that module. Furthermore, predicates reexported from a module can also be locally defined, in which case following the normal rules the local definition takes priority in the module, although the predicate definition actually exported is still the one from the reexport. But if a reexported predicate is defined locally and also exported, then the predicate seen by other modules is the local definition and not the reexport. This flexible approach allows for example making specialized modules which are the same as a reexported module but with some of the predicates redefined as determined by local predicate definitions. For example, the following module would be an instance of module `orig` but with predicate `pred/2` redefined (the redefinition can use the original definition in `orig`):

```
:- module(redef, [p/2]).

:- reexport(orig).

p(X,Y) :-
    ...
    orig:p(X1,Y1),
    ...
```

3.4.2 `User` Files and Multifile Predicates

For reasons mainly related to backwards compatibility with non-modular Prolog systems, there are some deviations from the visibility rules above which are common to other modular logic programming systems [Qui86, CW94]: the "`user`" module and *multifile* predicates.

`User` Files: To provide backwards compatibility with non-modular code, all code belonging to files which have no module declaration is assumed to belong

to a single special module called "`user`". These files are called "`user` files", as opposed to calling them modules (or packages –see later). All predicates in the `user` module are "exported". It is possible to make unrestricted calls from any predicate defined in a `user` file to any other predicate defined in another `user` file. However, and differently to other Prolog systems, predicates imported from a normal module into a `user` file are not visible within the other `user` files unless they are explicitly imported there as well. This at least allows performing separate static compilation of each `user` file, as all static predicate calls in a file are defined by just reading that file. Predicates defined in `user` files can be visible in regular modules, but such modules must explicitly import the "`user`" module, stating explicitly which predicates are imported from it.

The use of `user` files is discouraged because, apart from losing the separation of predicate names, their structure makes it impossible to detect many errors that the compiler detects in modules by looking at the module itself (and perhaps the interfaces of related modules). As an example, consider detecting undefined predicates: this is not possible in `user` files because a missing predicate in a `user` file may be defined in another `user` file and used without explicitly importing it. Thus, it is only possible to detect a missing predicate by examining all `user` files of a project, which is itself typically an unknown (and, in fact, not even in this way, since that predicate could even be meant to be typed in at the top level after loading the `user` files!). Also, global analysis of `user` files typically involves considerable loss of precision because all predicates are possible entry points (see Chapter 2). Note that it is often just as easy and flexible to use in place of `user` files modules which export all predicates (a feature available in Ciao replacing the export list by a variable), while being able to retain many of the advantages of modules.

Multifile Predicates: Multifile predicates are a useful feature (also defined in ISO-Prolog) which allows a predicate to be defined by clauses belonging to different files (modules in the case of Ciao). To make this fit in with the module system, in Ciao these predicates are implemented as if belonging to the special module "`multifile`". However, calls present in a clause of a multifile predicate

are always to visible predicates of the module where that clause resides. As a result, multifile predicates do not pose special problems to the global analyzer (which considers them alike to exported predicates) nor to code processing in general.

3.4.3 Dynamic Modules

The module system described so far is quite flexible but it is *static*, i.e., except in `user` files, it is possible to determine statically the set of imports and exports of a given module and the set of related modules, and it is possible to statically resolve to which module each call in the program refers to. This has many advantages: modular programs can be implemented with no run-time overhead with respect to a non-modular system and it is also possible to perform extensive static analysis for optimization and error detection. However, in practice it is sometimes very useful to be able to load code dynamically and call it. In Ciao this is fully supported, but only if the appropriate compiler library (defining, among other, `use_module/1`) is explicitly imported (this is done in the default package, see later). This importation can then be seen by compile-time tools which can act more conservatively as needed (allowing the adverse effects be limited to the modules which import the compiler). Also, when making standalone executables the compiler needs to be included only if dynamic use of modules is used in the application. The compiler library and executable creation are explained in detail in Chapter 5.

3.4.4 Dealing with Meta-Calls

As mentioned before, the fact that the module system is in this design predicate-based by default forces having to deal specially with meta-programming, since in that case functors can become predicate names and vice-versa.[8] This problem is solved in the Ciao system, as in similar systems [Qui86, CW94], through

[8]This problem can be elegantly avoided in Ciao by using higher-order, as explained in Chapter 4.

`meta_predicate` declarations which specify which arguments of predicates contain meta-data. However, because of the richer set of meta-predicate facilities and predicate types provided by Ciao, and to the benefit of the compile-time tools, there is a correspondingly richer set of types of meta-data (which allows among other things more error detection):

`goal`:
: denotes a goal (either a simple or a complex one) which will be called. The paradigmatic example is `:- meta_predicate(call(goal)).`

`clause`:
: denotes a clause, of a dynamic predicate, which will be asserted/retracted. An example would be `:- meta_predicate(asserta(clause)).`

`fact`:
: denotes a fact (a head-only clause), of a `dynamic` or `data` predicate. An example would be `:- meta_predicate(retractall(fact)).`

`spec`:
: denotes a predicate name, given as a term *Functor/Arity* (this kind of meta-term is sometimes used in builtin predicates, but seldom in user-defined predicates). An example would be `:- meta_predicate(abolish(spec)).`

`pred(`*N*`)`:
: denotes a predicate construct to be called by means of a `call/`*N* predicate call ($N \geq 1$). Paradigmatic examples would be the set of declarations `:- meta_predicate(call(pred(`*N*`), _, ... _))` for each `call/(`*N*`+1)` with $N \geq 1$ (the '_' arguments denote normal terms). However, note that this type of meta-predicates are surpassed by the new higher-order facilities of Ciao, explained in Chapter 4.

`addmodule`: This is not a real meta-argument type. Rather, it makes the compiler add an argument to the invocations of this predicate which contains

the module where the call is made, to allow certain kinds of low-level meta-programming. The most simple use of it is in the definition of the Ciao builtin `this_module/1`:

```
:- meta_predicate(this_module(addmodule)).
this_module(M, M).
```

The compiler, by knowing which predicates have meta-arguments, can verify if there are undetermined meta-calls (which for example affect the processing when performing global analysis), or else can determine (or approximate) the calls that these meta-arguments will produce.

3.4.5 Modular Syntax Enhancements

Traditionally (and also now in the ISO standard [PRO94, DEDC96]) Prolog systems have included the possibility of changing the syntax of the source code by the use of the `op/3` builtin/directive. Furthermore, in many Prolog systems it is also possible to define *expansions* of the source code (essentially, a very rich form of "macros") by allowing the user to define (or extend) a predicate typically called `term_expansion/2` [Qui86, CW94]. This is usually how definite clause grammars (DCG's) are implemented.

However, these features, in their original form, pose many problems for modular compilation or even for creating sensible standalone executables. First, the definitions of the operators and expansions are global, affecting a number of files. Furthermore, which files are affected cannot be determined statically, because these features are implemented as a side-effect, rather than a declaration, and they are meant to be active after they are read by the code processor (top-level, compiler, etc.) and to remain active from then on. As a result, it is impossible by looking at a source code file to know if it will be affected by expansions or definitions of operators, which may completely change what the compiler really sees. Furthermore, these definitions also affect how a compiled program will read terms (when using the term I/O predicates), which will also be affected by operators and expansions. However, in practice it is often desirable to use a set of

operators and expansions in the compilation process (which are typically related to source language enhancements) and a completely different set for reading or writing data (which can be related to data formatting or the definition of some application-specific language that the compiled program is processing). Finally, when creating executables, if the compile-time and run-time roles of expansions are not separated, then the code that defines the expansions must be included in the executable, even if it was only meant for use during compilation.

To solve these problems, in Ciao we have redesigned these features so that it is still possible to define source translations and operators but they are local to the module or `user` file defining them. Also, we have implemented these features in a way that has a well defined behavior in the context of a stand-alone compiler (the Ciao compiler, `ciaoc` – see Chapter 5). In particular, the directive `load_compilation_module/1` allows separating code that will be used at compilation time from code which will be used at run-time. It loads the module defined by its argument *into the compiler* (if it has not been already loaded). It differs from the `use_module/1` declaration in that the latter defines a use by the module being compiled, but does not load the code into the compiler itself. This distinction also holds in the Ciao interactive top-level, in which the compiler (which is the same library used by `ciaoc`) is also a separate module.

In addition, in order to make the task of writing expansions easier, the effects usually achieved through `term_expansion/2` can be obtained in Ciao by means of four different, more specialized directives, which affect only the current module. Each one defines a different target for the translations, the first being equivalent to the `term_expansion/2` predicate which is most commonly included in Prolog implementations. The argument for all of them is a predicate indicator of arity 2 or 3. When reading a file, the compiler (actually, the general purpose module processing library – see Chapter 5) invokes these translation predicates at the appropriate times, instantiating their first argument with the item to be translated (whose type varies from one kind of predicate to the other). If the predicate is of arity 3, the optional third argument is also instantiated with the name of the module where the translation is being done, which is sometimes needed during certain expansions. If the call to the expansion predicate is successful, the term

returned by the predicate in the second argument is used to replace the original. Otherwise, the original item is kept. The directives are:

`add_sentence_trans/1` : Declares a translation of the terms read by the compiler which affects the rest of the current text (module or `user` file). For each subsequent term (directive, fact, clause, ...) read by the compiler, the translation predicate is called to obtain a new term which will be used by the compiler in place of the term present in the file. An example of this kind of translation is that of DCG's.

`add_term_trans/1` : Declares a translation of the terms and sub-terms read by the compiler which affects the rest of the current text. This translation is performed after all translations defined by add_sentence_trans/1 are done. For each subsequent term read by the compiler, and recursively any subterm included in such a term, the translation predicate is called to possibly obtain a new term to replace the old one. Note that this is computationally intensive, but otherwise very useful to define translations which should affect any term read. For example, it is used to define *records* (feature terms [AKPS92]), in the Ciao standard library `argnames` (see Subsection 3.5.1).

`add_goal_trans/1` : Declares a translation of the goals present in the clauses of the current text. This translation is performed after all translations defined by `add_sentence_trans/1` and `add_term_trans/1` are done. For each clause read by the compiler, the translation predicate is called with each goal present in the clause to possibly obtain other goal to replace the original one, and the translation is subsequently applied to the resulting goal. Note that this process is aware of meta_predicate definitions (which reside in module interfaces). In the Ciao system, this kind of translation is used for example in the `functions` library, which provides functional syntax, because the translation of function calls needs to be applied to goals, since it adds new goals just before a goal containing function calls.

`add_clause_trans/1` : Declares a translation of the clauses of the current text. The translation is performed prior to `add_goal_trans/1` translations but af-

ter `add_sentence_trans/1` and `add_term_trans/1` translations. This kind of translation is defined for carrying out more involved translations and is related to the compiling procedure of Ciao. The usefulness of this translation is that information on the interface of related modules is available when it is performed, but on the other hand it must maintain the predicate defined by each clause, since the compiler has already made assumptions regarding which predicates are defined in the code. As an example, the object-oriented extension of Ciao (O'Ciao) uses this feature [PH99a, PB02].

Figure 3.1 shows, for an example clause of a program, which subterms each type of translation would be applied to, and also the order of translations. The principal functor of the head in the clause translation is dashed because the translation cannot change it.

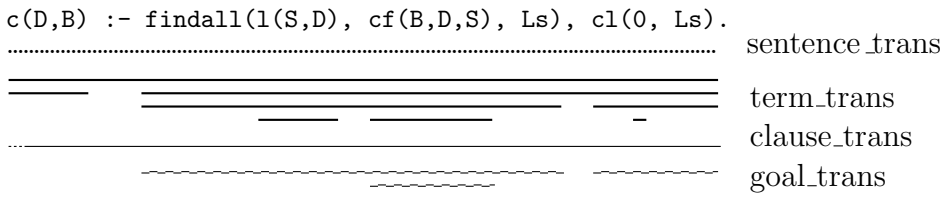


Figure 3.1: Subterms to which each translation type is applied in a clause

Finally, there is another directive in Ciao related to syntax extension, whose raison d'être is the parametric and extensible nature of the compiler framework: `new_declaration/2`. Note that in ISO-Standard Prolog declarations cannot be arbitrary Prolog goals. Thus, the Ciao compiler flags an error if a declaration is found which is not in a predefined set. However, a declaration `new_declaration(Decl, In_Itf)` can be used to state that `Decl` is a valid declaration in the rest of the current text. Such declarations are simply ignored by the compiler or top level, but can be used by other code processing programs. For example, in the Ciao system program assertions and machine-readable comments are defined in as new declarations and are processed by the `ciaopp` preprocessor and the automatic documenter [Her00]. `In_Itf` is a switch. If it is `on` this kind of declarations will be included in the module interface and thus will be visible

while processing other modules which make use of this one, when using the `c_itf` generic module processing framework (see Chapter 5 for details).

3.5 Packages

Experience using the Ciao module system shows that the local nature of syntax extensions and the distinction between compile-time and run-time work results in the libraries defining extensions to the language having a well defined and repetitive structure. These libraries typically consist of a main source file which defines only some declarations (operator declarations, declarations loading other modules into the compiler or the module using the extension, etc.). This file is meant to be *included* as part of the file using the library, since, because of their local effect, such directives must be part of the code of the module which uses the library. Thus, we will call it the "include file". Any auxiliary code needed at compile-time (e.g., translations) is put in a separate module which is to be loaded into the compiler via a `load_compilation_module` directive which is placed in the include file. Also, any auxiliary code to be used at run-time is placed in another module, and the corresponding `use_module` declaration is also placed in the include file. Note that while this run-time code could also be inserted in the include file itself, it would then be replicated in each module that uses the library. Putting it in a module allows the code to be shared by all modules using the library.

Libraries constructed in this manner are called "packages" in Ciao. The main file of such a library is a file which is to be *included* in the importing module. Many libraries in Ciao are packages: `dcg` (definite clause grammars), `functions` (functional syntax), `class` (object-oriented extension), `clpq` and `clpr` (contraints over rationals or reals), `persdb` (persistent database), `assertions` (to include assertions –see [PBH97, PBH00b]), etc. Such libraries can be loaded using a declaration such as `:- include(library(functions))`. For convenience (and other reasons related to ISO compatibility), this can also be written as `:- use_package(functions)`.

To simplify the use of Ciao Prolog to users accustomed to other Prolog sys-

tems, which may expect some predicates or features (e.g. DCG's) defined by default, there exists a package called `default` which is used by default by any source code, except when its load is inhibited. This package imports from the Ciao libraries predicates commonly available on other popular Prolog systems, and also makes use of other packages which define features usually found in those systems. The loading of this default package is inhibited in modules starting with the Ciao-specific `module/3` declaration (instead of the more common `module/2` declaration) or in `user` files starting with a (also Ciao-specific) `use_package/1` declaration. The `module/3` declaration includes an additional argument which is a lists of packages to load. Thus, `:- module(`*`M,Exports`*`)` is indeed equivalent to `:- module(`*`M,Exports`*`,[default])`, and `:- module(`*`M,Exports`*`,[])` defines a module where no packages are requested.

There is another feature which allows defining modules which do not start with a `:- module` declaration, and which is useful when defining language extensions: when the first declaration of a file is unknown, the declared library paths are browsed to find a package with the same name as the declaration, and if it is found the declaration is treated as a `module/3` declaration where its third argument contains that package. For example, the package which implements the object-oriented capabilities in Ciao is called "`class`": this way, one can start a class (a special module in Ciao) with the declaration "`:- class(myclass)`", which is then equivalent to defining a module which loads the `class` package (the exported predicates will need to be defined by `:- export` declarations). The `class` package then defines translations which transform the module code so that it can be used as a class, rather than as a simple module.

3.5.1 An Example Package: `argnames`

To clarify some of the concepts introduced in this chapter, we will describe as an example the implementation of the Ciao library package "`argnames`".[9] This library implements a syntax to access term arguments by name (also known

[9]This package uses only part of the functionality described. See the Ciao system libraries for many examples of Ciao packages.

as *records*). For example, Figure 3.2 shows a fragment of the famous "zebra" puzzle written using the package. The declaration `:- argnames` (where `argnames` is defined as an operator with suitable priority) assigns a name to each of the arguments of the functor `house/5`. From then on, it is possible to write a term with this functor by writing its name (`house`), then the infix operator `'$'`, and then, between brackets (which are as in ISO-Prolog), the arguments one wants to specify, using the infix operator `'=>'` between the name and the value. For example, `house${}` is equivalent in that code to `house(_,_,_,_,_)` and `house${nation=>Owns_zebra,pet=>zebra}` to `house(_,Owns_zebra,zebra,_,_)`.

```
:- use_package([argnames]).

:- argnames house(color, nation, pet, drink, car).

zebra(Owns_zebra, Drinks_water, Street) :-
    Street = [house${},house${},house${},house${},house${}],
    member(house${nation=>Owns_zebra,pet=>zebra}, Street),
    member(house${nation=>Drinks_water,drink=>water}, Street),
    member(house${drink=>coffee,color=>green}, Street),
    left_right(house${color=>ivory}, house${color=>green}, Street),
    member(house${car=>porsche,pet=>snails}, Street),
    ...
```

Figure 3.2: "zebra" program using `argnames`

The library which implements this feature is composed of two files, one which is the package itself, called `argnames`, and an auxiliary module which implements the code translations required, called `argnames_trans` (in this case no run-time code is necessary). They are shown in Figures 3.3 and 3.4 (this last one has been simplified for brevity by deleting error checking code).

The contents of package `argnames` are self-explanatory: first, it directs the compiler to load the module `argnames_trans` (if not already done before), which

```
:- load_compilation_module(library(argnames_trans)).
:- add_sentence_trans(argnames_def/3).
:- add_term_trans(argnames_use/3).
:- op(150, xfx, [$]).
:- op(950, xfx, (=>)).
:- op(1150, fx, [argnames]).
```

Figure 3.3: The package `argnames`.

contains the code to make the required translations. Then, it declares a sentence translation, which will handle the argnames declarations, and a term translation, which will translate any terms written using the argnames syntax. Finally, it declares the operators used in the syntax. Recall that a module using this package is in fact including these declarations into its code, so the declarations are local to the module and will not affect the compilation of other modules.

The auxiliary module `argnames_trans` is also quite straightforward: it exports the two predicates which will be used by the compiler to do the translations. Then it declares a data predicate (recall that a data predicate is a simplified dynamic predicate) which will store the declarations made in each module. Predicate `argnames_def/3` is simple: if the clause term is an `argnames` declaration, it translates it to empty code, but stores its data in the above mentioned data predicate. Note that the third argument is instantiated by the compiler to the module where the translation is being made, and thus is used so that the declarations of a module are not mingled with the declarations in other modules. The second clause is executed when the end of the module is reached. It takes care of deleting the data pertaining to the current module. Then, predicate `argnames_use/3` is in charge of making the translation of argname'd-terms, using the data collected by the other predicate. Although more involved, it is a simple Prolog exercise.

Note that the argnames library only affects the modules which load it. Thus, the operators involved (`argnames`, `$`, `=>`) can be used in other modules or libraries for different purposes. This would be very difficult to do with the traditional model.

```
:- module(argnames_trans, [argnames_def/3, argnames_use/3]).

:- data argnames/4.

argnames_def((:- argnames(R)), [], M) :-
        functor(R, F, N),
        assertz_fact(argnames(F,N,R,M)).
argnames_def(end_of_file, end_of_file, M) :-
        retractall_fact(argnames(_,_,_,M)).

argnames_use($(F,TheArgs), T, M) :-
        atom(F),
        argnames_args(TheArgs, Args),
        argnames_trans(F, Args, M, T).

argnames_args({}, []).
argnames_args({Args}, Args).

argnames_trans(F, Args, M, T) :-
        argnames(F, A, R, M),
        functor(T, F, A),
        insert_args(Args, R, A, T).

insert_args([], _, _, _).
insert_args('=>'(F,A), R, N, T) :-
        insert_arg(N, F, A, R, T).
insert_args(('=>'(F,A), As), R, N, T) :-
        insert_arg(N, F, A, R, T),
        insert_args(As, R, N, T).

insert_arg(N, F, A, R, T) :-
        N > 0,
        (   arg(N, R, F) ->
               arg(N, T, A)
        ;   N1 is N-1,
            insert_arg(N1, F, A, R, T)
        ).
```

Figure 3.4: The translation module `argnames_trans`.

3.6 Chapter Conclusions

We have presented a new module system for Prolog which achieves a number of fundamental design objectives such as being more amenable to effective global analysis, allowing separate compilation and sensible creation of standalone executables, extensibility in features and in syntax, etc. It has been also shown in other work that this module system can be implemented easily and can be applied successfully in several modular program processing tasks, from compilation to debugging to automatic documentation generation [CH99b, PH99c, HPB99, Her99, Her00] (see also Chapter 5). The proposed module system has been designed to stay as similar as possible to the module systems of the most popular Prolog implementations, but with a number of crucial changes that achieve the previously mentioned design objectives. We believe that it would not be difficult to incorporate these changes in the ISO-Prolog module standard or in the module systems of other Prolog systems.

Chapter 4

A Higher-Order Logic Programming Model for Ciao

4.1 Introduction

This chapter reports on the most recent design and implementation of higher-order in Ciao. We do not address here theoretical discussions about higher-order in logic programming in general — the interested reader is referred to the relevant bibliography [War82, Pfe88, NP92, Nai96, CKW93]. We propose an approach to add higher-order to Prolog which departs from previous approaches, and specifically from those based on Church's simple theory of types [Chu40]. Our aim was to define higher-order in the context of the standard Prolog language, which is untyped, and with the premise of not imposing a burden on the execution of first-order code. Furthermore, we wanted to improve the performance of other approaches based on term manipulation with regard to sensible global analysis of the program.

4.2 Higher-Order Vs. Meta-Programming

Here we are going to distinguish between two related but different concepts: higher-order programming and meta-programming. Meta-programming is the

manipulation of data representing code with the purpose of later executing it by its lifting to an executable status. It is traditionally accomplished in Prolog with standard term manipulation and the builtin `call/1`. For us the notion of higher-order programming implies having data, different from ordinary data, which represent processing units (in logic programming the natural choice will be predicates), and the ability of calling them instantiating its arguments. Ciao incorporates higher-order programming because the use of meta-programming has several drawbacks from the software engineering perspective, regarding sensible global analysis of the program and module separation, drawbacks not present in our higher-order proposal for Ciao. This good behavior is ensured by these two principles:

- Higher-order data is syntactically differentiated from ordinary data. In this way any name of the program can be easily told appart as a predicate name or a data functor name. Additionally, a predicate name occurs always with its proper arity. We feel these properties are crucial for a predicate-based module system, as is the current module system of Ciao.

- Higher-order data is like an "abstract data type": it is regarded as a "black box" which cannot be inspected or manipulated, except by specific operations defined on it. This other property makes higher-order more amenable to effective global analysis.

Furthermore, as it happens with all extensions in Ciao, higher-order will only be available in modules using the `hiord` package.[1]

4.3 Syntax of Higher-Order Data

Our proposal is to syntactically differentiate higher-order data from ordinary terms. Thus, in modules using the `hiord` package,[2] all terms surrounded by `{}`

[1]The higher-order implementation explained here is an evolution of the one present in the last officially distributed version of Ciao (v 1.10).

[2]It is a package and not a mere module because it adds new syntax – see Chapter 3.

will be considered higher-order data. As an example, the following goal tests whether all the elements of a list are greater than a certain number:

```
list(L,{# > N}).
```

Two different syntactic forms for writing higher-order data are provided: *predicate abstractions* and *closures*. Closures are in fact "syntactic sugar" for predicate abstractions, and can always be written otherwise. Predicate abstractions are our translation to logic programming of the lambda expressions of functional programming: they define unnamed predicates which will be ultimately executed by a higher-order call, unifying its arguments appropriately. The most general syntax for predicate abstractions is

```
{sharedvars -> ''(parameters) :- body}
```

where ***sharedvars*** is a sequence of distinct variables and ***parameters*** is the sequence of parameters. When ***sharedvars*** is void the arrow is not written, and when ***parameters*** is void no surrounding parenthesis are written. Also, when ***body*** is `true` the text "`:- true`" can be omitted. Note that the functor name in the head is the void atom `''`. Some examples are:

```
all_less(L1, L2) :- map(L1, {''(X,Y) :- X < Y}, L2).

same_mother(L) :- list(L, {M -> ''(S) :- child_of(S,M,_)}).

same_parents(L) :- list(L, {M,F -> ''(S) :- child_of(S,M,F)}).
```

Variables in ***sharedvars*** are shared with the rest of the clause and in successive invocations of the predicate abstraction, so that the instantiation of those variables affects the predicate abstraction. The rest of the variables in the predicate abstraction are local to the predicate abstraction, even if their names happen to coincide with variables outside the predicate abstraction. Thus, variables not appearing in ***sharedvars*** or in ***parameters*** are considered existentially quantified.

The decision of marking shared variables instead of existential variables is based on the following considerations:

- In that way it is made clear which variables from "the outside" can affect the predicate abstraction.
- Unique existential variables in the predicate abstraction can be written as anonymous variables (`_`).
- Code expansions are simplified, since new variables introduced by expansions should be existential.
- The compilation of the predicate abstraction is simplified.

All higher-order data (surrounded by `{}`) not having predicate abstraction syntax is a closure. In a closure, all occurrences of the atom `#` mark a parameter of the predicate abstraction it represents. All variables in a closure are shared with the rest of the clause (for compatibility with meta-programming). Thus, the following definition of `same_parents/2` is equivalent to the above one:

```
same_parents(L) :- list(L, {child_of(#,_M,_F)}).
```

Parameters, represented by `#`, are ordered as they are found in the closure. For example, the following definition of `all_less/2` is equivalent to the above one:

```
all_less(L1, L2) :- map(L1, {# < #}, L2).
```

If a different order is needed, a predicate abstraction should be used.

4.4 Builtins for Handling Higher-Order Data

We have just seen how higher-order data can be defined in the code, which could be considered the first operation of the "abstract data type" we are defining: a function which creates higher-order data. We present next the other operations available for higher-order data:

`apply(P,`*`parameters`*`)`

If `P` is bound to higher-order data, this call executes `P` instantiating its formal parameters to *`parameters`*. The number of parameters must match. If `P` is unbound, the execution proceeds by constraining it to succeed when executed with the given parameters (which is equivalent to "freezing" the invocation of the higher-order call until `P` is instantiated). In any other case (`P` is bound to a normal term) the builtin raises an error. The `hiord` package allows to write it with the special syntax `P(`*`parameters`*`)`. Note that this predicate is usually named `call/N` in the literature, we do not use this name to be able to distinguish `call(P)` (which is meta-programming) and `apply(P)`.

`shared_vars(P, VList)`

Returns in `VList` the list of variables which `P` shares with the rest of the program. It can be used for example to program a higher-order version of the Prolog meta-predicate `setof`.

`arity(P, N)`

Returns in `N` the arity of the predicate abstraction `P`.

Note that, given our definition of higher-order data, there is no provisions for creating new higher-order data at runtime (all higher-order data has to be defined in the code). This is not a lapse, rather, we feel that such a feature is seldom needed, an furthermore it definitely precludes sensible global analysis of the code. On the other hand, there is already a feature in standard Prolog for adding new code at runtime: dynamic code and the assert family of buitins. In Ciao, as explained in Chapter 3, this feature is relegated to a library module which has to be loaded explicitly (except when the `default` package is loaded). In that way, only the modules using that functionality have to be specially handled by global analysis. Thus, if in a given program a predicate abstraction needs to be defined at runtime, we can use in that program the library module for dynamic code, and make the predicate abstraction to call a predicate declared as dynamic, which can be assigned code in the usual manner. An example of this would be

the following:

```
:- use_package(hiord).
:- use_module(library(dynamic)).

:- dynamic def_at_runtime/2.

main :-
        ...
        create_code(X, Y, Code),
        asserta(def_at_runtime(X,Y) :- Code),
        ...
        map(L1, {def_at_runtime(#,#)}, L2),
        ...
```

Note that by this mechanism all the power of meta-programming can be incorporated into our higher-order proposal, but of course bringing along with it all of its hindrances.

4.5 Curry and Other Spices

In our proposal there is no equivalent of the functional curryfication, as we feel this notion is not natural in traditional Logic Programming. However, this does not reduce the power of the language, as its results can be obtained by deriving higher-order data from any other higher-order data, by instantiating any parameters desired. For example, given a variable `G3` which is bound to a ternary predicate abstraction, the unification `G2 = {G3(#,0,#)}` will create a binary predicate abstraction by assigning the second argument a given value (of course this case is not even handled by curryfication). As a more involved example, we show the definition of a predicate to test a binary predicate on all pairs of the cross-product of two lists, based on the predicate `list/2`, which tests a unary predicate on the elements of a list:

```
product_test([], _, _).
product_test([X|Xs], L, P) :-
    list(L, {P(X,#)}),
    product_test(Xs, L, P).

list([], _P).
list([X|Xs], P) :-
    P(X),
    list(Xs, P).
```

Note that in the above example `{P(X,#)}` is syntactic sugar for the predicate abstraction `{P,X -> ''(E) :- apply(P,X,E)}`

4.6 Higher-Order Unification

Our approach avoids the complications of other approaches regarding higher-order unification, due to the fact that we consider predicate abstractions as a black box which cannot be inspected. In our approach higher-order data is not unifiable with ordinary terms, and two different instances of higher-order data are unifiable if and only if they represent a direct call to the same existing predicate in the program. That is, these goals fail:

```
?- {p(#,#)} = p(_,_).
no
?- {p(#,X)} = {X -> ''(Y) :- p(Y,X)}.
no
?- {p(#,1)} = {p(#,1)}.
no
```

Whereas these succeed:

```
?- {p(#,#)} = {p(#,#)}.
yes
?- {p(#,#)} = {''(X,Y) :- p(X,Y)}.
yes
?- P = {p(#,1)}, Q = P, P = Q.
yes
```

This behavior is explained because higher-order data represents in general anonymous executable data which in principle should not be comparable with other anonymous executable data. The exception is when the executable data is indeed a named computation, that is, an existing predicate all of its arguments are used (and in the same order). We call that kind of predicate abstractions "named predicate abstractions". Thus, a named predicate abstraction is unifiable with other equivalent representation of the same higher-order data. Note that a named predicate abstraction is always representable as a closure containing a predicate name with all of its arguments being `#` parameters.

4.7 Implementation Issues

A predicate abstraction as

```
{sharedvars -> ''(parameters) :- body}
```

(or its equivalent as a closure), would be translated by the `hiord` package to a special term containing an atom *newname*, an integer *arity* (the length of *parameters*) and the list of variables *sharedvars* (plus possibly other data for optimization purposes), and a new program clause

```
newname(parameters,sharedvars) :- body.
```

where *newname* would be a new predicate name in the program.

There is a special case in that translation: if the predicate abstraction (or closure) represents a named predicate abstraction then no new program clause is generated and *newname* would be the name of the predicate involved (and *sharedvars* would be empty).

Upon calling `apply`, *newname* would be called instantiating the first *arity* variables with the supplied parameters, and the rest with the shared variables in *sharedvars* (which are carried by the higher-order structure).

4.8 Modularity and Global Analysis

As we have mentioned earlier, only those modules using the `hiord` package will be able to use higher-order, and thus global analysis needs only to be aware of it in those modules.

Regarding the interaction of higher-order and modularity, and in order to respect module separation, names inside a predicate abstraction (or a closure, which is syntactic sugar for it) appearing in the code of a module are interpreted in the context of that module, and not in the context of the module where the apply builtin is ultimately invoked. This implies that the new program clause created to define the "abstract" predicate we mentioned in the previous section will belong to the module where the predicate abstraction appears. This, in addition to the fact that predicate abstractions cannot be manipulated at run-time by the program, implies that when compiling a module all calls to the predicates defined in the module are statically defined, discounting the exported predicates.

Thus, abstract interpretation-based data-flow analysis of a module using predicate abstractions (see Chapter 2) can be effectively performed as follows:

- Consider the clause implicitly defined by the compilation of each predicate abstraction as a program clause (if it is not a named predicate abstraction) – see previous section.

- Consider as exported predicates the new predicates defined by those clauses. This two points solve the extra call pattern problem.

- Translate each predicate abstraction to a term with principal functor a name not present in the program, and with arguments the name of the new predicate defined and the list of the shared variables.

- The abstract execution of an `apply` call will make topmost the substitutions for all the arguments of the apply (including the first argument, which holds the higher-order data). Note that since our translation of a predicate

abstraction includes the list of shared variables, those variables will also become topmost.

Of course better results can be obtained by taking into account more specifically higher-order, but we leave this issue as future work.

4.9 Aggregation Predicates as Higher-Order Predicates

The standard meta-predicate builtins `setof/3`, `bagof/3` and `findall/3` can be reformulated in terms of our higher-order definition, and we feel the semantics and use of the new versions is greatly improved with respect to the old ones. Only two higher-order predicates would be needed: `solutions/2` and `set_solutions/2`. `solutions(Pred, List)`, where `Pred` is a predicate abstraction (or closure) with arity 1, unifies `List`, by backtracking, with all the lists which contain the successive instantiations of the argument of the predicate abstraction which produce the same instantiation of the shared variables of the predicate abstraction. Of course if the predicate abstraction has no shared variables, then it produces as a single solution the list of the successive instantiations of the argument of the predicate abstraction. `set_solutions(Pred, List)` is similar but sorts the lists and eliminates duplicates. Below we show some correspondences between calls to the meta-predicate builtins and calls to the higher-order counterparts:

```
   bagof(X, Y^p(X,Y,Z), L)
→ solutions({Z -> ''(X) :- p(X,_,Z)}, L)

   setof(X, Y^Z^V^(q(W,Y,Z), r(Y,V,X)), L)
→ set_solutions({W -> ''(X) :- q(W,Y,_), r(Y,_,X)}, L)

   findall(-(X,Y), p(a,_,X,Y), L)
→ solutions({''(-(X,Y)) :- p(a,_,X,Y)}, L)
```

Note that in the predicate abstractions all variables are existential except when

they are explicitly marked, whereas in the meta-predicate builtins the existential variables need to be marked (with a syntactic ad-hoc kludge), but not the shared ones. This is the reason why the two predicates `bagof/3` and `findall/3` exist, because the latter is a version of the former where all free variables are taken to be existentially quantified, thus simplifying its writing in these cases (but with possibly unexpected results). We feel that the predicate abstractions, apart from having a clearer semantics, mark the relevant variables for the understanding of the effects of the call: those that will be affected by the solution of the aggregation predicate.

4.10 Chapter Conclusions

We have presented a novel definition of higher-order for Logic Programming, which allows sensible global analysis of the program, module separation, and an efficient implementation. We argue that it also offers a clear semantics and that is of enough expressive power for most of the needs of Prolog programmers in this regard.

Part II

Implementation Issues

Chapter 5

A Generic Program Processing Library and Modular Compiler

Ciao Prolog incorporates a module system which allows separate compilation and sensible creation of standalone executables. We describe some of the main aspects of the Ciao modular compiler, `ciaoc`, which takes advantage of the characteristics of the Ciao Prolog module system to automatically perform separate and incremental compilation and efficiently build small, standalone executables with competitive run-time performance. `ciaoc` can also detect statically a larger number of programming errors. We also present a generic code processing library for handling modular programs, which provides an important part of the functionality of `ciaoc`. This library allows the development of program analysis and transformation tools in a way that is to some extent orthogonal to the details of the module system design, and has been used in the implementation of `ciaoc` and other Ciao system tools. We also describe the different types of executables which can be generated by the Ciao compiler, which offer different tradeoffs between executable size, startup time, and portability, depending, among other factors, on the linking regime used (static, dynamic, lazy, etc.). Finally, we provide experimental data which illustrate these tradeoffs.

5.1 Introduction

Ciao Prolog [BCC+97] is a next-generation logic programming system which, among other features, has been designed with modular incremental compilation, global analysis, debugging, and specialization in mind. The Ciao module system, while attempting to be compatible to a large extent with official and de-facto standards (i.e., with popular Prolog implementations and the ISO-Prolog module standard [PRO00]), includes a number of crucial changes that arguably enable building a better language and program development environment. In this chapter we describe the overall architecture of the Ciao standalone compiler, `ciaoc`, which takes advantage of the characteristics of the module system to achieve a number of global design objectives, including detecting a larger number of errors statically, performing modular incremental ("separate") compilation, supporting modular extensibility of the language in features and in syntax, efficiently building small, standalone executables with different executable size tradeoffs and competitive run-time performance, offering support for meta-programming and higher-order, and allowing global analysis, debugging, and specialization/optimization.

`ciaoc` does not treat the compilation process as a translation from a single, isolated Prolog source to, e.g., its WAM bytecode. Instead, a module is compiled taking into account its relationship with the modules it uses. Also, sets of modules comprising an application, together with the set of user or system libraries it uses, are processed globally and incrementally. As a result, the corresponding object code and any other processing output is kept always updated with respect to the source while recompiling the minimal required set of dependent modules after a change. This applies also when the compiler is used in the toplevel shell, which also tracks in the same way which loaded modules have changed and need updating. A correct image of the program is always kept in the toplevel without predicate duplications or old code lingering around, as can possibly happen with traditional toplevel shells. `ciaoc` treats modularly and incrementally other information in addition to module interfaces, such as the annotated and optimized programs produced by the `ciaopp` preprocessor [HBPLG99, HPBLG03].

One of the most interesting results of the development of the compiler, from

the software architecture point of view, has been that we have been able to abstract away into a generic code processing library much of the functionality related to the modular and incremental treatment of programs, even if they are multi-file and use multiple user and system libraries. This library allows the development of program analysis and transformation tools in a way that is to some extent orthogonal to the details of the module system design, and has been used also in other Ciao system tools such as the `ciaopp` preprocessor, the automatic documenter [Her99, Her00], etc. In fact, the whole compiler is itself also a library, which can be used by any Ciao executable. This has made it very easy for example to have a standalone command-line compiler (as is usual in other programming languages) and a script interpreter, along with the more typical Prolog interactive shell, and to ensure that they all behave in exactly the same way.

The rest of the chapter proceeds as follows: Section 5.2 presents the generic code processing library and its interaction with the Ciao module system. Section 5.3 briefly describes the compiler itself, `ciaoc`. Section 5.4 describes a number of interesting errors that `ciaoc` can detect statically. Section 5.6 presents the different types of executables that `ciaoc` can build. Section 5.8 presents performance data, which illustrate the different tradeoffs involved. Section 5.9 compares the `ciaoc` compiler with some other compilers. Finally, Section 5.10 presents our conclusions.

5.2 The Ciao Generic Code-Processing Framework

In our experience with code processing tools in general and with global analysis tools in particular (such as the PLAI analyzer [MH92, BdlBH99]), we have come across the difficulty of being able to reproduce exactly the logic of the compiler when reading source files. Note that in a practical system it is necessary to deal with syntax extensions (operators, expansions, etc.), inclusions of code, redefinition of prolog flags, modules, imports, exports, local definitions, reexports,

visibility rules, external code, etc. We have come to the conclusion that the best way of reproducing exactly the compiler processing logic is to factor out this part into a library module such that the same code is used for all code processing tools. However, because these tools perform different jobs the library must be appropriately parametrized. To this end, we have implemented in Ciao a generic code processing framework which is used by the (WAM) compiler and also by the rest of our code processing tools. Currently, this framework is used by, in addition to the low-level compiler, all the preprocessor components (global analyzers, parallelizers, specializers, etc.), the automatic documentation generator, and the assertion processing library [BCC+97]. Note that, for simplicity, in the following we sometimes refer to this code processing framework as "the compiler", and to process a module with it to "compile" such module.

5.2.1 Modular processing

Within the framework, modules are processed in a separate fashion, that is, the "processing" (compilation, analysis, etc) is applied to one module at a time, but in such a way that the necessary information from related modules is available. This information on a module, which other modules using it need when being processed, is usually called the *interface* of the module. In other modular languages, such as Modula, in which there is a *definition* part and an *implementation* part for each module, such information is stored in the definition part. In the Ciao module system, as in that of other Prolog systems (e.g., [Swe95, Qui86]), the definition part is included in the source along with the implementation part, for programmer convenience. However, in order to obtain and store the interface separately and fully support separate compilation, in Ciao the compiler automatically extracts the interface definition from the source file. The advantage of this approach is that the user does not need to write a separate definition part as this is automatically done by the compiler. The extracted definition is stored by the compiler in a separate file from the one containing the actual code of the module: the *interface* file. From that point on, the interface file will be used by the compiler any time the interface part of the corresponding module is

needed. For brevity, we will call such files "`.itf`" files, in reference to the ending used in the interface files managed by `ciaoc`. However, note that different tools may need additional data from related modules. For example, several Ciao tools need information on the assertions present in the program, and this is stored in additional `.asr` files.

The process of extracting the interface information is performed the first time a module is processed, either by direct request or as a result of the processing of another module which uses it. At any time, the `.itf` files are automatically managed by the framework, which regenerates them when the source of the associated module changes. Processing a module requires only its code and the interface files of the modules imported by that module (i.e., the source of these modules is not necessary). The visibility of the source and `.itf` files can be controlled with standard file access permissions.[1]

One advantage of this code processing method, based on interface files, is that it allows dealing with *incomplete programs*. That is, a module can be processed even if the related modules are still incomplete or completely unavailable. Making dummy definitions of the related modules, defining only their exports, the compiler will produce `.itf` files of them which can be used to process the module in consideration. This can be used also for independent development of different parts of the program, which can then perhaps be performed in parallel by different teams. This also makes the early detection of compile-time errors in the module under consideration possible without having to wait for the code of the related modules to be ready.

Another characteristic of the framework is the automatic incremental (modular) processing of programs. This means that when, e.g., compiling a whole

[1]Thus, a programmer can generate and publish (e.g., for others to see) the interface file at any time by simply running the compiler on the corresponding source file and granting read permission to the `.itf` file generated. The same programmer can prevent another programmer from overwriting the existing interface file from what may be a possibly intermediate state of the source by simply not granting read permission to the source file or write permission to the `.itf` file. Furthermore, a precise textual description of the module interface in several documentation formats can be also be generated by simply running the auto documenter [Her00] on the source.

application, the compiler is able to recompile only those modules which need to, automatically following module dependencies starting from the main module of the application. To this end, the `.itf` files in Ciao contain also the information needed to traverse module dependencies and to decide if a module has to be recompiled (reprocessed), thus avoiding reading the source code of unchanged modules every time an application is processed. This information includes which modules are imported, which predicates are imported or reexported from each, which external code is included, etc. In summary, it contains the information which composes the premises under which the compilation took place.

An advantage of separate/incremental compilation is that it typically results in a faster development cycle, because it is not necessary to recompile the whole program whenever *any* change is performed on one of the modules composing the program, and only those modules affected by the changes will be recompiled. On the down side, less than optimal results (in terms of error detection, degree of optimization, etc., depending on the particular code processing performed) may be obtained compared to an equivalent monolithic compilation, unless some help from the programmer is provided. Note that modularity helps error detection in either case (monolithic/incremental compilation).

These are of course all well known advantages of modular program structure and modular processing, exemplified respectively by languages such as Modula and tools such as Unix `make`. The main novelty is in making this processing generic (so that it is implemented in a consistent way across many tools), highly automated and convenient, and adapting it to a Prolog environment, by virtue of a suitable module system design.

5.2.2 Operation and implementation of the framework

We now briefly outline the implementation of the Ciao generic code processing framework. It is implemented as a library (called `c_itf`) which exports the following high-order predicate:

```
process_files_from(File, Mode, Type, TreatP, StopP, SkipP, RedoP)
```

This predicate starts a code processing loop with file `File` at its root. `Mode` is the type of code processing, which may be internal compilation into the executable (*incore compilation*), compilation to `.po` object files, a re-entrant compilation (started by a `load_compilation_module` directive, see Chapter 3), or any other kind of code processing, as defined by the user of the library. `Type` indicates whether the starting file has to be a module or not (some processes can only be performed on modules, not user files). `TreatP`, `StopP`, `SkipP` and `RedoP` are predicate abstractions (see Chapter 4) which are called in several moments in the code processing loop, and customize it to the required task. They are called with an additional argument which is the identifier of the file being treated at that point, under which relevant data is stored. Their meaning will be explained bellow. The framework takes care automatically of low-level tasks including reading into a canonical form the code of the module (if required) and dealing with syntax extensions (operators, expansions, etc.), inclusions of code, redefinition of prolog flags, modules, imports, exports, local definitions, reexports, visibility rules, external code, etc., both in the current module and in the related modules. All this data is easily accessible through the predicates exported by the `c_itf` library.

The code processing loop proceeds as follows: if the related `.itf` of `File` exists and is newer than the source, then the `.itf` file is read, else the source is. If `StopP` succeeds for the file, neither the file is processed nor the loop deepens from it, finishing this branch. Else, the compiler collects the interface data of files from which this file performs imports (its "related files"), reading their `.itf` files (or the source files if those do not exist). Now, if `SkipP` succeeds for the current file, it is not processed further, but this time its branch is followed, entering in the processing loop with the related files. Otherwise, when entering to process the file, if we have read the source file (because the `.itf` file did not exist or was older) we generate a new `.itf` file and call `TreatP` to effectively process the file. If we have read the `.itf` file, we check if any dependence of this file has changed (e.g. an included file has changed, a predicate which this file imported from other module is no longer exported, etc.) and, if it is so, we read the source and proceed as before. Otherwise, we call `RedoP` to verify if we have to process the file anyway,

although the `.itf` file does not have to be regenerated. This predicate may, for example, consult additional files which are specific to the particular processing being performed, containing data from the module (as the assertion-cache `.asr` files used by the assertion processing library). Inside `RedoP` it is also possible to perform tasks related with this file which only need the `.itf` information, failing at the end. When the process on this file is over, the loop proceeds with the unprocessed, related files in the same way until there is no more work to do.

5.3 Compilation of Modular Programs: The Ciao Compiler (`ciaoc`)

We now describe the tasks performed by the Ciao standalone compiler (`ciaoc`),[2] which makes use of the code processing loop presented above, customized for a familiar task: traditional WAM-level compilation. When discussing the functionality, we will use the familiar Unix `cc/make` combination as a reference, since there are many similarities.

The objective of compilation is generally twofold. Firstly, the syntactic analysis typically performed by traditional compilers allows detecting a good number of errors in the program without actually having to run it. Examples are simple syntactic errors, singleton variables, discontiguous clauses, undefined predicates, etc. The second aim of compilation is to generate a lower-level representation of the program which can be executed more efficiently. E.g., in the case of `cc` each source file is compiled into a separate object (`.o`) file containing relocatable machine code. In the case of `ciaoc`, each module is compiled into a separate object (`.po`) file, containing (by default) WAM bytecode. This is the code that will be executed by the Ciao bytecode interpreter at run-time.

The following two sections explain the different tasks of compiling a single

[2] As mentioned before, the Ciao compiler is really a library module and can be used from the command line using the `ciaoc` application, from the familiar interactive toplevel shell, etc. While in the discussion we will mention only `ciaoc`, the descriptions given apply equally to the use of the compiler from the toplevel shell or as a library from another program.

module and incrementally compiling a whole program.

5.3.1 Compiling a Single Module (with the Related Interfaces)

As mentioned before, one of the main benefits of modularity is that it fits very well with the idea of *separate compilation*. The minimal units which can be compiled separately correspond to modules. `cc` itself performs typically only separate compilation of a single file: it is run on a `.c` file and produces a `.o` file. In `ciaoc` this kind of compilation can be performed by selecting the `-c` flag. For example, the command '`ciaoc -c module-a.pl`' performs separate compilation of `module-a` producing `module-a.po`.

As mentioned in Section 5.2, some information on other modules may be required. In the case of `cc`, the needed information is typically added explicitly to the "module" under consideration (as a result, a reduced amount of error checking can be made).[3] In the case of `ciaoc`, the information needed from related modules to process a module and obtain its compiled version is included in the interface files of these modules, which are automatically managed by the compiler.

5.3.2 Incrementally Compiling a Whole Program

The main difference with the previous task is that in this case the objective is not to compile a single module, but rather a set of related modules which compose a whole program and to produce an executable. The compilation starts with the *main* module, which spawns the compilation of other modules from which predicates are being imported. Typically all related modules, i.e., the transitive closure and not just the first level, must be processed. Nevertheless, processing is performed one module at a time, i.e., the compiler processes the code of only one module at each step. In the `cc/make` case this corresponds to writing a `Makefile`

[3]In fairness, C is not really modular – we are using it as an example only because the related compilation tools are very well known.

(possibly aided by running the `makedepend` command) followed by issuing a `make` command. I.e., the dependencies among modules are first extracted and then based on them the `make` utility determines which modules have to be recompiled.

The Ciao compiler automatically performs this process. Global compilation starts from a module, which is typically the main module of an application. If the module has a recent `.itf` file, it is read in order to decide if the module has to be recompiled and also to be able to follow its dependencies without reading the source. Else, if the source code is newer than the interface file, it is read and stored in memory. Then, all interface files of the modules directly used by this module are accessed. This may involve reading and storing in memory the sources of those modules as well, for those which do not have an up-to-date `.itf` file (later, when those modules are to be processed, their sources are retrieved from memory). Now, the `.itf` file of the current module is generated if needed. This cannot be done before because the interface file includes, among other information, which predicates are imported from other modules. If a module imports "everything" from another module (allowed by the module system design) to produce its interface the other module needs to be accessed to know which predicates it exports.

At this point, we are ready to process the current module, in this case by compiling it to WAM bytecode.[4] But this is done only if the `.po` (object) file does not exist or is older than the current module, or if the compiler finds that the current module needs recompilation. An already compiled module which has not changed could need recompilation if, for example, it uses a predicate from another module an the latter module stops defining or exporting it.

The process continues following the dependencies with the rest of the modules, compiling only the changed files and using the precompiled `.itf` and `.po` files of the older ones. When all the bytecode object files (with a `.po` extension) are up

[4]The compiler also supports external modules written in other languages, and in this case it also automatically calls if needed the right compiler to produce an object file, also possibly regenerating the interface and type conversion code (which is produced automatically from type and mode declarations given in the Ciao assertion language [PBH97, PBH00b] for the external procedures).

to date, they are collected and linked by the compiler to build the executable. Note that in a recompilation only the source files which have changed from the previous compilation or which are affected by these same changes have to be read and compiled. From the rest only the `.itf` file is accessed.

Interestingly, we have observed that, if the code for all required modules is available, users tend to choose this compilation scheme over that of the previous section, even when small changes are made, since `ciaoc` automatically determines the modules related to the module under consideration and follows the dependencies among modules deciding which modules require recompilation, without requiring any input from the user. Also note that the compilation method of the previous section can be used in conjunction with a traditional `Makefile` (for example when using Prolog files in the context of a larger, multi-language program). However, the Makefile needs to be updated by hand when an interface-related change is made. It is also possible to combine the two approaches and use `ciaoc` from within the `Makefile` to maintain the automatically up to date the Prolog files in the project. This can be done by writing a dummy file which uses all the Prolog files involved.

5.4 Errors Detected by the Compiler

Having discussed the issue of separate compilation and incremental recompilation, in this section we address the second objective of compilation: the early detection of programming errors. This includes also giving warnings about situations which, although strictly correct, are likely a programmer mistake. Here we only deal with the errors detected by `ciaoc`, which performs global analysis at the level of predicate imports and exports. Note that using the Ciao Preprocessor, which performs extensive global dataflow analysis, more (and more involved) error situations can be detected. The most interesting errors detected by the `ciaoc` compiler are:

- *Syntax errors:* Ciao gives the context in which the error is located and the point in the file where this context is located. This allows other tools (e.g.,

the graphical development environment – `emacs` interface) to automatically locate the point in the source file.

- *Unknown directives/declarations:* In Ciao, as in ISO-Prolog [PRO94, DEDC96], directives are not conventional queries, so only the ones defined by the language (or in Ciao also by special declarations, see Chapter 3) are allowed.

- *Redefinition of control constructs:* Although in Ciao it is possible to redefine all "builtins", redefinition of control constructs such as "`,`", "`;`", "`->`", etc. is more often than not due to a programming error and is thus flagged by default, as in other systems. Note that a redefinition of said constructs is commonly due to a programmer's mistake, putting a "." in place of a "`,`", as in the following code, which would define a clause for `qsort/3` (incomplete) and a "fact" for `(,)/2`:

```
qsort([X|Xs],L,L2) :-
        partition(Xs,X,Left,Right).
        qsort(Left,L,[X|L1]),
        qsort(Right,L1,L2).
```

- *Illegal imports:* Occur when a module attempts to import a predicate from another module which is not exported by the second.

- *Illegal module qualifications:* In Ciao it is not allowed to bypass module constraints, so this error is issued when a qualified call (as `module:pred(Args)`) is found to a predicate in a module which is not imported by the current module.

- *Undefined predicate calls (warning):* When compiling a module, in contrast to when compiling a user (nonmodule) file, if a call to a predicate not defined in the module nor imported from another module is found, a warning is given. Note that the detection of this problem is not possible without a proper module system. It is a warning and not an error because the

predicate could be defined by a dynamically loaded module. It is possible to explicitly declare the predicates imported from dynamically loaded modules in order to avoid these warnings.

- *Undefined predicate exports (warning):* This is signaled when a predicate in the export list of a module is not defined in the module.

- *Predicates with the same name and different arities (warning):* This warning, from our experience, detects a great number of hard-to-find errors, derived from changing the arity of a predicate in several places but forgetting to change one of the defining clauses. The warning is not issued if the different arity versions are exported by the module, since in that case it is clear that the the different arities exist on purpose. As many Prolog programmers do commonly define predicates with the same name and different arities, this warning can be locally or globally disabled with a Prolog flag.

- *Discontiguous clauses (warning):* As discontiguous clauses of the same predicate are usually the product of mistakes (and also forbidden by ISO-Prolog), this warning is issued when such a case is detected. Nevertheless, these warnings can be switched off by a per-predicate declaration (as in ISO-Prolog) or with a global compiler flag.

- *Singleton variables (warning):* This is the classic test in Prolog compilers, which has proven extremely helpful in finding errors derived from the misspelling of variable names.

- *Local definition of an imported predicate (warning):* This is allowed in Ciao (see Chapter 3), but the warning is signaled because it may be due to a programmer's mistake (as the importation may be of all predicates defined in a module). It can be switched off by adding a `:- redefining` declaration, for a given predicate or also for any predicate.

- *Duplicated importation of a predicate (warning):* This is also allowed in Ciao, but as the previous warning it is signaled because it may be due to

a programmer's mistake. It can be switched off in the same way as the previous one.

- *Exporting multifile predicates (warning):* Multifile predicates (see Chapter 3 and [PRO94, DEDC96]) do not need to be exported to be accessible by other modules defining them, thus this warning.

- *Incompatible declarations of multifile predicates:* This error is issued when a multifile predicate is defined differently in several modules (for example, in one is defined *dynamic* while in other is not).

In our experience to date with the Ciao system these messages avoid a great number of otherwise time-consuming errors and result in a much faster development cycle. This situation is enhanced further if the preprocessor is also used. Note however, that some of the choices made in the Ciao module system, which is slightly more strict that the usual Prolog module systems, are necessary to be able to detect many of these errors.

5.5 The `initialization` Directive

The ISO Prolog standard, in its first part [PRO94, DEDC96], introduced very appropriately the `initialization/1` directive to specify goals to be executed just when a Prolog code starts execution. In that first part, the order of execution of different initialization directives was left undefined. Unfortunately, the second part of the standard, related to modules [PRO00], did not add any regulation about this issue.

When we implemented the `initialization/1` directive in Ciao, in the context of a modular Prolog implementation, we found that the order in which the goals present in different occurrences of the directive are executed is indeed critical. Note that in most cases the initialization directive of a module will involve the execution of code which needs to be executed before any (or some) of the exported predicates of the module are executed. Thus, in the event of two modules `a` and `b`, both containing an initialization directive, if `a` uses `b` then the

initialization of `b` must go before the initialization of `a`, as the initialization of `a` may involve a predicate exported by `b`. Of course in the event of mutual use an accessibility analysis would be needed to find the correct execution order. This restriction applies also in the event of transitive use: if `a` uses `b` which in turn uses `c`, the initialization of `c` must precede the initialization of `a` (even if `b` has no initialization). The reason is the same: the initialization of `a` may involve calling predicates of `b`, which in turn may involve calling predicates of `c`.

Under these premises, and in the context of separate compilation, the implementation of the `initialization/1` directive in Ciao is as follows:

1. For each `:- initialization(`*`Goal`*`)` directive in a module M, the compiler adds to the object code of the module a clause of an internal multifile predicate "`initialization(`*`M`*`) :- `*`Goal`*`.`". This multifile predicate is not normally accesible from user code.

2. Additionally, when a module is compiled its dependencies are also stored in the object code using another internal multifile predicate. Thus, the object code of module M will include a clause "`u(`*`M`*`,`*`J`*`).`" for each module J from which module M imports (that is, for each `:- use_module(`*`J`*`)` declaration it includes).

3. When an executable is compiled, an internal clause `main_module(`*`Main`*`)` is added to the code to record the main module of the application (that is, the module from which the rest of the code is required).

4. Besides, there is certain internal code which is included in all executables, such as a basic exception handler, and related to the subject under discussion, code to start the execution of the initialization directives. In the code shown below, predicate `initialize/0` takes care of the execution of the initialization code scattered in the object files which comprise the executable, in the right order (note that the predicates `main_module/1`, `u/2` and `initialization/1` below refer to the internal predicates mentioned

above):[5]

```
initialize:-
        main_module(M),
        initialize_module(M),
        fail.
initialize.

:- data initialized/1.

initialize_module(M) :- current_fact(initialized(M)), !.
initialize_module(M) :- asserta_fact(initialized(M)),
                        do_initialize_module(M).

do_initialize_module(M) :-
        u(M, N),
        initialize_module(N),
        fail.
do_initialize_module(M) :-
        initialization(M),
        fail.
do_initialize_module(_).
```

The code accounts for the cases in which a module has no initialization directive, has one or more, and initialization directives fail or succeed. Note that all solutions of the goals present in initialization directives are explored. When there are several initialization directives in the same module, they are executed in order, and a cut present in one cuts the execution of the rest below. Of course this last behavior could be easily avoided, but we feel that it can be useful indeed.

[5]The code uses the special kind of dynamic declaration `data`, see Chapter 3 for details.

5. In the toplevel shell, or in any case when a module M is loaded dynamically (which may in turn involve other module loadings), just after the loading is completed a goal `initialize_module(M)` is invoked to initialize the module and all its dependent modules appropriately. Note that in the case of reloadings, the compiler needs to explicitly handle data facts of `initialized/1`.

5.6 Types of Executables Created / Linking Regimes

The Ciao compiler can create different types of executables, depending on the linking regime used for the modules involved. Essentially, modules can be linked into the executable in three ways. If linked *statically*, then the bytecode of the module is added to the executable when building it. This has the advantage that this module does not need to be found elsewhere at execution time but results in a growth in the size of the executable. If linked *dynamically* then the bytecode for the module is not included in the executable but is instead searched for in certain directories defined by a set of *library paths* and loaded at the time when the application starts executing. This has the advantage of a smaller executable size but the required modules must be accessible a run-time for the application to be executed correctly. It is used most frequently with the standard libraries, which can often be assumed to be accessible in the execution environment. Finally if the module is linked *lazily*, then the code for the module is not included in the executable and is instead searched for in the *library paths* directories in the same way as with dynamic linking, but only if during execution the application calls a predicate defined in the module, and at the time of that first call. This has the advantage that the application can start more quickly and that modules which are not used in a particular run do not need to be loaded. Which of these regimes is used for a given module is controlled in a flexible way by associating a loading regime to a certain *path alias* [Swe95], so that a module referred to using that path alias is loaded using the corresponding regime. By default, modules in

`library` paths are loaded dynamically, whereas application-specific modules are included in the executable.

There is an additional "executable" type which is not created with the Ciao command-line compiler: Ciao "shell scripts." They are similar to, e.g., the UNIX shell scripts (or the way in which `perl` programs are typically run), but are executed by the Ciao script shell `ciao-shell` which, as mentioned before, is another application which uses the compiler library. The difference with a usual command interpreter script is that the source is compiled (by default; this can be overridden) the first time the application is started, or after a change in the source. This can sometimes be advantageous with respect to creating binary executables for small- to medium-sized programs that are modified often and perform relatively simple tasks. The advantage is that no explicit call to the compiler is necessary, and thus changes and updates to the program imply only editing the source file and invoking again the executable. The disadvantage is that startup of the script (the first time after it is modified) is slower than for an application that has been compiled previously. The interested reader is referred to [Her96, CHV96] for more information and some interesting applications of such scripts.

5.6.1 Issues in Lazy linking

Lazy linking is implemented by creating "stump" code which defines each predicate exported by the module as a call to load that module followed by a call to the predicate again. For example, predicate `foo/3` in (lazy-load) module `bar` would be implemented by something like:

```
foo(A,B,C) :- load_lib_lazy(bar), foo(A,B,C).
```

At the time of the last call after the loading, the new definition of predicate `foo/3` has replaced the stump definition, so that the recursive call executes the predicate as if it were loaded from the beginning. Each module to be lazily loaded is replaced in the executable by the code containing the stump predicates of that module, so that the first time an exported predicate of the module is called,

the module is loaded. Here the fact that the Ciao module system is strict is instrumental, since if this were not the case all predicates would need a stump definition, not only the exported ones, because external calls to any predicate of the module could happen.

Note that due to the way lazy linking is currently implemented, it is complicated to implement this type of loading for certain types of modules (which are then loaded eagerly in the current implementation). Even in the strict module system of Ciao, in which only exported predicates can be accessed, there are situations in which a module can use the code of the other module without executing an exported predicate of it. One of these situations is when a module exports a dynamic predicate, which can then be accessed by other modules, for example asserting clauses of it. The other situation is when several modules contain the same multifile predicate: when one of these modules calls the multifile predicate all clauses of the predicate need to be present, so the other modules need to be loaded also. These restrictions lead to a transitive relation of "requirement", which defines which other modules need to be loaded when a given module is loaded, in order to safely execute the code. The relation can be computed, as sketched before, as the transitive closure of relation $\mathcal{R}$, defined as the pseudo-clauses:

$\mathcal{R}$(A,B) :- {module A imports a dynamic predicate from module B}.
$\mathcal{R}$(A,B) :- {module A and module B share a multifile predicate}.

Notice that since the main module (the module which contains the starting predicate) is normally linked statically, as in any case it has to be loaded for the executable to start, the above requirement relation dictates that possibly other modules have to be linked the same way. In addition, the requirement restriction has to be observed also in the stump clauses: a stump clause of a predicate of a module has to load along with that module other modules required by it.

There are situations in which, in spite of the $\mathcal{R}$ requirement relation, there is no possibility that the first module uses code of the second before the second is loaded. This is the case when the manipulation of dynamic predicates of the second module or the calls to shared multifile predicates which occur in the first

module are always preceded by executing an exported predicate of the second module, since that way by the time the conflictive call is made the second module will be already loaded. Thus, the compiler provides a mechanism to specify that a module is safe to be lazily loaded, even if a conflict situation is detected by the compiler, because the programmer may know that that is the case. Note that in fact many of these situations can be detected by a more extensive global analysis than that which the current version of the compiler performs.

5.7 Active modules

Active modules [CH95b] provide a high-level model of inter-process communication and distributed execution. An *active module* is an ordinary module to which computational resources are attached, and which resides at a given location on the network. Compiling a module as an active module produces an executable which, when running, acts as a *server* for a number of predicates: the predicates exported by the module. Predicates exported by an active module can be accessed by a *client* program on the network by simply "using" the module. The process of "using" an active module does not involve transferring any code, but rather setting up things so that calls in the module using the active module are executed as remote procedure calls to the active module. This occurs in the same way independently of whether the active module and the using module are in the same machine or in different machines across the network.

From the implementation point of view, active modules are essentially daemons: executables which are started as independent processes at the operating system level. Communication with active modules is implemented using sockets (thus, the address of an active module is an IP socket address in a particular machine). Requests to execute goals in the module are sent through the socket by remote programs. When such a request arrives, the process running the active module takes it and executes it, returning through the socket the computed answers. These results are then taken and used by the remote processes. Backtracking over such remote calls works as usual and transparently. The only limitation (this may change in the future, but it is currently done for efficiency reasons) is

that all alternative answers are precomputed (and cached) upon the first call to an active module and thus *an active module should not export a predicate which has an infinite or too large number of answers.*

Except for having to compile it in a special way, an active module is identical from the programmer point of view to an ordinary module. A program using an active module imports it and uses it in the same way as any other module, except that it uses the declaration "`use_active_module`", defined in the Ciao package `actmods`, rather than the usual declaration "`use_module`". Also, an active module has an address (network address) which must be known in order to use it. Some "protocols" are provided to allow the clients to find out the addresses of the active modules they use, to avoid having to provide that address explicitly in the client code. The protocols involve a procedure to publish the address, to be used by the active module, and a procedure to locate the address of a given active module, used by the clients.

5.7.1 Compiling an Active Module

To compile a module as an active module, a method for "publishing" its network address needs to be provided, so that clients can access it. Using the Ciao standalone compiler, an active module is compiled using the `-a` option, which expects after that option a library file where the publishing code resides. For example, invoking the Ciao standalone compiler as

```
ciaoc -a 'actmods/filebased_publish' server
```

the module `server` is compiled as an active module, using the "filebased" method for publishing, located in the library path `actmods/filebased_publish`. The compilation can also be done from the interactive top-level shell using the predicate `make_actmod`/2, provided by the compiler library.

The compiler, to make up the active module executable, creates a main entry file which imports the module to be activated and the publishing library, plus a standard module included in all active module executables. This module defines an entry point which sets up the socket where the clients will connect, calls the

publish method in the corresponding module with the resulting network address, and enters a loop for processing all incoming predicate execution requests, calling the predicates in the activated module as needed. For this purpose, the list of exported predicates of the activated module is also collected in the compilation process and included in the executable as facts.

5.7.2 Using Active Modules

As said before, the library package `actmods` needs to be used in order to use active modules in a source file. Then, to import a series of predicates from an active module, a declaration "`:- use_active_module(Module,Predicates)`" is used, where `Module` is the name of the active module and `Predicates` is the list of predicates to be imported. The imported predicates must be exported by the active module. From this point on, the code should be written as if a standard `use_module`/2 declaration had been used. The `actmods` package defines an expansion so that a `use_active_module` declaration produces a series of predicate definitions to remotely call the imported predicates. For example, if the predicate *P* is imported from the active module *M*, the corresponding produced predicate would be defined as

```
P :- module_address(M,A), remote_call(A,P)
```

The definition is based on the `remote_call`/2 predicate, defined by the `actmods` package, and a hook predicate `module_address(Module,Address)` which must give, for each active module imported in the code, its address. The scheme is very flexible, because the predicate `module_address`/2 itself can be imported, thus allowing the creation of module libraries, corresponding to the publish methods above mentioned for the active module servers, which define the predicate in a suitable way. For example, for the `actmods/filebased_publish` module used above for the compilation of the active module, there is a corresponding `actmods/filebased_locate` to be used by client modules of the active module, which appropriately defines `module_address`/2. In this protocol, a directory accessible by all the involved machines (e.g., by NFS or Samba) is used to

store the addresses of the active modules, and the `module_address`/2 predicate would examine this directory to find the required data. For efficiency, the client methods maintain a cache of addresses, so that the server information only needs to be read from the file system the first time the active module is accessed.

The Ciao standard library provides several additional protocols for the communication of active module addresses to their clients. A particularly useful one is based on a *name server* for active modules. When active modules start, they communicate their address to the name server. When a client request the address of an active module, it asks the name server the active module address. This is all done transparently to the user. The name server must be running when the active module is started (and, of course, when the application using it is executed). The name server is itself an active module, but its address should be fixed and known prior the execution of other active modules or client applications.

5.7.3 Using Active Modules: an Example

In this section we will show the implementation of a simple remote database server using the primitives introduced in the previous section, and how the server would be used.

The code for the server is a normal module, which exports the relevant predicates. Note that it uses a `data` predicate, a special class of dynamic predicates which only stores facts (see Chapter 3):

```
:- module(database, [stock/2, add_items/2]).

:- data stock/2.

stock(p1, 23).
stock(p2, 45).
stock(p3, 12).

add_items(P, I) :-
    ( retract_fact(stock(P,K0)) -> true ; K0 = 0),
```

```
    KI is K0+I,
    asserta_fact(stock(P,KI)).
```

To compile the module as an active module, we have to select a method to "publish" its network address, say we choose `'actmods/filebased_publish'`. Then, at the OS shell we will do:

```
$ ciaoc -a 'actmods/filebased_publish' database
```

At this point the executable "database" would be started as a process (at the unix level "`database &`") and it would be ready for other modules to import it. The code of a module that uses the previous active module could start like this:

```
:- module(sales,_).

:- use_package(actmods).
:- use_module(library('actmods/filebased_locate')).
:- use_active_module(database, [stock/2,add_items/2]).

replenish(P) :-
        stock(P, S),
        ...
        add_items(P, X),
        ...
```

Note that we use the library `'actmods/filebased_locate'` in correspondence with the one chosen to compile the active module. Calls to `stock`/2 and `add_items`/2 in the previous module will be executed remotely by the active module "database". Except for the starting declarations, the code would be identical to the local case, when `use_module` replaces `use_active_module`.

For an explanation of the use of active modules in the context of WWW programming, see [CHV96, CH97, CH01] (the abstract of the last one is included in the Introduction, page 12).

5.8 A Preliminary Experimental Evaluation

The incremental compiler has been fully functional for quite some time now (6 years at the time of this writting) and has been used in a good number of academic and commercial applications, in addition to compiling all the components of the Ciao program development environment. While an exhaustive evaluation of the compiler is left as future work, in this section we present some preliminary experimental data on the compiler. All experiments were done in a Pentium-II dual processor at 400 MHz running Linux, shared with a number of other users.

First, and in order to test the incrementality of the compiler, we compare the times required to (re)compile a medium-sized application (in fact, the standalone compiler `ciaoc` itself) depending on the number of files changed. The size of the source comprising this application, including standard libraries, is of about 15,000 lines (53,000 words using UNIX `wc`).

	D	S
From scratch	15.59	18.29
Compiler files	9.77	12.57
Main file	2.82	5.61
No changes	2.62	5.30
No executable generation	2.13	2.13
Executable size (Kb)	167	1105

Table 5.1: Times (Secs.) to compile `ciaoc`, for different changes in source.

Table 5.1 shows times in seconds when a dynamic (D) or a static (S) executable is produced. The first row shows the compilation times starting from scratch, that is, no `.itf` (interface) nor `.po` (object) files exist of any source module, including system libraries, i.e., the complete 15,000 lines are compiled. The second row shows the compilation times when system libraries are precompiled, but all compiler-specific source is not. This includes 6 modules, which comprise 52% of the total number of lines and 49% of the total number of words in the whole source. The third row shows the compilation times when only the main

module of the application has to be recompiled; system libraries are all precompiled. The main module contains 2.8% of lines and 3.7% of words of the total. Note that in this case, as in the previous case, the fact that many modules are precompiled and unchanged is not known a priori by the compiler, which needs to check that this is so. The fourth row shows the times needed by the compiler to generate the executable file after checking that no source has changed from a previous compilation (all sources are precompiled), i.e., this is the time to link and write out the executable. The fifth row shows the times under the previous scenario but without generating the executable (thus there is no difference between dynamic or static compilation). The last row shows the size in kilobytes of the executables generated. From the numbers we can conclude that the system is indeed incremental, and incremental compilation is indeed advantageous. Also, we observe that the difference between producing a static or a dynamic executable is related mainly to the relative sizes of the executables, as was to be expected.

We now compare the different linking regimes for executables with regard to compilation time, starting time, size of executable, and memory consumed. The executables used in the tests were: `ciaosh`, the Ciao toplevel shell, a quite large application; `pldiff`, an application to compare Prolog files (à la UNIX `diff`), but disregarding formatting of the code or variable renaming; `wumpus`, an application to solve wumpus world problems [RN95], an AI classic; and `suite`, a set of test programs distributed with the Ciao system (includes several classical benchmarks: queens, fib, etc.).

The results are shown in Table 5.2. Data is given for static (S), lazy (L) and dynamic (D) linking type executables, and for each application. Times are expressed in seconds, sizes in kilobytes.

- T_c is the time that it takes to generate the corresponding executable, given that all source modules have precompiled objects. Note that in the case of lazy linking, a part of this time is spent in compiling the stump code of the lazily-loaded modules. Compilation time is always fastest for dynamic executables, slower for static executables.

- Sz is the size of the executable. Here again the lazy executable is somewhat bigger than its dynamic equivalent because of the inclusion of stump code.

- T_0 is the time to start the application, which includes the loading and partial re-linking of starting object code. Here the lazy executable is fastest, because lazily-loaded modules are not loaded and linked yet. This speedup depends on the amount of lazily-loaded modules compared to the total executable code. Note also that the dynamic executable is a bit slower than the corresponding static executable: in the latter all object code is available just there.

- T_1 is the time spent until work which forces the loading of all modules is done. Surprisingly, in a benchmark (`pldiff`) T_1 is smallest for the lazy executable, and in another (`ciaosh`) smaller for the lazy executable than for the dynamic executable. One explanation of this phenomenon might be that by spacing the file accesses, the operating system performs faster each access. Another explanation would be related with fewer page faults and/or more cache hits, since the lazily loaded modules start to execute right after they are loaded.

- M_0 is the memory taken up by the application at time T_0. The amount is quantized because Ciao reclaims memory in chunks, so two equal quantities may represent different real used amounts. The amount is smaller for the lazy executable since there are modules which have not been loaded yet, and which are only represented by their corresponding stump predicates (normally much smaller-sized). The dynamic executable uses slightly more memory than the static executable because the process of finding the modules dynamically consumes some memory.

- Finally, M_1 is the memory taken up by the application at time T_1. Here the lazy executable catches up with or passes the static executable in amount of memory. Also, one of the benchmark shows the dynamic executable using up more memory than the lazy executable. This is because the code which performs the dynamic load in the case of lazily-loaded modules (stump

code) is replaced by the object code of the module, while in the dynamic executable it remains in memory. This amount, nevertheless, is minimum, but, as explained before, due to the quantized amounts of memory reclaimed by the system it may appear bigger if it causes to cross a memory usage step.

	T_c	Sz	T_0	T_1	M_0	M_1
`ciaosh`						
S	6.38	1362	1.07	1.19	2704	2704
L	3.92	332	0.84	1.20	2440	2704
D	2.95	245	1.14	1.26	2704	2704
`pldiff`						
S	2.59	352	0.28	0.34	1776	2172
L	2.32	185	0.15	0.32	1644	2172
D	1.95	156	0.29	0.35	1780	2172
`wumpus`						
S	2.12	261	0.21	0.46	1644	2040
L	2.04	194	0.16	0.49	1644	2172
D	1.93	179	0.22	0.48	1780	2172
`suite`						
S	2.25	268	0.22	5.12	1644	3584
L	2.09	171	0.14	5.14	1512	3584
D	1.92	152	0.22	5.13	1780	3716

Table 5.2: Differences among static, lazy and dynamic executables (Secs., KB).

The results confirm our design intuition that producing static executables is mainly suited for applications which have been tested and which are ready to distributed stand-alone to other environments, where (Ciao) Prolog cannot be assumed to be installed a-priori. Producing dynamic executables, on the other hand, is the best choice for environments where (Ciao) Prolog can be assumed to be installed, and also at development time, when one wants to compile the

executable as fast as possible (however, the star during the development cycle is using the toplevel shell from the emacs interface). Finally, lazy executables are very suitable for applications which have parts which may not be used in every run. An example is the Ciao preprocessor, which can perform a very wide variety of tasks, but which in a given run can sometimes be used repeatedly for a very specific task comprising a very small fraction of its overall functionality. The lazy-load executable will automatically load only the small part actually being used, reducing load time and the size of the image in memory.

5.9 Comparison with Other Systems

We now make some comparisons between the Ciao compiler and the compilers of some other Prolog systems which are popular and of comparable run-time performance. In particular, we discuss differences with SICStus Prolog 3.7.1 (the version at the time of performing the tests) and Calypso/GNU Prolog 1.0.0 (which has a native-code compiler). Some observations regarding functionality:

- The 3.7.1 SICStus compiler is a very straightforward file-by-file incore compiler which performs virtually no local or inter-module checking at compile-time. It does not keep track of the fact that the compilation of a given file may affect the compilation of others. Also, all operators, expansions, etc. are global, so that they only need to be loaded, but not unloaded. Finally, the module system is supported dynamically, so that there is little overhead at compile-time related to module management.

- The Calypso/GNU Prolog compiler falls in between the SICStus and Ciao compilers. On one hand it does not support modules as SICStus or Ciao and it is not incremental. On the other hand it does perform some inter-file analysis to eliminate dead code and detect a number of errors (although, because of the lack of a module system, this can only be done at link time when the source of all the files of an application is present).

- In contrast, the Ciao compiler performs extensive inter-module dependency

analysis and error checking, even if only (re)compiling a single module. To this end, the compiler must keep track of all the interfaces and, to support incrementality, read and write interface information and bytecode from and to interface and object files. Also, since in the Ciao module system operators and expansions are local to each module and user file (which we believe is vital to realistically address separate compilation or modular global analysis), they must be loaded and unloaded into the compiler for each file. Also, the module system is implemented mostly statically, which requires program transformations at compile-time (in return for slightly faster execution).

- The executables of SICStus and Ciao are multi-platform, and can be executed in any machine where an "engine" is available. In contrast the Calypso/GNU executables are platform-dependent binary executables. This gives them an advantage in startup time at the expense of portability.

- SICStus and Calypso/GNU only generate static executables, whereas Ciao supports dynamic and lazy-load executables.

We leave performing a comparison of executable size and compilation speed as future work. However, preliminary results show that the size of the *static* executables generated is comparable for Calypso/GNU and Ciao. SICStus does not really build executables, but rather saved states. These saved states were much larger than the executables of Calypso/GNU and Ciao in SICStus version 3.6, but version 3.7 has an excellent *save program* facility which produces very small saved states. The size of these 3.7 saved states when added to the engine size is comparable to those of Calypso/GNU and Ciao.

As for compilation speed, the emphasis to date in the Ciao compiler has not been compilation performance, but rather programming error detection and developing the module system and the resulting incremental and separate compilation capabilities. The current situation seems to be that the Ciao compiler is of similar performance as the others for large programs and in incremental situations (e.g., recompiling one or a few files in a large executable). However, its seems

to be quite a bit slower for small programs or when compiling large programs from scratch, presumably because it must create all the interface and object files. We expect optimizations to improve the raw compiler speed. However, it must also be realized that since the Ciao compiler does much more, it necessarily must take longer to do it. One interesting point is that the compilation time scales well with size. This confirms some of the design decisions aimed at supporting programming-in-the-large.

5.10 Chapter Conclusions

We have presented the Ciao code processing framework, which allows the development of program analysis and transformation tools in a way that is largely orthogonal to the details of module system design. We believe that, for any system which provides a number of code processing tools, having such a framework available and used by all the tools simplifies tremendously the task of making the behavior of the tools consistent. The framework, and the compiler, which is an instance of it, provide separate and global incremental compilation, automatically following module dependencies and recompiling obsolete object and interface files. We have found this to be a very useful feature for any medium- to large-size project. We have presented also some error and warning messages that the compiler can detect statically, which are in practice of great help in avoiding a number of otherwise time-consuming errors. The combination of incremental compilation and additional static error detection results in our experience in a much faster development cycle than with more traditional environments. Regarding the `initialization/1` directive, we have explained a correct implementation of it in the context of modular compiling. We have presented the different types of executables that the compiler creates (including the novel concept of "active modules"), and through experimental data we have illustrated the different tradeoffs involved. We have found that the tradeoffs are such that there are different uses and environments which make each of these executable types to be the most suitable, and have therefore decided to keep them all as compiler options. Regarding the overall compiler behavior, in our own (admittedly, perhaps biased)

experience, we find the Ciao compiler addictive, despite the larger compilation times of this early version, due, among other reasons, to the above mentioned faster development cycle and the much improved error detection capabilities.

Part III

Applications of Global Analysis

Chapter 6

Parallelizing Prolog Programs using Global Analysis and Non-Strict Independence

In this last part of the thesis we present our work on automatic parallelization of logic programs based on global analysis. In fact, it was precisely our early work on this issue what lead us to pursue the design of a logic programming language with the characteristics that we have discussed so far.

Logic programming systems which exploit and-parallelism among non-deterministic goals rely on notions of independence among those goals in order to ensure certain efficiency properties. "Non-strict" independence (NSI) is a more relaxed notion than the traditional notion of "strict" independence (SI) which still ensures the relevant efficiency properties and can allow considerable more parallelism than SI. However, all compilation technology developed to date has been based on SI, because of the intrinsic complexity of exploiting NSI. This is related to the fact that NSI cannot be determined "a priori" as SI, but needs a compile-time global analysis of the code. This chapter fills this gap by developing a technique for compile-time detection of NSI. It also proposes algorithms for combined compile-time/run-time detection, presenting novel run-time checks for this type of parallelism. The approach is based on the knowledge of certain prop-

erties regarding the run-time instantiations of program variables —sharing and freeness— for which compile-time technology is available, with new approaches being currently proposed. The techniques are fully implemented in a parallelizing compiler and performance results from this implementation are presented.

6.1 Introduction

Several types of parallel logic programming systems and models exploit and-parallelism [Con83] among non-deterministic goals. Some examples are ROPM [RK89], AO-WAM [GJ89], DDAS/Prometheus [She92], systems based on the "Extended" Andorra Model [War90] such as AKL [JH91], and &-Prolog [HG91] (please see their references for other related systems). All these systems rely on some notion of independence (or the related notion of "stability" [HJ90]) among non-deterministic goals being run in and-parallel in order to ensure certain important efficiency properties. Two basic notions of independence are strict and non-strict independence [HR89, HR90].

Strict independence (SI) corresponds to the traditional notion of independence among goals [Con83, DeG84, HG91]: Two goals g_1 and g_2 are said to be strictly independent for a substitution θ iff $\text{var}(g_1\theta) \cap \text{var}(g_2\theta) = \emptyset$, where $\text{var}(g)$ is the set of variables that appear in g. Accordingly, n goals $g_1, \ldots, g_n$ are said to be strictly independent for a substitution θ if they are pairwise strictly independent for θ. Parallelization of strictly independent goals has the property of preserving the search space of the goals involved so that correctness and efficiency of the original program (using a left to right computation rule) are maintained and a no speed-down condition can be ensured [HR89]. A convenient characteristic of strict independence is that it is an "a-priori" condition, i.e. it can be tested at run-time ahead of the execution of the goals. Furthermore, tests for strict independence can be expressed directly in terms of groundness and pairwise independence of the variables involved. This allows relatively simple compile-time parallelization by introducing run-time tests in the program [DeG84, MH90]. These tests can then be partially eliminated at compile-time by direct application of groundness and sharing (independence) information obtained from global analysis [JL89, MH89,

BdlBH93].

Non-strict independence (NSI) is a relaxation of strict independence defined as follows [HR95]:

> Consider a collection of goals $g_1, \ldots, g_n$ and a substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i < j \leq n, v \in (var(g_i\theta) \cap var(g_j\theta))\}$. Let θ_i be any answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if the following conditions are satisfied:
>
> - $\forall x, y \in SH$, $\exists$ at most one $g_i\theta$ such that for a θ_i we have that $x\theta_i \not\equiv x$ or that $x \neq y$ and $x\theta_i = y\theta_i$.
> - $\forall x, y \in SH$, if $\exists$ $g_i\theta$ meeting the condition above, then $\forall g_j\theta, j > i$, such that $\{x, y\} \cap var(g_j\theta) \neq \emptyset$, g_j is a pure goal, and for all θ_j partial answer during the execution of $g_j\theta$ $x\theta_j \equiv x$ and $x\theta_i \neq y\theta_i$ if $x \neq y$.

Intuitively, the first condition of the above definition requires that at most one goal further instantiate a shared variable or alias a pair of variables. The second condition requires that any goal to the right of the one modifying the variables be pure and do not "touch" such variables in its execution. This ensures that its search space could not have been pruned by any bindings made to those variables and therefore it is safe to run it in parallel. Here pure is applied to a goal that has no extra-logical builtins which are sensitive to variable instantiation.

This definition is a generalization of that originally given in [HR90]. In the case in which no information is available on the purity of goals (and, thus, all goals have to be conservatively assumed to be impure) the definition of non-strict independence can be simplified as follows:

> Consider a collection of goals $g_1, \ldots, g_n$ and a substitution θ. Consider also the set of shared variables $SH = \{v \mid \exists i, j, 1 \leq i < j \leq n, v \in (var(g_i\theta) \cap var(g_j\theta))\}$. Let θ_i be any answer substitution for $g_i\theta$. The given collection of goals is non-strictly independent for θ if

$\forall x, y \in SH$, at most the rightmost $g_i\theta$ such that $x, y \in \text{var}(g_i\theta)$ has an answer substitution θ_i for which $x\theta_i \not\equiv x$ or $x \neq y$ and $x\theta_i = y\theta_i$.

That is, only the rightmost goal where a shared variable occurs can further instantiate it, and only the rightmost goal where two shared variables occur can alias them. Clearly, this second definition is easier to implement, not only because no information is needed regarding the purity of the goals, which is in practice actually relatively easy to obtain, but also because no information is needed regarding partial answers, which in general is more difficult to obtain from analyzers. This chapter is focused mainly on this second definition, although some results are offered for the previous, more general, one.

Non-strict independence is clearly a more powerful notion than strict independence since strictly independent goals are always non-strictly independent. Furthermore, it still preserves the same properties as strict independence with respect to correctness and efficiency. In practice, it has wide application for example in the parallelization of programs which use difference lists, and incomplete structures in general. In fact, studies of amounts of ideal parallelism in logic programs suggest that there is a potential for large speedups from the exploitation of non-strict independence [She92]. However, this potential remains untapped from the point of view of automatic parallelization. This is due to two factors. The first one is that non-strict independence is not an "a priori" condition, i.e. it cannot be expressed simply in terms of run-time tests (without running the goals). Thus, run-time detection by itself is ruled out. Unfortunately, compile-time detection is complicated by the fact that non-strict independence is not directly expressed in the same terms as the properties which are usually determined from global analysis.

Earlier studies [HR89] have suggested that coupling sharing and groundness analysis with freeness analysis could be instrumental in the task of non-strict independence detection. This has been one of the motivations behind the development of analyzers capable of inferring these three types of information [CDFB93, CF91, Sun91, MH91, DJBC93]. However, there still remained a semantic gap between the availability of that information and actually being able

to reason about the non-strict independence of a set of goals, since this topic is not addressed at all in a satisfactory way in either [HR89], [HR90] or [HR95], or in any other work that we are aware of. This chapter attempts to fill this gap. It aims to develop concrete techniques for determining non-strict independence at compile-time. For concreteness, this chapter focuses on a particular way of expressing sharing and freeness information, the sharing+freeness domain [MH91]. This allows a high degree of precision in the conditions involved, which are given in such a way that the implementation is straightforward. However, we believe that the ideas presented can also be used for related domains, provided that these domains give information about variable sharing and freeness.

A decision throughout our research has been to concentrate on the independence of two goals. This is convenient from a practical point of view because many parallelization algorithms work by repeatedly considering whether two goals are independent while, for example, building a dependency graph. The decision has also a sound theoretical foundation. Consider, for the case in which no information is available on the purity of goals, the following alternative definition of non-strict independence:

> Given two goals g_1 and g_2, where g_2 is to the right of g_1, and a substitution θ, consider the set of shared variables $SH = \text{var}(g_1\theta) \cap \text{var}(g_2\theta)$. The goals g_1 and g_2 are non-strictly independent for θ if for any answer substitution θ_1 of $g_1\theta$ and for all $x, y \in SH$, $x\theta_1$ is a variable and if $x \neq y$, $x\theta_1 \neq y\theta_1$.

Based on this definition, the definition involving n goals can be expressed as follows: $g_1, \ldots, g_n$ are non-strictly independent for a substitution θ if they are pairwise non-strictly independent for θ. Clearly, this is equivalent to the previous definition, and thus considering only pairs of goals can be done without loss of generality.

The rest of the chapter proceeds as follows: Section 6.2 explains the particular abstract interpretation domain for which the conditions of parallelism are given, the sharing+freeness domain, and introduces a novel pictorial representation for the abstract substitutions involved. Section 6.3 presents the sufficient conditions

proposed for compile-time detection of NSI. Section 6.4 deals with the combination of compile-time analyses and run-time checks for detecting NSI, presenting novel run-time checks for this type of parallelism. It also connects this method with the previously proposed techniques for the detection of strict independence. Section 6.5 presents a new algorithm for automatic parallelization suited to the special characteristics of non-strict independence. Section 6.6 deals with the issue of environment separation optimization. Section 6.7 illustrates the techniques proposed by using them to parallelize of a concrete program. Section 6.8 gives some experimental results showing the speedups obtained in several programs presenting non-strict independence but no strict independence. Section 6.9 discusses improvements to the compile-time analysis in order to enrich the information available for the parallelization. Finally, Section 6.10 gives the conclusions and suggests future work.

6.2 Understanding Sharing+Freeness Abstract Substitutions

It is assumed that the reader has sufficient background in abstract interpretation (see [CC92] for details).

The sharing+freeness abstract domain [MH91] (other related analyses for which our results may be valid include [CDFB93, CF91, DJBC93]) was proposed with the objective of obtaining at compile-time accurate variable *groundness*, *sharing*, and *freeness* information for a program, i.e., respectively, information on when a program variable will be bound to a ground term, when a set of program variables will be bound to terms with variables in common, and when a program variable will be unbound or bound only to other variables instead of to a complex term.

The abstract domain approximates this information by combining two components (in fact domains per se): the first component provides information on sharing (aliasing, independence) and groundness [JL89, MH89, MH92, JL92]; the second one provides information on freeness.

We will denote a sharing+freeness abstract substitution as a pair (sharing, freeness) as in $\widehat{\theta} = (\widehat{\theta}_{\mathrm{SH}}, \widehat{\theta}_{\mathrm{FR}})$. To distinguish abstract substitutions from concrete substitutions abstract substitutions will be represented by greek letters with a hat, the same greek letter without the hat representing a concrete substitution approximated by the abstract one. Sets will be denoted with square brackets in abstract substitutions (to distinguish them and because of the mnemonic connotations since they are to be represented in Prolog in the analyzer), and with braces in concrete substitutions (as usual). Following the standard notation, we will name the abstraction function α and the concretization function γ.

Informally, an abstract substitution in the sharing domain is a set of sets of program variables (a set of sharing sets), where sharing sets represent all *possible* sharing patterns among the program variables.

For example, given the following concrete substitution θ, $\widehat{\theta}_{\mathrm{SH}}$ is its abstraction in the sharing domain:

$$\begin{array}{lcl} \theta & = & \{\mathrm{X/f(1,a)},\ \mathrm{Y/A},\ \mathrm{Z/f(A,C,t(B))},\ \mathrm{W/[B,C]},\ \mathrm{V/D}\} \\ \widehat{\theta}_{\mathrm{SH}} & = & [[\mathrm{YZ}]\,[\mathrm{ZW}]\,[\mathrm{V}]] \end{array}$$

On the other hand, given the following sharing abstract substitution $\widehat{\theta}_{\mathrm{SH}}$, the θ_i are concrete substitutions approximated by it. The last column in the following represents the sharing sets "active" in each concrete substitution –we say that a set $L \in \widehat{\theta}_{\mathrm{SH}}$, where $\widehat{\theta}_{\mathrm{SH}}$ is a sharing abstract substitution, is **active** in a concrete substitution $\theta \in \gamma(\widehat{\theta}_{\mathrm{SH}})$ iff L is in the abstraction of θ:

$$\begin{array}{lcll} \widehat{\theta}_{\mathrm{SH}} & = & [[\mathrm{X}]\,[\mathrm{YZ}]\,[\mathrm{ZW}]] & \\ \theta_1 & = & \{\mathrm{X/A},\ \mathrm{Y/f(B,1)},\ \mathrm{Z/B},\ \mathrm{W/foo}\} & [[\mathrm{X}]\,[\mathrm{YZ}]] \\ \theta_2 & = & \{\mathrm{X/[\,]},\ \mathrm{Y/A},\ \mathrm{Z/[B|A]},\ \mathrm{W/t(B)}\} & [[\mathrm{YZ}]\,[\mathrm{ZW}]] \\ \theta_3 & = & \{\mathrm{X/t(0,1)},\ \mathrm{Y/atom},\ \mathrm{Z/A},\ \mathrm{W/A}\} & [[\mathrm{ZW}]] \end{array}$$

The component described above is essentially the abstract domain of Jacobs and Langen [JL89].

An abstract substitution in the freeness domain is a set of program variables (those that are known to be free).

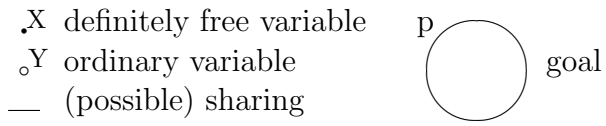


Figure 6.1: Types of objects in our pictorial representation.

It is important to point out that sharing information is not independent of freeness information, since known freeness of certain variables restricts the allowable combinations of sharing sets. The possible combinations of sharing sets a sharing+freeness abstract substitution $\widehat{\theta}$ represents are the subsets of the sharing component (the $S \in \wp(\widehat{\theta}_{\mathrm{SH}})$) that have one and only one sharing set including each variable in the freeness component ($\forall\, v \in \widehat{\theta}_{\mathrm{FR}}\ \dot{\exists}\, L \in S\ \ v \in L$).

The point above regarding sharing+freeness abstract substitutions, which is of great practical importance, may still be difficult to understand in the terms given so far. It is hoped that with the aid of the pictorial representation to be presented in the following section these issues will be greatly clarified.

6.2.1 Pictorial Representation of Substitutions

We have chosen a pictorial representation of substitutions in order to make it easier to understand abstract substitutions in the sharing+freeness domain and to follow the discussions and examples throughout the text. The idea of the pictures is to make the large amount of information contained in these abstract substitutions more explicit. Figure 6.1 illustrates the different types of objects used in this representation.

As mentioned before, an abstract sharing+freeness substitution is a compact representation of a finite number of possible sharing+freeness situations in the concrete domain. To reflect this a given sharing+freeness abstract substitution can be represented with a finite number of figures, each figure having the same freeness information (which is definite) but representing the different alternative coverings of free variables by the sharing sets.

As the sharing+freeness abstract substitutions give information in terms of program variables, only these variables appear in the figures. Variables in the

freeness component are represented by dots, the rest by circles. The sharing patterns are represented by lines connecting all the variables of the corresponding sharing set. If the sharing set has only one variable, and this variable is not in the freeness component, it is represented as a short line coming out of the variable. Note that all sharing sets that contain no free variables (including those just mentioned involving only one variable), may be either active or inactive in a concrete substitution, since they represent only "possible" sharing. Rather than having multiple figures for all the cases involved, all such sets are represented explicitly by lines in a given picture, under the assumption that those lines may or may not be present. Note also that lines connecting free variables on the other hand represent necessarily active sharing sets and cannot be removed.

The number of lines coming out of a circle represents the number of sharing sets containing the corresponding variable. However, multiple lines coming out of dots (free variables) all correspond to the same sharing set, since free variables must be in one and only one active sharing set (this is done to simplify the drawings). If no line comes out of a given dot this represents a sharing set containing only this variable. Note that a circle connected to one or more lines can in fact represent a free variable, since the freeness component names only the variables that are definitely known to be free. On the other hand, an isolated circle represents always a ground variable, since the variable is not a member of any sharing set. The resulting pictures are hypergraphs, since the edges connect an arbitrary number of vertices.

A goal is represented like a set in a Venn Diagram, the variables in the set being the goal variables. When we represent two goals, the first one (in the Prolog textual order) is to the left and the second one to the right, and the variables present in both goals are placed in the "intersection".

Figure 6.2 shows several examples representing in one or more pictures abstract substitutions. The number of pictures corresponds to the number of alternative coverings of free variables by the sharing sets.

In the first example, the abstract substitution is represented with only one picture, as there is only one possible covering of the free variables by the sharing sets. W is ground, X and Y are free, and Z is a term that contains X.

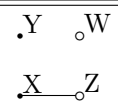
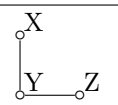
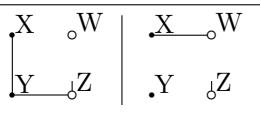
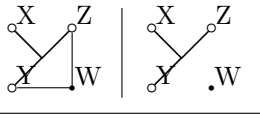
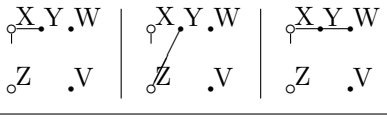
Variables / Abstract substitution	Representation
{X, Y, Z, W} / ([[Y] [XZ]], [XY])	
{X, Y, Z} / ([[XY] [YZ]], [ ])	
{X, Y, Z, W} / ([[XYZ] [XW] [Y] [Z]], [XY])	
{X, Y, Z, W} / ([[XYZ] [YZW] [W]], [W])	
{X, Y, Z, W, V} / ([[X] [XY] [YZ] [W] [XYW] [V]], [YWV])	

Figure 6.2: Examples of representation of abstract substitutions

The second example, since there are no free variables, also represents an abstract substitution with only one picture. The two sharing sets represented are thereby optional, so in fact we have four possibilities depending on which of the two are active.

In the third example we have two pictures: either the sharing set [XYZ] or the two sharing sets [XW] and [Y] are active, since these are the two possible coverings of the free variables X and Y. In the first picture W is ground, whereas Z is not ground and contains the variables X and Y, that are aliased, and may contain more variables. In the second picture, W is a term that contains X, Y is free and independent from the other variables, Z is also independent from the other variables (and it may be ground since its sharing set is optional).

The fourth example shows two pictures, depending on whether [YZW] or [W] is active. In both pictures there is an optional sharing among X, Y and Z. In the first picture, Y and Z also share the variable W.

Finally, the fifth example shows three pictures. In the first, the covering of the free variables Y, W and V is performed by the sharing sets [XY], [W] and [V]; in

the second, by [X], [YZ], [W] and [V]; and in the third, by [XYW] and [V]. There are no more combinations that cover all the free variables without overlapping. In the first picture, V and W are free variables and independent, Z is ground, and X contains the free variable Y and possibly others. In the second picture, V and W are also free variables and independent, Z contains the free variable Y, and X is also independent (as in one of the previous cases, we do not know if it is free, ground, or otherwise). In the third picture, V is free and independent, Z is ground, and X contains Y and W, which are aliased, as well as possibly more variables.

6.3 Conditions for Non-Strict Independence with Respect to the Information from Sharing+ Freeness Analysis

The definition of non-strict independence that we use (for two goals) is given in terms of the substitutions before and after the execution of the goal to the left, i.e. of its call and answer substitutions. Correspondingly, in the abstract domain we will consider that goal's abstract call and abstract answer substitutions.

Before stating the conditions it is important to understand in which form a goal can transform its abstract call substitution into its abstract answer substitution. Regarding the freeness component, what it can do is eliminate variables from the component (by instantiating them). Regarding the sharing component, it can eliminate sharing sets (by instantiating its variables to ground terms) or create more by union of the present sharing sets (by unifying variables from these sharing sets). If a variable in a sharing set is further instantiated but not made ground, the sharing set remains unchanged. Note also that when a sharing set contains one or more free variables, if it is active, there is a single shared run-time variable corresponding to these program variables. Recall also that two sharing sets containing the same free variable cannot be active at the same time. The following two sections deal with the cases that arise depending on whether purity

information is considered or not.

6.3.1 Conditions Disregarding Purity of Goals

As mentioned in the introduction we will consider the parallelization of pairs of goals. First let us state the conditions without purity information.

Let p and q be two goals, where q is to the right of p. Also let $\widehat{\beta}$ and $\widehat{\psi}$ be the abstract call and answer substitutions for p. So the situation is $\{\widehat{\beta}\}$ p $\{\widehat{\psi}\}$ ... q. We define the sets:

$$\begin{aligned} &\mathrm{S(p)} = \{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var(p)} \neq \emptyset\} \\ &\widehat{\mathrm{SH}} \ = \mathrm{S(p)} \cap \mathrm{S(q)} = \{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var(p)} \neq \emptyset \ \wedge \ L \cap \mathrm{var(q)} \neq \emptyset\} \end{aligned}$$

That is, S(p) is the set of all sharing sets of $\widehat{\beta}_{\mathrm{SH}}$ that contain a variable from p, and $\widehat{\mathrm{SH}}$ is the set of all sharing sets of $\widehat{\beta}_{\mathrm{SH}}$ that contain variables from p and from q (that is, the sharing sets that, if are active, contain run-time shared variables).

The following are our conditions for non-strict independence between p and q:

C1 $\quad \forall\, L \in \widehat{\mathrm{SH}} \ \ L \cap \widehat{\psi}_{\mathrm{FR}} \neq \emptyset$

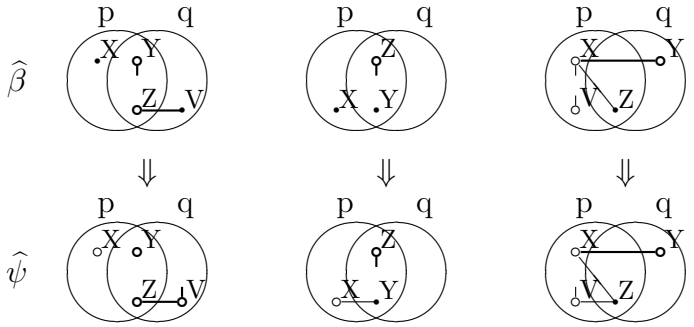
C2 $\quad \neg\, (\exists\, N_1 ... N_k \in \mathrm{S(p)} \ \exists\, L \in \widehat{\psi}_{\mathrm{SH}} \ \ (L = \bigcup_{i=1}^{k} N_i) \ \wedge \ N_1, N_2 \in \widehat{\mathrm{SH}}$
$\qquad\qquad \wedge \forall\, i, j \ \ 1 \leq i < j \leq k \ \ N_i \cap N_j \cap \widehat{\beta}_{\mathrm{FR}} = \emptyset)$

Condition C1 deals with preserving freeness of shared variables.[1] By checking that all sharing sets of $\widehat{\mathrm{SH}}$ have a free variable in the abstract answer substitution $\widehat{\psi}$, it is ensured that no run-time shared variable is further instantiated. Note that if there is more than one free variable in a sharing set, and one of them remains free, the other necessarily remain also free, since all coincide at run-time when the set is active.

Condition C2 is needed to preserve independence of shared variables: $N_1 ... N_k$ are sharing sets that p can unite (thus they come from S(p)) to derive the sharing set L of the abstract answer substitution, and at least two sharing sets contain shared variables (we can always name them N_1 and N_2). Furthermore, no two sharing sets N_i, N_j contain the same free variable, since otherwise they cannot be both active in one concrete substitution, making the union impossible. This

[1] We would like to thank M. Bruynooghe for suggesting improvements to our first C1.

Three examples where C1 fails: run-time shared variables can be further instantiated



Three examples where C2 fails: run-time shared variables can alias each other

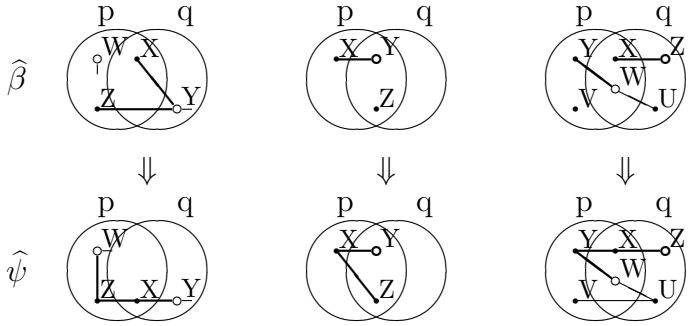


Figure 6.3: Situations where the conditions do not hold, and thus the goals are possibly not NSI

also ensures, given that the first condition is met, that N_1 and N_2 have different shared variables. Intuitively, it can be seen that if C1 and ¬C2 holds, p can possibly bind the two independent shared variables.

Figure 6.3 shows some situations where either C1 or C2 do not hold. The sharings drawn with thick lines are the faulty ones, i.e. for C1, the Ls that have no variables in $\widehat{\psi}_{\mathrm{FR}}$, and for C2, N_1 and N_2 in $\widehat{\beta}$ and L in $\widehat{\psi}$.

6.3.2 Conditions Considering Purity Information

This section relies on the assumption that we have purity information, and also that we can compute the least upper bound of the abstractions of the partial answers of a goal.

If we examine the conditions stated in the previous section, we can see that only the behavior of the first goal p is considered. But if we know that q is pure, the conditions can be further relaxed.

Let us now define our conditions for non-strict independence when q is pure. Let $\widehat{\beta}$ and $\widehat{\psi}$ be the abstract call and answer substitutions for p, and let $\widehat{\phi}$ be the least upper bound of the abstractions of the partial answers of q when called with $\widehat{\beta}$ as the abstract call substitution. The following are our conditions for non-strict independence between p and q in this case:

$$\text{C1}' \quad \forall\, L \in \widehat{\text{SH}} \;\; L \cap \widehat{\psi}_{\text{FR}} \neq \emptyset \;\vee\; L \cap \widehat{\phi}_{\text{FR}} \neq \emptyset$$

$$\text{C2}' \quad \neg\,(\exists\, N_1 ... N_k \in \text{S(p)} \;\; \exists\, L \in \widehat{\psi}_{\text{SH}} \;\; (L = \textstyle\bigcup_{i=1}^{k} N_i) \;\wedge\; N_1, N_2 \in \widehat{\text{SH}}$$
$$\wedge\, \forall\, i,j \; 1 \leq i < j \leq k \; N_i \cap N_j \cap \widehat{\beta}_{\text{FR}} = \emptyset \;\wedge\; N_1 \cap \widehat{\phi}_{\text{FR}} = N_2 \cap \widehat{\phi}_{\text{FR}} = \emptyset)$$

Condition C1′ differs from C1 in that it allows p to further instantiate a shared variable, provided that this variable is not touched by q (q does not further instantiate it under $\widehat{\beta}$, so it does not mind whether the variable is free or not). Condition C2′ now says that the union of N_1 and N_2 is legal if either of the shared variables in them is not touched by q (note that only if q further instantiates the two variables it can possibly be affected by these bindings).

6.4 Run-Time Checks for Non-Strict Independence

In the previous sections we have proposed conditions to be checked at compile-time in order to decide whether to run two goals in parallel. However, even if these conditions do not hold, we may yet try to execute them in parallel, provided that some a priori run-time checks succeed.

The purpose of the run-time checks is to ensure that goals will not be run in parallel when there is no non-strict independence, while allowing parallel execution in as many cases as possible when non-strict independence is present. This fact will be determined from the combination of compile-time analysis and the success of the run-time checks previous to the execution of the goals. Note that this is meaningful because the sharing component represents possible, not

definite sharing sets. Thus we may want to generate a test that determines in which particular case we are, when at least one case allows parallelization, but the others may not.

Due to the complexity of the special case when the second goal is pure, here we will only consider the general case. Let us analyze what to do when each of the conditions of the general case is violated.

6.4.1 Condition C1 Violated

$$[\exists\, L \in \widehat{\mathrm{SH}}\ \ L \cap \hat{\psi}_{\mathrm{FR}} = \emptyset]$$

In this case we need run-time checks to ensure that the sharing sets $L \in \widehat{\mathrm{SH}}$ not obeying C1 ("illegal sharing sets") are not active. But, if the rest of the sharing sets in $\widehat{\beta}_{\mathrm{SH}}$ cannot cover all the free variables of $\widehat{\beta}_{\mathrm{FR}}$ without overlapping, it is impossible for all the illegal sharing sets to be inactive, so the goals are definitely not NSI. Otherwise, we must try to generate the least number of checks which overrides every illegal sharing set without affecting the legal ones (to preserve parallelism in valid situations).

There are several checks that can be used to prevent the illegal sharing sets from being active. In increasing order of cost, as they shoud be tried, they are:

- If there exists a variable X such that it appears only in illegal sharing sets, then the check `ground(X)` ("X is bound to a ground term") overrides those illegal sharing sets containing X.

- Suppose that there exists a variable X and a list $\mathcal{F}$ of free variables from $\widehat{\beta}_{\mathrm{FR}}$ such that, for the sharing sets containing X, illegal ones do not contain variables of $\mathcal{F}$, and legal ones contain at least one. Then the check `allvars(X,`$\mathcal{F}$`)` ("every variable in X is in the list $\mathcal{F}$") overrides all the illegal sharing sets containing X, and only those. In fact, the check `ground(X)` above is a special case of this when $\mathcal{F} = [\,]$. Note that if $\mathrm{X} \in \mathrm{var(p)} \cap \mathrm{var(q)}$ then we always are in this case, since all sharing sets containing X are in $\widehat{\mathrm{SH}}$, so the ones that are legal contain free variables that remain free after executing p, and those that are illegal do not.

- If there exist two variables X and Y such that all sharing sets containing both are illegal, then the check `indep(X,Y)` ("X and Y do not share variables") overrides those illegal sharing sets.

- For each of the remaining illegal sharing sets, we choose two variables X and Y which are members of it, such that X $\in$ var(p) and Y $\in$ var(q). Note that the sharing sets in $\widehat{\mathrm{SH}}$ have a variable in both var(p) and var(q) or have one variable in var(p) and another variable in var(q). And, since the illegal sharing sets are in $\widehat{\mathrm{SH}}$, if they cannot be overridden by the `allvars/2` check then they are in this case. Furthermore, the legal sharing sets that contain both X and Y are for this very reason also in $\widehat{\mathrm{SH}}$, so they have free variables that remain free after executing p. Let $\mathcal{F}$ be the set of these free variables. Then the check `sharedvars(X,Y,`$\mathcal{F}$`)` ("every variable shared by X and Y is in the list of variables $\mathcal{F}$") overrides all the illegal sharing sets containing X and Y, and only those. Also, the check `indep(X,Y)` is a special case of this when $\mathcal{F} = [\,]$.

Figure 6.4 gives examples showing how the checks restrict the possible sharing sets.

6.4.2 Condition C2 Violated

$$[\exists\, N_1 ... N_k \in \mathrm{S(p)}\ \exists\, L \in \widehat{\psi}_{\mathrm{SH}}\ \ (L = \cup_{i=1}^{k} N_i)$$
$$\wedge\ N_1, N_2 \in \widehat{\mathrm{SH}} \wedge \forall\, i, j\ \ 1 \leq i < j \leq k\ \ \ N_i \cap N_j \cap \widehat{\beta}_{\mathrm{FR}} = \emptyset]$$

Once the checks for C1 have been computed, and taking into account only the sharing sets not rejected by these checks, the second condition is treated.

Now, for each L in the above formula, we compute the different groups of $N_1 ... N_k$ that p can unite to give the sharing set L, without taking into account the number of sharing sets N_i that are in $\widehat{\mathrm{SH}}$. The groups that have more than one sharing set in $\widehat{\mathrm{SH}}$ are the "illegal" groups. If there are no legal groups, and L is necessarily active in $\widehat{\psi}$ (this is so if L contains free variables that do not appear in other sharing sets of $\widehat{\psi}_{\mathrm{SH}}$), then necessarily p binds shared variables, so the goals are definitely not NSI. Otherwise, we need checks as for the first

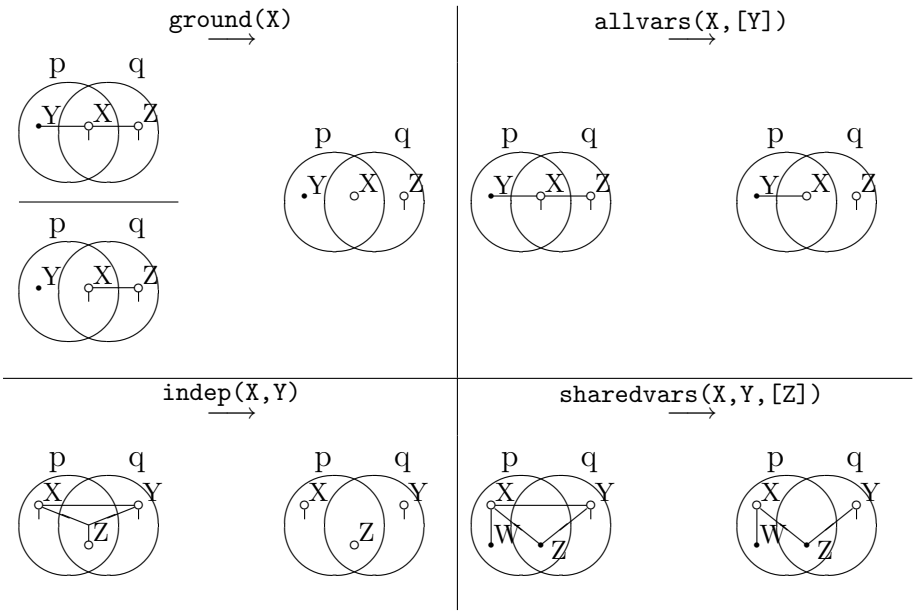


Figure 6.4: Example applications of the four checks

condition, now ensuring that at least one sharing set of each illegal group is not active, without affecting, if possible, sharing sets of the legal groups.

For example, suppose we are trying to parallelize the goal "`p(X,Y,Z,U), q(X,Y,W,V)`" and the abstract call and answer substitutions for `p(X,Y,Z,U)` are $\widehat{\beta}$ = ([[X] [XZ] [Y] [Z] [ZW] [U] [UW] [WV]], [YUV]) and $\widehat{\psi}$ = [[X] [YU] [UW] [WV]], [YV]). We have that $\widehat{\mathrm{SH}}$ = [[X] [XZ] [Y] [ZW] [UW]], and the illegal sharing sets for the first condition are [X], [XZ], [ZW] and [UW]. The check `ground(X)` overrides the first two, and the check `allvars(W,[V])` the last two (without affecting other sharing sets). The second condition holds, so we are ready to parallelize the two goals, the result being:

```
( ground(X), allvars(W,[V]) -> p(X,Y,Z,U) & q(X,Y,W,V)
                             ;  p(X,Y,Z,U),  q(X,Y,W,V) )
```

where "`A -> B ; C`" is the prolog *if-then-else* and "`&`" is the (unconditional) parallel operator. Figure 6.5 shows the restriction of the possible sharing sets made by the checks, and how this restriction make the goals non-strict independent.

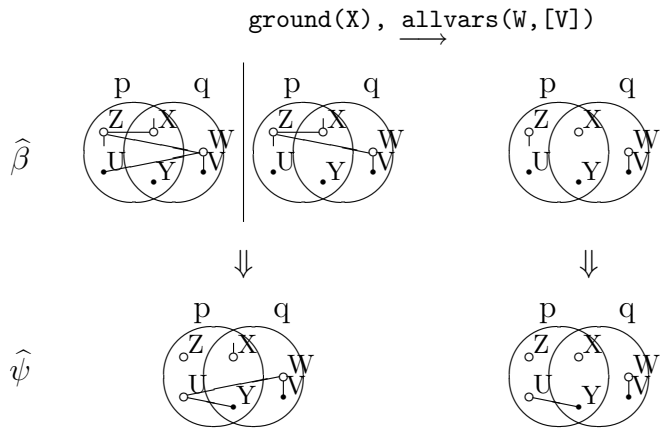


Figure 6.5: Restriction of the possible sharing sets by the checks

6.4.3 Run-Time Checks and Strict Independence

It is worth pointing out that if no information is obtained from the analysis (or no analysis is performed), and thus the abstract substitutions are $\top$, the run-time checks computed by the method presented here exactly correspond to the conditions traditionally generated for strict independence (shared program variables ground, other program variables independent, see e.g. [HR89] for more information). This is correct, since in absence of analysis information only strict independence is possible, and shows that the method presented is a strict generalization of the techniques which have been previously proposed for the detection of strict independence.

It can be easily shown how the tests reduce to those for strict independence: since there are no free variables in the abstract substitutions, every sharing set of $\widehat{\mathrm{SH}}$ is illegal with respect to the first condition. These sharing sets contain a shared program variable (and are overridden by a `ground/1` check on each) or program variables of both goals (overridden by an `indep/2` check on every pair).

For example, if we have a goal "`p(X,Y) & q(Y,Z)`" with $\widehat{\beta} = ([[\mathrm{X}]\,[\mathrm{Y}]\,[\mathrm{Z}]\,[\mathrm{XY}]\,[\mathrm{XZ}]\,[\mathrm{YZ}]\,[\mathrm{XYZ}]], [\,])$ (i.e. $\top$, equivalent to no information), then we have $\widehat{\mathrm{SH}} = [[\mathrm{Y}]\,[\mathrm{XY}]\,[\mathrm{XZ}]\,[\mathrm{YZ}]\,[\mathrm{XYZ}]]$. The check `ground(Y)` overrides all the illegal sharing sets except [XZ], which is overridden in turn by the check `indep(X,Z)`. Figure 6.6 depicts how the checks restrict the possible sharing sets.

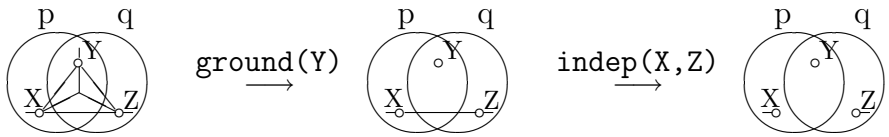


Figure 6.6: Restriction of the possible sharing sets performed by the checks

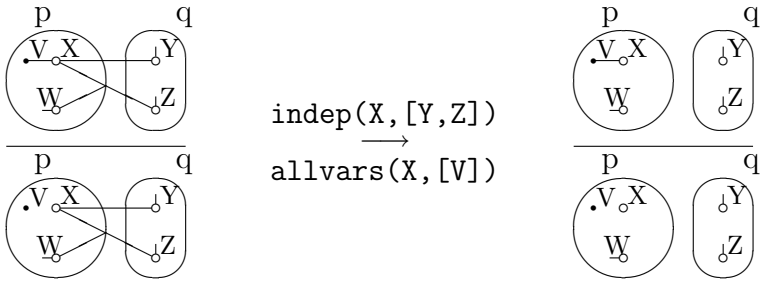


Figure 6.7: Restriction of the possible sharing sets performed by either check

Also, in the presence of sharing+freeness abstract information, the tests made with this method are equivalent or better than the traditional tests simplified with this information, even if only strict independence is present. As an example, let us study the case of the goal "`p(X,V,W) & q(Y,Z)`" with $\widehat{\beta}$ = ([[V] [VX] [Y] [XY] [Z] [XZW] [W]], [V]) (see Fig. 6.7). The traditional test for strict independence would be `indep(V,Y), indep(X,Y), indep(W,Y), indep(V,Z), indep(X,Z), indep(W,Z)` (perhaps written as `indep([V,X,W],[Y,Z])`). With the analysis information above, is simple to deduce that the tests `indep(V,Y)`, `indep(W,Y)` and `indep(V,Z)` are not needed. Not so obvious is to deduce that one of the test `indep(X,Z)` or `indep(W,Z)` can also be eliminated. So, in this latter case we come up with the simplified test `indep(X,Y), indep(X,Z)`, or `indep(X,[Y,Z])`.

On the other hand, applying the method presented here, we have that $\widehat{\mathrm{SH}}$ = [[XY] [XZW]]. Both sharing sets are illegal, since they do not contain free variables. The legal sharing set that contains X contains also the free variable V, and the two illegal sharing sets contain X but not this free variable, so `allvars(X,[V])` ensures that the illegal sharing sets are inactive, without affecting any legal sharing set. This test is clearly cheaper than the other, since it only needs to traverse X, whereas the other needs to traverse also Y and Z (in the worst case).

6.5 Parallelization under NSI: the URLP algorithm

Using the conditions stated in previous sections, an automatic parallelizer can insert into the program clauses parallel operators which allow to exploit and-parallelism (this process is called *annotation*). The existing annotation algorithms for automatic parallelization using strict independence, the CDG, UDG, and MEL algorithms [MH90], were not well suited due to several reasons for parallelization using non-strict independence. The MEL algorithm is a simplistic algorithm which cannot create nested parallel expressions, thus disallowing the complete exploitation of the parallelism in the clause. The CDG and UDG algorithms may reorder the goals found to be independent, which is valid for strict independence but not for non-strict independence, because that last notion is not symmetric.

Thus, we developed a new algorithm for annotation suited to the special characteristics of non-strict independence. We decided to make an annotator which will try to maximize the amount of parallelism created, but without including run-time checks. Therefore, it corresponds in a way to the UDG algorithm used for strict independence. Nevertheless, we think that the algorithm, due to its simplicity, can be easily adapted to deal with such checks. The new algorithm, named URLP (Unconditional Recursive Linear Parallelizer), shows a similar degree of parallelization as UDG, but preserving always the original order of the goals in the clause. URLP does not use the technique of dependency graphs as UDG, because the graphs lose the information about goal order.

The URLP algorithm is heuristic, starting with the sequence of goals of the body of the clause, and applying from left to right the rewriting rules of Figure 6.8, in the order they are enumerarted, until no change can be done.

Rule	Pattern	Condition	New Pattern
1	...A, B...	indep(A,B)	...A & B...
2	...PA, B...	(1)	...IA & (DA, B)...
3	...PA, PB...	(2)	...(PA, DB) & IB...

Where A and B represent single goals and PA, PB, DA, DB, IA and IB represent single goals or parallel expressions.
(1) IA (DA) is the parallel expression formed with the elements of PA from which B is independent (dependent). After the rule is applied, "DA, B" is parallelized recursively. The rule is only applied if IA is not empty.
(2) IB (DB) is the parallel expression formed with the elements of PB which are independent (dependent) from PA. The rule is only applied if IB is not empty.

Figure 6.8: Rewriting rules of the URLP algorithm.

The first rule is obvious: two contiguous goals which are independent can be executed in parallel. Second rule states that when we have a parallel expression followed by a single goal, then that goal can be executed in parallel with the members of the parallel expression from which it is independent, but must wait until the end of the execution of all goals from which it depends. Third rule is the reciprocal to the second: when we have a parallel expression (or a single goal) followed by a parallel expression, then the members of the last parallel expression which are independent from the first expression can be executed in parallel with it, and the rest must wait until the end of its execution.

Since at each step a simple rule is applied, it can be easily shown that the parallelization is correct. To see if it is optimal, however, we would need in the general case the execution times of the goals.

As an example, consider the parallelization of the body of a clause, which we will schematically represent as "a, b, c, d, e, f, g", where the dependencies given by the conditions of Section 6.3 are dep(a, b), dep(b, e), dep(b, f), dep(c, e), dep(d, g), and dep(f, g) (we denote by dep(X, Y) that Y is dependent from X). The steps followed by the algorithm are shown below:

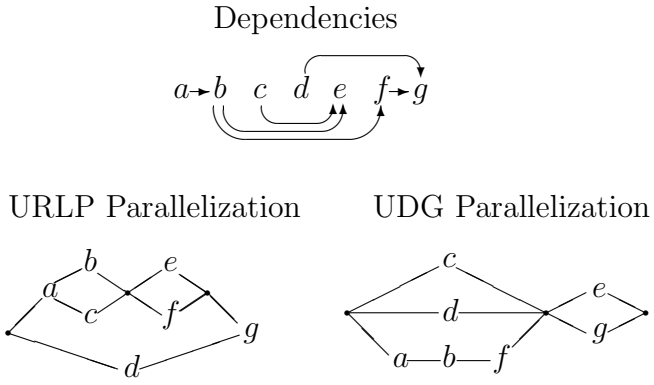


Figure 6.9: Example parallelizations using URLP and UDG.

Step	**Expression**	**Apply Rule**
0	a, b, c, d, e, f, g	1 with b, c
1	$a, b\,\&c, d, e, f, g$	2 with $b\,\&c, d$
2	$a, b\,\&c\,\&d, e, f, g$	2 with $b\,\&c\,\&d, e$
3	$a, d\,\&(b\,\&c, e), f, g$	2 with $d\,\&(b\,\&c, e), f$
4	$a, d\,\&(b\,\&c, e, f), g$	1 with e, f
5	$a, d\,\&(b\,\&c, e\,\&f), g$	3 with $a, d\,\&(b\,\&c, e\,\&f)$
6	$(a, b\,\&c, e\,\&f)\,\&d, g$	

The parallelization provided by UDG is "$c\,\&d\,\&(a, b, f), e\,\&g$", reversing the relative order of execution of the goals e and f, which is exactly what must be avoided for non-strict independence. Figure 6.9 illustrates in pictures the results.

6.6 Renaming and Substituting Variables

In order to prevent partial answers of a branch that ultimately fail from pruning the search space of other goals, parallel goals are in principle run in independent environments (see [HR89, HR90]). The standard solution for this problem is a run-time transformation of the goals to be executed in parallel. This transformation involves eliminating any shared variable among parallel goals by renaming or substituting all its occurrences so that no two occurrences in different goals remain the same, and adding some unification goals after the parallel conjunction to reestablish the lost links. Here we will attempt to perform this operation at

least in part at compile-time, by defining a new predicate, in a more efficient way than making `copy_term`'s of every goal and after the parallel conjunction unifying the goals and the copied versions. Note that a mere renaming of variables at compile-time is not sufficient in general: we can have terms with shared variables inside.

The transformation procedure looks at each sharing set of SH, computing the necessary renamings or substitutions for each goal with respect to this sharing set:

- If the sharing set does not contain variables of the goal, nothing must be done.
- If the sharing set contains only one variable of the goal, and the variable is free, we need a renaming of this variable.
- Else we need to substitute a free variable of the sharing set for a new one in each variable of the goal present in the sharing set (if the goal have free variables in the sharing set, one of them is used to make the substitution). This can be done with the following predicate:

  `subst_vars(`$[\mathtt{X}_1,\ldots,\mathtt{X}_n]$`,`$[\mathtt{X}'_1,\ldots,\mathtt{X}'_n]$`,Z,Z') :-`
  $\{\mathtt{Z}'$ *is a term equal to* `Z` *but with the variables* $\mathtt{X}_1,\ldots,\mathtt{X}_n$ *substituted for the variables* $\mathtt{X}'_1,\ldots,\mathtt{X}'_n$ *respectively*$\}$.

Then, we discard the computed transformations for the goal in which they are most expensive, (substitutions are more expensive than renamings, and the more substitutions, the more expensive).

Of course for all the variables renamed or substituted must be included unification goals (or "back-binding" goals) after the parallel conjunction.

As an example, consider the parallel expression `p(U,V,W) & q(V,W,X,Y) & r(W,Z)`, with the call abstract substitution $\widehat{\beta} = ([[\mathrm{UV}]\,[\mathrm{VWX}]\,[\mathrm{XY}]\,[\mathrm{Z}]], [\mathrm{UW}])$. Lets compute for each sharing set of SH = [[UV] [VWX]] and for each subgoal the transformation needed. For conciseness, we represent a renaming of a variable V as "ren(V)" and a substitution of a variable V inside W as "sv(V,W)".

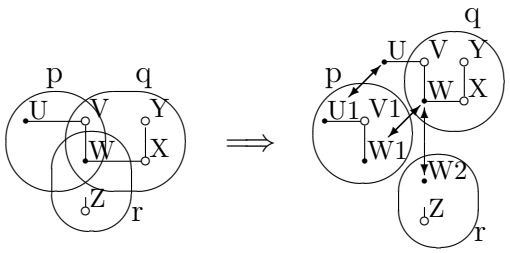


Figure 6.10: Representation of the effect of variable substitution in a parallel expression.

	p(U,V,W)	q(V,W,X,Y)	r(W,Z)
[UV]	sv(U,V)	sv(U,V)	$\emptyset$
[VWX]	sv(W,V)	sv(W,V), sv(W,X)	ren(W)

With the first sharing set, we discard the transformations for subgoal q (we could instead discard the transformations for p, since are equal). With the second sharing set, the transformations for subgoal q must be discarded. As remain two substitutions on the same variable of the same subgoal (sv(U,V) and sv(W,V) on p) they can be joined. So the parallel expression is transformed to:

```
subst_var([U,W],[U1,W1],V,V1),
p(U1,V1,W1) & q(V,W,X,Y) & r(W2,Z),
U=U1, W=W1, W=W2
```

Figure 6.10 illustrates in pictures the transformation done, the double arrows show the bindings of the back-binding goals.

6.7 Example Parallelization of a Program

As an example in this section we will show how to apply the proposed methods to a concrete program (quicksort using difference lists) in order to exploit the non-strictly independent and-parallelism it contains. Although the program is small, we think that it is of sufficient entity to show the potential of the proposal, and at

the same time it is small enough to allow presenting the complete parallelization process.[2]

The quicksort program we will use follows, with the abstract substitutions obtained by the analyzer annotated at each point of the program:

```
qsort(I,O) :-                          %[[O]],[O]
        qsort(I,O,[]).                 %[],[]
qsort([],L,L).
qsort([X|Xs],L,L2) :-                  %[[L],[L2],[Left],[Right],[L1]],
                                       % [L,Left,Right,L1]
        partition(Xs,X,Left,Right),    %[[L],[L2],[L1]],[L,L1]
        qsort(Left,L,[X|L1]),          %[[L,L1],[L2]],[L1]
        qsort(Right,L1,L2).            %[[L,L2,L1]],[]
partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right) :-  %[[Right],[Left1]],[Right,Left1]
        E=<C,                          %[[Right],[Left1]],[Right,Left1]
        !,
        partition(R,C,Left1,Right).    %[],[]
partition([E|R],C,Left,[E|Right1]) :-  %[[Left],[Right1]],[Left,Right1]
        E>C,                           %[[Left],[Right1]],[Left,Right1]
        partition(R,C,Left,Right1).    %[],[]
```

We will concentrate on the parallelization of the `qsort/3` predicate. Firstly, we will analyze whether it is possible to parallelize the first and second goal of the recursive clause of `qsort/3`, so we have that p = `partition(Xs,X,Left,Right)` and q = `qsort(Left,L,[X|L1])`, and the abstract substitutions involved are $\widehat{\beta}$ = ([[L] [L2] [Left] [Right] [L1]], [L Left Right L1]) and $\widehat{\psi}$ = ([[L] [L2] [L1]], [L L1]). Then, we compute the set $\widehat{\mathrm{SH}}$ = [[Left]]. Condition C1 is not met, since "Left" is not in $\widehat{\psi}_{\mathrm{FR}}$, and furthermore this is a free variable that does not appear in another sharing set in $\widehat{\beta}_{\mathrm{SH}}$, so it is sure that the goals are not non-strictly independent.

[2]Note that this (or any other practical) example cannot be parallelized automatically using the compile-time techniques presented in [HR89, HR90, HR95]. The techniques that we propose are, to our knowledge, the only ones capable of parallelizing programs such as these without any run-time tests or any locks in the unification

In a similar manner it can be shown that the first and third goal of the clause are not non-strictly independent either.

Finally, let us try with the second and third goals in the same clause. Now p = `qsort(Left,L,[X|L1])`, q = `qsort(Right,L1,L2)`, $\widehat{\beta} = ([[\mathrm{L}]\,[\mathrm{L2}]\,[\mathrm{L1}]], [\mathrm{L}\,\mathrm{L1}])$ and $\widehat{\psi} = ([[\mathrm{L}\,\mathrm{L1}]\,[\mathrm{L2}]], [\mathrm{L1}])$. The shared sharing sets are $\widehat{\mathrm{SH}} = [[\mathrm{L1}]]$. But now the conditions hold: $\mathrm{L1} \in \widehat{\psi}_{\mathrm{FR}}$ and no sharing sets meet $\neg$C2. Thus in this case we have non-strict independence, and no run-time checks are needed (note also that the goals are not strictly independent, since they share the free variable "L1").

The last step is to see whether we need to rename or substitute any variable in the goals. In either the two goals we only need to rename the variable "L1", so the predicate `qsort/3` would be left as:

```
qsort([],L,L).
qsort([X|Xs],L,L2):-
        partition(Xs,X,Left,Right),
        qsort(Left,L,[X|L1]) & qsort(Right,L1_prime,L2),
        L1=L1_prime.
```

6.8 Some Experimental Results

We have measured the speedups obtained using the techniques presented for a number of programs that have NSI but no SI. The programs were automatically parallelized with our parallelizing compiler. This compiler is a modification of the &-Prolog compiler, which was originally designed to exploit strict independence. New annotator and dependency analysis modules were added which implement the techniques presented so far. Only unconditional parallelism was used (i.e. no run-time checks were generated). The programs where then executed using from 1 to 10 processors on a Sequent Symmetry on the &-Prolog system [HG90], an efficient parallel implementation of full Prolog that can exploit independent and-parallelism among non-deterministic goals.

The results are given in Table 6.1. Speedups are relative to the sequential execution on one processor. The performance of &-Prolog on one processor,

Table 6.1: Speedups of several programs with NSI

Bench	# of processors									
	1	2	3	4	5	6	7	8	9	10
array2list	0.78	1.54	2.34	3.09	3.82	4.64	5.41	5.90	6.50	7.22
flatten	0.54	1.07	1.61	2.07	2.52	3.05	3.62	4.14	4.46	4.83
hanoi_dl	0.56	1.13	1.68	2.25	2.73	3.23	3.70	4.34	4.84	5.25
qsort	0.91	1.65	2.20	2.53	2.75	2.86	3.00	3.14	3.30	3.33
sparse	0.99	1.92	2.79	3.68	4.50	5.06	5.78	6.75	8.10	8.26

even when running parallelized programs, is about 95% of the performance of the sequential system (SICStus Prolog [Car88]) on which it is based, itself one of the most popular Prolog systems. Thus, we argue, the speedups obtained are meaningful and useful, and we believe that the results obtained are quite encouraging. The differences between the sequential execution and the execution of the parallelized program on one processor is most due to the environment separation issue, mentioned in Section 6.6.

A description of the programs used follows: the **array2list** program is a subroutine of the SICStus prolog "arrays.pl" library. It translates an extendable array into a list of index-element pairs. The input array used to measure the speedups had 2048 elements. The **flatten** program is a subroutine that flattens a list of lists of any complexity into a plain list. The speedups were measured with an input list of 987 elements with recursive "depth" of seven. The **hanoi_dl** program is the well-known benchmark that computes the solution of the towers of Hanoi problem, but programmed with difference lists. It was run for 13 rings. The **qsort** program is the sorting algorithm quicksort using difference lists. The speedups were measured sorting a list of 300 elements. Finally, the **sparse** program is a subroutine that transforms a binary matrix (in the form of list of lists) into a list of coordinates of the positive elements, i.e. a sparse representation. It was run with an input matrix of 32×128 elements, with 256 positive elements.

6.9 Towards an Improved Analysis for Non-Strict Independence

We have so far presented a method for detecting non-strict independence from the information provided by a straightforward analysis based on the Sharing+ Freeness domain. In light of this method we were able to understand more clearly in what way the analysis itself can be improved to increment the amount of parallelism that can be exploited automatically.

A first way to do this is by combining Sharing+Freeness with other analyses that can improve the accuracy of the sharing and freeness information. A class of such analyses includes those that use linearity, such as the Asub domain [Søn86] (among others). In fact, this idea has already been incorporated in our system by using the techniques described in [CMB+93], and the results are used by the non-strict independence parallelizing compiler by simply focusing only on the Sharing+Freeness part. However, the improvement that can be obtained by these means is limited, as long as the sharing and freeness information is restricted to program variables.

A better improvement could be achieved by gaining access to information inside the terms to which program variables are bound at run-time, in order to check the possible instantiations of free variables inside these terms. To achieve this goal, sharing and freeness could be integrated (by using the techniques of [CMB+93] or [CLV94]) with other analyses, like the depth-k [ST84] domain, or, even better, "pattern" [CLV94] or any other recursive type analysis (see [BJ88]), at least for lists. This would allow dealing, for example, with lists of free variables. These alternatives will be studied in future work. However, note that the approach presented here is still valid directly or with very slight modifications for these more sophisticated types of analyses.

6.10 Chapter Conclusions

We have presented several techniques for achieving the compile-time detection of non-strict independence. The proposed techniques are based on the availability of certain information about run-time instantiations of program variables –sharing and freeness– for which compile-time technology is available, and for the inference of which new approaches are being currently proposed. We have also presented techniques for combined compile-time/run-time detection of NSI, proposing new kinds of run-time checks for this type of parallelism as well as the algorithms for implementing such checks. Experimental results showing the speedups found in some programs presenting NSI have also been given. The results were obtained by integrating the algorithm that detects non-strict independence (and others needed to exploit this kind of independence) in our parallelizing compiler, that already included a sharing+freeness analyzer, obtaining a complete compile-time parallelizer capable of detecting non-strict independence. We find that the results are encouraging.

We are also planning on looking, in the light of the techniques developed, to more sophisticated abstract analyses that may provide more accurate information, in order to increment the amount of parallelism exploitable automatically.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The main objective of this thesis was to design and develop a practical, efficient, standard logic programming system, suited both for programming in the small as well as programming in the large, highly extensible and reconfigurable, and above all supportive of global analysis, optimization, and verification. We feel that the resulting system, the next-generation logic programming system Ciao, fulfills quite satisfactorily the proposed goals.

We have shown that global analysis of unrestricted Logic Programs based on abstract interpretation is feasible, and optimizations based on global analysis can be guaranteed correct for programs which use the full power of the Prolog language. We have also introduced several types of program annotations that can be used to both increase the accuracy and efficiency of the analysis and to express its results. The proposed techniques offer different trade-offs between accuracy, analysis cost, and user involvement. We argue that the presented combination of known and novel techniques offers a comprehensive solution for the correct analysis and optimization of arbitrary programs using the full power of the language.

Regarding language design, we have presented a new module system for Prolog which achieves a number of fundamental design objectives such as being more

amenable to effective global analysis, allowing separate compilation and sensible creation of standalone executables, extensibility in features and in syntax, etc. It has been also shown in other work that this module system can be implemented easily and can be applied successfully in several modular program processing tasks, from compilation to debugging to automatic documentation generation. The proposed module system has been designed to stay as similar as possible to the module systems of the most popular Prolog implementations, but with a number of crucial changes that achieve the previously mentioned design objectives. We believe that it would not be difficult to incorporate these changes in the ISO-Prolog module standard or in the module systems of other Prolog systems. We have presented also a novel definition of higher-order for Logic Programming, which allows sensible global analysis of the program, module separation, and an efficient implementation. We argue that it also offers a clear semantics and that is of enough expressive power for most of the needs of Prolog programmers in this regard.

Regarding implementation issues, we have presented the Ciao code processing framework, which allows the development of program analysis and transformation tools in a way that is largely orthogonal to the details of module system design. We believe that, for any system which provides a number of code processing tools, having such a framework available and used by all the tools simplifies tremendously the task of making the behavior of the tools consistent. The framework, and the compiler, which is an instance of it, provide separate and global incremental compilation, automatically following module dependencies and recompiling obsolete object and interface files. We have found this to be a very useful feature for any medium- to large-size project. We have presented also some error and warning messages that the compiler can detect statically, which are in practice of great help in avoiding a number of otherwise time-consuming errors. The combination of incremental compilation and additional static error detection results in our experience in a much faster development cycle than with more traditional environments. We have presented the different types of executables that the compiler creates (including the novel concept of "active modules"), and through experimental data we have illustrated the different tradeoffs involved.

We have found that the tradeoffs are such that there are different uses and environments which make each of these executable types to be the most suitable, and have therefore decided to keep them all as compiler options. Regarding the overall compiler behavior, in our own (admittedly, perhaps biased) experience, we find the Ciao compiler addictive, despite the larger compilation times of the early versions, due, among other reasons, to the above mentioned faster development cycle and the much improved error detection capabilities.

Lastly, regarding global analysis applications, we have presented several techniques for achieving the compile-time detection of non-strict independence. The proposed techniques are based on the availability of certain information about run-time instantiations of program variables –sharing and freeness– for which compile-time technology is available, and for the inference of which new approaches are being currently proposed. We have also presented techniques for combined compile-time/run-time detection of NSI, proposing new kinds of run-time checks for this type of parallelism as well as the algorithms for implementing such checks. Experimental results showing the speedups found in some programs presenting NSI have also been given. The results were obtained by integrating the algorithm that detects non-strict independence (and others needed to exploit this kind of independence) in our parallelizing compiler, that already included a sharing+freeness analyzer, obtaining a complete compile-time parallelizer capable of detecting non-strict independence. We find that the results are encouraging.

7.2 Future Work

Although we are reasonably satisfied with the Ciao system resulting from this thesis, we acknowledge that some work needs still to be done to make the complete program development system (including the global analyzer, the specializer, the parallelizer, etc) a seamless and completely integrated tool. Also, the full application of global analysis to all aspects of the development cycle, for example to the debugging and verification process or to the compilation to native machine code via an intermediate language, is still ongoing work (see [HPBLG03] and [MCH03, MCH04]).

On the other hand, it should be noted that Ciao is designed precisely with the goal of being an evolving system, to be able to incorporate upcoming technologies and models by its modular redefinition, and thus it will never be finished! Some of the extensions we are currently either designing or already developing are finite domain constraints, declarative input/output, constraint handling rules, improved object-oriented programming support, other control rules including tabling, improved concurrency and distibuted execution operators, enhanced multi-atributed variables, etc.

Bibliography

[AK93] Hassan Aït-Kaci. An Introduction to LIFE – Programming with Logic, Inheritance, Functions and Equations. In *Proceedings of the 1993 International Symposium on Logic Programming*, 1993.

[AKPS92] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. In *Proc. Fifth Generation Computer Systems 1992*, pages 1012–1021, 1992.

[APH04] E. Albert, G. Puebla, and M. Hermenegildo. An Abstract Interpretation-based Approach to Mobile Code Safety. In *Proc. of Compiler Optimization meets Compiler Verification (COCV'04)*, Electronic Notes in Theoretical Computer Science 132(1), pages 113–129. Elsevier - North Holland, April 2004.

[BCC+97] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and on-line version of the manual available at `http://www.ciaohome.org`.

[BCC+01] F. Bueno, D. Cabeza, M. Carro, J. Correas, J. Gómez, M. Hermenegildo, P. López, G. Puebla, and C. Vaucheret. Agent Programming in Ciao Prolog. In *10 th. Portuguese Conference on Artificial Intelligence (EPIA)*, number 2258 in LNAI. Springer-Verlag, December 2001.

[BCHP95] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Data–flow Analysis of Standard Prolog Programs. In *ICLP95 WS on Abstract Interpretation of Logic Languages*, Japan, June 1995.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[BdlBCH93] F. Bueno, M. García de la Banda, D. Cabeza, and M. Hermenegildo. The &–Prolog Compiler System — Automatic Parallelization Tools for LP. Technical Report CLIP5/93.0, Computer Science Dept., Technical U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1993.

[BdlBH93] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. Technical Report TR CLIP7/93.0, T.U. of Madrid (UPM), Facultad Informática UPM, 28660-Boadilla del Monte, Madrid-Spain, October 1993.

[BdlBH94] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case

Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[BdlBH+01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey. A Model for Inter-module Analysis and Optimizing Compilation. In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[BIM89] BIM, B-3078 Everberg, Belgium. *BIM Prolog release 2.4*, March 1989.

[BJ88] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, August 1988. MIT Press.

[BLGPH04] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP1/04, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2004.

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19–20:443–502, July 1994.

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

[Car88] M. Carlsson. *Sicstus Prolog User's Manual*. Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.

[CDFB93] Michael Codish, Dennis Dams, Gilberto File, and Maurice Bruynooghe. Freeness Analysis for Logic Programs - And Correctness? In *Proc. Int'l. Conf. on Logic Programming*. MIT Press, 1993. To appear.

[CDG93] M. Codish, S. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL'93*, pages 451–464, Charleston, South Carolina, 1993. ACM.

[CdlBBH94] M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. Goal Dependent vs Goal Independent Analysis of Logic Programs. In F. Pfenning, editor, *Fifth International Conference on Logic Programming and Automated Reasoning*, number 822 in LNAI, pages 305–320, Kiev, Ukraine, July 1994. Springer-Verlag.

[CF91] A. Cortesi and G. File. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *ACM Symposium on Partial Evaluation and Semantic Based Program Manipulation*, pages 52–61, New York, 1991.

[CFR92] A. Cortesi, G. File, and S. Rossi. Abstract interpretation of prolog: the treatment of the built-ins. In *Proc. of the 1992 GULP Conference on Logic Programming*, pages 87–104. Italian Association for Logic Programming, June 1992.

[CG93] D. Cabeza Gras. Parallelization of Prolog Programs Using the Notion of Non-Strict Independence. Master's thesis, T. University of Madrid (UPM), Facultad de Informática, Madrid, 28660, December 1993.

[CGC+03a] J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for CLP Systems. In *2003 International Conference on Logic Programming*, number 2916 in LNCS, pages 481–482, Heidelberg, Germany, December 2003. Springer-Verlag. Extended abstract.

[CGC+03b] J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for CLP Systems (And Two Useful Implementations). In M. Carro and J. Correas, editors, *Second CoLogNet Workshop on Implementation Technology for Computational Logic Systems (Formal Methods '03 Workshop)*, pages 51–64, School of Computer Science, Technical University of Madrid, September 2003. Facultad de Informatica.

[CGC+04] J. Correas, J. M. Gomez, M. Carro, D. Cabeza, and M. Hermenegildo. A Generic Persistence Model for CLP Systems (And Two Useful Implementations). In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 104–119, Heidelberg, Germany, June 2004. Springer-Verlag.

[CGT98] D. Cabeza, S. Genaim, and C. Taboch. WOF Design. Technical Report D2.1.M2, RADIOWEB Project, January 1998.

[CH94] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[CH95a] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from `http://www.cliplab.org/`.

[CH95b] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from http://www.cliplab.org/.

[CH96] D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.cliplab.org/`.

[CH97] D. Cabeza and M. Hermenegildo. WWW Programming using Computational Logic Systems (and the PiLLoW/Ciao Library). In *Proceedings of the Workshop on Logic Programming and the WWW at WWW6*, San Francisco, CA, April 1997.

[CH99a] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 110–128. N.M. State U., December 1999.

[CH99b] D. Cabeza and M. Hermenegildo. The Ciao Modular Compiler and Its Generic Program Processing Library. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*, pages 147–164. N.M. State U., December 1999.

[CH00a] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

[CH00b] D. Cabeza and M. Hermenegildo. The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[CH00c] D. Cabeza and M. Hermenegildo. The Ciao Module System: A New Module System for Prolog. In *Special Issue on Parallelism and Implementation of (C)LP Systems*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[CH01] D. Cabeza and M. Hermenegildo. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.

[Che87] W. Chen. A theory of modules based on second-order logic. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 24–33, San Francisco, 1987.

[CHGT98a] D. Cabeza, M. Hermenegildo, S. Genaim, and C. Taboch. Design of a Generic, Homogeneous Interface to Relational Databases. Technical Report D3.1.M1-A1, CLIP7/98.0, RADIOWEB Project, September 1998.

[CHGT98b] D. Cabeza, M. Hermenegildo, S. Genaim, and C. Taboch. Layout and style (last) language, a preliminary design. Technical Report D2.2.M2, RADIOWEB Project, December 1998.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CHV96] D. Cabeza, M. Hermenegildo, and S. Varma. The PiLLoW/Ciao Library for INTERNET/WWW Programming using Computational Logic Systems. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet`.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[CLMV96] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Proving properties of logic programs by abstract diagnosis. In M. Dams, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop*, number 1192 in Lecture Notes in Computer Science, pages 22–50. Springer-Verlag, 1996.

[CLV94] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *POPL'94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 227–239, Portland, Oregon, January 1994. ACM.

[CMB+93] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

[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.

[CW94] M. Carlsson and J. Widen. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, April 1994.

[Deb89a] S. K. Debray. Flow analysis of dynamic logic programs. *Journal of Logic Programming*, 7(2):149–176, September 1989.

[Deb89b] 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.

[Deb92] S. K. Debray, editor. *Journal of Logic Programming, Special Issue: Abstract Interpretation*, volume 13(1–2). North-Holland, July 1992.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

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

[DJBC93] V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness Analysis in the Presence of Numerical Constraints. In *Tenth International Conference on Logic Programming*, pages 100–115. MIT Press, June 1993.

[GDL92] Roberto Giacobazzi, Saumya Debray, and Giorgio Levi. A generalized semantics for constraint logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 581–591, ICOT, Japan, 1992. Association for Computing Machinery.

[GDMS02] María J. García de la Banda, Bart Demoen, Kim Marriott, and Peter J. Stuckey. To the Gates of HAL: A HAL Tutorial. In *International Symposium on Functional and Logic Programming*, pages 47–66, 2002.

[GGL94] M. Gabbrielli, R. Giacobazzi, and G. Levi. Goal independency and call patterns in the analysis of logic programs. In *ACM Symposium on Applied Computing*, pages 394–399. ACM Press, 1994.

[GJ89] G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.

[GM86] J.A. Goguen and J. Meseguer. Eqlog: equality, types, and generic modules for logic programming. In *Logic Programming: Functions, Relations, and Equations*, Englewood Cliffs, 1986. Prentice-Hall.

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

[HC97] M. Hermenegildo and The CLIP Group. Programming with Global Analysis. In *Proceedings of ILPS'97*, pages 49–52, Cambridge, MA, October 1997. MIT Press. (abstract of invited talk).

[HCC95] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[Her96] M. Hermenegildo. Writing "Shell Scripts" in SICStus Prolog, April 1996. Posting in `comp.lang.prolog`. Available from `http://www.cliplab.org/`.

[Her99] M. Hermenegildo. A Documentation Generator for Logic Programming Systems. In *ICLP'99 Workshop on Logic Programming Environments*, pages 80–97. N.M. State University, December 1999.

[Her00] M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[HG90] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

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

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, June 1990.

[HL94] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[HPBLG03] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[HPMS95] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR90] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[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.

[HWD92] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[JL92] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[KL02] A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4–5):32, July 2002.

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

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

[LRV94] B. Le Charlier, S. Rossi, and P. Van Hentenryck. An Abstract Interpretation Framework Which Accurately Handles Prolog Search–Rule and the Cut. In *International Symposium on Logic Programming*, pages 157–171. MIT Press, November 1994.

[LV94] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. *ACM Transactions on Programming Languages and Systems*, 16(1):35–101, 1994.

[MCH03] J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Type and Determinism Information: Preliminary Results. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, pages 89–102, Bombay, December 2003.

[MCH04] J. Morales, M. Carro, and M. Hermenegildo. Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In *Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages*, number 3057 in LNCS, pages 86–103, Heidelberg, Germany, June 2004. Springer-Verlag.

[MH89] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Inter-

pretation. 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.

[Mil89] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, pages 79–108, 1989.

[MMNH01] S. Muñoz, J.J. Moreno-Navarro, and M. Hermenegildo. Efficient Negation Using Abstract Interpretation. In *Proc. of the Eigth International Conference on Logic Programming and Automated Reasoning*, LNAI. Springer-Verlag, December 2001.

[MMRS55] John McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence. Report, manuscript, MITAI, Cambridge, MA, August 1955.

[MO84] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MP89] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Logic Programming: Proc. of the Sixth International Conference*, pages 284–299. MIT Press, Cambridge, MA, 1989.

[MSJ94] K. Marriott, H. Søndergaard, and N.D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[Nai96] L. Naish. Higher-order logic programming. Technical Report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1996. URL: http://www.cs.mu.oz.au/~lee/papers/ho/.

[NP92] G. Nadathur and F. Pfenning. The type system of a higher-order logic programming language. Report cs-1992-02, Duke University, 1992.

[O'K85] R.A. O'Keefe. Towards an algebra for constructing logic programs. In *IEEE Symposium on Logic Programming*, pages 152–160, Boston, Massachusetts, July 1985. IEEE Computer Society.

[PB02] A. Pineda and F. Bueno. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint and LOgic Programming Systems (ICLP associated workshop)*, Copenhagen, July 2002.

[PBH97] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997. Available from `ftp://clip.dia.fi.upm.es/pub/papers/assert_lang_tr_disciplde liv.ps.gz` as technical report CLIP2/97.1.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[PBH00b] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[PCH+04] G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe

and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004.

[Pfe88] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *Proc. ACM Conference on Lisp and Functional Programming*, 1988.

[PH95] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[PH99a] A. Pineda and M. Hermenegildo. O'Ciao: An Object Oriented Programming Model for (Ciao) Prolog. Technical Report CLIP 5/99.0, Facultad de Informática, UPM, July 1999.

[PH99b] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[PH99c] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *ICLP'99 Workshop on Optimization and Implementation of Declarative Languages*, pages 45–61. U. of Southampton, U.K, November 1999.

[PH00] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *Special Issue on*

Optimization and Implementation of Declarative Programming Languages, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[PRO94] International Organization for Standardization, National Physical Laboratory, Teddington, Middlesex, England. *PROLOG. ISO/IEC DIS 13211 — Part 1: General Core*, 1994.

[PRO00] International Organization for Standardization, 1, rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland. *PROLOG. ISO/IEC DIS 13211-2 — Part 2: Modules*, 2000.

[Pue97] G. Puebla. *Advanced Compilation Techniques based on Abstract Interpretation and Program Transformation*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, November 1997.

[Qui86] Quintus Computer Systems Inc., Mountain View CA 94041. *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[RK89] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

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

[SCWY91] V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.

[She92] K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, U. of Cambridge, 1992.

[Søn86] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[SSW93] K. Sagonas, T. Swift, and D.S. Warren. The XSB Programming System. In *ILPS Workshop on Programming with Logic Databases*, number TR #1183, pages 164–164. U. of Wisconsin, October 1993.

[ST84] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[Sun91] R. Sundarajan. An Abstract Interpretation Scheme for Groundness, Freeness, and Sharing Analysis of Logic Programs. Technical Report CIS-TR-91-06, U. of Oregon, Eugene, Oregon 97403, October 1991.

[Swe95] Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.

[Tay91] A. Taylor. High Performance Prolog Implementation through Global Analysis. Slides of the invited talk at PDK'91, Kaiserslautern, 1991.

[VD92] P. Van Roy and A.M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.

[War82] D.H.D. Warren. Higher-order extensions to prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

[War90] D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

[WC87] D.S. Warren and W. Chen. Formal semantics of a theory of modules. Technical report 87/11, SUNY at Stony Brook, 1987.

[WHD88] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

[Win92] W. Winsborough. Multiple Specialization using Minimal-Function Graph Semantics. *Journal of Logic Programming*, 13(2 and 3):259–290, July 1992.