UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Técnicas Avanzadas de Compilación para Programación Lógica

Advanced Compilation Techniques for Logic Programming

Tesis Doctoral

José F. Morales

Julio de 2010

Técnicas Avanzadas de Compilación para Programación Lógica

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

Candidato: **José Morales**

Ingeniero en Informática
Universidad Politécnica de Madrid
España

Director: **Manuel Carro**

Profesor Titular de Universidad

Co-Director: **Manuel Hermenegildo**

Catedrático de Universidad

Madrid, Julio de 2010

A mis padres y hermano,
a Estefanía y a nuestra hija Natalia.

Agradecimientos

Me gustaría expresar mi más sincero y profundo agredecimiento a mi director de tesis, Manuel Carro, por su continua ayuda en mi doctorado e investigación, compartir conmigo su talento y conocimientos y darme la oportunidad de trabajar en este área. Quisiera hacer público mi agradecimiento a Manuel Hermenegildo, director del grupo CLIP, por su sabiduría e incomparable energía.

También me gustaría agradecer a los miembros actuales y pasados del grupo CLIP, que han hecho posible y contribuido al desarrollo de los sistemas Ciao Prolog y el preprocesador CiaoPP, sin los cuales mi trabajo no hubiera posible: Francisco Bueno, Germán Puebla, Elvira Albert, Puri Arenas, Pedro López, Daniel Cabeza, Jesús Correas, Claudio Vaucheret, Claudio Ochoa, Edison Mera, Astrid Beascoa y muchos otros.

Este trabajo no habría sido posible sin la teoría, herramientas y sistemas fruto de muchos otros investigadores. Quisiera agradecer a los los autores y desarrolladores de otros sistemas Prolog y de programación lógica, especialmente aquellos que conocí durante mi investigación de doctorado: Jan Wielemaker, Bart Demoen, Kostis Sagonas, Vitor Santos Costa... Todavía tengo muchas cosas que aprender de ellos.

Como nota personal, estoy en deuda con mis padres y hermano, que siempre han apoyado y entendido en cada decisión que he tomado en mi vida, incluyendo este doctorado. y a mi compañera e hija, cuyo amor incondicional me animó a terminar esta tesis.

Finalmente, me gustaría agradecer a las instituciones y proyectos de investigación que han financiado mis actividades investigadoras como estudiante de postgrado: la Universidad Politécnica de Madrid (UPM), el Ministerio de Educación a través de los proyectos de investigación *MERIT* y *CUBICO*, la Comunidad de Madrid a través del proyecto *PROMESAS* y al programa Information Society Technologies de la European Commission, Future and Emerging Technologies, a través de los proyectos *ASAP* y *MOBIUS*.

Sinopsis

Los lenguajes de programación declarativos permiten expresar programas en un lenguaje que es más cercano al problema que a los detalles de implementación. A pesar de la genericidad de esta definición, Lloyd propone una noción más clara de la *declarativididad* [Llo94], definiendo los programas como teorías en una lógica adecuada y la computación deducción en base a la teoría.

Prolog es uno de los lenguajes de programación del paradigma lógico más importantes, cuya teoría es la de la la deducción lógica. Implementaciones eficientes capaces de competir con muchos otros lenguajes de alto nivel y su flexibilidad, han hecho de Prolog un importante punto de partida para desarrollar nuevas ideas, como la programación con restricciones y multi-paradigma, mezclando programación funcional, orientada a objetos e imperativa.

Aunque el estado del arte de las implementaciones de Prolog esta altamente optimizado para el tipo de problemas de busqueda para los que este está diseñado y que puede competir con muchas implementaciones de otros paradigmas — tanto lógico, funcional, como imperativo — su naturaleza caracteristicas dinámicas y declarativas imponen una considerable limitación en eficiencia. Un ambicioso objetivo para las implementaciones de Prolog, compartido por muchos otros lenguajes declarativos, consiste en el desarrollo de téctnicas que superen esta limitación sin sacrificar la expresividad del lenguaje.

El objetivo de esta tesis es el desarrollo y mejora de técnicas avanzadas de compilacion para Prolog, en principio ortogonales a extensiones como la programación lógica con restricciones [JM87], Prolog con tabulación [War92] o CHR sobre Prolog [SF08].

Las principales contribuciones presentadas en esta tesis pueden resumirse en:

- Un compilador optimizante de Prolog, donde predicados seleccionados son compilados a C y diseñado para aceptar información de alto nivel, obtenida

mediante análisis automático y expresada en un lenguaje estandarizado de aserciones.

- Un enfoce automático a la generación de máquinas abstractas, donde el conjunto de instrucciones y la representación de código de *byte* y datos son definidas de forma individual.

- Una descripción del conjunto de instrucciones completo de una máquina abstract para Prolog, en un dialecto de Prolog extendido para manejar cambios de estado (en forma de variables mutables) y adecuado para realizar transformaciones automáticas de programa.

- Una representación de *tagged words* (palabras con etiquetas) en un lenguaje de alto nivel, explorando variantes para los casos de 32 y 64 bit.

- Un marco de trabajo paramétrico para la generación de variaciones de máquinas abstractas, para explorar optimizaciones de forma general o enfocadas a un conjunto particular de programas.

- Un estudio de la combinación de técnicas de compilación optimizante en código fuente y en código de bajo nivel, en un caso de prueba real para sistemas ubícuos.

Resumen*

Motivación

La programación, como esfuerzo matemático para diseñar un algoritmo que resuelva un problema específico, junto con la tarea de codificación en un lenguaje ejecutable por un computador, puede definirse de forma precisa. Aun no siendo una tarea fácil, determinar si un programa es correcto y adecuado para una problema dado puede realizarse mediante pruebas (formales) sobre la corrección, la complejidad en uso de memoria y tiempo y datos empíricos en una máquina particular sobre su rendimiento y tamaño de código.

Cuando el objetivo es encontrar una codificación óptima de un algoritmo, el lenguaje de programación adecuado puede ser cuestionable. En términos de *computabilidad*, la elección no permite escribir más algoritmos de aquellos ya expresables en máquinas de Turing y tampoco permite escribir algoritmos que superen las limitaciones del *hardware*. Para una plataforma determinada, la mejor solución si nos centramos en el problema del rendimiento y el uso de memoria, será siempre expresable en lenguaje máquina (pues cualquier otro lenguaje es directa o indirectamente traducido a éste o interpretado mediante otro programa).

Sin embargo, los lenguajes máquina han pasado a ser usados sólo en dominios muy concretos, viéndose desplazados por lenguajes de bajo nivel (como el lenguaje C), que surgieron principalmente como una forma de aliviar los problemas de portabilidad en diferentes arquitecturas, facilitar tareas de programación pesadas y que gracias al desarrollo de la compilación optimizante ofrecen un buen rendimiento.

*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.

Aún así, en estos lenguajes el programador tiene un control estricto sobre los recursos y las operaciones hardware. Los programas incluyen detalles muy precisos sobre el flujo de control, el tamaño y las forma de los datos, lo que hace que los programadores experimentados sean capaces de comprender que está ocurriendo en la ejecución con gran precisión, incluso al nivel de registros máquina.

El punto de vista de la programación como ingeniería es muy diferente. Los programadores, como seres humanos, están ligados a fechas de entrega, cometen errores e introducen fallos (*bugs*) en la programación. Por otro lado, los requisitos del *software* no siempre están especificados y pueden ser imprecisos, o verse modificados durante el desarrollo, requiriendo cambios en los programas. Más aún, el proceso de desarrollo no es lineal en la práctica, lo que significa que muchas partes del código de un proyecto pueden ser esbozadas como prototipos y refinadas más adelante. Incluso si estos problemas no fueran suficientes, existen restricciones fuertes de seguridad, donde hay ciertos fallos que no son admisibles en absoluto: por ejemplo, que una aplicación externa acceda directamente a los recursos de la máquina. A ésto se añaden otros factores, como el balance entre el coste humano de producir programas optimizados frente al coste de invertir en recursos *hardware* para aliviar ineficencias (como usar, siempre que sea posible, más procesadores, procesadores más rápidos y más memoria, en ocasiones acompañandose de un mayor consumo energético).

En resumen, podemos observar una importante asimetría entre qué codificación de un programa es mejor de cara a la ejecución eficiente en una máquina y cuál desde el punto de vista de los factores prácticos del desarrollo.

Esta asimetría es salvable en ocasiones. Por ejemplo, el tiempo de ejecución no se dedica uniformemente al código: hay secciones que concentran la mayor parte de la ejecución. Se acepta como una buen compromiso reescribir las partes críticas de una aplicacion en lenguajes de bajo nivel (atendiendo a los detalles necesarios para asegurar la eficiencia, incluso si se sacrifica la reutilización, claridad, etc.) y preservar un estilo limpio en el resto del código.

Sin embargo, aunque existen técnicas de verificación de código que pueden ayudar en probar la corrección u otras propiedades del código de bajo nivel, en ocasiones se ha sacrificado las ventajas de una especificación más abstracta. Poder preservar la abstracción del problema a lo largo de todo el programa, mediante el desarrollo de lenguajes y el desarrollo de traducciones eficientes a

código ejecutable, es uno de los más temas de investigación más importantes en este área.

De los Lenguajes de Bajo Nivel a la Programación Declarativa

Al contrario que en los lenguajes de bajo nivel, la abstracción en lenguajes de alto nivel permiten expresar programas en un lenguaje que es más cercano al problema que se pretende resolver. Uno de estos enfoques es la *programación declarativa*, que intuitivamente — a veces de forma no muy precisa — es definida como un estilo de programación donde los programas definen *qué* se ha de resolver, pero no *cómo* ha de hacerse. Una explicación más profunda de esta frase es necesaria, como Lloyd indica en [Llo94]. Usando la terminología de la equación de Kowalski *algoritmo = lógica + control* [Kow79], que define un algoritmo como la combinación de una *lógica* y un mecanismo de *control*, un lenguaje de programación declarativo sólo necesita describir la lógica del programa y no su control.

Pero esta definición, que refleja la idea intuitiva del paradigma, puede ser equívoca dado que no todos los problemas que son expresables en la lógica pueden ser resueltos automáticamente. Lloyd describe que la idea principal de la programación declarativa consiste en que los programas son teorías (en una lógica adecuada) y que la computación es la deducción en esa teoria. Una lógica adecuada debe tener una teoría modelo, una teoría de prueba, teoremas de corrección (todas las respuestas computadas deben ser correctas) y preferentemente, teoremas de completitud (todas las respuestas correctas son computadas). La visión de Lloyd es suficientemente amplia para agrupar muchos paradigmas de programación bajo el ámbito de la programación declarativa: programación funcional (computación como evaluación de funciones matemáticas) programación lógica (computación basada en pruebas de teoremas y lógica matemática) y otros métodos formales.

Retos de la Programación Declarativa

La lógica formal detrás de la programación declarativa a menudo simplifica el razonamiento automático sobre los programas y los programas escritos en un lenguaje que está más cerca a la especificación son más fáciles de escribir y verificar. Esto tiene algunas implicaciones profundas:

- Debido a la amplia expresividad del lenguaje, los mecanismos de ejecución

tienen un carácter muy general, lo que se traduce en ocasiones (especialmente en los sistemas menos maduros) en un rendimiento pobre, comparado con aquél que ofrecen los lenguajes de bajo nivel. Por ejemplo, los lenguajes donde la aritmética es por defecto de multiple precisión pueden ser menos eficientes que aquellos donde el usuario hace explícito el tamaño máximo de cada variable numérica, pero liberan al programador de esta tarea.

- Las transformaciones y los análisis de programas (como evaluación parcial e interpretación abstracta) son en principio más fáciles de aplicar.

- No describir un problema a un nivel detallado y específico a la máquina implica que existe más libertad por parte de los mecanismos de ejecución para encontrar soluciones. Sin cambiar el programa original, es posible adaptarlo a distintas plataformas *hardware* y aprovechar sus características especiales (como paralelismo, instrucciones SIMD, etc.).

El Lenguaje Prolog

Prolog es un lenguaje basado en la lógica de primer orden. La historia del lenguaje ha sido descrita en detalle en la literatura [Col93]. El uso de la deducción lógica como computación fue propuesta en 1960 por Cordell Green y otros, pero no fue hasta que Colmerauer y Kowalski crearon Prolog que comenzó a ser viable. Prolog nació como un projecto dirigido al procesamiento del lenguaje natural, cuya primera versión preliminar data de finales de 1971. A lo largo de los años se convirtió en un lenguaje de propósito general.

Sin embargo, Prolog también ha sido considerado como un lenguaje de bajo nivel entre el lenguaje lógico, debido a sus limitadas propiedades de terminación y su uso frecuente de efectos laterales. No obstante, desde el nacimiento de Prolog, han aparecido soluciones a la formalización de los efectos laterales en programación declarativa para su uso en aplicaciones reales, que han sido exploradas tanto en programación funcional como lógica. Prolog ha demostrado ser un lenguaje muy flexible para desarrollar nuevas ideas, como la programación con restricciones y lenguajes multiparadigmas que mezclan programación funcional, orientada a objetos e imperativa. En general, los sistemas Prolog ofrecen un largo repertorio de técnicas para conseguir buen rendimiento, como se muestra en el Capítulo 2.

Importantes lenguajes que se derivan o heredan algunos aspectos de Prolog son LIFE [AKP91] (lógico, functional, orientado a objetos), Mozart-Oz [HF00] o Mercury [SHC96].

Objetivos de la Tesis

Como ya se ha mencionado antes, uno de los objetivos más ambiciosos de la programación declarativa es la mejora de las técnicas de compilación para reducir la diferencia de rendimiento, como lenguaje de programación general, respecto a lenguajes más tradicionales como la programación imperativa.

En esta dirección, el objetivo de esta tesis es el desarrollo de técnicas avanzadas de compilación para programación lógica, tanto a código nativo como a código de *byte* emulado. Los primeros trabajos existentes en compilación de Prolog a código nativo, a pesar su lentitud y grandes ejecutables, mostraron buenos resultados en comparación con código emulado. Sin ambargo, estos trabajos fueron parcialmente abandonados debido a importantes mejoras en técnicas de emulación y a los problemas de portabilidad de los compiladores de código nativo.

No obstante, es claro que algunos algoritmos sólo pueden ser ejecutados de forma eficiente como código muy especializado, que está fuera del ámbito de técnicas de emulación genéricas y sólo disponible cuando se usa código de bajo nivel. En esta tesis, exploramos soluciones a este problema, que aunque están enfocadas a Prolog también son en principio aplicables a otros lenguajes o extensiones, como programación lógica con restricciones [JM87], Prolog con tabulación [War92] o CHR sobre Prolog [SF08]. La relevancia de estos resultados también puede extenderse a otros lenguajes que comparten algunos mecanismos de ejecución (por ejemplo, lenguajes dinámicamente tipados), ya que las técnicas de optimización son similares.

Estructura de la Tesis

Esta tesis se divide en tres partes conceptuales. La primera parte (Capítulo 2) ofrece una introducción a las técnicas de compilación de Prolog. La segunda parte (Capítulos 3, 4, 5 y 6) desarrolla las técnicas de compilación descritas en esta tesis

y las evalúa. La parte final (Capítulo 7) muestra un caso de estudio que usa las técnicas anteriormente desarrolladas.

Implementación y Optimización Sequencial de Prolog: una (Breve) Introducción [Capítulo 2]

En este capítulo se introducen varias de las técnicas para la implementación de Prolog que han sido propuestas desde el sistema original escrito por Colmerauer y Roussel [Col93]. Una forma interesante de clasificar los sistemas Prolog (que de hecho es extensible a muchos otros lenguajes) divide las implementaciones en interpretes, compiladores a código de *byte* y compiladores a código de bajo nivel. En los intérpretes la ejecución suele ser precedida de un leve preproceso o traducción, pero la mayor parte del trabajo es realizado en tiempo de ejecución. En compiladores a código de bajo nivel, el código generado es ejecutable directamente (o indirectamente a través de un compilador de bajo nivel) por la máquina. Alternativamente, la semántica de un lenguaje puede hacerse corresponder a una máquina abstracta con un conjunto de instrucciones y un esquema de compilación predefinidos, de tal forma que el lenguaje fuente es simplificado a este código antes de la ejecución. Este código simplificado puede usarse como representación intermedia para generación de código de bajo nivel o codificado como código de *byte* y ejecutado por un intérprete, llamado *emulador*.

En este capítulo se revisan las máquinas abstractas más extendidas para Prolog, la *Máquina Abstracta de Warren* (*Warren Abstract Machine*, WAM) [War83, AK91], algunas variantes y alternativas y se introducen los avances más relevantes de implementación, junto con la descripción de los sistemas Prolog más significativos que las introducen o utilizan: los primeros intérpretes (como C-Prolog [Per87]), emuladores de código de *byte*, compiladores de código nativo y propuestas híbridas que combinan varias técnicas.

Compilación de Prolog a Código Nativo [Capítulo 3]

La emulación de código de *byte* puede ser muy útil y eficiente, en los casos donde se necesitan programas ejecutables pequeños y tiempos de compilación reducidos, así como para partes donde el rendimiento no es crítico y/o tienen una fuerte componente simbólica. Por otro lado, la emulación limita el grado de optimización

que puede realizarse, por ejemplo, al no permitir la especialización muy detallada del código de las instrucciones. En algunos casos, generar el código más eficiente necesita optimizaciones en todos los niveles y eventualmente traducir a código nativo.

En este capítulo se describen los resultados de un prototipo de compilador de Prolog a C, `ciaocc`. A través de un análisis automático del programa Prolog y haciendo uso de un lenguaje estandarizado de aserciones, `ciaocc` está diseñado para aceptar diferente tipo de información de alto nivel, que es usada posteriormente para optimizar el código C resultante. El código no interpretado es directamente ejecutado por la máquina gracias a compiladores de C estándar. El fundamento principal del proceso de traducción consiste en el *desplegado* (*unfolding*) del emulador de código de *byte* respecto a un código de *byte* concreto. Este diseño tiene algunas consecuencias obvias: permite reutilizar gran parte de la maquinaria del emulador de código de *byte*. Por ejenplo, predicados ya escritos en C, definiciones de datos, áreas de memoria y las rutinas de recolección de basura, etc. Este diseño también está preparado para combinar código de *byte* emulado con código nativo de una forma relativamente sencilla.

Este capítulo estudia el rendimiento de programas compilados a C frente a los que usan código de *byte*, usando y sin usar las optimizaciones que se pueden obtener mediante la información de análisis.

Un Generador Genérico de Emuladores [Capítulo 4]

Las primeras decisiones en el diseño de una máquina abstracta pueden afectar al rendimiento de un emulador y al tamaño los programas de código de *byte* de formas que son en ocasiones difíciles de prever. Entre estas decisiones se incluye la representación de los datos e instrucciones en memoria, así como el juego de instrucciones y su nivel de especialización (a través de instrucciones que, aunque no son imprescidibles, se introducen por representar casos específicos que pueden ejecutarse de forma más eficiente que por medio de instrucciones genéricas). Una vez se establecen y el sistema es implementado, algunos de estos parámetros son extremadamente difíciles de alterar. Esto hace que el estudio de diferentes alternativas de la máquina abstracta, o la generación de versiones específicas de ésta con propósitos específicos, sea bastante arduo. Incluso si esta tarea es realizada por programadores experimentados, si se trabaja con sistemas muy

optimizados necesita gran cantidad de tiempo y es propensa a errores debido al nivel de complejidad del código.

Con el objetivo de hacer más fácil estas tareas, este capítulo propone un enfoque sistemático para la generación automática de máquinas abstractas. Este nuevo enfoque consiste en la separación conceptual del juego de instrucciones de la codificación de datos y del código de *byte*. Es decir, las instrucciones son simplificadas para definir sólamente la semántica de la máquina abstracta y la representación de datos e instrucciones es proporcionada por una serie de reglas. Dichas definiciones se combinan automáticamente para obtener el código que una vez compilado da lugar a un emulador. A su vez, las reglas de representación son usadas por el compilador para generar, a partir de la representación intermedia, el código de *byte*.

Ilustramos la viabilidad de este enfoque mediante la implementación de un generador de máquinas abstractas (comparables en eficiencia y arquitectura con otras consideradas como parte del "estado del arte") basadas en la WAM. Basado en este generador, se ha experimentado con esquemas de especialización de la máquina abstracta para un conjunto particular de programas. Como ejemplo práctico, se estudió la reducción del tamaño del emulador mediante la eliminación de instrucciones no usadas.

Descripción y Optimización de Máquinas Abstractas en un Dialecto de Prolog [Capítulo 5]

Incluso si los inconvenientes de las representaciones de bajo nivel son separados en una capa de abstracción diferente, como se describe en el capítulo anterior, las máquinas abstractas para Prolog (o lenguajes relacionados) siguen siendo grandes e intrincadas.

En este capítulo mostramos como la semántica de la mayor parte de los componentes básicos de las máquinas virtuales (eficientes) para Prolog pueden ser descritos usando (una variante de) Prolog que preserva gran parte de su semántica. Este lenguaje está diseñado para ser ejecutado de forma eficiente una vez compilado, pero a su vez ser de mayor nivel de abstracción que el lenguaje C.

Desde este lenguaje, las descripciones de los componentes básicos (como las instrucciones) son compiladas a C y ensambladas para construir un emulador

de código de *byte* completo. Gracias al mayor nivel de abstracción del lenguaje utilizado y su cercanía a Prolog, la descripción de la máquina abstracta puede ser manipulada usando técnicas de compilación y optimización de Prolog con relativa facilidad. También se ve simplificada la incorporación de optimizaciones más sofisticadas. Se muestra como, trás la aplicación selectiva de transformaciones en el juego de instrucciones, podemos obtener implementaciones de máquinas abstractas cuyo rendimiento es comparable al de implementaciones realizadas a mano y cuidadosamente optimizadas. En nuestro caso, fue posible incluso mejorar la máquina de partida mediante la exploración automática de un gran número combinaciones, que habría sido muy difícil realizar de forma manual.

Comparación de Variaciones de Esquemas de *Tags* Usando un Generador de Máquinas Abstractas [Capítulo 6]

En este capítulo se estudia, en el contexto de máquinas abstractas basadas en la WAM, como variaciones en la codificación de información de tipos en *palabras etiquetadas* (*tagged words*) y sus operaciones básicas asociadas afectan el rendimiento y el uso de memoria. La codificación de este tipo de datos y las operaciones asociadas son realizadas en el dialecto de Prolog descrito en el capítulo anterior y un generador automático construye tanto la máquina abstracta que utiliza esta codificación como el compilador asociado a código de *byte*. Las anotaciones que acepta el lenguaje hacen posible imponer restricciones en la representación final de las *palabras etiquetadas*, como el espacio de direccionamiento efectivo (fijando, por ejemplo, el tamaño de la palabra del procesador/arquitectura destino), la disposición de los bits de *etiqueta* y *valor* dentro de la *palabra etiquetada* y como las operaciones básicas son implementadas.

En cuanto a lo resultados experimentales, en este capítulo se sigue una metodología similar al Capítulo 5, consiguiendo generar automáticamente código que presenta optimizaciones relativas a la representación de *palabras etiquetadas*, equivalentes a las utilizadas en máquinas escritas a mano y altamente optimizadas. En nuestro caso, se ha sido capaz de mejorar una configuración inicial que representa la máquina de partida y obtener un mayor espacio de direccionamiento y rendimiento. Adicionalmente, evaluamos un gran número de combinaciones de párametros en dos escenarios. En el primero, se intenta obtener una máqui-

na de propósito general. En el segundo, se generan automáticamente máquinas abstractas ajustadas para un programa particular.

Caso de Estudio: Procesamiento de Sonido en Tiempo Real [Capítulo 7]

Este capítulo recapitula y hace uso de algunos de los resultados presentados en los capítulos anteriores a través del estudio, en un caso concreto, de la viabilidad del los lenguajes de programación lógica de alto nivel en el diseño e implementación de aplicaciones dirigidas a computación ubicua.

El caso estudio es un *espacializador de sonido* ("sound spatializer"), que dadas las señales en tiempo real de posición y sonido monoaural, generan sonido biaural que simula originarse en una posición determinada del espacio. El uso de transformaciones avanzadas en tiempo de compilación y optimizaciones han hecho posible ejecutar código escrito en un estilo claro de alto nivel, ajustandose las limitaciones de tiempo de ejecución y memoria impuestas por un dispositivo ubicuo. El ejecutable final es comparado positivamente frente a implementaciones similares escritas en C. Creemos que este caso es representativo de un amplia clase de problemas para computación ubicua y que las técnicas que mostramos aquí pueden usarse en un gran número de escenarios. Esto apunta a la posibilidad de usar lenguajes de alto nivel, con su flexibilidad asociada y herramientas de compilación y análisis sofisticadas, al área de sistemas ubicuos y de tiempo real sin detrimento de eficiencia.

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA

Advanced Compilation Techniques for Logic Programming

PhD Thesis

José F. Morales
July 2010

Advanced Compilation Techniques for Logic Programming

A PhD Thesis
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: **José Morales**
Ingeniero en Informática
Universidad Politécnica de Madrid
España

Advisor: **Manuel Carro**
Profesor Titular de Universidad

Co-Advisor: **Manuel Hermenegildo**
Catedrático de Universidad

Madrid, July 2010

To my parents and brother,
Estefanía, and our daughter Natalia.

Acknowledgments

I would like to express my deep and sincere gratitude to my supervisor, Manuel Carro, for the continuous support of my Ph.D study and research and who gave me the opportunity to work in this fantastic topic. He shared with me a lot of his expertise and research insight. I owe gratitude to Manuel Hermenegildo, for leading the CLIP group with incomparable wisdom and energy.

I wish to thank current and past members of the CLIP group who made Ciao Prolog and the CiaoPP preprocessor possible, and have contributed in its development: Francisco Bueno, Germán Puebla, Elvira Albert, Puri Arenas, Pedro López, Daniel Cabeza, Jesús Correas, Claudio Vaucheret, Claudio Ochoa, Edison Mera, Astrid Beascoa, and others.

This work would not have been possible without the theory, tools and systems that many other researchers did before me. For their openness to share their problems, solutions, ideas, comments, and suggestions, I am in great debt with the authors and developers of other Prolog and logic programming systems, specially those that I met during my Ph.D research: Jan Wielemaker, Bart Demoen, Kostis Sagonas, Vitor Santos Costa... I still have many things to learn from them.

Last, I am in debt with my parents and brother, who always supported and understood me in every decision I took in my life, including this Ph.D, and my partner and daughter, whose unconditional love encouraged me to complete this thesis.

Finally, I am grateful to the institutions and projects which have funded my postgraduate research activities, such as the Technical University of Madrid (UPM), the Spanish Ministry of Education under the *MERIT* and *CUBICO* projects, the Madrid Regional Government under the *PROMESAS* project, the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the *ASAP* and *MOBIUS* projects, and the Universidad Complutese de Madrid (UCM).

Abstract

Declarative programming languages allow the expression of programs in a language that is closer to the problem than to the implementation details. Regardless the generality of that definition, a more clear idea of declarativeness is proposed by Lloyd[Llo94], who proposes that programs are theories in some suitable logic, and computation is deduction from the theory. In logic programming, where Prolog is one of the most popular incarnations of that paradigm, the theory is that of logical deduction. Efficient implementations able to compete with many other high-level languages, and its flexibility, made Prolog a very good framework to develop new ideas, such as constraint programming, and multi-paradigm programming merging functional programming, object oriented programming, and imperative programming.

Although the state of the art of Prolog implementations is highly optimized for the kind of search problems it is designed, and it can compete with many language implementations for other paradigms — both logic, functional, and imperative — its dynamism and declarative nature imposes a considerable efficiency gap. An ambitious goal for Prolog implementations, shared with many other declarative languages, is closing this gap while not sacrificing expressivity.

The objective of this thesis is the development and improvement of advanced techniques for compilation of Prolog, orthogonal to many extensions such as constraint logic programming [JM87], Prolog with tabling [War92], CHR over Prolog [SF08].

The main contributions presented in this thesis can be summarized as:

- An optimizing compiler of Prolog, where selected predicates can be compiled to C, and designed to accept high-level information, obtained from automatic analysis, and expressed in a standardized language of assertions.

- An automatic approach to the generation of abstract machines, where the instruction set, and the bytecode and data representation can be defined individually.
- Description of the full instruction set of a Prolog abstract machine in a Prolog dialect extended with state-changes (as mutable variables), amenable to program transformations.
- Representation of tagged words in the higher-level language, exploring many alternatives for the 32 and 64 bit cases.
- A parametric framework to generate variations of abstract machines, to explore optimizations in general or targeting a particular set of programs.
- Study of the combination of the source-level and low-level optimization compilation techniques in a real test case for embedded devices.

Contents

List of Figures

List of Tables

1 Introduction

Summary

An overview of this thesis is presented in this chapter. It puts in perspective the result of this work and how it contributes to the state of the art of modern compiler technology for declarative languages. We outline the thesis structure and briefly describe each chapter, the related publication and the collaboration with other authors.

1.1 Overview and Motivation

Programming, as a mathematical effort to design an algorithm that solves an specific problem and implementing it in a particular computer language, can be precisely defined. If we can provide (formal) proofs about its correctness, time, and memory complexity, and on a particular machine, about its performance, memory usage, and code size, then we can determine whether we succeeded and we have written a program that accomplish the desired task.

Let us focus on performance and memory usage, and ignore that modern hardware is able to run *inefficient* programs in a reasonable time, since it only means that it could do the same task using less *energy*. We also ignore that the cost of employing good programmers may exceed that of deploying faster hardware. If we are looking for the best solutions for a problem, one might question the role of developing new language abstractions, since they will not represent more programs than those already expressible in machine languages.[1]

[1]Even if we finally encode part of the program as data and write an interpreter.

Machine languages are rarely used, today's low-level languages[2] primarily arised as a way to alleviate portability problems among different architectures and ease tedious programming tasks, and good compilation techniques and optimizations made those languages, such as the C language, a great replacement for hand-written machine code. Still, in those languages the programmer has a tight control on the resources and hardware operations, and programs include precise details about the control flow, and data size and shape. There exists for those languages a well defined translation algorithm. A clear example of a language tightly bound to machine code is C: experienced programmers often accurately imagine what is happening during execution with quite precision, even at register-level.

The engineering view of programming is quite different. Programmers, as human beings, are bound to deadlines, prone to commit errors and introduce bugs. On the other hand, the requirements are not always speficied in a formal language, and they can be imprecise, or be modified over time, requiring changes in the programs. Moreover, the development process is not linear in practice, which means that some pieces of code are often sketched as prototypes and refined later, writting larger code by aggregating smaller program units (components). The smaller code units are often reused, specialized, and modified. Even if those problems were not complex enough by themselves, there exists another constraint: for execution of untrusted code, it is not possible or desirable to allow a program to directly access the machine resources, for security reasons.

Summarizing, we can see here an important asymmetry between what encoding of a program is better for a machine (that *prefers* efficient code) and for a human (that has enough problems managing the code and making sure that the code is correct). Given that execution time is not spent uniformly across all the program source, well engineered programs often distinguishes parts where inefficiency is not an issue, written in a clean (closer to the problem) and flexible (easier to manage) style, from the speed-critical ones written in terms of machine operations. Advanced software verification techniques may even help in automatically proving correctness or other desirable properties of that code. However, it is still written in terms of machine operations even in cases where it is known a simpler executable specification. This motivates a very active branch

[2]Note that the classification of low-level and high-level languages has changed through the history of computing as new abstractions have appeared and proved useful and practical.

of computer science focused on improving language abstractions, and the most efficient translations to executable code.

1.1.1 From Low Level to Declarative Programming

Contrary to low-level languages, abstraction in high-level languages allow the expression of programs in a language that is closer to the problem to be solved. A promising and maturing approach is *declarative programming*, intuitively — and sometimes inaccurately — defined as a programming style where programs state *what* is to be solved, but not *how* it is done. We feel that a deeper explanation of that sentence is required, as Lloyd points out in [Llo94]. Using the terminology of Kowalski's equation *algorithm* = *logic* + *control* [Kow79], which informally states that an algorithm is the combination of some *logic* and *control*, a declarative programming language only describes the logic of the program and not its control. But that definition, which reflects the intuitive idea of the paradigm, can be completelly misleading since not *every* problem expressible in the *logic* could be automatically solved. Lloyd describes that the key idea of declarative programming is that programs are theories (in some suitable logic), and computation is deduction from the theory. A suitable logic has a model theory, a proof theory, soundness theorems (all the computed answers should be correct), and, preferably, completeness theorems (all the correct answers should be computed). Lloyd's view is wide enough to group many programming paradigms under the scope of declarative programming: functional programming (computation as evaluation of mathematical function), logic programming (computation based on theorem proving and mathematical logic), and other formal methods.

1.1.2 Challenges of Declarative Programming

The formal logics behind declarative programming often simplifies automated reasoning about programs, and programs written in a language that are close to the specification are easier both to write and verify. However a main drawback is that the programmer is implicitly forced to know how close is the programming language to its logic and how costly is its execution. That has some profound implications:

- Generality often translates (specially in earlier or naive language implementations) in poor performance when compared with the same problem encoded in a low-level language (e.g. languages with unbounded arithmetic may be more costly than those where the user specifies the maximum size of each cell - but frees the programmer from concerns about precission limits).

- Correct, and interesting, program transformations and analysis (e.g. like partial evaluation or abstract interpretation) and easier to engineer.

- Not describing a problem at a very detailed and machine specific level means that there is more freedom in the implementation of the language logic. Without changes in the original problem, it can be adapted to different hardware platforms, and take advantage of their special features (e.g. parallelism, SIMD).

Note that how different program units are designed, organized, and interact with each other, defines different programming styles such as structured programming, object-oriented programming (e.g. modules, classes, etc.). We will not treat those aspects in this work, since they can be often applied in a language independently of whether it has native support for those techniques.

1.1.3 The Prolog Language

Prolog is a language whose logic, using the Kowalski view, is (unsorted) first order logic. The history of the language has been described in detail in the literature [Col93]. The use of logical deduction as computation was devised in the 1960 by Cordell Green and others, but it was not until Colmerauer and Kowalski's creation of Prolog that it started to become feasible. Prolog was born of a project aimed at processing natural languages. Its first preliminary version dated at the end of 1971, and over the years, it became a full fledged language.

However, Prolog has been seen as a low level language among the logical paradigm, due to its poor termination properties, and frequent use of side-effects. Notwithstanding, since the birth of Prolog, solutions to the formalization of side-effects in declarative programming have appeared, that has been explored in both functional and logic programming when trying to use declarative programming in real world applications. Prolog has proved to be a very flexible language and

framework to develop new ideas, such as constraint programming, and multi-paradigm programming merging functional programming, object oriented programming, and imperative programming. In general, Prolog systems offers large repertoire of techniques to achieve good performance, as it will be reviewed in Chapter 2.

Notable languages that derivate or inherit some aspects from Prolog, are LIFE [AKP91] (logic, functional, object-oriented, contraint of a calculus of order-orted feature approximations), Mozart-Oz [HF00], or Mercury [SHC96].

1.2 Thesis Objectives

An ambitious goal in declarative programming is the enhancement of the implementation techniques to close the efficiency gap, as a general purpose language, with more traditional paradigms, such as imperative programming.

In that direction, the objective of this thesis is the development of advanced compilation techniques for logic programming, both to native code and to bytecode. Earlier works on optimized compilation of Prolog to native code showed impressive results when compared to emulated code, albeit large compilation times, with non-homogeneous speed-ups for the benchmarked programs, and generating very large executables. Modern implementations of Prolog improved emulation techniques so much that the idea of native code generation was partially abandoned. Other problems include portability issues among different architectures, but it is clear that some algorithms can only be efficiently executed as very specialized code, which is out of the scope of pure emulation techniques and only available at the native level. We explored solutions to those problem in this thesis. Note that the techniques developed in this thesis are not in odds with other extensions that preserve Prolog unification of logical variables and non-deterministic control implemented as backtracking, such as constraint logic programming [JM87], Prolog with tabling [War92], CHR over Prolog [SF08]. How those results are relevant to other languages that share some execution mechanisms (for example, dynamically typed languages) is not negligible, since optimization techniques are similar.

A list of the goals that has been addressed in this work are decribed below:

- Develop a compilation scheme that translates Prolog to low level code, in

a portable way, and allowing execution of programs that mix native code and emulated bytecode.

- A modular low level specialization algorithm that uses program annotations obtained by means of program analysis (e.g. information about shape of data and control flow).
- A scheme to share instruction and data definitions between the compiler to low-level code and the abstract machine definition. The approach we followed automatizes the generation of abstract machine generation.
- As a special instance of the compilation techniques to low-level code, reflect in a variant of Prolog the semantics of the abstract machine instructions and the Prolog data types, minimizing the amount of required hand-written C code, allowing more automatic transformations and further extending the portability.
- Evaluate the techniquies in a variety of platforms (servers, desktop machines, and embedded devices).

We will now describe the thesis structure and briefly comment each chapter.

1.3 Structure of the Thesis

This thesis is divided in three conceptual parts. The first part (Chapter 2) provides a background on the compilation techniques for Prolog, as a survey of past and modern systems and their contributions. The middle part (Chapters 3,4,5,6) elaborate on the compilation techniques and evaluates them. The final part (Chapter 7) shows a case study using the developed techniques. We introduce each chapter below.

1.3.1 Sequential Implementation and Optimization of Prolog: a (Short) Survey [Chapter 2]

This chapter surveys several techniques for implementing Prolog that have been devised since the original system written by Colmerauer and Roussel [Col93]. An

interesting classification for Prolog (which is, in fact, extensible to many other languages) divides the implementations in interpreters, compilers to *bytecode*, and compilers to a lower-level language. In interpreters a slight preprocessing or translation might be done before program execution, but the bulk of the work is done at runtime by the interpreter. In compilers to low-level code, the generated code is directly executable. Alternatively, a language can be mapped to an abstract machine with a predefined instruction set and a compilation scheme, so that interpreted language is simplified before the actual execution. This simplified code can be used as an intermediate representation for low-level code generation, or encoded as *bytecode* and executed by an interpreter, called *emulator*. This chapter surveys the most relevant advances in those compilation techniques through the description of current and past Prolog and logic-programming systems, from early interpreters (such as C-Prolog [Per87]), to emulators of bytecode, native code generators, and hybrid approaches. We will review the most extended abstract machine for Prolog, the Warren Abstract Machine (WAM) [War83, AK91], some variations and alternatives, and many implementation and optimization techniques, updating existing surveys [Van94] with more recent developments.

1.3.2 Compiling Prolog to Native Code [Chapter 3]

Bytecode emulation can be very useful and efficient, with small executable programs and quick compile times, and in any case for non-performance bound portions of large symbolic data sets and programs. On the other hand, it puts a limit on the level at which optimizations can be performed, e.g. by not allowing the fine-grained specialization of the instructions code. In order to generate the highest performance code it seems appropriate to perform optimizations at all levels and to eventually translate to machine code. This chapter will describe the results for a prototype compiler of Prolog to C, `ciaocc`. Via an automatic analysis of the initial Prolog program, and expressed in a standardized language of assertions, `ciaocc` is designed to accept different kinds of high-level information, which is later used to optimize the resulting C code. Uninterpreted code is directily executed by the machine by making use an off-the-shelf C compiler. The unfolding of a bytecode emulator with respect to the particular bytecode corresponding to the Prolog program is the basis of the involved translation process. That simple design has some obvious consequences: it allows reusing a sizable

amount of the machinery of the bytecode emulator. For example, the predicates already written in C, data definitions, memory management routines and areas, etc. Its design is as well prepared to mix emulated bytecode with native code in a relatively straightforward way. We report on the performance of programs compiled to C, both with and without making use of simplifications made possible by analysis information.

1.3.3 A Generic Emulator Generator [Chapter 4]

Early decisions in the design of abstract machines may affect the performance of the emulator and the size of the bytecode programs in ways that are often difficult to foresee. That is, abstract machines face complex, and often interacting, decisions regarding data representation, instruction design, instruction encoding, or instruction specialization levels. Once settled down, some of those parameters can hardly be changed. This makes it hard studying different alternatives by implementing abstract machine variants, or the generation of specific implementations for particular purposes. Even if this task can be achieved by experienced programmers, it is a time-consuming and error-prone task because of the level of complexity and optimization of competitive implementations. With the goal of making alternative versions of the abstract machine easier to produce, this chapter proposes a systematic approach to the automatic generation of implementations of abstract machines. The novel approach is the conceptual separation of the instruction set from the data and bytecode representation, and the description of an algorithm to combine them together automatically. That is, instructions define the semantics of the machine, but do not provide the details to encode bytecode instructions at low level. Complementary definitions give the representation definitions. We illustrate the practicality of the approach by reporting on an implementation of a generator of production-quality WAMs which are specialized for executing a particular fixed (set of) program(s). The experimental results show that the approach is can also be effective in reducing emulator size.

1.3.4 Description and Optimization of Abstract Machine in a Dialect of Prolog [Chapter 5]

Even if low-level representation issues are moved in a separate abstraction layer, as it is described in the previous chapter, abstract machines for Prolog and related languages end up being large and intricate. In this chapter we show how the semantics of most basic components of an efficient virtual machine for Prolog can be described using (a variant of) Prolog which retains much of its semantics, but is of higher-level than C. These descriptions are then compiled to C and assembled to build a complete bytecode emulator. Thanks to the high level of the language used and its closeness to Prolog, the abstract machine description can be manipulated using standard Prolog compilation and optimization techniques with relative ease. Incorporating sophisticated optimizations, both at the design and at the implementation levels, is simplified. We also show how, by applying program transformations selectively, we obtain abstract machine implementations whose performance can match and even exceed that of state-of-the-art, highly-tuned, hand-crafted emulators.

1.3.5 Comparing Tag Scheme Variations Using an Abstract Machine Generator [Chapter 6]

In this chapter we study, in the context of a WAM-based abstract machine for Prolog, how variations in the encoding of type information in *tagged words* and in their associated basic operations impact performance and memory usage. We use a high-level language, described in the previous chapter, to specify encodings and the associated operations. An automatic generator constructs both the abstract machine using this encoding and the associated Prolog-to-bytecode compiler. *Annotations* in this language make it possible to impose constraints on the final representation of tagged words, such as the effectively addressable space (fixing, for example, the word size of the target processor / architecture), the layout of the tag and value bits inside the tagged word, and how the basic operations are implemented. We evaluate a large number of combinations of the different parameters in two scenarios. In the first one, it tries to obtain an optimal general-purpose abstract machine. In the second one, specially-tuned abstract machine are automatically generated for a particular program. We conclude that we are

able to automatically generate code featuring all the optimizations present in a hand-written, highly-optimized abstract machine and we can also explore emulators with larger addressable space and better performance than the original starting point.

1.3.6 A Case Study: Real-Time Sound Processing [Chapter 7]

This chapter summarizes and puts to good use some of the results presented in the previous chapters through the study, on a concrete case of the feasibility of using a high-level, general-purpose logic language in the design and implementation of applications targeting wearable computers. The case study is a "sound spatializer" which, given real-time signals for monaural audio and heading, generates stereo sound which appears to come from a position in space. The use of advanced compile-time transformations and optimizations made it possible to execute code written in a clear style without efficiency or architectural concerns on the target device, while meeting strict existing time and memory constraints. The final executable compares very well with a similar implementation written in C. We believe that this case is representative of a wider class of common pervasive computing applications, and that the techniques we show here can be put to good use in a range of scenarios. This points to the possibility of applying high-level languages, with their associated flexibility, conciseness, ability to be automatically parallelized, sophisticated compile-time tools for analysis and verification, etc., to the embedded systems field without paying an unnecessary performance penalty.

1.4 Main Contributions

The main contributions of this thesis are enumerated below.

- The main contribution in this work is a compilation framework for executing Prolog code compiled by mixing interpreted, bytecode emulated, native code, and optimized native code:
  - It is designed to *accept different kinds of high-level information*, typically obtained via automatic analysis of the initial Prolog program and expressed in a standardized assertion language.

  - It uses high-level information, which models how the program behave at runtime, to *optimize* the resulting C code, which is then processed by an *off-the-shelf* C compiler.
  - It is based on translation process that essentially mimics the *unfolding of a bytecode emulator* with respect to the particular bytecode corresponding to the Prolog program.
  - A *flexible design* of the instructions and their lower-level components, which allows reusing a sizable amount of the bytecode emulator machinery.

  Another contribution for this framework is a performance report of programs compiled by using bytecode and native code (with and without analysis information).

- Contributions to automatize the building process of abstract machines:
  - A systematic approach to the automatic generation of implementations of abstract machines.
  - Separation of the abstract machine definition in different parts: instruction set, and the representation of internal data and bytecode representation.
  - Alternative versions of the abstract machine are therefore easier to produce, and variants of their implementation can be created mechanically. This even allows generating implementations tailored to a particular context.
  - We illustrate the practicality of the approach by reporting on an implementation of a generator of WAMs which are specialized for executing a particular fixed (set of) program(s). The experimental results show that the approach is effective in reducing emulator size.

- Contributions to the use of a Prolog derivate as the language to describe the instruction set of a Prolog abstract machine:
  - Design of a Prolog derivate that can describe the most basic components of an efficient virtual machine (in this case, for Prolog).

  - A compilation algorithm that can generate C code from that language and build a complete bytecode emulator.
  - Thanks to the high level of the language used and its closeness to Prolog the abstract machine descriptions can be manipulated using standard Prolog compilation and optimization techniques with relative ease.
  - We also show how, by applying program transformations selectively, we obtain abstract machine implementations whose performance can match and even exceed that of highly-tuned, hand-crafted emulators.

- Contributions to the use of a Prolog derivate as the language to describe the basic data types of a Prolog abstract machine:
  - *Annotations* in this language make it possible to impose constraints on the final representation of tagged words, such as the effectively addressable space (fixing, for example, the word size of the target processor / architecture), the layout of the tag and value bits inside the tagged word, and how the basic operations are implemented.
  - We evaluate a large number of combinations of the different parameters in two scenarios: a) trying to obtain an optimal general-purpose abstract machine and b) automatically generating a specially-tuned abstract machine for a particular program.
  - Results that indicate that we are able to automatically generate code featuring all the optimizations present in a hand-written, highly-optimized abstract machine and we can also obtain emulators with larger addressable space and better performance.

- Contributions to the study of the feasibility of high-level, general-purpose logic language in the design and implementation of applications targeting wearable computers:
  - The use of advanced compile-time transformations and optimizations in a "sound spatializer" which, given real-time signals for monaural audio and heading, generates stereo sound which appears to come from a position in space.

  - Experimental tests that show that the final executable compares favorably with a similar implementation written in C.

All of these results has been published and presented at international conferences with peer-to-peer review. Some of these contributions have been made in collaboration with other researchers:

- Chapter 3 is mainly based on [MCH04], co-authored with Manuel Carro, and Manuel Hermenegildo, and was published and presented at the 6$^{\text{th}}$ International Symposium on Practical Aspects of Declarative Languages (PADL 2004). Minor parts of the chapter are based on optimizations presented in [CMM^{+}06] (on which is based the Chapter 7, as explained below).

- Chapter 4 is based on [MCPH05], written jointly with Manuel Carro, Germán Puebla, and Manuel Hermenegildo. It was published and presented at the 21$^{\text{st}}$ International Conference on Logic Programming (ICLP 2005).

- Chapter 5 is based on [MCH07] and an extended version [MCH09, MCH10]. The first paper is co-authored with Manuel Carro and Manuel Hermenegildo, and has been published and presented at the 2006 International Symposium on Logic-Based Program Synthesis and Transformations (LOPSTR 2006).

- Chapter 6 is based on [MCH08], written jointly with Manuel Carro and Manuel Hermenegildo, and published and presented at the 10$^{\text{th}}$ ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'08).

- Chapter 7 is based on [CMM^{+}06], co-authored with Manuel Carro, Henk L. Muller, Germán Puebla, and Manuel Hermenegildo. It was published in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) in 2006.

2

Sequential Implementation and Optimization of Prolog: a (Short) Survey

Summary

An interesting classification for Prolog, extensible to many other languages, divides the implementations in interpreters, compilers to *bytecode*, and compilers to a lower-level language. In this chapter we will have a quick historical view from the first interpreters to the birth of the Warren's Abstract Machine to alternative designs and advanced implementation and optimization techniques, for both bytecode emulation, compilation to native code, and hybrid compilation schemes that combine both techniques.

2.1 Introduction

Many of the advances that appeared in the area of sequential Prolog execution, are interesting not only for Prolog, but also for the implementation of other logic and non-logic languages, and are compatible with other efforts focused on parallel execution [San00] or improved expressivity (such as tabling [War92]). To completely understand the design and optimizations of modern Prolog systems we need to take into account that language implementations are full of subtle details, sometimes stemming from of a long process of trial and error, with many decisions based on the uses of the language but also on the evolution of the available hardware.

We will briefly review in this chapter the state of the art in sequential Prolog implementations. It differs from existing surveys (such as [Van94]) in that it takes into account recent contributions and advances. This chapter is organized as follows: first, we quickly review the transition from the first interpreters to the conception of an abstract machine (the WAM) that underlies most Prolog systems, and we will describe some alternative designs (Section 2.2). Then, improvements for the WAM will be presented (Section 2.3), which are in principle independent from whether the abstract machine is emulated or taken as reference for native code generation. The rest chapter closes with the description of some details related to WAM implementation. First, for WAM-based high-efficiency bytecode emulators (Section 2.4), then for native code generators (Section 2.5), and finally for the combination of both approaches (Section 2.6). All of them where experimentally shown to positively impact on the performance of whole systems.

2.2 From Early Interpreters to the WAM

Kowalski was one of the first proposing that a set of first order rules and axioms has a procedural interpretation under a fixed proof procedure. In 1972, that proposal became reality as the first Prolog interpreters, written in Algol-W and Fortran by Colmerauer and Roussel [Col93]. Those early systems were still a long way from what is now the state-of-the-art, mainly due to technological limitations of the time, and because many decisions had to be taken not only about the best way to encode and evaluate programs, but also on how to represent data and terms. The fact that they were *interpreted* systems — by definition, programs that evaluates other programs with almost no involved preprocessing — was not an issue for early systems. Notwithstanding, they did major contributions in the memory management techniques for Prolog and the design of efficient abstract-machine based systems.

From the proposals that followed the interpreting approach, one of the most successful implementations of Prolog was the C-Prolog intepreter [Per87], implemented by Fernando Pereira in 1982 and based on an earlier interpreter written in IMP for the EMAS operating system by Luis Damas. At that time, Prolog was already recognized as powerful, simple, clear, and readable programming language,

especially suitable for high-level symbolic programming tasks. The interpreter was written in C and designed for 32-bit machines with a large address space (for the time). The full system, based on Edinburgh DEC-10 Prolog, contained the interpreter and a wide range of builtins. The major advantage of C-Prolog was its portability: it was available for UNIX and VAX/VMS operating systems and was easy to port to other systems.

2.2.1 Abstract Machines: from the PLM to the WAM

In 1977, with the help of Fernando Pereira and Luis Pereira, David H. D. Warren wrote DEC-10 Prolog, the first Prolog compiler [War77]. Although DEC-10 Prolog was designed to generate DEC-10 assembly code, the main difference with previous implementations being that Warren had in mind an *abstract* machine to execute Prolog programs, which was called the Prolog Machine (PLM).[1] [2]

The representation technique for terms was devised by Boyer and Moore in [BM72], *structure-sharing*, where the fixed part of a term is separated from the variable part. DEC-10 Prolog supported mode declarations, first argument indexing, and its performance was similar to Lisp systems of the day. It was a very influential system, since its syntax became the de-facto *Edinburgh* standard. By 1980, it was the first system to have a heap garbage collector and last call optimization.

In 1980, Bruynooghe [Bru82] and Mellish [Mel82] devised techniques for improving the performance of Prolog execution. Their proposal consisted in abandoning the idea of sharing structures, and using structure copying, since it increases locality of reference and simplifies memory management. Later, a machine with only seven instructions and using structure copying was proposed by Bowen, Byrd, and Mellish [Bow83]. These techniques, along with tail recursion optimization, where integrated by David H. D. Warren in a technical report describing what became the Warren Abstract Machine [War83, AK91]. With a

[1] Not to be confused with the Berkeley PLM microprogrammed microprocessor, a hardware implementation of the yet to be invented WAM.

[2] Abstract machines has become one of the most successful techniques when efficently implementing programming languages. Many popular languages such as Java and C#, or scripting languages like JavaScript, Python, or Ruby, are executed by means of bytecode compilation, or use it as an intermediate at some point in the compilation chain to generate native code.

design cleaner than the PLM,[3] the WAM became the foundation for many Prolog implementations and is still present in the design of many highly optimized systems [Swe99, Qui86, SCDRA00, BCC+02].

2.2.2 The WAM at a Glance

Intuitively, the operational semantics that the WAM describes reminds that of a procedural language where assignment has been replaced by unification of terms and conditional branching by backtracking. In its basic form, it contains many subtle optimizations. The emphasis of the default optimizations proposed in the WAM, and many others that appeared after the original design, are either reducing the memory required to execute Prolog programs, reducing the evaluation time, or both. A detailed description can be found in [AK91]. We will see a brief description of the basic architecture of WAM-based emulators.

Term Representation: Terms are the most important data type in the WAM. They are represented as *tagged* words, each word containing a *tag* field and a *value*. The tag field allows discriminating at runtime the type of the value. Although many kinds of tagged words may be defined, they can be reduced to atomic values (such as numbers or constants), compound terms, and variables. The value field is usually the value itself (the number or index to a symbol table) in atomic values. In the other cases, like compound terms, the value field is interpreted as a reference that points sequence of tagged words containing the name/arity and the arguments. Unbound variables are implemented as self-referential pointers, so that when two variables are unified, one of them is modified to point to the other. Therefore, to access the value a variable is bound to, it is necessary to follow a chain of pointers.

Memory Areas: Some registers and memory areas are common for register-based machines for imperative languages and the WAM. There are a set of temporal registers to pass arguments or store temporal values, a local stack to store the enviroments (the local frames or activation records in standard compiler terminology) to implement recursive procedure/predicate calls (and store variables

[3] A detailed description of the PLM and influence in the design of the WAM is explained with great detail in [Hod90]

that must survive across calls), and a global heap (to store data that survives the exit from a procedure/predicate). In order to implement backtracking efficiently, the WAM introduces some additions and modifications:

- The heap is able to keep a chronological order of the objects it holds, and there is special register points to the top of the heap.
- The combination of a *choice point* stack and a *trail* stack keeps track of saved execution states.[4]

The trail is used to store the address of every unbound variables that are unified, and the choice stack is filled special enviroments called *choice points*. Choice points are created whenever a predicate with more than one clause (more than one execution alternative), to save the current execution state, by storing pointers to the top of each stack (the heap, local stack, and trail), the argument registers, and the pointer to the next execution alternative. Choice points also protects frame environments of non-deterministic predicates from being overwritten: when computing the top of the frame stack to create a new environment, the maximum of the address of the latest active frame and frame stored in the latest choice point are considered. When a goal fails (e.g. unification fails) the execution is recovered from the latest choice point in the choice point stack. Backtracking to a saved state involves several operations: *undoing* the variable bindings (*untrailing*), restoring the pointer to the current frame, restoring the rest of the saved registers, and patching the choice point for the next alternative, or removing it if no alternatives are left for the predicate. Setting the top of the heap to a previous value automatically reclaims memory on backtracking.

Instruction Set: In its basic form, the WAM defines a reduced set of instructions, dealing with each of the emulator operations. The instructions represent quite high-level operations that can easily be executed by general purpose machines. For illustration purposes we enumerate a simplified instruction set below, omitting special optimized instructions that will be commented on later:

[4]Choice stacks and local stack may live in the same region, growing in opposite directions, or be different stacks. Alternatively, a single stack may contain both choices and frame enviroments, but this approach consumes more memory.

- Term creation and unification: passing goal arguments (put_variable V_n R_i, put_value V_n R_i, put_constant C R_i, put_structure F/N R_i), unifying head arguments (get_variable V_n R_i, get_value V_n R_i, get_constant C R_i, get_structure F/N R_i), unifying structure arguments (unify_variable V_n, unify_value V_n, unify_constant C V_n, unify_void N).

- Predicate call and environments: calling predicates (call P N), returning successfully from a predicate (proceed), creating or destroying local frames (allocate, deallocate)

- Choice point management (try, retry, trust).

Optimizations: The WAM introduces very relevant optimizations that play a significant role in minimizing the memory consumption and execution time of many programs by orders of magnitude. Two of the most important are:

- *Indexing*: Prolog backtracking explores all the clauses of a predicate are in the order they appear in the code, even if some of them are mutually exclusive. Indexing builds a tree based on the type of the arguments (usually, *hash* tables for the first argument value, up to the principal functor) to discard clauses that are known to fail for the current calling pattern. Predicate entries are modified with special instructions that jump to the appropiate code for each clause (or list of clauses). This very important optimization not only reduces memory usage (e.g. avoiding the creation of choice points for calling patterns with which only one clause is known to be compatible), but it also avoids linear search of alternatives in many cases.

- *Tail recursion optimization*: Another powerful optimization is the last-call optimization, that removes the last local frame in tail-calls (that allows using recursive predicates as loops with constant memory usage) before jumping to the goal code.

In addition to the previous optimizations, other minor improvements — which depending on the program may suppose a constant-factor improvement in speed and memory savings — are specified in the WAM design:

- *Specialized Terms for Lists*: Lists (e.g. `[1,2,3]` or `[1|[2|[3|[]]]]`) are data structure so common in many programs that many emulators use a specialized representation for the list constructor (`[_|_]`), reserves a tag for that purpose, and specialized instructions (put_nil R_i, put_list R_i, get_nil R_i, get_list R_i, unify_nil).

- *Unsafe Variables:* Unbound variables can be created in the frame stack. Those variables are called *unsafe*, since they may lead to dangling pointers. Unifying potentially unsafe variables with structure arguments or goal arguments, is performed with special unify instructions that *globalizes* (moves to the heap) the unsafe variables.

- *Conditional Trailing*: Trailing is only done for variables that are older than the most recent choice point. Variables newer than the choice point will be automatically recovered from the memory on backtracking, which makes trailing them superfluous. This optimization saves both time and memory, at the cost of a *trailing check*.

2.2.3 Alternatives to the WAM

The WAM defines an architecure for executing Prolog with good performance. However, as it will be seen in the following sections, there exists many variations that lead to even better efficiency. We select here some major departures from the WAM that, although may share many parts like term representation, deserve being classified in their own section.

TOAM: The temporal registers in WAM-based emulators are used to pass arguments in predicate invocation or to store temporary results. Despite being called registers, they are usually mapped to memory in many implementations for current architectures Neng-Fa Zhou proposed in his abstract machine, the ATOAM (yet Another Tree-Oriented Abstract Machine) [Zho94], that saving arguments directly in the stacks, would simplify many operations. In [CZ07], this design is refined towards the TOAM Jr. (the current machine for B-Prolog), that employs no registers for temporary data and offers variable-size instructions for encoding unification and procedure calls. The main difference of the TOAM from the WAM is that it is a stack machine instead of a register machine. Arguments

are passed through the stack in a frame, and one frame is used per procedure call. This has the inconvenience that effective tail-recursion, which keeps the stack usage, is harder to implement in the TOAM. To solve this problem, it implemented several optimizations to do enviroment reuse.

The WAM cannot reuse enviroments, but it can avoid creating them. Environment avoidance is useful not only for *leaf procedures* (e.g. `inc(X,Y) :- Y is X + 1`), but also for clauses with only built-ins one last call (e.g. `foo(X) :- Y is X + 1, bar(Y)`). Whether the two techniques could be merged was studied for a limited set of benchmarks by Bart Demoen and Phoung-Lan Nguyen [DN08]. In that paper, it is shown that enviroment reuse is useful for deterministic recursive predicates such as `tak/4`, and that by emitting precisely fine-tuned bytecode, it is possible to implement enviroment reuse in a register-based WAM.

The Vienna Abstract Machine (VAM): In [KNW90] an alternative to the inference in the WAM is presented. In contrast to the WAM, inference in the VAM is performed by unifying a goal and a head immediately. A detailed comparison between the WAM and the VAM is described in [Kra94]. Two implementations of the VAM existed. $\text{VAM}_{2\text{P}}$ required two instruction pointers, one for the goal code and other for the head code. At execution, $\text{VAM}_{2\text{P}}$ fetches one instruction from the goal code and one instruction from the head code and executes the combined instruction. More optimizations were possible since more information was available at the same time. An alternative $\text{VAM}_{1\text{P}}$ implementation was designed for native code compilation, by unfolding goal calls. The main drawback of that approach is the cost of potentially multiplying the code size. Notwithstanding, the fact that many of the advantages of the VAM could be possibly be obtained statically by source-to-source optimizations applied at the Prolog level [PH03, Win92], and implementation issues which make optimizations difficult (such as taking into account two instruction opcodes at the same time), may be a possible reason for not seeing VAM-related optimizations in modern Prolog implementations.

2.3 Further Optimizing the WAM

Other optimizations which are independent of whether bytecode or native code is generated are possible.

Different Term Representations: In WAM-based systems, there are many chooses that can be taken when representing terms. In [DN00], four different term representations are tested:

- *wam-vars*: Allow unbound variables in local stack (that is, the *unsafe* variable optimization).
- *heap-vars*: Unbound variables are always initialized in the heap.
- *parma-vars*: Use PARMA-style [Tay89] binding between variables (unifying two variables create a cycle; although binding follows all variables in the cycle and trailing needs two words, the cost of dereferencing is constant).
- *tag-on-data*: Tag-less pointers, a full word sized tag resides on data like in [TN94] (but without term compression).

The results indicated that all of them were interesting in some cases, for performance or flexibility. However, performance variations and the set benchmarks in the paper are not enough to make final statements in large programs, except the expected increase in heap usage for *heap-vars* (due to unbound variables), *tag-on-data* (due to larger tags), and in trail usage for *parma-vars*. In [SD02], it is proposed an improved trailing scheme that reduces trail usage in 50% for unconditional variable-variable trailing. Together with an analysis and optimization technique it considerably reduces trail usage for some programs.

Term Matching/Building: In the WAM, the translation to instructions of unification goals and head arguments (e.g. `p(f([1,2], a, g(b))) :- Body`) is done as a flattening to elementary unifications (e.g. `p(A1) :- A1 = f(X1, a, X2), X1 = [1,2], X2 = g(b), Body`), which is translated to `get_`$*$ instructions for the principal functor, followed by `unify_`$*$ instructions for the arguments. The *get* instruction switches the WAM to either *read* or *write* mode. Depending on the mode, the instructions build new term or match against existing ones.

Schemes for generating optimal structures are presented in [MD91], along with a description of several optimizations for minimizing the cost of checking the read/write flag (i.e. two-stream unification, two emulator loops).

Shallow Backtracking: Backtracking is a fundamental Prolog mechanism to implement search, but also conditional execution (*if-then-else* is defined as definition of disjunct alternative clauses). In [Car89], Mats Carlsson re-introduced an optimization called *shallow* backtracking, that is still present in SICStus [Swe99] and Ciao [HBC+08]. Unlike in the WAM, the PLM considered a distinction between two kind of backtracking, depending on when the failure happens:

- Failure in unification of the head of a clause (or the first unifications before execution of predicates or some builtins).
- Failure in deeper parts of the body.

The idea is to make the `try` and `try_me_else` instructions save only a small part of the machine state, and postpone the completion of the choicepoint to the execution of a `neck` instruction. The results in the paper showed that the speedup gained by this optimisation was significant (7%-15%).

Dynamic indexing: First-argument indexing is implemented in most Prolog systems (and part of the original WAM definition). However, programs have to be modified to take advantage of indexing. As logic programming applications grow in size, or larger data sets are used (for some applications), first-argument indexing is not appropiate. Indexing on multiple arguments is an alternative, but it is is unpractical due to the size of the indexing trees that can result. In [SCSL07a] a proposal for building the indexing tree on demand at runtime is shown. As a first step towards runtime optimization of Prolog programs, the technique obtains considerable performance speed-ups (from a few percent to orders of magnitude) and has the advantage of being much less costly than creating the full multi-argument indexes.

Variable Shunting during Garbage Collection: The WAM garbage collector can be extended to shunt (short-circuit) chains of bound variables [SC91]. That saves space, by making it possible to allocate intermediate cells, and speeds

up code since dereferencing those cells is cheaper. This optimization is implemented in SICStus and Ciao Prolog. Variants of variable shunting have been implemented in Yap [CC01].

2.4 Advanced Emulator Implementations

While some WAM-based systems were actually implemented directly in hardware, the most successful implementations are based on *bytecode* emulators, where the *bytecode* is an binary encoding of the abstract machine instructions. A clear advantage of emulators is its portability and compact code representation. Bytecode emulators, most of them written in low-level languages like C[5], are usually as portable as interpreters and offer very good performance and reduced program size, a crucial issue for very large programs and symbolic data sets. Basically, emulators are built as loops that run continously fetching bytecode instructions and executing them. We will comment (mostly) on portable implementation techniques (without remodeling the abstract machine) that have successfully helped in improving the performance of emulators. Although many of them appeared in other systems, Yap Prolog was one of the first systems combining most of them, as reported in [SCDRA00].

2.4.1 Mapping Hardware and WAM Registers

In emulators written in C, it is possible to hint C compiler for better register assignment than what it is able to do automatically, based on knowledge about the control flow inside the emulator itself. Usually, this optimization consists of directly mapping hardware registers (which in some architectures may be limited) to WAM registers; for example, the WAM program counter is a clear candidate. This is even more important if WAM registers are declared as global variables or fields of a WAM state structure. A related optimization consists of using the C *register* modifier, which is used as a hint by the C compiler to keep the variable in a register when possible. Note that reserving a fixed register for all the emulation loop may affect negatively the performance. A common solution

[5] A notable exception is Quintus Prolog implemented in a generic RTL language ("PROGOL").

is keeping the WAM register in a normal variable and *caching* in a local variable of the emulation loop for reasonable sections of code.

2.4.2 Cheaper Instruction Fetching

Instruction fetching is another source of overhead for bytecode emulators. The following techniques alleviate it:

Threaded Code Emulation: The opcode field in instructions is replaced by a pointer to the actual machine code which implements the instruction. Executing an instruction thus fetches the opcode field and just jumps to it.

Threaded-operands: The encoding of a bytecode instruction with an arbitrary temporal X register as opcode, is usually represented as a number. For example, the X_i register can be represented as `i`. During instruction decoding, `i` is read and the actual C variable containg the register is expressable as `X[i]`. It is also possible to encode directly in the opcode a relative or absolute address of the register (e.g. patch `&X[i]` for every X operand with value `i` in bytecode), which depending on the architecture may save some assembler instructions and cycles.

Prefetching: Threaded code removes the overhead of a `switch` statement, but it still generates a CPU stall. To jump to the next instruction, the program counter must be incremented, the new opcode fetched, the CPU will stall waiting for the opcode, and then the jump will be done once the data has arrived. To avoid this problem, the new opcode can be prefetched in a hardware register allowing the CPU to know the jump address in advance.

2.4.3 Augmented Instruction Set

The WAM defines a very simple instruction set that may however be counterproductive in bytecode emulators. Many optimizations can be applied by enriching the emulator with special cases or instructions that group complex operations.

Instruction Specialization: When some information is known at compile time about some of the instruction operands, it is possible to choose a specialized instruction. Although this technique is more useful for native compilation (since the number of specialized instruction may be very large), many instructions in modern systems are just specializations of more general cases (e.g. `put_nil` is a special case of `put_constant` C, where C is `[]`).

Built-ins as Instructions: Some very common predicates, like meta-predicates, integer arithmetic, unification, term manipulation built-ins (e.g. `functor/3`, `arg/3`) can appear directly as instructions. Not only this may skip the creation of frame environments but built-ins are special predicates that do not always require a complete frame or choice point.

Instruction Merging: By joining several instructions into a single one, the emulation overheads are reduced since only one instruction fetching is required. In [NCS01] may combinations and specialization of WAM instructions (done by hand) is tested against standard benchmarks for Prolog. In the context of the SICStus emulator, they found a 10% gain, while the bytecode size reduction was about 15%. In the TOAM Jr. [CZ07] not only parameters are passed through the stacks, but it also temporary results. This makes common instruction sequences more frequent and merging of long instruction sequences easier. The introduction of merging in B-Prolog version 7.0 made the code more compact and showed significant (between 48% and 77%, depending in the architecture) speedups. Note that merging also plays an important role on the optimizations made by the C compiler. For merged instructions, the C compiler can propagate information between two pieces of code that are now adjacent and optimize them (e.g. reuse temporal results).

2.5 Compilers to Native and Low-Level Code

In bytecode emulators, a major hurdle when implementing the semantics of the abstract machine instructions is minimizing the emulation costs. In general, source code is translated to bytecode and bytecode is executed by the emulator. As seen in Section 2.4.3, more efficiency can be obtained by reducing the costs of

emulation, but depending on the application requirements, the gain may not be enough.

A different approach that tries to achieve more efficiency than emulation is native compilation, which is based on a different principle: the Prolog source is translated to an intermediate code (equivalent to the abstract machine instructions, but which may be of different granularity than that of WAM emulators), and that code is translated to native code (which is platform-specific and directly executable) or a low-level language for which native code compilers are available. Performance is potentially enhanced, since no emulation component is involved, at the cost of sacrifying not only portability but also the flexibility of bytecode emulation (for example, dynamic patching of bytecode is not possible or very complicated, by e.g. performing native code recompilation at runtime or patching assembler code).

Depending on how the translation of the WAM (or other machine) to the *host* language (assembly code or a low-level language) is performed, we will distinguish two cases:

Direct translation to native code: the entire WAM (term representation, stacks, control mechanism) is fully implemented. That often requires quite low-level operations.

Indirect translation through a lower level language: emitted code emulates the desired operational semantics, although it uses some indirect path (e.g. passing continuations) that would not be necessary in a *direct* translation. This approach is suitable when targeting imperative languages like C or Java bytecode. The reasons for taking this approach are grounded on portability issues (e.g. adhering to C standard) or taking advantage of the target language compiler (e.g. C compiler optimizations that should be rewritten if assembler code is emitted by hand).

Native *direct* compilation of Prolog has been used since the early times of the DECsystem-10 Prolog, and in other systems like SICStus (only for the Sparc architecture). However, the symbolic nature of Prolog, advances in emulation, and hardware changes makes the idea of plain compilation obsolete in most cases. For example, performance evaluations showed that well-tuned emulators can beat for some programs Prolog compilers which generate machine code directly but which

do not perform extensive optimization [DC01]. However, the strongest point of native compilation and its main advantage over emulators is that different, fine-grain optimizations can be applied to each chunk of code depending on its usage. We will see some of the more effective proposals.

2.5.1 Global Static Analysis

Global analysis statically (i.e. without running the program) infers program information related to the posssible program executions. This information can be used to simplify optimize programs, by e.g. simplifying general execution mechanisms to better suit the situation at hand. In [HWD92], the issue of practicality of global flow analysis in logic programing is addressed, concluding that the overhead of global flow analysis, quite precise and useful, are not prohibitive in terms of speed of analysis.[6] Other works yielded preliminary results to sustain the idea, like the VAM_{AI} abstract machine for abstract interpretation of Prolog [KB95], that indicated that dataflow analysis could be performed very efficiently. So far, abstract interpretation has proved to be a very precise technique for global flow analysis of Prolog and existing analyzers [HPBG05] and systems use abstract interpretation to implement and drive code optimizations.

However, taking advantage of those optimizations requires either a large amount of specialized instructions or fine-grained instructions that can be simplified individually. The WAM-based instruction sets employed in emulators are not the best candidates as target for an optimizing compiler.

2.5.2 Low-granularity Instruction Set

One of the first systems to exploit dataflow analysis in compilation was Aquarius Prolog [VR90]. By using global analysis, it derived type information, to specialize unifications, and determininism information to optimize control structures like conditional branches.

Since, as we said before, the high-level compact instructions in the WAM were too coarse grained to do bytecode-based analysis and optimizations, Van Roy used a system derived from the WAM, the BAM or Berkeley Abstract Machine, that retained good features of the WAM, but was more easily optimized and closer

[6] It was applied in reducing run-time checks in independent and-parallelism.

to the actual underlying physical machine, since its instruction granularity was smaller.

In [Deb92] a low-granulary WAM for low-level optimizations is proposed, that can be used to express low-level optimizations that reduce tag manipulation, dereferencing, trail testing, environment allocation, redundant bound checks. That work is presented as a source-to-source solution to for WAM optimization.

Doing native code compilation, many specializations were possible. Unification in Prolog is the general mechanism to implement simpler operations in other languages, like pattern-matching, parameter passing, assigning values to variables, allocate memory, and do conditional branching. Van Roy identified how global information could be used to simplify unification and gain performance, trying to specialize unification to the simplest possible code. Another system that used similar techniques was Parma Prolog [Tay91a], that targeted the MIPS architecture. Both systems showed promising results for some programs, exceeding the performance of previous Prolog implementations by an order of magnitude.

2.5.3 WAM-level Multiple Specialization

The WAMCC compiler was extended by M. Ferreira and L. Damas in [FD99] with program specialization, and later refined in [FD02]. the analysis that was done in Prolog to WAM compilation and generating local specialized instructions, and propagating that information through the entire program. It used an abstract interpretation-based local analysis at the predicate level and generated different versions for each detected entry pattern. The poly-variant unfolding of predicates allowed the predicate-level (local) analysis to propagate inter-procedurally relevant information. Code expansion was controlled by limiting the specialization to some (controlled) pertentage of the predicates. The average speedup was of 50% over WAMCC.

2.5.4 Static Typing

Prolog is a dynamically typed language. The WAM uses tag bits to distinguish the type of each term at run-time. It is known that keeping track of tag bits adds some overhead in most architectures (some bits are reserved to store bits, address space may be limited, tag bits have to be removed and inserted, etc.) Tag bits

can be avoided in languages with static polymorphic typechecking [App89], with the only exception of the runtime information required for user-defined variant records, but in most cases this lead to improvements in space and time efficiency.

A way to improve the efficiency of Prolog is by making the language statically typed. That means a drastic departure from the language: since some programs are no longer well-typed, it restricts the definition of valid programs. This constraint has some repercussions for analysis:

- More information at compile time about program behaviour is usualy obtained.
- Analysis is cheaper than for dynamically typed languages (e.g. those based on abstract interpretation).

The most notable current proposal to add static type checking in a logic language with Prolog-style nondeterminism is Mercury.[7]

Mercury: The Mercury language [SHC96] is a declarative logic programming language. Although Mercury differs considerably from Prolog in some aspects (e.g. typing system, goal reordering, etc.), many of its implementation and optimization techniques are applicable also to Prolog. It is statically typed and has mode and determinism annotations. It emits different code for each predicate with different modes and determinism. That restricts the language semantics, but it has a simple execution model that takes advantage of the information available to generate efficient code, significantly faster than other logic programming languages.

Typed Prolog: Using dynamic and static typing in a language is often motivated by the trade-off between the flexibility offered by the former and the correctness properties inferred from the latter. In [SSCWD08] Tom Schrijvers proposed an integration of static and dynamically typed systems for Prolog. It places a portable Hindley-Milner type system on top of dynamically typed Prolog implementations (working for Yap, SWI and ported with minor effects to other systems like Ciao). Typing is not mandatory in that system. Typed and untyped

[7] Other strongly-typed languages in the logic paradigm are Gödel [HL94] or λProlog [NM88].

code can be mixed, [8] and the type checker can insert dynamic type checks at the boundaries between typed and untyped code. Although this extension only concerns code safety, and code optimization has not been attempted, the integration with static Hindley-Milner typing, which is cheaper than abstract interpretation, is interesting.

2.5.5 Back-ends for Native and Low-level Code

As previously said in the beginning of this section, native code compilers for Prolog produce directly executable programs by generating — in what is usally called the compiler *back-end* — native code from an intermediate representation (either specialized WAM instructions, or an instruction set with lower granularity, like the BAM for Aquarius Prolog). A main issue with native compilers is that the output code is architecture-dependent, which makes porting and maintaining a non-trivial task. Some examples of early systems that emitted native code are DECsystem-10 Prolog, Aquarius [VD92], versions of SICStus Prolog [Swe99] for some architectures, and BIM-Prolog [Mar93].

Common Intermediate Representations: Developing back-ends to native code can be simplified by using an intermediate *register transfer language* (RTL), although different translations of this code are needed for different architectures. GNU Prolog [DC01] is a complete Prolog system with many advanced features (such as finite domain constraints) that is implemented in this way. Instead of supporting a rich language able to generate very optimized code, its back-end is limited to a subset of the assembler language. The code for very simple instructions is expanded (e.g. moving data between registers), but for the more complex ones (e.g. unifications), procedure calls are emitted. It results in reduced compilation times and small executables.

Other intermediate languages for the native compilation of Prolog programs is presented in [LSC94], that designs a compiler for Yap Prolog able to maintain both emulated and native code. The goal of that proposal was improving the performance of Prolog using native-code compilation for user-defined predicates,

[8] A similar approach is usualy taken to interface foreign typed code (C) and untyped Prolog code: runtime type checks are mandartory in the boundaries between both languages

and achieve the same (or better) performance than other systems using native code generation (like Aquarius or native SICstus Prolog).

Low-level Languages: The alternative to generating assembler code is using a (lower-level) language, such as, e.g., C-- [JRR99] or C, for which compilers are readily available. Translating to a low-level language such as C is interesting because it makes portability easier, as C compilers exist for most architectures and C is low-level enough as to express a large class of optimizations which cannot be captured solely by means of Prolog-to-Prolog transformations.

The WAMCC [CD95] system, a predecessor of GNU Prolog, translated Prolog to C via the WAM. It was simple, efficient and portable. It did not incorporate complex optimizations, but it was as efficient as emulated Quintus Prolog 2.5 and 30% slower than SICStus running via native code compilation.

A typical translation scheme maps clauses or procedures to C functions. That causes considerable overhead during function call and return and makes last call optimization difficult. In WAMCC, this overhead is alleviated by assembler directives that bypass function calls. Unfortunately, this solution was highly dependant on the specific architecture, needed modifications when being ported, and still suffers from code explosion. It made some delicate assumptions about how the control flow is implemented in assembler, sometimes invalidating analysis and optimizations performed by the C compiler. It required GCC in order to generate native code, which made compilation slower than in its successor GNU Prolog (that provided compilation times between 5 and 10 times faster).

A similar technique is implemented in several Mercury back-ends (the *asm* compilation grades).

Higher-level C: Alternatively, in some logic languages it is possible to avoid the use of non-portable assembler directives while not sacrificing performance, as it is done for Mercury in its higher-level C back-end [HS02]. This back-end is based on continuation passing (forward execution is implemented as tail calls, consecutive statements represent alternatives, and returning from a function implements failure). By using a translation scheme that is not based on language tricks, and where predicate arguments and local variables are mapped to C arguments and variables, the back-end is more portable (to languages other than C),

and the compiler is able to perform more optimizations.

A drawback of relying on mapping source language elements to C (e.g. logic variables to C variables) is that the implementation of some features, like garbage collection (GC), is complicated. Mercury makes use of the external Boehm GC [BW88] to reclaim memory automatically. However, not being connected with any kind of type information, this kind of GC algorithms has the disadvantage of being conservative.[9] Although explicit techniques have been devised to make the GC type-accurate in the case of Mercury [Hen02], other optimizations, for example, fast heap unrolling on backtracking or memory optimizations (like variable shunting in the WAM), are very hard to implement without a tight cooperation with the external GC.

2.6 Hybrid Compilation

Traditionally, WAM-based systems compile all predicates are using the same bytecode instructions. A notable exception are system predicates, which are implemented in the target language (e.g. C), but callable as usual predicates. Most systems allow this heterogeneous mix, but the compilation from Prolog always produces bytecode, and non-bytecode predicates are limited to predicates interfacing foreign functions or built-ins. An interesting approach is the combination of different kind of compilation techniques.

Benefits of Combining Techniques: It has been shown that for some applications, compilation to lower-level code can achieve faster programs by eliminating interpretation overhead and performing lower-level optimizations. This difference gets larger as more sophisticated forms of code analysis are performed as part of the compilation process and instructions — which can be unfolded multiple times — can be highly specialized and optimized, individually. Although compilation to native code does not suffer from emulation overheads and offers a much fine-grained optimization level, sometimes (e.g. when the definition of many instructions is expanded) the executable size can be too large (with pos-

[9] It cannot distinguish the type of each memory cell, e.g. a natural number containing `0x123000` and a pointer referencing the `0x123000` address. It conservatively treates any number as a possible pointer.

sible negative impact on the processor instruction cache), or the optimization opportunities too few (e.g. the inferred information during analysis may be too general and of little help to specialize the code, resulting in limited performance gains not worth a probably increased compilation time). Interpreters in turn have potentially smaller load/compilation times and are often a good solution due to their simplicity when speed is not a priority. Emulators occupy an intermediate point in complexity and cost, specially interesting when not enough information is available about the program — or when the program exhibits such a dynamic behaviour that all the information that can be collected is useless for optimizations.

A major criticism to native code generation for Prolog was presented in [DM92], stating that the performance differences between Aquarius and BIM4 were due to bias toward programs with a strong arithmetic component. For other programs, native code compilation suffers from code explosion problems, since the bytecode of a predicate (specially when containing large terms) is often much more compact than plain translation to C. A better solution can be obtained with a compromise which combines both techniques for selected program parts.

2.6.1 Partial Translation

A general-purpose technique called *partial translation* was presented and studied for translation of Prolog to C translation in [TDBD96]. It compiled selected, potentially large, sequences of the simple emulator instructions to native code, and left to the emulator the complex control structure, some large instructions, and the management of the symbol table. The native code was linked with the emulator itself to produce a stand-alone application, which was able to switch between native and emulated code as required. Its performance ranged between emulated code and native code, while supporting modular compilation, allowing some control of the trade-off between speed and size, and beign fully portable (as long as a C compiler was available).

2.6.2 Exo-compilation

The idea that the implementation of each predicate does not need to be uniform is revisited in [DNSCS07], under the name of *exo-compilation*. It shows that a new compilation scheme for large sets of wide, regular facts, frequent in the context of ILP, is effective in reducing the memory footprint to about one third of the normal WAM compilation without slowdowns. This is a special case when specializing compilation for a particular class of code, where all arguments in the facts are atoms, and queries often introduces lots of void variables, and is very useful in practice for a certain type of programs. It was implemented in the hProlog bytecode emulator , but the idea is applicable to the compilation to C or native code.

3

Compiling Prolog to Native Code

Summary

We describe the current status of and provide performance results for a prototype compiler of Prolog to C, `ciaocc`. `ciaocc` is novel in that it is designed to accept different kinds of high-level information, typically obtained via an automatic analysis of the initial Prolog program and expressed in a standardized language of assertions. This information is used to optimize the resulting C code, which is then processed by an off-the-shelf C compiler. The basic translation process essentially mimics the unfolding of a bytecode emulator with respect to the particular bytecode corresponding to the Prolog program. This is facilitated by a flexible design of the instructions and their lower-level components. This approach allows reusing a sizable amount of the machinery of the bytecode emulator: predicates already written in C, data definitions, memory management routines and areas, etc., as well as mixing emulated bytecode with native code in a relatively straightforward way. We report on the performance of programs compiled by the current version of the system, both with and without analysis information.

3.1 Introduction

It is safe to say that different approaches in compilation (Chapter 2) are useful in different situations and perhaps even for different parts of the same program. The emulator approach can be very useful during development, and in any case for non-performance bound portions of large symbolic data sets and programs.

On the other hand, in order to generate the highest performance code it seems appropriate to perform optimizations at all levels and to eventually translate to machine code. The selection of a language such as C as an intermediate target can offer a good compromise between opportunity for optimization, portability for native code, and interoperability in multi-language applications.

In `ciaocc` we have taken precisely such an approach: we implemented a compilation from Prolog to native code via an intermediate translation to C which optionally uses high-level information to generate optimized C code. Our starting point is the standard version of Ciao Prolog [BCC+02], essentially an emulator-based system of competitive performance. Its abstract machine is an evolution of the &-Prolog abstract machine [HG91], itself a separate branch from early versions (0.5–0.7) of the SICStus Prolog abstract machine.

`ciaocc` adopts the same scheme for memory areas, data tagging, etc. as the original emulator. This facilitates mixing emulated and native code (as done also by SICStus) and has also the important practical advantage that many complex and already existing fragments of C code present in the components of the emulator (builtins, low-level file and stream management, memory management and garbage collection routines, etc.) can be reused by the new compiler. This is important because our intention is not to develop a prototype but a full compiler that can be put into everyday use and developing all those parts again would be unrealistic.

A practical advantage is the availability of high-quality C compilers for most architectures. `ciaocc` differs from other systems which compile Prolog to C in that that the translation includes a scheme to optionally optimize the code using higher-level information available at compile-time regarding determinacy, types, instantiation modes, etc. of the source program.

Maintainability and portability lead us also not to adopt other approaches such as compiling to C--. The goal of C-- is to achieve portable high performance without relinquishing control over low-level details, which is of course very desirable. However, the associated tools do not seem to be presently mature enough as to be used for a compiler in production status within the near future, and not even to be used as base for a research prototype in their present stage. Future portability will also depend on the existence of back-ends for a range of architectures. We, however, are quite confident that the backend which now generates C

code could be adapted to generate C-- (or other low-level languages) without too many problems.

The high-level information, which is assumed expressed by means of the powerful and well-defined assertion language of [PBH00a], is inferred by automatic global analysis tools. In our system we take advantage of the availability of relatively mature tools for this purpose within the Ciao environment, and, in particular the preprocessor, CiaoPP [HPBG05]. Alternatively, such assertions can also be simply provided by the programmer.

Our approach is thus different from, for example, `wamcc`, which also generated C, but which did not use extensive analysis information and used low-level tricks which in practice tied it to a particular C compiler, `gcc`. Aquarius [VD92] and Parma [Tay90] used analysis information at several compilation stages, but they generated directly machine code, and it has proved difficult to port and maintain them. Notwithstanding, they were landmark contributions that proved the power of using global information in a Prolog compiler.

A drawback of putting more burden on the compiler is that compile times and compiler complexity grow, specially in the global analysis phase. While this can turn out to be a problem in extreme cases, incremental analysis in combination with a suitable module system [CH00] can result in very reasonable analysis times in practice.[1] Moreover, global analysis is not mandatory in `ciaocc` and can be reserved for the phase of generating the final, "production" executable. We expect that, as the system matures, `ciaocc` itself (now in a prototype stage) will not be slower than a Prolog-to-bytecode compiler.

3.2 Basic Compilation Scheme

The compilation process starts with a preprocessing phase which normalizes clauses (i.e., aliasing and structure unification is removed from the head), and expands disjunctions, negations and if-then-else constructs. It also unfolds calls to `is/2` when possible into calls to simpler arithmetic predicates, replaces the cut by calls to the lower-level predicates `metachoice/1` (which stores in its argument the address of the current choicepoint) and `metacut/1` (which performs a cut to the choicepoint whose address is passed in its argument), and performs a sim-

[1] See [HPBG05] and its references for reports on analysis times of CiaoPP.

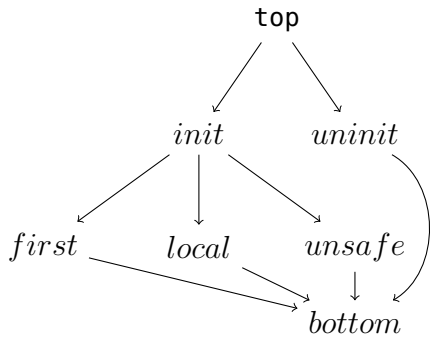

Figure 3.1: Lattice of WAM types.

ple, local analysis which gathers information about the type and freeness state of variables.[2] Having this analysis in the compiler (in addition to the analyses performed by the preprocessor) improves the code even if no external information is available. The compiler then translates this normalized version of Prolog to WAM-based instructions (at this point the same ones used by the Ciao emulator), and then it splits these WAM instructions into an intermediate low level code and performs the final translation to C.

3.2.1 Typing WAM Instructions

WAM instructions dealing with data are handled internally using an enriched representation which encodes the possible instantiation state of their arguments. This allows using original type information, and also generating and propagating lower-level information regarding the type (i.e., from the point of view of the tags of the abstract machine) and instantiation/initialization state of the variables (which is not seen at a higher level). Unification instructions are represented as $\langle TypeX,\ X\rangle = \langle TypeY,\ Y\rangle$, where $TypeX$ and $TypeY$ refer to the classification of WAM-level types (see Figure 3.1), and X and Y refer to variables, which may be later stored as WAM X or Y registers or directly passed on as C function arguments. *init* and *uninit* correspond to initialized (i.e., free) and uninitialized

[2] In general, the types used throughout this chapter are *instantiation types*, i.e., they have mode information built in (see [PBH00a] for a more complete discussion of this issue). *Freeness of variables* distinguishes between free variables and the *top* type, "term", which includes any term.

put_variable(I,J)	≡	⟨uninit,I⟩ = ⟨uninit,J⟩
put_value(I,J)	≡	⟨init,I⟩ = ⟨uninit,J⟩
get_variable(I,J)	≡	⟨uninit,I⟩ = ⟨init,J⟩
get_value(I,J)	≡	⟨init,I⟩ = ⟨init,J⟩

unify_variable(I[, J])	≡	`if` (initialized(J)) `then`
		⟨uninit,I⟩ = ⟨init,J⟩
		`else`
		⟨uninit,I⟩ = ⟨uninit,J⟩

unify_value(I[, J])	≡	`if` (initialized(J)) `then`
		⟨init,I⟩ = ⟨init,J⟩
		`else`
		⟨init,I⟩ = ⟨uninit,J⟩

Table 3.1: Representation of some WAM unification instructions with types.

variable cells. *First*, *local*, and *unsafe* classify the status of the variables according to where they appear in a clause.

Table 3.1 summarizes the aforementioned representation for some selected cases. The registers taken as arguments are the temporary registers *x(I)*, the stack variables *y(I)*, and the register for structure arguments *n(I)*. The last one can be seen as the second argument, implicit in the *unify_ ** WAM instructions. A number of other temporal registers are available, and used, for example, to hold intermediate results from expression evaluation. **_constant*, **_nil*, **_list* and **_structure* instructions are represented similarly. Only *x(·)* variables are created in an uninitialized state, and they are initialized on demand (in particular, when calling another predicate which may overwrite the registers and in the points where garbage collection can start). This representation is more uniform than the traditional WAM instructions, and as more information is known about the variables, the associated (low level) types can be refined and more specific code generated. Using a richer lattice and initial information (Section 3.3), a more descriptive intermediate code can be generated and used in the back-end.

```
while (code != NULL) {
  code = ((Continuation (*)(State *))code)(state);
}
```

```
Continuation foo(State *state) {
  ...
  state->cont = &foo_cont;
  return &bar;
}

Continuation foo_cont(State *state) {
  ...
  return state->cont;
}
```

Figure 3.2: The C execution loop and block scheme.

3.2.2 Generation of the Intermediate Low Level Code

WAM-like control and data instructions (Table 3.2) are then split into simpler ones (Table 3.3) (of a level similar to that of the BAM [VR90]) which are more suitable for optimizations, and which simplify the final code generation. The *Type* argument in the unification instructions reflects the type of the their arguments: for example, in the instruction *bind*, *Type* is used to specify if the arguments contain a variable or not. For the unification of structures, write and read modes are avoided by using a two-stream scheme [Van94] which is implicit in the unification instructions in Table 3.1 and later translated into the required series of assignments and jump instructions (*jump*, *cjump*) in Table 3.2. The WAM instructions *switch_on_term*, *switch_on_cons* and *switch_on_functor* are also included, although the C back-end does not exploit them fully at the moment, resorting to a linear search in some cases. A more efficient indexing mechanism will be implemented in the near future.

Builtins return an exit state which is used to decide whether to backtrack or not. Determinism information, if available, is passed on through this stage and used when compiling with optimizations (see Section 3.3).

Choice, stack and heap management instructions	
no_ choice	Mark that there is no alternative
push_ choice(Arity)	Create a choicepoint
recover_ choice(Arity)	Restore the state stored in a choicepoint
last_ choice(Arity)	Restore state and discard latest choice point
complete_ choice(Arity)	Complete the choice point
cut_ choice(Chp)	Cut to a given choice point
push_ frame	Allocate a frame on top of the stack
complete_ frame(FrameSize)	Complete the stack frame
modify_ frame(NewSize)	Change the size of the frame
pop_ frame	Deallocate the last frame
recover_ frame	Recover after returning from a call
ensure_ heap(Amount, Arity)	Ensure that enough heap is allocated.
Unification	
load(X, Type)	Load X with a term
trail_ if_ conditional(A)	Trail if A is a conditional variable
bind(TypeX, X, TypeY, Y)	Bind X and Y
read(Type, X)	Begin read of the structure arguments of X
deref(X, Y)	Dereference X into Y
move(X, Y)	Copy X to Y
globalize_ if_ unsafe(X, Y)	Copy (safely) X to stack variable Y
globalize_ to_ arg(X, Y)	Copy (safely) X to structure argument Y
jump(Label)	Jump to *Label*
cjump(Cond, Label)	Jump to *Label* if *Cond* is true
not(Cond)	Negate the *Cond* condition
test(Type, X)	True if X matches *Type*
equal(X, Y)	True if X and Y are equal
Indexing	
switch_ on_ type(X, Var, Str, List, Cons)	Jump to the label that matches the type of X
switch_ on_ functor(X, Table, Else)	
switch_ on_ cons(X, Table, Else)	

Table 3.2: Control and data instructions.

3.2.3 Compilation to C

The final C code conceptually corresponds to an unfolding of the emulator loop with respect to the particular sequence(s) of WAM instructions corresponding to the Prolog program. Each basic block of bytecode (i.e., each sequence beginning in a label and ending in an instruction involving a possibly non-local jump) is translated to a separate C function, which receives (a pointer to) the state of the abstract machine as input argument, and returns a pointer to the continuation. This approach, chosen on purpose, does not build functions which are too large for the C compiler to handle. For example, the code corresponding to a head unification is a basic block, since it is guaranteed that the labels corresponding to the two-stream algorithm will have local scope. A failure during unification is implemented by (conditionally) jumping to a special label, *fail*, which actually implements an exit protocol similar to that generated by the general C translation. Figure 3.2 shows schematic versions of the execution loop and templates of the functions that code blocks are compiled into.

This scheme does not require machine-dependent options of the C compiler or extensions to ANSI C. One of the goals of our system — to study the impact of optimizations based on high-level information on the program — can be achieved with the proposed compilation scheme, and, as mentioned before, we give portability and code cleanliness a high priority. The option of producing more efficient but non-portable code can always be added at a later stage.

3.2.4 Examples

We will illustrate briefly the different compilation stages using the well-known factorial program, whose original and normalized code is shown in (Figure 3.3). We have chosen it due to its simplicity, even if the performance gain is not very high in this case. The WAM code corresponding to the recursive clause is listed in the leftmost column of Table 3.3, while the internal representation of this code appears in the middle column of the same table. Variables are annotated using information which can be deduced from local clause inspection.

This WAM-like representation is translated to the low-level code as shown in Figure 3.4 (ignore, for the moment, the framed instructions; they will be discussed in Section 3.3). This code is what is finally translated to C.

Original

```
fact(0, 1).
fact(X, Y) :-
    X > 0,
    X0 is X - 1,
    fact(X0, Y0),
    Y is X * Y0.
```

Normalized

```
fact(A, B) :- 0 = A, 1 = B.
fact(A, B) :-
    A > 0,
    builtin__sub1_1(A, C),
    fact(C, D),
    builtin__times_2(A, D, B).
```

Figure 3.3: Original and normalized code of the factorial example.

For reference, executing `fact(100, N)` 20000 times took 0.65 seconds running emulated bytecode, and 0.63 seconds running the code compiled to C (a speedup of 1.03). This did not use external information, used the emulator data structures to store Prolog terms, and performed runtime checks to verify that the arguments are of the right type, even when this is not strictly necessary. Since the loop in Figure 3.2 is a bit more costly (by a few assembler instructions) than the WAM emulator loop, the speedup brought about by the C translation alone is, in many cases, not as relevant as one may think at first.

3.3 Optimized Compilation via Moded Types and Determinism

In order to improve the generated code using global information, the compiler can take into account types, modes, determinism and non-failure properties [HPBG05] coded as assertions [PBH00a] — a few such assertions can be seen in the example which appears later in this section. Automatization of the compilation process is achieved by using the CiaoPP analysis tool in connection with `ciaocc`. CiaoPP implements several powerful analysis (for modes, types, and determinacy, besides other relevant properties) which are able to generate (or check) these assertions. The program information that CiaoPP is currently able to infer automatically is actually enough for our purposes (with the single exception stated in Section 3.5).

WAM code	Without Types/Modes	With Types/Modes
put_constant(0,2)	0 = ⟨uninit,x(2)⟩	0 = ⟨uninit,x(2)⟩
builtin_2(37,0,2)	⟨init,x(0)⟩ > ⟨int(0),x(2)⟩	⟨int,x(0)⟩ > ⟨int(0),x(2)⟩
allocate	builtin__push_frame	builtin__push_frame
get_y_variable(0,1)	⟨uninit,y(0)⟩ = ⟨init,x(1)⟩	⟨uninit,y(0)⟩ = ⟨var,x(1)⟩
get_y_variable(2,0)	⟨uninit,y(2)⟩ = ⟨init,x(0)⟩	⟨uninit,y(2)⟩ = ⟨int,x(0)⟩
init([1])	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩	⟨uninit,y(1)⟩ = ⟨uninit,y(1)⟩
true(3)	builtin__complete_frame(3)	builtin__complete_frame(3)
function_1(2,0,0)	builtin__sub1_1(⟨init,x(0)⟩, ⟨uninit,x(0)⟩)	builtin__sub1_1(⟨int,x(0)⟩, ⟨uninit,x(0)⟩)
put_y_value(1,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨var,y(1)⟩ = ⟨uninit,x(1)⟩
call(fac/2,3)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨init,x(1)⟩)	builtin__modify_frame(3) fact(⟨init,x(0)⟩, ⟨var,x(1)⟩)
put_y_value(2,0)	⟨init,y(2)⟩ = ⟨uninit,x(0)⟩	⟨int,y(2)⟩ = ⟨uninit,x(0)⟩
put_y_value(2,1)	⟨init,y(1)⟩ = ⟨uninit,x(1)⟩	⟨number,y(1)⟩ = ⟨uninit,x(1)⟩
function_2(9,0,0,1)	builtin__times_2(⟨init,x(0)⟩, ⟨init,x(1)⟩,⟨uninit,x(0)⟩)	builtin__times_2(⟨int,x(0)⟩, ⟨number,x(1)⟩, ⟨uninit,x(0)⟩)
get_y_value(0,0)	⟨init,y(0)⟩ = ⟨init,x(0)⟩	⟨var,y(0)⟩ = ⟨init,x(0)⟩
deallocate	builtin__pop_frame	builtin__pop_frame
execute(true/0)	builtin__proceed	builtin__proceed

Table 3.3: WAM code and internal representation without and with external types information. Underlined instruction changed due to additional information.

3.3.1 Extended Typing of WAM Instructions

The generation of low-level code using additional type information makes use of a lattice of moded types obtained by extending the *init* element in the lattice in Figure 3.1 with the type domain in Figure 3.5. `str(N/A)` corresponds to (and expands to) each of the structures whose name and arity are known at compile time. This information enriches the *Type* parameter of the low-level code. Information about the determinacy / number of solutions of each call is carried over into this stage and used to optimize the C code.

```
fact(x(0), x(1)) :-
  push_choice(2)
  ensure_heap(callpad,2)
  deref(x(0),x(0))
   cjump(not(test(var,x(0))),V3)
   load(temp2,int(0))
   bind(var,x(0),nonvar,temp2)
   jump(V4)
 V3:
   cjump(not(test(int(0),x(0))),fail)
 V4:
   deref(x(1),x(1))
   cjump(not(test(var,x(1))),V5)
   load(temp2,int(1))
   bind(var,x(1),nonvar,temp2)
   jump(V6)
 V5:
   cjump(not(test(int(1),x(1))),fail)
 V6:
   complete_choice(2)
;
```

```
last_choice(2)
load(x(2),int(0))
>(x(0),x(2))
push_frame
move(x(1),y(0))
move(x(0),y(2))
init(y(1))
complete_frame(3)
builtin__sub1(x(0), x(0))
move(y(1),x(1))
modify_frame(3)
fact(x(0), x(1))
recover_frame
move(y(2),x(0))
move(y(1),x(1))
builtin__times(x(0), x(1), x(0))
deref(y(0),temp)
deref(x(0),x(0))
=(temp,x(0))
pop_frame
```

Figure 3.4: Low level code for the `fact/2` example (see also Section 3.3).

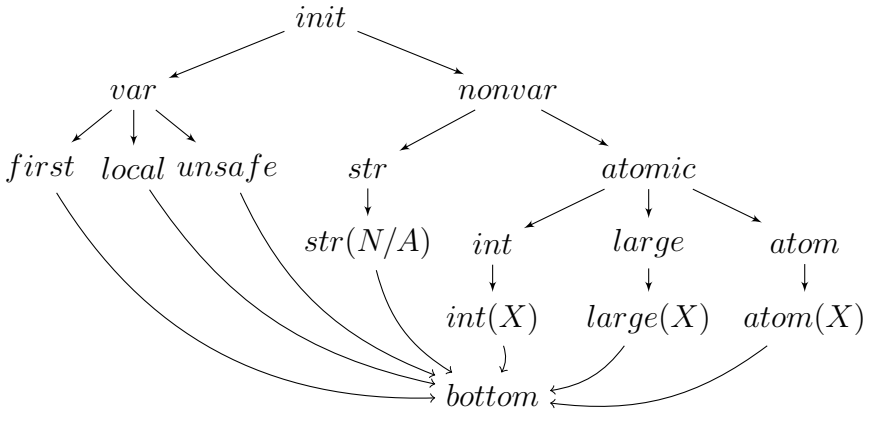

Figure 3.5: Extended *init* subdomain.

3.3.2 Generation of Optimized C Code

In general, information about types and determinism makes it possible to avoid some runtime tests. The standard WAM compilation also performs some optimizations (e.g., classification of variables and indexing on the first argument), but they are based on a per-clause (per-predicate, in the case of indexing) analysis, and in general it does not propagate the deduced information (e.g. from arithmetic builtins). A number of further optimizations can be done by using type, mode, and determinism information:

Unify Instructions: Calls to the general *unify* builtin are replaced by the more specialized *bind* instruction if one or both arguments are known to store variables. When arguments are known to be constants, a simple comparison instruction is emitted instead.

Two-Stream Unification: Unifying a register with a structure/constant requires some tests to determine the unification mode (read or write). An additional test is required to compare the register value with the structure/constant. These tests can often be removed at compile-time if enough information is known about the variable.

Indexing: Index trees are generated by selecting literals (mostly builtins and unifications), which give type/mode information, to construct a decision tree on the types of the first argument.[3] When type information is available, the search can be optimized by removing some of the tests in the nodes.

Avoiding Unnecessary Variable Safety Tests: Another optimization performed in the low level code using type information is the replacement of globalizing instructions for unsafe variables by explicit dereferences. When the type of a variable is nonvar, its globalization is equivalent to a dereference, which is faster.

Uninitialized Output Arguments: When possible, letting the called predicate fill in the contents of output arguments in pre-established registers avoids

[3]This is the WAM definition, which can of course be extended to other arguments.

allocation, initialization, and binding of free variables, which is slower.

Selecting Optimized Predicate Versions: Calls to predicates can also be optimized in the presence of type information. Specialized predicate versions (in the sense of low level optimizations) can be generated and selected using call patterns deduced from the type information. The current implementation does not generate specialized versions of user predicates, since this can already be done extensively by CiaoPP [PH03]. However it does optimize calls to internal *builtin* predicates written in C (such as, e.g., arithmetic builtins), which results in relevant speedups in many cases.

Determinism: These optimizations are based on two types of analysis. The first one uses information regarding the number of solutions for a predicate call to deduce, for each such call, if there is a known and fixed fail continuation. Then, instructions to manage choicepoints are inserted. The resulting code is then re-analyzed to remove these instructions when possible or to replace them by simpler ones (e.g., to restore a choice point state without untrailing, if it is known at compile time that the execution will not trail any value since the choice point was created). The latter can take advantage of additional information regarding register, heap, and trail usage of each predicate.[4] In addition, the C back-end can generate different argument passing schemes based on determinism information: predicates with zero or one solution can be translated to a function returning a boolean, and predicates with exactly one solution to a function returning `void`. This requires a somewhat different translation to C (which we do not have space to describe in full) and which takes into account this possibility by bypassing the emulator loop, in several senses similarly to what is presented in [HS02].

3.3.3 Examples

Let us assume that it has been inferred that `fact/2` (Figure 3.3) is always called with its first argument instantiated to an integer and with a free variable in its second argument. This information is written in the assertion language for

[4]This is currently known only for internal predicates written in C, and which are available by default in the system, but the scheme is general and can be extended to Prolog predicates.

```
fact(A, B) :- true(int(A)), 0 = A, true(var(B)), 1 = B.
fact(A, B) :-
    true(int(A)), A > 0,
    true(int(A)), true(var(C)), builtin__sub1_1(A, C),
    true(any(C)), true(var(D)), fact(C, D),
    true(int(A)), true(int(D)), true(var(B)), builtin__times_2(A, D, B).
```

Figure 3.6: Annotated factorial (using type information).

example as:[5]

```
:- true pred fact(X, Y) :  int * var => int * int.
```

which reflects the types and modes of the calls and successes of the predicate. That information is also propagated through the normalized predicate producing the annotated program shown in Figure 3.6, where program-point information is also shown.

The WAM code generated for this example is shown in the rightmost column of Table 3.3. Underlined instructions were made more specific due to improved information — but note that the representation is homogeneous with respect to the "no information" case. The impact of type information in the generation of low-level code can be seen in Figure 3.4. Instructions inside the dashed boxes are removed when type information is available, and the (arithmetic) builtins enclosed in rectangles are replaced by calls to specialized versions which work with integers and which do not perform type/mode testing. The optimized `fact/2` program took 0.54 seconds with the same call as in Section 3.2.4: a 20% speedup with respect to the bytecode version and a 16% speedup over the compilation to C without type information.

[5]The `true` prefix implies that this information is to be *trusted and used*, rather than to be *checked* by the compiler. Indeed, we require the stated properties to be correct, and `ciaocc` does not check them: this is a task delegated to CiaoPP. Wrong *true* assertions can, therefore, lead to incorrect compilation. However, the assertions generated by CiaoPP are guaranteed correct by the analysis process.

3.4 Unboxing of Constants

The strategy for compilation to native code used so far preserves the original data representation of the WAM: data is still stored in tagged words (i.e., *boxed*). This does not incur a big performance penalty in most cases, since C compilers generate efficient code to do the tagging/untagging, and the overhead is, in general, relatively small in comparison with what is done with the data itself.

This overhead is however comparatively large for operations which are simple enough to be translated to a single assembler instruction. Arithmetic operations stand out, and floating-point arithmetic suffers from an additional overhead: floating-point numbers are not carried around directly in a tagged word; rather, the tagged word points to a structure which holds the floating-point number. Therefore, boxing and unboxing a floating-point number are comparatively costly operations which, in principle, have to be repeated every time a floating-point operation is performed. Additionally, keeping floating-point numbers boxed needs more memory and garbage collection has to be called more often.

Another disadvantage of keeping numerical values in boxed form is that when compiling to native code via C, the C compiler does not *see* native machine data (e.g., `int`s, `float`s, `double`s), since they are encoded inside tagged words. This makes it difficult for the compiler to apply many useful optimizations (instruction reordering, use of machine registers, inlining, etc.) devised for more idiomatic C programs.

Unboxing has been studied and applied in functional programming [Ler92, Pet89] with good speedup results. This is helped in part by the use of strict type systems and the lack of different instantiation modes. Strict typing (and compulsory information about modes and determinism) applies also to the case for Mercury, which does not need boxing and unboxing. An interesting related approach is that of [PL91] for Haskell, where the kernel language was augmented with types to denote explicitly unboxed values and the simplifications to remove redundant operations were formalized as program transformations. However, language differences and the issues that that work focuses on (strictness, polymorphism, etc.) makes applying directly these techniques difficult in our case.

Unboxing for CLP systems, which are untyped and dynamically tagged, has received comparatively little attention. For example, Aquarius [VR90] did not

perform boxing/unboxing, and mainstream CLP systems, such as SICStus, do not use it when compiling to native code. The closest work is perhaps [BD95] which proposes a compilation strategy for the concurrent, committed-choice logic language Janus which, starting from a program annotated with type and mode declarations, performs a series of analysis to determine the best representation of each procedure argument and to avoid redundant boxing/unboxing operations.

3.4.1 Overview of The Algorithm

We share in fact some ideas with [BD95], although the languages are quite different. Similar type and mode annotations are required, which are inferred automatically in our case. However, we have to infer also information about determinism and non-failure, which is implicit in the language design of Janus, as it does not support backtracking or failure. A similarity with [PL91] is that we have formulated the solution as a source-to-source transformation on an extended language which includes boxing and unboxing operations. The implementation used in our experiments supports unboxed representations for some basic, native types and for temporal variables with a restricted lifetime, in order to ensure that there will be no interaction with garbage collection.

Our approach to boxing/unboxing removal works by exposing the code of builtins which inspect word tags. They typically share a similar structure: perform type checking on input arguments, unbox values, operate on them, box output values, and unify with output arguments. Informally, the process we use to detect and remove unneeded boxing/unboxing changes is:

1. Unfold builtin definitions to make type checking, unboxing, and boxing visible.

2. Make a forward pass to remove redundant unboxing operations. An abstract state relating boxed variables with their unboxed version is kept. It is updated with each unbox operation by adding a pair of linked variables (corresponding to boxed/unboxed views of the same entity) and by removing the pair when the versions become out-of-sync or, for temporal variables, when they become out of scope. This state is consulted to check for the availability of unboxed versions of variables when needed.

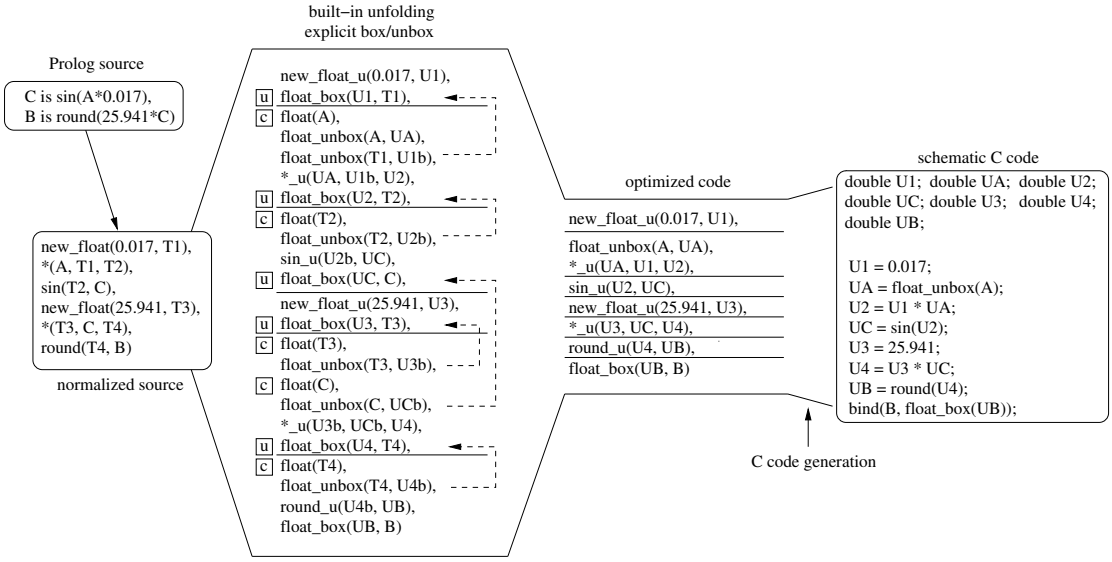

Figure 3.7: Unboxing optimization.

3. Make a backward pass to remove unnecessary box operations whose result is not used any longer.

3.4.2 Example

Figure 3.7 sketches how the algorithm behaves for a short piece of code corresponding to the body of `find_skip/2` (Section 7.3.4). The initial code is shown in the box at the top left. The next box contains the same code after splitting arithmetic expressions into basic operations which still work on boxed data and adding number-creation primitives. Each of these primitives is later expanded into smaller components which either create or disassemble boxed values or work directly with unboxed numbers.

Dashed lines with arrows relate pairs of unbox / box operations which can be simplified by a forward pass, since the boxed versions of the variables are not used between them. Goals marked with c denote checks (coming from builtin expansion) which are statically known to be true at runtime, either because of assertions at source level or thanks to information gathered in the fragment of code being compiled. They can be safely removed. Finally, goals marked with u are marked as unnecessary during the backward pass because their output value is not used.

The next two stages show the intermediate program after removal of dead code and, finally, the corresponding C code. Only one boxing and one unboxing operations (for the input and output parameters, respectively) are needed, and intermediate variables have been mapped to C (native) variables. Additionally, since mode information tells us that the second argument is always free variable, only a very specialized form of unification (the call to bind(), in fact a pointer assignment with trailing) is needed.

Performance Evaluation This optimization technique and the combination with all the others has been studied in a real-life application in Chapter 7.

3.5 Experimental Results

Program	Bytecode (Std. Ciao)	Non opt. C	Opt1. C	Opt2. C
queens11 (1)	691	391 (1.76)	208 (3.32)	166 (4.16)
crypt (1000)	1525	976 (1.56)	598 (2.55)	597 (2.55)
primes (10000)	896	697 (1.28)	403 (2.22)	402 (2.22)
tak (1000)	9836	5625 (1.74)	5285 (1.86)	771 (12.75)
deriv (10000)	125	83 (1.50)	82 (1.52)	72 (1.74)
poly (100)	439	251 (1.74)	199 (2.20)	177 (2.48)
qsort (10000)	521	319 (1.63)	378 (1.37)	259 (2.01)
exp (10)	494	508 (0.97)	469 (1.05)	459 (1.07)
fib (1000)	263	245 (1.07)	234 (1.12)	250 (1.05)
knights (1)	621	441 (1.46)	390 (1.59)	356 (1.74)
Average Speedup		(1.46 – 1.43)	(1.88 – 1.77)	(3.18 – 2.34)

Table 3.4: Bytecode emulation vs. unoptimized, optimized (types), and optimized (types and determinism) compilation to C. *Arithmetic – Geometric* means are shown.

We have evaluated the performance of a set of benchmarks executed by emulated bytecode, translation to C, and by other programming systems. The benchmarks, while representing interesting cases, are not real-life programs, and some of them have been executed up to 10.000 times in order to obtain reasonable and

stable execution times. Since parts of the compiler are still in an experimental state, we have not been able to use larger benchmarks yet. All the measurements have been performed on a Pentium 4 Xeon @ 2.0GHz with 1Gb of RAM, running Linux with a 2.4 kernel and using `gcc` 3.2 as C compiler. A short description of the benchmarks follows:

crypt: Cryptoarithmetic puzzle involving multiplication.
primes: Sieve of Erathostenes (with N = 98).
tak: Takeuchi function with arguments `tak(18, 12, 6, X)`.
deriv: Symbolic derivation of polynomials.
poly: Symbolically raise `1+x+y+z` to the 10^{th} power.
qsort: QuickSort of a list of 50 elements.
exp: 13^{7111} using both a linear- and a logarithmic-time algorithm.
fib: F_{1000} using a simply recursive predicate.
knight: Chess knight tour in a 5×5 board.

A summary of the results appears in Table 3.4. The figures between parentheses in the first column is the number of repetitions of each benchmark. The second column contains the execution times of programs run by the Ciao bytecode emulator. The third column corresponds to programs compiled to C without compile-time information. The fourth and fifth columns correspond, respectively, to the execution times when compiling to C with type and type+determinism information. The numbers between parentheses are the speedups relative to the bytecode version. All times are in milliseconds. Arithmetic and geometric means are also shown in order to diminish the influence of exceptional cases.

Program	GProlog	WAMCC	SICStus	SWI	Yap	Mercury	$\frac{\text{Opt2. C}}{\text{Mercury}}$
queens11 (1)	809	378	572	5869	362	106	1.57
crypt (1000)	1258	966	1517	8740	1252	160	3.73
primes (10000)	1102	730	797	7259	1233	336	1.20
tak (1000)	11955	7362	6869	74750	8135	482	1.60
deriv (10000)	108	126	121	339	100	72	1.00
poly (100)	440	448	420	1999	424	84	2.11
qsort (10000)	618	522	523	2619	354	129	2.01
exp (10)	—	—	415	—	340	—	—
fib (1000)	—	—	285	—	454	—	—
knights (1)	911	545	631	2800	596	135	2.63
						Average	1.98 – 1.82

Table 3.5: Speed of other Prolog systems and Mercury

Table 3.5 shows the execution times for the same benchmarks in five well-known Prolog compilers: GNU Prolog 1.2.16, `wamcc` 2.23, SICStus 3.8.6, SWI-Prolog 5.2.7, and Yap 4.5.0. The aim is not really to compare directly with them, because a different underlying technology and external information is being used, but rather to establish that our baseline, the speed of the bytecode system (Ciao), is similar and quite close, in particular, to that of SICStus. In principle, comparable optimizations could be made in these systems. The cells marked with "—" correspond to cases where the benchmark could not be executed (in GNU Prolog, `wamcc`, and SWI, due to lack of multi-precision arithmetic).

We also include the performance results for Mercury [SHC96] (version 0.11.0). Strictly speaking the Mercury compiler is not a Prolog compiler: the source language is substantially different from Prolog. But Mercury has enough similarities to be relevant and its performance represents an upper reference line, given that the language was restricted in several ways to allow the compiler, which generates C code with different degrees of "purity", to achieve very high performance by using extensive optimizations. Also, the language design requires the necessary information to perform these optimizations to be included by the programmer as part of the source. Instead, the approach that we use in Ciao is to infer automatically the information and not restricting the language.

Going back to Table 3.4, while some performance gains are obtained in the *naive* translation to C, these are not very significant, and there is even one program which shows a slowdown. We have tracked this down to be due to a combination of several factors:

- The simple compilation scheme generates clean, portable, "trick-free" C (some compiler dependent extensions would speed up the programs). The execution profile is very near to what the emulator would do.

- As noted in Section 3.2, the C compiler makes the fetch/switch loop of the emulator a bit cheaper than the C execution loop. We have identified this as a cause of the poor speedup of programs where recursive calls dominate the execution (e.g., `factorial`). We want, of course, to improve this point in the future.

- The increment in size of the program (to be discussed later — see Table 3.6) may also cause more cache misses. We also want to investigate this point

in more detail.

As expected, the performance obtained when using compile-time information is much better. The best speedups are obtained in benchmarks using arithmetic builtins, for which the compiler can use optimized versions where several checks have been removed. In some of these cases the functions which implement arithmetic operations are simple enough as to be inlined by the C compiler — an added benefit which comes for free from compiling to an intermediate language (C, in this case) and using tools designed for it. This is, for example, the case of `queens`, in which it is known that all the numbers involved are integers. Besides the information deduced by the analyzer, hand-written annotations stating that the integers involved fit into a machine word, and thus there is no need for infinite precision arithmetic, have been manually added.[6]

Determinism information often (but not always) improves the execution. The Takeuchi function (`tak`) is an extreme case, where savings in choicepoint generation affect execution time. While the performance obtained is still almost a factor of 2 from that of Mercury, the results are encouraging since we are dealing with a more complex source language (which preserves full unification, logical variables, cuts, `call`/1, database, etc.), we are using a portable approach (compilation to standard C), and we have not yet applied all possible optimizations.

A relevant point is to what extent a sophisticated analysis tool is useful in practical situations. The degree of optimization chosen can increase the time spent in the compilation, and this might preclude its everyday use. We have measured (informally) the speed of our tools in comparison with the standard Ciao Prolog compiler (which generates bytecode), and found that the compilation to C takes about three times more than the compilation to bytecode. A considerable amount of time is used in I/O, which is being performed directly from Prolog, and which can be optimized if necessary. Due to a well-developed machinery (which can notwithstanding be improved in a future by, e.g, compiling CiaoPP itself to C), the global analysis necessary for examples is really fast and never exceeded twice the time of the compilation to C. Thus we think that the use of global analysis to obtain the information we need for `ciaocc` is a practical option

[6]This is the only piece of information used in our benchmarks that cannot be currently determined by CiaoPP. It should be noted, though, that the absence of this annotation would only make the final executable less optimized, but never incorrect.

already in its current state.

Table 3.6 compares object size (in bytes) of the bytecode and the different schemes of compilation to C and using the same compiler options in all cases. While modern computers usually have a large amount of memory, and program size hardly matters for a single application, users stress computers more and more by having several applications running simultaneously. On the other hand, program size does impact their startup time, important for small, often-used commands. Besides, size is still very important when addressing small devices with limited resources.

As mentioned in Section 2, due to the different granularity of instructions, larger object files and executables are expected when compiling to C. The ratio depends heavily on the program and the optimizations applied. Size increase with respect to the bytecode can be as large as 15× when translating to C without optimizations, and the average case sits around a 7-fold increase. This increment is partially due to repeated code in the indexing mechanism, which we plan to improve in the future.[7] Note that, as our framework can mix bytecode and native code, it is possible to use both in order to achieve more speed in critical parts, and to save program space otherwise. Heuristics and translation schemes like those described in [TDBD96] can hence be applied (and implemented as a source to source transformation).

The size of the object code produced by `wamcc` is roughly comparable to that generated by `ciaocc`, although `wamcc` produces smaller intermediate object code files. However the final executable / process size depends also on which libraries are linked statically and/or dynamically. The Mercury system is somewhat incomparable in this regard: it certainly produces relatively small component files but then relatively large final executables (over 1.5 MByte).

Size, in general, decreases when using type information, as many runtime type tests are removed, the average size being around five times the bytecode size. Adding determinism information increases the code size because of the additional inlining performed by the C compiler and the more complex parameter passing code. Inlining was left to the C compiler; experiments show that more aggressive inlining does not necessarily result in better speedups.

[7] In all cases, the size of the bytecode emulator / runtime support (around 300Kb) has to be added, although not all the functionality it provides is always needed.

Program	Bytecode	Non opt. C	Opt1. C	Opt2. C
queens11	7167	36096 (5.03)	29428 (4.10)	42824 (5.97)
crypt	12205	186700 (15.30)	107384 (8.80)	161256 (13.21)
primes	6428	50628 (7.87)	19336 (3.00)	31208 (4.85)
tak	5445	18928 (3.47)	18700 (3.43)	25476 (4.67)
deriv	9606	46900 (4.88)	46644 (4.85)	97888 (10.19)
poly	13541	163236 (12.05)	112704 (8.32)	344604 (25.44)
qsort	6982	90796 (13.00)	67060 (9.60)	76560 (10.96)
exp	6463	28668 (4.43)	28284 (4.37)	25560 (3.95)
fib	5281	15004 (2.84)	14824 (2.80)	18016 (3.41)
knights	7811	39496 (5.05)	39016 (4.99)	39260 (5.03)
Average Increase		(7.39 – 6.32)	(5.43 – 4.94)	(8.77 – 7.14)

Table 3.6: Compared size of object files (bytecode vs. C) including *Arithmetic - Geometric* means.

It is interesting to note that some optimizations used in the compilation to C would not give comparable results when applied directly to a bytecode emulator. For example, a version of the bytecode emulator hand-coded to work with small integers (which can be boxed into a tagged word) performed worse than that obtained doing the same with compilation to C. That suggests that when the overhead of calling builtins is reduced, as is the case in the compilation to C, some optimizations which only produce minor improvements for emulated systems acquire greater importance.

3.6 Conclusions

We have reported on the scheme and performance of `ciaocc`, a Prolog-to-C compiler which uses type analysis and determinacy information to improve code generation by removing type and mode checks and by making calls to specialized versions of some builtins. We have also provided performance results. `ciaocc` is still in a prototype stage, but it already shows promising results.

The compilation uses internally a simplified and more homogeneous representation for WAM code, which is then translated to a lower-level intermediate code, using the type and determinacy information inferred by CiaoPP. This code is fi-

nally translated into C by the compiler back-end. The intermediate code makes the final translation step easier and will facilitate developing new back-ends for other target languages.

We have found that optimizing a WAM bytecode emulator is more difficult and results in lower speedups, due to the larger granularity of the bytecode instructions. The same result has been reported elsewhere [Van94], although some recent work tries to improve WAM code by means of local analysis [FD99].

4

A Generic Emulator Generator

Summary

Implementors of abstract machines face complex, and often interacting, decisions regarding, e.g., data representation, instruction design, instruction encoding, or instruction specialization levels. These decisions affect the performance of the emulator and the size of the bytecode programs in ways that are often difficult to foresee. Furthermore, studying different alternatives by implementing abstract machine variants is a time-consuming and error-prone task because of the level of complexity and optimization of competitive implementations, which makes them generally difficult to understand, maintain, and modify. This also makes it hard to generate specific implementations for particular purposes. We propose a systematic approach to the automatic generation of implementations of abstract machines which is aimed at harnessing some of these difficulties. Different parts of the abstract machine definition (e.g., the instruction set or the internal data and bytecode representation) are kept separate and automatically assembled in the generation process. Alternative versions of the abstract machine are therefore easier to produce, and variants of their implementation can be created mechanically. This even allows generating implementations tailored to a particular context. We illustrate the practicality of the approach by reporting on an implementation of a generator of production-quality WAMs which are specialized for executing a particular fixed (set of) program(s). The experimental results show that the approach is effective in reducing emulator size.

4.1 Introduction

Abstract machines are currently a popular alternative to native code compilation because they offer practical advantages, especially for programs written in high-level languages with complex features. When used in place of, or in combination with, native code compilation, these advantages include for example increased portability, small executable code size, simpler security control through sandboxing, or increased compilation speed. The use of intermediate abstract machines requires combining several components. In order to execute programs written in a *source language* $\mathcal{L}_s$, a compiler from $\mathcal{L}_s$ to the *abstract machine language*, $\mathcal{L}_a$, is needed. An interpreter for $\mathcal{L}_a$, usually written in some lower-level language $\mathcal{L}_c$, the *implementation language*, for which there is a compiler to native code, performs the actual execution.[1] Traditional approaches start with a fixed set of abstract machine instructions and then develop the $\mathcal{L}_s$ compiler and the $\mathcal{L}_a$ interpreter. Further development tends to work on both the $\mathcal{L}_s$ compiler and $\mathcal{L}_a$ interpreter implementation, often keeping the instruction set relatively unchanged.

One important concern when implementing such interpreters is that of efficiency (see [VR90, Van94, SC99, DN00, DC01]), which greatly depends on the complexity of $\mathcal{L}_s$ and, of course, on the compiler and emulator technology. As a result, efficient emulators are very often difficult to understand, maintain, and modify. This makes the implementation of *variants* of abstract machines a hard task, since both the compiler and emulator, which are rather complex, have to be rewritten by hand for each variation. Variants of emulators have been (naturally) used to evaluate different implementation options for a language [DN00], often manually. Automating the creation of these variants, in addition to making it easier to test them, will allow tailoring a general design to particular applications or environments with little effort. A particularly daunting task is to adapt existing emulators to resource-constrained applications, such as those found in pervasive computing. While this can clearly be done by carefully rewriting existing emulators, selecting alternative data representations, and, maybe, adapting them to the type of expected applications, we deem that this task is a too difficult one, especially taking into account the amount (and, especially, variety) of different

[1] Implementations of abstract machines are usually termed *virtual machines*. We will, however, use the term *emulator* or *bytecode interpreter* to denote a virtual machine. This is in line with the tradition used in the implementation of logic programming languages.

small devices which are ubiquitous nowadays.

In this work we propose an approach in which, rather than being hand-written, emulators and (back-end) compilers are automatically generated from a high-level description of the abstract machine instruction set. This makes it possible to easily experiment with alternative abstract machines and to evaluate the impact of different implementation decisions, since the corresponding emulator and compiler are obtained automatically.

In order to do so, rather than considering emulators for a particular abstract machine, we formalize emulators as parametric programs, written, for purposes of improved expressiveness, in a syntactical extension of $\mathcal{L}_c$ that can represent directly elements of $\mathcal{L}_a$ and which receive two inputs: a *program* to be executed, written in language $\mathcal{L}_a$, and a description of the abstract machine language $\mathcal{L}_a$ in which the operational definition of each instruction of $\mathcal{L}_a$ is given in terms of $\mathcal{L}_c$. I.e., we define a generic emulator as a procedure

$$\textbf{interpret}(P, \mathcal{M})$$

which takes as input a program P in the abstract machine language $\mathcal{L}_a$ and a definition $\mathcal{M}$ of the abstract machine itself and interprets P according to $\mathcal{M}$.

For the sake of maintainability and ease of manipulation, $\mathcal{L}_a$ is to be as close as possible to its conceptual definition. This usually affects performance negatively, and therefore a refinement step, based on pass separation [Han91], a form of staging transformations [JS86], is taken to convert programs written in $\mathcal{L}_a$ into programs written in $\mathcal{L}_b$, the *bytecode language*, which is a lower-level representation (with concerns such as data alignment, word size, etc. in mind) for which faster interpreters can be written in $\mathcal{L}_c$. By formalizing adequately the transformation from $\mathcal{L}_a$ to $\mathcal{L}_b$ it is possible to do automatically:

- The translation of programs from $\mathcal{L}_a$ into $\mathcal{L}_b$.
- The generation of efficient emulators for programs in $\mathcal{L}_b$ based on interpreters for $\mathcal{L}_a$.
- The generation of compilers from $\mathcal{L}_s$ to $\mathcal{L}_b$ based on compilers from $\mathcal{L}_s$ to $\mathcal{L}_a$.

A high-level view of the different elements we will describe in this chapter appears in Figure 4.1. When the abstract machine description $\mathcal{M}$ is available, it is possible (at least conceptually) to *partially evaluate* [JGS93, Fut71] the procedure **interpret** into an emulator for a (now fixed) $\mathcal{M}$. Of course, it is desirable that the emulator to as efficient as possible, and since its implementation language will finally be that of **interpret**, the interpreter itself should be written in $\mathcal{L}_c$. Although this approach is attractive in itself, it has the disadvantage that the existence of a partial evaluator of programs written in $\mathcal{L}_c$ is required. Depending on $\mathcal{L}_c$, this may or may not be a good approach in practice.

A well known result in partial evaluation is that it is possible to partially evaluate a partial evaluator w.r.t. itself and a particular program P as static data, and the result is a partial evaluator specialized to perform the partial evaluation of the particular program P. In our case, by taking the parametric interpreter as static data for the partial evaluator, we can obtain an emulator generator (*emucomp*), which will produce an efficient emulator when supplied with a description of an abstract machine. This approach, known as the second Futamura projection [Fut71], not only requires the availability of a partial evaluator for programs in $\mathcal{L}_c$ but also needs the partial evaluator to be self-applicable. Somewhat surprisingly, the structure of emulator generators is often easy to understand. In fact, the approach we will follow in this work is to write such an emulator generator directly by hand. The emulator generator we propose has been defined in such a way that, and as will be discussed below, it produces emulators whose code is comparable to those produced by a skilled programmer when provided with the description of an abstract machine.

Table 4.1 summarizes the meaning and relationships among the interpreters and emulators that we will define in this chapter. This table will be useful for reference throughout the discussion.

The benefits of our approach are multifold. Writing an emulator generator is clearly much more profitable than writing a particular emulator (though more difficult to achieve for the general case) since, with no performance penalty, it will make it possible to easily experiment with multiple variations of the original abstract machine. For example, and as discussed later, it is straightforward to produce reduced emulators. As an example of the application of our technique, and taking as starting point the instruction set of an existing emulator (a

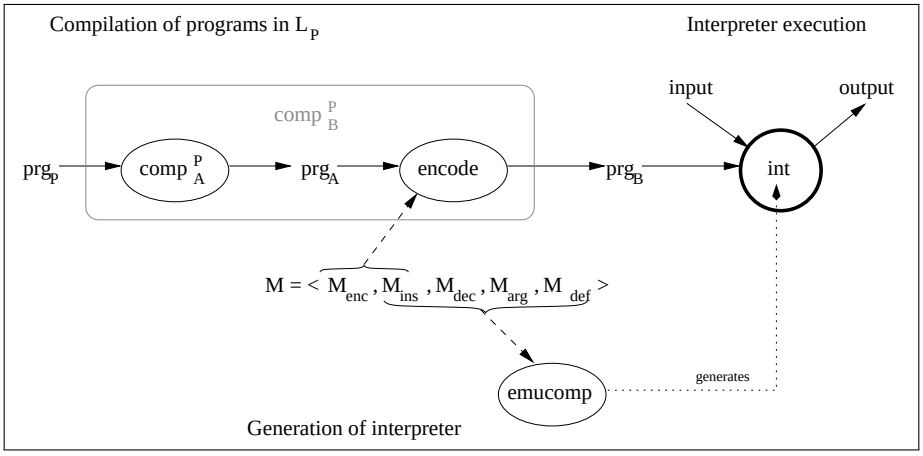

Figure 4.1: "Big Picture" view of the generation of emulators

$\mathbf{int_A}(p, program)$	Interpreter for $\mathcal{L}_a$
$\mathbf{int_1}(p, program, \mathcal{M})$	Interpreter for $\mathcal{L}_a$ which is parametric with respect to the abstract machine definition $\mathcal{M}$
$\mathbf{int_2}(p, prg, \mathcal{M})$	Interpreter for $\mathcal{L}_b$ (i.e., emulator) which reuses the structure of $\mathbf{int_1}$ and applies the augmented abstract machine (Section 4.2.3 and Figure 4.6)
$\mathbf{int_3} \equiv [\![spec]\!](\mathbf{int_2}, \mathcal{M})$	Interpreter for $\mathcal{L}_b$ which is specialized for a *fixed* abstract machine $\mathcal{M}$
$emucomp \equiv [\![spec]\!](spec, \mathbf{int_2})$	Generator of $\mathcal{L}_b$ emulators for some abstract machine definition
$[\![spec]\!](\mathbf{int_2}, \mathcal{M}) \equiv emucomp(\mathcal{M})$	Correctness condition for the emulator compiler

Table 4.1: Series of interpreters and emulators in this chapter

production-quality implementation of a modern version of the Warren Abstract Machine for Prolog [War83, AK91]), we generate emulators which are *sliced* with respect to the set of abstract machine instructions which a given application or sets of applications are going to actually use, thus producing smaller and resource friendlier executables.

4.2 Algorithm for the Generation of Emulators

In this section we will develop an emulator generator which takes a description of the machine and can produce emulators which are very close (and in some cases identical) to what a skilled programmer would craft.

Our initial source language is $\mathcal{L}_s$, and we assume that there is a compiler *comp* from $\mathcal{L}_s$ to $\mathcal{L}_a$, a symbolic representation of a lower-level language $\mathcal{L}_b$ intended to be interpreted by an emulator. We want *comp* to be relatively simple and independent from the low-level details of the implementation of the final emulator. The definition of $\mathcal{L}_a$ will be kept separate in $\mathcal{M}$ so that it can be used later (Section 4.2.2) in a generic interpreter. Instructions in $\mathcal{L}_a$ can, in general, consult and modify a global state and change the control flow with (conditional) *jump*/*call* instructions.

4.2.1 Scheme of a Basic Interpreter

Let us consider interpreters based on a main loop implementing a fetch-execute cycle. Figure 4.2 portrays an example, where that cycle is performed by a tail-recursive procedure. The reason to choose this scheme is because it allows a shorter description of some further transformations. Note that, since it is tail recursive, it can be converted easily (and automatically) into a proper loop without the need for additional data structures (stacks, etc.). The function:

$$fetch_A : locator_A \times program_A \rightarrow \langle ins_A, locator_A \rangle$$

returns, for a given program and program point, the instruction at that point (of type ins_A, a tuple containing instruction name and arguments) and the next location in the program, in sequential order. This abstracts away program counters, which can be symbolic, and indirections through the program counter. We will reuse this function, in different contexts, in the following sections.

Example 4.2.1 ($\mathcal{L}_a$ instructions and their semantics written in $\mathcal{L}_c$)**:** The left hand side of each of the branches in the *case* expression of Figure 4.2 corresponds to one instruction in $\mathcal{L}_a$. The interpreter $\mathbf{int_A}$ is written in $\mathcal{L}_c$ (syntactically extended to represent $\mathcal{L}_a$ instructions), and the semantics of each instruction is given in terms of $\mathcal{L}_c$ in the right hand side of the corresponding branch. The

$$
\begin{array}{l}
\mathbf{int_A}(p, program) \equiv \\
\quad \langle ins, p' \rangle = fetch_A(p, program) \\
\quad \text{case } ins \text{ of} \\
\qquad \langle \mathbf{move}, [\mathbf{r}(i), \mathbf{r}(j)] \rangle \quad : reg[j] := reg[i];\ \ p'' := p' \\
\qquad \langle \mathbf{jump}, [\mathbf{label}(l)] \rangle \quad : p'' := l \\
\qquad \langle \mathbf{call}, [\mathbf{label}(l)] \rangle \quad : push(p');\ \ p'' := l \\
\qquad \langle \mathbf{ret}, [] \rangle \quad : p'' := pop() \\
\qquad \langle \mathbf{halt}, [] \rangle \quad : return \\
\qquad \text{otherwise} \quad : \mathbf{error} \\
\quad \mathbf{int_A}(p'', program)
\end{array}
$$

Figure 4.2: An example of a simple $\mathcal{L}_a$-level interpreter

$$
\begin{array}{l}
\mathbf{int_1}(p, program, \mathcal{M}) \equiv \\
\quad \langle \langle name, args \rangle, p' \rangle = fetch_A(p, program) \\
\quad \text{if } \neg valid_A(\langle name, args \rangle, \mathcal{M}_{ins}, \mathcal{M}_{absexp}) \text{ then } \mathbf{error} \\
\quad cont = \lambda a \rightarrow [p'' := a] \\
\quad [\![\mathcal{M}_{def}(p', cont, name, \mathcal{M}_{args}(args))]\!] \\
\quad \mathbf{int_1}(p'', program, \mathcal{M})
\end{array}
$$

Figure 4.3: Parametric interpreter for $\mathcal{L}_a$

implementation of the memory model is implicit at the right hand side of the *case* branches; we assume that appropriate declarations for types and global variables exist. $\mathcal{L}_a$ instructions (and related data structures) are able to **move** data between registers, do **jump**s, **call**s to and **ret**urn from subroutines, and stop the execution with the **halt** instruction. Alternative interpreters can be crafted by changing the way $\mathcal{L}_a$ instructions are implemented. This must, of course, be done homogeneously across all the instruction definitions.

4.2.2 Parameterizing the Interpreter

In order to make interpreters parametric with respect to the abstract machine definition, we need to settle on an interpreter scheme first, which we show in

Figure 4.3, and to make the definition of the abstract machine precise. We will use a piecewise definition

$$\mathcal{M} = (\mathcal{M}_{def}, \mathcal{M}_{arg}, \mathcal{M}_{ins}, \mathcal{M}_{absexp})$$

of $\mathcal{L}_a$ which is passed as a parameter to the interpreter scheme and which relates different parts of the abstract machine with a feasible implementation thereof. The meaning of each component of $\mathcal{M}$ (see also Example 4.2.2 for a concrete case) is as follows:

$\mathcal{M}_{def}$ Provides the correspondence between every instruction of $\mathcal{L}_a$ and the code to execute it in $\mathcal{L}_c$.

$\mathcal{M}_{arg}$ The correspondence between every argument for the instructions in $\mathcal{L}_a$ and the corresponding data in $\mathcal{L}_c$. $\mathcal{M}_{args}$ generalizes $\mathcal{M}_{arg}$ by mapping lists of arguments in $\mathcal{L}_a$ into lists of arguments in $\mathcal{L}_c$. The definitions of $\mathcal{M}_{def}$ and $\mathcal{M}_{arg}$ are highly dependent, and quite often updating one will require changes in the other.

$\mathcal{M}_{ins}$ The instruction set, described as the signature of every instruction in $\mathcal{L}_a$, i.e., the instruction name and which kinds of expressions in $\mathcal{L}_a$ can be handled by that instruction. The format is given as a list of abstract expressions of $\mathcal{L}_a$, whose definition is also included in $\mathcal{M}$ (see next item). For example, a **jump** instruction might be able to jump to a (static) label, but not to the address contained in a register, or a **move** instruction might be able to store a number in a register but not directly in a memory location. Note that in general we want to be able to use the same instruction name with different formats.

$\mathcal{M}_{absexp}$ An abstraction function which returns the type of an instruction argument.

The interpreter in Figure 4.3 is parametric w.r.t. a definition of the semantics of the abstract machine language $\mathcal{L}_a$, described in terms of the implementation language $\mathcal{L}_c$. First, for every instruction, arguments in $\mathcal{L}_a$ are translated into arguments in $\mathcal{L}_c$ by $\mathcal{M}_{args}$. Then, $\mathcal{M}_{def}$ selects the right code for the instruction. Both $\mathcal{M}_{def}$ and $\mathcal{M}_{arg}$ are functions which return unevaluated pieces of code, which

are meant to be executed by $\mathbf{int_1}$ — this is marked by enclosing the function call within double square brackets — in the following iteration of the emulator. The next program location is set by a function *cont* which is handed in to $\mathcal{M}_{def}$ as an argument. The language expressions not meant to be evaluated, but to be passed as data are enclosed inside square brackets, such as $[p'' := a]$. The context should be enough to distinguish them from those used to access array elements or to denote lists.

In order to ensure that no ill-formed instruction is executed (for example, because a wrongly computed location tries to access instructions outside the program scope), the function $valid_A$ checks that the instruction named *name* can understand the arguments $args$ which it receives. It needs to traverse every argument, extract its type, which defines an argument format, and check that the instruction *name* can be used with arguments following that format.

Example 4.2.2 (Definitions for a trivial abstract machine in $\mathbf{int_1}$)**:** In the definitions for $\mathcal{M}$ in Figure 4.4, the higher-order argument *cont* is used in order to set the program counter to the address of the instruction to be executed next. The instruction definitions do not check operator and operand types, since that has been taken care of by $valid_A$ by checking that the type of every argument matches those accepted by the instruction to be executed.

Instructions can in general admit several argument formats. For example, arithmetic instructions might accept integers and floating-point numbers. That would make $\mathcal{M}_{ins}$ have several entries for some instructions. This is not a serious problem, as long as $\mathcal{M}_{absexp}$ returns all abstractions for a given pattern and there is a suitable selection rule (e.g., the most concrete applicable pattern) is used to choose among different possibilities. For the sake of simplicity we will not deal with that case. Multi-format instructions are helpful when compiling weakly-typed languages, or languages with complex inheritance rules, where types of expressions might not be completely known until runtime. If this happens, compiling to a general case to be dynamically checked is the only solution.

4.2.3 A More Specialized Intermediate Language and Its Interpreter

The symbolic nature of $\mathcal{L}_a$, which should be seen as an intermediate language, makes it convenient to express instruction definitions and to associate internally

$$
\begin{array}{ll}
\mathcal{M}_{def}(next, cont, name, args) = & \mathcal{M}_{ins} = \\
\quad \text{case } \langle name, args \rangle \text{ of} & \quad \{\ \langle \mathbf{move}, [\mathbf{r}, \mathbf{r}] \rangle \\
\quad\quad \langle \mathbf{move}, [a, b] \rangle \rightarrow [a := b;\ \ cont(next)] & \quad\quad \langle \mathbf{jump}, [\mathbf{label}] \rangle \\
\quad\quad \langle \mathbf{jump}, [a] \rangle \rightarrow [cont(a)] & \quad\quad \langle \mathbf{call}, [\mathbf{label}] \rangle \\
\quad\quad \langle \mathbf{call}, [a] \rangle \rightarrow [push(next);\ \ cont(a)] & \quad\quad \langle \mathbf{ret}, [] \rangle \\
\quad\quad \langle \mathbf{ret}, [] \rangle \rightarrow [cont(pop())] & \quad\quad \langle \mathbf{halt}, [] \rangle\ \} \\
\quad\quad \langle \mathbf{halt}, [] \rangle \rightarrow [return] & \\
 & \\
\mathcal{M}_{arg}(arg) = & \mathcal{M}_{absexp}(arg) = \\
\quad \text{case } arg \text{ of} & \quad \text{case } arg \text{ of} \\
\quad\quad \mathbf{r}(i) \rightarrow reg[i] & \quad\quad \mathbf{r}(\_) \rightarrow \mathbf{r} \\
\quad\quad \mathbf{label}(l) \rightarrow l & \quad\quad \mathbf{label}(\_) \rightarrow \mathbf{label} \\
 & \quad\quad \text{otherwise} \rightarrow \bot
\end{array}
$$

Figure 4.4: Definition of $\mathcal{M}$ for our example

properties to them, but it is not designed to be directly executed. Most emulators use a so-called *bytecode* representation, where many details have been settled: operation codes for each instruction (which capture the instruction name and argument types), size of every instruction, fixed values of some arguments, etc. In return, bytecode interpreters are quite fast, because a great deal of the work $\mathbf{int_1}$ does has been statically encoded, so that several sources of overhead can be removed. In short, the bytecode design focuses on achieving speed.

On the other hand, working right from the beginning with a low-level definition is cumbersome, because many decisions percolate through the whole language and seemingly innocent changes can force the update of a significant part of the bytecode language definition (and, therefore, of its emulator). This is the main reason to keep $\mathcal{L}_a$ at a high level, with many details still to be filled in. It is however possible to translate $\mathcal{L}_a$ into a lower-level language, $\mathcal{L}_b$, closer to $\mathcal{L}_c$ and easier to represent using $\mathcal{L}_c$ data structures. That process can be instrumented so that programs written in $\mathcal{L}_a$ are translated into $\mathcal{L}_b$ and interpreters for $\mathcal{L}_a$ are transformed into interpreters for $\mathcal{L}_b$ using a similar encoding. Figure 4.5 depicts the temporal stage were a pass separation is introduced to change the representation language. Translating from $\mathcal{L}_a$ to $\mathcal{L}_b$ is done by a function:

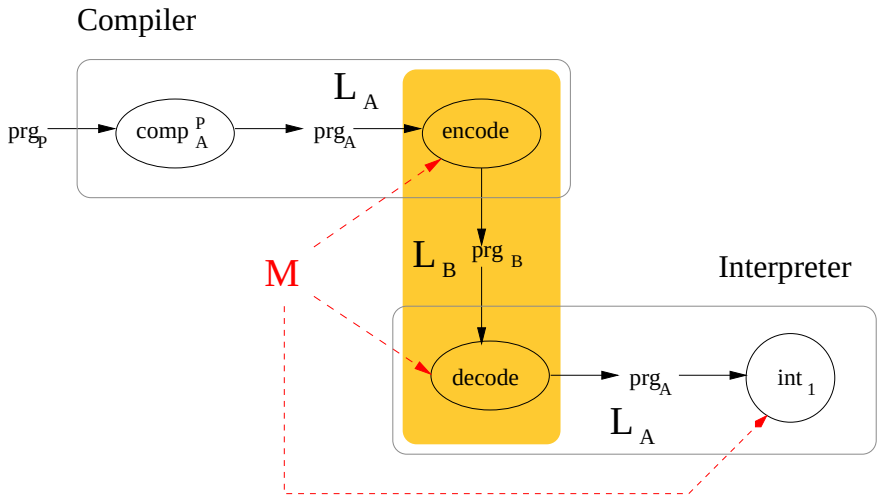

Figure 4.5: Pass separation

$$encode : \mathcal{L}_a \rightarrow \mathcal{L}_b$$

encode accepts instructions in $\mathcal{L}_a$ (including name and arguments) and returns tokens in $\mathcal{L}_b$. The encoding function has to:

1. Assign a unique operation code (*opcode*) to each instruction in $\mathcal{L}_a$ when the precondition expressed by $valid_A$ holds (a compile-time error would be raised otherwise). This moves the overhead of checking formats from runtime to compile-time.

2. Take the arguments of instructions in $\mathcal{L}_a$ and translate them into $\mathcal{L}_b$.

encode is also used to generate a compiler from $\mathcal{L}_s$ into $\mathcal{L}_b$ from a compiler from $\mathcal{L}_s$ into $\mathcal{L}_a$ (Figure 4.1). As *encode* gives a unique opcode to every combination of instruction name and format, it has an associated function:

$$decode : \mathcal{L}_b \rightarrow \mathcal{L}_a$$

which brings bytecode instructions back to its original form.[2] In order to capture the meaning of *encode* / *decode*, we augment and update the abstract machine

[2]Both *encode* and *decode* may need to resolve symbols. As this is a standard practice in compiling (which can even be delayed until link time), we will not deal with that problem here.

$$
\begin{array}{lll}
\mathcal{M}_{ins'}(opcode) = & \mathcal{M}_{enc}(arg) = & \mathcal{M}_{dec}(t, f) = \\
\quad \text{case } opcode \text{ of} & \quad \text{case } arg \text{ of} & \quad \text{case } \langle t, f \rangle \text{ of} \\
\quad\quad 0 \rightarrow \langle \mathbf{move}, [\mathbf{r}, \mathbf{r}] \rangle & \quad\quad \langle \mathbf{r}(a) \rangle \rightarrow a & \quad\quad \langle a, \mathbf{r} \rangle \rightarrow \mathbf{r}(a) \\
\quad\quad 1 \rightarrow \langle \mathbf{jump}, [\mathbf{label}] \rangle & \quad\quad \langle \mathbf{label}(l) \rangle \rightarrow symbol(l) & \quad\quad \langle l, \mathbf{label} \rangle \rightarrow \mathbf{label}(l) \\
\quad\quad 2 \rightarrow \langle \mathbf{call}, [\mathbf{label}] \rangle & & \\
\quad\quad 3 \rightarrow \langle \mathbf{ret}, [] \rangle & & \\
\quad\quad 4 \rightarrow \langle \mathbf{halt}, [] \rangle & &
\end{array}
$$

Figure 4.6: New parts of the abstract machine definition

$$
\begin{array}{ll}
\mathbf{int_2}(p, prg, \mathcal{M}) \equiv & decode_{ins}(\langle f_1, \ldots, f_n \rangle, p, prg, \mathcal{M}) = \\
\quad opcode = prg[p] & \quad \langle \langle d_1, \ldots, d_n \rangle, p + 1 + n \rangle \text{ where} \\
\quad \langle name, format \rangle = \mathcal{M}_{ins'}(opcode) & \quad d_i = \mathcal{M}_{dec}([prg[p + i]], f_i) \\
\quad \langle args, p' \rangle = decode_{ins}(format, [p], [prg], \mathcal{M}) & \\
\quad cont = \lambda a \rightarrow [\mathbf{int_2}(a, prg, \mathcal{M});\ return] & \\
\quad [\![\mathcal{M}_{def}(p', cont, name, \mathcal{M}_{args}(args));\ cont(p')]\!] &
\end{array}
$$

Figure 4.7: Parametric interpreter for $\mathcal{L}_b$

definition to be $\mathcal{M} = (\mathcal{M}_{def}, \mathcal{M}_{arg}, \mathcal{M}_{ins'}, \mathcal{M}_{absexp}, \mathcal{M}_{enc}, \mathcal{M}_{dec})$ (see Figure 4.6 and Example 4.2.3). $\mathcal{M}_{ins'}$ is derived from $\mathcal{M}_{ins}$ by capturing the opcode assignment. It accepts an opcode and returns the corresponding instruction in $\mathcal{L}_a$ as a pair $\langle name, format \rangle$. Argument encoding is taken care of by a new function $\mathcal{M}_{enc}$. $\mathcal{M}_{dec}$ is the inverse of $\mathcal{M}_{enc}$.

An interpreter $\mathbf{int_2}$ for $\mathcal{L}_b$ (see Figure 4.7) can be derived from $\mathbf{int_1}$ with the help of bytecode decoding. $\mathbf{int_2}$ receives an (extended) definition of $\mathcal{M}$ and uses it to retrieve the original instruction $\langle name, format \rangle$ in $\mathcal{L}_a$ corresponding to an opcode in a bytecode program (returned by $program[p]$, where p is a program counter in $\mathcal{L}_b$). The arguments are brought from the domain of $\mathcal{L}_b$ back to the domain of $\mathcal{L}_a$ by $\mathcal{M}_{dec}$, and code and argument translations defined by $\mathcal{M}_{def}$ and $\mathcal{M}_{arg}$ can then be employed as in $\mathbf{int_1}$.

We want to note that in Figure 4.7 the recursive call has been placed inside the continuation code, which avoids the use of the intermediate variable p'' used in Figure 4.2 and makes it easier to apply program transformations.

$$
\begin{array}{l|l}
emucomp(\mathcal{M}) = & inscomp(opcode, \mathcal{M}) = \\
\quad [\,\mathbf{emu_B}(p, prg) \equiv & \quad [\mathcal{M}_{def}(p', cont, name, \mathcal{M}_{args}(args));\ cont(p')] \\
\quad\quad \text{case } get\_opcode(p, prg) \text{ of} & \quad \text{where} \\
\quad\quad\quad opcode_1 : inscomp(opcode_1, \mathcal{M}) & \quad\quad \langle name, format\rangle = \mathcal{M}_{ins'}(opcode) \\
\quad\quad\quad \ldots & \quad\quad \langle args, p'\rangle = decode_{ins}(format, [p], [prg], \mathcal{M}) \\
\quad\quad\quad opcode_n : inscomp(opcode_n, \mathcal{M})] & \quad\quad cont = \lambda a \rightarrow [\mathbf{emu_B}(a, prg);\ return] \\
\quad\quad \text{where } opcode_i \in domain(\mathcal{M}_{ins'}) &
\end{array}
$$

Figure 4.8: Emulator compiler

Example 4.2.3 (Encoding instructions)**:** Every combination of instruction name and format from Example 4.2.2, Figure 4.4, is assigned a different opcode. $\mathcal{M}_{ins'}$ retrieves both the corresponding instruction name and format for every opcode. In Figure 4.9, the sample $\mathcal{L}_a$ program at the top is translated by *encode* into the program $\mathcal{L}_b$ at the bottom, which can be interpreted by $\mathbf{int_2}$ using the definitions for $\mathcal{M}$. Figure 4.10 summarizes, for the first instruction in Figure 4.9, the role of the new components of the abstract machine definition when encoding and decoding instructions.

4.2.4 A Final Emulator

The interpreter $\mathbf{int_2}$ in Section 4.2.3 still has the overhead associated with using continuously the abstract machine definition $\mathcal{M}$. However, once $\mathcal{M}$ is fixed, it is possible to *instantiate* the parts of $\mathbf{int_2}$ which depend statically on $\mathcal{M}$, in order to obtain another emulator $\mathbf{int_3}$. This can be seen as a partial evaluation of $\mathbf{int_2}$ with respect to $\mathcal{M}$, i.e., $\mathbf{int_3} \equiv [\![spec]\!](\mathbf{int_2}, \mathcal{M})$. This returns an emulator written in $\mathcal{L}_c$ and without the burden of translating instructions in $\mathcal{L}_b$ to the level of $\mathcal{L}_a$ in order to access the corresponding code and argument definitions in $\mathcal{M}_{def}$ and $\mathcal{M}_{arg}$. Finally, and although $program[p]$ is not known at compile time, we can introduce a case statement which enumerates the possible values for the opcode. This is a common technique to make partial evaluation possible in cases where a given argument is not *static* [JGS93], but the set of values the argument can take is finite.

Since the structure of the interpreter is fixed, a compiler of emulators could

$\mathcal{L}_a$ **program:**

```
move  r(0) r(2)
move  r(1) r(0)
move  r(2) r(1)
halt
```

$\mathcal{L}_b$ **program:**

0	0	2	0	1	0	0	2	1	4

Figure 4.9: Sample program

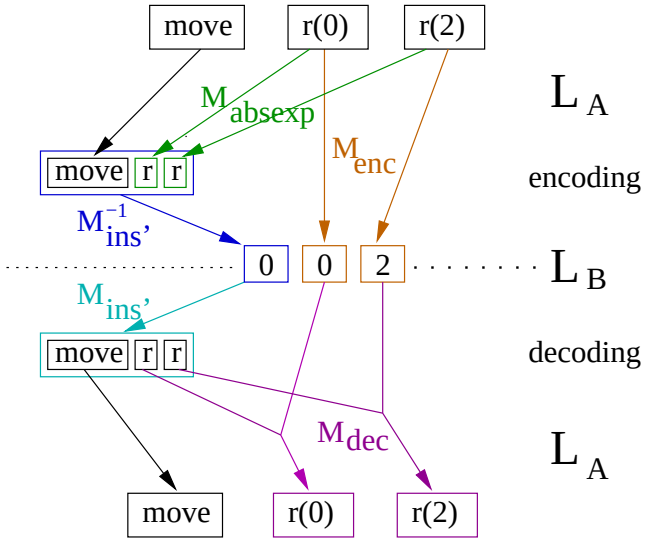

Figure 4.10: From symbolic code to bytecode and back

be generated by specializing the partial evaluator for the case of $\mathbf{int_2}$, i.e.,

$$emucomp : \mathcal{M} \rightarrow code_C$$
$$emucomp = [\![spec]\!](spec, \mathbf{int_2})$$

which is equivalent to the emulator compiler in Figure 4.8. It reuses the definition of $decode_{ins}$ seen in the previous section. Note that, as stated before, this emulator compiler has a regular structure, and we have opted to craft it manually, instead of relying on a self-applicable partial evaluator for $\mathcal{L}_c$. This emulator compiler, of course, does not need to be written in $\mathcal{L}_c$, and, in fact, in our particular implementation it is coded in Prolog and it generates emulators in C.

Example 4.2.4 (The generated emulator)**:** Figure 4.11 depicts an emulator for our working example, obtained by specializing $\mathbf{int_2}$ with respect to the machine

$$
\begin{array}{l}
\mathbf{emu_B}(p, program) \equiv \\
\quad \text{case } program[p] \text{ of} \\
\qquad 0 : reg[program[p+1]] := reg[program[p+2]]; \\
\qquad\quad \mathbf{emu_B}(p+3, program); return \\
\qquad 1 : \mathbf{emu_B}(program[p+1], program); return \\
\qquad 2 : push(p+2); \\
\qquad\quad \mathbf{emu_B}(program[p+1], program); return \\
\qquad 3 : \mathbf{emu_B}(pop(), program); return \\
\qquad 4 : return; return
\end{array}
$$

Figure 4.11: Generated emulator

definition in Example 4.2.3. Note the recursive call and *return*s at the end of every case branch which ensure that no other code after those statements is executed. All the recursive calls are, in fact, tail recursions: even if there are statements after them, these are *return*s which just pop return addresses. Therefore, there is no real work performed after the return from any recursive calls.

4.3 An Example Application: Minimal and Alternative Emulators

We will illustrate our technique with two (combined) applications: generating WAM emulators which are specialized for executing a fixed set of programs, and using different implementations of the WAM data structures. The former is very relevant in the context of applications meant for embedded devices and pervasive computing. It implements an automatic specialization scheme which starts at the Prolog level (by taking as input the programs to be executed) and, by slicing the abstract machine definition, traverses the compilation chain until a final emulator, tailored to these programs, is generated. The latter makes it possible to easily experiment with alternative implementation decisions.

We have already introduced how a piecewise definition of an abstract machine can make emulator generation automatic. In the rest of this section we will see how this technique can be used in order to generate such application-specific

emulators, and we will report on a series of experiments performed around these ideas. We will focus, for the moment, on generating correct emulators of *minimal size*, although the technique can obviously also be applied to investigating the impact of alternative implementations on performance.

4.3.1 Obtaining Specialized Emulators

The objective of specializing a program with respect to some criteria is to obtain a new program that preserves the initial semantics and is smaller or requires fewer operations. The source and target language are typically the same; this is expected, since specialization which operates across different translation levels is harder. It is however highly interesting, and applicable to several cases, such as the compilation to virtual machines and JIT compilation.

Among previous experiences which cross implementation boundaries we can cite [FD99], where automatically specialized WAM instructions are used as an intermediate step to generate C code which outperforms compilers to native code, and the Aquarius Prolog compiler [VD92] which carried analysis information along the compilation process to generate efficient native code.

As already mentioned, simplifying automatically hand-coded emulators (in order to speed them up or to reduce the executable size) written in $\mathcal{L}_c$ requires a specializer for $\mathcal{L}_c$ programs. Furthermore, in order to perform a good job, such specializer must be able to understand the emulator structure, a task which can be quite difficult for efficient, complex emulators. Even in the case that the emulator can be dealt with, there are very few information sources to use in order to perform useful optimizations: the input data is, in principle, any bytecode program.

One way to propagate bytecode properties about a particular program p down to the emulator so that the specializer can do some effective optimization is by partially evaluating the emulator w.r.t. p. This approach, though very powerful in principle, also has some drawbacks. First, the partial evaluator must be powerful enough so as to accurately handle the emulator, which is a complex piece of code. Second, partial evaluation is in general aimed at optimizing run time regardless of the size of the resulting program. When specializing w.r.t. large bytecode programs, partial evaluation can potentially produce very large programs. Also, this approach will be affected by some of the advantages and disadvantages of

native code vs. bytecode-based systems in issues such as portability, compactness, etc.

An alternative approach which is guaranteed not to produce code explosion is to express the specialization of the emulator in terms of *slicing* [Tip95, RT96, Wei99]. The aim of slicing is, given a program P and a certain *slicing criterion* ϕ, to obtain another program P_ϕ which has been obtained from P by removing statements and arguments which are not required for the slicing criterion ϕ. A slicing algorithm and the properties of the emulator input that it focuses on, ϕ, can be defined. Examples of those properties are bytecode reachable points, output variables, etc., which are needed to ensure that the semantics of the execution of the bytecode are preserved. Then, the slicing algorithm will transform the emulator so that only the parts of the emulator needed to maintain those properties (or a conservative approximation thereof) are kept.

One problem with this approach is that the bytecode is quite low level and the emulator too complicated to be automatically manipulated. However, our emulator generation scheme makes this problem more tractable. In our case $\mathcal{L}_b$ programs are generated from a higher-level representation which can be changed quite freely (even enriched with compiler-provided information to be later discarded by *encode*) and which aims at being easily manageable rather than efficient. It seems therefore more convenient to work at the level of $\mathcal{L}_a$ to extract the slicing information, since it offers more simplification opportunities. It has to be noted that transforming the emulator code, written in $\mathcal{L}_c$, using some $\mathcal{L}_a$ properties may be extremely difficult: to start with, suitable tools to work with $\mathcal{L}_c$ are needed, and they should be able to understand the relationship between $\mathcal{L}_b$ and $\mathcal{L}_a$ elements. It is much easier to work at the level of the definition of the abstract machine $\mathcal{M}$, where $\mathcal{L}_a$ is completely captured, and where its relationship with $\mathcal{L}_b$ is contained.

We therefore formulate a slicing transformation that deals directly with $\mathcal{M}$ and whose result is used to generate a sliced emulator $\mathbf{emu_s}$:

$$\mathbf{emu_s} = emucomp([\![slice_{\mathcal{M}}]\!](\mathcal{M}, \phi))$$

$\mathbf{emu_s}$ can also be viewed as the result of slicing $emucomp(\mathcal{M})$ (i.e., $\mathbf{emu_B}$) with a particular slicing algorithm that, among other things, preserves the (loop)

structure of the emulator.[3] That is, $slice_{\mathcal{M}}$ deals with the instruction set or the instruction code definitions, and leaves complex data and control issues, quite common in efficient emulators, untouched and under the control of *emucomp*. Slicing can change all the components of the definition of $\mathcal{M}$, including $\mathcal{M}_{def}$, which may cause the compiled emulator to lose or specialize instructions. Note that when $\mathcal{M}_{ins}$ is modified, the transformation affects the compiler, because the *encode* function uses definitions in $\mathcal{M}$.

4.3.2 Some Examples of Opportunities for Simplification

There is a variety of simplifications at the level of $\mathcal{M}$ that preserve the loop structure. They can be expressed in terms of the previously presented technique.

Instruction removal: Programs compiled into bytecode can be scanned and brought back into $\mathcal{L}_a$ using $\mathcal{M}_{ins'}$ to find the set I of instructions used in them. $\mathcal{M}$ is then sliced with respect to I and a new, specialized emulator is created as in Section 4.2.4. The new emulator may be leaner than the initial one since it probably has to interpret fewer instructions.

Removing format support: If $\mathcal{L}_a$ has instructions which admit arguments of different types (e.g., arithmetical operations which admit both integers and floating point numbers), programs that only need support for some of the available types can be executed in a reduced emulator. This can be achieved, again, by slicing $\mathcal{M}$ with respect to the remaining instruction and argument formats.

Removing specialized instructions: $\mathcal{M}$ can define specialized instructions (for example, for special argument values) or collapsed instructions (for often-used instruction sequences). Those instructions are by definition redundant, but sometimes useful for the sake of performance. However, not all programs require or benefit from them. When the compiler to $\mathcal{L}_a$ can selectively choose whether using or not those versions, a smaller emulator can be generated.

Obtaining the optimal set of instructions (w.r.t. some metric) for a particular program is an interesting problem. It is however out of the scope of this work.

[3] Due to the simplicity of the interpreter scheme, this is not a hard limitation for most emulator transformations, as long as the transformation output is another emulator.

4.3.3 Experimental Evaluation

We tested the feasibility of the techniques proposed herein for the particular case of the compilation of Prolog to a WAM-based bytecode. We started off with Ciao [BCC+02], a real, full-fledged logic programming system, featuring a (Ciao-)Prolog to WAM compiler, a complex bytecode interpreter (the emulator) written in C, and the machinery necessary to generate multi-platform, bytecode-based executables. We refactored the existing emulator as an abstract machine as described in the previous sections, and we implemented an emulator compiler which generates emulators written in C. We also implemented a slicer to remove unused instructions from the abstract machine definition.

Specialized emulators were built for a series of benchmark programs. For each of them, the WAM code resulting from its compilation was scanned to collect the set I of actually used instructions, and the general instruction set $\mathcal{M}_{ins}$ was sliced with respect to I in order to remove unused instructions. The resulting description was used to encode the WAM code into bytecode and to generate the specialized emulator. We have verified that, when no changes are applied to the abstract machine description, the generated emulator and bytecode representation are as optimized as the original ones. Orthogonally, we defined three slightly different instruction sets and generated specialized emulators for each of these sets and each of the benchmark programs, and we measured the resulting size (Table 4.3).

The benchmarks feature both symbolic and numerical computation, and they are thus representative of several possible scenarios. Although this benchmark set includes some widely known programs, we briefly describe all of them in Table 4.2 for the sake of completeness.

Specially interesting is the final set of signal processing programs, which are part of a signal processing application for wearable computing. Although they are not large, they do perform a real-life task by spacializing CD-quality sound in real time when executed in a Gumstix small computer. A more detailed description of the application and characteristics appears in Chapter 7.

It has to be noted that, although most of these benchmarks are of moderate size, our aim in this section is precisely to show how to reduce automatically the footprint of an otherwise large engine for these particular cases. On the other hand, reduced size does not necessarily make them unrealistic, in the sense that they effectively perform non-trivial tasks. As an example, `stream_opt` pro-

hw	The ubiquitous "Hello world!" program.
boyer	Simplified Boyer-Moore theorem prover kernel.
crypt	Cryptoarithmetic puzzle involving multiplication.
deriv	Symbolic derivation of polynomials.
exp	Computation of 13^{7111} with a linear- and a logarithmic-time algorithm.
fact	Compute the factorial of a number.
fib	Simply recursive computation of the n^{th} Fibonacci number.
knights	Chess knight tour, visiting only once every board cell.
nrev	Naive reversal of a list using append.
poly	Raises symbolically the expression `1+x+y+z` to the n^{th} power.
primes	Sieve of Eratosthenes.
qsort	Implementation of QuickSort.
queens11	Place N (with $N = 11$) chess queens in a $N \times N$ chessboard without mutual attacks.
query	poses a natural language query to a knowledge database containing information about country names, population, and area.
stream	generates 3-D stereo audio from monoaural audio, compass, and GPS signals to simulate the movement of a subject in a virtual world (see the case study in Chapter 7).
stream_dyn	An improved version of `stream` which can use any number of different input signals and sampling frequencies.
stream_opt	An optimized version where number of signals and sampling frequency is fixed.
tak	Computation of the Takeuchi function.

Table 4.2: Benchmark descriptions.

cesses audio in real time and with constant memory usage using Ciao Prolog in a 200MHz GumStix (a computer the size of a chewing gum).

The whole compilation process is fairly efficient. On a Pentium M at 1400MHz, with 512MB of RAM, and running Linux 2.6.10, the compiler compiles itself and generates a specialized emulator in 31.6 seconds: less than 0.1 seconds

	Basic				***ivect***				***iblt***			
	loop (29331)		**bytecode**		**loop (33215)**		**bytecode**		**loop (34191)**		**bytecode**	
	full	strip	full	strip	full	strip	full	strip	full	strip	full	strip
hw	28%	71%	33116	48	29%	74%	31548	48	29%	75%	31136	48
boyer	26%	46%	40198	8594	27%	50%	38606	8542	28%	52%	38168	8512
crypt	27%	58%	33922	2318	28%	62%	32306	2242	28%	63%	31842	2186
deriv	27%	56%	33606	2002	28%	59%	32022	1958	28%	61%	31606	1950
exp	28%	59%	32102	498	29%	63%	30542	478	29%	63%	30114	458
fact	28%	69%	31756	152	29%	72%	30216	152	29%	73%	29804	148
fib	28%	70%	31758	154	29%	74%	30218	154	29%	74%	29798	142
knights	27%	54%	32306	702	28%	56%	30726	662	29%	57%	30298	642
nrev	27%	65%	31866	262	28%	69%	30322	258	28%	70%	29910	254
poly	26%	48%	34682	3078	27%	52%	33098	3034	27%	53%	32664	3008
primes	27%	56%	32082	478	28%	61%	30526	462	29%	62%	30102	446
qsort	27%	58%	32334	730	28%	61%	30778	714	28%	62%	30370	714
queens11	28%	55%	32248	644	29%	59%	30696	632	29%	60%	30220	564
query	28%	59%	32816	1212	29%	63%	31256	1192	29%	64%	30840	1184
st. dyn	25%	42%	36060	2992	25%	45%	34420	2920	26%	45%	33890	2802
st. opt	26%	46%	35152	2084	26%	49%	33516	2016	26%	49%	32990	1902
stream	26%	46%	34496	1428	27%	49%	32868	1368	28%	49%	32402	1314
tak	28%	67%	31886	282	29%	70%	30334	270	29%	71%	29910	254
Average	27%	56%			27%	60%			28%	61%		

Table 4.3: Emulator sizes for different instruction sets

to generate the code of the emulator loop itself, 11.3 seconds to compile the compiler to bytecode (written in Prolog), and 20.3 seconds to compile all the C code, e.g., all Prolog-accessible predicates written in C (e.g., builtins and associated glue code) and the generated emulator using `gcc` with optimization grade `-O2`. Both the Prolog compiler and emulator generator are written in (Ciao-)Prolog.

The results of the benchmarks are in Table 4.3, were different instruction sets were used. Columns under the ***basic*** column correspond to the instruction set of the original emulator. The ***ivect*** column presents the case for an instruction set where several compact instructions which are specialized to move register values before calls to predicates have been added to the studied emulator. Finally, columns below the column ***iblt*** show results for an instruction set where specialized WAM instructions for the arithmetic builtins have been added to the emulator. In each of these set of columns, and for each benchmark, we studied the impact of specialization in the emulator size (the **loop** columns) and bytecode size (the **bytecode** columns).

The **bytecode** columns show two different figures: *full* is the bytecode size

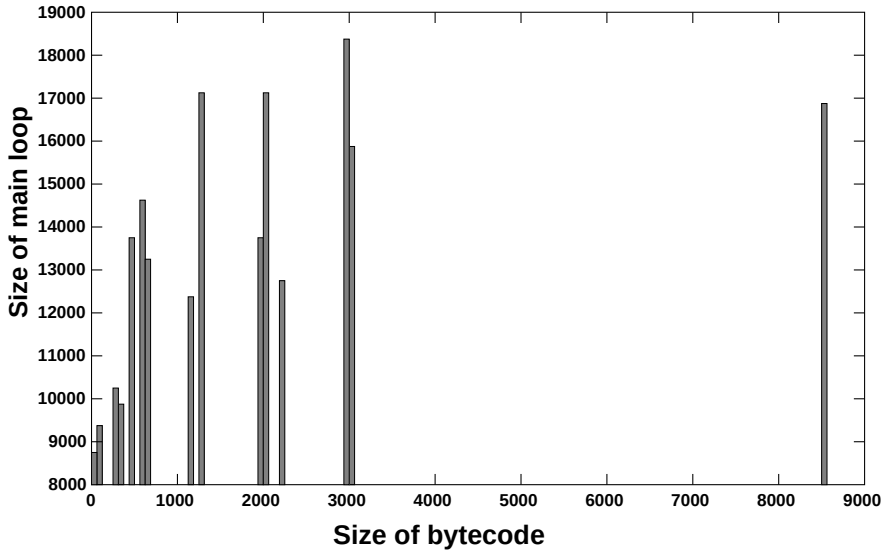

Figure 4.12: Relationship between stripped bytecode size (x axis) and emulator size (y axis)

including all libraries used by the program and the initialization code (roughly constant for every program) which is automatically added by the standard compiler. The numbers in the *strip* column were obtained after performing dead code elimination at the Prolog level (such as removing unused Prolog library modules and predicates, producing specialized versions, etc. using information from analysis — see, e.g., [PH03] and its references) and then generating the bytecode. This specialization of Prolog programs at the source and module level is done by the Ciao preprocessor and is beyond the scope of this work.

The **loop** columns contain, right below the label, the size of the main loop of the standard emulator with no specialization. For each benchmark we also show the percentage of *reduction* achieved with bytecode generated from full or specialized program with respect to the original, non-specialized emulator — the higher, the more savings.

Even in the case when the emulator is specialized with respect to the *full* bytecode, we get a steady savings of around 27%, including library and initialization code. We can deduce that this is a good approximation of the amount of reduction that can be expected from typical programs where no redundant code exists. Of course, programs which use all the available WAM instructions can

be crafted on purpose, but this is not the general case. In our experience, not even the compiler itself uses all the abstract machine instructions: we also generated an abstract machine specialized for it which was simpler (although only marginally) than the original one.

The savings obtained when the emulator is generated from specialized bytecode are more interesting. Savings range from 45% to 75%, averaging 60%. This shows that substantial size reductions can be obtained with our technique. The absolute sizes do not take into account ancillary pieces, such as I/O and operating system interfaces, which would be compiled or not with the main emulator as necessary, and which are therefore subject to a similar process of selection.

It might be expected that smaller programs would result in more emulator minimization. In general terms this is so, but with a wide variation, as can be seen in Figure 4.12. Thus, predicting in advance which savings will be obtained from a given benchmark in a precise way is not immediate.

4.4 Conclusions

We have presented the design and implementation of an emulator compiler that generates efficient code using a high-level description of the instruction set of an abstract machine and a set of rules which define how intermediate code is to be represented as bytecode. The approach allowed separating details of the low-level data and code representation from the set of instructions and their semantics. We were therefore able to perform, at the abstract machine description level, transformations which affect both the bytecode format and the generated emulator without sacrificing efficiency.

We have applied our emulator compiler to a description of the abstract machine underlying a production, high-quality, hand-written emulator. The automatically generated emulator is as efficient as the original one. By using a slicer at the level of the abstract machine definition, we were able to reduce automatically its instruction set, producing a smaller, dedicated, but otherwise completely functional, emulator. By changing the definition of the code corresponding to the instructions we were able to produce automatically emulators with substantial internal implementation differences, but still correct and efficient.

There is also a strong connection with Chapter 3: the fundamental pieces of

the C code generation performed there and the code definitions for instructions in $\mathcal{L}_a$ are intimately related, and we have reached a single abstract machine definition in the Ciao system which is used both to generate bytecode emulators and to compile to C code.

5

Description and Optimization of Abstract Machines in a Dialect of Prolog

Summary

In order to achieve competitive performance, abstract machines for Prolog and related languages end up being large and intricate, and incorporate sophisticated optimizations, both at the design and at the implementation levels. At the same time, efficiency considerations make it necessary to use low-level languages in their implementation. This makes them laborious to code, optimize, and, especially, maintain and extend. Writing the abstract machine (and ancillary code) in a higher-level language can help tame in part this inherent complexity. In this paper we show how the semantics of most basic components of an efficient virtual machine for Prolog can be described using (a variant of) Prolog which retains much of its semantics. These descriptions are then compiled to C and assembled to build a complete bytecode emulator. Thanks to the high level of the language used and its closeness to Prolog, the abstract machine description can be manipulated using standard Prolog compilation and optimization techniques with relative ease. We also show how, by applying program transformations selectively, we obtain abstract machine implementations whose performance can match and even exceed that of state-of-the-art, highly-tuned, hand-crafted emulators.

5.1 Introduction

Abstract machines have shown useful in order define theoretical models and implementations of software and hardware systems. In particular, they have been widely used to define execution models and as implementation vehicles for many languages, most frequently in functional and logic programming, but more recently also in object-oriented programming, with the Java abstract machine [GJSB05] being a very popular recent example. There are also early examples of the use of abstract machines in traditional procedural languages (e.g., the P-Code used as target in early Pascal implementations [NAJ+81]) and in other realms such as, for example, operating systems (e.g., Dis, the virtual machine for the Inferno operating system [DPP+97]).

The practical applications of the indirect execution mechanism that an abstract machine represents are countless: portability, generally small executables, simpler security control through sandboxing, increased compilation speed, etc. However, unless the abstract machine implementation is highly optimized, these benefits can come at the cost of poor performance. Designing and implementing fast, resource-friendly, competitive abstract machines is a complex task. This is specially so in the case of programming languages where there is a wide separation between many of their basic constructs and features and what is available in the underlying off-the-shelf hardware: term representation vs. memory words, unification vs. assignment, automatic vs. manual memory management, destructive vs. non-destructive updates, backtracking and tabling vs. Von Neumann-style control flow, etc. In addition, the extensive code optimizations required to achieve good performance make development and, especially, maintenance and further modifications non-trivial. Implementing or testing new optimizations is often involved, as decisions previously taken need to be revisited and low-level, tedious recoding is often necessary to test a new idea.

Improved performance has been achieved by post-processing the input program (often called *bytecode*) of the abstract machine (*emulator*) and generating efficient native code — sometimes closer to an implementation directly written in C. This is technically challenging and it makes the overall picture more complex when bytecode and native code can be combined, usually by dynamic recompilation. This in principle combines the best of both worlds and deciding when and

how native code (which may be large and/or costly to obtain) is generated based on runtime analysis of the program execution, and leaving the rest as bytecode. Some examples are the Java HotSpot VM [PVC01], the Psyco [Rig04] extension for Python, or for logic programming, all-argument predicate indexing in recent versions of Yap Prolog [SCSL07b], the dynamic recompilation of asserted predicates in BinProlog [Tar06], etc. Note, however, that the initial complexity of the virtual machine and all of its associated maintenance issues have not disappeared, and emulator designers and maintainers still have to struggle against thousands of lines of low level code.

A promising alternative, on which this paper focuses, is to explore the possibility of rewriting most of the runtime and virtual machine in the source, high-level language (or a close dialect of it), and, use all the available compilation machinery to obtain native code from it. Ideally, this native code should provide a runtime comparable to the hand-crafted one in terms of performance, while keeping the amount of low-level parts to a minimum. This is the approach taken in this work, where we apply it to to write Prolog emulators in a Prolog dialect. As we will see later, the approach is interesting not only to simplify the task of developers but also to widen the application domain of the language to other kind of problems which could extend far from just emulators, mature compilation technology, reuse analysis and transformation passes, and make easier automatizing tedious optimization techniques for emulators (such as specializing emulator instructions). The advantages of using a higher-level language are rooted, on one hand, on the capability of hiding implementation details that a higher-level language provides and, on the other hand, on its amenability to transformation and manipulation. These, as we will see, are key for our goals, as they reduce error-prone, tedious programming work, while making it possible to describe at the same time complex abstract machines in a concise and correct way.

A similar objective has been pursued elsewhere. For example, the JavaInJava [Tai98] and PyPy [RP06] projects aimed similar goals. The initial performance figures reported for these implementations highlight how challenging it is to make them competitive with existing hand-tuned abstract machines: JavaInJava started with an initial slowdown of approximately 700 times w.r.t. then-current implementations, and PyPy started at the 2000÷ slowdown level. Competitive execution times were only possible after changes in the source language and com-

pilation tool chain, by restricting it or adding annotations. For example, the slowdown of PyPy was reduced to 3.5÷ – 11.0÷ w.r.t. CPython [RP06]. These results can be partially traced back to the attempt to reimplement the whole machinery at once, which has the disadvantage of making such a slowdown almost inevitable. This makes it difficult to use the generated virtual machines as "production" software (which would therefore be routinely tested) and, especially, it makes it difficult to study how a certain, non-local optimization will carry over to a complete, optimized abstract machine.

Therefore, and we think that interestingly, in our case we chose an alternative approach: gradually porting selected key components (such as, e.g., instruction definitions) and combining this port with other emulator generation techniques [MCPH05]. At every step we made sure experimentally[1] that the performance of the original emulator was maintained throughout the process. The final result is an emulator completely written in a high-level language which, when compiled to native code, does not loss any performance w.r.t. a manually written one — and, as a very relevant by product, a language and a compiler thereof of a high-level enough to make several program transformation and analysis techniques applicable, while offering the possibility of being compiled into efficient native code. While this language is a general-purpose one and it can be used to implement arbitrary programs, throughout this paper we will focus on its use to write abstract machines and to easily generate variations.

The rest of the paper will proceed as follows: Section 5.2 gives an overview of the different parts of our compilation architecture and information flow and compares it with a more traditional setup. Section 5.3 presents the source language with which we will write our abstract machine, and we will justify the design decisions (e.g., typing, destructive updates, etc.) based on the needs of applications which demand high performance. Section 5.3.4 summarizes how compilation to efficient native is done through C. Section 5.4 describes language extensions which were devised specifically to write abstract machines (in particular, the WAM) and Section 5.5 explores how the added flexibility of the high-level language approach can be taken advantage of in order to easily generate variants

[1] And also with stronger means: for some core components we checked that the binary code produced from the high-level definition of the abstract machine and that coming from the hand-written one were identical.

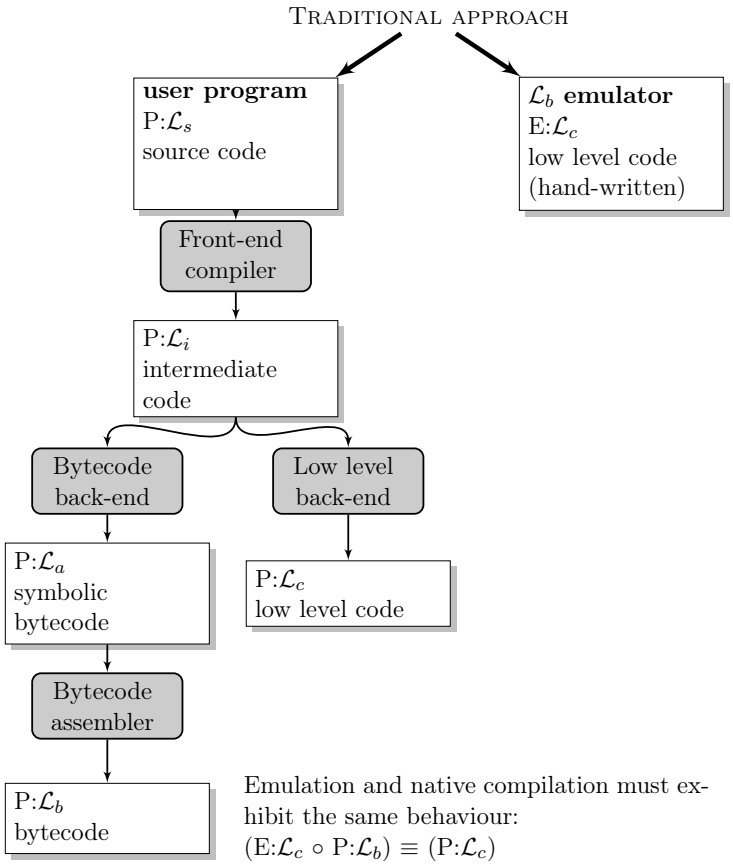

Figure 5.1: Traditional compilation architecture.

of abstract machines with different core characteristics. We experimentally study the performance that can be attained with these variants.

5.2 Overview of our Compilation Architecture

The compilation architecture we present here uses several different languages, language translators, and program representations which must be defined and placed in the "big picture". For generality, and since those elements are common to most bytecode-based systems, we will refer to them by more abstract names when possible or convenient, although in our initial discussion we will equate them with the actual languages in our production environment.

The starting point in this work is the Ciao Prolog system [HBC+08], contain-

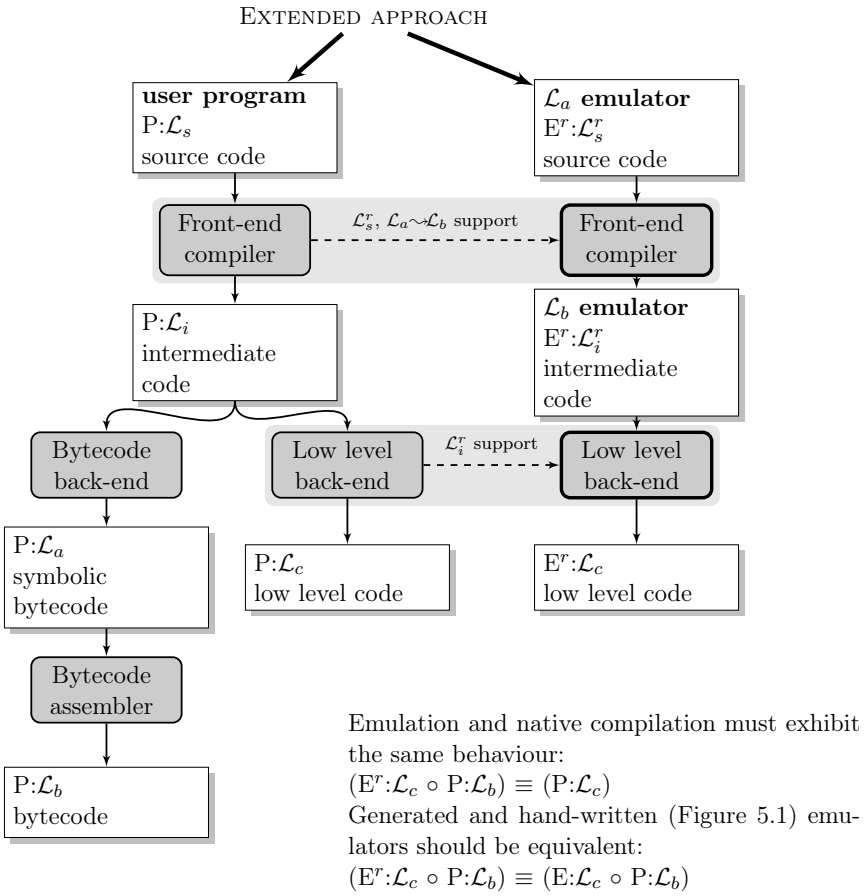

Figure 5.2: Extended compilation architecture.

ing an efficient, WAM-based [War83, AK91], abstract machine coded in C (an independent evolution which forked from the SICStus 0.5/0.7 virtual machine), the compiler to bytecode with an experimental extension to emit optimized C code described in Chapter 3, and the CiaoPP preprocessor (a program analysis, specialization and transformation framework) [BDD+97, HPB99, PBH00b, HPBG05].

We will denote the source Prolog language as $\mathcal{L}_s$, the symbolic WAM code as $\mathcal{L}_a$, the byte-encoded WAM code as $\mathcal{L}_b$, and the C language as $\mathcal{L}_c$. The different languages and translation processes described in this section are typically found in most systems, and this scheme could be easily applied to other situations. We will use N:L to denote the program N written in language L. Thus, we can distinguish in the *traditional approach* (Figure 5.1):

Front-end compiler: P:$\mathcal{L}_s$ is compiled to P:$\mathcal{L}_i$, where $\mathcal{L}_s$ is the source language and $\mathcal{L}_i$ is an intermediate representation. For simplicity, we suppose that this phase includes any analysis, transformation, and optimization.

Bytecode back-end: P:$\mathcal{L}_i$ is translated to P:$\mathcal{L}_a$, where $\mathcal{L}_a$ is the symbolic representation of the bytecode language.

Bytecode assembler: P:$\mathcal{L}_a$ is encoded into P:$\mathcal{L}_b$, where $\mathcal{L}_b$ defines encoded bytecode streams (a sequence of bytes whose design is focused on interpretation speed).

To execute $\mathcal{L}_b$ programs, a hand-written emulator E:$\mathcal{L}_c$ (where $\mathcal{L}_c$ is a lower-level language) is required, in addition to a collection of runtime code (written in $\mathcal{L}_c$). Alternatively, as commented before, more advanced systems are able to directly translate intermediate code by means of a **low level back end**, which compiles a program P:$\mathcal{L}_i$ directly into its,P:$\mathcal{L}_c$ equivalent.[2] This avoids the need for any emulator if all programs are compiled to native code, but additional runtime support is usually necessary.

The *classical architecture* needs manual coding of a large and complex piece of code, the emulator, using a low level language, often missing the opportunity to reuse compilation techniques that may have been developed for high-level languages, not only to improve efficiency, but also program readability, correctness, etc.

In the *extended compilation scheme* we make the hypothesis that we can design a dialect from $\mathcal{L}_s$ and write the emulator for symbolic $\mathcal{L}_a$ code, instead of byte-encoded $\mathcal{L}_b$, in that language. We will call $\mathcal{L}_s^r$ the extended language and E^r:$\mathcal{L}_s^r$ the emulator. Note that the mechanical generation of an interpreter for $\mathcal{L}_b$ from an interpreter for $\mathcal{L}_a$ was previously described and successfully tested in the *emulator generator* in Chapter 4. Adopting it for this work was perceived as a correct decision, since it moves low-level implementation (and bytecode representation) issues to a translation phase, thus reducing the requirements on the language to preserve emulator efficiency, and in practice making the code easier to manage. The new translation pipeline, depicted in the *extended approach* path (Figure 5.2)

[2]Note that in JIT systems the low level code is generated from the encoded bytecode representation. For simplicity we keep the figure for static code generation, which takes elements of $\mathcal{L}_i$ as the input.

shows the following processes (the dashed lines represent the modifications that some of the elements undergo with respect to the original ones):

Extended front-end compiler: it compiles $\mathcal{L}_s^r$ programs into $\mathcal{L}_i^r$ programs (where $\mathcal{L}_i^r$ is the intermediate language with extensions required to produce efficient native code). This compiler includes the *emulator generator*. That framework makes it possible to write instruction definitions for an $\mathcal{L}_a$ emulator, a separate bytecode encoding rules, and process them together to obtain an $\mathcal{L}_b$ emulator. I.e., it translates $\mathrm{E}^r{:}\mathcal{L}_s^r$ to an $\mathrm{E}^r{:}\mathcal{L}_i^r$ emulator for $\mathcal{L}_b$.

Extended low level back-end: it compiles $\mathcal{L}_i^r$ programs into $\mathcal{L}_c$ programs. The resulting $\mathrm{E}^r{:}\mathcal{L}_i^r$ is finally translated to $\mathrm{E}^r{:}\mathcal{L}_c$, which should be equivalent (in the sense of producing the same output) to $\mathrm{E}{:}\mathcal{L}_c$.

The advantage of this scheme lies in its flexibility to share optimizations, analysis and transformations at different compilation stages (e.g., it could reuse the same machinery for partial evaluation, constant propagation, common subexpression elimination, etc.) which are commonly reimplemented for high- and low-level languages.

5.3 The imProlog Language

We describe in this section our $\mathcal{L}_s^r$ language, imProlog, and the analysis and code generation techniques used to compile it into highly efficient code.[3] This Prolog variant is motivated by the following problems:

- It is hard to guarantee that certain overheads in Prolog that are directly related with the language expressiveness (e.g., boxed data for dynamic typing, trailing for non-determinism, uninstantiated free variables, multiple-precision arithmetic, etc.) will always be removed by compile-time techniques.

[3]The name imProlog stands for *im*perative *Prolog*, because its purpose is to make typically imperative algorithms easier to express in Prolog, but minimizing and controlling the scope of impurities.

- Even if the overhead could be eliminated, the cost of some basic operations, such as modifying a single attribute in a custom data structure, is not constant in the declarative subset of Prolog. For example, the cost of replacing the value of an argument in a Prolog structure is, in most straightforward implementations, linear w.r.t. the number of arguments of the structure, since a certain amount of copying of the structure spine is typically involved. In contrast, replacing an element in a C structure is a constant-time operation.

We will now present the different elements that comprise the imProlog language and we will gradually introduce a number of restrictions on the kind of programs which we admit as valid. The main reason to impose these restrictions is to achieve the central goal in this paper: generating efficient emulators starting from a high-level language.

In a nutshell, the imProlog language both restricts and extends Prolog. The impure features (e.g., the dynamic database) of Prolog are not part of imProlog and only programs meeting some strict requirements about determinism, modes in the unification, and others, are allowed. In a somewhat circular fashion, the requirements we impose on these programs are those which allow us to compile them into efficient code. Therefore, implementation matters somewhat influence the design and semantics of the language. On the other hand, imProlog also extends Prolog in order to provide a single, well-defined, and amenable to analysis mechanism to implement constant-time access to data: a specific kind of mutable variables. Thanks to the restrictions aforementioned and this mechanism imProlog programs can be compiled into very efficient low-level (e.g., C) code.

Since we are starting from Prolog, which is well understood, and the restrictions on the admissible programs can be introduced painlessly (they just reduce the set of programs which are deemed valid by the compiler), we will start by showing, in the next section, how we tackle efficient data handling, which as we mentioned departs significantly, but in a controlled way, from the semantics of Prolog.

Notation. We will use lowercase math font for variables (x, y, z, ...) in the rules that describe the compilation and language semantics. Prolog variable names will be written in math capital font (X, Y, Z, ...). Keywords and identifiers in the

target C language use bold text (**return**). Finally, sans serif text is used for other names and identifiers (f, g, h, ...). The typography will make it possible to easily distinguish a compilation pattern for '$\mathsf{f}(a)$', where 'a' may be any valid term, and '$\mathsf{f}(A)$', where 'A' is a Prolog variable with the concrete name "A". Similarly, the expression $f(a_1, ..., a_n)$ denotes any structure with functor name f and n arguments, whatever they may be. It differs from $\mathsf{f}(A_1, ..., A_n)$, where the functor name is fixed to f and the arguments are variables). If n is 0, the previous expression is tantamount to just f.

5.3.1 Efficient Mechanisms for Data Access and Update

In this section we will describe the formal semantics of *typed mutable variables*, our proposal to provide efficient (in terms of time and memory) data handling in imProlog. These variables feature backtrackable destructive assignment and are accessible and updatable in constant time through the use of a unique, associated identifier. This is relevant for us as it is required to efficiently implement a wide variety of algorithms, some of which appear in WAM-based abstract machines,[4] which we want to describe in enough detail as to obtain efficient executables.

There certainly exist a number options for implementing constant-time data access in Prolog. Dynamic predicates (assert / retract), can in some cases (maybe with the help of type information) provide constant-time operations; existing destructive update primitives (such as setarg/3) can do the same. However, it is difficult to for the analyses normally used in the context of logic programming deal with them in a precise way, in a significant part because their semantics was not devised with analysis in mind, and therefore they are difficult to optimize as much as we need herein.

Therefore, we opted for a generalization of mutable variables with typing constraints as mentioned before. In our experience, this fits in a less intrusive way with the rest of the logic framework, and at the same time allows us to generate efficient code for the purpose of the work in this paper.[5] Let us introduce some

[4] One obvious example is the unification algorithm of logical variables, itself based on the Union-Find algorithm. An interesting discussion of this point is available in [SF06], where a CHR version of the Union-Find algorithm is implemented and its complexity studied.

[5] Note that this differs from [MCH07], where changes in mutable data were non-backtrackable side effects. Notwithstanding, performance is not affected in this work, since we restrict at

preliminary definitions before presenting the semantics of mutable variables:

Type: τ is a unary predicate that defines a set of values (e.g., regular types as in [DZ92, GdW94, VB02]). If $\tau(v)$ holds, then v is said to contain values of type τ.

Mutable identifier: the identifier id of a mutable is a unique atomic value that uniquely identifies a mutable and does not unify with any other non-variable except itself.

Mutable store: φ is a mapping $\{id_1/(\tau_1,\ val_1),\ ...,\ id_n/(\tau_n,\ val_n)\}$, where id_i are mutable identifiers, val_i are terms, and τ_i type names. The expression $\varphi(id)$ stands for the pair $(\tau,\ val)$ associated with id in φ, while $\varphi[id/(\tau,\ val)]$ denotes a mapping φ' such that

$$\varphi'(id_i) = \begin{cases} (\tau,\ val) & \text{if } id_i = id \\ \varphi(id_i) & \text{otherwise} \end{cases}$$

We assume the availability of a function $\mathsf{new\_id}(\varphi)$ that obtains a new unique identifier not present in φ.

Mutable environment: By $\varphi_0 \rightsquigarrow \varphi \ \vdash \ g$ we denote a judgment of g in the context of a mutable environment $(\varphi_0,\ \varphi)$. The pair $(\varphi_0,\ \varphi)$ makes the relation between the initial and final mutable stores, respectively, and the interpretation of the g explicit.

We can now define the rules that manipulate mutable variables (Figure 5.3). For the sake of simplicity, they do not impose any evaluation order. In practice, and in order to keep the computational cost low, we will use the Prolog resolution strategy, and impose limitations on the instantiation level of some particular terms; we postpone discussing this issue until Section 5.3.2:

Creation: The (M-NEW) rule defines the creation of new mutable placeholders. A goal $m = \mathsf{initmut}(\tau,\ v)$[6] checks that the term v is a solution of τ (i.e., it

compile-time the emulator instructions to be deterministic.

[6]To improve readability, we use the functional notation of [CCH06] for the new initmut/3 and (@)/2 *built-in* predicates.

$$\text{(M-New)}\ \frac{\vdash\ \tau(v) \quad id = \mathsf{new\_id}(\varphi_0) \quad \varphi = \varphi_0[id/(\tau,\ v)] \quad \vdash\ m = id}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ m = \mathsf{initmut}(\tau,\ v)}$$

$$\text{(M-Read)}\ \frac{\vdash\ m = id \quad (\_,\ v) = \varphi(id) \quad \vdash\ x = v}{\varphi \rightsquigarrow \varphi\ \vdash\ x = m@}$$

$$\text{(M-Assign)}\ \frac{\vdash\ m = id \quad (\tau,\ \_) = \varphi_0(id) \quad \vdash\ \tau(v) \quad \varphi = \varphi_0[id/(\tau,\ v)]}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ m \Leftarrow v}$$

$$\text{(M-Type)}\ \frac{\vdash\ m = id \quad (\tau,\ \_) = \varphi(id)}{\varphi \rightsquigarrow \varphi\ \vdash\ \mathsf{mut}(\tau,\ m)}$$

$$\text{(M-Weak)}\ \frac{\vdash\ a}{\varphi \rightsquigarrow \varphi\ \vdash\ a}$$

$$\text{(M-Conj)}\ \frac{\varphi_0 \rightsquigarrow \varphi_1\ \vdash\ a \quad \varphi_1 \rightsquigarrow \varphi\ \vdash\ b}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ (a \wedge b)}$$

$$\text{(M-Disj-1)}\ \frac{\varphi_0 \rightsquigarrow \varphi\ \vdash\ a}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ (a \vee b)} \qquad \text{(M-Disj-2)}\ \frac{\varphi_0 \rightsquigarrow \varphi\ \vdash\ b}{\varphi_0 \rightsquigarrow \varphi\ \vdash\ (a \vee b)}$$

Figure 5.3: Rules for the implicit mutable store (operations and logical connectives).

has type τ), creates a new mutable identifier id that did not appear as a key in φ_0 and does not unify with any other term in the program, defines the updated φ as φ_0 where the value associated with id is v, and unifies m with id. These restrictions ensure, in practice, that m was an unbound variable before the mutable was created.

Access: Reading the contents of a mutable variable is defined in the (M-Read) rule. The goal $x = m@$ holds if the variable m is unified with a mutable identifier id, for which an associated v value exists in the mutable store φ, and the variable x unifies with v.[7]

Assignment: Assignment of values to mutables is described in the (M-Assign) rule. The goal $m \Leftarrow v$, which assigns a value to a mutable identifier, holds iff:

[7]Since the variable in the mutable store is constrained to the type, it is not necessary to check that x belongs to that type.

- m is unified with a mutable identifier id, for which a value is stored in φ_0 with associated type τ.
- v has type τ, i.e., the value to be stored is compatible with the type associated with the mutable.
- φ_0 is the result of replacing the associated type and value for id by τ and v, respectively.

Typing: The (M-TYPE) rule allows checking that a variable contains a mutable identifier of a given type. A goal $\mathsf{mut}(\tau,\ m)$ is true if m is unified with a mutable identifier id that is associated with the type τ in the mutable store φ.

Note that although some of the rules above enforce typing constraints, the compiler, as we will see, is actuallly able to statically remove these checks and there is no dynamic typing involved in the execution of admissible imProlog programs. The rest of the rules define how the previous operations on mutable variables can be joined and combined together, and with other predicates:

Weakening: The weakening rule (M-WEAK) states that if some goal can be solved without a mutable store, then it can also be solved in the context of a mutable store that is left untouched. This rule allows the integration of the new rules with predicates (and built-ins) that do not use mutables.

Conjunction: The (M-CONJ) rule defines how to solve a goal $a \wedge b$ (written as $(a,\ b)$ in code) in an environment where the input mutable store φ_0 is transformed into φ, by solving a and b and connecting the output mutable store of the former (φ_1) with the input mutable store of the later. This conjunction is not commutative, since the updates performed by a may alter the values read in b. If none of those goals modify the mutable store, then commutativity can be ensured. If none of them access the mutable store, then it is equivalent to the classic definition of conjunction (by applying the (M-WEAK) rule).

Disjunction: The disjunction of two goals is defined in the (M-DISJ) rule, where $a \vee b$ (written as $(a\ ;\ b)$ in code) holds in a given environment if either a or b holds in such environment, with no relation between the mutable

stores of both branches. That means that changes in the mutable store would be *backtrackable* (e.g., any change introduced by an attempt to solve one branch must be undone when trying another alternative). As with conjunction, if the goals in the disjunction do not update nor access the mutable store, then it is equivalent to the classic definition of disjunction (by applying the (M-WEAK) rule).

Mutable terms, conceptually similar to the mutables we present here, were introduced in SICStus Prolog as a replacement for setarg/3, and also appear in proposals for global variables in logic programming (such as [Sch97], Bart Demoen's implementation of (non)backtrackable global variables for hProlog/SWI-Prolog), or imperative assignment [GKPC85]. In the latter case there was no notion of types and the terms assigned to had to be (ground) atoms at the time of assignment.

We will consider that two types unify if their names match. Thus, typing in mutables divide the mutable store in separate independent regions, which will facilitate program analysis. For the purpose of this work we will treat mutable variables as a native part of the language. It would however be possible to emulate the mutable store as a pair of additional arguments, threaded from left to right in the goals of the predicate bodies. A similar translation is commonly used to implement DCGs or state variables in Mercury [SHC96].

5.3.2 Compilation Strategy and imProlog Subset Considered

In the previous section the operations on mutable data were presented separately from the host language, and no commitment was made regarding their implementation other than assuming that it could be done efficiently. However, when the host language $\mathcal{L}_s^r$ has a Prolog-like semantics (featuring unification and backtracking) and even if backtrackable destructive assignment is used, the compiled code can be unaffordably inefficient for deterministic computations unless extensive analysis and optimization is performed. On the other hand, the same computations may be easy to optimize in lower-level languages.

A way to overcome those problems is to specialize the translated code for a relevant subset of the initial input data. This subset can be *abstractly* specified: let us consider a predicate bool/1 that defines truth values, and a call bool(X)

where X is known to be always bound to a de-referenced atom. The information about the dereferencing state of X is a partial specification of the initial conditions, and replacing the call to the generic bool/1 predicate by a call to another predicate that implements a version of bool/1 that avoids unnecessary work (e.g., there is no need for *choice points* or *tag* testing on X) produces the same computation, but using code that is both shorter and faster. To be usable in the compilation process, it is necessary to propagate this knowledge about the program behavior as predicate-level assertions or program-point annotations, usually by means of automatic methods such as static analysis. Such techniques has been tested elsewhere [War77, Tay91b, VD92, Van94, MCH04].

This approach (as most automatic optimization techniques) has obviously its own drawbacks when high-performance is a requirement: *a*) the analysis is not always precise enough, which makes the compiler require manual annotations or generate under-optimized programs; *b*) the program is not always optimized, even if the analysis is able to infer interesting properties about it, since the compiler may not be clever enough to improve the algorithm; and *c*) the final program performance is hard to estimate, as we leave more optimization decisions to the compiler, of which the programmer may not be aware.

For the purposes of this paper, cases *a* and *b* do not represent a major problem. Firstly, if some of the annotations cannot be obtained automatically and need to be provided by hand, the programming style still encourages separation of optimization annotations (as hints for the compiler) and the actual algorithm, which we believe makes code easier manage. Secondly, we adapted the language imProlog and compilation process to make the algorithms implemented in the emulator easier to represent and compile. For case *c*, we took an approach different from that in other systems. Since our goal is to generate low level code that *ensures* efficiency, we impose some constraints on the compilation output to avoid generation of code known to be suboptimal. This restricts the admissible code and the compiler informs the user when the constraints do not hold, by reporting efficiency errors. This is obviously too drastic a solution for general programs, but we found it a good compromise in our application.

pred	::=	*head* :- *body*	(Predicates)
head	::=	$\lfloor\beta\rfloor id(var, ..., var)\lfloor\beta\rfloor$	(Heads)
body	::=	*goals* | (*goals* → *goals* ; *body*)	(Body)
goals	::=	$\lfloor\beta\rfloor goal, ..., \lfloor\beta\rfloor goal$	(Conjunction of goals)
goal	::=	*var* = *var* | *var* = *cons* |	(Unifications)
		var = *var*@ | *var* ⇐ *var* |	(Mutable ops.)
		id(*var*, ..., *var*)	(Built-In/User call)
var	::=	uppercase name	(Variables)
id	::=	lowercase name	(Atom names)
cons	::=	atom names, integers, floats, characters, etc.	(Other constants)
β	::=	abstract substitution	(Program point anots.)

Figure 5.4: Syntax of normalized programs.

Preprocessing imProlog Programs

The compilation algorithm starts with the expansion of syntactic extensions (such as, e.g., functional notation), followed by normalization and analysis. Normalization is helpful to reduce programs to simpler building blocks for which compilation schemes are described.

The syntax of the normalized programs is shown in Figure 5.4, and is similar to that used in `ciaocc` (Chapter 3). It focuses on simplifying code generation rules and making analysis information easily accessible. Additionally, operations on mutable variables are considered built-ins. Normalized predicates contain a single clause, composed of a head and a body which contains a conjunction of goals or *if-then-else*s. Every goal and head are prefixed with *program point* information which contains the abstract substitution inferred during analysis, relating every variable in the goal / head with an abstraction of its value or state. However, compilation needs that information to be also available for every temporal variable that may appear during code generation and which is not yet present in the normalized program. In order to overcome this problem, most auxiliary variables are already introduced before the analysis. The code reaches an homogeneous form by requiring that both head and goals contain only syntactical variables as arguments, and making unification and matching explicit in the body of the

clauses. Each of these basic data-handling steps can therefore be annotated with the corresponding abstract state. Additionally, unifications are restricted to the *variable-variable* and *variable-constant* cases. As we will see later, this is enough for our purposes.

Program Transformations. Normalization groups together the bodies of the clauses of the same predicate in a disjunction, sharing common new variables in the head arguments, introducing unifications as explained before, and taking care of renaming variables local to each clause.[8] As a simplification for the purpose of this paper, we restrict ourselves to treating atomic, ground data types in the language. Structured data is created by invoking built-in or predefined predicates. Control structures such as disjunctions (_ ; _) and negation (\+ _) are only supported when they can be translated to *if-then-else*s, and a compilation error is emitted (and compilation is aborted) if they cannot. If cuts are not provided, mode analysis and determinism analysis helps in detecting the mutually exclusive prefixes of the bodies, and delimit them with cuts. Note that the restrictions that we impose on the accepted programs make it easier to treat some *non-logical* Prolog features, such as *red* cuts, which make language semantics more complex but are widely used in practice. We allow the use of *red* cuts (explicitly or as (... → ... ; ...) constructs) as long as it is possible to insert a mutually exclusive prefix in all the alternatives of the disjunctions where they appear: (b_1, !, b_2 ; b_3) is treated as equivalent to (b_1, !, b_2 ; \+ b_1, !, b_3) if analysis (e.g., [DLGH97]) is able to determine that b_1 does not generate multiple solutions and does not further instantiate any variables shared with the head or the rest of the alternatives.

Predicate and Program Point Information. The information that the analysis infers from (and is annotated in) the normalized program can be divided into predicate-level assertions and program point assertions. Predicate-level assertions relate an abstract input state with an output state ($\lfloor\beta_0\rfloor f(a_1, ..., a_n)\lfloor\beta\rfloor$), or state facts about some properties (see below) of the predicate. Given a predicate f/n, the properties needed for the compilation rules used in this work are:

- $\mathsf{det}(f/n)$: The predicate f/n is deterministic (it has exactly one solution).

[8] This is required to make their information independent in each branch during analysis and compilation .

- semidet(f/n): The predicate f/n is semideterministic (it has one or zero solutions).

We assume that there is a single call pattern, or that all the possible call patterns have been aggregated into a single one, i.e., the analysis we perform does not take into account the different modes in which a single predicate can be called. Note that this does not prevent effectively supporting different separate call patterns, as a previous specialization phase can generate a different predicate version for each calling pattern.

The second kind of annotations keeps track of the abstract state of the execution at each program point. For a goal $\lfloor\beta\rfloor g$, the following judgments are defined on the abstract substitution β on variables of g:

- $\beta \vdash \mathsf{fresh}(x)$: The variable x is a fresh variable (not instantiated to any value, not sharing with any other variable).

- $\beta \vdash \mathsf{ground}(x)$: The variable x contains a ground term (it does not contain any free variable).

- $\beta \vdash x{:}\tau$: The values that the variable x can take at this program point are of type τ.

Overview of the Analysis of Mutable Variables

The basic operations on mutables are restricted to some instantiation state on input and have to obey to some typing rules. In order to make the analysis as parametric as possible to the concrete rules, these are stated using the following assertions on the predicates @/2, initmut/3, and $\Leftarrow$/2:

:– **pred** @(+mut(T), $-T$).
:– **pred** initmut(+($\hat{}\,T$), +T, $-$mut(T)).
:– **pred** (+mut(T)) $\Leftarrow$ (+T).

These state that:

- Reading the value associated with a mutable identifier of type mut(T) (which must be ground) gives a value type T.

- Creating a mutable variable with type T (escaped in the assertion to indicate that it is the type name that is provided as argument, not a value of that type) takes an initial value of type T and gives a new mutable variable of type $\mathsf{mut}(T)$.

- Assigning a mutable identifier (which must be ground and of type $\mathsf{mut}(T)$) a value requires the last one to be of type T.

Those assertions are able to instruct the type analysis on the meaning of the built-ins, requiring no further changes w.r.t. equivalent[9] type analyses for plain Prolog. However, in our case more precision is needed sometimes. E.g., given $\mathsf{mut}(\mathsf{int},\ A)$ and $(A \Leftarrow \mathsf{3},\ \mathsf{p}(A\mathtt{@}))$ we want to infer that $\mathsf{p/1}$ is called with an integer value 3 and not with any integer (as inferred using just the assertion). With no information about the built-ins, that code is equivalent to $(T_0 = \mathsf{3},\ A \Leftarrow T_0,\ T_1 = A\mathtt{@},\ \mathsf{p}(T_1))$, and no relation between T_0 and T_1 is established.

However, based on the semantics of mutables variables and their operations (Figure 5.3), it is possible to define an analysis based on abstract interpretation to infer properties of the values stored in the mutable store. To natively understand the built-ins, it is necessary to abstract the mutable identifiers and the mutable store, and represent it in the abstract domain, for which different options exist.

One is to explicitly keep the relation between the abstraction of the mutable identifier and the variable containing its associated value. For every newly created mutable or mutable assignment, the associated value is changed, and the previous code would be equivalent to $(T = \mathsf{3},\ A \Leftarrow T,\ T = A\mathtt{@},\ \mathsf{p}(T))$. The analysis in this case will lose information when the value associated with the mutable is unknown. That is, given $\mathsf{mut}(\mathsf{int},\ A)$ and $\mathsf{mut}(\mathsf{int},\ B)$, it is not possible to prove that $A \Leftarrow \mathsf{3},\ \mathsf{p}(B\mathtt{@})$ will not call $\mathsf{p/1}$ with a value of 3.

Different abstractions of the mutable identifiers yield different precision levels in the analysis. E.g., given an abstract domain for mutable identifiers that distinguishes newly created mutables, the chunk of code $(A = \mathsf{initmut}(\mathsf{int},\ \mathsf{1}),\ B = \mathsf{initmut}(\mathsf{int},\ \mathsf{2}),\ A \Leftarrow \mathsf{3},\ \mathsf{p}(B\mathtt{@}))$ has enough information to ensure that B is unaffected by the assignment to A. In the current state, and for the purpose of the paper, the abstraction of mutable identifiers is able to take into account newly

[9] In the sense that the definition of how built-ins behave are not hard-wired into the analysis itself.

created mutables and mutables of a particular type. When an assignment is performed on an unknown mutable, it only needs to change the values of mutables of exactly the same type, improving the precision.[10] If mutable identifiers are interpreted as pointers, that problem is related to *pointer aliasing* in imperative programming [HP00].

5.3.3 Data Representation and Operations

Data representation in most Prolog systems often chooses a general mapping from Prolog terms and variables to C data so that full unification and backtracking can be implemented. However, for the logical and mutable variables of imProlog, we need the least expensive mapping to C types and variables possible, since anything else would bring an unacceptable overhead in critical code (such as emulator instructions). A general way to overcome this problem, which is taken in this work, is to start from a general representation, fixing data at compile time, and replacing it by a more specific encoding.

Let us recall the general representation in WAM-based implementations [War83, AK91]. The abtract machine state is composed of a set of registers and stacks of memory cells. The values stored in those registers and memory cells are called *tagged* words. Every tagged word has a *tag* and a *value*, the tag indicating the kind of value stored in it. Possible values are constants (such as integers up to some fixed length, indexes for atoms, etc.), or pointers to larger data which does not fit in the space for the value (such as larger integers, multiple precision numbers, floating point numbers, arrays of arguments for structures).[11] There exist a special *reference* tag that indicates that the value is a pointer to another cell. That reference tag allows a cell to point to itself (for unbound variables), or set of cells point to the same value (for unified variables). As we had the assumption that mutable variables can be efficiently implemented, we want to point out that these representations can be extended for this case, using, for example, an additional *mutable* tag, to denote that the value is a pointer to another cell which contains the associated value. How terms are created and unified using the a

[10] That was enough to specialize pieces of imProlog code implementing the unification of tagged words, which was previously optimized by hand.

[11] Those will be ignored in this paper, since all data can be described using atomic constants, mutable identifiers, and built-ins to control the emulator stacks.

tagged cell representation is well described in the relevant literature.

When a Prolog-like language is (naively) translated to C, a large part of the overhead comes from the use of tags (including bits reserved for automatic memory management) and machine words with fixed sizes (e.g., `unsigned int`) for tagged cells. If we are able to enforce a fixed tag for every variable (which we can in principle map to a word) at compile time at every program point, those additional tag bits can be removed from the representation and the whole machine word can be used for the value. Then, it is possible to make use of different C types for each kind of value (e.g., `char`, `float`, `double`, etc.). Moreover, the restrictions that we have imposed on program determinism (Section 5.3.2), variable scoping, and visibility of mutable identifiers make trailing unnecessary.

C Types for Values

The associated C type that stores the value for an imProlog type τ is defined in a two-layered approach. First, the type τ is inferred by means of Prolog type analysis. In our case we are using, a regular type analysis [VB02], which can type more programs than a Hindley-Damas-Milner type analysis [DM82]. Then, for each variable a compatible *encoding type*, which contains all the values that the variable can take, is assigned, and the corresponding C type for that encoding type is used. Encoding types are Prolog types annotated with an assertion that indicates the associated C type. A set of heuristics is used to assign economic encodings so that memory usage and type characteristics are adjusted as tightly as possible. Consider, for example, the type flag/1 defined as

```
:- regtype flag/1 + low(int32).
flag := off | on.
```

It specifies that the values in the declared type must be represented using the `int32` C type. In this case, off will be encoded as `0` and on encoded as `1`. The set of available encoding types must be fixed at compile time, either defined by the user or provided as libraries. Although it is not always possible to automatically provide a mapping, we believe that this is a viable alternative to more restrictive typing options such as Hindley-Damas-Milner based typings. A more detailed description of data types in imProlog can be seen in [MCH08].

Mapping imProlog Variables to C Variables

The logical and mutable variables of imProlog are mapped onto imperative, low level variables which can be global, local, or passed as function arguments. Thus, pointers to the actual memory locations where the value, mutable identifier, or mutable value are stored may be necessary. However, as stated before, we need to statically determine the number of references required. The *reference mode*s of a variable will define the shape of the memory cell (or C variable), indicating how the value or mutable value is accessed:

- **0v**: the cell contains the value.
- **1v**: the cell contains a pointer to the value.
- **0m**: the cell contains the mutable value.
- **1m**: the cell contains a pointer to the mutable cell, which contains the value.
- **2m**: the cell contains a pointer to another cell, which contains a pointer to the mutable cell, which contains the mutable value.

For an imProlog variable x, with associated C symbol x, and given the C type for its value $c\tau$, Table 5.1 gives the full C type definition to be used in the variable declaration, the *r-value* (for the left part of C assignments) and *l-value* (as C expressions) for the reference (or address) to the variable, the variable value, the reference to the mutable value, and the mutable value itself. These definitions relate C and imProlog variables and will be used later in the compilation rules. Note that the translation for ref_rval and val_lval is not defined for **0m**. That indicates that it is impossible to modify the mutable identifier itself for that mutable, since it is fixed. This tight mapping to C types, avoiding when possible unnecessary indirections, allows the C compiler to apply optimizations such as using machine registers for mutable variables.

The following algorithm infers the *reference mode* (refmode(_)) of each predicate variable making use of type and mode annotations:

1: Given the head $\lfloor\beta_0\rfloor f(a_1, ..., a_n)\lfloor\beta\rfloor$, the ith-argument mode argmode(f/n, i)

	refmode(x)				
	0v	**1v**	**0m**	**1m**	**2m**
Final C type	$c\tau$	$c\tau$ *	$c\tau$	$c\tau$ *	$c\tau$ * *
ref_rval⟦x⟧	&x	x	-	&x	x
val_lval⟦x⟧	x	*x	-	x	*x
val_rval⟦x⟧	x	*x	&x	x	*x
mutval/val_lval⟦x⟧			x	*x	**x
mutval/val_rval⟦x⟧			x	*x	**x

Table 5.1: Operation and translation table for different mapping modes of imProlog variables

for a predicate argument a_i, is defined as:

$$\mathsf{argmode}(f/n,\, i) = \begin{cases} \text{in} & \text{if } \beta \vdash \mathsf{ground}(a_i) \\ \text{out} & \text{if } \beta_0 \vdash \mathsf{fresh}(a_i),\ \beta \vdash \mathsf{ground}(a_i) \end{cases}$$

2: For each predicate argument a_i, depending on $\mathsf{argmode}(f/n,\, a_i)$:

- If $\mathsf{argmode}(f/n,\, a_i) = \text{in}$, then
  - if $\beta \vdash a_i{:}\mathsf{mut}(t)$ then $\mathsf{refmode}(a_i) = \mathbf{1m}$, else $\mathsf{refmode}(a_i) = \mathbf{0v}$.
- If $\mathsf{argmode}(f/n,\, a_i) = \text{out}$, then
  - if $\beta \vdash a_i{:}\mathsf{mut}(t)$ then $\mathsf{refmode}(a_i) = \mathbf{2m}$, else $\mathsf{refmode}(a_i) = \mathbf{1v}$.

3: For each unification $\lfloor\beta\rfloor\, a = b$:

- if $\beta \vdash \mathsf{fresh}(a)$, $\beta \vdash \mathsf{ground}(b)$, $\beta \vdash b{:}\mathsf{mut}(t)$, then $\mathsf{refmode}(a) = \mathbf{1m}$.
- Otherwise, if $\beta \vdash \mathsf{fresh}(a)$, then $\mathsf{refmode}(a) = \mathbf{0v}$.

4: For each mutable initialization $\lfloor\beta\rfloor\, a = \mathsf{initmut}(t,\, b)$:

- if $\beta \vdash \mathsf{fresh}(a)$, $\beta \vdash \mathsf{ground}(b)$, $\beta \vdash b{:}\ \mathsf{mut}(t)$, then $\mathsf{refmode}(a) = \mathbf{0m}$.

5: Any case not contemplated above is a compile-time error.

Escape analysis of mutable identifiers. According to the compilation scheme we follow, if a mutable variable identifier cannot be reached outside the scope of a predicate, it can be safely mapped to a (local) C variable. That requires the equivalent of escape analysis for mutable identifiers. A conservative approximation to decide that mutables can be assigned to local C variables is the following: the mutable variable identifier can be read from, assigned to, and passed as argument to other predicates, but it cannot be assigned to anything else than other local variables. This is easier to check and has been precise enough for our purposes.

5.3.4 Code Generation Rules

Schematically, compilation processes a set of predicates, each one composed of a *head* and *body* as defined in Section 5.3.2. The body can contain control constructs, calls to user predicates, calls to built-ins, and calls to external predicates written in C. For each of these cases we will summarize the compilation as translation rules, where p stands for the predicate compilation output that stores the C functions for the compiled predicates. The compilation state for a predicate is denoted as θ, and it is composed of a set of variable declarations and a mapping from identifiers to *basic blocks*. Each basic block, identified by δ, contains a sequence of sentences and a terminal control sentence.

Basic blocks are finally translated to C code as labels, sequences of sentences, and jumps or conditional branches generated as *goto*s and *if-then-else*s. Note that the use of labels and jumps in the generated code should not make the C compiler generate suboptimal code, as simplification of control logic to basic blocks and jumps is one of the first steps performed by C compilers. It was experimentally checked that using *if-then-else* constructs (when possible) does not necessarily help mainstream C compilers in generating better code. In any case, doing it so is a code generation option.

For simplicity, in the following rules we will use the syntax $\langle\theta_0\rangle\ \forall i{=}1..n\ \ g\ \langle\theta_n\rangle$ to denote the evaluation of g for every value of i between 1 and n, where the intermediate states θ_j are adequately threaded to link every state with the following one.

$$\text{(Conj)}\quad \frac{\begin{array}{l}\langle\theta_0\rangle\ \mathsf{bb\_new} \Rightarrow \delta_b\ \langle\theta_1\rangle \\ \langle\theta_1\rangle\ \mathsf{gcomp}(a,\ \eta[\mathsf{s}{\mapsto}\delta_b],\ \delta) \Rightarrow \langle\theta_2\rangle \\ \langle\theta_2\rangle\ \mathsf{gcomp}(b,\ \eta,\ \delta_b) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}((a,\ b),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(IfThenElse)}\quad \frac{\begin{array}{l}\langle\theta_0\rangle\ \mathsf{bb\_newn}(2) \Rightarrow [\delta_t,\ \delta_e]\ \langle\theta_1\rangle \\ \langle\theta_1\rangle\ \mathsf{gcomp}(a,\ \eta[\mathsf{s}{\mapsto}\delta_t,\ \mathsf{f}{\mapsto}\delta_e],\ \delta) \Rightarrow \langle\theta_2\rangle \\ \langle\theta_2\rangle\ \mathsf{gcomp}(\mathit{then},\ \eta,\ \delta_t) \Rightarrow \langle\theta_3\rangle \quad \langle\theta_3\rangle\ \mathsf{gcomp}(\mathit{else},\ \eta,\ \delta_e) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}((a \rightarrow \mathit{then}\ ;\ \mathit{else}),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(True)}\quad \frac{\langle\theta_0\rangle\ \mathsf{emit}(\mathbf{goto}\ \eta(\mathsf{s}),\ \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{gcomp}(\mathsf{true},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle} \qquad \text{(Fail)}\quad \frac{\langle\theta_0\rangle\ \mathsf{emit}(\mathbf{goto}\ \eta(\mathsf{f}),\ \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{gcomp}(\mathsf{fail},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

Figure 5.5: Control compilation rules.

Compilation of Goals

The compilation of goals is described by the rule $\langle\theta_0\rangle\ \mathsf{gcomp}(\mathit{goal},\ \eta,\ \delta) \Rightarrow \langle\ \theta\rangle$. η is a mapping which goes from continuation identifiers (e.g., s for the success continuation, f for the failure continuation, and possibly more identifiers for other continuations, such as those needed for exceptions) to basic blocks identifiers. Therefore $\eta(\mathsf{s})$ and $\eta(\mathsf{f})$ denote the continuation addresses in case of success (resp. failure) of *goal*. The compilation state θ is obtained from θ_0 by *appending* the generated code for *goal* to the δ basic block, and optionally introducing more basic blocks connected by the continuations associated to them.

The rules for the compilation of control are presented in Figure 5.5. We will assume some predefined operations to request a new basic block identifier (bb_new) and a list of new identifiers (bb_newn), and to add a C sentence to a given basic block (emit). The conjunction $(a,\ b)$ is translated by rule (Conj) by reclaiming a new new basic block identifier δ_b for the subgoal b, generating code for a in the target δ, using as success continuation δ_b, and then generating code for b in δ_b. The construct $(a \rightarrow b\ ;\ c)$ is similarly compiled by the (IfThenElse) rule. The compilation of a takes place using as success and failure continuations the basic block identifiers where b and c are emitted, respectively. Then, the process continues by compiling both b and c using the original continuations. The goals true and fail are compiled by emitting a jump statement (**goto** _) that goes directly to the success and failure continuation (rules (True) and (Fail)).

As stated in Section 5.3.2, there is no compilation rule for disjunctions (a ; b). Nevertheless, program transformations can change them into *if-then-else* structures, following the constraints on the input language. E.g., ($X = 1$; $X = 2$) is accepted if X is ground on entry, since the code can be translated into the equivalent ($X = 1 \rightarrow \mathsf{true}$; $X = 2 \rightarrow \mathsf{true}$). It will not be accepted if X is unbound, since the *if-then-else* code and the initial disjunction are not equivalent.

Note that since continuations are taken using C **`goto _`** statements, there is a great deal of freedom in the physical ordering of the basic blocks in the program. The current implementation emits code in an order roughly corresponding to the source program, but it has internal data structures which make it easy to change this order. Note that different orderings can impact performance, by, for example, changing code locality, affecting how the processor speculative execution units perform, and changing which **`goto _`** statements which jump to an immediate label can be simplified by the compiler.

Compilation of Goal Calls

External predicates explicitly defined in C and user predicates compiled to C code have both the same external interface. Thus we use the same call compilation rules for them.

Predicates that may fail are mapped to functions with *boolean* return types (indicating success / failure), and those which cannot fail are mapped to procedures (with no return result – as explained later in Section 5.3.4). Figure 5.6 shows the rules to compile calls to external or user predicates. Function $\mathsf{arg\text{-}pass}(f/n)$ returns the list $[r_1, ..., r_n]$ of argument passing modes for predicate f/n. Depending on $\mathsf{argmode}(f/n, i)$ (see Section 5.3.3) , r_i is $\mathsf{val\_rval}$ for in or $\mathsf{ref\_rval}$ for out. Using the translation in Table 5.1, the C expression for each variable is given as $r_i[\![a_i]\!]$. Taking the C identifier assigned to predicate ($\mathsf{c\_id}(f/n)$), we have all the pieces to perform the call. If the predicate is semi-deterministic (i.e., it either fails or gives a single solution), the (CALL-S) rule emits code that checks the return value and jumps to the success or failure continuation. If the predicate is deterministic, the (CALL-D) rule emits code that continues at the success continuation. To reuse those code generation patterns, rules (EMIT-S) and (EMIT-D) are defined.

$$\text{(CALL-S)}\quad \frac{\begin{array}{l}\mathsf{semidet}(f/n)\\ [r_1,\ ...,\ r_n] = \mathsf{argpass}(f/n)\\ \forall i{=}1..n\ \ c_i = r_i[\![a_i]\!]\\ c_f = \mathsf{c\_id}(f/n)\quad \langle\theta_0\rangle\ \mathsf{emit\_s}(c_f\texttt{(}c_1\texttt{, ..., }c_n\texttt{)},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(f(a_1,\ ...,\ a_n),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(EMIT-S)}\quad \frac{\langle\theta_0\rangle\ \mathsf{emit}(\texttt{if (}expr\texttt{) goto }\eta(\mathsf{s})\texttt{; else goto }\eta(\mathsf{f})\texttt{; },\ \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{emit\_s}(expr,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(CALL-D)}\quad \frac{\begin{array}{l}\mathsf{det}(f/n)\\ [r_1,\ ...,\ r_n] = \mathsf{argpass}(f/n)\\ \forall i{=}1..n\ \ c_i = r_i[\![a_i]\!]\\ c_f = \mathsf{c\_id}(f/n)\quad \langle\theta_0\rangle\ \mathsf{emit\_d}(c_f\texttt{(}c_1\texttt{, ..., }c_n\texttt{)},\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(f(a_1,\ ...,\ a_n),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(EMIT-D)}\quad \frac{\begin{array}{l}\langle\theta_0\rangle\ \mathsf{emit}(stat,\ \delta) \Rightarrow \langle\theta_1\rangle\\ \langle\theta_1\rangle\ \mathsf{emit}(\texttt{goto }\eta(\mathsf{s}),\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{emit\_d}(stat,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

Figure 5.6: Compilation of calls.

$$\text{(UNIFY-FG)}\quad \frac{\begin{array}{l}\mathsf{var}(a)\quad \mathsf{var}(b)\quad \beta \vdash \mathsf{fresh}(a)\quad \beta \vdash \mathsf{ground}(b)\\ c_a = \mathsf{val\_lval}[\![a]\!]\quad c_b = \mathsf{val\_rval}[\![b]\!]\\ \langle\theta_0\rangle\ \mathsf{emit\_d}(c_a\texttt{=}c_b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\lfloor\beta\rfloor a = b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(UNIFY-GG)}\quad \frac{\begin{array}{l}\mathsf{var}(a)\quad \mathsf{var}(b)\quad \beta \vdash \mathsf{ground}(a)\quad \beta \vdash \mathsf{ground}(b)\\ c_a = \mathsf{val\_rval}[\![a]\!]\quad c_b = \mathsf{val\_rval}[\![b]\!]\\ \langle\theta_0\rangle\ \mathsf{emit\_s}(c_a\texttt{==}c_b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\lfloor\beta\rfloor a = b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(INSTANCE-FC)}\quad \frac{\begin{array}{l}\mathsf{var}(a)\quad \mathsf{cons}(b)\quad \beta \vdash \mathsf{fresh}(a)\\ c_a = \mathsf{val\_lval}[\![a]\!]\quad c_b = \mathsf{encodecons}(b,\ \mathsf{encodingtype}(a))\\ \langle\theta_0\rangle\ \mathsf{emit\_d}(c_a\texttt{=}c_b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\lfloor\beta\rfloor a = b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

Figure 5.7: Unification compilation rules.

Compilation of Built-in Calls

When compiling goal calls, we distinguish the special case of built-ins, which are natively understood by the imProlog compiler and which treats them specially.

$$\text{(InitMut)}\quad \frac{\mathsf{var}(a)\quad \beta \vdash \mathsf{fresh}(a)\quad \mathsf{refmode}(a) = \mathbf{0m}\quad \langle\theta_0\rangle\ \mathsf{gcomp}(a \Leftarrow b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{gcomp}(a = \mathsf{initmut}(\tau,\ b),\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(AssignMut)}\quad \frac{\begin{array}{l}\mathsf{var}(a)\quad \beta \vdash \mathsf{ground}(a)\quad \beta \vdash a{:}\mathsf{mut}(\_)\\ \mathsf{var}(b)\quad \beta \vdash \mathsf{ground}(b)\\ c_a = \mathsf{mutval/val\_lval}[\![a]\!]\\ c_b = \mathsf{val\_rval}[\![b]\!]\\ \langle\theta_0\rangle\ \mathsf{emit\_d}(c_a{=}c_b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(a \Leftarrow b,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

$$\text{(ReadMut)}\quad \frac{\begin{array}{l}\mathsf{var}(a)\quad \beta \vdash \mathsf{ground}(a)\quad \beta \vdash a{:}\mathsf{mut}(\_)\quad \beta \vdash \mathsf{ground}(a@)\\ \mathsf{var}(b)\quad \beta \vdash \mathsf{fresh}(b)\\ c_a = \mathsf{mutval/val\_rval}[\![a]\!]\\ c_b = \mathsf{val\_lval}[\![b]\!]\\ \langle\theta_0\rangle\ \mathsf{emit\_d}(c_b{=}c_a,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(b = a@,\ \eta,\ \delta) \Rightarrow \langle\theta\rangle}$$

Figure 5.8: Compilation rules for mutable operations.

The unification $a = b$ is handled as shown in Figure 5.7. If a is a fresh variable and b is ground (resp. for the symmetrical case), the (Unify-FG) rule specifies a translation that generates an assignment statement that copies the value stored in b into a (using the translation for their *r-value* and *l-value*, respectively). When a and b are both ground, the built-in is translated into a comparison of their values (rule (Unify-GG)). When a is a variable and b is a constant, the built-in is translated into an assignment statement that copies the C value encoded from b, using the encoding type required by a, into a (rule (Instance-FC)). Note that although full unification may be assumed during program transformations and analysis, it must be ultimately reduced to one of the cases above. Limiting to the simpler cases is expected, in order to avoid bootstrapping problems when defining the full unification in imProlog as part of the emulator definition.

The compilation rules for operations on mutable variables are defined in Figure 5.8. The initialization of a mutable $a = \mathsf{initmut}(\tau,\ b)$ (rule (InitMut)) is compiled as a mutable assignment, but limited to the case where the reference mode of a is **0m** (that is, it has been inferred that it will be a local mutable variable). The built-in $a \Leftarrow b$ is translated into an assignment statement

(PRED-D)

$$\frac{\begin{array}{l}\mathsf{det}(name)\quad([a_1,\ ...,\ a_n],\ body) = \mathsf{lookup}(name)\\ \theta_0 = \mathsf{bb\_empty}\\ \langle\theta_3\rangle\ \mathsf{bb\_newn}(2) \Rightarrow [\delta,\ \delta_s]\ \langle\theta_4\rangle\\ \langle\theta_4\rangle\ \mathsf{gcomp}(body,\ [\mathsf{s}{\mapsto}\delta_s],\ \delta) \Rightarrow \langle\theta_5\rangle\\ \langle\theta_5\rangle\ \mathsf{emit}(\texttt{return},\ \delta_s) \Rightarrow \langle\theta\rangle\\ c_f = \mathsf{c\_id}(name)\\ argdecls = \mathsf{argdecls}([a_1,\ ...,\ a_n])\quad vardecls = \mathsf{vardecls}(body)\quad code = \mathsf{bb\_code}(\delta,\ \theta)\\ \langle p_0\rangle\ \mathsf{emitdecl}(\texttt{void}\ c_f\texttt{(}argdecls\texttt{) \{}\ vardecls\texttt{;}\ code\ \texttt{\}}) \Rightarrow \langle p\rangle\end{array}}{\langle p_0\rangle\ \mathsf{pcomp}(name) \Rightarrow \langle p\rangle}$$

(PRED-S)

$$\frac{\begin{array}{l}\mathsf{semidet}(name)\quad([a_1,\ ...,\ a_n],\ body) = \mathsf{lookup}(name)\\ \theta_0 = \mathsf{bb\_empty}\\ \langle\theta_3\rangle\ \mathsf{bb\_newn}(3) \Rightarrow [\delta,\ \delta_s,\ \delta_f]\ \langle\theta_4\rangle\\ \langle\theta_4\rangle\ \mathsf{gcomp}(body,\ [\mathsf{s}{\mapsto}\delta_s,\ \mathsf{f}{\mapsto}\delta_f],\ \delta) \Rightarrow \langle\theta_5\rangle\\ \langle\theta_5\rangle\ \mathsf{emit}(\texttt{return TRUE},\ \delta_s) \Rightarrow \langle\theta_6\rangle\\ \langle\theta_6\rangle\ \mathsf{emit}(\texttt{return FALSE},\ \delta_f) \Rightarrow \langle\theta\rangle\\ c_f = \mathsf{c\_id}(name)\\ argdecls = \mathsf{argdecls}([a_1,\ ...,\ a_n])\quad vardecls = \mathsf{vardecls}(body)\quad code = \mathsf{bb\_code}(\delta,\ \theta)\\ \langle p_0\rangle\ \mathsf{emitdecl}(\texttt{bool}\ c_f\texttt{(}argdecls\texttt{) \{}\ vardecls\texttt{;}\ code\ \texttt{\}}) \Rightarrow \langle p\rangle\end{array}}{\langle p_0\rangle\ \mathsf{pcomp}(name) \Rightarrow \langle p\rangle}$$

Figure 5.9: Predicate compilation rules.

(rule (ASSIGNMUT)), that copies the value of b as the mutable value of a. The (READMUT) rule defines the translation of $b = a@$, an assignment statement that copies the value stored in the mutable value of a into b, which must be a fresh variable. Note that the case $x = a@$ where x is not fresh can be reduced to $(t = a@,\ x = t)$, with t a new variable, for which compilation rules exist.

Compilation of Predicates

The rules in the previous sections defined how goals are compiled. In this sections we will use those rules to compile predicates as C functions. Figure 5.9 provides rules that distinguish between deterministic and semi-deterministic predicates. For a predicate with $name = f/n$, the $\mathsf{lookup}(name)$ function returns its arguments and body. The information from analysis of encoding types and reference

modes (Section 5.3.3) is used by argdecls and vardecls to obtain the list of argument and variable declarations for the program. On the other hand, bb_code is a predefined operation that flattens the basic blocks in its second argument θ as a C block composed of labels and statements. Finally, the emitdecl operation is responsible for inserting the function declarations in the compilation output p. Those definitions are used in the (PRED-D) and (PRED-S) rules. The former compiles deterministic predicates by binding a single success to a **return** statement, and emits a C function returning no value. The latter compiles semi-deterministic predicates by binding the continuations to code that returns a *true* or *false* value depending on the success and failure status. Note that this code matches exactly the scheme needed in Section 5.3.4 to perform calls to imProlog predicates compiled as C functions.

A Compilation Example

In order to clarify how the previous rules generate code, we include here a short code snippet (Figure 5.10) with several types of variables accessed both from the scope of their first appearance, and from outside that frame. We show also how this code is compiled into two C functions. Note that redundant jumps and labels have been simplified. It is composed of an encoding type definition flag/1, two predicates that are compiled to C functions (p/1 semi-deterministic, swmflag/1 deterministic), and two predicates with annotations to unfold the code during preprocessing (mflag/2 and swflag/2). Note that by unfolding the mflag/2 predicate, an illegal code (passing a reference to a local mutable) becomes legal. Indeed, this kind of predicate unfolding has proved to be a good, manageable replacement for the macros which usually appear in emulators written in lower-level languages and which are often a source of mistakes.

Related Compilation Schemes

Another compilation scheme which produces similar code is described in [HS02]. There are, however, significant differences, of which we will mention just a few. One of them is the source language and the constraints imposed on it. In our case we aim at writing a WAM emulator in imProlog from which C code is gen-

Source:

```
:- regtype flag/1 + low(int32).
flag := off | on.
:- pred p(+I) :: flag.
p(I) :-
    mflag(I, A),
    A = B,
    swmflag(B),
    A@ = on.
:- pred mflag/2 + unfold.
mflag(I, X) :-
    X = initmut(flag, I).
:- pred swmflag(+I) :: mut(flag).
swmflag(X) :-
    swflag(X@, X2),
    X <= X2.
:- pred swflag/2 + unfold.
swflag(on, off).
swflag(off, on).
```

Output:

```
bool p(int32 i) {
    int32 a;
    int32 *b;
    int32 t;
    b = &a;
    swmflag(b);
    t = a;
    if (t == 1) goto l1; else goto l2;
l1: return TRUE;
l2: return FALSE;
}

void smwflag(int32 *x) {
    int32 t;
    int32 x2;
    t = *x;
    if (t == 1) goto l1; else goto l2;
l1: x2 = 0;
    goto l3;
l2: x2 = 1;
l3: *x = x2;
    return;
}
```

Figure 5.10: imProlog compilation example

erated with the constraint that it has to be identical (or, at least, very close) to a hand-written and hand-optimized one, including the implementation of the internal data structures. This has forced us to pay special attention to the compilation and placement of data, mutable variables, and ignore at this moment non-deterministic control. Also, in this work we use an intermediate representation based on basic blocks, which would make it easier to interface with internal back-ends for compilers other than GCC, such as LLVM [LA04] (which enables JIT compilation from the same representation).

5.4 Extensions for Emulator Generation in imProlog

The dialect and compilation process that has been described so far is general enough to express the instructions in a typical WAM emulator, given some basic built-ins about operations on data types, memory stacks, and O.S. interface. However, combining those pieces of code together to build an *efficient* emulator requires a compact encoding of the bytecode language, and a bytecode fetching and dispatching loop that usually needs a tight control on low-level data and operations that we have not included in the imProlog language. In Chapter 4 we showed that it is possible to automate the generation of the emulator from generic instruction definitions, and annotations stating how the bytecode is encoded and decoded. Moreover, this process was found to be highly mechanizable, while making instruction code easier to manage and other optimizations (such as instruction merging) easier to perform. In this section we show how this approach is integrated in the compilation process, by including the emulator generation as part of the compilation process.

5.4.1 Defining WAM Instructions in imProlog

The definition of every WAM instruction in imProlog looks just like a regular predicate, and the types, modes, etc. of each of their arguments have to be declared using (Ciao) assertions. As an example, Figure 5.11 shows imProlog code corresponding to the definition of an instruction which tries to unify a term and a constant. The **pred** declaration states that the first argument is a mutable variable and that the second is a tagged word containing a constant. It includes a sample implementation of the WAM dereference operation, which follows a reference chain and stops when the value pointed to is the same as the pointing term, or when the chain cannot be followed any more. Note the use of the native type tagged/2 and the operations tagof/2 and tagval/2 which access the tag and the associated value of a tagged word, respectively. Also note that the tagval/2 of a tagged word with ref results in a mutable variable, as can be recognized in the code. Other native operations include trail_cond/1, trail_push/1, and operations to manage the emulator stacks. Note the special predicates next_ins and fail_ins. They execute the next instruction or the failure instruction, respectively. The purpose of the next instruction is to continue the emulation of the next bytecode

```
:- pred u_cons(+, +) :: mut(tagged) * constagged.
u_cons(A, Cons) :-
    deref(A@, T_d),
    ( tagof(T_d, ref) → bind_cons(T_d, Cons), next_ins
    ; T_d = Cons → next_ins
    ; fail_ins
    ).

:- pred deref/2.
deref(T, T_d) :-
    ( tagof(T, ref) →
        T_1 = ~tagval(T)@,
        (T = T_1 → T_d = T_1 ; deref(T_1, T_d))
    ; T_d = T
    ).

:- pred bind/2.
bind_cons(Var, Cons) :-
    (trail_cond(Var) → trail_push(Var) ; true),
    ~tagval(Var) ⇐ Cons.
```

Figure 5.11: Unification with a constant and auxiliary definitions.

instruction (which can be considered as a recursive call to the emulator itself, but which will be defined as a built-in). The failure instruction must take care of unwinding the stacks at the WAM level and selecting the next bytecode instruction to execute (to implement the failure in the emulator). As a usual instruction, it can be defined by calling built-ins or other imProlog code, and it should finally include a call to next_ins to continue the emulation. Since this instruction is often invoked from other instructions, a special treatment is given to share its code, which will be described later.

The compilation process is able to unfold (if so desired) the definition of the predicates called by u_cons/2 and to propagate information from them inside

the instruction, in order to optimize the resulting piece of the emulator. After the set of transformations that instruction definitions are subject to, and other optimizations on the output (such as transformation of some recursions into loops) the generated C code is of high quality (see, for example, Figure 5.14, for the code corresponding to a specialization of this instruction).

Our approach has been to define a reduced number of instructions (50 is a ballpark figure) and let the merging and specialization process (see Section 5.5) generate all instructions needed to have a competitive emulator. Note that efficient emulators tend to have a large number of instructions (hundreds, or even thousands, in the case of Quintus Prolog) and many of them are variations (obtained through specialization, merging, etc., normally done manually) on "common blocks." These common blocks are the simple instructions we aim at representing explicitly in imProlog.

In the experiments we performed (Section 5.5.3) the emulator with a largest number of instructions had 199 different opcodes (not counting those which result from padding some other instruction with zeroes to ensure a correct alignment in memory). A simple instruction set is easier to maintain and its consistency is easier to ensure. Complex instructions are generated automatically in a (by construction) sound way from this initial "seed".

5.4.2 An Emulator Specification in imProlog

Although imProlog could be powerful enough to describe the emulation loop, as mentioned before we leverage on previous work in which $\mathcal{L}_c$ emulators were automatically built from definitions of instructions written in $\mathcal{L}_a$ and their corresponding code written in $\mathcal{L}_c$(Chapter 4). Bytecode representation, compiler back-end, and an emulator (including the emulator loop) able to understand $\mathcal{L}_b$ code can be automatically generated from those components. In our current setting, definitions for $\mathcal{L}_a$ instructions are written in $\mathcal{L}_s^r$ (recall Figure 5.2) and these definitions can be automatically translated into $\mathcal{L}_c$ by the imProlog compiler. We are thus spared of making this compiler more complex than needed. More details on this process will be given in the following section.

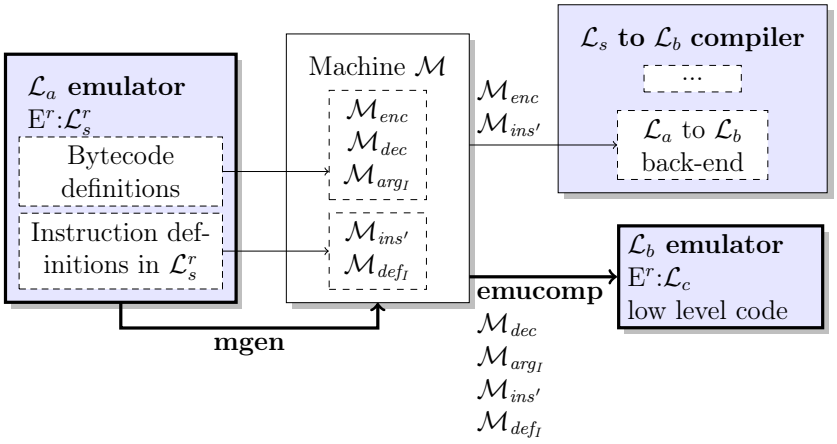

Figure 5.12: From imProlog definitions to $\mathcal{L}_b$ emulator in $\mathcal{L}_c$.

5.4.3 Assembling the Emulator

We will describe now the process that takes an imProlog representation of an abstract machine and obtains a full-fledged implementation of this machine. The overall process is sketched in Figure 5.12, and can be divided into two stages, which we have termed *mgen* and *emucomp*. The emulator definition, $\mathcal{E}$, is a set of predicates and assertions written in imProlog, and *mgen* is basically a normalization process where the source $\mathcal{E}$ is processed to obtain a machine definition $\mathcal{M}$. This definition contains components describing the instruction semantics written in imProlog and a set of *hints* about the bytecode representation (e.g., numbers for the bytecode instructions). $\mathcal{M}$ is then processed by an emulator compiler *emucomp* which generates a bytecode emulator for the language $\mathcal{L}_b$, written in the language $\mathcal{L}_c$. The machinery to encode $\mathcal{L}_a$ programs into the bytecode representation $\mathcal{L}_b$ is also given by definitions in $\mathcal{M}$.

We use the terminology described in Chapter 4 to denote the components of $\mathcal{M}$ as follows:

$$\mathcal{M} = (\mathcal{M}_{enc}, \mathcal{M}_{dec}, \mathcal{M}_{arg_I}, \mathcal{M}_{def_I}, \mathcal{M}_{ins'})$$

Firstly, the relation between $\mathcal{L}_a$ and $\mathcal{L}_b$ is given by means of several components:[12]

[12]The complete description includes all elements for a WAM: `X` and `Y` registers, atoms, numbers, functors, etc.

$\underline{\mathcal{M}_{enc}}$ declares how the bytecode encodes $\mathcal{L}_a$ instructions and data: e.g., `X(0)` is encoded as the number `0` for an instruction which needs access to some `X` register.

$\underline{\mathcal{M}_{dec}}$ declares how the bytecode should be decoded to give back the initial instruction format in $\mathcal{L}_a$: e.g., for an instruction which uses as argument an `X` register, a `0` means `X(0)`.

The rest of the components of $\mathcal{M}$ capture the meaning of the (rather low level) constituents of $\mathcal{L}_a$, providing a description of each instruction. Those components do not make bytecode representation issues explicit, as they have already been specified in $\mathcal{M}_{enc}$ and $\mathcal{M}_{dec}$. In this chapter, and unlike the formalization presented in Chapter 4, definitions for $\mathcal{L}_a$ instructions are given in $\mathcal{L}_s^r$ instead of $\mathcal{L}_c$. The reason for this change is that in Chapter 4 the final implementation language ($\mathcal{L}_c$, in which emulators were generated) was also the language in which each basic instruction was assumed to be written. However, in our case, instructions are obviously written in $\mathcal{L}_s^r$ (i.e., imProlog, which is more amenable to automatic program transformations) and it makes more sense to use it directly in the definition of $\mathcal{M}$. Using $\mathcal{L}_s^r$ requires, however, extending and/or modifying the remaining parts of $\mathcal{M}$ with respect to the original definition as follows:

$\underline{\mathcal{M}_{arg_I}}$ which assigns a pair (T, mem) to every expression in $\mathcal{L}_a$, where T is the type of the expression and mem is the translation of the expression into $\mathcal{L}_c$. For example, the type of `X(0)` is `mut(tagged)` and its memory location is `&(x[0])`, assuming `X` registers end up in an array.[13]

$\underline{\mathcal{M}_{def_I}}$ which contains the definition of each instruction in $\mathcal{L}_s^r$.

$\underline{\mathcal{M}_{ins'}}$ which describes the instruction set with opcode numbers and the format of each instruction, i.e., the type in $\mathcal{L}_a$ for each instruction argument: e.g., `X` registers, `Y` registers, integers, atoms, functors.

The rest of the components and $\mathcal{M}_{ins'}$ are used by the emulator compiler to generate an $\mathcal{L}_b$ emulator written in $\mathcal{L}_c$. A summarized definition of the emulator

[13]This definition has been expanded with respect to its original $\mathcal{M}_{arg}$ definition in order to include the imProlog type in addition to the memory location.

compiler and how it uses the different pieces in $\mathcal{M}$ can be found in Figure 5.13. The (EMU) rule defines a function that contains the emulator loop. It is similar to the (PRED-D) rule already presented, but takes parts of the source code from $\mathcal{M}$. It generates a list of basic block identifiers for each instruction, and a basic block identifier for the emulator loop entry. The (SWR) rule is used to insert a *switch* statement that implements the opcode fetching, and jumps to the code of each instruction. The (INS) rule is used to generate the code for each instruction. To implement the built-ins next_ins and fail_ins, two special continuations ni and fi are stored in the continuation mapping. The continuation to the failure instruction is bound to the δ_f basic block identifier (assuming that the op_f opcode is that of the failure instruction). The (FAILINS) rule includes a special case in gcomp that implements this call. The continuation to the next instruction is a pair of the basic block that begins the emulator switch, and a piece of C code that moves the bytecode pointer to the next instruction (that is particular to each instruction, and is returned by insdef alongside with its code). The (NEXTINS) rule emits code that executes that code and jumps to opcode fetching.

The abstract machine component $\mathcal{M}_{ins'}$ is used to obtain the *name* and data *format* of the instruction identified by a given *opcode*. From the format and $\mathcal{M}_{arg_I}$ definition, a type, an encoding type, and a custom *r-value* for each instruction argument are filled. In this way, the compilation process can transparently work with variables whose value is defined from the operand information in a bytecode stream (e.g., an integer, a machine register, etc.).

Relation with Other Compilation Schemes. The scheme of the generated emulator code is somewhat similar to what the Janus compilation scheme [GBD92] produces for general programs. In Janus, addresses for continuations are either known statically (e.g., for calls, and therefore a direct, static jump to a label can be performed) or are popped from the stack when returning. Since labels cannot be directly assigned to variables in standard C, an implementation workaround is made by assigning a number to each possible return address (and it is this number which is pushed onto / popped from the stack) and using a *switch* to relate these numbers with the code executing them. In our case we have a similar *switch*, but it relates each opcode with its corresponding instruction code, and it is executed every time a new instruction is dispatched.

$$\frac{\begin{array}{l}\mathcal{M}_{ops} = [op_1, \ldots, op_n] \\ \theta_0 = \mathsf{bb\_empty} \\ \langle\theta_0\rangle\ \mathsf{bb\_newn}(n+1) \Rightarrow [\delta, \delta_1, \ldots, \delta_n]\ \langle\theta_1\rangle \\ \langle\theta_1\rangle\ \mathsf{emit\_switch}(\texttt{get\_opcode()}, \mathcal{M}_{ops}, [\delta_1, \ldots, \delta_n], \delta) \Rightarrow \langle\theta_2\rangle \\ \langle\theta_2\rangle\ \forall i{=}1..n\ \mathsf{inscomp}(op_i, \delta, \delta_i, [\mathsf{fi}{\mapsto}\delta_f])\ \langle\theta\rangle \\ code = \mathsf{bb\_code}(\delta, \theta) \\ \langle p_0\rangle\ \mathsf{emitdecl}(\texttt{void emu() \{ } code \texttt{ \}}) \Rightarrow \langle p\rangle\end{array}}{\langle p_0\rangle\ \mathsf{emucomp} \Rightarrow \langle p\rangle}\ (\textsc{Emu})$$

$$\frac{\langle\theta_0\rangle\ \mathsf{emit}(\texttt{switch (}x\texttt{) \{case } v_1\texttt{: goto } \delta_1\texttt{; ...; case } v_n\texttt{: goto } \delta_n\texttt{; \}}, \delta) \Rightarrow \langle\theta\rangle}{\langle\theta_0\rangle\ \mathsf{emit\_switch}(x, [v_1, \ldots, v_n], [\delta_1, \ldots, \delta_n], \delta) \Rightarrow \langle\theta\rangle}\ (\textsc{Swr})$$

$$\frac{\begin{array}{l}(body, nextp) = \mathsf{insdef}(opcode) \\ \langle\theta_2\rangle\ \mathsf{gcomp}(body, \eta[\mathsf{ni}{\mapsto}(\delta_0, nextp)], \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{inscomp}(opcode, \delta_0, \delta, \eta) \Rightarrow \langle\theta\rangle}\ (\textsc{Ins})$$

$$\frac{\begin{array}{l}(\delta_s, nextp) = \eta(\mathsf{ni}) \\ \langle\theta_0\rangle\ \mathsf{emit}(nextp, \delta) \Rightarrow \langle\theta_1\rangle \\ \langle\theta_1\rangle\ \mathsf{emit}(\texttt{goto } \delta_s, \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\mathsf{next\_ins}, \eta, \delta) \Rightarrow \langle\theta\rangle}\ (\textsc{NextIns}) \qquad \frac{\begin{array}{l}\delta_s = \eta(\mathsf{fi}) \\ \langle\theta_0\rangle\ \mathsf{emit}(\texttt{goto } \delta_s, \delta) \Rightarrow \langle\theta\rangle\end{array}}{\langle\theta_0\rangle\ \mathsf{gcomp}(\mathsf{fail\_ins}, \eta, \delta) \Rightarrow \langle\theta\rangle}\ (\textsc{FailIns})$$

Figure 5.13: Emulator compiler.

We want to note that we deliberately stay within standard C in this presentation: taking advantage of C extensions, such as storing labels in variables, which are provided by `gcc` and used, for example, in [HCS95, CD95], is out of the scope of this paper. These optimizations are not difficult to add as code generation options, and therefore they should not be part of a basic scheme. Besides, that would make it difficult to use compilers other than `gcc`.

Example 5.4.1: As an example, from the instruction in Figure 5.11, which unifies a term living in some variable with a constant, we can derive a specialized version in which the term is assumed to live in an `X` register. The declaration:

```
:- ins_alias(ux_cons, u_cons(xreg_mutable, constagged)).
```

assigns the (symbolic) name ux_cons to the new instruction, and specifies that the first argument lives in an X register. The declaration:

```
:- ins_entry(ux_cons).
```

indicates that the emulator has an entry for that instruction.[14] Figure 5.14 shows the code generated for the instruction (right) and a fragment of the emulator generated by the emulator compiler in Figure 5.13.

5.5 Automatic Generation of Abstract Machine Variations

Using the techniques described in the previous section we now address how abstract machine variations can be generated automatically. Substantial work has been devoted to abstract machine generation strategies such as, e.g., [DN00, NCS01, CZ07], which explore different design variations with the objective of putting together highly optimized emulators. However, as shown previously, by making the semantics of the abstract machine instructions explicit in a language like imProlog, which is easily amenable to automatic processing, such variations can be formulated in a straightforward way mostly as automatic transformations. Adding new transformation rules and testing them together with the existing ones becomes then a relatively easy task.

We will briefly describe some of these transformations, which will be experimentally evaluated in Section 5.5.3. Each transformation is identified by a two-letter code. We make a distinction between transformations which change the instruction set (by creating new instructions) and those which only affect the way code is generated.

[14] We optionally allow a pre-assignment of an opcode number to each instruction entry. Different assignments of instruction numbers to opcodes can impact the final performance, as they dictate how the code is laid out in the emulator switch which affects, for example, the behavior of the cache.

```
loop:
  switch(Op(short,P,0)) {
  ...
  case 97: goto ux_cons;
  ...
  }
  ...
ux_cons:
  tagged t;
  t = X(Op(short,P,2));
  deref(&t);
  if (tagged_tag(t) != REF)
    goto ux_cons__0;
  bind_cons(t, Op(tagged,P,4));
  goto ux_cons__1;
ux_cons__0:
  if (t != Op(tagged,P,4))
    goto fail_ins;
ux_cons__1:
  P = Skip(P,8);
  goto loop;
  ...
```

```
void deref(tagged_t *a0) {
  tagged_t t0;
deref:
  if (tagged_tag(*a0) == REF)
    goto deref__0;
    else goto deref__1;
deref__0:
  t0 = *(tagged_val(*a0));
  if ((*a0) != t0)
    goto deref__2;
    else goto deref__1;
deref__2:
  *a0 = t0;
  goto deref;
deref__1:
  return;
}
```

Figure 5.14: Code generated for a simple instruction.

5.5.1 Instruction Set Transformations

Let us define an instruction set transformation as a pair (*ptrans*,*etrans*), so that *ptrans* transforms programs from two symbolic bytecode languages $\mathcal{L}_a$ and (a possibly different) $\mathcal{L}_a$' [15] and *etrans* transforms the abstract machine definition within $\mathcal{L}_s^r$. Figure 5.15 depicts the relation between emulator generation, program compilation, program execution, and instruction set transformations. The full emulator generation includes *etrans* as preprocessing before *mgen* is performed. The resulting emulator is able to interpret transformed programs after *ptrans* is applied (before bytecode encoding), that is, the new compiler is obtained by including *ptrans* as a new compilation phase.

[15] Those languages can be different, for example, if the transformation adds or removes some instructions

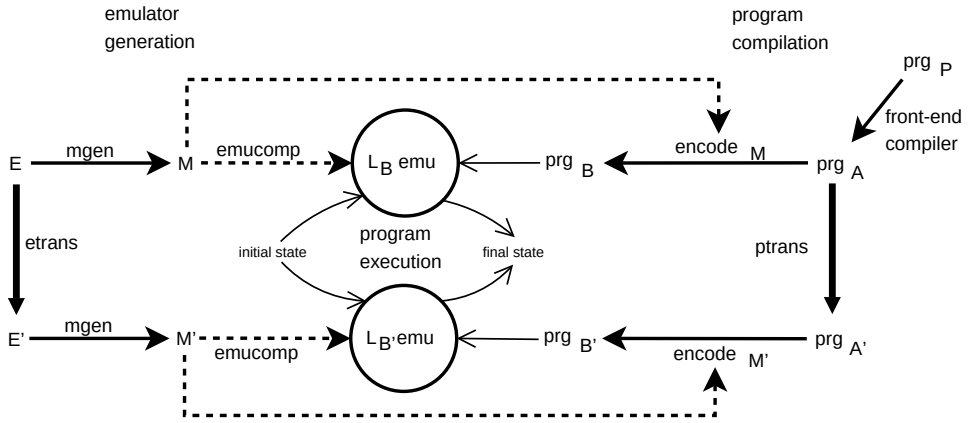

Figure 5.15: Application of an instruction set transformation (*ptrans*, *etrans*).

Note that both *ptrans* and *etrans* are working at the symbolic bytecode level. It is easier to work with a symbolic $\mathcal{L}_a$ program than with the stream of bytes that represents $\mathcal{L}_b$ code, and it is easier to transform instructions written in $\mathcal{L}_s^r$ and specified at the $\mathcal{L}_a$ level, than those already written in $\mathcal{L}_c$ code (where references to $\mathcal{L}_b$ code and implementation details obscure the actual semantics). Reasoning about the correctness of the global transformation that affects the $\mathcal{L}_a$ program and the instruction code is also easier in the $\mathcal{L}_s^r$ specification of the emulator instructions than in a low-level $\mathcal{L}_c$ emulator (assuming the correctness of the emulator generation process).

In the following sections we will review the instruction set transformations currently available. Although more transformations can of course be applied, the current set is designed with the aim of generating, from simple imProlog definitions, an emulator which is as efficient as a hand-crafted, carefully tuned one.

Instruction Merging [im]

Instruction Merging generates larger instructions from sequences of smaller ones, and is aimed at saving fetch cycles at the expense of a larger instruction set and, therefore, an increased `switch` size. This technique has been used extensively in high-performance systems (e.g., Quintus Prolog, SICStus, Yap, etc.). The performance of different combinations has been studied empirically [NCS01], but in

that work new instructions were generated by hand, although deciding which instructions had to be created was done by means of profiling. In our framework only a single declaration is needed to emit code for a new, merged instruction. Merging is done automatically through code unfolding and based on the definitions of the component instructions. This makes it possible, in principle, to define a set of (experimentally) optimal merging rules. However, finding exactly this set of rules is actually not straightforward.

Merging rules are specified separately from the instruction code itself, and these rules state how basic instructions have to be combined. To start with, we will need to show how instructions are defined based on their abstract versions. For example, definition:

$$\mathsf{move}(A,\ B) \coloneq B \Leftarrow A@\ .$$

moves data between two locations, i.e., the contents of the a mutable into the b mutable. In order to specify precisely the source and destination of the data, it is necessary to specify the instruction *format*, with a declaration such as:

```
:- ins_alias(movexy, move(xreg_mutable, yreg_mutable)).
```

which defines a *virtual* instruction named movexy, that corresponds to the instantiation of the code for move/2 for the case in which the first argument corresponds to an X register and the second one corresponds to a Y register. Both registers are seen from imProlog as mutable variables of type *mut(tagged)*. Then, and based on this more concrete instruction, the declaration

```
:- ins_entry(movexy + movexy).
```

forces the compiler to actually use during compilation an instruction composed of two virtual ones and to emit bytecode containing it (thanks to the *ptrans* transformation in Figure 5.15, which processes the program instruction sequence to replace occurrences of the collapsed pattern by the new instruction). Emulator code will be generated implementing an instruction which merges two movexy instructions (thanks to the *etrans* transformation). The corresponding code is equivalent to:

```
:– ins_entry(movexy_movexy).
:– pred movexy_movexy(xreg_mutable, yreg_mutable,
                      xreg_mutable, yreg_mutable).
movexy_movexy(A, B, C, D) :– B ⇐ A@, D ⇐ C@ .
```

This can later be subject to other transformations and used to generate emulator code as any other imProlog instruction.

Single-Instruction Encoding of Sequences of the Same Instruction [ie]

In some cases a series of similar instructions (e.g., unify_with_void) with different arguments can be collapsed into a single instruction with a series of operands which correspond to the arguments of each of the initial instructions. For example, a bytecode sequence such as:

```
unify_with_void(x(1)),          unify_with_void(x(2)),
unify_with_void(x(5))
```

can be compiled into:

```
unify_with_void_n([x(1), x(2), x(5)])
```

which would perform exactly as in the initial instruction series, but taking less space and needing fewer fetch cycles. Such an instruction can be created, emitted, and the corresponding emulator code generated automatically based on the definition of unify_with_void.

In order to express this *composite* instruction within imProlog using a single predicate, unify_with_void_n needs to receive a fixed number of arguments. A different predicate for each of the possible lengths of the array would have to be generated otherwise. A single argument actually suffices; hence the square brackets, which are meant to denote an array.

The imProlog code which corresponds to the newly generated instruction is, conceptually, as follows:

```
unify_with_void_n(Array) :-
    array_len(Array, L),
    unify_with_void_n_2(0, L, Array).
unify_with_void_n_2(I, L, Array) :-
    ( I = L → true
    ; elem(I, Array, E),
      unify_with_void(E),
      I1 is I + 1,
      unify_with_void_n_2(I1, L, Array)
    ).
```

It should be clear here why a fixed number of arguments is needed: a series of unify_with_void_n/1, unify_with_void_n/2, etc. would have to be generated otherwise. Note that the loop code ultimately calls unify_with_void/1, the $\mathcal{L}_s^r$ reflection of the initial instruction.

In this particular case the compiler to $\mathcal{L}_c$ performs some optimizations not captured in the previous code. For example, instead of traversing explicitly the array with an index, this array is expanded and inlined in the bytecode and the program counter is used to retrieve the indexes of the registers by incrementing it after every access. As the length of the array is known when the bytecode is generated, it is actually explicitly encoded in the final bytecode. Therefore, all of the newly introduced operations (array_len/2, elem/3, etc.) need constant time and are compiled efficiently.

Instructions for Special Built-Ins [ib]

As mentioned before, calling external library code or internal predicates (classically termed "built-ins") requires following a protocol, to pass the arguments, to check for failure, etc. Although the protocol can be the same as for normal predicates (e.g., passing arguments as X registers), some built-ins require a special (more efficient) protocol (e.g. passing arguments as $\mathcal{L}_c$ arguments, avoiding movements in X registers). Calling those special built-ins is, by default, taken care of by a generic family of instructions, one per arity. This is represented as the instructions:

```
:- ins_alias(bltin1d, bltin1(bltindet(tagged), xreg)).
bltin1(BltName, A) :- BltName(A@).
:- ins_alias(bltin2d, bltin2(bltindet(tagged, tagged), xreg, xreg)).
bltin2(BltName, A, B) :- BltName(A@, B@).
```

where each bltinI/i + 1 acts as a bridge to call the external code expecting i parameters. The *BltName* argument represents a predicate abstraction that will contain a reference to the actual code of the built-in during the execution. The type of the *BltName* argument reflects the accepted calling pattern of the predicate abstraction. When compiling those instructions to $\mathcal{L}_c$ code, that predicate abstraction is efficiently translated as an unboxed pointer to an $\mathcal{L}_c$ procedure.[16] With the definition shown above, the imProlog compiler can generate, for different arities, an instruction which calls the built-in passed as first argument.

However, by specifying at compile time a predefined set of built-ins or predicates written in $\mathcal{L}_c$ (that is, a static value for *BltName* instead of a dynamic value), the corresponding instructions can be statically specialized and an instruction set which performs direct calls to the corresponding built-ins can be generated. This saves an operand, generating slightly smaller code, and replaces an indirection by a direct call, which saves memory accesses and helps the processor pipeline, producing faster code.

5.5.2 Transformations of Instruction Code

Some transformations do not create new instructions; they perform instead a number of optimizations on already existing instructions by manipulating the code or by applying selectively alternative translation schemes.

[16] In the bytecode, the argument that corresponds to the predicate abstraction is stored as a number that uniquely identifies the built-in. When the bytecode is actually loaded, this number is used to look up the actual address of the built-in in a table maintained at runtime. This is needed since, in general, there is no way to know which address will be assigned to the entry point of a given built-in in different program executions.

Unfolding Rules [ur]

Simple predicates can be unfolded before compilation. In the case of instruction merging, unfolding is used to merge two (or more) instructions into a single piece of code, in order to avoid fetch cycles (Section 5.5.1). However, uncontrolled unfolding is not always an advantage, because an increased emulator size can affect negatively the cache behavior. Therefore the ***ur*** option turns on or off a predefined set of rules to control which instruction mergings are actually performed. Unfolding rules follow the scheme:

:– **ins_entry**(Ins_1 + Ins_2 + ... + Ins_n, $WhatToUnfold$).

where Ins_1 to Ins_n are the basic instructions to be merged, and $WhatToUnfold$ is a rule specifying exactly which instruction(s) has to be unfolded when ***ur*** is activated. As a concrete example, the unfolding rule:

:– **ins_entry**(alloc + movexy + movexy, 1).

means that in the instruction to be generated by combining one alloc and two movexy, the code for alloc is inlined (the value of the last argument 1 refers to the *first* instruction in the sequence), and the (shared) code for movexy + movexy is invoked afterwards. A companion instruction merging rule for movexy + movexy exists:

:– **ins_entry**(movexy + movexy, all).

which states that the code for both movexy has to be unfolded in a combined instruction. The instruction alloc + movexy + movexy would generate code for alloc plus a call to movexy + movexy. The compiler eventually replaces this call by an explicit jump to the location of movexy + movexy in the emulator. The program counter is updated accordingly to access the arguments correctly.

Alternative Tag Switching Schemes [ts]

Tags are used to determine dynamically the type of basic data (atoms, structures, numbers, variables, etc.) contained in a (tagged) memory word. Many

instructions and built-ins (like unification) take different actions depending on the type (or tag) of the input arguments. This is called *tag switching*, and it is a heavily-used operation which is therefore worth optimizing as much as possible. The tag is identified (and the corresponding action taken) using tag switching such as:

$$(tagtest_1 \rightarrow tagcode_1 \ ; \ ... \ ; \ tagtest_n \rightarrow tagcode_n)$$

where every $tagtest_i$ has the form $\mathsf{tagof}(v, tag_i)$ (i.e., code that performs a different action depending on the tag value of a tagged v). The ***ts*** option chooses between either a switch control structure (when enabled) or a set of predefined test patterns based on tag encodings and assertions on the possible tags (when disabled).

Both possibilities are studied in more detail in [MCH08]. Since the numbers that encode the tags are usually small, it is easy for a modern C compiler (e.g., `gcc`) to generate an indirection table and jump to the right code using it (that is, it does not require a linear search). It is difficult, however, to make the C compiler aware that checks to ensure that the tag number will actually be one of the cases in the *switch* are, by construction, unnecessary (i.e., there is no need for a *default* case). This information could be propagated to the compiler with a type system which not all low-level languages have. The alternative compilation scheme (rule (TIF)) makes explicit use of tag-checking primitives, where the sequence of $ctest_i$ and the code of each branch depends on the particular case.

The latter approach is somewhat longer (and more complex as the number of allowed tags grows) than the former. However, in some cases there are several advantages to the latter, besides the already mentioned avoidance of boundary checks:

- Tags with higher runtime probability can be checked before, in order to select the right branch as soon as possible.
- Since the evaluation order is completely defined, tests can be specialized to determine as fast as possible which alternative holds. For example, if by initial assumption v can only be either a heap variable, a stack variable, or a structure (having a different tag for each case), then the tests can check if it is a heap variable or a stack variable and assume that it is a structure in the last branch.

Deciding on the best option has to be based on experimentation, the results of which we summarize in Section 5.5.3 and in tables 5.4 and 5.5.

Connected Continuations [cc]

Some actions can be repeated unnecessarily because they appear at the end of an operation and at the beginning of the next one. Often they have no effect the second time they are called (because they are, e.g., tests which do not change the tested data, or data movements). In the case of tests, for example, they are bound to fail or succeed depending on what happened in the previous invocation.

As an example, in the fragment deref(T), (tagof(T, ref) $\rightarrow$ A ; B) the test tagof(R, ref) is performed just before exiting deref/1 (see Figure 5.11). Code generation for instructions which include similar patterns is able to insert a jump either to A or B from the code generated for deref/1. This option enables or disables this optimization for a series of preselected cases, by means of code annotations similar to the ones already shown.

Read/Write Mode Specialization [rw]

WAM-based implementations use a flag to test whether heap structures are being read (matched against) or written (created). According to the value of this flag, which is set by code executed immediately before, several instructions adapt their behavior with an internal, local *if-then-else*.

A common optimization is to partially evaluate the *switch* statement which implements the fetch-and-execute cycle inside the emulator loop. Two different switches can be generated, with the same structure, but with heap-related instructions specialized to perform either reads or writes [Car91]. Enabling or disabling the ***rw*** optimization makes it possible to generate instruction sets (and emulators) where this transformation has been turned on or off.

This is conceptually performed by generating different versions of the instructions code depending on the value of a mutable variable mode, which can only take the values read or write. Deciding whether to generate different code versions or to generate *if-then-else*s to be checked at run-time is done based on a series of heuristics which try to forecast the complexity and size of the resulting code.

5.5.3 Experimental Evaluation

We present in this section experimental data regarding the performance achieved on a set of benchmarks by a collection of emulators, all of which were automatically generated by selecting different combinations of the options presented in previous sections. In particular, by using all **compatible** possibilities for the transformation and generation options given in Section 5.5 we generated 96 different emulators (instead of $2^7 = 128$, as not all options are independent; for example, **ie** needs **im** to be performed). This bears a close relationship with [DN00], but here we are not changing the internal data structure representation (and of course our instructions are all initially coded in imProlog). It is also related to the experiment reported in [NCS01], but the tests we perform are more extensive and cover more variations on the kind of changes that the abstract machine is subject to. Also, [NCS01] starts off by being selective about the instructions to merge, which may seem a good idea but, given the very intricate dependencies among different optimizations, can also result in a loss of optimization opportunities. In any case, this is certainly a point we want to address in the future by using instruction-level profiling.

Although most of the benchmarks we used are relatively well known, a brief description follows:

boyer Simplified Boyer-Moore theorem prover kernel.

crypt Cryptoarithmetic puzzle involving multiplication.

deriv Symbolic derivation of polynomials.

factorial Compute the factorial of a number.

fib Simply recursive computation of the n^{th} Fibonacci number.

knights Chess knight tour, visiting only once every board cell.

nreverse Naive reversal of a list using append.

poly Raises symbolically the expression `1 + x + y + z` to the n^{th} power.

primes Sieve of Eratosthenes.

qsort Implementation of QuickSort.

queens11 N-Queens with $N = 11$.

query Makes a natural language query to a knowledge database with information about country names, population, and area.

tak Computation of the Takeuchi function.

Our starting point was a "bare" instruction set comprising the common basic blocks of a relatively efficient abstract machine (the "optimized" abstract machine of Ciao 1.13, in the 'optim_comp' directory in the Ciao 1.13 repository).[17] The Ciao abstract machines have their remote roots in the emulator of SICStus Prolog 0.5/0.7 (1986-89), but have evolved over the years quite independently and been the object of many optimizations and code rewrites resulting in performance improvements and much added functionality.[18] The performance of this, our *baseline* engine matches that of modern Prolog implementations. Table 5.2 helps evaluating the speed of this baseline optimized Ciao emulator w.r.t. to the relatively unoptimized Ciao 1.13 emulator compiled by default in the Ciao distribution and other well-known Prolog systems: Yap 5.1.2, hProlog 2.7, SWI-Prolog 5.6.55.

Figures 5.16 (in page 137) to 5.37 (in page 153) summarize graphically the results of the experiments, as the data gathered —96 emulators $\times$ 13 benchmarks = 1248 performance figures— is too large to be comfortably presented in regular tables.

Each figure presents the speedup obtained by different emulators for a given benchmark (or all benchmarks in the case of the summary tables). Such speedups are relative to some "default" code generation options, which we have set to be those which were active in the Ciao emulator we started with (our baseline), and which therefore receive speedup 1.0. Every point in each graph corresponds to the relative speed of a different emulator obtained with a different combination of the options presented in Sections 5.5.1 and 5.5.2.

[17] Changes in the optimized version include tweaks to clause jumping, arithmetic operations and built-ins and some code clean-ups that reduce the size of the emulator loop.

[18] This includes modules, attributed variables, support for higher order, multiprocessing, parallelism, tabling, modern memory management, etc., etc.

Benchmark	Yap 5.1.2	hProlog 2.7	SWI 5.6.55	Ciao-std 1.13	Ciao-opt (baseline)
boyer	1392	1532	11169	2560	1604
crypt	3208	2108	36159	6308	3460
deriv	3924	3824	12610	6676	3860
exp	1308	1740	2599	1400	1624
factorial	4928	2368	16979	3404	2736
fft	1020	1652	14351	2236	1548
fib	2424	1180	8159	1416	1332
knights	2116	1968	11980	3432	2352
nreverse	1820	908	18950	3900	2216
poly	1328	1104	6850	1896	1160
primes	4060	2004	28050	3936	2520
qsort	1604	1528	8810	2600	1704
queens11	1408	1308	24669	3200	1676
query	632	676	6180	1448	968
tak	3068	1816	27500	5124	2964

Table 5.2: Speed comparison with other Prolog systems.

The options are related to the points in the graphs as follows: each option is assigned a bit in a binary number, where '1' means activating the option and '0' means deactivating it. Every value in the y axis of the figures corresponds to a combination of the three options in Section 5.5.1. Note that only 6 combinations out of $2^3 = 8$ possible ones are plotted due to dependencies among options. Options in Section 5.5.2, which correspond to transformations in the way code is generated and which need four bits, are encoded using 16 different dot shapes. Every combination of emulator generation options is thus assigned a different 7-bit number encoded as a dot shape and y coordinate. The x coordinate represents the speedup as presented before (i.e., relative to the hand-coded emulator currently in Ciao 1.13).

Different selections for the bits assigned to the y coordinate and to the dot shapes would of course yield different plot configurations. However, our selection seems intuitively appropriate, as it uses two different encodings for two different

Instruction Generation			Instruction Transformations			
Instruction Encoding **(ie)**	Special Builtins **(ib)**	Instruction Merging **(im)**	Tag Switching **(ts)**	Connected Conts. **(cc)**	Unfolding Rules **(ur)**	R/W Mode **(rw)**

Table 5.3: Meaning of the bits in the plots.

families of transformations, one which affects the bytecode language itself, and another one which changes the way these bytecode operands are interpreted. Table 5.3 relates the bits in these two groups, using the same order as in the plots.

Every benchmark was run several times on each emulator to make time measures stable. The hosts used were an x86 machine with a Pentium 4 processor running Linux and an iMac with a PowerPC 7450 running Mac OS X. Arithmetic and geometric[19] averages of all benchmarks were calculated and are shown in Figures 5.16, 5.17, 5.32, and 5.33. Their similarity seems to indicate that there are no "odd" behaviors off the average. Additionally, we are including detailed plots for every benchmark and all the engine generation variants, following the aforementioned codification, first for the x86 architecture (Figures 5.19 to 5.31) and for the PowerPC architecture (Figures 5.34 to 5.46), in the same order in both cases. Plots for specially relevant cases are shown first, followed by the rest of the figures sorted following an approximate (subjective) "more sparse" to "less sparse" order.

General Analysis

The best speedup among all tried options, averaged across the exercised benchmarks and with respect to the baseline Ciao 1.13 emulator, is 1.05× for the x86 processor (Table 5.4, top section, under the column *w.r.t. def.*) and 1.01× for the PowerPC (Table 5.5, top section, same column heading). While this is a modest average gain, some benchmarks achieve much better speedups. An alternative interpretation of this result is that by starting with a relatively simple instruction set (coded directly in imProlog) and applying automatically and systematically a set of transformation and code generation options which can be trusted to be correct, we have managed to match (and even exceed) the time performance of

[19]The geometric average is known to be less influenced by extremely good or bad cases.

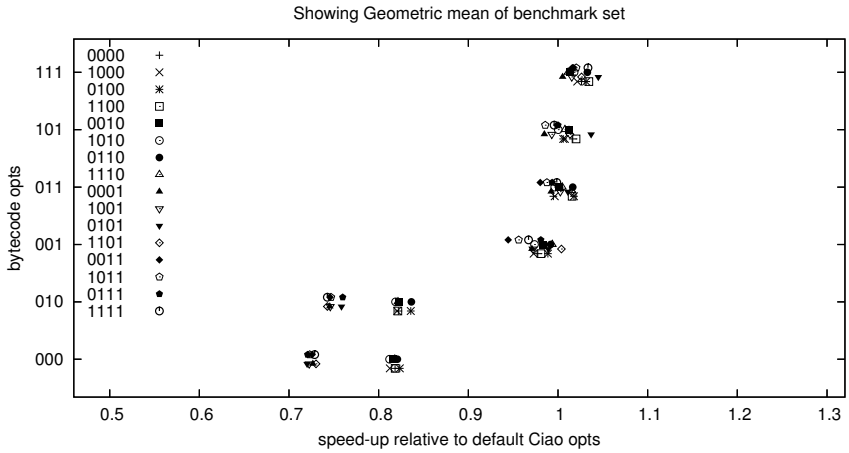

Figure 5.16: Geometric average of all benchmarks (with a dot per emulator) — Intel.

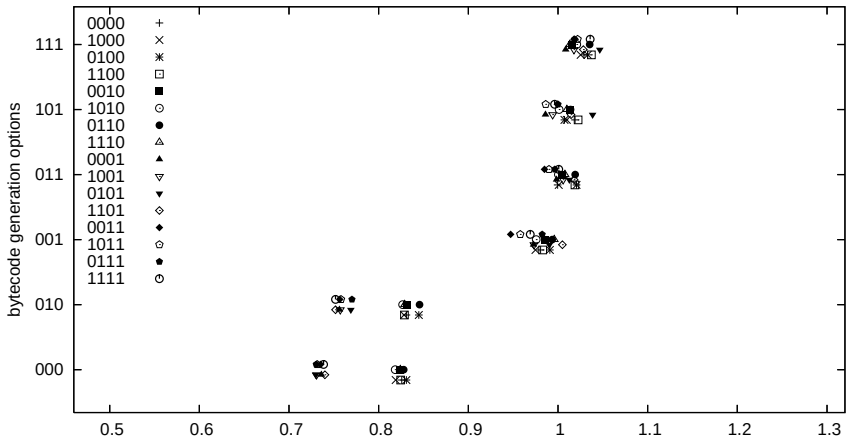

Figure 5.17: Arithmetic average of all benchmarks (with a dot per emulator) — Intel.

an emulator which was hand-coded by very proficient programmers, and in which decisions were thoroughly tested along several years. Memory performance was left untouched. Note (in the same tables) that the speedup obtained with respect to the basic instruction set (under the column labeled *w.r.t. base*) is significantly higher.

Figure 5.16 depicts the geometric average of the executions of all benchmarks in an Intel platform. It aims at giving an intuitive feedback of the overall performance of the option sets, and indeed a well defined clustering around eight centers is clear. Figure 5.17, which uses the arithmetic average, is very similar

(but not identical — it is very slightly biased towards higher speedups), and it shows eight well-defined clusters as well.

From these pictures we can infer that bytecode transformation options can be divided into two different sets: one which is barely affected by options of the generation of code for the emulator (corresponding to the upper four clusters), and another set (the bottom four clusters) in which changes to the generation of the emulator code does have an effect in the performance.

In general, the results obtained in the PowerPC show fewer variations than those obtained in an x86 processor. We attribute this behavior to differences between these two architectures, as they greatly affect the optimization opportunities and the way the C compiler can generate code. For example, the larger number of general-purpose registers available in a PowerPC seems to make the job of the C compiler less dependent on local variations of the code (as the transformations shown in Section 5.5.2 produce). Additionally, internal differences between both processors (e.g., how branch prediction is performed, whether there is register renaming, shadow registers, etc.) can also contribute to the differences we observed.

As a side note, while Figures 5.16 and 5.17 portray an average behavior, there were benchmarks whose performance depiction actually match this average behavior very faithfully —e.g., the simply recursive Factorial (Figure 5.19), which is often disregarded as an unrealistic benchmark but which, for this particular experiment, turns out to predict quite well the (geometric) average behavior of all benchmarks. Experiments in the PowerPC (Figures 5.32 and 5.34) generate similar results.

Figure 5.18, unlike the rest of the experimental results, presents the size of the WAM loop (using actual i86 object code size measured in bytes) for each bytecode and $\mathcal{L}_c$ code generation option. This size is independent from the benchmarks, and therefore only one plot is shown. It resembles notably the depictions of the speedup graphs. In fact, a detailed inspection of the distribution of low-level code generation options (where each of them corresponds to one of the 16 different dot shapes) inside each bytecode language option shows some correlation among larger and faster emulators. This is not surprising as some code generation schemes which tend to increase the size do so because they generate additional, specialized code.

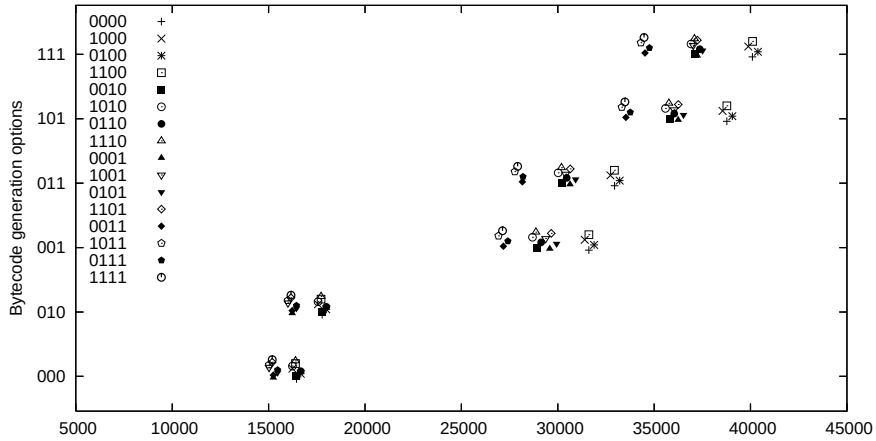

Figure 5.18: Size (in bytes) of WAM emulator with respect to the generation options (i86).

As in the case for speed, $\mathcal{L}_b$ generation options are the ones which influence most heavily the code size. This is understandable because some options (for example, the ***im*** switch for instruction merging, corresponding to the leftmost bit of the "bytecode generation options") increment notably the size of the emulator loop. On the other hand, code generation options have a less significant effect, as they do not necessarily affect all the instructions.

It is to be noted that the generation of specialized switches for the write and read modes of the WAM (the ***rw*** option) does not increase the size of the emulator. The reason is that when the ***rw*** flag is checked by all the instructions which need to do so (and many instructions need it), a large number of *if-then-else* constructions with their associated code are generated. In the case of the specialized switches, only an *if-then-else* is needed and the savings from generating less branching code make the emulator smaller.

A More Detailed Inspection of Selected Cases

Figures 5.20 (Queens 11) and 5.21 (Cryptoarithmetic puzzle) show two cases of interest. The former corresponds to results which, while departing from the average behavior, still resembles it in its structure, although there is a combination of options which achieves a speedup (around 1.25) that is significantly higher than average. Figure 5.21 shows a different landscape where variations on the code

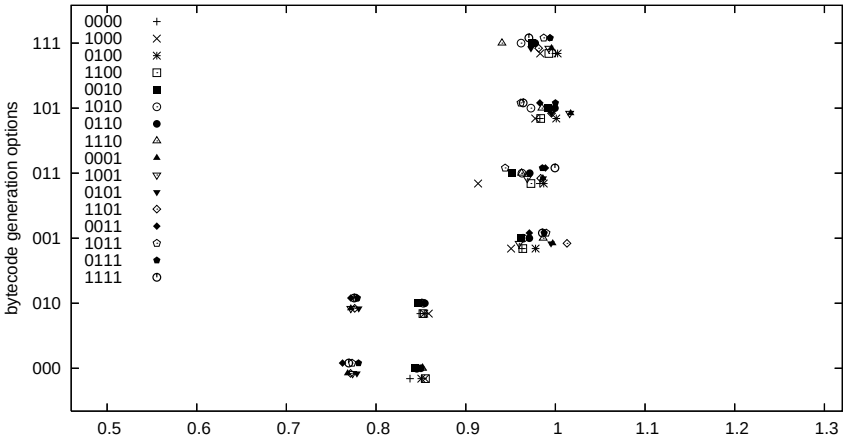

Figure 5.19: Factorial involving large numbers — Intel.

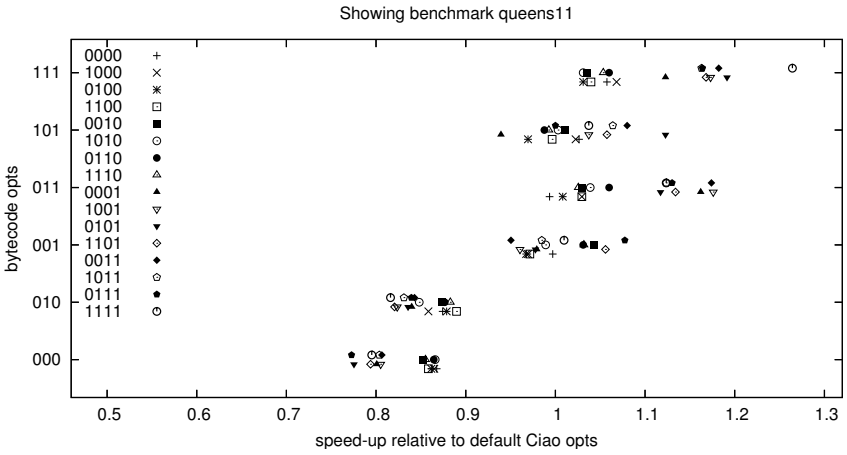

Figure 5.20: Queens (with 11 queens to place) — Intel.

generation scheme appear to be as relevant as those on the bytecode itself. Both benchmarks are, however, search-based programs which perform mainly arithmetic operations (with the addition of some data structure management in the case of the Queens program), and could in principle be grouped in the same class of programs. This points to the need to perform a finer grain analysis to determine, instruction by instruction, how every engine/bytecode generation option affects execution time, and also how these different options affect each other.

Studying which options are active inside each cluster sheds some light about their contribution to the overall speedup. For example, the upper four clusters of

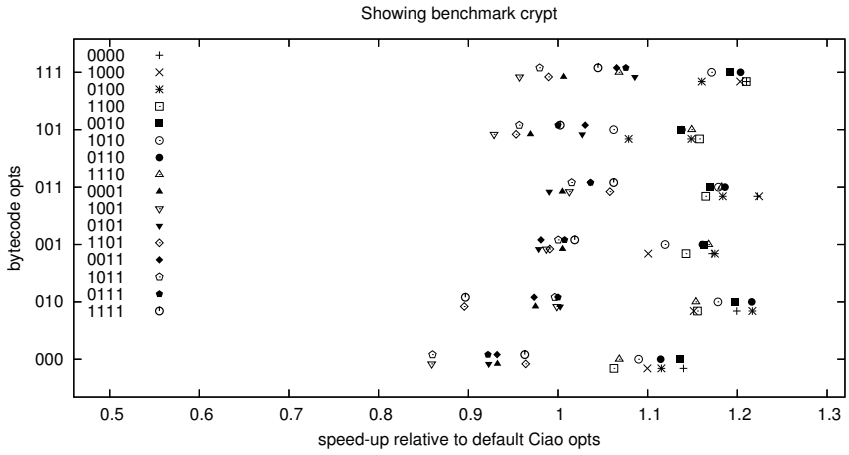

Figure 5.21: Cryptoarithmetic puzzle — Intel.

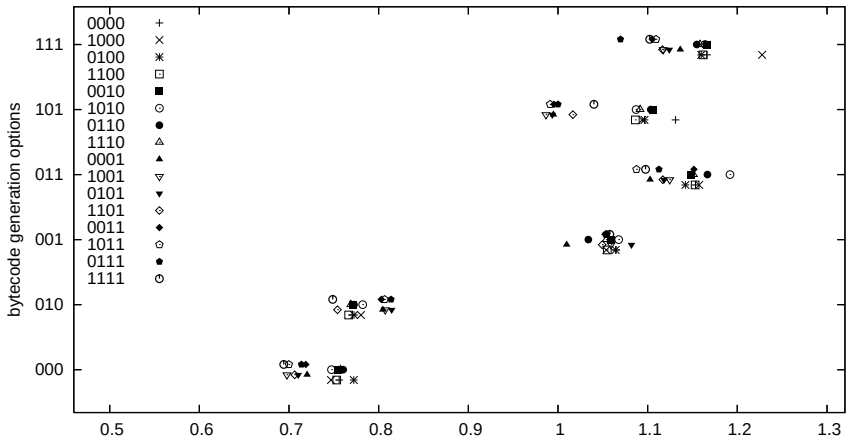

Figure 5.22: Computation of the Takeuchi function — Intel.

Figures 5.16 and 5.17 have in common the use of the ***ib*** option, which generates specialized instructions for built-ins. These clusters have consistently better (and, in some cases, considerably better) speedups than the clusters which do not have it activated. It is, therefore, a candidate to be part of the set of "best options". A similar pattern, although less acute, appears in the results of the PowerPC experiments (Figures 5.32 and 5.33).

The two leftmost clusters of the group of four at the bottom correspond to executions of emulators generated with the ***rw*** specialization activated, and the two clusters at their right do not have it activated. It can come as a surprise

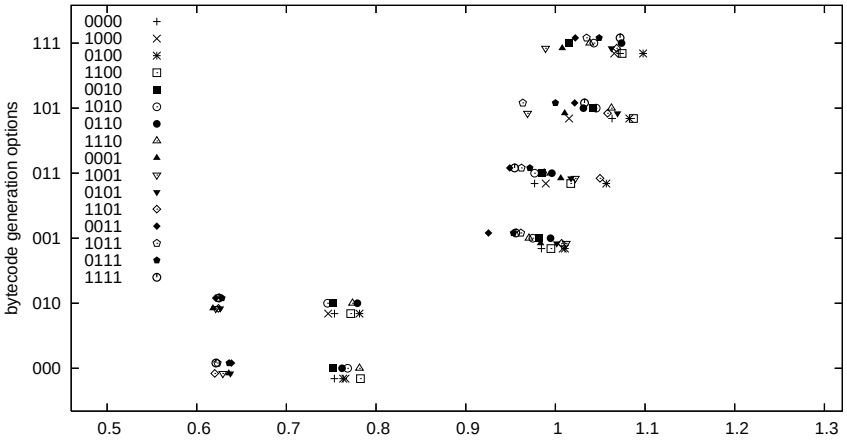

Figure 5.23: Symbolic derivation of polynomials — Intel.

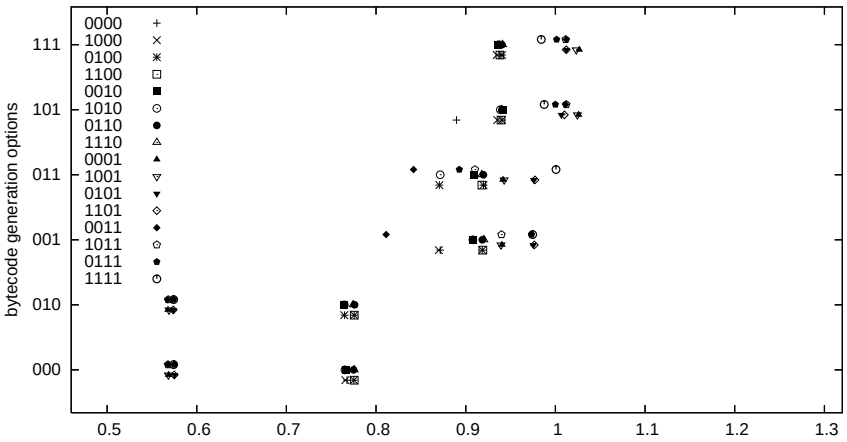

Figure 5.24: Naive reverse — Intel.

that using separate switches for read/write modes, instead of checking the mode in every instruction which needs to do so, does not seem to bring any advantage in the Intel processor. Indeed, a similar result was already observed in [DN00], and was attributed to modern architectures performing branch prediction and speculative work with redundant units. This is likely to be carried out with more accuracy at the smaller scale level of a short *if-then-else*.

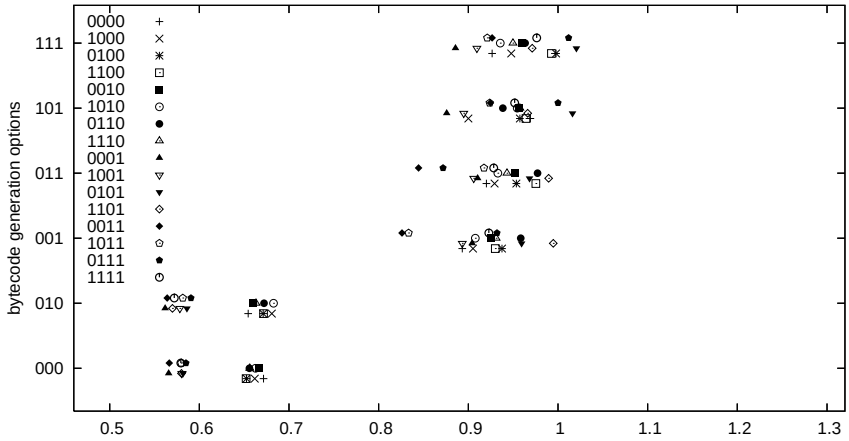

Figure 5.25: Symbolic exponentiation of a polynomial — Intel.

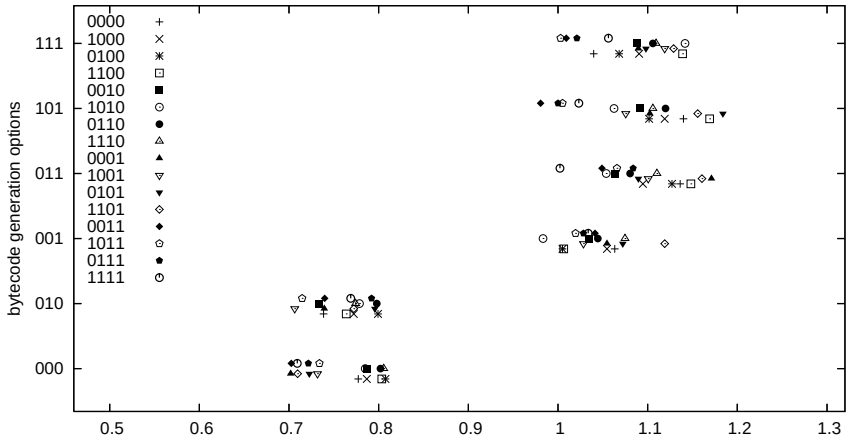

Figure 5.26: Version of Boyer-Moore theorem prover — Intel.

Best Generation Options and Overall Speedup

An important general question is *which options should be used for the "stock" emulator to be offered to general users.* Our experimental results show that options cannot be considered in isolation — i.e., the overall option set constructed by taking separately the best value for every option does not yield a *better set* (defined as the best options obtained by averaging speedups for every option set). As we have seen, there is some interdependence among options. A more realistic answer is that the average best set of options should come from selecting the

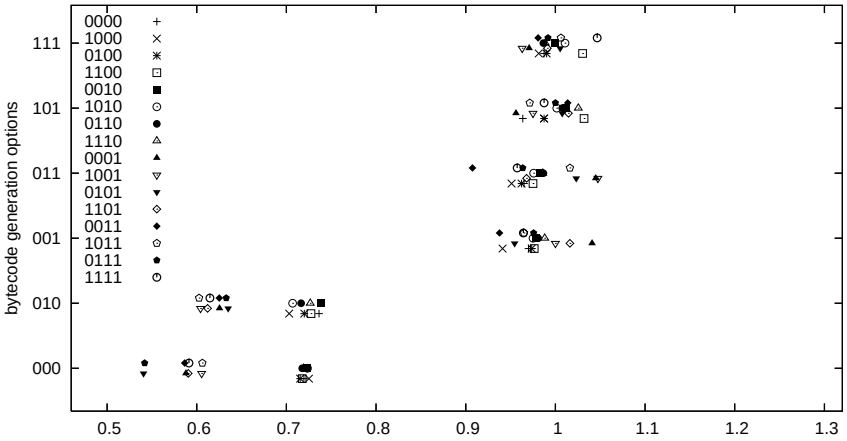

Figure 5.27: QuickSort — Intel.

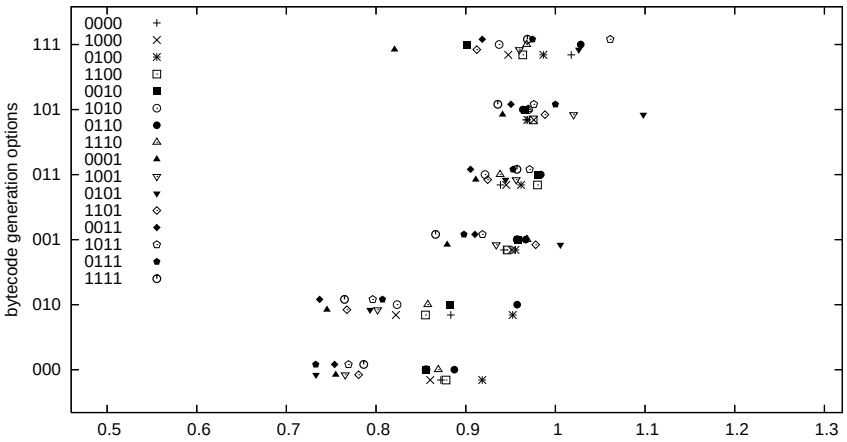

Figure 5.28: Calculate primes using the sieve of Eratosthenes — Intel.

rightmost point in the plot corresponding to average speedups. We must however bear in mind that averages always suffer the problem that a small set of good results may bias the average and, in this case, force the selection of an option set which performs worse for a larger set of benchmarks

In order to look more closely at the effects of individual options (without resorting to extensively listing them and the obtained performance), Tables 5.4 and 5.5 show which options produced the best and the worst results time-wise for each benchmark. We include the geometric average as an specific case and the Ciao-1.10 baseline options as reference.

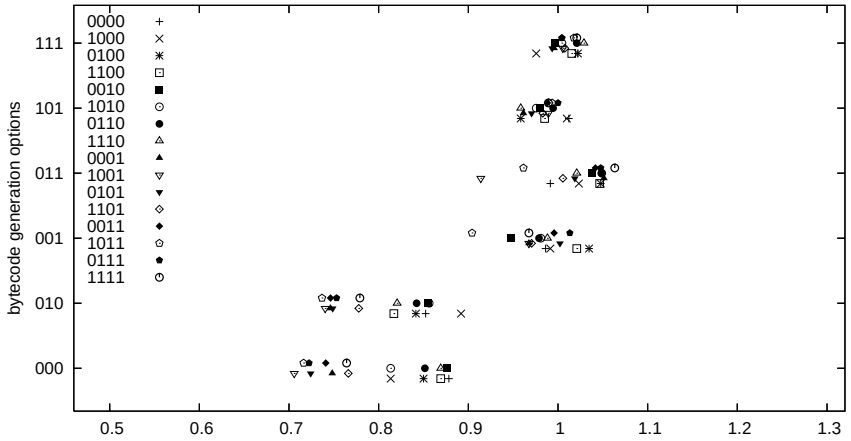

Figure 5.29: Natural language query to a geographical database — Intel.

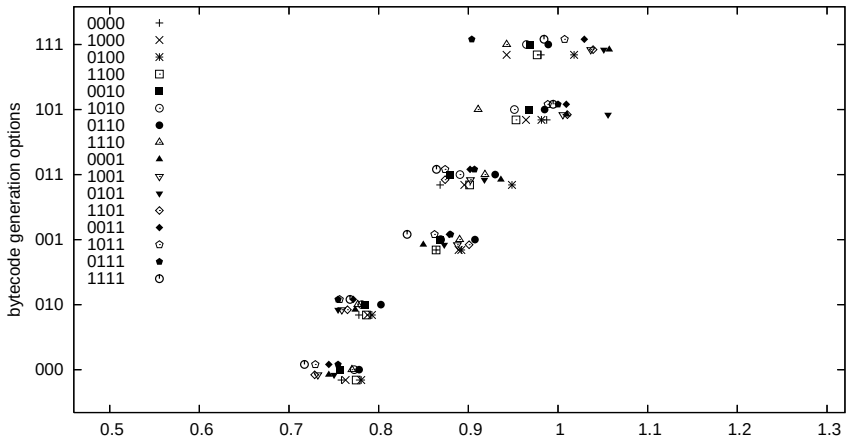

Figure 5.30: Chess knights tour — Intel.

It has to be noted that the best/worst set of options is not the negation of the worst/best options: there are plenty of cases where the same option was (de)activated both for the best and for the worst executions. The observed situation for the PowerPC architecture (Table 5.5) is more homogeneous: at least some better/worst behaviors really come from combinations which are complementary, and, in the cases where this is not so, the amount of non-complementary options goes typically from 1 to 3 — definitely less than in the x86 case.

Despite the complexity of the problem, some conclusions can be drawn: instruction merging (***im***) is a winner for the x86, probably followed by having a

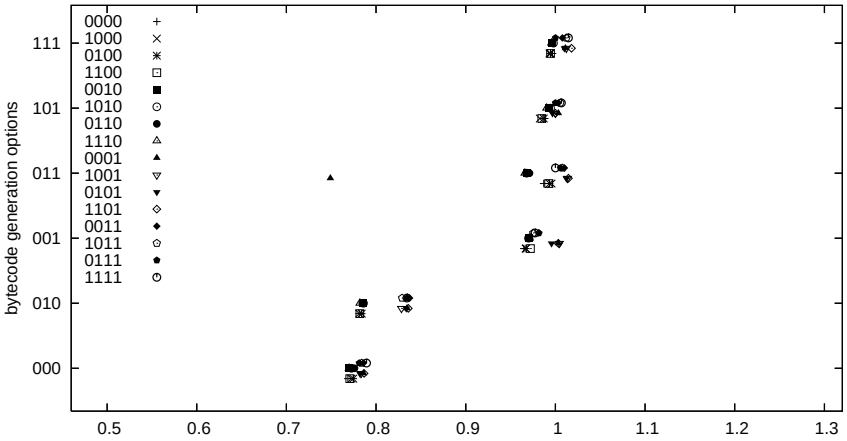

Figure 5.31: Simply recursive Fibonacci — Intel.

variable number of operands (***ie***), and then by specialized calls to built-ins (***ib***). The first and second options save fetch cycles, while the third one saves processing time in general. It is to be noted that some options appear both in the best and worst cases: this points to interdependencies among the different options.

The performance table for the PowerPC (Table 5.5) also reveals that instruction merging, having a variable number of operands, and generating specialized instructions for built-ins, are options which bring performance advantages. However, and unlike the x86 table, the *read/write mode* specialization is activated in all the lines of the "best performance" table, and off in the "worst performance". A similar case is that of the *tag switching schema*, in the sense that the selection seems clear in the PowerPC case.

The transformation rules we have applied in our case are of course not the only possible ones, and we look forward to enlarging this set of transformations by, for example, performing a more aggressive merging guided by profiling.[20] Similar work, with more emphasis on the production of languages for microprocessors is presented in [Hol93], where a set of benchmarks is used to guide the (constrained) synthesis of such a set of instructions.

We want to note that although exceeding the speed of a hand-crafted emulator is not the main concern in this work,[21] the performance obtained by the imple-

[20]Merging is right now limited in depth to avoid a combinatorial explosion in the number of instructions.

[21]In order to do that, a better approach would probably be to start off by finding performance

mentation of the emulator in imProlog allows us to conclude that the imProlog approach can match the performance of lower-level languages, while making it possible to apply non-trivial program transformation techniques.

In Chapter 7, additional experiments carried out in a very different scenario (that of embedded systems and digital signal processing which pertains to a realm traditionally considered disadvantageous for symbolic languages) showed also very good performance —only 20% slower than a comparable C program— and also very good speedups (up to 7-fold compared with an implementation running on a bytecode emulator). Analysis and compilation techniques similar to those applied in this paper were used, but put to work in a program using the full Prolog language.

5.6 Conclusions

We have designed a language and its compiler (imProlog, a variation of Prolog with some imperative features) and used it to describe the semantics of instructions of a bytecode interpreter (the Ciao engine). In our experience, the port replaced duplicated code structures, redundant hand-made specializations, and a large number of C macros (which notwithstanding helps in writing less code, but are not easily recognized and understood by automatic compilation tools), by more abstract predicates, and in the worst case, program annotations to guide the transformations. The result was an emulator as efficient as the original, but which avoided some typical software defects that hinder highly-tuned emulators.

The imProlog language, with the proposed constraints and extensions, is semantically closer enough to Prolog to share analysis, optimization and compilation techniques, but at the same time, it is designed to make translation into very efficient C code possible. The low-level code for each instruction and the definition of the bytecode is taken as input by a previously developed emulator generator to assemble full high-quality emulators. Since the process of generating instruction code and bytecode format is automatic, we were able to produce and test different versions thereof to which several combinations of code generation options were applied. This approach makes it possible to perform non-trivial

bottlenecks in the current emulator and redesigning / recoding it. We want to note, however, that we think that our approach can greatly help in making this redesign and recoding easier.

transformations on both the emulator and the instruction level (e.g., unfolding and partial evaluation of instruction definitions, instruction merging or specialization, etc.). The different transformations and code generation options, result in different grades of optimization / specialization and different bytecode languages from a single (higher-level) abstract machine definition.

We have also studied how these combinations perform with a series of benchmarks in order to find, e.g., what is the "best" average solution and how independent coding rules affect the overall speed. We have in this way as one case the regular emulator we started with (and which was decomposed to break complex instructions into basic blocks). However, we also found out that it is possible to outperform it by using some code patterns and optimizations not explored in the initial emulator, and, what is more important, starting from abstract machine definitions in imProlog.

Performance evaluation of non-trivial variations in the emulator code showed that some results are hard to predict and that there is no absolute winner for all architectures and programs. On the other hand, it is increasingly difficult to reflect all the variations in a single source using more traditional methods like `m4` or `cpp` macros. Automatic program manipulation at the emulator level represents a very attractive approach, and although difficult, the problem becomes more tractable when the abstraction level of the language to define the virtual machine is raised and enriched with some problem-specific declarations.

Benchmark	Best performance								
	ie	ib	im	ts	cc	ur	rw	Speed-up	
baseline	x		x		x	x	x	w.r.t. def.	w.r.t. base
all (geom.)	x	x	x		x		x	1.05	1.28
boyer	x		x		x		x	1.18	1.52
crypt		x	x	x				1.22	1.07
deriv	x	x	x		x			1.10	1.46
factorial	x		x				x	1.02	1.21
fib	x	x	x	x	x		x	1.02	1.32
knights	x	x	x				x	1.06	1.39
nreverse	x	x	x				x	1.03	1.34
poly	x	x	x		x		x	1.02	1.52
primes	x		x		x		x	1.10	1.26
qsort		x	x	x			x	1.05	1.46
queens11	x	x	x	x	x	x	x	1.26	1.46
query		x	x	x	x	x	x	1.06	1.21
tak	x	x	x	x				1.23	1.62

Benchmark	Worst performance								
	ie	ib	im	ts	cc	ur	rw	Speed-up	
baseline	x		x		x	x	x	w.r.t. def.	w.r.t. base
all (geom.)					x		x	0.70	0.88
boyer							x	0.70	0.90
crypt				x			x	0.86	0.75
deriv		x					x	0.62	0.82
factorial						x	x	0.76	0.99
fib		x	x				x	0.75	0.91
knights				x	x	x	x	0.72	0.97
nreverse		x				x	x	0.57	0.95
poly		x					x	0.56	0.74
primes					x	x	x	0.73	0.84
qsort					x		x	0.54	0.84
queens11					x	x	x	0.77	0.75
query				x			x	0.71	0.89
tak				x	x	x	x	0.69	0.92

Table 5.4: Options which gave best/worst performance (x86).

Benchmark	Best performance								
	ie	ib	im	ts	cc	ur	rw	Speed-up	
baseline	x		x		x	x	x	w.r.t. def.	w.r.t. base
all (geom.)	x	x	x		x	x	x	1.01	1.21
boyer	x	x	x				x	1.02	1.25
crypt		x	x		x		x	1.00	1.13
deriv	x		x		x	x	x	1.00	1.30
factorial	x		x		x	x	x	1.00	1.02
fib		x	x		x		x	1.03	1.17
knights	x	x	x				x	1.00	1.10
nreverse		x	x		x		x	1.02	1.20
poly	x	x	x		x	x	x	1.01	1.35
primes	x	x	x		x	x	x	1.02	1.33
qsort	x	x	x		x	x	x	1.01	1.17
queens11	x	x	x			x	x	1.06	1.33
query	x	x	x	x	x	x	x	1.01	1.20
tak	x	x	x		x		x	1.01	1.22

Benchmark	Worst performance								
	ie	ib	im	ts	cc	ur	rw	Speed-up	
baseline	x		x		x	x	x	w.r.t. def.	w.r.t. base
all (geom.)				x				0.82	0.99
boyer				x		x		0.81	0.99
crypt		x		x	x	x		0.87	0.98
deriv		x		x		x		0.76	0.99
factorial				x		x		0.85	0.97
fib								0.94	0.99
knights				x		x		0.82	1.00
nreverse		x			x			0.74	0.98
poly				x				0.74	0.98
primes				x		x		0.86	0.97
qsort				x				0.75	0.99
queens11				x				0.88	0.99
query				x				0.82	0.99
tak				x		x		0.78	0.99

Table 5.5: Options which gave best/worst performance (PowerPC).

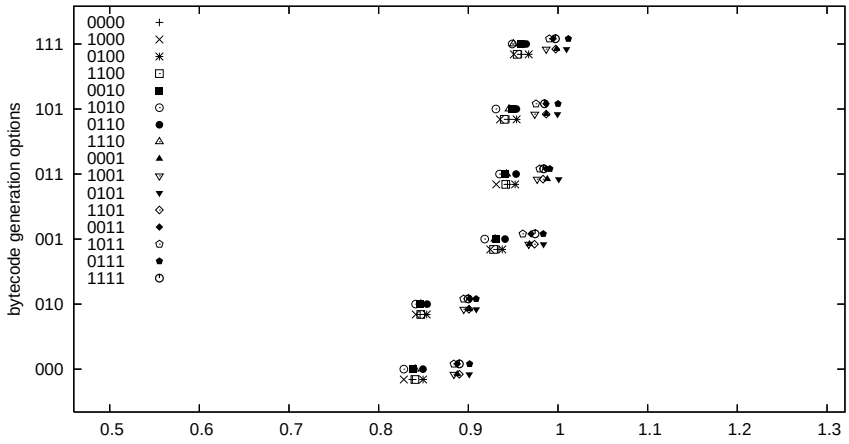

Figure 5.32: Geometric average of all benchmarks (with a dot per emulator) — PowerPC.

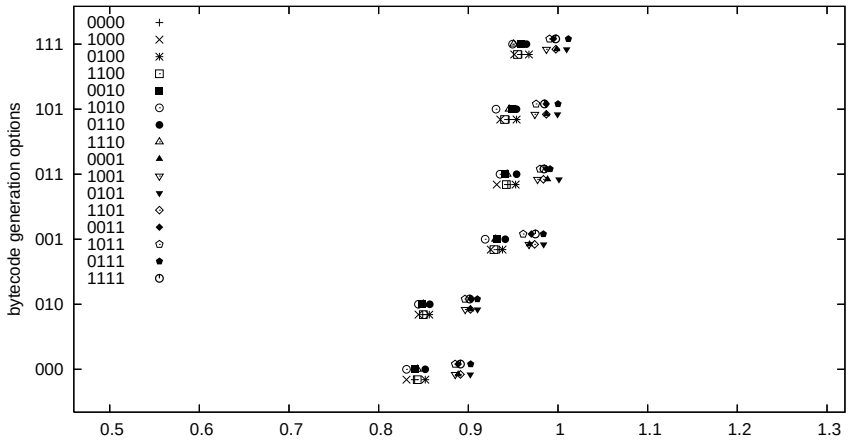

Figure 5.33: Arithmetic average of all benchmarks (with a dot per emulator) — PowerPC.

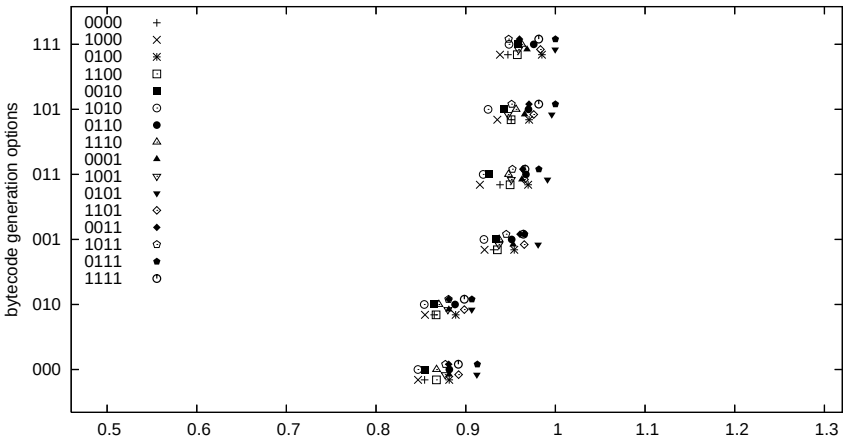

Figure 5.34: Factorial involving large numbers — PowerPC.

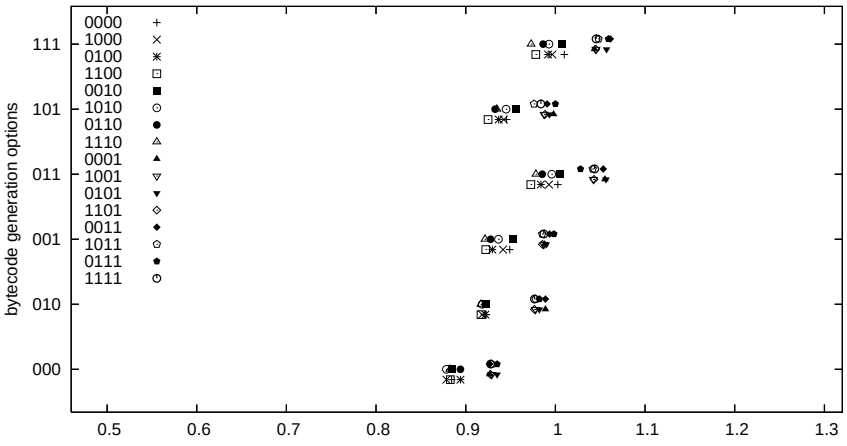

Figure 5.35: Queens (with 11 queens to place) — PowerPC.

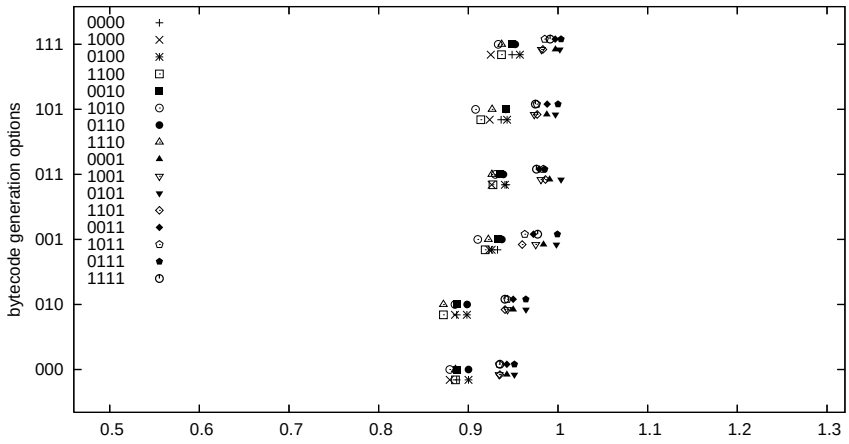

Figure 5.36: Cryptoarithmetic puzzle — PowerPC.

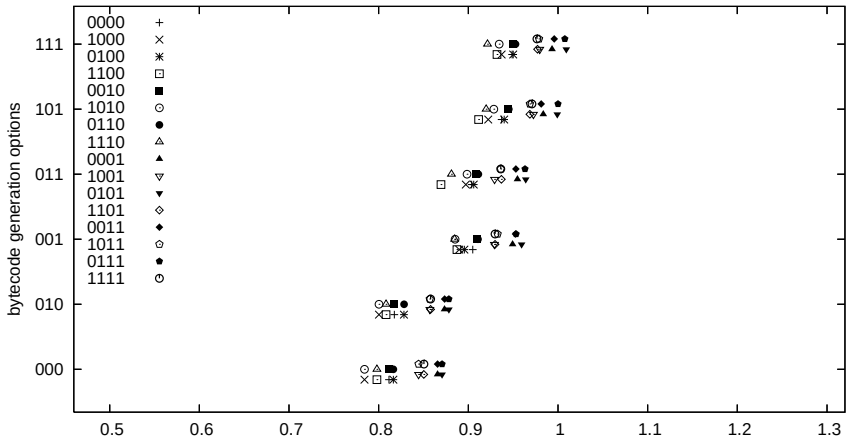

Figure 5.37: Computation of the Takeuchi function — PowerPC.

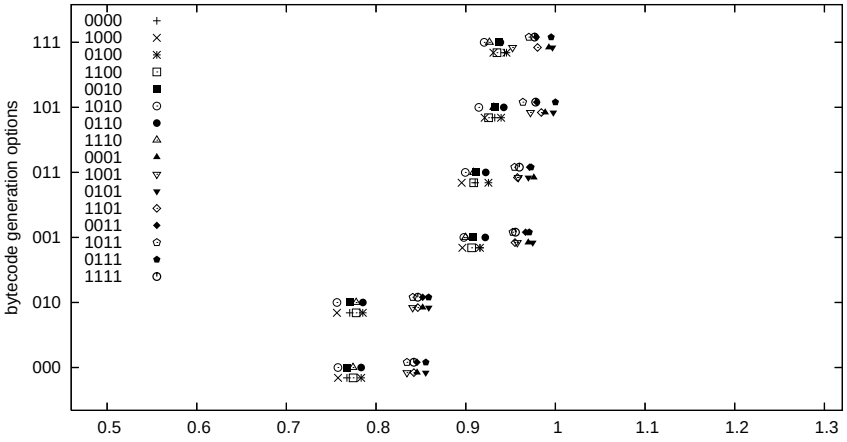

Figure 5.38: Symbolic derivation of polynomials — PowerPC.

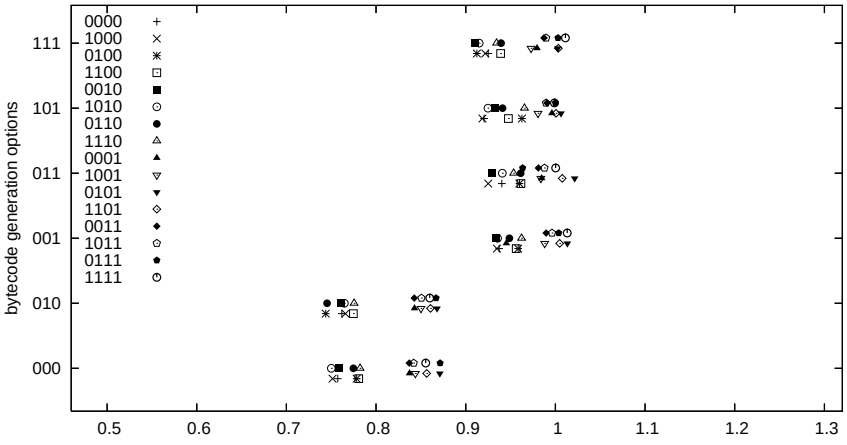

Figure 5.39: Naive reverse — PowerPC.

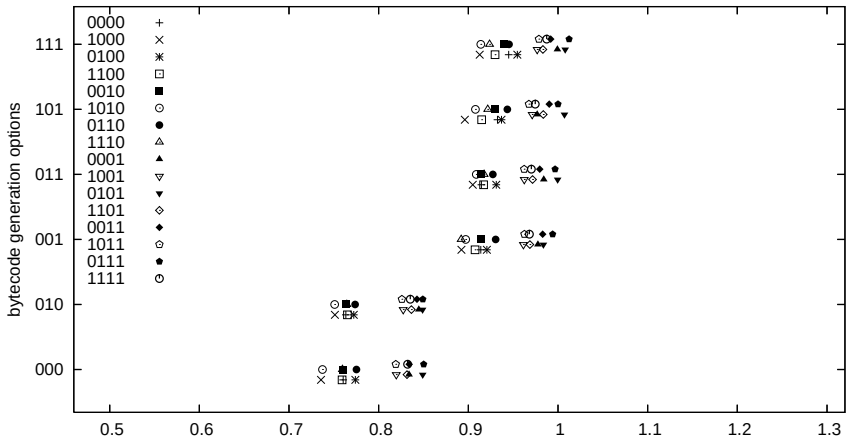

Figure 5.40: Symbolic exponentiation of a polynomial — PowerPC.

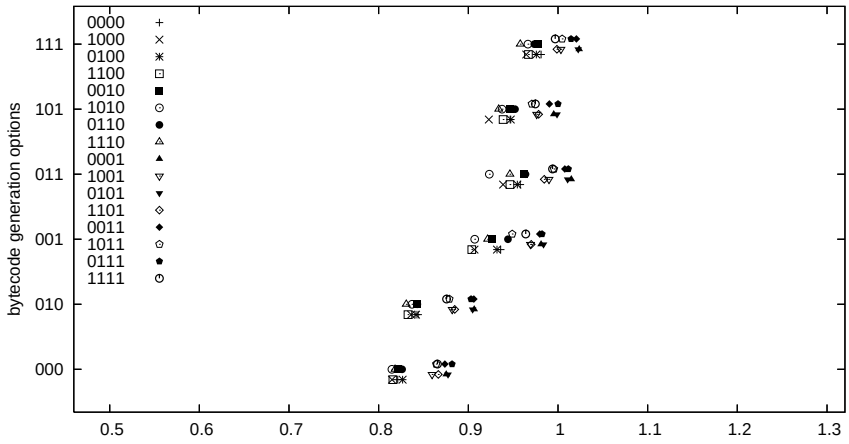

Figure 5.41: Version of Boyer-Moore theorem prover — PowerPC.

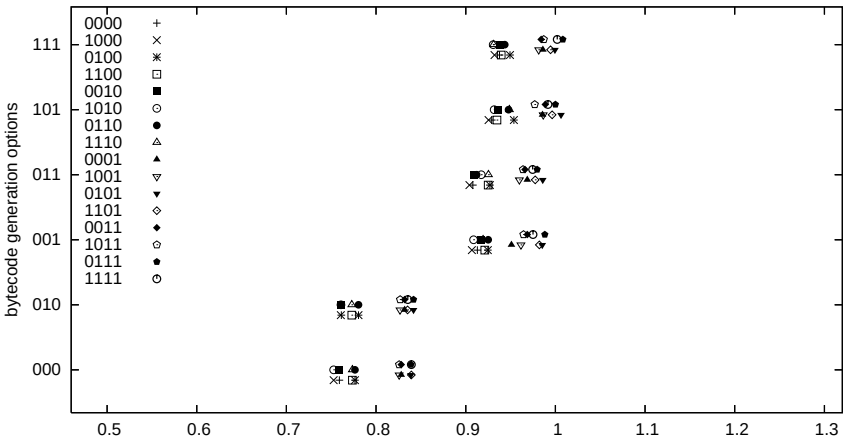

Figure 5.42: QuickSort — PowerPC.

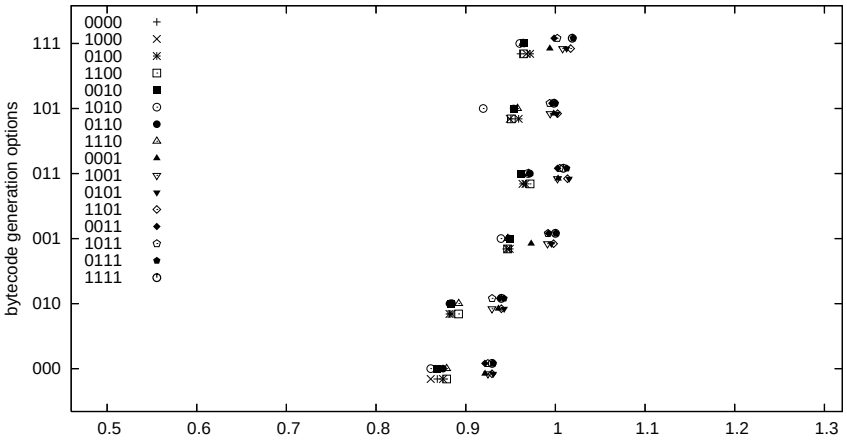

Figure 5.43: Calculate primes using the sieve of Eratosthenes — PowerPC.

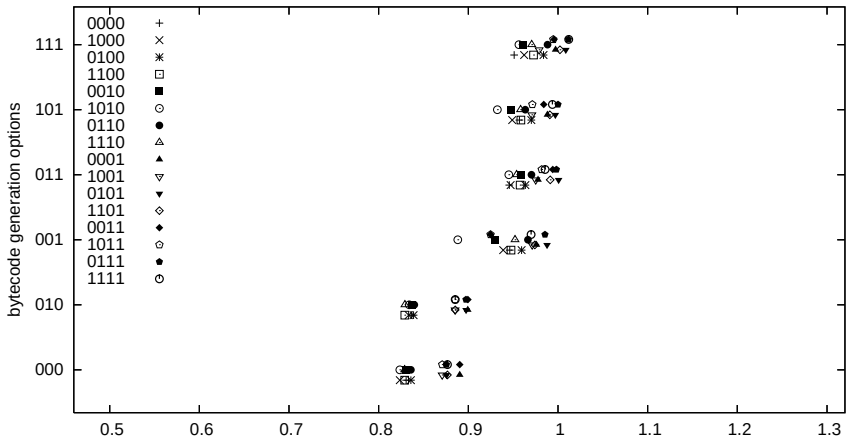

Figure 5.44: Natural language query to a geographical database — PowerPC.

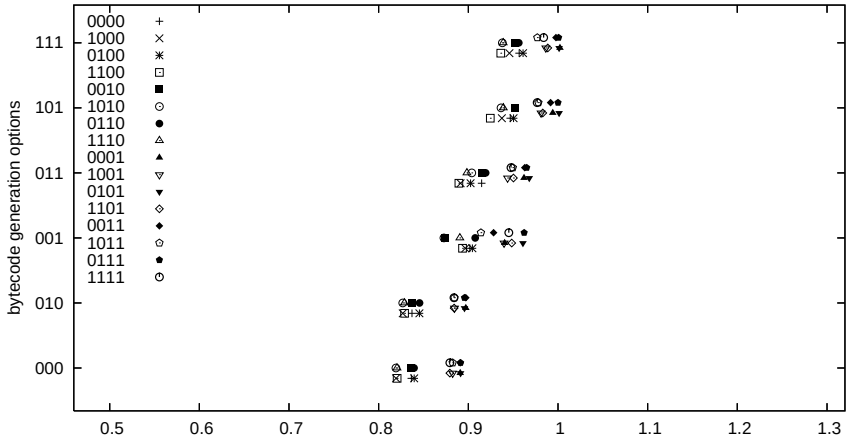

Figure 5.45: Chess knights tour — PowerPC.

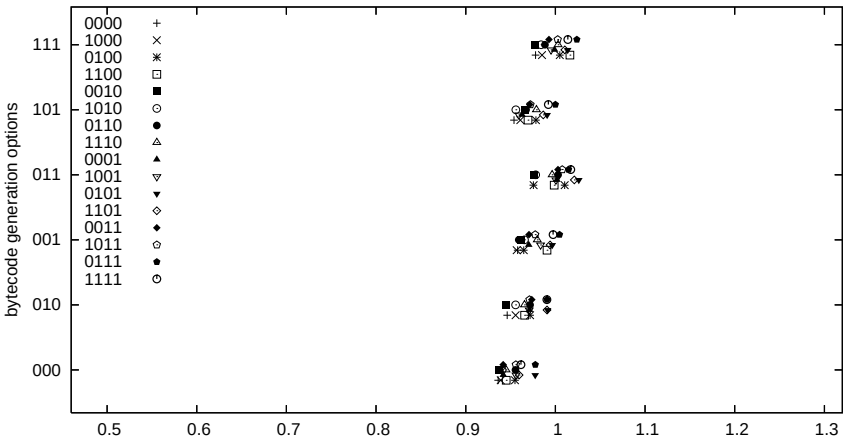

Figure 5.46: Simply recursive Fibonacci — PowerPC.

6

Comparing Tag Scheme Variations Using an Abstract Machine Generator

Summary

In this chapter we study, in the context of a WAM-based abstract machine for Prolog, how variations in the encoding of type information in *tagged words* and in their associated basic operations impact performance and memory usage. We use a high-level language to specify encodings and the associated operations. An automatic generator constructs both the abstract machine using this encoding and the associated Prolog-to-bytecode compiler. *Annotations* in this language make it possible to impose constraints on the final representation of tagged words, such as the effectively addressable space (fixing, for example, the word size of the target processor / architecture), the layout of the tag and value bits inside the tagged word, and how the basic operations are implemented. We evaluate a large number of combinations of the different parameters in two scenarios: a) trying to obtain an optimal general-purpose abstract machine and b) automatically generating a specially-tuned abstract machine for a particular program. We conclude that we are able to automatically generate code featuring all the optimizations present in a hand-written, highly-optimized abstract machine and we can also obtain emulators with larger addressable space and better performance.

6.1 Introduction

Dynamically typed languages, where the type of an expression may not be completely known at compile time, have experienced increased popularity in recent years. These languages offer a number of advantages over statically typed ones: the source code of the programs tend to be more compact (e.g., type declarations are not present and castings are unneeded), faster to develop, and easier to reuse.

In statically typed languages with prescriptive types the type of each expression is to be declared in the source code. This makes it possible to generate tighter data encodings and remove all type checks at compile time, which in general results in faster execution times and reduced memory usage and, hopefully, in the static detection of more programming errors. At the same time, extra redundancy is added in the cases where types can be inferred, and calls for a more rigorous (or strict) developing methodology. Additionally, some program and data structures which are natural for dynamically typed languages need to be adapted significantly to be cast into common, well-known type systems.

It is interesting to note that recent advances in program analysis can potentially reconcile to a large extent this apparent dichotomy by inferring automatically the properties needed to perform the optimizations brought about by static typing and by using such properties to perform non-trivial error and property checking which subsumes traditional type-based static checking [HPBG05, CMM+06, Rig04, CF91]. Also, even in the cases where types and other information cannot be statically inferred, performance of dynamic languages can be quite competitive, since modern compilers and abstract machines have evolved significantly and offer a high degree of optimization.

However, the same high degree of specialization and optimization that brings about the performance of modern compilers and abstract machines implies significant complexity in their design which in turn makes it very difficult to explore a large design space of modifications in order to obtain significant further advances, at least "by hand."

In this chapter we explore, in the context of a state-of-the-art WAM-based [War83, AK91] implementation of Prolog, how a number of variations in the way the core data structures of an abstract machine are implemented impact

performance and memory usage. Our objective is to draw general conclusions about which are the best optimization options. We use a Prolog-inspired high-level language to specify encodings, the associated operations, and in general to generate complete abstract machines using these variations. This language has a type and property system which, while quite flexible, is decidable at compile time, so that very efficient low-level code (C, in our case) can be generated.

6.2 Implementation of Dynamic Typing

Dynamic typing requires type information to be available at run time, and therefore it is customary to store it within the actual data during execution. In this chapter we focus on the widespread implementation technique which uses *tagged words*. In this approach a (usually reduced) set of *tags* is used to represent type identifiers which are used as the first level information regarding the type of the contents of the memory word. Other ancillary information, such as the garbage collection (GC) bits as needed by some GC algorithms, is often stored in the tag. These tags are usually attached to the actual data value, and they are typically stored together in the same machine word. There are multiple ways to do this using fewer or more bits for the tag, placing it in the upper or lower part of the word, etc. However, not all schemes to pair up tag and value offer the same performance, memory consumption, and addressable space size. The quality of a tagged word implementation actually depends on a compromise between the available addressable space and performance. The latter is dominated by the cost of operations on the tagged type: setting and getting the tag, reading the value, and manipulating the GC-related information. Without explicit hardware support for tagged words (the norm in today's general-purpose processors), implementers must rely on simple, natively supported operations like shifting and masking, and often the effectively addressable space has to be reduced in order to, e.g., be able to *pack* a (reduced-size) pointer in a word using the space left by the tag and the GC information.

6.2.1 Performance of Different Encoding Schemes

Some bit layouts appear, at least a priori, to allow implementing more efficiently the most often used operations. In practice, experimental results indicate that with modern processors a realistic cost estimation is very difficult to perform a priori due to the inherent complexity of the processor operation, since too many parameters have to be taken into account: superscalar architectures, out-of-order execution, branch prediction, size of the pipeline, different cache sizes and levels, SIMD instructions, etc. That makes it very difficult to extract performance conclusions from the source code which implements the operations on the tagged data, even if the final assembler code is known and can be analyzed thoroughly: every basic operation depends, in fact, on the previous history and current state of the processor units. The situation is aggravated when the overall abstract machine complexity is taken into account. The implementation of the basic operations is often quite tricky, and high-performance code is usually achieved by creating by hand a number of specialized cases of these operations to be used in different contexts (i.e., when certain preconditions are met). As a result the code describing the abstract machine is usually large and any change is tedious and error-prone, which limits the capacity to explore alternatives systematically.

In order to overcome these problems, and building on the work presented in Chapter 4 and 5 we have constructed a framework for generating abstract machines (and their corresponding compiler) *automatically* based on high-level descriptions of the placement of tags, GC bits, and values inside a tagged word, the size of the tagged word, the size of the native machine word, as well as some variations on the *shape* of the basic operations which deal with these tagged words. In all combinations, we aim at optimizing the resulting C code as much as possible, up to the point in which the result is often indistinguishable (in some cases identical) from what a highly skilled programmer would have written.

The concrete virtual machine skeleton we base our variations on is an implementation of the well-known WAM for Prolog. We argue that tagging schemes of modern Prolog virtual machines have requirements which are not unlike those of other dynamic languages: basic operations, encodings, and optimization possibilities at this level are similar in all of them. Additionally, many *statically-typed* programming languages present also dynamic features such as class inheritance, dynamic dispatching, disjunction of data structures, etc., which require similar

techniques.

In all these cases, variations on the tagging scheme and ancillary operations significantly affect both the generated code and the size of the data structures, and can have a critical impact on execution speed and memory consumption. In fact, as a result of the (non-obvious) complexity of the basic tag operations and their very frequent use, small variations in their implementation can have a significant effect on performance even in small programs.

Note that the purpose of this work is not to demonstrate that we are using the overall best scheme for implementing a Prolog abstract machine (which includes decisions about term representations [DN00], whether or not to use stack variables, last call optimizations, ..., as well as parameters that affect performance including the bytecode encoding, instruction set, unification algorithm, C compiler, etc.), but rather to show how a higher level description (higher than C code) can be parameterized to generate variations of a tagging scheme, and to study the different results in terms of performance and memory usage.

6.3 Describing Types in imProlog

As mentioned before, classical implementations of abstract machines involve non-trivial algorithms with data representations defined at the bit level, and very involved (hand-)codifications of the operations on them. The C language is a common implementation vehicle because the programmer can control many low level aspects and the quality of assembler code generated by current compilers is quite high, so that it is essentially a high-level way of generating quasi-optimal assembler code. The usual development techniques involve using preprocessor macros and conditional code, inlining C functions, using bit packed structures, etc. However, in more involved cases (e.g., when extensive instruction merging is to be performed, or when variations are being explored), this approach falls short, and some sort of automatic code generation (which can in part be done with advanced preprocessors) needs to be adopted to control complexity and avoid coding errors.

In this chapter we use the abstract machine generator framework described in Chapter 4 and we use the imProlog language (Chapter 5) to write the abstract machine and to specify the low-level data layout. imProlog is a restricted Prolog

subset extended with *mutable variables* as first-order citizens and specific annotations to declare constraints on how to encode the tagged words in the machine memory. The compilation process ensures that the optimizations which allow efficient native code to be generated (controlled unfolding, specialization, unboxing of types, avoiding unnecessary trailing, backtracking, etc.) are met by the initial code, and rejects the program / module otherwise. Although the compiler and the language is not yet suitable (or intended) for writing general-purpose applications, it has been successfully applied in the implementation of a very efficient Prolog emulator that is highly competitive with hand-written state of the art Prolog emulators such as those of Ciao [BCC+09], Yap [SCDRA00], SICStus [Swe99], hProlog etc., and which is used as the basis for our experiments.

The C code that the imProlog compiler generates could obviously have been hand-written given enough time and patience. However, as we stated before, our goal is to reduce the burden on the abstract machine designer (which is likely to reduce programming errors) through the use of a higher-level language and improve correctness (the imProlog compiler can statically detect ill-typed operations with respect to the expected tags, without introducing run-time checks), while at the same time making it easier to maintain and extend the code.

The approach also allows exploring more possibilities of optimization. This includes stating high-level optimization rules and letting the compiler determine (maybe with additional compilation hints) whether applying them to generating acceptable native code is feasible or not at each point. Such optimization rules are used at any point in which they can be applied while generating the C code —i.e., in all parts of the abstract machine code corresponding to the implementation of bytecode instructions and other operations. This relieves the programmer from repeatedly having to remember to apply the same coding tricks at all points where similar operations are implemented, which also reduces the possibilities of introducing errors.

6.3.1 Types in imProlog

Since imProlog is the implementation language, the data representations used in the abstract machines generated are described as imProlog types. The language used in imProlog to describe types is (as in Ciao and CiaoPP [HPBG05]) the same that is used to write programs. Types are written as predicates, and the

set of solutions of the predicate is the set of values in the type. Type definitions in imProlog can have recursive calls, disjunctions, etc., as long as the compiler is able to analyze them and extract enough information to generate an efficient, unique encoding for the type.

From a syntactic point of view, Ciao's functional notation [CCH06] is convenient to write type definitions, and we will use it for conciseness in this chapter. This notation makes the default assumption that function outputs correspond to the last predicate argument (i.e., $\mathsf{X} = {\sim}\mathsf{p}(T_1,\ldots, T_{n-1})$ stands for $\mathsf{p}(T_1,\ldots, T_{n-1}, T_n)$, $\mathsf{X} = T_n$). The syntax for feature terms uses $\mathsf{A}{\cdot}\mathsf{B}$ as a function that returns the contents of feature B in A. For mutables, the goal $\mathsf{A} \Leftarrow \mathsf{B}$ sets B as the content of A, and the function $@\mathsf{A}$ obtains the value stored in A. The type ${\sim}\mathsf{mut}(\mathsf{T})$ stands for the mutables whose values are restricted to those of type T.

The imProlog compilation process distinguishes the type a variable holds at a given time and the type that describes the possible values that can be encoded in that variable (the *encoding* type). Choosing the right encoding for each variable generates a statically typed (imProlog) program, in which the type of every expression must be either inferred by the imProlog compiler or explicitly stated so that unique memory encodings for variables and mutables can be statically determined. The compiler will reject programs with non-unique feasible type encodings for expressions / variables.

Although this presents a limitation for general-purpose programming, we think that it can be tolerated in the case of writing an abstract machine, where high performance is paramount. If a unique data representation cannot be statically determined, the compiler would need to generate dynamic tests to distinguish between the different encodings for the same data (e.g., comparing variables containing atoms which are encoded in different ways) which may result in a performance loss. Detecting that this is the case would be difficult if the compiler does not inform the user that this overhead has been introduced.[1]

[1] In fact, these errors can be interpreted as a quality report. An interesting option would be to define a tolerance value that allows or disallows certain ways of generating code.

6.3.2 Feature Terms and Disjunctions of Types

In Chapter 5 imProlog types were limited to built-in types, which are seen as black boxes, and which reflect machine and low-level types. As a natural step, we have extended the imProlog language and compiler with the required machinery to express the complex types needed to define *tagged words*: structured types (based on feature terms), and disjunctions of types (based on dependent types). For the sake of efficiency, we have improved the support for low-level encoding at the level of bits. The implemented support for dependent types allows indexing the feature term w.r.t. a single atomic feature. The restriction in that case is that the discriminant field must have the same encoding properties in all cases. When one of the type features is accessed and the discriminant value is known at compile time, no checks are necessary, since the type encoding is precisely defined. On the other hand, if the discriminant value is not unequivocally defined at compile time, the code is wrapped in tests that cover all the possible cases and which associate them with a code version specialized for the case in hand.

6.3.3 Defining a Hierarchy of Types

Inheritance can be captured by the semantics of logical implication [AKN88], and this idea has been used to define our type hierarchy by means of feature terms and disjunctions of types. We will use this hierarchy to define the *tagged* type (Section 6.4). Inheritance makes it possible to define a (complex) set of types as a group of predicates making up a hierarchy tree where each type is a node, the root type is the root of the tree and the final types, which are not needed by any other type, its leaves. Using this coding style we can group together definitions belonging to the same type in an "object-oriented" fashion. Although it is out of the scope of this work, this makes it possible to extend or specialize the type just by adding or removing final types. The general skeleton of each type definition includes the following rules:

Rule 1 : supertype(T) :− type(T).

Parent-child relationship between some type and its supertype (except for the root type). This is to be read as "type is a kind of supertype".

Rule 2 : type(T) :− T·kind = 'unique−atom', type$_i$(T).

This rule defines a final type (a leaf in the tree) in the hierarchy. This gives the basic solution for the type, which must be mutually exclusive with all other types. Only leaf nodes define the *kind* feature.

Rule 3 : $\mathsf{type_i(T)} :- \mathsf{supertype_i(T)}, \ldots$

The definition of the features that are specific to the type. It may include the parent's invariant and extend it with other features.

Rule 4 : $\mathsf{operation(A, \ldots)} :- \mathsf{type(A)}, \ldots$

One rule for each operation on the type, using first argument indexing, when available, for efficiency.

The solutions (i.e., values belonging to the type) given by a type declaration are those generated by the type itself and by all of its descendants. The predicates $\mathsf{type_i}$ and type differ as follows: the former defines the contents that are exclusive to type type (and, optionally, the types it inherits from) and the latter defines the type itself (and, recursively, all the types which inherit from it).

6.4 Specifying the Tagged Data Type

We define in imProlog a type that implements the basic data structure (*tagged* type) for the Ciao abstract machine. It is a discriminated union of heap variables, stack variables, constrained variables, structures, list constructors, atoms, and numbers. This type (or that of mutables constrained to values of this type) appears everywhere in the abstract machine state (heap cells, temporal registers, values saved in a choice point, local values in local frames, etc.).

The novelty of this approach —for the generation of abstract machines— is that the full definition is composed of the high level description, used by analysis and optimizations, and the low level annotations and optional user-defined compilation rules that determine how optimal native code is generated for the type definition and operations.

6.4.1 The Tagged Hierarchy

The tagged type can be viewed as the hierarchy in Figure 6.1. A simplified subset of the description for the type is shown in Figure 6.2, for generic nodes, and in

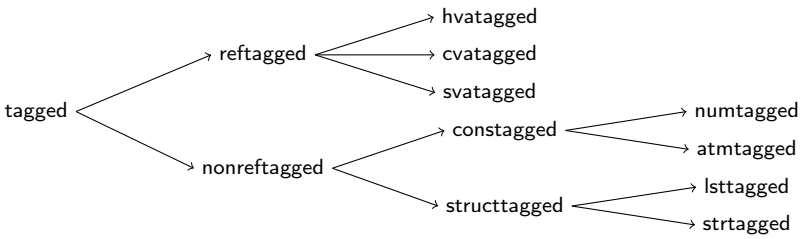

Figure 6.1: Tagged hierarchy.

Figure 6.3, for the leaf nodes. In this case the feature that distinguishes all leaves is called *tag* (the type of the *tag* feature returns the set of all required tags). We also describe, as an example, the code for the operation unify_cons, that unifies the tagged in its first argument with a single-cell tagged (a tagged whose value is perfectly defined without consulting any heap or stack location, such as small integers or atoms).

That specification defines the possible tagged words. The defined tagged word may optionally contain two GC bits [ACHS88] or place them in an external memory area, depending on whether predicate `ext_gc_bits` is true or false. Partial evaluation is used to statically specialize $\texttt{tagged}_i$ with respect to the usage of external GC bits.

Note that low-level encoding details are not present at this point; they are introduced later as annotations. This separation facilitates automatic handling of more properties that it would be possible with a low-level implementation (where the domain information for tagged words is lost when they are translated to integers). For example, it is possible to specify that all members of the heap have values of type `nonstacktagged`, to detect errors or remove checks for svatagged automatically, where the type is defined as:

```
nonstackreftagged(T) :- hvatagged(T) ; cvatagged(T).
nonstacktagged(T) :- nonstackreftagged(T) ; nonreftagged(T).
```

The tagged type

```
tagged_i(T) :-
  ( ext_gc_bits → true
  ; T·marked = ∼bool, T·forward = ∼bool ).
unify_cons(R, Cons) :- derefvar(R), unify_cons_d(R, Cons).
...
```

References (a kind of tagged)

```
tagged(T) :- reftagged(T).
reftagged_i(T) :-
        tagged_i(T),
        T·ref = ∼mut(tagged).
unify_cons_d(R, Cons) :- reftagged(@R),
        ( trail_cond(@R) →
            trail_push(@R)
        ; true
        ),
        (@R)·ref ⇐ Cons.
...
```

Non-references (a kind of tagged)

```
tagged(T) :- nonreftagged(T).
nonreftagged_i(T) :- tagged_i(T).
unify_cons_d(R, Cons) :- nonreftagged(@R),
        @R == Cons.
...
```

Constants (a kind of non-reference)

```
nonreftagged(T) :- constagged(T).
constagged_i(T) :- nonreftagged_i(T).
...
```

Figure 6.2: Some generic nodes in the hierarchy.

Heap variables (a kind of reference)

```
reftagged(T) :- hvatagged(T).
hvatagged(T) :- T·tag = hva, hvatagged_i(T).
hvatagged_i(T) :- reftagged_i(T).
trail_cond(A) :- hvatagged(A), A·ref < (∼s)·heap_uncond.
...
```

Small integers (a kind of constant)

```
constagged(T) :- numtagged(T).
numtagged(T) :- T·tag = num, numtagged_i(T).
numtagged_i(T) :- constagged_i(T), T·num = ∼int.
```

Figure 6.3: Some leaf nodes in the tagged hierarchy.

6.5 Optimizing Type Encodings

The data type that customarily constitutes the building unit of WAM-based abstract machines is a machine word (or *tagged*) where the value of some of its bits (the *tag*) provides the meaning of the rest of them. We obtain the same effect from higher level type descriptions for the *tagged* type by constraining the size available to represent the type and each of its features. In this section we will describe the annotations and low-level optimizations used to implement the *tagged* type. The set of all of these annotations constitute the parameter values that define a *tag scheme variation*.

6.5.1 Bit-level Encoding

The basic data types supported in imProlog include the unboxed form of atoms (encoding types that define a set of atoms, where each atom is represented as a number), numbers (limited to native types, such as 32 bit or 64 bit integers, floats, etc.), and mutables (as local C variables or stored as pointers to a memory location where the value is contained). This is possible because of the restrictions on the values which can be stored in a variable to avoid boxing: they must be completely instantiated and contain values defined by their type. Also, dereference chains of arbitrary length are not allowed, only fixed-length chains as specified in the

type. With bit-level annotations for features, we limit the total size available to represent the data type, and the bits available for each of the features and their relative bit position.

The final bit layout is determined by annotations that control the bit offset where each feature is placed. In particular, they may be allocated in the *upper* or *lower* free part of the data bits, or even *split* between these two locations. For the *tagged* type, we abbreviate the storage location of the tag bits as `h` for upper bits, `l` for lower bits, or `s` for split bits. For GC marks, we use `H` for upper bits, `L` for lower bits, and *external* when they are allocated in a separate section of the stack (enabled by `ext_gc_bits`/0).

The bit layout affects how feature values can be extracted: if tags are in the lower part, a mask just gives its value; if they are in the upper part, a left rotation is needed. However, extracting a tag value is not so common when optimized tag tests are used, since most operations are reduced to bit tests. The location of tags and GC bits affects the access to pointers and other values. A field is extracted by masking out other fields and shifting the value accordingly. If the fields to be removed are known at compile time, then subtraction is used instead of masking, so that use can be made of the indirect addressing mode in the assembler code (pointer+offset), and combined at compile time with other arithmetic operations such as, e.g., the additions that need to be performed when we use a displaced pointer from a *tagged* or add two small numbers.

In the case of pointers, the fixed pointer bits must be placed back, with a previous shift to reintroduce the alignment bits, if they have been used (note that the shifting to extract the pointer and reinsert the alignments may be canceled out, such as in `s` and `hL` in 32 bits or `lH` or `lL` in 64 bits). Note that each bit placement has its advantages and disadvantages, since optimizing some operations with one configuration sometimes makes others more costly, which ultimately complicates performance estimation.

6.5.2 Trade-off: Limited Address Space

Encoding both fixed-length reference chains and mutables in the presence of bit-size restrictions implies some implementation tricks to define pointers where some bits are free to be used for other purposes. For example, alignment bits can be reused or some extra bits can be made available by fixing the location of a memory

region with the `mmap` system call. Note that storing n-bit pointers in less than n bits results in address space limitation. Such limitation may not be an issue in 64 bit machines (were the address space is huge, much larger than the physical memory, as it was also the case for 32 bit pointers in the not-so-remote past), but currently it may be a problem in 32 bit machines where more than 1GB of physical memory is common nowadays. In the case of our *tagged* word representation, the available *address space* is determined by the number of bits required for tags and GC, the tagged word size and the use of external or internal GC bits. Here we see even more clearly why a flexible way to implement abstract machines with different tagged schemes matters, since a scheme that represents a good compromise between address space and performance for a 32 bit machine may not be the best for 64 bits (even more, if we consider architectural differences). Also, while it is tempting to decide to simply concentrate on 64-bit machines, since they are now the norm for new general-purpose machines, it should be noted that 32 bit machines are still in widespread use and, furthermore, other computing platforms (such as, e.g., portable gaming devices, cell phones, etc.) have physical limitations which make them have 32-bit address buses.

6.5.3 More Control over Tag Representation

Although an automatic assignment of encoding values for each of the atoms in a type may be efficient for most applications, it is not enough to implement optimal operations for the *tagged* type, where choosing the right numeric value for each *tag* may play an important role in reducing the number of assembler instructions required to implement essential basic operations on the *tagged*. The reason is that for some encodings a single assembler instruction can check for more than one tag at the same time, yielding faster code to switch on tag values. To this end, the imProlog compiler allows explicit definition of atom encodings and a large set of predefined compilation patterns to build indexing trees. imProlog does not have a *switch* construct, and, as Prolog, it is based on indexing to optimize clause selection, in two steps defined as Figure 6.4 illustrates.

Index extraction: Group together discontiguous clauses and transform the program to make explicit the possible indexing keys (unfolding cheap type checks, removing impossible cases, adding the default case). E.g., suppose that

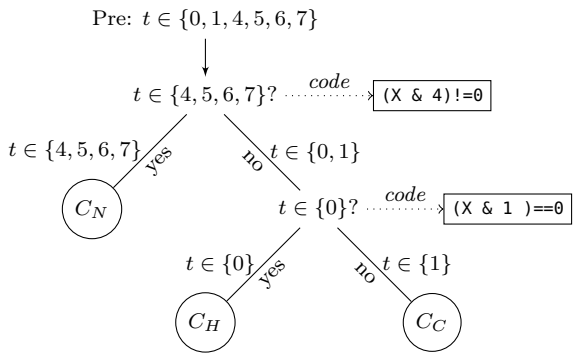

Figure 6.4: Obtaining indexing code.

nonstacktagged(X) holds on entry, then for the code on the left we obtain the index on the `tag` feature of X on the right:

$$
\begin{array}{l|l}
\begin{array}{l}
(\ \mathsf{hvatagged(X)} \rightarrow C_H \\
;\ \mathsf{cvatagged(X)} \rightarrow C_C \\
;\ \mathsf{svatagged(X)} \rightarrow C_S \\
;\ \mathsf{nonreftagged(X)} \rightarrow C_N \\
)
\end{array}
&
\begin{array}{l}
(\ \mathsf{X{\cdot}tag = hva} \rightarrow C_H \\
;\ \mathsf{X{\cdot}tag = cva} \rightarrow C_C \\
;\ (\ \mathsf{X{\cdot}tag = num} \\
\quad ;\ \mathsf{X{\cdot}tag = atm} \\
\quad ;\ \mathsf{X{\cdot}tag = lst} \\
\quad ;\ \mathsf{X{\cdot}tag = str})\ \rightarrow C_N \\
)
\end{array}
\end{array}
$$

Low level indexing: Compile low level indexing, by taking into account where the tag is located (e.g., lower or upper bits) and how it is encoded (e.g., hva: 0, cva:1, ..., str:7). Depending on whether optimized tests are used or not, we distinguish several compilation modes, which we will call the `swmode` parameter: **sw0**, **sw1**, and **sw2**.

In **sw0** mode, the code generator emits a simple if-then-else (when two cases are used), or a C switch statement (usually compiled internally as a indirect jump to an array that maps from tag values to code labels) — see Figure 6.5 for an example where the tag is stored in the lower 3 bits of X.

Note that although C compilers can generate very clever code for switches, they are unable to infer the value of bit-encoded members or make use of user-provided assertions (so they cannot optimize some tests). Although the indirect jump version may look very optimized, it may hinder the benefits of the branch

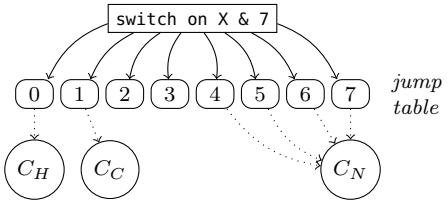

Figure 6.5: Code generation using **sw0**.

prediction unit in modern processors [MYJ07]. In **sw1** the compiler tries to emit better code by grouping together tests for several values in a single expression (that can be cheaply compiled in few assembler instructions by the C compiler) that we will call *bitfield test*. We call a bitfield test with precondition S and postcondition P, for the k-bits in the p-bit offset, a low level test *Test* that given a n-bit word `X`, where t=`(X >>` p`)&((1<<`k`)-1)`, checks that if $t \in S$ and *Test* is true, then $t \in P$. Examples of tests for the 3 lower bits are:

- Pre:$t \in \{0..7\}$ Post:$t \in \{4,5,6,7\}$ Code: `(X & 4)!=0`
- Pre:$t \in \{0,1\}$ Post:$t \in \{0\}$ Code: `(X & 1)==0`
- Pre:$t \in \{4,5,6,7\}$ Post:$t \in \{4,6\}$ Code: `(X & 1)==0`

With the optimized tests, the search space is divided in two (the values in postcondition, and the values in the precondition minus the postcondition). When **sw2** is selected, more complex indexing trees are treated, that would otherwise be translated to C switches. Switch rules represent heuristics to reach the desired branches more efficiently. The rules are specified as nested disjunctions (that define a binary tree where the leaf nodes are sets of values) and are supposed to cover all cases so that the implicit precondition is the union of all the sets. For example: $\{4,5,6,7\} \vee (\{0\} \vee \{1\})$ states that it should do a test to distinguish between $\{4,5,6,7\}$ (which is a leaf node) $\{0,1\}$, then do the same for $\{0\}$ to distinguish between $\{0\}$ and $\{1\}$. When the test code is inserted, we obtain the low level definition of the indexing code (Figure 6.4).

6.5.4 Extending Garbage-collector for External and Internal GC Bits

GC bits are only active during garbage collection. With external GC bits, When GC starts a special section is reserved to store GC bits for every WAM stack.

A GC pointer resolution operation which obtains the location of the GC bits for a pointer from some memory region, while reasonably fast, accumulates an excessive overhead when it is performed for every GC bit access.

To minimize the number of GC pointer resolutions without having to maintain two slightly different GC algorithm codifications (for internal and external GC bits), we define an abstract "pointer to tagged" data type. When internal GC bits are selected, the pointer is a normal pointer. When external GC bits are used, the pointer to tagged is actually a pair of pointers: one points to the actual tagged word and the other one to the byte that contains the GC bits. Pointer displacement, read, and write operations are done for both pointers, obtaining a similar performance for the case of external and internal GC bits.

The GC algorithm (and the code itself) is therefore generic with respect to the format of the tagged words.

6.5.5 Interactions with Bytecode

There is an implicit dependence between the abstract machine definition and the compiler that generates bytecode to be executed in that machine. By extending the framework to describe the *tagged* type, the assumptions about data sizes become a parameter for the compiler (in addition to other parameters described in Chapter 4 and 5, related to the accepted instruction set, instruction merging, how instructions are encoded, etc. [MCH07, MCPH05]). When the abstract machine code is generated, all this information is fed into the compiler; this approach ensures that the compiler and the abstract machine are synchronized. This is needed, for example, when the compiler requires the size of a *tagged* to insert heap overflow tests prior to the construction of known terms, or to decide whether multiple-precision arithmetic is needed when building an integer, which depends on whether the number of bits used to represent the value of a small number (num feature in numtagged) is enough to store the value.

The *tagged* size also affects the size of opcodes and operands in bytecode instructions. In some architectures n-bit words must be n-bit aligned (for efficiency or architectural constraints). While it is possible to introduce paddings to keep operands always aligned [SC99], in order to allow a more compact bytecode representation, the instruction set in the standard Ciao system uses instructions whose size is not a multiple of the largest machine word (32 bits in a 32-bit machine).

In that case, several opcodes stand for differently padded versions of the same instruction.

In our system the instruction set is generated automatically from a higher-level description, creating all the necessary padded versions automatically. E.g., the following code defines an instruction that unifies an X register (a mutable variable whose low-level address is specified by an operand in the bytecode representation containing an indirect offset to an address in the global state), and a `constagged`:

```
:- entry(u_cons(xreg,constagged)).
u_cons(A, Cons) :- T ⇐ @A, unify_cons(T, Cons).
```

where the code that links the operands of that bytecode with the arguments is synthesized by the abstract machine generator. In the instruction above, if the instruction opcodes are stored as a 16 bit word and the X registers as a 16-bit word, the second operand (that is 32-bit sized) is aligned to 32 bits or not, depending on whether the instruction begins at a 32-bit aligned address or not. Then, two versions of the same instruction (padded and non-padded) are emitted; in the latter a 16-bit pad before the Cons operand is inserted to ensure that it is aligned to 32-bits.

6.6 Evaluation of Tag Scheme Variations

We tested 48 combinations of the available address space possibilities, bit layouts, and optimizations. For each combination we generate an abstract machine on which 22 benchmarks (see table 6.1) are executed. Some of these benchmarks are well known and relatively small, while others (for example, `witt` and `wumpus`) can be considered having a medium size (from 400 to 500 lines). In any case, these benchmarks exercise operations whose performance will be greatly affected by the different compilation options under study, so they can be taken as reasonable witnesses of this impact in efficiency. We measured the memory usage and the total execution time, with and without GC. Each benchmark was executed three times, taking the shortest execution. The experiments were run on four machines, with different architectures or processors, taking from 5 to 12 hours, depending on the machine speed. In the machines where it was possible, the O.S. was running in native 64-bit mode (executing also the 32-bit tests).

boyer	Simplified Boyer-Moore theorem prover.
crypt	Arithmetic puzzle involving multiplication.
deriv	Symbolic derivation of polynomials.
exp	Work out 13^{7111}.
factorial	Compute the factorial of a number.
fft	Fast Fourier transform
fib	Simply recursive computation of the n^{th} Fibonacci number.
guardians	Prison guards playing game.
jugs	Jugs problem.
knights	Chess knight tour, visiting only once every board cell.
nreverse	Naive reversal of a list using append.
poly	Raises symbolically the expression `1+x+y+z` to the n^{th} power.
primes	Sieve of Eratosthenes.
qsort	Implementation of QuickSort.
queens11	N-Queens with $N = 11$.
query	Natural language query to a database with information about countries.
tak	Computation of the Takeuchi function.
trie	Word indexer using tries.
wave_arr	Signal processing using updatable arrays.
wave_dyn	Signal processing using dynamic facts.
witt	A conceptual clustering algorithm.
wumpus	Wumpus world game.

Table 6.1: Benchmark descriptions.

We will use an abbreviated notation for the different combinations of compilation options and architectures, using the following codes:

highbittags	`h`	splitbittags	`s`	lowbittags	`l`
lowbitgc	`L`	highbitgc	`H`		
sw0	`0`	sw1	`1`	sw2	`2`

`sparc64`	64-b. Sparc/Sol. 5.10, 3MB cache
`p4-64`	64-b. Intel P4/Lin. 2.6, 2MB cache
`xeon-32`	32-b. Intel Xeon/Lin. 2.6, 512Kb cache
`coreduo-32`	32-b. Intel Core Duo/Lin. 2.6, 2Mb cache

The C compiler (gcc) we used to compile the emulators had a different version in each architecture. This may affect the conclusions in which speedups from different architectures are compared, which we think is sensible only for `xeon-32` and `coreduo-32` (Section 6.6.3). In these two cases, however, the "best options" happen to be the same. This makes us confident that the results will also be applicable in a more long term, when current compilers have been outdated.

For each architecture the speedups and memory usage figures are normalized w.r.t. a default case which corresponds to `hL2` with 26 bits of address space (the smallest we tested). This normalization makes comparisons among different address spaces meaningful using speedups w.r.t. the fixed basic case. We want to note that this *default case* is in itself quite efficient: it was generated using as basis the Ciao virtual machine (itself an offspring of the SICStus 0.6/0.7 virtual machine) with some improvements (computed goto/threaded bytecode, smarter instruction opcode assignment, ...) written in imProlog which made the virtual machine faster than the stock Ciao one. Table 6.2 helps to evaluate the speed of the *default* emulator (in the `coreduo-32` machine) w.r.t. to Ciao 1.13 and other well-know Prolog systems: Yap 5.1.2, hProlog 2.7, SWI-Prolog 5.6.55.

6.6.1 Address Limits and Memory Usage

The different addressing possibilities are described in Table 6.3. The leftmost column gives a name to each of the combinations and the one to its right describes the (physical) size of the *tagged* word (32 or 64 bits), the size of the stored pointer (64 bits when running in a 64 bit architecture and O.S.), and whether GC bits are internal or not to the tagged word (`extgc`). The option `qtag`, used in the default tag scheme in Ciao, reserves one bit to represent special functors for *blobs*. Note that not all combinations are possible in all studied architectures. The following columns provide the number of bits available for pointers, the number of bytes to be aligned to, and the resulting limits to addressable space.

The growth ratio in memory consumption that a given choice implies (w.r.t. the default case) appears in Table 6.4. Within each case, the figures are the same for every architecture in which the case can be implemented. However, every memory zone has a different growth ratio, and the relative growth for every area is not homogeneous for all the combinations. The reason is that the memory

Benchmark	Yap	hProlog	SWI	Def.	1.13
boyer	1392	1532	11169	1604	2560
crypt	3208	2108	36159	3460	6308
deriv	3924	3824	12610	3860	6676
exp	1308	1740	2599	1624	1400
factorial	4928	2368	16979	2736	3404
fft	1020	1652	14351	1548	2236
fib	2424	1180	8159	1332	1416
knights	2116	1968	11980	2352	3432
nreverse	1820	908	18950	2216	3900
poly	1328	1104	6850	1160	1896
primes	4060	2004	28050	2520	3936
qsort	1604	1528	8810	1704	2600
queens11	1408	1308	24669	1676	3200
query	632	676	6180	968	1448
tak	3068	1816	27500	2964	5124

Table 6.2: Speed comparison (`coreduo-32`).

Name	Options	Ptr. bits	Align. (bytes)	Limits (Mb)
addr26	tagged32*pointer32*qtag	26	4	256
addr27	tagged32*pointer32	27	4	512
addr29	tagged32*pointer32*extgc	29	4	2048
addr30	tagged64*pointer32	30	8	4096
addr32	tagged64*pointer32*extgc	32	8	4096
addr59	tagged64*pointer64	59	8	full
addr61	tagged64*pointer64*extgc	61	8	full

Table 6.3: Options related to the address space.

space used by objects does not change uniformly for all objects.[2] Therefore, data about actual memory usage has to be taken experimentally, as the ratio between these object types is different for each program.

[2]For example, large numbers or floating point numbers have a special, structure-based representation (a *blob*) whose size does not grow linearly with the size of the tagged word.

Addr.	Total	Program	Heap	Local	Trail	Choice	GC'ed memory
addr26	1.00	1.00	1.00	1.00	1.00	1.00	1.00
addr27	1.00	1.00	1.00	1.00	1.00	1.00	1.00
addr29	1.09	1.00	1.25	1.25	1.25	1.25	1.10
addr30	1.53	1.22	2.03	1.59	2.15	1.33	1.29
addr32	1.63	1.22	2.28	1.79	2.42	1.50	1.38
addr59	1.92	1.81	2.03	2.00	2.10	2.00	1.29
addr61	2.02	1.81	2.28	2.25	2.36	2.25	1.38

Table 6.4: Memory growth ratio.

The differences in program memory come from the need to use extra *padding* bytes in some instructions, which have a different impact in the different tag schemes.

As objects in different tagging schemes take different amounts of memory (for example, `addr30` can address 4GB of memory, but each tagged word takes twice as much space as in `addr29`), Table 6.3 is not really useful to decide which is the best scheme memory-wise: it is much more useful to reason about the number of *objects* which can actually fit in the memory addressable so that, leaving aside speed considerations, using `addr29` may be more advantageous, as it uses half as much memory.[3] This is shown in Table 6.5, where the memory address limits have been *adjusted* taking into account the data in Table 6.4, to finally work out a ratio of the number of objects which can actually be created. Note that in practice all the physically available memory is addressable in 64-bit architectures.

Both `addr30` and `addr32` use two 32-bit (4 bytes) words for a tagged word in a 32-bit architecture. Pointers must therefore be aligned to 4-byte boundaries, and their two less significant bits have to be zero. Hence, both `addr32` and `addr30` can address 4GB of memory. The latter is included in order to study how using external GC bits impacts performance. Although the underlying emulator differs, those schemes are similar to the one used in ECLiPSe [ECR93].

[3]Also, note that stock Linux kernels make only 3Gb available for user processes in 32-bit architectures.

Addr.	Total	Program	Heap	Local	Trail	Choice
addr26	256 (1.00)	256 (1.00)	256 (1.00)	256 (1.00)	256 (1.00)	256 (1.00)
addr27	512 (2.00)	512 (2.00)	512 (2.00)	512 (2.00)	512 (2.00)	512 (2.00)
addr29	1876 (7.33)	2048 (8.00)	1638 (6.40)	1638 (6.40)	1638 (6.40)	1638 (6.40)
addr30	2669 (10.43)	3355 (13.11)	2018 (7.89)	2579 (10.08)	1906 (7.45)	3076 (12.02)
addr32	2513 (9.82)	3355 (13.11)	1794 (7.01)	2293 (8.96)	1694 (6.62)	2734 (10.68)
addr59	all memory	all memory	all memory	all memory	all memory	all memory
addr61	all memory	all memory	all memory	all memory	all memory	all memory

Table 6.5: Effective addressable limits (and ratio w.r.t. default case) for address space options.

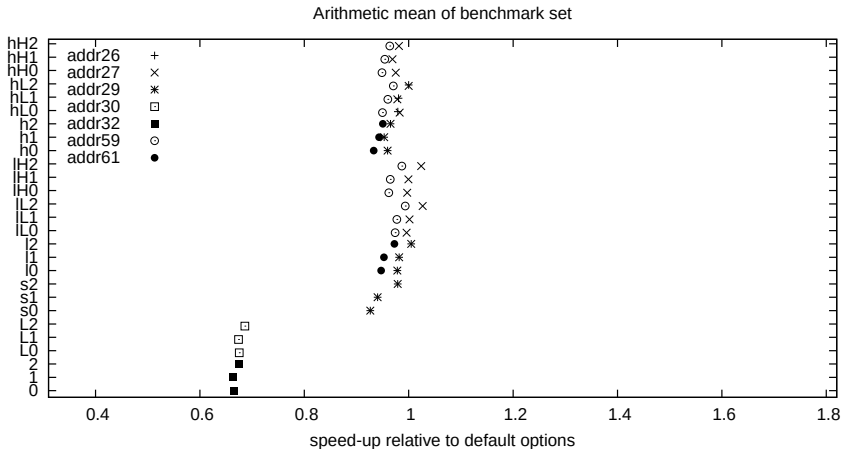

Figure 6.6: Arithmetic average of speedups (`sparc64`).

6.6.2 General Speed-up Analysis

We have summarized in a series of plots (Figures 6.6.2, 6.6.2, 6.6.2, 6.6.2) the arithmetic[4] average of speedups obtained assigning the different combinations of placements of GC bits, tags, and tag-related primitives in the `Y` axis, the option which selects the address space in the shapes of the dots, and the speedup in the `X` axis.

Some general conclusions can be drawn from these plots. First, the Sparc 64 architecture seems to have a very regular behavior (probably due to its large number of registers), except for some specific combinations of options which, non

[4]Plots with geometric average were practically identical.

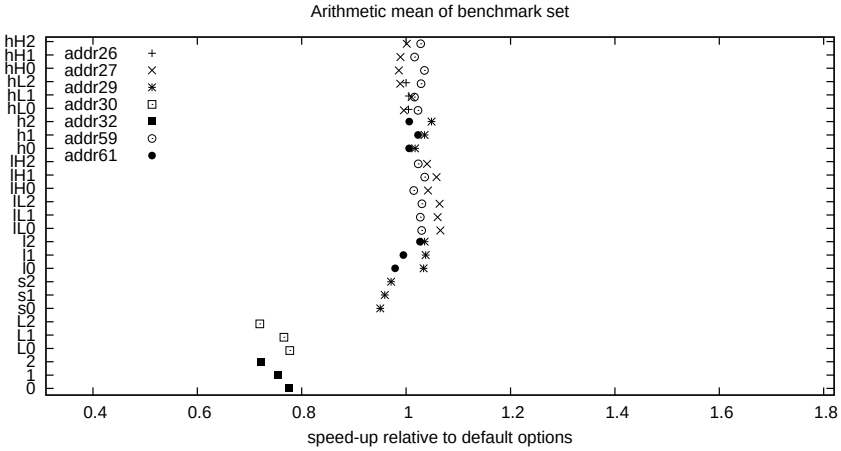

Figure 6.7: Arithmetic average of speedups (`p4-64`).

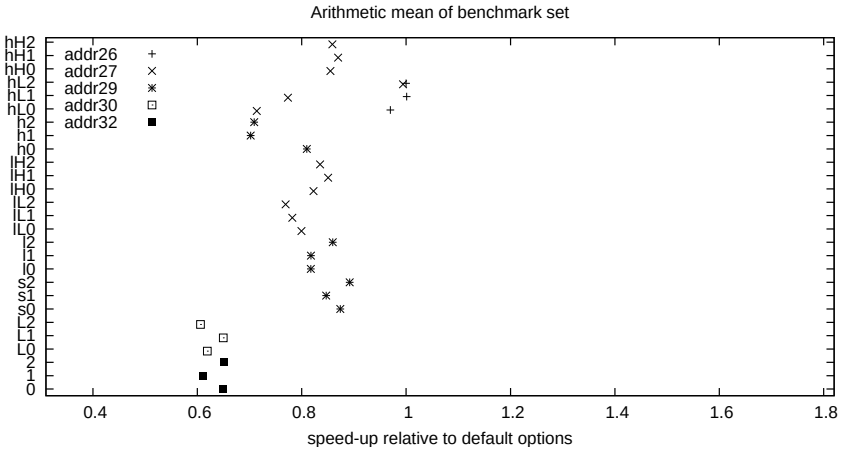

Figure 6.8: Arithmetic average of speedups (`xeon-32`)

.

surprisingly, perform badly in all architectures. The Intel processors, on the other hand, show wide variations, both for the different options in a given processor and also among the tested processor families. The latter can be attributed to the large differences in the internal architecture among Intel processors, which share little more than a common assembler language.

Although careful inspection of the plots (including the extensive per-program data available at
`http://clip.dia.fi.upm.es/~jfran/tagschemes`) can be of help to draw some

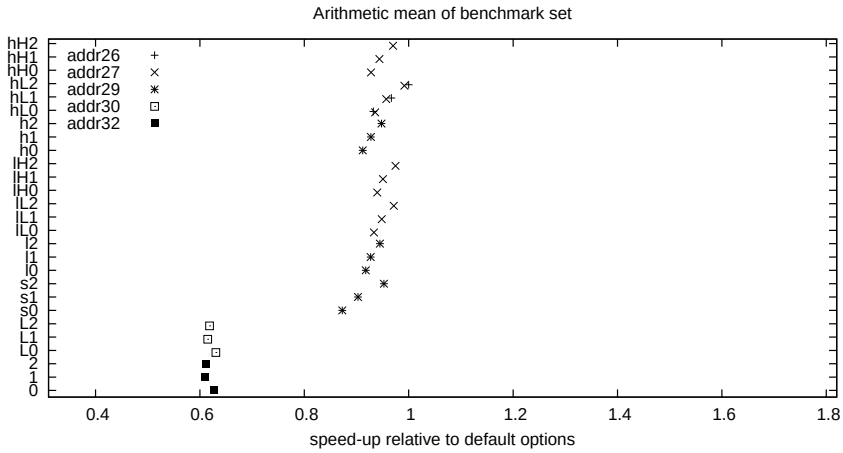

Figure 6.9: Arithmetic average of speedups (`coreduo-32`)

.

definite conclusion, resorting to visual inspection is too error-prone, so we performed an analytical study of the available data.

6.6.3 General-purpose Abstract Machine

Determining the combination of options which gives the best performance for a general-purpose abstract machine (where *general-purpose* is in our case reduced to the universe of programs in Table 6.1) is not easy: there is no absolute winner combination of tagging scheme / access primitive for any architecture / addressing range, and combinations which perform well (or very well) for some programs often perform badly (or very badly) for other programs.

Therefore, we decided to resort to a method based on *ballots* to decide the best option encoding / tag primitive implementation for each addressing space. In this setting, programs are voters, and the different combinations are candidates which voters *rank* using the runtime they offer for each voter. The idea is to select the candidate which is, in some sense, most preferred. Among the many ways to decide between candidates given this information, *Condorcet* methods are among the best known and preferred in many situations. We have selected the *Schulze* method [5] which it is used in other related areas and which determines a winner combination. This winner is removed from the list of candidates and the

[5] `http://m-schulze.webhop.net/schulze1.pdf`

algorithm is applied again, to generate a global preference list, its last member being the the worst combination – the *loser*.[6]

The reason not to simply select the combination with the best average is that extreme cases can affect this average more than what is desirable. Removing the best and worst combination (to exclude extreme elements) was an option we did not want to take, because we wanted to take into account all the available information.

The results of this process are summarized in Tables 6.6 (without GC time) and 6.7 (including GC time), where for each architecture and address space, the best and worst tag scheme combinations are shown, together with the average speedup w.r.t. a default combination. For each of these, the best and worst speedup in the benchmarks is shown, as well as the ratio (W/L) between these speedups. This ratio gives a raw idea of what variation can be expected for each address space and architecture combination.

Let us note first that comparing speedups in different architectures is not really needed (or even meaningful). If a single tagging scheme / address space option were the winner hands down, then a single answer (a single code scheme) could be given for a machine, but a per-architecture best option can typically be decided at compile time with adequate macro definitions. On the other hand, comparing the best schemes in each of the architectures may give some insight on which code schemes are favored by the C compiler, optimizer, and processor, which may guide future decisions.

Selecting between tag and GC bit placement and code for tag management operations is somewhat independent from the addressing scheme selected. On one hand, schemes which need native 64 bits (e.g, `addr59` and `addr61`) are only applicable to 64-bit machines, and therefore cannot be compared with those designed to be used in 32-bit machines. Additionally, if some tag scheme A gives better performance, but less effective address space than some other tag scheme B, then a decision should be made depending on the expected needs of speed vs. runtime memory needs.

In order to decide which schemes are the best, we first observe that the options favored by the Schulze method are the same regardless of whether GC time is

[6]The Schulze method may end up in a draw, in which case we select the combination with the best average speedup to be the winner.

taken into account or not in all cases but one: for the `p4-64` architecture and the `addr29` tag scheme in tables 6.6 and 6.7, the winners are, respectively, `h2` and `h1`. However, these two code generation options gave almost identical speedup results, so the difference can be safely ignored, and we can, therefore, focus on just one of the speedup tables.

Options for Native 64-bit Implementations: In the two 64-bit architectures the native 64-bit tagging scheme is reasonably fast w.r.t. the default 32-bit one, and both (`addr59` and `addr61`) give a similar speedup. Therefore, and looking at the actual memory consumption figures in Table 6.4, we select `addr59`, since it would make better use of the really available memory in the machine. In that case, GC bits in the higher part (`H`) is the recommended option, while tag bits are better placed in the lower (`l`) part for `sparc64` and in the higher (`h`) part for `p4-64`. In general, predefined switch rules (`2`) gave the best results, except in the scheme `addr95` in `p4-64`.

Options for 32-bit Architectures: Architectures with only 32-bit addresses (`xeon-32` and `coreduo-32`, in our case), can take advantage of the full address range by using two 32-bit words (i.e., schemes `addr30` and `addr32`). This, however, comes at the cost of a noticeable slowdown. In return, a larger set of objects can be kept in memory. Between these two schemes, and if memory usage is a concern, we would select `addr30`. If speed is a primary concern, then the default setup (`addr26`) gives the best results. However, for a not very large price in speed (especially in `coreduo-32` machines), the number of objects in memory can be increased almost eightfold. For memory-demanding applications (which may even not run in the memory available with `addr26`) this would obviously be an advantage and the results suggest using `addr29` in these cases. For the selected case, the best performance is obtained by using external GC bits and split tag bits (`s`).

Options for 32-bit Tagged Words in 64-bit Machines: Although the natural option for a 64-bit machine would be a 64-bit native implementation, there are cases where this may not be completely advantageous. The space that a 64-bit tagged word takes up is 8 bytes, and if a 64-bit machine has less than 4Gb of memory, as is often the case currently on, e.g., normal desktops and laptops, more 32-bit-based objects can fit in it without leaving any memory unreachable. In this case, again, the best options in terms of addressing space, `addr30` and `addr32`

pay a high price in speed. A better compromise uses `addr29`, as it provides good speed and a reasonably large memory address space. For the selected case, best options are similar to those for 64-bit in the same machines: tag bits are better placed in the lower (`l`) part for `sparc64` and in the higher (`h`) part for `p4-64`.

Impact of External GC Bits on Performance: Turning on external GC bit support (moving the GC bits out of the word) increases addressable space, at the expense of a somewhat more complex access to the GC bits and, perhaps more importantly, less cache locality, which however affects execution only when GC is performed. This is the reason why speed-ups are more modest when GC is turned on in the combinations where GC bits are external to the tagged word (`addr29`, `addr32`, `addr61`). Its impact is however not dramatically large, and can be accepted in exchange for the increased address space if it is really needed.

6.6.4 Per-program Abstract Machines

Our framework makes it possible to generate an executable which contains bytecode, native code, and an abstract machine specialized for a given Prolog program, using any of the generation options we have discussed so far. It is, thus, natural, to generate program-specific abstract machines and measure their performance. Table 6.8 shows the results of this experiment. Each case was obtained by finding the best combination of tag/GC placement options for each benchmark, then averaging over the benchmarks.

A comparison of the speedups in Table 6.6 and 6.7 with Table 6.8 shows that program-specific abstract machines have, on average, better performance than an "agreed" single abstract machine. The difference is not very big and, for general usage, a single abstract machine with the right selection of options (see Section 6.6.3) is probably enough. However, for some benchmarks the best abstract machine is more than twice as fast as the worst one (e.g., `addr27` in `xeon-32`, column "W/L Sabs"), and in some other benchmarks they difference reaches the 38% (again, `xeon-32` but in the `addr26` row in the "Sabs" column).

6.7 Final Remarks

It is interesting to observe that the 32 and 64 bit cases have a close performance, but of course there is a huge difference in address space (and a noticeable increase in memory consumption). Double word representations, which are easier to implement, have been observed to be in general slower. Thus, for languages or applications with a reduced set (around 8) of types of small objects requiring runtime type information it is clearly worth using in-word tags.

It is also interesting to note that with our approach we have been able to automatically generate code featuring all the optimizations present in a hand-written, highly-optimized abstract machine (the base case that we compare to) and we have also been able to obtain emulators with larger addressable space and/or better performance.

The results indicate that it is difficult to recommend a particular bit layout or tag switch implementation that will be ideal for all situations, since the best option set changes depending on the architecture and the memory requirements of the application. In this sense the results presented provide a recommendation table which indicates the best option for each situation (the results are of course limited to the architectures studied). In this sense, further interesting results of this work are a) that it is possible to construct a framework that allows a flexible and parametric implementation of tagged data which can be adapted to support different schemes with modest effort, and b) that the variability in results observed indicates that constructing such as framework is indeed useful not just for experimentation, but also for production since it allows generating the right machine for each architecture or even for each application.

The system in which these techniques have been implemented and wich which the experiments have been performed is part of the development branch of Ciao Prolog.

(tags,gcbits,swmode) and speedups for host **sparc64**											
Addr.	Winner				Loser				W/L		
	best	avg.	max	min.	**worst**	avg.	max	min.	avg.	max	min
addr26	hL2	1.00	1.00	1.00	hL1	0.98	1.00	0.94	1.02	1.06	1.00
addr27	lL2	1.04	1.33	1.00	hH1	0.97	1.00	0.90	1.07	1.47	1.00
addr29	l2	1.04	1.28	0.96	s0	0.95	1.00	0.89	1.09	1.31	1.00
addr30	2	0.69	0.96	0.59	1	0.68	0.96	0.58	1.02	1.05	0.99
addr32	2	0.69	0.96	0.60	1	0.68	0.96	0.58	1.02	1.05	0.99
addr59	lH2	1.00	1.35	0.86	hH0	0.96	1.35	0.83	1.04	1.11	1.00
addr61	l2	1.00	1.35	0.85	h0	0.96	1.35	0.83	1.05	1.11	1.00
(tags,gcbits,swmode) and speedups for host **p4-64**											
addr26	hL1	1.01	1.09	0.94	hL2	1.00	1.00	1.00	1.01	1.09	0.94
addr27	lL2	1.08	1.24	0.86	hL2	0.99	1.07	0.88	1.09	1.24	0.84
addr29	h2	1.08	1.25	0.86	s1	0.98	1.05	0.89	1.09	1.28	0.87
addr30	0	0.80	0.99	0.69	2	0.73	1.00	0.62	1.09	1.19	0.99
addr32	0	0.81	1.00	0.67	2	0.74	0.99	0.62	1.09	1.26	1.01
addr59	hH0	1.07	1.31	0.77	hL1	1.04	1.29	0.77	1.03	1.09	0.96
addr61	l2	1.08	1.31	0.79	l0	1.04	1.24	0.70	1.05	1.32	0.91
(tags,gcbits,swmode) and speedups for host **xeon-32**											
addr26	hL2	1.00	1.00	1.00	hL0	0.97	1.31	0.64	1.07	1.55	0.77
addr27	hL2	0.99	1.36	0.62	hL0	0.72	0.95	0.58	1.40	1.72	1.05
addr29	s2	0.92	1.21	0.65	h1	0.72	0.93	0.60	1.28	1.67	0.97
addr30	0	0.62	0.95	0.39	2	0.61	0.76	0.37	1.02	1.27	0.80
addr32	0	0.66	0.89	0.42	1	0.62	0.80	0.44	1.07	1.48	0.71
(tags,gcbits,swmode) and speedups for host **coreduo-32**											
addr26	hL2	1.00	1.00	1.00	hL0	0.94	1.01	0.70	1.07	1.44	0.99
addr27	hL2	0.99	1.02	0.90	lL0	0.95	1.17	0.70	1.05	1.42	0.85
addr29	s2	0.98	1.07	0.82	s0	0.90	1.00	0.69	1.10	1.18	1.00
addr30	0	0.64	0.98	0.51	1	0.62	0.97	0.51	1.03	1.12	0.97
addr32	0	0.64	0.99	0.55	1	0.62	0.98	0.52	1.03	1.11	0.99

Table 6.6: Schulze winner for each address space and machine (GC time not taken into account).

(tags,gcbits,swmode) and speedups for host **sparc64**											
Addr.	Winner				Loser				W/L		
	best	avg.	max	min.	**worst**	avg.	max	min.	avg.	max	min
addr26	hL2	1.00	1.00	1.00	hL0	0.98	1.02	0.94	1.02	1.07	0.98
addr27	lL2	1.03	1.22	0.98	hH1	0.97	1.00	0.89	1.06	1.37	1.00
addr29	l2	1.00	1.11	0.85	s0	0.93	0.99	0.79	1.09	1.23	1.01
addr30	2	0.69	0.97	0.59	1	0.67	0.97	0.58	1.02	1.05	0.99
addr32	2	0.68	0.96	0.57	1	0.66	0.96	0.56	1.02	1.05	0.99
addr59	lH2	0.99	1.36	0.85	hH0	0.95	1.35	0.82	1.04	1.10	1.00
addr61	l2	0.97	1.34	0.80	h0	0.93	1.33	0.80	1.04	1.11	1.00
(tags,gcbits,swmode) and speedups for host **p4-64**											
addr26	hL1	1.01	1.09	0.94	hL2	1.00	1.00	1.00	1.01	1.09	0.94
addr27	lL2	1.06	1.24	0.86	hH1	0.99	1.11	0.90	1.08	1.25	0.90
addr29	h1	1.04	1.26	0.74	s0	0.95	1.04	0.73	1.09	1.29	0.92
addr30	0	0.78	1.02	0.65	2	0.72	0.99	0.61	1.08	1.19	0.99
addr32	0	0.77	0.99	0.58	2	0.72	0.98	0.53	1.08	1.18	1.01
addr59	hH0	1.04	1.30	0.77	lH0	1.01	1.23	0.80	1.02	1.18	0.90
addr61	l2	1.03	1.30	0.70	l0	0.98	1.24	0.70	1.05	1.32	0.91
(tags,gcbits,swmode) and speedups for host **xeon-32**											
addr26	hL2	1.00	1.00	1.00	hL0	0.97	1.31	0.64	1.06	1.55	0.77
addr27	hL2	0.99	1.36	0.62	hL0	0.71	0.95	0.58	1.40	1.72	1.05
addr29	s2	0.89	1.18	0.61	h1	0.70	0.93	0.60	1.28	1.67	0.98
addr30	0	0.62	0.95	0.40	2	0.61	0.76	0.38	1.02	1.27	0.80
addr32	0	0.65	0.89	0.42	1	0.61	0.80	0.43	1.07	1.48	0.71
(tags,gcbits,swmode) and speedups for host **coreduo-32**											
addr26	hL2	1.00	1.00	1.00	hL0	0.93	0.99	0.70	1.08	1.42	1.01
addr27	hL2	0.99	1.02	0.90	lL0	0.93	1.07	0.70	1.07	1.40	0.94
addr29	s2	0.95	1.01	0.75	s0	0.87	0.98	0.68	1.09	1.18	1.01
addr30	0	0.63	0.97	0.51	1	0.61	0.96	0.51	1.03	1.12	0.97
addr32	0	0.63	0.95	0.52	1	0.61	0.93	0.52	1.03	1.12	0.99

Table 6.7: Schulze winner for each address space and machine (GC time included).

Addr.	Without GC						With GC					
	Sabs			W/L Sabs			Sabs			W/L Sabs		
	avg.	max	min	avg.	max	min	avg.	max	min	avg.	max	min
speedups for host **sparc64**												
addr26	1.00	1.03	1.00	1.03	1.07	1.00	1.00	1.02	1.00	1.03	1.07	1.00
addr27	1.04	1.33	1.00	1.08	1.47	1.00	1.03	1.22	1.00	1.07	1.37	1.01
addr29	1.04	1.29	0.99	1.10	1.40	1.00	1.01	1.11	0.85	1.09	1.29	1.01
addr30	0.69	0.96	0.59	1.02	1.05	1.00	0.69	0.97	0.59	1.02	1.05	1.01
addr32	0.69	0.96	0.60	1.02	1.06	1.00	0.68	0.96	0.57	1.02	1.06	1.00
addr59	1.02	1.35	0.86	1.07	1.43	1.00	1.00	1.36	0.85	1.06	1.35	1.00
addr61	1.01	1.35	0.86	1.05	1.11	1.00	0.97	1.34	0.82	1.05	1.11	1.00
speedups for host **p4-64**												
addr26	1.03	1.13	1.00	1.05	1.13	1.01	1.02	1.13	1.00	1.05	1.13	1.01
addr27	1.12	1.29	1.01	1.18	1.41	1.03	1.11	1.29	1.01	1.17	1.41	1.03
addr29	1.12	1.29	1.01	1.17	1.32	1.02	1.08	1.29	0.78	1.16	1.32	1.02
addr30	0.80	1.01	0.69	1.10	1.19	1.01	0.78	1.02	0.65	1.09	1.19	1.01
addr32	0.81	1.00	0.67	1.10	1.26	1.01	0.78	0.99	0.58	1.09	1.18	1.01
addr59	1.14	1.33	0.83	1.15	1.40	1.04	1.09	1.32	0.81	1.14	1.38	1.04
addr61	1.11	1.31	0.81	1.12	1.33	1.02	1.06	1.30	0.71	1.11	1.33	1.02
speedups for host **xeon-32**												
addr26	1.08	1.38	1.00	1.24	1.56	1.00	1.08	1.38	1.00	1.23	1.56	1.00
addr27	1.03	1.36	0.77	1.54	2.13	1.27	1.02	1.36	0.77	1.53	2.13	1.25
addr29	1.03	1.23	0.75	1.48	1.74	1.12	1.00	1.23	0.73	1.47	1.74	1.11
addr30	0.69	0.99	0.39	1.20	1.58	1.03	0.69	0.99	0.40	1.19	1.58	1.03
addr32	0.73	0.89	0.45	1.27	1.51	1.04	0.71	0.89	0.45	1.27	1.51	1.04
speedups for host **coreduo-32**												
addr26	1.00	1.04	1.00	1.08	1.44	1.01	1.00	1.03	1.00	1.08	1.42	1.01
addr27	1.02	1.24	0.98	1.13	1.43	1.05	1.01	1.16	0.94	1.13	1.40	1.05
addr29	1.01	1.19	0.97	1.14	1.43	1.02	0.98	1.04	0.75	1.13	1.40	1.05
addr30	0.64	0.98	0.55	1.04	1.12	1.01	0.63	0.97	0.55	1.04	1.12	1.01
addr32	0.64	0.99	0.55	1.04	1.13	1.00	0.63	0.95	0.53	1.04	1.13	1.00

Table 6.8: Speedup results using the best options for each benchmark.

7

A Case Study: Real-Time Sound Processing

Summary

In this chapter we study, through a concrete case, the feasibility of using a high-level, general-purpose logic language in the design and implementation of applications targeting wearable computers. The case study is a "sound spatializer" which, given real-time signals for monaural audio and heading, generates stereo sound which appears to come from a position in space. The use of advanced compile-time transformations and optimizations made it possible to execute code written in a clear style without efficiency or architectural concerns on the target device, while meeting strict existing time and memory constraints. The final executable compares favorably with a similar implementation written in C. We believe that this case is representative of a wider class of common pervasive computing applications, and that the techniques we show here can be put to good use in a range of scenarios. This points to the possibility of applying high-level languages, with their associated flexibility, conciseness, ability to be automatically parallelized, sophisticated compile-time tools for analysis and verification, etc., to the embedded systems field without paying an unnecessary performance penalty.

7.1 Introduction

In recent years software has become truly ubiquitous: a large part of the functionality of many devices is now provided by an *embedded* program, which often

implements the core tasks that such devices perform. This includes from simple timers in ovens or fuzzy logic based monitoring and control software in household appliances, to sophisticated real-time concurrent systems in cars and cell phones. Upcoming *wearable computing* applications envision an integration of such devices even into clothing.

A range of micro-controllers is available for these purposes which, when compared with the processors currently used in workstations or laptops, are much less expensive and consume a reduced amount of power (starting at micro-Watts for the simplest ones). In return such processors have limited memory (from hundreds of bytes to perhaps a few megabytes total) and speed (up to at most a few hundred megahertz clock rates, and with little or no instruction parallelism). Basically, lower clock rates consume less power and simpler processors with less storage are cheaper.

As a result of this, frequent requirements on *embedded programs* is that they be able to use minimum storage, execute few instructions, and meet strict timing constraints, since all this brings down both cost and power consumption. The importance of these requirements depends of course on the domain. Because of these requirements, programs are often developed in low-level languages including, in many cases, directly in assembler [Wol05]. Some of those programs are written on micro-controllers in order to completely minimize power consumption while others are written using also small, but more general-purpose computing platforms [MM03, MR00, RM02]. In most cases, platform limitations drive the whole development cycle, diverting attention from modularity, reusability, code maintainability, etc.

At the same time, and despite resource and program development technology constraints, the functionality implemented by embedded systems is often quite sophisticated. This can include, even for the smallest devices, non-trivial matrix operations (as in, e.g., Kalman filters [WB95], used in GPS receivers), or intensive, real-time operations on data streams (including spatialization, as in the digital sound processing example that we will study in this chapter). In addition, more sophisticated functionality and more automated operation is always demanded by users. Furthermore, those systems often face strict correctness requirements because of the nature of the application or simply because of the higher cost of fixing bugs once the system is deployed. In practice, and in order to deal with

these conflicting requirements, applications are often coded also in a high-level or specification language which is used for prototyping and verification, in addition to the above mentioned low-level language, which constitutes the implementation. Unfortunately, often no real link between these two codings of the problem exists.

Coding at a Higher Level

A number of recent proposals make it easier to code stream processing routines. They are usually based on connecting processing blocks (available in a library or provided by the user) using a textual programming language (e.g. [TKA02]) or visual depictions thereof (e.g. [Lee03]). In many cases their abstraction level is adequate to use them as specification languages, but code generation is sometimes not automatic, or the resulting code needs to be fine-tuned by hand. Data and control models are often that of a procedural / O.O. language, which makes the application of some program analysis and transformation techniques somewhat challenged. Domain-specific program transformation techniques exist, but they have only a limited use in the case of a general embedded system. We want to note that defining processing blocks and applying domain-specific transformations is in principle possible for languages of any type and level.

In contrast, the availability of optimizing technology for high-level languages makes their direct use to implement (and not just to specify and to prototype) an attractive alternative. First, using high-level languages makes it easier to write better programs, with fewer errors, in less time, and with less effort. Problems can be formulated at a higher level of abstraction and much of the low-level detail that must be dealt with when using, e.g., C or assembler (such as manual memory management, ensuring safe typing, complex data structure management, etc.), which complicate and obfuscate the coding of algorithms, are taken care of automatically. These languages also make it easier to detect any remaining bugs and also to verify the correctness of programs automatically. Finally, high-level languages are also useful in the context of the general trend in processor design towards multi-core chips. Dual processor designs (with four threads total) are present already in mainstream laptops and the expectations are to double the number of cores and threads every two years at fixed cost. Since the motivation behind these multi-core designs is precisely to gain performance while keeping resource consumption down, this trend is also likely to hit the micro-controller

arena. Parallelized programs will be required to exploit the performance that the chip can deliver, and the parallelization task will add to the burden on the programmer. High-level languages are relevant in this context because they have been shown to be easier to parallelize automatically [GPA+01].

The challenge in using high-level languages in embedded and wearable devices is to be able to generate automatically executables that are as efficient as required by the platform (with memory, speed, and energy consumption close to hand-coded low-level implementations). A particular challenge is to achieve this even if numeric or data-intensive computations are involved. While some interesting work has been done regarding the use functional programs in embedded systems [PHE99, Wal95], the use of (constraint) logic programming (CLP) systems in this context has received comparatively little attention. CLP, and, in particular, the availability of logical variables, search, and constraints in a programming language can be attractive because these features can make it easier to provide sophisticated problem solving, optimization, and reasoning capabilities in devices. This is in line with the demands for higher and more automated functionality from users. The purpose of this chapter is to investigate for a particular case study (a sound spatializer embedded in a wearable computer) the feasibility of coding it using a very high level, multiparadigm programming system supporting predicates, logical variables, dynamic typing, search, and constraints in combination with functions, higher order, objects, etc. (in particular, the Ciao system [BCC+09, HBC+99, HPBG05]).

However, the point of this work is not to use all these capabilities extensively[1] but instead to study whether current state of the art tools for compile-time analysis, verification, specialization, and low-level optimization are powerful enough to optimize away the default functionality available in such a rich language, including all its libraries, for a program such as the spatializer which only needs a fraction of them. This will require optimizing away all the overhead needed for supporting backtracking, full unification, tagged values, infinite precision arithmetic, etc., which are present by default in the language *for program sections that do not need these features* and see whether it is possible to produce in this way

[1] A brief account of how CLP characteristics can be of use to define and implement processes on streams appears in [Ste97]. Our work is complementary in that we do not deal with how to define and compose basic building blocks, but rather on how to optimize them.

executables for the wearable computer that are competitive in terms of speed, memory consumption, etc., when compared to a solution in a low-level language (in our case, C). This presents challenges that, while having some similarities, are also different for example from those which appear when optimizing programs written in other languages: dealing with logical variables and argument modes (i.e., procedure arguments are not known *a priori* to be input or output), dealing with backtracking and multiple solutions, eliminating dynamic typing (when compared to strongly typed languages), etc.

A Concrete Problem and its Motivation

The case study we chose is a stylized (but fully functional) version of a real wearable computing application (designed for the new Bristol CyberJacket) in which a set of virtual sounds are projected into a physical space. The user experiences these *soundscapes* through a set of headphones attached to the wearable computer (which has limited available power). An example of the use of such a *sound spatialization* device is a "talking museum" where any object, from the actual exhibits to the walls or doors of the rooms, can appear to be talking to the visitor. A compass is fixed on the user headphones which provides information on head orientation. The wearable computer is also aware of the user's location, through GPS for outdoor locations and through an ultrasonic positioning system [MM03] for indoor installations. With these two sources of information the wearable device can determine where a sound should be positioned relative to the user. By calculating the angle at which the sound is with respect to the head, the delay that the sound will experience at each ear can be calculated, and this allows spatializing the sound [Bla83]. For the sake of simplicity, and since we want to show actual code, we will present a version in which position is not dealt with, and only sound direction is taken into account.

This concrete case study was selected because of its characteristic nature: it requires core functionality present in many wearable computing applications. Handling streams of data such as audio and video and collections of positions is frequent in pervasive and wearable systems. In many common scenarios one or more sensors will produce data streams to be received and used by an actuator. These sensors can generate data at different, unrelated, but generally fixed, and sometimes very high, rates. Additionally, this case does not belong to the

Figure 7.1: Sound spatializer prototype, with Gumstix (bottom left) and compass (right) attached to headphone.

restricted class of synchronous systems and the operation (and, therefore, time) of some of the actuators depend on the particular data coming in. Therefore, this case study exemplifies a family of programs to which techniques similar to those we will show here can be applied. Very often (including, for example, our case) these problems have, in addition to resource constraints, hard real-time constraints where there are exact deadlines within the system. Of course the objective is to be able to support, in addition to such lower-level data integration tasks, higher-level functionality. But the point of the study is to see if the lower-level tasks can be handled efficiently enough, since the suitability of the programming language used for the higher-level tasks is taken for granted.

7.2 The Sound Spatializer

The problem we focus on is spatializing sound in real time by processing a monaural stream into a stereo one so that the sound appears to come from a position in space when played through a set of headphones. Angle information comes from

a compass mounted on the headphones. When the head turns, the compass will register a change in heading and the spatialization unit should change accordingly the direction from which the sound seems to originate to create the illusion that it remains fixed at a certain spacial point. Our fully functional prototype has a small processor board, compass, and battery, all integrated on a pair of headphones (see Figure 7.1). The sound stream is a series of samples (16-bit integers, coming either from some external source or from flash memory) and the compass data is read as floating-point numbers measuring the heading in degrees relative to North.

We will assume that signals are delivered at *a priori* known rates, and we will apply analysis and optimization tools in order to reduce the resources (processor cycles, mainly) needed to deliver sound in a timely manner. We want to, at least, be able to execute spatialization in real time on a small processor (described in Section 7.2.3). Any gains beyond real-time execution will allow us to lower the clock rate of the processor, which reduces power consumption, in turn increasing battery life. In the following subsections we will discuss the requirements in detail.

7.2.1 Sound Spatialization Basics

Figure 7.2 sketches how the ear localizes sound emanating from some point in space. When the head does not face the sound source, sound waves travel a different distance to each ear (D_L and D_R, respectively). Therefore the left and right ears receive the same sound with a slight phase shift, in the order of a millisecond, determined on the basis of the difference $D_L - D_R$. This enables the brain to determine the direction of the sound. Calculating that shift is the starting point for spatialization, as each earphone is to output one stream of sound samples, which is in turn a possibly delayed copy of the initial sound stream. The *absolute* distances to the sound sources can also be used to modify the volume of the sound, although in practice attenuation information is hardly used by the brain when determining the source of a sound. $D_L - D_R$ obviously depends on the angle C and the size of the head.

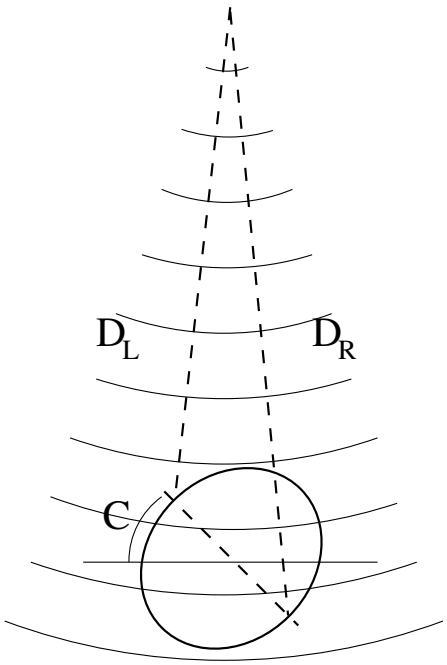

Figure 7.2: Sound samples reaching the ears.

7.2.2 Sound Quality and Spatial Localization

High sampling rates are needed to model small head movements. A relative displacement of 3.43 mm. corresponds to a difference of 10 μs, which needs a sampling rate of 100 KHz. The higher the sampling rate, the better the spatialization, but the more processing is required: there is a trade-off between quality of spatialization and processing power.

One of the requirements of the final application is that CD-quality sound has to be produced, i.e., 44,100 16-bit samples per second (sps), which can model a relative displacement between ears of about 7.5 mm. (a rotation of 2 degrees.) Our program should therefore be able to process 44,100 16-bit sps and deliver 88,200 16-bit sps (on the two output channels). Concurrently, data from the compass has to be read and used to produce the sound streams.

7.2.3 Hardware Characteristics of the Platform

The target architecture is modest in comparison with modern desktops or laptops: it is a Gumstix board equipped with a 200MHz XScale processor, 64Mb of RAM

and 4Mb of flash memory, which acts as permanent storage, and running a version or Linux (see `http://www.gumstix.com/` for more information). The Gumstix board is around 25 times slower (depending on the application, of course) than a 1.5GHz SpeedStep Centrino, at a fraction of the power usage. Memory and storage limitations are obviously significant and relevant to our application, since we want to run a non-terminating process, and thus garbage collection is critical.

7.2.4 Hard Real-time

A stereo sample should ideally be generated every 22 μs (1/44,100), as input samples arrive. In practice, sound buffers require blocks (typically of 256 samples) to be written to the sound card. Thus we are required to produce a block of sound samples every 6 ms. Two issues can prevent us from meeting this hard real-time deadline: process scheduling may swap out our program for more than 6 ms. and executions of the garbage collector could take more than that.

The former can be worked around, if necessary, by switching to some form of real-time Linux. However, if fewer processor cycles are needed by the application (our goal), it will be less likely to be affected by the O.S. scheduling. In our case this proved not to be a problem in the end.

The latter could hinder the use of high-level languages, as automatic memory management is one of the characteristics which makes them less error-prone. Some languages have undergone a careful and interesting revision of the memory management model and primitives in order to adapt to real-time requirements [BG00] (but, arguably losing part of the initial elegance). In our case recent work in this regard is aimed at inferring bounds on memory consumption at compile-time in order to guarantee compliance with memory constraints without modifying the language. In any case in our concrete case study the built-in memory management of the (Ciao) system proved sufficient to not cause any noticeable interruption in the sound while keeping memory consumption constant and within reasonable limits.

In both cases, increasing the sound card buffer size would give more freedom to the application. However, this creates a lag between the compass movements and the sound emission which would render the application unacceptably sluggish and destroy the illusion of spatialization.

7.2.5 Compass and Concurrency

Reading the compass data (which is done from a serial interface) may take an unknown amount of time, because due to limitations of the hardware data may be corrupted. In order not to block or obfuscate the rest of the application code, a separate thread is started which asynchronously reads data from the compass and posts it for the main program. Communication is performed via an atomically updatable, concurrent dynamic database [CH99]. This isolates low-level details of the compass from the rest of the program. However, it makes it necessary for the analysis tools to understand this communication by giving them an appropriate description. Ciao includes an assertion language which was used to annotate the interface to the compass (Section 7.4.2) appropriately.

Note that the scheduling is handled by the Gumstix operating system. In other scenarios, CLP-based tools have shown their usefulness at precomputing feasible schedulings using system specifications written as logic programs which are then automatically specialized to reduce or eliminate scheduling overhead [Leu01].

7.3 Program Code and Source-Level Transformations

7.3.1 Naive implementation

A naive implementation of the sound spatialization algorithm is shown in Figure 7.3. A function f takes the current samples of the sound stream and the direction stream, and produces a stereo sample. We encapsulate knowledge about when to skip samples and any history needed in a separate object "state," making f a pure function. The three stream objects all have preset periodicities and are initialized with their expected sampling rates.

This code is naive in that inside the function f one needs to perform trigonometric functions, but these only need to be executed once every compass poll (in our case, once every 4,410 sound samples instead of every sample). In general, a function f that operates on n inputs s_0, s_1, ..., s_{n-1} can be projected onto a series of functions f_0, ..., f_{n-1} such that

$$f(s_0, s_1, \ldots, s_{n-1}) = f_0(s_0, f_1(s_1, \ldots f_{n-1}(s_{n-1}) \ldots))$$

If the s_i are ordered according to their update rates so that s_0 has the fastest one

```
mono      := new InputPeriodicStream(sound_sps);
direction := new InputPeriodicStream(compass_sps);
stereo    := new OutputPeriodicStream(sound_sps);
while (true) DO
  state = f(mono.current(),
            direction.current(),
            state);
  stereo.output(state);
end
```

Figure 7.3: Single-loop algorithm for the spatializer.

```
mono      := new InputPeriodicStream(sound_sps);
direction := new InputPeriodicStream(compass_sps);
stereo    := new OutputPeriodicStream(sound_sps);

while (true) do
  state := f_c(direction.current(), mono.current(), state);
  samp_sound := sound_sps / compass_sps;

  while (samp_sound > 0) do
    state := f_m(mono.current(), state);
    stereo.output(state);
    samp_sound := samp_sound - 1;
  end
end
```

Figure 7.4: A nested-loop sound spatializer.

and s_{n-1} has the slowest one, the initial program can be rewritten to save the results of function applications by computing $f_{n-1}(s_{n-1})$ in the outer loop (with the lowest frequency) and proceeding inwards across nested loops until $f_0(s_0, \cdot)$ is computed in the innermost loop. Note that in our example code we only deal with the case in which the two frequencies divide each other. This is not the case for arbitrary sensors.

The code for the sound spatializer, according to this decomposition, is shown in Figure 7.4. The function f has been decomposed into f_m and f_c, and two loops have been created. The outer loop computes f_c when a new compass signal is available, whereas the inner loop applies f_m at a higher frequency. More efficiency is attained at the cost of a slightly more complex code (which has however a clear

```
spatialize(SamplesRemaining, SampleL, SampleR, CurrSkip):-
   new_sample_cycle(SamplesRemaining, NewCycle,
                    CurrSkip, NewSkip,
                    SampleL, SampleR,
                    NewSampleL, NewSampleR),
   new_sample(NewSampleR, R, RestSampleRight),
   new_sample(NewSampleL, L, RestSampleLeft),
   play_sample(R, L),
   spatialize(NewCycle, RestSampleLeft,
              RestSampleRight, NewSkip).

new_sample_cycle(0, ~audio_per_compass, CurrSkip,
                 NewSkip, SL, SR, NSL, NSR):-
   find_skip(~read_compass, NewSkip),
   skip(NewSkip - CurrSkip, SL, SR, NSL, NSR).
new_sample_cycle(Cycle, Cycle - 1,
                 Sk, Sk, SL, SR, SL, SR):- Cycle > 0.

new_sample([Sample|Rest], Sample, Rest):-
   var(Sample) -> read_sample(Sample) ; true.
```

Figure 7.5: Main loop for the sound spatializer reading from a compass.

structure) and the decomposition of f.

7.3.2 High-level Code for the Sound Spatializer

To go from the schematic code to a full implementation in a low-level imperative language requires quite a bit of coding where, e.g., memory management (allocation and management of buffers), data types and sizes, explicit synchronization, etc. need to be taken into account. Given our objectives, instead we wrote a complete sound spatializer in Ciao whose *actual* core code is shown in Figure 7.5 (we do leave out however for brevity some low-level details that deal with obtaining compass data and sending audio data, which were notwithstanding fully implemented in the code which was benchmarked in this chapter). Note that while the code has of course to deal with some low-level details, such as actually reading stream information and outputting sounds, there are many others (such as internal buffer information, types and precision of variables, etc.) which do not need to be explicitly stated.

A Note on Syntax: Ciao allows the use of functional notation with no execution time penalty [CCH06]. The prefix operator ~ enables the use of a predicate as a function by making its last argument correspond to the function result. Hence, the goal `?- append([1], [a], R).` can be written as `?- R = ~append([1], [a]).` Predicates can also be defined in functional syntax, by using `:=` instead of `:-` (Figure 7.6). This assumes that the last argument will represent the function result. Arithmetic expressions are also translated.

The sound stream is represented as an open-ended (incomplete), unbound-length list of samples (of some opaque type) which is incrementally instantiated as more samples are needed. This list is held in memory and the unnecessary items (the samples which have already reached the farthest ear and are unreachable in the program) are eventually and automatically deallocated.

On the other hand, the compass is explicitly polled (this is the functionality offered by the hardware) by a separate thread and communicated through the predicate `read_compass/1` which returns the latest read value. Based on it, `find_skip/2` determines the current difference (in number of samples) between the left and the right ear. This is used by `skip/6` which returns new sample lists (which are, at the virtual machine level, pointers to the initial, monaural sample list) for the left and right channels.

The code in Figure 7.6 represents physical units (such as the speed of sound in the air) and laws (e.g., the amount of space corresponding to every sample, depending on the sampling frequency) or parameters defining particular scenarios (such as the distance between ears).

We evaluated the different stages of optimization of the sound spatializer by processing a 120-second track while sampling the compass 10 times per second, using both the original version and an automatically specialized version (Section 7.3.4). Assessment is based on measuring the total processing time required and comparing it with the track duration, which indicates how well the bandwidth can be sustained by telling us how busy the processor is. We also recorded whether there were any artifacts such as clicks and silences. Their presence would reveal issues with garbage collection or swapping. The results are summarized in Table 7.1 where scenarios which generated acceptable sound are marked in boldface.

The code in Figures 7.5 and 7.6 can be compiled to bytecode and it can

```
sound_sps   := 44100.     % Samples per second
compass_sps := 10.        % Samples per second
sound_speed := 343.       % Meters
head_radius := 0.1.       % Meters
pi          := 3.141592.

audio_per_compass :=
    integer(~sound_sps / ~compass_sps).

samples_per_meter :=
   ~sound_sps / ~sound_speed.

ear_dif(Angle) :=
   ~head_radius * sin((Angle * ~pi) / 180).

find_skip(Angle) :=
    round(~samples_per_meter * 2 * ~ear_dif(Angle)).
```

Figure 7.6: Physical model in the sound spatializer.

Compilation mode	**Non-Specialized**			**Specialized**		
	i686	**Gumstix**		**i686**	**Gumstix**	
	secs.	secs.	Utilization	secs.	secs.	Utilization
Bytecode	**4.70**	115.95	96.6%	**3.91**	103.49	86.2%
Compiling to C	**3.87**	98.08	81.7%	**3.36**	**88.27**	73.6%
Id. + semidet	**3.28**	**92.42**	77.0%	**2.85**	**83.74**	69.8%
Id. + mode/type annotation	**3.00**	**88.38**	73.6%	**2.57**	**79.42**	66.2%
Id. + arithmetic	**2.90**	**85.70**	71.4%	**2.47**	**78.01**	65.0%

Table 7.1: Speed results and processor utilization for a benchmark with different compilation regimes.

deliver spatialized sound with the required quality in a modern desktop or laptop computer, while responding in real time to the signals received from a compass. However it falls short in our target platform: generating stereo samples for a 120-second track takes 115.95 seconds, which means the processor is busy 96.6% ($= \frac{115.95}{120} \times 100$) of the time (Table 7.1). The remaining processor time is not enough to cope with the rest of the O.S. tasks without introducing noticeable clicks. To improve this situation we take advantage of the amenability of high-level languages to advanced program analysis and transformation in order to

produce better executables without changing the original code. In particular we used **(i)** partial evaluation (to specialize parts of the program), **(ii)** abstract-interpretation based compile-time analysis to ensure that the program will not raise any run-time exceptions (due to illegal modes, types, etc.) and to extract information in order to **(iii)** perform optimizing compilation to native code (via C) using the information on modes, types, determinism, and non-failure gathered during analysis.

7.3.3 Compile-time Checking

The aim of compile-time checking is to guarantee statically that some program will satisfy certain *correctness criteria*, which in principle may be arbitrary. Static correctness proofs are certainly of utmost practical relevance in systems of high dependability or where updating the software is burdensome or costly. However, in most programming languages today the correctness criterion is type correctness, and compile-type checking boils down to type checking.

In the case of logic programs, arguments can in principle be input or output without further restrictions. This results in a very flexible programming language, where procedures are *reversible*. However, it is often the case that predefined (system) predicates require their arguments to satisfy certain calling conventions involving both types and modes (instantiation degree). Failing to satisfy such calling conventions is considered an error. For example, traditional Prolog systems check at run-time such calling conventions and errors are issued if the conventions are violated. In contrast to traditional CLP systems, in the Ciao analyzer and preprocessor, CiaoPP [HPBG05], information obtained by static analysis is used to reason about such calling conventions. To this end, the system has an assertion language [PBH00a] which allows explicitly and precisely stating calling conventions, i.e., preconditions for predicates. The Ciao system libraries are annotated to state pre- and post-conditions for library predicates. Several assertions expressing different pre-conditions and their associated post-conditions can co-exist for procedures which are multi-directional.

Static analysis in CiaoPP is based on abstract interpretation [CC77], and it is thus guaranteed to provide safe approximations of program behavior. Such safe approximations can be used in order to prove the absence of violations of a set of assertions, which can express more properties than just type coherence, and thus

the absence of run-time errors.

For example, in the case of our implementation of the stream interpreter, we use the system predicate `is/2`. The arithmetic library in Ciao contains an assertion of the form:

```
:- trust pred is(X,Y) : arithexpression(Y) => num(X).
```

which requires the second argument to `is/2` to be an arithmetic expression (which is a regular type also defined in the arithmetic library) *containing no unbound variables*, and also provides the information that on success the first argument will be instantiated to a number. Analysis information using the *eterms* [VB02] abstract domain allows CiaoPP to guarantee at compile time that the program satisfies the calling conventions for system predicates (in this example just `is/2`) used in the program. Thus, the compiler *certifies* that no run-time errors will be produced during the execution of our code for the stream interpreter. The same applies to other predicates which access external entities (e.g., compass data) and whose behavior was modeled using Ciao assertions (see Section 7.4.2).

The user may optionally provide assertions for his/her own procedures. If available, CiaoPP will try to check at compile time such assertions. Clearly, the more effort the user puts into writing assertions, the more guarantees we have of the program being correct.

7.3.4 Partially Evaluating the Program

The code in Figure 7.6 performs repeatedly the same set of operations, many of them involving constants. While the part of the main loop dealing with arithmetic is not called a large number of times (because of the low sampling rate of the compass), opportunities for partial evaluation to improve execution time certainly exist. Indeed, all the code in Figure 7.6 is reduced to a single clause:

```
find_skip(A,B) :-
        C is sin(A*0.017453288889),
        B is round(25.94117647058824*C) .
```

Moreover, the calculations involving constant numerical values are performed at compile-time and the results propagated to the appropriate places in the pro-

gram.[2] Loops and other parts of the program are also specialized, but the effect in those program points is less relevant. Input/output and other library built-ins are handled since they are appropriately annotated with assertions where they are defined.

Partial evaluation by itself gave, on average, speedups ranging from a factor of 1.15 to 1.2 on an i686 and around a factor of 1.1 on a Gumstix, when the compass is polled at 10Hz (see Table 7.1). On the Gumstix, partial evaluation decreases the processor utilization to 86.2% —substantially better than with the non-specialized code.

Although these results are encouraging, specialization by itself did not increase performance to a level where the spatializer really runs reliably in real-time on our target platform. Therefore, our next step towards gaining efficiency (and, as before, keeping the initial code untouched) was to optimize away the bytecode interpretation overhead by compiling the Ciao program into native code, using progressively more compile-time information in order to generate code as optimal as possible.

7.4 Compilation to Native Code

Two separate issues affect the performance of the sound spatializer: the time taken to process each sample, regardless of how it is processed, and the time taken to compute the new delay to be applied to the output streams. The former concerns mainly data-structure and control compilation (how the main loop is mapped into the lower-level language, how data structures are handled, and how data is read from and written to the streams). The latter is dominated fundamentally by costly (at least from the point of view of Ciao) floating-point arithmetic.

We attacked these problems by compiling to native code via C, using the schema presented in Chapter 3. As we also wanted to identify the impact of different technologies in the efficiency of the application, we proceeded stepwise:

[2]The reader may notice that C compilers also evaluate statically expressions containing constants. The situation is however different: in our case separate predicates (c.f., functions) are being evaluated statically guided by the calls made to them. If they were called from elsewhere in the program, the original definitions would have been kept together with the specialized versions.

we initially used only the information present explicitly in the original program, and later we used the extensive compile-time information gathered though global analysis.

7.4.1 Naive Compilation to Native Code

Compiling to native code without using information about types, modes, determinism, non-failure, etc. preserves exactly the data structures created when interpreting bytecode. Memory usage, existence (or not) of choice points, etc. do not change either, so any improvements in performance come mainly from reducing the time used in instruction fetching within the main virtual machine loop. Better data locality can help, but access patterns are difficult to predict and therefore this cannot usually be trusted as a source of improvement.

Despite the limited speedup that is obtained in the absence of additional information (Table 7.1), this was actually a turning point in our case: the processor utilization in the Gumstix decreased to 81.7% for the non-specialized program and to 73.6% for the partially evaluated version. The performance of the former is not enough to give a smooth playback; however, the latter is fast enough to play and to poll the compass at an adequate pace, while supporting some minimal additional load on the host processor. It is however not a satisfactory solution yet, as it was easy to produce noticeable interruptions in the playback just by adding a light load on the Gumstix.

7.4.2 Optimized Compilation

One of the tasks that non statically-typed languages have to perform at runtime is checking types and, for a logic-based language, also modes. Note that, unlike other declarative languages such as Mercury [SHC96] or Haskell [Jon03], Ciao programs do not need to include any type, mode, determinism, or non-failure declarations. Mode and determinism annotations are not needed in functional languages because all functions produce a single solution and their arguments are input.

Analysis information can be used to optimize native code generation in several points. For example, type information can be used to choose a more efficient, closer to the machine, representation. If mode information is also available,

```
:- true pred new_sample_cycle(A,B,C,D,E,F,G,H)
    :  (int(A), term(B), int(C), term(D),
        term(E), term(F), rt2(G),  rt2(H))
    => (int(A),  int(B), int(C), int(D),
        rt2(E),  rt2(F), rt2(G), rt2(H))
    +  (is_det, mut_exclusive).

new_sample_cycle(0,4410,C,D,E,F,G,H) :-
        find_skip(~read_compass,D),
        skip(D-C,E,F,G,H).
new_sample_cycle(A,A-1,C,C,E,F,E,F) :- A > 0.

:- regtype rt2/1.

rt2([A|B]) :- term(A), term(B) .
```

Figure 7.7: Part of the information inferred for the compass program.

the overhead involved in parameter passing and unification can be reduced by, e.g., compiling the latter into simple low-level assignments, perhaps with trailing. Last, determinism and non-failure information make it possible to reduce or avoid the creation of choicepoints since the compiler can know beforehand that no backtracking will be performed. This is, of course, only a partial list.

The analyzer we used (CiaoPP) is able to infer automatically a significant amount of information, provided that the *boundaries* of the program are well defined. For example, when there is communication with the outside world and the type of incoming data is relevant, then this data has to be described (via assertions in our framework). In our case study the only external data we need to deal with is that coming from the compass, since the sound samples themselves are treated as opaque data. Data coming from the compass is always a floating-point number. To reflect this, we added the following assertion for the `read_compass/1` predicate

```
:- trust pred read_compass(X) : var(X) => flt(X).
```

to the module encapsulating the compass access. This assertion should be read as: *"in any call to `read_compass/1`, the argument should be free when calling the predicate and it will be instantiated to a floating-point number upon success."* No other information is needed to infer accurate information regarding all the types,

modes, and determinism of the whole program. However, if this information is not provided little useful information can be inferred and most of the improvements that will be described in the following sections cannot be achieved. We want to note that in bigger, modular applications, boundary information is usually provided as part of the module interfaces (and it may have been automatically inferred), or it can be generated if all source code, libraries included, is available.

Figure 7.7 shows a selection of the information CiaoPP can deduce for the predicate `new_sample_cycle/8`. Much more information on sharing (pointer aliasing) and freeness (pointer initialization) was produced, which we omit since it is not instrumental for our case. However, it would be vital if we were to parallelize the code automatically.

`read_compass/1`, as we discussed previously, performs communication with the concurrent process that reads the compass, and its behavior is modeled with the assertion previously shown. With this information, the predicate `new_sample_cycle/8` is inferred to be deterministic and the clauses are found to be mutually exclusive (as expressed by the `(is_det, mut_exclusive)` assertion). This means that a more efficient compilation scheme, which does not produce superfluous code to handle backtracking, can be used.

Additionally, the open-ended list used to hold the samples to output is approximated with the type `rt2/1`, which only states that the argument is a *cons* cell. This information, albeit not complete, is enough for a lower-level compiler to generate better code which avoids testing at runtime the type of a parameter.

If determinism and non-failure inference are used, the processor utilization is reduced to 77% (for the non-specialized program, which is now able to generate stereo samples and poll the compass simultaneously with quite acceptable sound) and to 69.8% (for the specialized version). If mode (variable instantiation state at predicate entry and exit) and type inference are also used, the processor utilization gets further reduced to 73.6% and 66.2% for the non-specialized and specialized programs, respectively.

Optimizing Arithmetic Operations The unboxing optimization described in Section 3.4 presents another opportunity to reduce processor utilization. As before, this optimization was applied both to the non-specialized and to the specialized Ciao program, leading to some performance gains: the unboxing op-

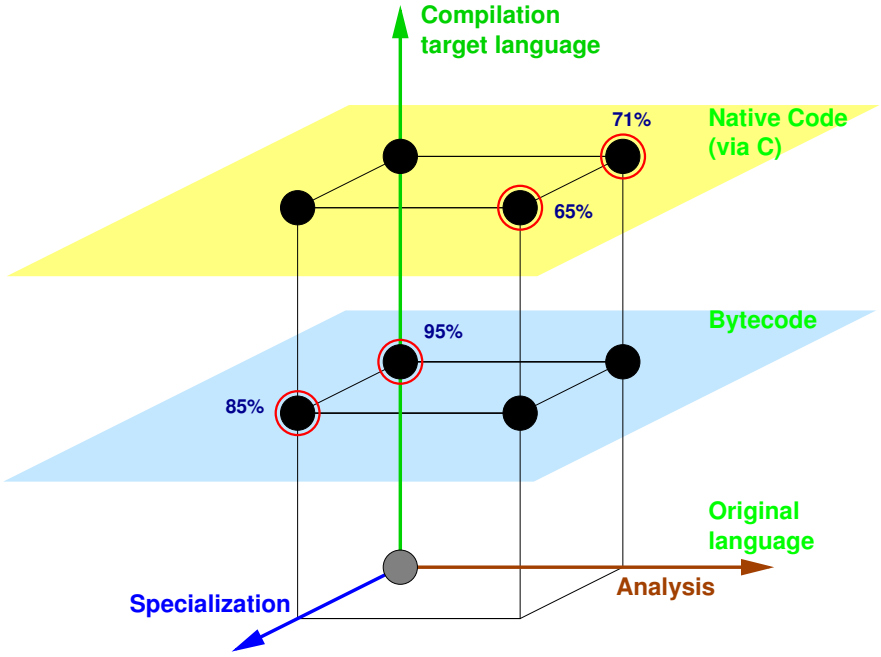

Figure 7.8: Global view of the experiments.

timization made it possible to reduce processor utilization to 71.4% for the non-specialized program and to 65% when running the specialized one. In both cases this is enough for the Gumstix to respond adequately to compass movements, even if there are several other (non CPU-bound) processes running on it.

7.5 Summary of the Experiments

Although we already presented some results in the previous sections, we will summarize our experiments and put them in the light of a new scenario we did not discuss before in order to make the presentation as clear as possible. A rough classification of the experiments performed, the processor utilization, and a pictorial summary of their characteristics, is shown in Figure 7.8.

7.5.1 Basic Results

All tests were run on a Gumstix, as commented throughout this chapter, and on a SpeedStep Centrino @ 1.4GHz. Table 7.1 shows performance figures for both.

Compilation mode	Non-Spec.	Specialized
Bytecode	25.64	14.00
Compiling to C	21.59	11.99
Id. + semidet	19.59	11.53
Id. + modes/types	19.19	11.08
Id. + arithmetic	6.97	3.62

Table 7.2: Results with a higher compass polling rate.

While the input stream is not infinite, after a few seconds both CPU usage and memory consumption stabilize, which makes us confident that the program would be able to run indefinitely.

The original non-specialized program running on a virtual machine is fairly efficient, especially taking into account that it is written in a style which is very close to a specification: buffer sizes are not stated anywhere (they self-adjust dynamically), memory management is automatic, etc. But there is not enough spare time to produce a sustained high quality sound stream on a Gumstix. A combination of specialization plus compilation to C, or compilation to C plus compile-time information, is enough to make the program deliver acceptable sound. However, the CPU usage in the Gumstix is still too high and any other activity on the same board causes audible interferences. It is only when both specialization plus analysis information are used to compile to C that other processes can be supported on the same board without noticeable interferences.

The best version runs, on the Gumstix, 1.5 times faster than the initial one. The difference is larger for the i686 case, as the speedup is around 1.9. However, those speedups also depend on particular scenario characteristics, such as polling frequencies, and, as we will see, other scenarios can exhibit very different behaviors.

7.5.2 Increasing the Sampling Frequency

The optimizations on arithmetic operations affect mainly a tiny fragment of code which computes the phase shift between the two ears and which is executed infrequently (10 times per second) with the current compass hardware. A faster poll rate, or the need to process other signals coming at a higher frequency would

require a larger fraction of processing time to be spent on computing the heading data.

To set up an extreme situation, we have simulated the case where heading data is provided at the same rate as the audio data (44,100 Hz). Note that this is the highest polling rate which makes sense, since a faster rate would actually discard compass data until the next audio sample is available. Table 7.2 summarizes the results under that assumption for an i686. In that scenario we measured a 7-fold speedup between the slowest and the fastest executable. This is indeed a very good result, and an extrapolation to the Gumstix suggests that with our current analysis and compilation technology the software running on the Gumstix would be very close to supporting compass sampling at 22,050 Hz.

The improvement introduced by using unboxed data and by specializing the program is much higher than in the previous set of tests. The reason is the same for both cases: more time is comparatively spent on arithmetic operations. Therefore, compile-time specialization, which evaluates many floating-point operations at compile time, simplifies fragments of code whose execution would take a substantial portion of the execution time (compare the left and right columns in Table 7.2). Something similar happens with the low-level optimization of floating-point arithmetic: operations are not removed, but they become much cheaper instead (last and next-to-last rows in Table 7.2)

7.5.3 A Comparison with C

We wanted to determine how far we are from an implementation written directly in C. We wrote a C program which mimics the Ciao one in the sense that it offers the same flexibility: it uses dynamic memory, buffer size is not statically determined, etc. It was written by an experienced C programmer and it does not incur any unnecessary overheads. The results are highly encouraging: the C program was only between 20% (for the tests in Table 7.2) to 40% faster (for the tests in Table 7.1) on an i686 processor. Interestingly, this C program did not behave as smoothly as expected when executed on the Gumstix: memory management caused audible clicks, and writing an *ad-hoc* memory manager would probably have been needed — or sacrificing flexibility by using static data structures. Additionally, the complexity of the C code would have made tuning the application much more difficult.

7.6 Conclusions

In this chapter we have shown how a set of advanced analysis, transformation, and compilation tools can be applied to a program written in a high-level CLP language which deals with a combination of numerical and symbolic processing (in the form of data structures) to generate an executable which runs adequately in terms of time, memory, and feedback to the user on a pervasive computing platform. We believe that the techniques we show here can be effectively used in a broader set of scenarios.

The application we used is a sound spatializer, intended to run on a wearable computer, the Bristol "CyberJacket". There were hard requirements regarding timing, sound quality, and non-functional behavior. The application code was deliberately not "tricky", but clear and as declarative as possible; it was not changed or adapted (by hand) in any of the experiments. The initial executions (using a bytecode interpreter in the wearable computer) did not meet the stated requirements, but a series of analysis, specialization, and optimizing compilation stages, which we reported on, managed to make it run well within spec on the target machine. All of them were carried on using the Ciao/CiaoPP programming environment. In an alternative, more demanding scenario, needing more arithmetic operations, our code performs within 20%-40% of a comparable C program.

It is difficult to single out a compilation stage which can be attributed the majority of the benefits. In the first (non arithmetic intensive) scenario, specialization caused most of the speedup because of the reduction in the number of arithmetic operations and calls performed. However, in the second scenario, boxing / unboxing removal was the clear winner. The rest of the optimizations were not highly relevant in this case, but we believe they would have been if more symbolic processing were needed. In any case, the information gathered by the analysis was also used by the low-level optimizing compiler.

8

Conclusions and Future Work

Summary

This chapter summarizes the conclusions from previous chapters, and shows the relationship of what has been achieved with the thesis objectives. It also points to future work in the area of the thesis.

8.1 Conclusions

A motivation of this thesis has been the development of native compilation techniques for logic programming that could be combined with bytecode emulation, as well as a framework that simplifies the creation of virtual machines, both included in the development version of Ciao Prolog.

Among the most relevant conclusions from this work we can mention:

- We have developed a Prolog-to-C compiler that uses type analysis and determinacy information to improve code generation, and we have provided performance results.

  Annotated WAM-like code: The compiler to C code uses a simplified representation for WAM code that includes analysis information (using type and determinacy information inferred by means of abstract interpretation), which is then translated to a lower-level intermediate code.

Effective combination of source-level and low-level optimizations: We have shown with a realistic case (a real-time sound processing tool) (Chapter 7) that a combination of analysis, specialization, and optimizing compilation, carried out in the Ciao/CiaoPP enviroment, followed by optimized code generation, was able to produce code that performed within 20%-40% of a comparable C program, and which had a 7-fold speedup w.r.t. the program executed by a bytecode emulator.

- We have presented (Chapter 4) the design and implementation of an emulator compiler that generates efficient code using a high-level description of the instruction set of an abstract machine and a set of rules which define how intermediate code is to be represented as bytecode.

  Data and code representation abstractions: We proposed a separation of the emulator definition into distinct components. That is, the low-level data and bytecode representation, and the set of instructions and their corresponding semantics. By doing this separation and developing an automatic approach to combine them in a full-fledged emulator, we were able to perform, at the abstract machine description level, transformations which affect both the bytecode format and the instruction set.

  Emulator Minimization: As an initial application and example for the technique, we experimented with emulator minimization which, given a set of programs, can generate emulators where unusued instructions (and their implementations) have been completely removed from the instruction set.

  A framework for exploring emulator optimizations: We have studied how these combinations perform with a series of benchmarks in order to find, e.g., what is the "best" average solution and how independent coding rules affect the overall speed or other desired properties.

- Dealing with an abstraction for the actual semantics of the instructions in a way that they can be manipulated automatically in order to do program transformations on the instruction code themselves, is a harder problem.

The approach that has been taken in this thesis (Chapter 5) describes the semantics of instructions of a Prolog bytecode interpreter — the Ciao engine — and some data types (tagged words) necessary for the emulator (Chapter 6) in a specially designed language (imProlog) derived from Prolog that is semantically closer enough to it to reuse analysis, optimization, and compilation techniques, but which is at the same time designed to be translated into very efficient C code.

Better source quality: The approach based on program manipulation techniques, with optional program annotations, allowed for a cleaner implementation of the emulator, in comparison with the original macro-based C code: it contained less duplicated code structures and hand-made specializations. Combined with the emulator generator framework (Chapter 4) we were able to produce and test different emulator versions to which several combinations of code generation options were applied.

Reproducing hand-made optimizations: We were able to perform non-trivial transformations on both the emulator and the instructions: instruction merging, specialization, and many other optimizations that are typically implemented by hand in bytecode emulators.

Tag schemes: We have been able to automatically generate code featuring many optimizations present in a hand-written, highly-optimized abstract machine (the base case that we compare to) relating manipulation of tagged data types, studying the performance of many 32 and 64 bit cases.

Applying together many of the proposed techniques, we found several combinations that performed better than our initial emulator, and which had a larger address space.

8.2 Future Work

Among the work which this thesis leaves open for the future we want to cite the following.

Inter-procedural low-level optimizations: Most of the inter-procedural analysis results that are used in this thesis to perform optimizations is derived from source code properties. We intend to extend the analysis with lower-level properties: for example, those required to perform inter-procedural and inter-modular boxing and unboxing.

Memory optimizations: We also want to study compilation schemes aimed at saving memory space which, although not a problem in the cases we studied, can be a concern in other scenarios. For example, generating automatically "hints" for the garbage collector or compile-time garbage collection [MRJB01].

Back-ends: We want to study which other optimizations that can be added to the generation of C code without breaking its portability. We are also interested on using intermediate representation to generate code for other back-ends (for example, GCC RTL, CIL, Java bytecode, etc.).

Just-in-time (JIT) and hybrid approaches: It is certainly worth exploring the combination of bytecode and native code compilation, both statically (by selecting statically which predicates will be compiled to native code) and dynamically (by selecting and compiling the predicates during program execution, e.g., driven by profiling information).

Full imProlog: We are interested in the generation of efficient code for Prolog with imProlog features (like mutable states) for non-typical applications of logic programming where algorithms with state changes have a primary role. This work would combine the compilation techniques already put to work in the optimization of Prolog applications in pervasive computing (Chapter 7) and the generation of emulators code (Chapter 5).

The emulator compiler is a very flexible framework that can be used to perform extensive experimentation with variations of abstract machine instruction sets and bytecode representations. Some of the future lines of research that can be derived from this are the following:

Further emulator specialization: As an extension of the abstract machine minimization developed in Chapter 4, many other simplifications can be

performed, affecting the instruction set, the implementation of data structures, or the code for the built-ins. It could be possible to refine the ancillary machinery in order to generate fast and small executables from generic code by performing dead code elimination, slicing, and partial evaluation based on abstract representations of the input bytecode language.

Instruction merging and specialization: Instruction merging been explored in [NCS01] for a fixed set of benchmarks, but emulators were hand-coded, somewhat limiting the per-application use of this approach. A future line of work consits on using information from execution profiling in order to merge instructions which frequently appear contiguous, and specialize them with respect to often-used argument values, etc. We would also like to study the applicability of our scheme (or variations on it) on abstract machines which have a different design, such as, for example, those which feature stack-based parameter passing like the TOAM [Zho94].

Reduce the optimization search space: Expanding the number of transformations and optimizations would give a combinatorial explosion on the number of new instructions. In order to attack the problem, it would be necessary to prune the search space by, for example, detecting orthogonal parameters and maximizing them independently, and devising heuristics that explore the more promising combinations sooner.

Specially tuned-abstract machines: We found that the outcome of many optimizations is only posive or appreciable on some kind of problems and/or architectures. The automatic emulator generation is useful not just for experimentation, but also for generating the right emulator for each architecture or even for each application.

Bibliography

[ACHS88] K. Appleby, M. Carlsson, S. Haridi, and D. Sahlin, *Garbage Collection for Prolog Based on WAM*, Communications of the ACM **31** (1988), no. 6, 719–741.

[AK91] Hassan Ait-Kaci, *Warren's Abstract Machine, A Tutorial Reconstruction*, MIT Press, 1991.

[AKN88] H. Ait-Kaci and R. Nasr, *Integrating Data Type Inheritance into Logic Programming*, Data Types and Persistence (P. Atkinson, P. Buneman, and R. Morrison, eds.), Springer, Berlin, Heidelberg, 1988, pp. 121–136.

[AKP91] H. Aït-Kaci and A. Podelski, *Towards a Meaning of LIFE*, Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, Springer LNCS 528, 1991, pp. 255–274.

[App89] Andrew W. Appel, *Runtime Tags Aren't Necessary*, LISP and Symbolic Computation **2** (1989), no. 2, 153–162.

[BCC+02] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla, *The Ciao Prolog System. Reference Manual (v1.8)*, The Ciao System Documentation Series–TR CLIP4/2002.1, School of Computer Science, Technical University of Madrid (UPM), May 2002, System and on-line version of the manual available at `http://www.ciaohome.org`.

[BCC+09] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.), *The Ciao System. Ref. Manual (v1.13)*, Tech.

report, School of Computer Science, T.U. of Madrid (UPM), 2009, Available at `http://www.ciaohome.org`.

[BD95] Peter A. Bigot and Saumya K. Debray, *A Simple Approach to Supporting Untagged Objects in Dynamically Typed Languages*, International Logic Programming Symposium, 1995, pp. 257–271.

[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*, Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97 (Linköping, Sweden), U. of Linköping Press, May 1997, pp. 155–170.

[BG00] Greg Bollella and James Gosling, *The Real-Time Specification for Java*, Computer **33** (2000), no. 6, 47–54.

[Bla83] Jens Blauert, *Spatial hearing : The psychophysics of human sound localization*, The MIT Press, 1983.

[BM72] R. Boyer and J. More, *The sharing of structure in theorem-proving programs*, Machine Intelligence 7 (1972), 101–116.

[Bow83] D. Bowen et al., *A Portable Prolog Compiler*, Logic Programming Workshop '83, Universidade Nova de Lisboa, June 1983, pp. 74–83.

[Bru82] M. Bruynooghe, *The Memory Management of Prolog Implementations*, Logic Programming (K.L. Clark and S.-A. Tärnlund, eds.), Academic Press, 1982, pp. 83–98.

[BW88] Hans-Juergen Boehm and Mark Weiser, *Garbage collection in an uncooperative environment*, Softw. Pract. Exper. **18** (1988), no. 9, 807–820.

[Car89] M. Carlsson, *On the Efficiency of Optimizing Shallow Backtracking in Compiled Prolog*, Sixth International Conference on Logic Programming (G. Levi and M. Martelli, eds.), MIT Press, June 1989, pp. 3–16.

[Car91] Mats Carlsson, *The SICStus emulator*, Tech. Report T91:15, Swedish Institute of Computer Science, 1991.

[CC77] P. Cousot and R. Cousot, *Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, POPL'77, ACM, 1977, pp. 238–252.

[CC01] Luís Fernando Castro and Vítor Santos Costa, *Understanding Memory Management in Prolog Systems*, Proceedings of the 17th International Conference on Logic Programming (London, UK), Springer-Verlag, 2001, pp. 11–26.

[CCH06] A. Casas, D. Cabeza, and M. Hermenegildo, *A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems*, The 8th International Symposium on Functional and Logic Programming (FLOPS'06) (Fuji Susono (Japan)), April 2006, pp. 142–162.

[CD95] Philippe Codognet and Daniel Diaz, *WAMCC: Compiling Prolog to C*, International Conference on Logic Programming (Leon Sterling, ed.), MIT PRess, June 1995, pp. 317–331.

[CF91] Robert Cartwright and Mike Fagan, *Soft Typing*, Programming Language Design and Implementation (PLDI 1991), SIGPLAN, ACM, 1991, pp. 278–292.

[CH99] M. Carro and M. Hermenegildo, *Concurrency in Prolog Using Threads and a Shared Database*, 1999 International Conference on Logic Programming, MIT Press, Cambridge, MA, USA, November 1999, pp. 320–334.

[CH00] D. Cabeza and M. Hermenegildo, *A New Module System for Prolog*, International Conference on Computational Logic, CL2000, LNAI, no. 1861, Springer-Verlag, July 2000, pp. 131–148.

[CMM+06] M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo, *High-Level Languages for Small Devices: A Case Study*, Compilers,

Architecture, and Synthesis for Embedded Systems (Krisztian Flautner and Taewhan Kim, eds.), ACM Press / Sheridan, October 2006, pp. 271–281.

[Col93] A. Colmerauer, *The Birth of Prolog*, Second History of Programming Languages Conference, ACM SIGPLAN Notices, March 1993, pp. 37–52.

[CZ07] Cuny Cs and Neng-Fa Zhou, *A Register-Free Abstract Prolog Machine with Jumbo Instructions*, International Conference on Logic Programming, 2007.

[DC01] Daniel Diaz and Philippe Codognet, *Design and Implementation of the GNU Prolog System*, Journal of Functional and Logic Programming **2001** (2001), no. 6, 2001.

[Deb92] Saumya K. Debray, *A Simple Code Improvement Scheme for Prolog*, Journal of Logic Programming **13** (1992), 57–88.

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

[DM82] L. Damas and R. Milner, *Principal type-schemes for functional programs*, Proc. 9th Annual Symposium on Principles of Programming Languages, 1982, pp. 207–212.

[DM92] Bart Demoen and André Mariën, *Can Prolog Execute as Fast as Aquarius*, Tech. Report CW144, Department of Computer Science, K.U.Leuven, 1992.

[DN00] Bart Demoen and Phuong-Lan Nguyen, *So Many WAM Variations, So Little Time*, Computational Logic 2000, Springer Verlag, July 2000, pp. 1240–1254.

[DN08] Bart Demoen and Phuong-Lan Nguyen, *Environment reuse in the wam*, ICLP '08: Proceedings of the 24th International Conference

on Logic Programming (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 698–702.

[DNSCS07] B. Demoen, P. Nguyen, V. Santos Costa, and Z. Somogyi, *Dealing with large predicates: Exo-compilation in the WAM and in Mercury*, Proceedings of CICLOPS 2007, 7th International Colloquium on Implementation of Constraint and LOgic Programming Systems (Porto, Portugal) (S. Abreu and V. Santos Costa, eds.), September 2007, pp. 117–131.

[DPP+97] Sean Dorward, Rob Pike, David Leo Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom, *Inferno*, 42nd IEEE International Computer Conference, IEEE, 1997.

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

[ECR93] ECRC, *Eclipse user's guide*, European Computer Research Center, 1993.

[FD99] M. Ferreira and L. Damas, *Multiple Specialization of WAM Code*, Practical Aspects of Declarative Languages, LNCS, no. 1551, Springer, January 1999.

[FD02] Michel Ferreira and Luís Damas, *WAM Local Analysis*, Proceedings of CICLOPS 2002 (Copenhagen, Denmark) (Bart Demoen, ed.), Department of Computer Science, Katholieke Universiteit Leuven, June 2002, pp. 13–25.

[Fut71] Y. Futamura, *Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler*, Systems, Computers, Controls **2** (1971), no. 5, 45–50.

[GBD92] D. Gudeman, K. De Bosschere, and S.K. Debray, `jc`*: An Efficient and Portable Sequential Implementation of Janus*, Proc. of 1992 Joint International Conference and Symposium on Logic Programming, MIT Press, November 1992, pp. 399–413.

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

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *Java(tm) language specification, the (3rd edition)*, Addison-Wesley Professional, 2005.

[GKPC85] F. Giannesini, H. Kanoui, R. Pasero, and M. Van Caneghem, *Prolog*, InterEditions, 87 Avenue du Maine, 75014, Paris, 1985, ISBN 2-7296-0076-0.

[GPA+01] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo, *Parallel Execution of Prolog Programs: a Survey*, ACM Transactions on Programming Languages and Systems **23** (2001), no. 4, 472–602.

[Han91] John Hannan, *Staging Transformations for Abstract Machines*, Partial Evaluation and Semantics-Based Program Manipulation (PEPM), ACM SigPlan Notices, 1991.

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

[HBC+08] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla, *An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy*, Festschrift for Ugo Montanari (Pierpaolo Degano, Rocco De Nicola, and Jose Meseguer, eds.), LNCS, no. 5065, Springer-Verlag, June 2008, pp. 209–237.

[HCS95] F. Henderson, T. Conway, and Z. Somogyi, *Compiling Logic Programs to C Using GNU C as a Portable Assembler*, ILPS 1995 Post-

conference Workshop on Sequential Implementation Technologies for Logic Programming, December 1995, pp. 1–15.

[Hen02] Fergus Henderson, *Accurate garbage collection in an uncooperative environment*, ISMM '02: Proceedings of the 3rd international symposium on Memory management (New York, NY, USA), ACM, 2002, pp. 150–156.

[HF00] S. Haridi and N. Franzén, *The Oz Tutorial*, DFKI, February 2000, Available from http://www.mozart-oz.org.

[HG91] M. Hermenegildo and K. Greene, *The &-Prolog System: Exploiting Independent And-Parallelism*, New Generation Computing **9** (1991), no. 3,4, 233–257.

[HL94] P. Hill and J. Lloyd, *The Goedel Programming Language*, MIT Press, Cambridge MA, 1994.

[Hod90] Joshua S. Hodas, *Compiling Prolog - From the PLM to the WAM and Beyond*, 1990.

[Hol93] Bruce K. Holmer, *Automatic Design of Computer Instruction Sets*, Ph.D. thesis, University of California at Berkeley, 1993.

[HP00] Michael Hind and Anthony Pioli, *Which pointer analysis should I use?*, International Symposium on Software Testing and Analysis, 2000, pp. 113–123.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno, *Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging*, The Logic Programming Paradigm: a 25–Year Perspective (K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, eds.), Springer-Verlag, July 1999, pp. 161–192.

[HPBG05] M. Hermenegildo, G. Puebla, F. Bueno, and P. López García, *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*, Science of Computer Programming **58** (2005), no. 1–2, 115–140.

[HS02] F. Henderson and Z. Somogyi, *Compiling Mercury to High-Level C Code*, Proceedings of Compiler Construction 2002 (R. Nigel Horspool, ed.), LNCS, vol. 2304, Springer-Verlag, April 2002, pp. 197–212.

[HWD92] M. Hermenegildo, R. Warren, and S. K. Debray, *Global Flow Analysis as a Practical Compilation Tool*, Journal of Logic Programming **13** (1992), no. 4, 349–367.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall, New York, 1993.

[JM87] J. Jaffar and S. Michaylov, *Methodology and Implementation of a CLP System*, Fourth International Conference on Logic Programming, University of Melbourne, MIT Press, 1987, pp. 196–219.

[Jon03] Simon Peyton Jones (ed.), *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, 2003.

[JRR99] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig, *C--: A Portable Assembly Language that Supports Garbage Collection*, International Conference on Principles and Practice of Declarative Programming (Gopalan Nadathur, ed.), Lecture Notes in Computer Science, no. 1702, Springer Verlag, September 1999, pp. 1–28.

[JS86] U. Jørring and W.L. Scherlis, *Compilers and staging transformations*, Thirteenth ACM POPL, 1986, pp. 86–96.

[KB95] Andreas Krall and Thomas Berger, *The VAM_{AI} - an abstract machine for incremental global dataflow analysis of Prolog*, ICLP'95 Post-Conference Workshop on Abstract Interpretation of Logic Languages (Tokyo) (Maria Garcia de la Banda, Gerda Janssens, and Peter Stuckey, eds.), Science University of Tokyo, 1995, pp. 80–91.

[KNW90] Andreas Krall, Ulrich Neumerkel, and Technische Universität Wien, *The Vienna Abstract Machine*, In PLILP'90, LNCS, Springer, 1990, pp. 121–135.

[Kow79] R. Kowalski, *Algorithm = logic + control*, Communications of the ACM **22** (1979), no. 7, 424–436.

[Kra94] Andreas Krall, *Implementation Techniques for Prolog*, Proceedings of the Tenth Logic Programming Workshop, WLP 94, 1994, pp. 1–15.

[LA04] Chris Lattner and Vikram Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, CGO, 2004.

[Lee03] Edward A. Lee, *Overview of the Ptolemy Project*, Tech. Report UCB/ERL M03/25, University of California at Berkeley, July 2003.

[Ler92] Xavier Leroy, *Unboxed Objects and Polymorphic Typing*, Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Albequerque, New Mexico), 1992, pp. 177–188.

[Leu01] M. Leuschel, *Design and Implementation of the High-Level Specification Language CSP(LP)*, PADL'01 (I. V. Ramakrishnan, ed.), Lecture Notes in Computer Science, vol. 1990, Springer-Verlag, March 2001, p. 14.

[Llo94] John W. Lloyd, *Practical Advtanages of Declarative Programming*, GULP-PRODE (1), 1994, pp. 18–30.

[LSC94] R. Lopes and V. Santos Costa, *The YAIL: An Intermediate Language for the Native Compilation of Prolog Programs*, 3rd COMPULOG NET Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages (Bonn), September 1994.

[Mar93] André Mariën, *Improving the Compilation of Prolog in the Framework of the Warren Abstract Machine*, Ph.D. thesis, Katholieke Universiteit Leuven, September 1993.

[MCH04] J. Morales, M. Carro, and M. Hermenegildo, *Improving the Compilation of Prolog to C Using Moded Types and Determinism Information*, Proceedings of the Sixth International Symposium on Practical

Aspects of Declarative Languages (Heidelberg, Germany), Lecture Notes in Computer Science, vol. 3057, Springer-Verlag, June 2004, pp. 86–103.

[MCH07] J.F. Morales, M. Carro, and M. Hermenegildo, *Towards Description and Optimization of Abstract Machines in an Extension of Prolog*, Logic-Based Program Synthesis and Transformation (LOPSTR'06) (Germán Puebla, ed.), LNCS, no. 4407, July 2007, pp. 77–93.

[MCH08] J. Morales, M. Carro, and M. Hermenegildo, *Comparing Tag Scheme Variations Using an Abstract Machine Generator*, 10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08), ACM Press, July 2008, pp. 32–43.

[MCH09] J.F. Morales, M. Carro, and M. Hermenegildo, *Description and Optimization of Abstract Machines in a Dialect of Prolog*, Technical Report CLIP4/2009.0, Technical University of Madrid (UPM), School of Computer Science, UPM, October 2009.

[MCH10] ———, *Description and Optimization of Abstract Machines in an Extension of Prolog*, Submitted to Theory and Practice of Logic Programming (2010).

[MCPH05] J. Morales, M. Carro, G. Puebla, and M. Hermenegildo, *A Generator of Efficient Abstract Machine Implementations and its Application to Emulator Minimization*, International Conference on Logic Programming (Maurizio Gabbrielli and Gopal Gupta, eds.), LNCS, no. 3668, Springer Verlag, October 2005, pp. 21–36.

[MD91] Andre Marien and Bart Demoen, *A New Scheme for Unification in WAM*, Proceedings of The International Symposium on Logic Programming (San Diego), October 1991, pp. 257–271.

[Mel82] C.S. Mellish, *An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter*, Logic Programming (K.L. Clark and S.-A. Tärnlund, eds.), Academic Press, 1982, pp. 99–106.

[MM03] Mike McCarthy and Henk Muller, *No Pingers: Ultrasonic Indoor Location Sensing without RF Synchonisation*, Tech. Report 003-004, University of Bristol, Department of Computer Science, May 2003.

[MR00] Henk Muller and Cliff Randell, *An Event-Driven Sensor Architecture for Low Power Wearables*, ICSE 2000, Workshop on Software Engineering for Wearable and Pervasive Computing, ACM/IEEE, June 2000, pp. 39–41.

[MRJB01] Nancy Mazur, Peter Ross, Gerda Janssens, and Maurice Bruynooghe, *Practical aspects for a working compile time garbage collection system for mercury*, Proceedings of the 17th International Conference on Logic Programming (London, UK), Springer-Verlag, 2001, pp. 105–119.

[MYJ07] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones, *Faster Laziness Using Dynamic Pointer Tagging*, ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ACM, 2007, pp. 277–288.

[NAJ+81] K. V. Nori, Urs Ammann, Kathleen Jensen, H. H. Nageli, and Christian Jacobi, *Pascal-p implementation notes*, Pascal - The Language and its Implementation (D. W. Barron, ed.), John Wiley, 1981, pp. 125–170.

[NCS01] H. Nässén, M. Carlsson, and K. Sagonas, *Instruction Merging and Specialization in the SICStus Prolog Virtual Machine*, Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM Press, 2001, pp. 49–60.

[NM88] G. Nadathur and D. Miller, *An Overview of λProlog*, Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle), MIT Press, 1988, pp. 810–827.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo, *An Assertion Language for Constraint Logic Programs*, Analysis and Visualization Tools

for Constraint Programming (P. Deransart, M. Hermenegildo, and J. Maluszynski, eds.), LNCS, no. 1870, Springer-Verlag, September 2000, pp. 23–61.

[PBH00b] ———, *Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs*, Logic-based Program Synthesis and Transformation (LOPSTR'99), LNCS, no. 1817, Springer-Verlag, March 2000, pp. 273–292.

[Per87] F. Pereira, *C-Prolog User's Manual, Version 1.5*, University of Edinburgh, 1987.

[Pet89] John Peterson, *Untagged Data in Tagged Environments: Choosing Optimal Representations at Compile Time*, Fourth International Conference on Functional Programming Languages and Computer Architecture, ACM Press, September 1989, pp. 89–99.

[PH03] G. Puebla and M. Hermenegildo, *Abstract Specialization and its Applications*, ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03), ACM Press, June 2003, Invited talk, pp. 29–43.

[PHE99] John Peterson, Paul Hudak, and Conal Elliott, *Lambda in Motion: Controlling Robots with Haskell*, PADL, 1999, pp. 91–105.

[PL91] Simon L. Peyton Jones and J. Launchbury, *Unboxed Values as First Class Citizens in a Non-strict Functional Language*, Proceedings of the Conference on Functional Programming and Computer Architecture (Cambridge, Massachussets, USA) (J. Hughes, ed.), Springer-Verlag LNCS523, 26–28 August 1991, pp. 636–666.

[PVC01] Michael Paleczny, Christopher A. Vick, and Cliff Click, *The Java hotspot server compiler.*, Java Virtual Machine Research and Technology Symposium, 2001.

[Qui86] Quintus Computer Systems Inc., Mountain View CA 94041, *Quintus Prolog User's Guide and Reference Manual—Version 6*, April 1986.

[Rig04] Armin Rigo, *Representation-based Just-In-Time Specialization and the Psyco Prototype for Python*, PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (New York, NY, USA), ACM Press, 2004, pp. 15–26.

[RM02] Cliff Randell and Henk L. Muller, *The Well Mannered Wearable Computer*, Personal and Ubiquitous Computing **6** (2002), no. 1, 31–36.

[RP06] A. Rigo and S. Pedroni, *PyPy's Approach to Virtual Machine Construction*, Dynamic Languages Symposium 2006, ACM Press, October 2006.

[RT96] T. Reps and T. Turnidge, *Program Specialization via Program Slicing*, Partial Evaluation. Dagstuhl Castle, Germany, February 1996 (O. Danvy, R. Glück, and P. Thiemann, eds.), Springer LNCS 1110, 1996, pp. 409–429.

[San00] Santos Costa, V., *Parallelism and implementation technology for logic programming languages*, Encyclopedia of Computer Science and Technology **42** (2000), 197–237.

[SC91] Dan Sahlin and Mats Carlsson, *Variable Shunting for the WAM*, Tech. report, Research Report SICS/R–91/9107, SICS, 1991.

[SC99] Vítor Santos-Costa, *Optimising Bytecode Emulation for Prolog*, International Conference on Principles and Practice of Declarative Programming (PPDP'99), LNCS, vol. 1702, Springer-Verlag, 1999, pp. 261–277.

[SCDRA00] V. Santos-Costa, L. Damas, R. Reis, and R. Azevedo, *The Yap Prolog User's Manual*, 2000, Available from `http://www.ncc.up.pt/~vsc/Yap`.

[Sch97] Peter Schachte, *Global Variables in Logic Programming*, ICLP'97, MIT Press, 1997, pp. 3–17.

[SCSL07a] V. Santos Costa, K. Sagonas, and R. Lopes, *Demand-Driven Indexing of Prolog Clauses*, Proceedings of the International Conference of Logic Programming, 2007, pp. 40–58.

[SCSL07b] Vítor Santos-Costa, Konstantinos Sagonas, and Ricardo Lopes, *Demand-Driven Indexing of Prolog Clauses* , International Conference on Logic Programming, LNCS, vol. 4670, Springer Verlag, 2007, pp. 395–409.

[SD02] Tom Schrijvers and Bart Demoen, *Combining an improvement to PARMA trailing with trailing analysis*, PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming (New York, NY, USA), ACM, 2002, pp. 88–98.

[SF06] Tom Schrijvers and Thom Frühwirth, *Optimal Union-Find in Constraint Handling Rules*, Theory and Practice of Logic Programming **6** (2006), no. 1, 213–224.

[SF08] Tom Schrijvers and Thom Frühwirth (eds.), *Constraint Handling Rules - Current Research Topics*, Lecture Notes in Artificial Intelligence, vol. 5388, Springer-Verlag, December 2008.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway, *The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language*, Journal of Logic Programming **29** (1996), no. 1–3, 17–64.

[SSCWD08] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen, *Towards Typed Prolog*, ICLP '08: Proceedings of the 24th International Conference on Logic Programming (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 693–697.

[Ste97] R. Stevens, *A Survey of Stream Processing*, Acta Informatica **34** (1997), 491–541.

[Swe99] Swedish Institute for Computer Science, PO Box 1263, S-164 28

Kista, Sweden, *SICStus Prolog 3.8 User's Manual*, 3.8 ed., October 1999, Available from `http://www.sics.se/sicstus/`.

[Tai98] Antero Taivalsaari, *Implementing a Java Virtual Machine in the Java Programming Language*, Tech. report, Sun Microsystems, March 1998.

[Tar06] Paul Tarau, *BinProlog 2006 version 11.x Professional Edition User Guide*, BinNet Corporation, 2006, Available from `http://www.binnetcorp.com/`.

[Tay89] A. Taylor, *Removal of Dereferencing and Trailing in Prolog Compilation*, Sixth International Conference on Logic Programming, MIT Press, June 1989, pp. 48–60.

[Tay90] ———, *LIPS on a MIPS: Results from a Prolog Compiler for a RISC*, 1990 International Conference on Logic Programming, MIT Press, June 1990, pp. 174–189.

[Tay91a] ———, *High-Performance Prolog Implementation*, Ph.D. thesis, Basser Department of Computer Science, Unversity of Sidney, June 1991.

[Tay91b] A. Taylor, *High Performance Prolog Implementation through Global Analysis*, Slides of the invited talk at PDK'91, Kaiserslautern, 1991.

[TDBD96] Paul Tarau, Koen De Bosschere, and Bart Demoen, *Partial Translation: Towards a Portable and Efficient Prolog Implementation Technology*, Journal of Logic Programming **29** (1996), no. 1-3, 65–83.

[Tip95] Frank Tip, *A Survey of Program Slicing Techniques*, Journal of Programming Languages **3** (1995), 121–189.

[TKA02] W. Thies, M. Karczmarek, and S. Amarasinghe, *StreamIt: A Language for Streaming Applications*, International Conference on Compiler Construction, LNCS, no. 2304, Springer Verlag, 2002, pp. 179–196.

[TN94] Paul Tarau and Ulrich Neumerkel, *A Novel Term Compression Scheme and Data Representation in the BinWAM*, PLILP '94: Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming (London, UK), Springer-Verlag, 1994, pp. 73–87.

[Van94] P. Van Roy, *1983-1993: The Wonder Years of Sequential Prolog Implementation*, Journal of Logic Programming **19/20** (1994), 385–441.

[VB02] C. Vaucheret and F. Bueno, *More Precise yet Efficient Type Inference for Logic Programs*, International Static Analysis Symposium, Lecture Notes in Computer Science, vol. 2477, Springer-Verlag, September 2002, pp. 102–116.

[VD92] P. Van Roy and A.M. Despain, *High-Performance Logic Programming with the Aquarius Prolog Compiler*, IEEE Computer Magazine (1992), 54–68.

[VR90] P.L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, Ph.D. thesis, Univ. of California Berkeley, 1990, Report No. UCB/CSD 90/600.

[Wal95] Malcolm Wallace, *Functional Programming and Embedded Systems*, Ph.D. thesis, York University, January 1995.

[War77] D.H.D. Warren, *Applied logic—its use and implementation as programming tool*, Ph.D. thesis, University of Edinburgh, 1977, Also available as SRI Technical Note 290.

[War83] ———, *An Abstract Prolog Instruction Set*, Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

[War92] D. S. Warren, *Memoing for logic programs*, Communications of the ACM **35** (1992), no. 3, 93–111.

[WB95] G. Welch and G. Bishop, *An Introduction to the Kalman Filter*, Tech. Report TR95-041, Department of Computer Science, University of North Carolina - Chapel Hill, November 1995.

[Wei99] Mark Weiser (ed.), *Special issue on program slicing*, vol. 40, Information and Software Technology, no. 11/12, Elsevier, November 1999.

[Win92] W. Winsborough, *Multiple Specialization using Minimal-Function Graph Semantics*, Journal of Logic Programming **13** (1992), no. 2 and 3, 259–290.

[Wol05] Michael Wolfe, *How Compilers and Tools Differ for Embedded Systems*, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, ACM and IEEE Computer Society, September 2005, Keynote Speech.

[Zho94] Neng-Fa Zhou, *Parameter Passing and Control Stack Management in Prolog Implementation Revisited*, ACM Transactions on Programming Languages and Systems **18** (1994), 752–779.