UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

TRANSFORMACIÓN Y ANÁLISIS DE CÓDIGO DE BYTES ORIENTADO A OBJETOS.

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Miguel Gómez-Zamalloa Gil

Bajo la dirección de la doctora

Elvira Albert Albiol

Madrid, 2010

- **ISBN: 978-84-693-4645-7**

Transformación y Análisis de Código de Bytes Orientado a Objetos

TESIS DOCTORAL

Memoria presentada para obtener el grado de doctor en Ingeniería Informática por

Miguel Gómez-Zamalloa Gil

Dirigida por la profesora

Elvira Albert Albiol

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
Madrid, Julio de 2009

Transformation and Analysis of Object-Oriented Bytecode

PhD THESIS

Miguel Gómez-Zamalloa Gil
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Advisor: Elvira Albert Albiol

July, 2009

Resumen

Predecir el comportamiento de los programas antes de su ejecución es cada vez más importante, especialmente teniendo en cuenta que éstos son cada vez más complejos y son utilizados frecuentemente en situaciones críticas, como operaciones médicas, control aéreo u operaciones bancarias. El *análisis estático de programas* es el proceso por el cual el comportamiento de los programas es analizado sin llegar a ejecutar su código. Tradicionalmente, la mayoría de análisis han sido formulados al nivel del código fuente. No obstante, puede darse el caso de que el análisis deba tratar con código compilado, o código de bytes. Esta situación se da en particular cuando un consumidor de código está interesado en verificar ciertas propiedades de programas de un tercero, pero no tiene acceso directo al código fuente, como suele pasar con el software comercial y con el código móvil. Un ejemplo particularmente interesante es el *análisis del consumo de memoria*, el cual puede ser muy útil en contextos en los cuales el consumidor de código quiere verificar que el programa recibido puede ejecutarse sin que su consumo de memoria exceda un límite dado.

Desafortunadamente, razonar sobre programas reales de código de bytes (con orientación a objetos) es una tarea complicada y costosa. Además de las características propias de la orientación a objetos como la herencia y las invocaciones virtuales, un analizador de código de bytes tiene que tratar con ciertas complicaciones propias de los lenguajes de bajo nivel como la ausencia de estructura de control, el uso de la pila de operandos, etc. Una práctica habitual consiste en resolver el problema en dos pasos, de forma que en primer lugar se transforma, o *decompila*, el programa de código de bytes a una *representación intermedia* de más alto nivel, para poder así formular el análisis sobre dicha representación. Esto permite abstraer las características particulares del lenguaje y así poder desarrollar las

herramientas de análisis sobre representaciones más sencillas. La mayoría de los enfoques desarrollan decompiladores *ad hoc*, es decir, decompiladores exclusivamente diseñados para llevar a cabo una transformación particular. Existe no obstante una alternativa al desarrollo de decompiladores ad hoc, llamada *decompilación interpretativa* por *evaluación parcial*. Como veremos, ésta permite decompilar programas evaluando parcialmente un intérprete respecto a éstos.

Esta tesis contribuye a mejorar el estado del arte en la transformación y el análisis de lenguajes de código de bytes, en concreto: (1) proponiendo e implementando un esquema formal para la decompilación automática por compilación interpretativa de programas de código de bytes (con orientación a objetos) a representaciones intermedias de más alto nivel, en particular utilizando programación lógica; (2) estudiando las aplicaciones prácticas que se tienen gracias a disponer de dichas representaciones; y (3) diseñando e implementando un análisis de consumo de memoria para lenguajes de código de bytes con *recolección de basura*.

Abstract

Predicting the behavior of programs before their actual execution becomes more and more relevant as programs increase in complexity and become used in critical situations such as medical operations, flight control or banking cards. *Static program analysis* is the process of automatically analyzing the behavior of programs without actually executing the code. Traditionally, most analyses have been formulated at the source code level. However, it can be the case that the analysis must consider the compiled code, or bytecode, instead. This may happen, in particular, when the code consumer is interested in verifying some properties of 3rd party programs, but has no direct access to the source code, as usual for commercial software and in mobile code. A particularly interesting example is *memory consumption analysis*, which can be very useful in contexts where the code consumer wants to verify that the received program can run within the actual memory available.

Unfortunately, reasoning about realistic (object-oriented) bytecode programs is rather complicated and time consuming. In addition to the object-oriented features such as inheritance and virtual method invocations, a bytecode analyzer has to deal with several low-level language features like the unstructured control flow, the usage of the operand stack, etc. A usual practice is to first transform, or *decompile*, the bytecode program into a higher-level *intermediate representation*, and then develop the analysis over such representation. This allows abstracting away the particular bytecode language features and developing the analysis tools on much simpler representations. Most of the approaches develop *ad-hoc* decompilers, i.e., decompilers exclusively designed to carry out the particular transformation. There is however an alternative to the development of dedicated decompilers which is the so called *interpretive decompilation* by *partial evaluation*,

which allows decompiling programs by partially evaluating an interpreter w.r.t. them.

This thesis contributes to improve the state-of-the-art in the transformation and analysis of bytecode languages by: (1) providing and implementing a formal framework for the automatic decompilation of (object-oriented) bytecode programs to higher-level intermediate representations, in particular represented using logic programming, by means of interpretive decompilation; (2) studying the practical applications that having such representations can have; and (3) designing and implementing a live memory consumption analysis for bytecode languages with *garbage collection*.

Agradecimientos

La realización de esta tesis no habría sido posible sin la ayuda y el apoyo de mucha gente, y me gustaría aprovechar esta oportunidad para expresarles mi más profunda gratitud.

En primer lugar, me gustaría dar las gracias a mi directora, Elvira Albert, por introducirme en el mundo de la investigación, y por su ayuda incalculable con la tesis. Agradezco su paciencia, sentido del humor y todo el ánimo que he recibido desde el principio. Quiero agradecer también a Germán Puebla sus valiosos consejos y todas las cosas que me ha enseñado (algunas voluntariamente y otras simplemente gracias a su manera de trabajar día a día). En tercer lugar, quiero dar las gracias a toda la gente del equipo COSTA, por crear ese magnífico ambiente de trabajo, especialmente Samir, por estar siempre ahí, Puri por ese sentido del humor, Damiano, etc.

Aprovecho también para dar las gracias a toda la gente del grupo CLIP, especialmente a Edison y a Claudio, por su ayuda solucionando esos problemas técnicos que solía tener con Linux, con las instalaciones de Ciao, etc, etc. Una mención especial es para el director del grupo CLIP, Manuel Hermenegildo, a quien siempre estaré muy agradecido por habernos traído y enseñado el verdadero mundo de la investigación.

Una parte importante de mi formación durante estos años (como investigador y como persona) ha sido posible gracias a las distintas estancias de investigación y visitas que he podido realizar. Quiero por tanto agradecérselo a toda la gente involucrada, desde Paco López, por haberlo hecho posible, hasta toda la gente que me ha acogido y ayudado en mis diferentes destinos. En particular, John Gallagher por ser siempre tan cercano (nunca olvidaré aquel partido R. Madrid - Sevilla en su casa), Gourinath Banda por su ayuda incalculable en Roskilde (la bici, esas pizzas indias, etc,

etc), Michael Leuschel por aquellos días charlando sobre evaluación parcial, Jorge Pérez por su ayuda en Bolonia, Roberto Bagnara por ese magnífico día en Parma, y Andy King por su impresionante dedicación (aquello fue investigación pura, ya casi había olvidado lo que era).

Quiero también dar las gracias a la gente de mi departamento en la Universidad Complutense por ayudar a crear ese magnífico ambiente de trabajo que tengo la suerte de disfrutar. En particular, a Ana Gil por guiarme, a Teresa Martínez por toda la ayuda con la burocracia, al despacho 220 (el futuro de nuestro departamento!), etc, etc.

Finalmente, esta tesis no habría sido posible sin el apoyo y ayuda de mi familia y amigos, especialmente mi mujer, Ali, y mi madre.

Acknowledgments

The making of this thesis would not have been possible without the help and support of many people and I would like to take this opportunity to express my gratitude to all of them.

First, I would like to thank my advisor, Elvira Albert, for introducing me to the world of research, and for her invaluable help with this thesis. I am deeply grateful for her patience with me, her sense of humor and all the encouragement I received from the beginning. Secondly, I want to thank Germán Puebla for his valuable comments, suggestions, and for all the things he has taught me (some on purpose and some others just because of the way he works day by day). Thirdly, I also thank the people of the COSTA team, for creating such an amazing working environment, especially Samir for being always there, Puri for her sense of humor, Damiano, etc.

I also would like to thank all the people in the CLIP lab, especially Edison and Claudio, for their help in sorting out the technical problems I used to have with Linux, the Ciao installations, etc, etc. A special mention goes to its director, Manuel Hermenegildo, to whom I will be always very grateful for bringing the real research world to us.

An important part of my formation during these years (as a researcher and as a person) has been possible thanks to the several research stays and visits I have been able to do. I therefore want to thank all the people involved, from Paco López, for making them possible, to the people hosting and helping me in my different destinations. In particular, I thank John Gallagher for being always so accessible (I will never forget that R. Madrid vs. Sevilla match at his place), Gourinath Banda for his invaluable help (the bike, those indian pizzas, etc, etc), Michael Leuschel for those days of discussions about PE, Jorge Pérez for his help in Bologna, Roberto Bag-

nara for that wonderful day in Parma, and Andy King for his impressive dedication (that was pure research, I had almost forgot what was that).

I also would like to thank the people in my department at the Complutense University for creating this wonderful working environment, in particular I thank Ana Gil for guiding me, Teresa Martínez for her help with all the bureaucracy, the 220 office (the future of the department!), etc, etc.

Finally, the completion of this thesis would not have been possible without the support of my family and friends, especially my wife Ali and my mom.

Índice general

Parte I

Versión en Castellano (Spanish Version)

Capítulo 1

Introducción: Motivación y Contribuciones

1.1. Lenguajes de Código de Bytes

Los lenguajes de programación pueden categorizarse, en general, de acuerdo al modelo de ejecución en el que sus programas se ejecutan. En este sentido, se clasifican en una de estas dos categorías: *compilados* o *interpretados*. En los lenguajes compilados, el *código fuente* es primer lugar traducido, o compilado, a un conjunto de instrucciones específicas de hardware, normalmente conocido como *código objeto*. El programa es entonces ejecutado corriendo el código objeto en el hardware correspondiente. Por el contrario, en los lenguajes interpretados, el código fuente se ejecuta directamente en un intérprete. Esta distinción aplicada a lenguajes de programación es algo confusa, pues en principio, cualquier lenguaje podría ser compilado o interpretado. La categorización, por tanto, refleja habitualmente el modelo de ejecución más popular del lenguaje en cuestión y no sus propiedades. Cada una de las alternativas tiene sus propias ventajas y desventajas. Por ejemplo, ejecutar un programa objeto en la máquina correspondiente, tiende a ser mucho más rápido que ejecutar el programa fuente usando un intérprete del lenguaje en cuestión (aproximadamente en un ratio de 10:1). Por otro lado, los lenguajes interpretados proporcionan cierta flexibilidad respecto a los lenguajes compilados, por ejemplo, facilidad de implementación, facilidad de depuración, y lo más importante, independencia de plataforma.

```
void foo(int n,int m){
    this.f = n + m;
}
```

```
void foo(int,int)
  0: aload_0        3: iadd
  1: iload_1        4: putfield f
  2: iload_2        7: return
```

Figura 1.1: Programa Java Bytecode de ejemplo

Una combinación de ambos enfoques, conocida como *compilación a código de bytes* o *interpretación de código de bytes*, está siendo ampliamente utilizada. Los modelos de ejecución basados en compilación a *código de bytes*, traducen primeramente el código fuente a una representación intermedia, conocida como *código de bytes* (en inglés "bytecode"). Por brevedad, utilizaremos el término "bytecode" a partir de ahora. El bytecode no es el código máquina para ninguna máquina en particular, y puede ser portable entre diferentes arquitecturas. El bytecode es entonces interpretado, o ejecutado en una *máquina virtual*. El término bytecode proviene de los repertorios de instrucciones en los que éstas incluyen un código de operación de un byte de longitud seguido de una serie de parámetros opcionales. Las instrucciones bytecode son habitualmente similares a las instrucciones hardware tradicionales. Así por ejemplo, los lenguajes bytecode tienen un flujo de control desestructurado con varios tipos de saltos (condicionales e incondicionales) y utilizan habitualmente una pila de operandos para realizar cálculos auxiliares. Sin embargo, el hecho de que las instrucciones bytecode estén en principio pensadas para ser ejecutadas por software, hace que éstas tengan en ocasiones cierta complejidad, especialmente en el caso de lenguajes bytecode orientados a objetos y declarativos. Para hacerse una idea, la Figura 1.1 muestra el código fuente (Java) y el correspondiente bytecode de un método que toma dos números enteros, los suma, y asigna el resultado al atributo `f` del objeto *this*. Nótese que, en el bytecode de Java, el objeto *this* se pasa explícitamente en la variable local 0. Por tanto, la instrucción `aload_0`, apila la referencia al objeto *this*, en la cima de la pila de operandos.

En cuanto a eficiencia, se puede decir que el modelo de compilación a bytecode, se encuentra en algún punto entre el modelo puro basado en

compilación y el basado en interpretación; mientras que mantiene las ventajas de los modelos basados en interpretación, en particular, la independencia de plataforma. Más aún, nada obliga a un lenguaje bytecode a ser exclusivamente interpretado. De hecho, la compilación *just-in-time* (JIT) puede usarse para acelerar la ejecución del bytecode. Los compiladores JIT convierten el código fuente, o bytecode en este caso, a código nativo gradualmente durante la ejecución del programa, obteniéndose así un mejor rendimiento. La compilación a bytecode junto con la compilación JIT puede por tanto combinar la mayoría de las ventajas de los modelos de ejecución basados en compilación y los basados en interpretación. Ésta es la principal razón del éxito de los entornos de programación de *Microsoft .NET* y *Java*, los cuales son, sin lugar a dudas, los entornos de programación más utilizados en la actualidad.

Java Bytecode *Java Bytecode* es el lenguaje que la máquina virtual de Java (JVM) [65] ejecuta. Fue originalmente diseñado por *Sun Microsystems* como un lenguaje intermedio en el entorno de desarrollo de Java. Una instrucción Java Bytecode consiste en un código de operación, el cual especifica la operación a ser ejecutada, seguido de cero o más operandos con los valores que se utilizan en la operación en cuestión. La JVM utiliza, entre otras, las siguientes estructuras de datos, que las instrucciones manipulan durante su ejecución: el *contador de programa*, que contiene el índice de la instrucción actual, la *pila de operandos* y el *array de variables locales*, en los cuales se almacenan los parámetros, variables y resultados intermedios, el *montículo* o "heap" (utilizaremos el término "heap" a partir de ahora), en el cual se almacenan los objetos y arrays, y la habitual *pila de llamadas* o *pila de "frames"* para tratar con las llamadas y vueltas de llamada a métodos. El lenguaje Java Bytecode incluye, por un lado, las instrucciones habituales de bajo nivel para: transferir valores entre la pila de operandos y el array de variables locales (y viceversa), realizar operaciones aritméticas, saltar condicional o incondicionalmente a otras partes del código, llamar y volver de métodos, etc. Por ejemplo, la instrucción "`iload_1`" carga la variable local 1 en la cima de la pila de operandos, y la instrucción "`iadd`" suma (y desapila) los dos valores de la cima de la pila, y apila el resultado. Por otro lado, Java Bytecode, al tener orientación a objetos y concurrencia, incluye también instrucciones para: crear objetos y arrays, escribir y leer

atributos y elementos de arrays, realizar invocaciones virtuales, adquirir y liberar monitores, etc. Por ejemplo, la instrucción "`putfield f`" escribe el valor que hay en la cima de la pila, en el atributo `f` del objeto referenciado por la dirección de memoria almacenada bajo la cima de la pila.

Aunque Java es el lenguaje más común que se compila a Java Bytecode, hay sin embargo muchos compiladores de diferentes lenguajes de alto nivel a Java Bytecode. Algunos de los más conocidos son: *jython* para programas *Python*, *jRuby* para *Ruby* and *jGNAT* para *Ada*.

El Lenguaje Intermedio Común de .NET El *Lenguaje Intermedio Común* ("*Common Intermediate Language*" o CIL) es el lenguaje bytecode intermedio utilizado en el entorno .NET. Así, los diferentes lenguajes fuente utilizados en .NET, se compilan a CIL. Como Java Bytecode, CIL es un lenguaje orientado a objetos y basado en pila. Incluye por tanto la misma clase de instrucciones bytecode. A diferencia de Java Bytecode, CIL no está pensado para ser interpretado. Fue sin embargo, desde sus comienzos, pensado para ser compilado a código máquina utilizando compilación JIT. Incluso, en ocasiones, el bytecode se compila por completo a código máquina antes de ser ejecutado para mejorar el rendimiento. El entorno .NET es una de las piedras angulares de la tecnología moderna de Microsoft, y se utiliza para el desarrollo de la mayoría de aplicaciones creadas para la plataforma Windows.

Existen otros muchos lenguajes bytecode bien conocidos y ampliamente utilizados, tanto imperativos, como el *p-code* utilizado en algunas implementaciones de Pascal; como declarativos, como el bytecode *WAM*, utilizado en la mayoría de implementaciones de Prolog, el bytecode de *Haskell Hugs'98*, o el bytecode *Erlang BEAM*, por nombrar algunos.

Esta tesis está principalmente centrada en lenguajes bytecode imperativos y con orientación a objetos. En particular, como veremos, los diferentes contenidos técnicos de la tesis, así como los distintos prototipos implementados, consideran subconjuntos representativos de Java Bytecode.

1.2. Análisis Estático de Programas

Predecir el comportamiento de los programas antes de su ejecución es cada vez más relevante, especialmente teniendo en cuenta que éstos son cada vez más complejos y son frecuentemente utilizados en situaciones críticas, como operaciones médicas, control aéreo o tarjetas bancarias. Ser capaces de demostrar de forma automática que los programas cumplen con sus especificaciones funcionales, es un factor básico para su éxito. El *análisis estático de programas* es el proceso por el cual el comportamiento de los programas es analizado sin llegar a ejecutar su código. Por el contrario, cuando el análisis se realiza ejecutando el programa, éste se denomina *análisis dinámico*. Los análisis estáticos clásicos tratan de inferir propiedades de los programas como: *ausencia de errores*, *terminación*, *coste* o *consumo de recursos* (tiempo o memoria), *vida de variables*, *forma de punteros*, etc. Habitualmente, los análisis estáticos basan su funcionamiento en métodos formales. Algunos de los más habituales son: la *interpretación abstracta*, el *chequeo de modelos* y los *sistemas de tipos*. Esta tesis está basada principalmente en el análisis estático basado en interpretación abstracta.

Interpretación Abstracta. La técnica de la interpretación abstracta [29] proporciona un marco general para computar aproximaciones seguras (es decir, abstracciones) del comportamiento de los programas. Su principal aplicación práctica es el análisis estático formal. Los analizadores basados en interpretación abstracta, infieren información de los programas interpretándolos ("ejecutándolos"), utilizando valores abstractos en lugar de valores concretos. Estos analizadores son paramétricos respecto al llamado *dominio abstracto*, el cual proporciona una representación finita de un conjunto posiblemente infinito de valores. Dominios diferentes capturan clases distintas de propiedades, con un nivel distinto de precisión, y a un coste computacional diferente.

Los analizadores basados en interpretación abstracta se han estudiado tanto en el contexto de lenguajes declarativos como en el de lenguajes imperativos. A continuación enumeramos algunos sistemas de análisis:

El Analizador ASTRÉE. *ASTRÉE* [30] es un analizador estático de programas, desarrollado en la *École Normale Supérieure* por Cousot *et. al.*, que es capaz de demostrar ausencia de errores en tiempo de ejecución en programas C. ASTRÉE fue capaz por ejemplo de demostrar, de forma totalmente automática, la ausencia de errores en el software primario de control aéreo del Airbus A340, un programa de 132.000 líneas.

El Sistema CiaoPP. CiaoPP [48] es el preprocesador basado en interpretación abstracta del sistema de *Programación Lógica con Restricciones*, "Constraint Logic Programming" (CLP), Ciao-Prolog [81]. Éste incluye un buen número de funcionalidades para realizar depuración, análisis y transformación *fuente a fuente* de programas Ciao-Prolog. Algunas de las propiedades que el sistema es capaz de inferir son: *tipos*, *modos* y otras propiedades de instanciación de variables, *no fallo*, *determinismo*, *cotas* en el *coste de recursos*, cotas del tamaño de los términos del programa, etc. CiaoPP es también capaz de realizar varios tipos de transformaciones fuente a fuente de programas, como *especialización* de programas, *paralelización* de programas (incluyendo *control de granularidad*), etc.

Otros sistemas de análisis estático bien conocidos (no comerciales y comerciales) son: *Lint*, *CCA* y *BOON* para programas C, *CodeSonar* para C++, *Fluid* y *jLint* para Java, y muchos otros. Otros analizadores estáticos no han llegado a ser herramientas autointegradas sino que aparecen integrados en diversos compiladores. Un ejemplo de esto es el verificador de la JVM que integra un analizador del flujo de datos ("data-flow analysis").

Tradicionalmente, la mayoría de análisis han sido formulados al nivel del código fuente. No obstante, puede darse el caso de que el análisis deba tratar con código compilado, o bytecode. Esta situación se da en particular cuando un consumidor de código está interesado en verificar ciertas propiedades de programas de un tercero, pero no tiene acceso directo al código fuente, como suele pasar con el software comercial y con el código móvil. Éste es el marco general en el que nació la idea del *Código con demostración*, "Proof-Carrying Code" [73]: para que un usuario pueda verificar cierto código, éste debe venir acompañado de una *demostración* de que se cumplen ciertas propiedades de seguridad, referidas al código compilado o bytecode (posiblemente inferida por análisis estático), de forma que el usuario pueda

chequear la corrección de la demostración proporcionada y verificar que las propiedades efectivamente se cumplen (por ejemplo, que el código no requiere más de una cierta cantidad de memoria para ser ejecutado, o que ejecuta en menos de una cierta cantidad de tiempo).

Existe por tanto la necesidad de desarrollar herramientas de análisis y verificación que trabajen directamente al nivel de programas bytecode. Desafortunadamente, razonar sobre programas reales bytecode (con orientación a objetos) es una tarea complicada y costosa. Además de las características propias de la orientación a objetos como la herencia y las invocaciones virtuales, un analizador de bytecode tiene que tratar con ciertas complicaciones propias de los lenguajes de bajo nivel como la ausencia de estructura de control, el uso de la pila de operandos, etc.

1.3. Del Bytecode a Representaciones Intermedias

En el contexto del análisis de lenguajes bytecode, una práctica habitual consiste en resolver el problema en dos pasos: (1) transformar el programa bytecode a una *representación intermedia* (RI) de más alto nivel, y (2) formular el análisis sobre dicha RI. Esto permite abstraer las características particulares del lenguaje bytecode y así poder desarrollar las herramientas de análisis sobre representaciones más sencillas de tratar. Otra ventaja importante de este enfoque, es que éste permite la posibilidad de reutilizar la fase de análisis (fase (2)) para poder analizar distintos lenguajes (bytecode y no bytecode), siempre que éstos puedan ser transformados a la misma RI. Utilizaremos a partir de ahora el término *decompilación* para referirnos a la transformación de bytecode a la RI, pues se traduce un lenguaje de bajo a alto nivel.

La mayoría de los enfoques desarrollan decompiladores *dedicados*, o *ad hoc*, es decir, decompiladores exclusivamente diseñados para llevar a cabo una decompilación particular. Existe no obstante una alternativa al desarrollo de decompiladores dedicados, llamada *decompilación interpretativa* por *evaluación parcial*. Como veremos, ésta permite decompilar programas evaluando parcialmente un intérprete respecto a éstos.

Evaluación Parcial. La *Evaluación Parcial* (EP) [55] es una técnica, basada en semántica, de transformación fuente a fuente de programas, que permite especializar programas respecto a parte de sus datos de entrada. Se suele llamar de hecho *especialización de programas*. Consideremos un programa P, y sus datos de entrada I divididos en I_{static} y $I_{dynamic}$. I_{static} son los datos estáticos, es decir, los datos conocidos en tiempo de compilación, y $I_{dynamic}$ son el resto de los datos. Podemos ver la ejecución del programa P como una función de los datos de entrada a los de salida de la siguiente forma:

$$P : I_{static} \times I_{dynamic} \longrightarrow O$$

Un evaluador parcial trasforma el par $\langle P, I_{static} \rangle$ en $P' : I_{dynamic} \longrightarrow O$, realizando las computaciones de P que dependen de I_{static} en tiempo de compilación. Se denomina a P' el *programa residual*, el cual debería ser más eficiente que el programa original P.

El Enfoque Interpretativo de Compilación. Una aplicación particularmente interesante de la EP, primeramente descrita por Yoshihiko Futamura en los años 70 [40], aparece cuando el programa P a ser evaluado parcialmente es un intérprete de un lenguaje de programación. Esto se conoce como el *enfoque interpretativo de compilación* o la *primera proyección de Futamura*. Asumamos un intérprete, escrito en un lenguaje *objetivo* o "target" L_T, que interpreta programas escritos en un lenguaje fuente o "source", L_S. Entonces, si I_{static} es un programa fuente, escrito en L_S, la evaluación parcial del intérprete respecto al programa (datos), producirá P', una versión del intérprete que sólo puede ejecutar ese programa fuente, que está escrita en el lenguaje de implementación del intérprete, L_T, y que no necesita el código fuente para poder interpretarlo. P' puede considerarse como una versión compilada de I_{static} al lenguaje objetivo L_T. La compilación interpretativa permite por tanto compilar programas escritos en L_S a otro lenguaje L_T, evaluando parcialmente un intérprete de L_S escrito en L_T respecto a ellos.

En el caso particular de la decompilación de lenguajes bytecode, el enfoque interpretativo de compilación nos permitiría decompilar un programa bytecode escrito en un lenguaje bytecode BC, a una representación

de más alto nivel, digamos que al lenguaje HL, evaluando parcialmente un intérprete de BC escrito en HL. Este enfoque es, en principio, más genérico, flexible, más seguro y más fácil de mantener que un decompilador dedicado para la misma tarea. Dichas ventajas serán discutidas más adelante en la Sección 2. El enfoque interpretativo, aunque es en principio muy atractivo, no ha sido muy utilizado en la práctica, principalmente debido a la dificultad de encontrar estrategias de EP con las cuales se puedan obtener decompilaciones efectivas, o de calidad, y de forma eficiente.

1.4. Análisis de Consumo del Heap para Bytecode

La investigación sobre el consumo de recursos de los programas comenzó con el trabajo de Wegbreit es 1975 [86], en el cual se propuso un análisis del rendimiento de un programa basado en la derivación de una expresión matemática que representaba su comportamiento en tiempo de ejecución. El enfoque para realizar análisis estático de coste es el siguiente: dado un programa de entrada, (1) en una primera fase, el análisis de coste genera un sistema de ecuaciones de coste, "cost equation system" (CES) a partir del programa, que captura las relaciones entre las diferentes partes del código. Un CES es un conjunto de ecuaciones de recurrencia que expresa el coste del programa en términos de los tamaños de sus argumentos de entrada. (2) En la segunda fase, el CES se trata de resolver o aproximar, típicamente utilizando técnicas algebraicas, obteniéndose una *forma cerrada* (por ejemplo, sin incluir recurrencias) que representa una *cota superior* (o *cota inferior*) del coste.

Los análisis de coste se han estudiado de forma intensiva en el contexto de la programación declarativa, tanto para programación funcional [78, 79, 43, 15], como para programación lógica [33, 34], y también en el contexto de lenguajes imperativos de alto nivel (centrándose principalmente en la estimación de tiempos en el *caso peor* y en el diseño de *modelos de coste* [89]). Tradicionalmente, como pasa con la mayoría de análisis estáticos, los análisis de coste han sido formulados al nivel del código fuente. Sin embargo, como hemos visto, existen situaciones en las que no se tiene acceso a éste, sino que sólamente se tiene acceso al código compilado o al

bytecode. Recientemente, en [2] se ha propuesto un esquema genérico para el análisis de coste de Java Bytecode, el cual constituye la base formal sobre la que se sustenta el sistema COSTA [5].

El Sistema COSTA**.** COSTA [5] es un prototipo de investigación que es capaz de realizar automáticamente **A**nálisis de **COS**te y **T**erminación para Java Bytecode. El sistema recibe como entrada un programa bytecode y un modelo de coste, elegido a partir de una selección de descripciones de recursos, y trata de obtener una cota del consumo de recursos del programa respecto al modelo de coste dado. COSTA sigue el enfoque estándar para realizar el análisis de coste, es decir, primeramente produce un CES, el cual es una forma extendida de relaciones de recurrencia, y después trata de obtener una forma cerrada, que representa una cota superior del coste del programa, utilizando para ello un resolutor de ecuaciones de recurrencia propio [9].

Una de las aplicaciones más interesantes del análisis de coste, la cual presenta importantes retos por resolver, es el análisis de consumo del heap. Éste trata de inferir cotas del espacio de heap consumido por un programa. De nuevo, los análisis de heap se han formulado habitualmente al nivel del código fuente (ver por ejemplo [83, 49, 85, 53] en el contexto de la programación funcional y [51, 23] para lenguajes imperativos de alto nivel). En el contexto de lenguajes bytecode, el análisis de consumo de heap puede tener aplicaciones muy interesantes. Por ejemplo, la *certificación de cotas de recursos*, "resource bound certification" [32, 8, 10, 50, 22], propone utilizar propiedades de seguridad incluyendo requerimientos de coste, es decir, el código recibido ha de adherirse a unos requerimientos específicos respecto a su consumo de memoria. También, las cotas del consumo del heap pueden resultar útiles en sistemas empotrados ("embedded systems"), por ejemplo, en tarjetas inteligentes en las cuales la memoria es limitada y no puede recuperarse de forma sencilla.

Desafortunadamente, la gestión automática de memoria, también llamada *recolección de basura* ("garbage collection" o GC), la cual es utilizada cada vez más habitualmente en lenguajes bytecode como en Java Bytecode y en el .NET CIL, provoca que el problema de predecir la memoria utilizada por un programa sea mucho más difícil. Una primera aproximación al

problema es inferir el *consumo total* de memoria, es decir, la cantidad acumulada de memoria alojada por el programa ignorando el efecto del GC. Si se dispone de dicha cantidad de memoria, está asegurado que el programa puede ejecutar, incluso aún cuando no se aplica el GC. Sin embargo, ésta es una estimación muy pesimista del consumo real del programa. Recientemente, en [83, 18, 24] se han propuesto *análisis de consumo del heap activo*, "live heap space analysis", los cuales tratan de aproximar el tamaño de la memoria *activa* en el heap durante la ejecución del programa, resultando en una aproximación mucho más precisa. Dichos enfoques, están sin embargo restringidos a cotas polinomiales y a métodos no recursivos [18], o a cotas lineales, en este caso, capaz de tratar recursión [24].

1.5. Objetivos y Contribuciones

El principal objetivo de esta tesis es mejorar el estado del arte en la transformación y el análisis de lenguajes bytecode, en concreto: (1) proponiendo e implementando un esquema formal para la decompilación automática por compilación interpretativa de programas bytecode (con orientación a objetos) a representaciones intermedias de más alto nivel, en particular utilizando programación lógica (LP); (2) estudiando las aplicaciones prácticas que se tienen gracias a disponer de dichas RIs basadas en LP; y (3) diseñando e implementando un análisis de consumo de la memoria activa para lenguajes bytecode con recolección de basura. Más detalladamente, las contribuciones de esta tesis son las siguientes:

1. **Decompilación interpretativa de bytecode a LP:** Ha habido en la literatura varias *pruebas de concepto* mostrando que el enfoque interpretativo es factible [62, 47, 75, 63]. Sin embargo, en la práctica, a la hora de decompilar lenguajes y programas complejos, aún quedan varias cuestiones por resolver. Éstas incluyen: *escalabilidad*, que a su vez depende de la *composicionalidad* del enfoque, y *efectividad*, es decir, obtener programas decompilados de calidad. Esta tesis presenta, por lo que conocemos, el primer esquema de decompilación interpretativa que es capaz de decompilar lenguajes bytecode reales a una representación de alto nivel. En particular, decompilamos Java Bytecode a Prolog.

a) **Estrategias de control:** Una de las principales dificultades de la decompilación interpretativa, y de la EP en general, es el ser capaz de tratar adecuadamente con signaturas infinitas. Hemos propuesto técnicas novedosas que nos han permitido definir reglas de control sofisticadas. En particular, hemos introducido la relación de la *subsunción homeomórfica basada en tipos*, una generalización de la relación original de *subsunción homeomórfica*, que proporciona resultados más precisos en presencia de signaturas infinitas. Hemos mostrado como esta técnica, a parte de resultar crucial en la especialización de intérpretes, mejora el estado del arte de las herramientas de especialización "online". Este trabajo fue primeramente propuesto en el Artículo 3 (ver el Apéndice A) y posteriormente extendido y reformulado en el Artículo 4, el cual ha sido publicado en la revista "*Information Processing Letters*".

b) **Controlando la *polivarianza* de la EP:** Incluso una vez después de haber integrado la *subsunción homeomórfica basada en tipos* en la EP, los programas decompilados que se obtienen tienden a tener demasiadas versiones especializadas (redundantes) para algunos predicados. Este aspecto se ha estudiado en el Artículo 2, donde se proponen técnicas avanzadas para controlar la *polivarianza* del proceso de EP, es decir, evitar tener dichas versiones especializadas redundantes.

c) **Cómo escribir el intérprete de bytecode:** Como se ha mostrado en trabajos previos de compilación interpretativa, las características del intérprete en cuestión, resultan cruciales para poder obtener una especialización exitosa. Hemos identificado los aspectos necesarios que el intérprete debe tener para poder obtener un esquema composicional de decompilación.

d) **Decompilación óptima:** Aseguramos la calidad de la decompilación, tanto en términos de efectividad como de eficiencia, proponiendo una serie de *criterios de optimalidad*. Éstos básicamente requieren que: (1) la decompilación no genere código más de una vez para cada punto de programa, y (2) que haya como máximo una regla residual para cada bloque del programa byte-

code. Proponemos un esquema de decompilación que es *óptimo* respecto a estos criterios de optimalidad. Esto asegura tanto la escalabilidad del proceso como la calidad de las decompilaciones. Este trabajo junto con lo descrito en el punto (c) dieron lugar al Artículo 5.

e) **Tratando con la orientación a objetos:** Mostramos como nuestro esquema se puede adaptar fácilmente para tratar las características de la orientación a objetos. En particular, proponemos mecanismos para: tratar con el heap y con sus instrucciones asociadas, representar clases por medio de módulos Prolog, y representar invocaciones virtuales por medio de llamadas Prolog con calificación de módulo.

f) **Implementación y evaluación experimental:** Todas las técnicas mencionadas han sido implementadas e integradas en un decompilador prototipo de Java Bytecode secuencial a Prolog, llamado jbc2prolog. Presentamos resultados experimentales usando dicho prototipo (utilizando, y contrastando con, otros sistemas). En particular, se han estudiado tanto la escalabilidad como la eficiencia de nuestro enfoque, utilizando el conjunto de "benchmarks" JOlden [54]. El trabajo descrito en los puntos (b), (c), (d), (e) y (f), ha dado lugar al Artículo 6, el cual ha sido recientemente publicado por la revista "*Information and Software Technology*". Este Artículo, por tanto, lleva a cabo el objetivo (1) (ver más arriba).

2. **Aplicaciones de la decompilación interpretativa:** Utilizar un lenguaje declarativo para definir las RIs ofrece ventajas importantes. En particular, se pueden reutilizar las potentes y avanzadas herramientas de análisis y transformación de programas existentes en el contexto de la programación declarativa (ya probadas correctas y efectivas) para el análisis y transformación de programas bytecode. Este trabajo se corresponde por tanto con el objetivo (2).

   a) **Reutilizando herramientas de análisis de LP:** Utilizando el sistema `CiaoPP` sobre nuestros programas decompilados, hemos sido capaces de demostrar ciertas propiedades no triviales de programas Java Bytecode como terminación y *ausencia de*

errores, así como, para programas sencillos, inferir cotas de su consumo de recursos. Este trabajo aparece en el Artículo 1.

b) **Generación de datos de prueba:** Uno de los enfoques estándar para generar automáticamente datos de prueba consiste en hacer *ejecución simbólica* de los programas [28, 56]. En ésta, los contenidos de las variables son expresiones en lugar de valores concretos. El hecho de que nuestros programas decompilados sean programas Prolog *ejecutables*, nos permite poder utilizar técnicas inherentes a la CLP (como el *backtracking* y la manipulación de restricciones) para realizar la ejecución simbólica. Hemos desarrollado un esquema novedoso de *generación de casos de prueba* utilizando nuestros programas (C)LP decompilados. Mostramos además como la fase de generación de casos de prueba en CLP, puede verse como otra EP, lo que nos permite obtener no sólo casos de prueba, sino también *generadores de casos de prueba*. Este trabajo ha dado lugar al Artículo 7. Como contribución tangencial, hemos aplicado la misma idea para generar automáticamente casos de prueba para Prolog. Un estudio preliminar en esta dirección aparece en el Artículo 8.

3. **Análisis de consumo del heap:**

a) **Consumo total del heap:** En primer lugar, hemos desarrollado una aplicación novedosa del analizador de coste presentado en [3], para inferir cotas superiores del consumo del heap de programas secuenciales Java Bytecode. Para ello, simplemente hemos propuesto un modelo de coste que define el coste de las instrucciones que alojan memoria, en términos de las unidades de heap (memoria) que consumen. Podemos entonces generar *relaciones de coste* del consumo del heap que pueden ser utilizadas directamente para inferir cotas superiores en el consumo del heap de programas Java Bytecode.

b) **Análisis de consumo del heap *activo* en lenguajes con GC:** En presencia de recolección automática de basura, el enfoque propuesto proporciona estimaciones demasiado pesimistas del consumo real de los programas. Esta tesis presenta un esquema general para inferir el consumo *pico* del heap durante la

ejecución de un programa bytecode, es decir, el máximo uso de memoria *activa* durante su ejecución, no estando restringido a ninguna clase de complejidad (como pasa con enfoques anteriores).

c) **Implementación y evaluación experimental:** Los análisis han sido implementados e integrados en el sistema COSTA. Hemos realizado una evaluación experimental con una serie de aplicaciones Java que hacen un uso intensivo del heap, incluyendo los "benchmarks" JOlden [54]. Los resultados demuestran que nuestro enfoque es capaz de obtener cotas del consumo de heap activo razonablemente precisas de forma totalmente automática. Todo este trabajo sobre el análisis de consumo del heap ha dado lugar a los Artículos 9 y 10, llevándose a cabo por tanto el objetivo (3).

1.6. Organización de la Tesis

Esta tesis sigue el formato "tesis por publicaciones" y consiste por tanto en una introducción describiendo sus principales objetivos, contribuciones y conclusiones, la cual se presenta en los Capítulos 1, 2, 3, 4 y 5, y, el conjunto de publicaciones editadas que han dado lugar a la tesis, presentadas en el formato y longitud en el que aparecen en las correspondientes publicaciones, como un apéndice.

El resto de la tesis se organiza de la siguiente manera: El Capítulo 2 ofrece una visión general de todo el trabajo correspondiente a la contribución (1). En particular, la Sección 2.1 proporciona resumidamente los fundamentos básicos de la EP de programas lógicos, después, se presentan en la Sección 2.2 los retos que aparecen al especializar un intérprete de bytecode, la Sección 2.3 introduce la relación de *subsunción homeomórfica basada en tipos*, las Secciones 2.4 y 2.5 resumen los detalles técnicos de los esquemas de decompilación modular y óptimo, la Sección 2.6 discute brevemente algunos detalles de implementación, así como la evaluación experimental llevada a cabo, y finalmente la Sección 2.7 presenta el trabajo relacionado en decompilación (interpretativa).

El Capítulo 3 proporciona una visión general del trabajo realizado sobre las aplicaciones que surgen al utilizar nuestro enfoque de decompila-

ción interpretativa, bien para analizar programas bytecode (Sección 3.1), o bien para realizar generación automática de datos de entrada para pruebas (Sección 3.2). El Capítulo 4 resume nuestro trabajo sobre el análisis de consumo del heap (Sección 4.1) y su extensión para considerar el efecto de la recolección de basura (Sección 4.2), y discute finalmente sobre el trabajo relacionado en el área (Sección 4.3). Finalmente, el Capítulo 5 presenta las conclusiones de la tesis y discute algunas líneas de trabajo en progreso y futuro.

Todos los detalles técnicos aparecen en los Artículos que han dado lugar a la tesis, los cuales aparecen íntegramente en el Apéndice A.

Capítulo 2

Decompilación Interpretativa de Bytecode a LP

Decompilar lenguajes bytecode a representaciones intermedias es hoy en día una práctica habitual en el desarrollo de analizadores, verificadores, chequeadores de modelos, etc. Por ejemplo, en el contexto de código móvil, al no tenerse acceso al código fuente, la decompilación puede facilitar la reutilización de herramientas de análisis y chequeadores de modelos existentes. En general, las representaciones intermedias de alto nivel permiten abstraer las características particulares de cada lenguaje, permitiendo por tanto que las herramientas puedan trabajar sobre representaciones más sencillas. Así por ejemplo, Java Bytecode ha sido decompilado a una representación basada en reglas en [2], a programas basados en cláusulas en [69], a una representación basada en código de *tres direcciones* en el sistema Soot [84] y al lenguaje procedural tipado BoogiePL en [36]. Así mismo, en [87] el análisis de programas Java es formalizado y llevado a cabo utilizando programas Datalog, y, en [47] programas de código ensamblador PIC son transformados a programas lógicos para su posterior análisis. Estos trabajos han demostrado que, las representaciones basadas en reglas usadas en la programación declarativa en general—y en LP en particular—proporcionan un formalismo adecuado a la hora de definir dichas representaciones intermedias. Por ejemplo, como puede verse en [2, 69, 47], la pila de operandos utilizada en los lenguajes bytecode, puede representarse explícitamente por medio de variables lógicas, y el flujo de control desestructurado puede transformarse en reglas condicionales y recursivas.

Las representaciones intermedias resultantes simplifican en gran medida el desarrollo de las herramientas arriba mencionadas para lenguajes modernos y, además, herramientas existentes desarrolladas para lenguajes declarativos (las cuales han demostrado ser efectivas) pueden ser directamente aplicadas.

La mayoría de los enfoques citados con anterioridad desarrollan decompiladores dedicados o *ad hoc*, para llevar a cabo la decompilación correspondiente. Como se comentó en la Sección 1.3, una alternativa muy prometedora al desarrollo de decompiladores dedicados es la decompilación interpretativa por evaluación parcial. Las ventajas de la (de)compilación interpretativa respecto los decompiladores dedicados son bien conocidas y discutidas en la literatura de la EP. Brevemente, éstas incluyen:

1. *Flexibilidad*: Es muy sencillo modificar el intérprete correspondiente para *afinar* la decompilación (por ejemplo, para observar nuevas propiedades de interés). Así por ejemplo, en el Artículo 1, un intérprete de Java Bytecode es instrumentado con un argumento adicional que almacena la *traza* de instrucciones bytecode para poder así acumular la historia de la ejecución. Los programas decompilados usando este intérprete incluyen por tanto en sus reglas un argumento adicional con la traza de ejecución al nivel de Java Bytecode. Esta traza permite observar distintas propiedades de los programas. Por ejemplo, se puede demostrar la ausencia de errores en tiempo de ejecución a base de demostrar que la traza no contiene instrucciones relacionadas con errores o determinados tipos de excepciones.

2. *Seguridad*: Es en principio mucho más difícil demostrar, o *confiar* en que un decompilador dedicado preserve la semántica del programa original. Por ejemplo, a la hora de definir nuestro intérprete de Java Bytecode, hemos utilizado la especificación formal Bicolano [77], la cual fue escrita, y verificada, utilizando el asistente de demostraciones Coq [11].

3. *Mantenibilidad*: Los cambios introducidos en la semántica del lenguaje pueden ser fácilmente reflejados en el intérprete. Más adelante, veremos como efectivamente definir, así como hacer evolucionar, un intérprete de bytecode en Prolog es una tarea bastante sencilla.

El reto se encuentra ahora en definir un esquema de decompilación interpretativa práctico y escalable, que sea capaz de obtener programas decompilados de calidad. Una vez seamos capaces de definir dicho esquema, será posible entonces hacer valer las ventajas arriba mencionadas.

En la literatura ha habido varias pruebas de concepto demostrando que la decompilación interpretativa es efectivamente aplicable (ver por ejemplo [47, 62]), sin embargo aún quedan muchas cuestiones por resolver si queremos aplicarlo a la decompilación de lenguajes y programas reales. A continuación, la Figura 2.1 enumera dichas cuestiones para facilitar posteriores referencias. Esta tesis responde afirmativamente a dichas cuestiones

a) *¿es el enfoque escalable?*

b) *¿preservan los programas decompilados la estructura de los programas originales?*

c) *¿es comparable la calidad de los programas decompilados con la de los programas obtenidos utilizando decompiladores ad hoc?*

Figura 2.1: Cuestiones por resolver de la decompilación interpretativa

proponiendo un esquema modular de decompilación, que asegura la obtención de decompilaciones de calidad que preservan la estructura de los programas originales.

El resto del capítulo está organizado de la siguiente manera:

- En primer lugar, la Sección 2.1 presenta informalmente los fundamentos de la EP de programas lógicos necesarios para comprender los detalles presentados en el resto del capítulo. Dichos fundamentos se presentan formalmente en la Sección 2 del Artículo 6.

- La Sección 2.2 introduce las dificultades y retos que surgen al especializar un intérprete de bytecode utilizando un ejemplo representativo.

- La Sección 2.3 presenta informalmente la *Subsunción homeomórfica basada en tipos*, una extensión de la relación de *subsunción homeomórfica* original que, teniendo en cuenta información del progra-

ma, obtiene resultados más precisos en presencia de signaturas infinitas. En los Artículos 4 y 3 se presentan los correspondientes detalles formales así como una evaluación experimental detallada.

- Las Secciones 2.4 y 2.5 introducen los ingredientes necesarios para desarrollar un esquema de decompilación *modular* y *óptimo*, capaz de tratar los puntos *a)*, *b)* y *c)* (ver Figura 2.1). Primeramente definimos la noción de *optimalidad* por medio de una serie de *criterios de optimalidad*. Después, presentamos las limitaciones de la decompilación *no-modular* e identificamos los componentes necesarios para posibilitar la definición de un esquema modular. Éstos incluyen, el cómo escribir el intérprete y el cómo controlar un evaluador parcial "online" para poder preservar la estructura del programa original respecto a las invocaciones a métodos. Finalmente, introducimos un esquema de decompilación interpretativa capaz de responder a los puntos *(a)*, *(b)* y *(c)*, produciendo programas decompilados cuya calidad es equivalente a la obtenida utilizando decompiladores ad hoc. Esto requiere una decompilación *a nivel de bloques* que evite las duplicaciones y reevaluaciones de código.

- La Sección 2.6 resume los resultados experimentales obtenidos con dos prototipos de decompilador de Java Bytecode a Prolog que incorporan las técnicas mencionadas. Dichos resultados demuestran empíricamente la escalabilidad y eficiencia del enfoque propuesto sobre una serie de programas reales escritos en Java Bytecode.

- Finalmente, la Sección 2.7 presenta el trabajo relacionado en el campo de la decompilación de lenguajes bytecode.

Para concretar, nuestro enfoque de decompilación ha sido formalizado en el contexto de la EP de programas lógicos. No obstante, las ideas propuestas que han hecho posible su aplicación práctica son de interés para la (de)compilación interpretativa de cualquier par de lenguajes origen y destino.

2.1. Fundamentos Básicos de la Evaluación Parcial de Programas Lógicos

Asumimos que el lector está familiarizado con las nociones básicas de programación lógica [67]. Ejecutar un programa lógico P para un átomo A consiste en construir un árbol SLD para $P \cup \{A\}$ y extraer de él las sustituciones computadas para cada rama no fallida del árbol. La evaluación parcial se basa en dicho enfoque de ejecución, aunque con dos diferencias fundamentales:

- Para garantizar la terminación del proceso de *desplegado*, o "*unfolding*", es en ocasiones necesario no desplegar un objetivo, dejando por tanto una hoja en el árbol con un objetivo no vacío, y posiblemente no fallido. El árbol SLD resultante se dice que es un árbol *parcial*. Nótese que incluso si los árboles SLD para todas las posibles *consultas*, o "*queries*", son finitos, los árboles SLD que se construyen en el proceso de evaluación parcial pueden ser infinitos. Esto ocurre porque, al no disponerse en tiempo de EP de los valores concretos correspondientes a los argumentos dinámicos, el árbol de EP puede tener más ramas, en particular infinitas, que el árbol SLD de ejecución. Qué átomo seleccionar de cada objetivo y cuándo parar el proceso de desplegado queda determinado por la llamada *regla de desplegado*.

- El evaluador parcial, en general, tiene que construir varios árboles SLD para garantizar que todos los átomos que aparecen en las hojas de éstos queden *cubiertos* por la raíz de algún árbol (esto se conoce como la condición de *recubrimiento* de la EP [66]). El llamado *operador de abstracción* realiza *generalizaciones* en los átomos que han de ser evaluados parcialmente para evitar que se computen árboles SLD parciales para un número infinito de átomos. Cuando todos los átomos quedan cubiertos, entonces ya no hay necesidad de construir más árboles y el proceso termina.

La esencia de la mayoría de algoritmos de evaluación parcial de programas lógicos (ver por ejemplo [41]), aparece reflejada en el algoritmo de la Figura 2.2, el cual es paramétrico respecto a la regla de desplegado, unfold, y al operador de abstracción, abstract. La función EP toma un programa

```
1: function EP (P, A, S)
2:     S_0 := S;  i := 0;
3:     repeat
4:         L^pe := unfold(S_i, P, A);
5:         S_{i+1} := abstract(S_i, L^pe, A);
6:         i := i + 1;
7:     until S_i = S_{i-1}      % (modulo renaming)
8:     return codegen(L^pe, unfold);
```

Figura 2.2: Algoritmo genérico de EP de programas lógicos

P, un conjunto (posiblemente vacío) de anotaciones $\mathcal{A}$, y un conjunto inicial de llamadas S. En cada iteración, el llamado *control local* es llevado a cabo por la regla de desplegado unfold (Línea 4), la cual toma el conjunto actual de átomos S_i, el programa y las anotaciones, y construye un árbol parcial SLD para cada llamada en S_i. En el control global, llevado a cabo por el operador de abstracción abstract, cuando hay llamadas en las hojas de los árboles que no están aún *cubiertas*, el operador abstract las añade al nuevo conjunto de átomos a ser evaluado parcialmente. Para garantizar la terminación del proceso de control global, es decir, para asegurar que la condición $S_i = S_{i-1}$ se cumple, estos átomos deben *generalizarse* de forma adecuada.

La evaluación parcial de P con respecto a S es entonces extraída sistemáticamente del conjunto resultante de llamadas L^{pe} de la última iteración, en la llamada fase de generación código, codegen en L8. Se utiliza entonces la noción de *resultante* para generar las reglas de programa asociadas a cada derivación de la raíz a cada hoja del árbol SLD para el conjunto final L^{pe}. Dada una derivación SLD de $P \cup \{A\}$, con $A \in L^{pe}$ terminando en B y siendo θ la composición de los *unificadores más generales posibles* (ver [67]) de los pasos de derivación, la regla "$\theta(A)$:- B" se denomina un *resultante* asociado a dicha derivación. Una EP se define como el conjunto de resultantes (cláusulas) asociados a las derivaciones de los árboles SLD parciales para $P \cup L^{pe}$. El programa resultante se denomina habitualmente *programa especializado* o *programa residual*. La Sección 2 del Artículo 6 presenta más formalmente los fundamentos de evaluación parcial de programas lógicos.

2.1.1. Evaluación Parcial "Online" frente a "Offline"

Es bien sabido que tanto la calidad de los programas especializados como el tiempo que requiere el proceso de EP, dependen en gran medida de las estrategias de control utilizadas. Tradicionalmente se han considerado dos enfoques diferentes de EP, "*online*" y "*offline*". En la EP online, todas las decisiones de control se toman "al vuelo" durante la fase de especialización teniendo en cuenta la historia pasada. En el enfoque offline, todas las decisiones de control son tomadas antes de la propia fase de especialización. Éstas se basan en descripciones abstractas de los datos en lugar de en los datos concretos. Normalmente, la estrategia de control se representa por medio de anotaciones en el programa, las cuales constituyen en realidad el único criterio que controla la EP. Por ejemplo, en el control local, una anotación podría indicar explícitamente que un átomo no debe desplegarse. En el control global, las anotaciones normalmente especifican, para cada llamada, qué argumentos deben generalizarse (en este caso, reemplazarse por variables libres). Dichas anotaciones, en algunos casos son generadas automáticamente por un análisis de "binding-time" [31, 64], y en otros casos son proporcionadas, parcial o totalmente, por el usuario.

De acuerdo a esta clasificación, el algoritmo de EP que proponemos se puede considerar como un híbrido pues el conjunto de anotaciones $\mathcal{A}$ puede proporcionar información a los operadores de control, como pasa en la EP offline, y incluye reglas de control basadas en la historia de especialización, como pasa en la EP online. La principal ventaja del enfoque offline es que, una vez son tomadas todas las decisiones de control, la fase de EP es en principio mucho más simple y eficiente. Por otro lado, la EP online, aunque es en principio más ineficiente, es estrictamente más potente pues las decisiones de control pueden basarse en los valores concretos en lugar de en abstracciones de éstos. Por tanto, aunque los resultados obtenidos por un evaluador parcial offline pueden siempre replicarse usando uno online, en muchos casos, los resultados obtenidos usando EP online no pueden reproducirse usando EP offline.

En este trabajo estamos interesados en investigar hasta donde se puede llegar usando un enfoque online, interesándonos por tanto primeramente en obtener resultados precisos sin importarnos la eficiencia. De esta forma esperamos poder obtener decompilaciones de alta calidad que no podrían obtenerse utilizando EP offline. Más adelante, afrontaremos el reto de la

eficiencia del proceso tratando de no perder calidad. Como veremos, muchas de las lecciones aprendidas en ésta tesis son de interés tanto para el campo de la EP online como offline.

2.2. Retos en la Especialización de Intérpretes de Bytecode

Esta sección ilustra por medio de un ejemplo, los retos a los que hay que enfrentarse a la hora de especializar un intérprete de bytecode. La Figura 2.3 muestra un fragmento de un intérprete de bytecode implementado en Prolog. Se asume que el código de cada método del programa bytecode viene representado como un conjunto de hechos `bytecode/3` tal que, para cada par $pc_i : bc_i$ en el código del método m, se tiene un hecho `bytecode(`m`,`pc_i`,`bc_i`)`. El estado que el intérprete manipula es de la forma `st(Fr,FrStack)`, donde `Fr` representa el "frame" (o contexto) actual y `FrStack` la pila de frames (o pila de llamadas), implementada como una lista Prolog. Los frames son de la forma `fr(M,PC,OStack,LocalV)`, donde `M` representa el método actual, `PC` el contador de programa, `OStack` la pila de operandos y `LocalV` la lista de variables locales. El predicado `main/3`, dado el método a ser interpretado `Method` y sus argumentos de entrada `InArgs`, construye primeramente un estado inicial llamando al predicado `build_s0/3`, y llama después al predicado `execute/2`, devolviéndose el resultado en la variable `Res`, la cual representa la cima de la pila de operandos al final de la ejecución. A su vez, `execute/2` llama al predicado `step/3`, el cual produce `S'`, el estado inmediatamente después de ejecutar el correspondiente bytecode, y llama recursivamente al predicado `execute/2` con `S'` hasta que se encuentre una instrucción `return` con la pila de llamadas vacía en el estado. Por brevedad, sólo mostramos la definición del predicado `step/3` para una selección de instrucciones bytecode, y omitimos el código de algunos predicados auxiliares. En particular, no mostramos el código de `build_s0/3`, el cual fue explicado con anterioridad, `next/3`, que produce el siguiente contador de programa dado el actual, y `split_OS/4`, que divide la pila de operandos actual entre la lista de parámetros del método llamado y el resto.

La figura 2.4 muestra el programa bytecode que utilizaremos como ejem-

```
main(Method,InArgs,Res) :-
   build_s0(Method,InArgs,S0),
   execute(S0,Sf),
   Sf = st(fr(_,_,[Res|_],_),_)).

execute(S,S) :-
   S = st(fr(M,PC,_,_),[]),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(fr(M,PC,_,_),_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).

step(goto(PC),S,S') :-
   S = st(fr(M,_,OS,LV),FrS),
   S' = st(fr(M,PC,OS,LV),FrS).
```

```
step(push(X),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[X|OS],L),FrS).
...
step(invoke(M'),S,S') :-
   S = st(fr(M,PC,OS,LV),FrS),
   split_OS(M',OS,Args,OS''),
   build_s0(M',Args,
            st(fr(M',PC',OS',LV'),_)),
   S' = st(fr(M',PC',OS',LV'),
           [fr(M,PC,OS'',LV)|FrS]).
step(return,S,S') :-
   S = st(fr(_,_,[RV|_],_),
          [fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[RV|OS],LV),FrS).
```

Figura 2.3: Fragmento de un intérprete de bytecode

plo. Arriba, mostramos el código fuente Java y abajo el bytecode correspondiente. Nótese que el código fuente sólo se muestra para facilitar la comprensión del programa, pues el decompilador trabaja directamente sobre el bytecode. El ejemplo consiste en una serie de métodos que realizan diferentes cálculos aritméticos. El método `gcd` calcula el máximo común divisor, `abs` el valor absoluto y `fact` el factorial implementado de forma recursiva. El método `count` no tiene ningún significado especial, simplemente incrementa un contador, inicializado a 0, hasta que su valor llega al valor del argumento de entrada.

Para poder obtener una decompilación *efectiva*, es necesario disponer de estrategias de control (es decir, operadores unfold y abstract) lo suficientemente potentes como para librarse de la llamada *capa de interpretación*. Es por ello, que en nuestros primeros experimentos en el Artículo 1, utilizamos estrategias de control *agresivas* basadas en la *subsunción homeomórfica* [57, 61]. En el control local, entendemos por agresivas a aquellas reglas de desplegado capaces de expandir las derivaciones lo más posible siempre que no haya problemas de terminación. En el control global, nos referi-

```
int count(int n){
   int i = 0;
   while (i < n) i++;
   return i;}

int gcd(int x,int y){
   int res;
   while (y != 0){
      res = x%y; x = y;
      y = res;}
   return abs(x);}
```

```
int abs(int x){
   if (x < 0) return -x;
   else return x;
}

int fact(int x){
   if (x == 0)
      return 1;
   else
      return x*fact(x-1);
}
```

```
Method count
  0:push(0)
  1:store(1)
  2:load(1)
  3:load(0)
  4:ifge(3)
  5:inc(1,1)
  6:goto(2)
  7:load(1)
  8:return
```

```
Method gcd
  0:load(1)
  1:if0eq(11)
  2:load(0)
  3:load(1)
  4:rem
  5:store(2)
  6:load(1)
  7:store(0)
  8:load(2)
  9:store(1)
  10:goto 0
  11:load(0)
  12:invoke(abs)
  13:return
```

```
Method abs
  0:load(0)
  1:if0ge(5)
  2:load(0)
  3:neg
  4:return
  5:load(0)
  6:return
```

```
Method fact
  0:load(0)
  1:if0ne(4)
  2:push(1)
  3:return
  4:load(0)
  5:load(0)
  6:push(1)
  7:sub
  8:invoke(fact)
  9:mul
  10:return
```

Figura 2.4: Código fuente y bytecode del programa de ejemplo

mos a operadores de abstracción que generalizan en el menor número de situaciones posible sin hacer peligrar la terminación.

La Figura 2.5 muestra el programa decompilado obtenido utilizando el evaluador parcial disponible en el sistema `CiaoPP` [48]. Para estos experimentos preliminares, usamos la *subsunción homeomórfica* para controlar tanto el nivel local como el global. Mirando al código obtenido, podemos observar lo siguiente:

1. El evaluador parcial no ha sido capaz de decompilar satisfactoriamen-

```
main(count,[N],A) :-
    % out of memory error

main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% Full interpreter
```

Figura 2.5: Código decompilado para el ejemplo. Primer intento.

te el método `count`. Realmente se queda sin memoria por problemas de terminación, en este caso, tanto en el control local como en el global. El problema es que la *subsunción homeomórfica* no garantiza la terminación de la EP en programas que pueden generar potencialmente un número infinito de valores, como es el caso de nuestro intérprete de bytecode (en particular, por el uso del predicado `is/3`).

2. La composicionalidad respecto a métodos del programa original se ha perdido en el programa decompilado. Esto puede verse observando el código correspondiente al método `gcd`. Nótese que en la versión decompilada no aparece la llamada a `abs`, sino que el código correspondiente a éste aparece como "inline". Aunque esto puede considerarse como una característica positiva desde el punto de vista de la especialización, las consecuencias pueden degradar considerablemente la eficiencia y la calidad del proceso, y de hecho hace imposible el poder escalar al considerar programas reales que incluyan por ejemplo llamadas a librerías.

3. El código decompilado correspondiente al método `fact` contiene básicamente al intérprete completo y no aparece en la figura por motivos de espacio. Este problema fue primeramente detectado en [42] y aparece al tratar de decompilar por EP un método recursivo. Como veremos, este problema, así como la solución que propondremos más adelante, está muy relacionado con el problema de la composicionalidad explicado en el punto anterior.

4. Si miramos el código de los predicados `main(gcd,...)` y `execute/3`, podemos observar que hay duplicaciones de código. Y lo que es peor, el evaluador parcial produce estas duplicaciones porque parte del programa bytecode se reevalúa en el proceso de EP. En nuestra evaluación experimental demostraremos que el tener o no tener estas duplicaciones (reevaluaciones), hace la diferencia entre que el decompilador sea o no capaz de escalar en la práctica.

Las soluciones a estos problemas se resumen en los siguientes tres retos:

- **Reto I. Tratamiento de signaturas infinitas en la EP:** Estudiaremos en primer lugar las soluciones existentes identificando sus debilidades, y propondremos entonces la *subsunción homeomórfica basada en tipos*. Este asunto será discutido en la Sección 2.3 y elaborado con más detalle en los Artículos 3 y 4.

- **Reto II: Un esquema de decompilación modular.** Incluso después de solucionar el *Reto I*, veremos que es necesario diseñar un esquema de decompilación modular que preserve la composicionalidad respecto a métodos del programa original, y que además resuelva el problema de los programas recursivos. Dicho esquema de decompilación se introduce en la Sección 2.4 y se estudia con más nivel de detalle en el Artículo 6.

- **Reto III: Decompilación óptima.** En nuestros primeros experimentos utilizando el esquema de decompilación modular con programas reales, observaremos que aún no es posible escalar con éxito. Introduciremos entonces un esquema *óptimo* de decompilación que asegura que los tiempos de decompilación y los tamaños de los programas decompilados, crecen linealmente respecto al tamaño de los programas bytecode de entrada. Esto se consigue principalmente evitando las duplicaciones y reevaluaciones de código. Éste asunto se introduce en la Sección 2.5 y se estudia con más detalle en el Artículo 6.

2.3. Reto I: Tratamiento de Signaturas Infinitas en la EP

2.3.1. La Subsunción Homeomórfica

La relación de *subsunción homeomórfica*, "homeomorphic embedding" (HEm) [57, 60, 61], se ha convertido en una técnica muy popular para supervisar la terminación de métodos online de transformación y especialización, resultando esencial para obtener optimizaciones potentes, por ejemplo en el contexto de la EP online. Intuitivamente, el HEm es un orden estructural bajo el cual una expresión t_1 *subsume* a una expresión t_2, escrito como $t_2 \trianglelefteq t_1$, si t_2 puede obtenerse a partir de t_1 borrando algunos operadores. Por ejemplo, $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}} + \mathtt{W}) \underline{\times} (\underline{\mathtt{U}} \underline{+} \mathtt{s}(\underline{\mathtt{V}})))$ subsume a $\mathtt{s}(\mathtt{U} \times (\mathtt{U} + \mathtt{V}))$.

La relación HEm puede usarse para garantizar terminación gracias a la siguiente propiedad: si asumimos que el conjunto de constantes y functores es finito, toda secuencia infinita de expresiones $t_1, t_2, \ldots$, contiene al menos un par de elementos t_i y t_j con $i < j$ tal que $t_i \trianglelefteq t_j$. Por tanto, al computar iterativamente una secuencia $t_1, t_2, \ldots, t_n$, podemos garantizar su finitud utilizando el HEm como un "silbato". Cuando una expresión t_{n+1} va a añadirse a la secuencia, primero se chequea que se cumple $t_i \not\trianglelefteq t_{n+1}$ para todo i tal que $1 \leq i \leq n$. Intuitivamente, la computación puede progresar mientras la nueva expresión obtenida no sea mayor que (no subsuma a) ninguna de las expresiones previamente computadas, pues esto podría significar que estamos ante una secuencia infinita. El éxito del HEm se debe a que las secuencias pueden crecer considerablemente antes de que el silbato se active, en general más que con otras técnicas para garantizar terminación, lo cual suele significar una mayor efectividad de las transformaciones.

Aunque se ha demostrado que el HEm es una técnica potente para computaciones simbólicas, éste aún puede presentar algunas dificultades, en particular en presencia de signaturas infinitas. En el caso de programas lógicos, éstas pueden aparecer al utilizar algunos "builtins" de Prolog como `is/2`, `functor/3` y `name/2`. Se han definido variantes del HEm para tratar con signaturas infinitas (ver por ejemplo [60, 6]), sin embargo éstas tienden a ser demasiado conservadoras en la práctica.

2.3.2. Ejemplo Motivador

Consideremos el método `count` que aparece en la parte izquierda de la Figura 2.4. El método recibe un número entero, inicializa un contador a "0" (ver bytecodes `0` y `1`) y ejecuta un bucle que incrementa el contador en uno en cada iteración (bytecode `5`), hasta que el valor llega al valor del argumento de entrada (la condición se chequea en los bytecodes `2`, `3` y `4`). El método devuelve el valor del contador en los bytecodes `7` y `8`. Para decompilar el método `count`, evaluamos parcialmente el intérprete de la Figura 2.3 respecto al bytecode del método `count` empezando por el átomo `main(count,[N],I)`, donde `N` representa el argumento de entrada y `I` el valor de retorno (es decir, la cima de la pila al final de la computación).

En la Figura 2.6 se muestra (una versión reducida de) uno de los árboles SLD que da lugar a una decompilación efectiva del método `count`, y al que nos referiremos posteriormente. Para simplificar la comprensión, aparte del átomo de entrada `main/3`, solo mostramos los átomos correspondientes al predicado `execute/2`, pues es el único predicado recursivo del programa. Así, cada flecha en el árbol se corresponde realmente con varios pasos de derivación. Nótese que, algunas de las operaciones dentro del cuerpo de cada regla del predicado `step`, pueden quedar residuales al necesitar datos no conocidos en tiempo de EP. La regla de computación utilizada por el operador de desplegado es capaz de residualizar llamadas que no estén suficientemente instanciadas, y seleccionar así átomos del objetivo que no sean necesariamente los de más a la izquierda de forma segura [7], en particular, se seleccionarán llamadas a átomos `execute/2`. Representaremos estas llamadas residuales como etiquetas asociadas a las ramas del árbol.

Utilizando el HEm original

Consideremos primero un evaluador parcial que utiliza el HEm para controlar la terminación tanto en el control local como en el global. Como puede verse en la figura, el valor del `PC` "`2`" se corresponde con la entrada del bucle. Aplicando el HEm, la evaluación contiene una subsecuencia de átomos de esta forma: $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},1]),[]),\texttt{S}_\texttt{f})$, $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},2]),[]),S_f),\ldots$, la cual aparece marcada en la figura con rectángulos de línea discontinua. Dicha secuencia se corresponde con las sucesivas iteraciones con-

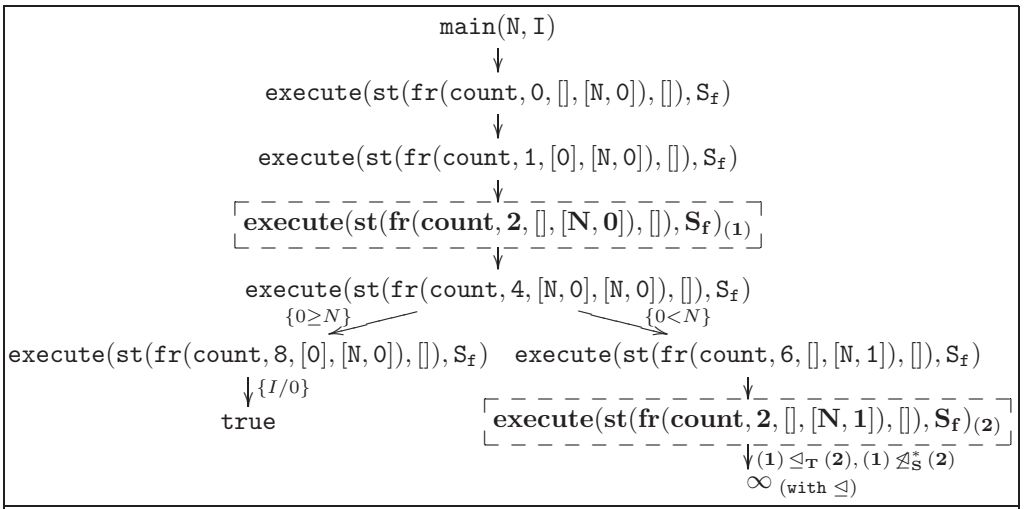


Figura 2.6: Árbol SLD de desplegado y código decompilado del ejemplo

secutivas del bucle, en las cuales el control vuelve a la cabeza de éste (ver el valor 2 en el valor del PC del estado), y el valor del contador (variable de la segunda posición de la lista) se va incrementando de uno en uno. Esta secuencia puede crecer de forma infinita, pues el HEm no la detecta como potencialmente peligrosa (ver "∞ (with $\trianglelefteq$)" en la figura). Esto ocurre debido al uso que hace el intérprete del operador de Prolog `is/2`, rompiéndose así la propiedad de finitud de signatura que se cumple en los programas lógicos puros.

Para obtener una decompilación de calidad, es necesario que el valor del contador (variable local 1) sea filtrado, pero no así el del PC. Como vemos en la figura, esto requiere parar la derivación cuando aparezca el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 1]), []), \texttt{S}_\texttt{f})$ (marcado como $(\mathbf{1}) \trianglelefteq_{\mathbf{T}} (\mathbf{2})$), y generalizarlo con respecto al átomo anterior encerrado en el rectángulo con línea discontinua, resultando así el átomo $\texttt{execute}(\texttt{st}(\ \texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, \texttt{X}]), []), \texttt{S}_\texttt{f})$.

Recuperando la terminación: Subsunción con filtrado de números

En programas que contienen predicados aritméticos de Prolog pero que no generan infinitos functores vía `functor/3`, `=../2`, etc., una solución inmediata para recuperar la terminación es utilizar la relación $\trianglelefteq_{num}$. Ésta, es una adaptación del HEm que simplemente filtra los valores numéricos, es decir, cualquier número subsume a otro número. En el ejemplo, el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},1]),[]),\ \texttt{S}_\texttt{f})$ subsume a $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[N,0]),[]),S_f)$ bajo $\trianglelefteq_{num}$, evitándose así la *no terminación*. Desafortunadamente, esta modificación del HEm, es demasiado simplista, y da lugar a una perdida excesiva de precisión. Por ejemplo, al especializar $\texttt{main}(\texttt{count},[\texttt{N}],\texttt{I})$, los primeros dos átomos de `execute/2` son $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},0,[],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$ y $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},1,[0],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$. Usando $\trianglelefteq_{num}$, el silbato se activa en este punto y el desplegado tiene que parar. Esto provoca que este último átomo sea generalizado en el control global produciéndose $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{X},\texttt{Y},[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$. Esto no es aceptable en el caso de la especialización de nuestro intérprete, pues se pierde la pista de la siguiente instrucción a ser ejecutada—lo que provoca que no se pueda eliminar la capa de interpretación—y de hecho, en la mayoría de casos provoca que el programa residual obtenido contenga prácticamente el intérprete por completo.

Incrementando la Precisión: Símbolos Estáticos del Programa

Una manera sintáctica de mejorar la precisión asegurando al mismo tiempo terminación, propuesta en [60], consiste en considerar dos conjuntos de símbolos: uno con aquellos que aparecen explícitamente en el programa y el objetivo, y otro con el conjunto infinito de símbolos que el programa puede generar potencialmente. Denotaremos esta relación como $\trianglelefteq^*_S$. Al comparar dos términos, nos quedamos con los símbolos que pertenezcan al conjunto finito y filtramos el resto. Bajo esta relación, el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},1,[0],\ [\texttt{N},0]),[]),\texttt{S}_\texttt{f})$ no subsume al átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},0,[],[\texttt{N},0]),[]),\texttt{S}_\texttt{f})$, pues los números `0` y `1` son símbolos estáticos diferentes del programa. Por tanto, en este caso, el evaluador parcial no se ve obligado a generalizarlos preservándose así el valor del `PC`.

Desafortunadamente, la relación $\trianglelefteq^*_S$ resulta no comportarse tampoco de forma óptima en nuestro caso, pues $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},2,[],[\texttt{N},1]),[]),\texttt{S}_\texttt{f})$

no subsume a $\texttt{execute}(\ \texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$. Esto significa que el proceso de desplegado continua con una segunda iteración del bucle. Aunque está garantizado que el proceso termina, se desplegarán tantas iteraciones del bucle como distintas constantes numéricas consecutivas aparezcan en el programa, en este caso 8. No será posible por tanto obtener la decompilación óptima que aparece en la parte de abajo de la Figura 2.6. Para obtener dicha decompilación, es necesario que el evaluador parcial generalice el contador del bucle lo antes posible, es decir, que el átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 1]), []), \texttt{S}_\texttt{f})$ subsuma a $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$.

Intuitivamente, la razón por la que esta relación no se comporta de forma óptima es porque ésta no es capaz de distinguir entre los distintos argumentos y los trata todos por igual. En resumen, este ejemplo sugiere que es necesario tener una relación de subsunción que sea capaz de tener información de contexto en cuenta: en particular, dicha relación dependiente del contexto, debería tratar de forma diferente el valor del PC y el valor de la variable del contador.

2.3.3. Subsunción Homeomórfica basada en Tipos

En presencia de signaturas infinitas, existe un método general para definir relaciones de subsunción homeomórfica; en [60] se define la *subsunción homeomórfica extendida* basada en resultados previos de Kruskal [57] y Dershowitz [37]. Esta solución define una familia de relaciones de subsunción, donde una relación subsidiaria de orden, definida sobre los símbolos de función del programa, juega un papel esencial. No obstante, veremos que ésta no resuelve realmente el problema en la práctica, pues no propone ningún mecanismo automático para encontrar la relación "correcta" entre los símbolos de función.

Esta tesis propone la *subsunción homeomórfica basada en tipos*, "type-based homeomorphic embedding" (TbHEm), una relación que mejora el HEm original haciendo uso de información adicional proporcionada en forma de tipos. Veremos como este enfoque puede verse como una forma de generar instancias concretas de la relación de HEm extendida como definió Leuschel, incluyendo la posibilidad de tener en cuenta la semántica del programa. Los tipos requeridos para guiar al TbHEm pueden darse manual-

mente o, lo que es más interesante, pueden inferirse automáticamente por análisis de programas, como discutimos en el Artículo 3.

La observación principal en al que se basa el TbHEm es que, incluso aunque una expresión este definida sobre una signatura infinita, es posible que sólo tome un conjunto finito de valores sobre el dominio correspondiente para cada computación. Para realizar dicha distinción, nuestra relación se define sobre tipos, los cuales se estructuran en una partición finita (posiblemente vacía) y una partición infinita (también posiblemente vacía). Intuitivamente, el TbHEm permite expandir secuencias mientras, al comparar subtérminos de un tipo infinito, los valores concretos que aparecen en la expresión se mantengan en la partición finita del tipo correspondiente.

Utilizando el TbHEm para controlar la EP del intérprete de bytecode

En el caso de nuestro intérprete de bytecode, el argumento del `PC` se puede definir por un tipo estructurado de forma que el intervalo acotado en el cual éste se mueve constituye su partición finita, y el resto de los números enteros forma su parte de infinita. De esta manera, el TbHEm no generalizará el `PC` mientras su valor permanezca dentro del intervalo acotado.

Para inferir este tipo, utilizaremos técnicas de análisis existentes, en particular, usaremos el análisis de *tipos buenos* ("well-typings") descrito por Bruynooghe *et al.* [20][1]. Éste infiere el siguiente tipo τ_{PC} para el contador de programa del intérprete de la Figura 2.3, teniendo en cuenta el programa bytecode de la Figura 2.4:

τ_{PC} `--> -4; 0; 1; 2; 3; 4; 5; 6; 7; 8; num`

Se puede interpretar que el tipo τ_{PC} consiste en una partición finita (las constantes numéricas) y una partición infinita (el resto de los números distintos de las constantes). Es decir, el tipo se puede interpretar como $\tau_{PC} \rightarrow F; I$ donde la partición F es $\{-4, 0, 1, 2, \ldots, 8\}$ y $I = \texttt{num} \setminus F$. Usando esta regla de tipo, el TbHEm asegura que el contador de programa nunca será abstraído durante la EP, mientras su valor se mantenga en el rango esperado (las constantes numéricas). El átomo $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 1, [0], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$ no subsume a

[1]Disponible on-line en `http://saft.ruc.dk/Tattoo/`

$\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 0, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f})$ usando esta definición de tipo, por tanto, la derivación puede avanzar. Esto evita la necesidad de generalizar el PC lo que provocaría que no pudiésemos obtener una especialización efectiva. La derivación bien terminará, o bien el valor del PC se repetirá por algún salto hacia atrás en el código (bucle). En este caso, el TbHEm, también escrito $\trianglelefteq_T$, detectará el átomo correspondiente como peligroso, por ejemplo, $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 0]), []), \texttt{S}_\texttt{f}) \trianglelefteq_\texttt{T} \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, 2, [], [\texttt{N}, 1]), []), \texttt{S}_\texttt{f})$, como vemos en la Figura 2.6.

El programa decompilado que obtenemos usando los tipos inferidos en combinación con el TbHEm se muestra en la parte baja de la Figura 2.6. Se puede observar que esta decompilación es óptima[2] en el sentido de que la capa de interpretación se ha eliminado totalmente o no aparece código residual superfluo.

Aparte de la inferencia de "well-typings" que hemos visto, en el Artículo 3 se bosqueja como utilizar un análisis de cotas numéricas para inferir información que puede ser útil para el TbHEm. Este tipo de análisis calcula sobreaproximaciones del conjunto de valores que los argumentos del programa pueden tomar. Intuitivamente, si podemos probar que dicho conjunto está acotado, entonces sabemos que la partición infinita del tipo es vacía, y por tanto podemos aplicar se forma segura el HEm tradicional (mejorando así la efectividad de la EP).

Nótese que, determinar el conjunto exacto de símbolos que pueden aparecer en tiempo de ejecución en un punto específico del programa, y en particular determinar si ese conjunto es finito, está estrechamente relacionado con el problema de la terminación, y es por tanto indecidible. Sin embargo, cuanto mejores sean los tipos, más agresiva será la EP sin sacrificar en ningún caso su terminación. Si los tipos derivados tienen particiones finitas demasiado pequeñas, entonces probablemente se producirán demasiadas generalizaciones resultando en una especialización muy pobre; mientras que si éstos son demasiado grandes, entonces la especialización tenderá a ser demasiado agresiva, produciendo posiblemente versiones innecesarias.

[2]Veremos después que ésta puede mejorarse

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% full int.
```

```
main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).

execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).
```

Figura 2.7: Código decompilado para el ejemplo después de superar el Reto 1

2.4. Reto II: Decompilación Modular

Una vez se ha superado el problema de las signaturas infinitas, la clase de programas bytecode que podemos decompilar con éxito es considerablemente más amplia. Otro aspecto importante, que no hemos discutido en esta introducción, es que utilizando el operador clásico de abstracción basado simplemente en el HEm original, o incluso mejorándolo con el TbHEm, los programas decompilados que obtenemos tienden a tener demasiadas versiones especializadas (redundantes) para algunos predicados. Este problema se estudia con detalle en el Artículo 2 donde se propone un operador de abstracción avanzado el cual es capaz de controlar la *polivarianza* del proceso de EP, es decir, es capaz de evitar tener dichas versiones especializadas redundantes. Como mostramos en el Artículo 2, esto nos permite obtener mejores decompilaciones, de forma más eficiente, lo que amplia aún más la clase de programas que podemos decompilar con éxito. No obstante, incluso mejorando nuestro evaluador parcial para que incluya tanto el TbHEm como este operador de abstracción mejorado, el esquema de decompilación resultante resulta aún insatisfactorio a la hora de decompilar programas reales, pues entre otras cosas, y como veremos, la composicionalidad del programa original respecto a las llamadas a métodos se pierde en la decompilación.

Consideremos de nuevo nuestro ejemplo de la Figura 2.4. El código decompilado que obtenemos usando el evaluador parcial mejorado se muestra en la Figura 2.7. Se puede observar que éste es básicamente el mismo de la Figura 2.5 excepto para el método `count`, para el que se obtiene ahora el código que mostramos en la parte baja de la Figura 2.6. Es importante hacer notar que este ejemplo no es suficientemente complejo como para poner en evidencia el problema de la polivarianza que el nuevo operador de abstracción del Artículo 2 resuelve. El lector interesado puede consultar el Artículo 2 donde se presenta un ejemplo representativo en este sentido.

A la vista del ejemplo, identificamos las siguientes cuatro limitaciones del esquema actual de decompilación (a partir de ahora llamado decompilación *no-modular*) denotadas como **(L1)**... **(L4)**. Nótese que dichas limitaciones, así como la forma de resolverlas que explicamos más adelante, son también relevantes para el caso de la decompilación por medio de EP puramente offline.

(L1) Los métodos aparecen "inlined" en los diferentes contextos en los que son llamados, lo que hace que se pierda la estructura original del código. Por ejemplo, la invocación a `abs` desde `gcd` (línea 12 de `gcd`) no aparece en el código decompilado. Como resultado, el código decompilado para `gcd` tiene dos casos *base* en los que aparecen "inlined" los correspondientes "builtins" de `abs`, es decir, `A>=0`, `B<0` y `A is -B`. Esto ocurre porque las llamadas a métodos se tratan de forma "small-step" en el intérprete, es decir, el código de los métodos invocados se despliega como se estuviese incluido dentro el método que lo invoca.

(L2) Como consecuencia, el proceso de decompilación es muy ineficiente cuando aparecen muchas llamadas a métodos. Por ejemplo, si se tienen n llamadas a un mismo método, éste será decompilado n veces. Incluso aún peor, si aparece una invocación a método dentro de un bucle, el código será decompilado en el caso mejor 2 veces, al tenerse que realizar la correspondiente generalización en el control global antes de llegar al punto fijo en la EP. Esto podría incluso ser peor en el caso de bucles anidados.

(L3) El esquema no-modular no trabaja de forma incremental, en el sentido de que no permite la decompilación *separada* de métodos sino que redecompila todas las llamadas. Por tanto, decompilar un lenguaje real es totalmente inviable, pues han de considerarse las librerías, cuyo código podría incluso no estar disponible. La limitación L2 junto con la L3

responden negativamente al punto *(a)* de la Figura 2.1.

(L4) El programa decompilado contendrá básicamente el intérprete completo cuando aparezcan métodos recursivos. Esta es la razón por la que en el programa decompilado de la Figura 2.7 no aparece el código correspondiente al método recursivo `fact`. El problema con la recursión es el siguiente. Asumamos que queremos decompilar el método recursivo $m1$ cuyo código es de la siguiente forma $\langle pc_0 : bc_0, \ldots, pc_j : invoke(m1), \ldots, pc_n : return\rangle$. Hay una primera decompilación para $A_k = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv}), []), \texttt{S}_\texttt{f})$ en la que la pila de llamadas es vacía. Al decompilarla, aparece una llamada de la forma $A_l = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}', \texttt{lv}'), [\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv})]), \texttt{S}_\texttt{f})$, con la pila de llamadas conteniendo la anterior llamada, al llegar a la llamada recursiva. En este punto, la derivación debe pararse pues $A_k \trianglelefteq_T A_l$. Para asegurar terminación, el control global generaliza estas llamadas a $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \_, \_), \_), \texttt{S}_\texttt{f})$, donde $\_$ denota una variable libre, siendo por tanto la pila de llamadas desconocida. Como consecuencia, al evaluar la instrucción *return*, la continuación obtenida de la pila de llamadas es desconocida produciéndose la llamada $\texttt{execute}(\texttt{st}(\texttt{fr}(\_, \_, \_, \_), \_), \texttt{S}_\texttt{f})$, que habrá de decompilada. El hecho de que el método y el contador de programa sean desconocidos provoca que sea imposible eliminar la capa de interpretación, y de hecho, el código decompilado contendrá potencialmente el intérprete al completo. Esta situación se da al decompilar el método `fact`. Las limitaciones L1 y L4 responden negativamente al punto *(b)* (ver Figura 2.1).

A continuación identificamos los ingredientes necesarios para definir un esquema de decompilación *modular*. Entendemos por decompilación *modular*, una decompilación en la que la unidad de procesamiento es el método, es decir, se decompila un método cada vez. Mostraremos como dicho esquema resuelve las cuatro limitaciones descritas de la decompilación nomodular y responde afirmativamente a los puntos *(a)* y *(b)* de la Figura 2.1. Básicamente necesitaremos: (i) Dar un tratamiento composicional a las invocaciones a método. Veremos que esto se puede conseguir considerando un intérprete implementado utilizando una semántica "big-step". (ii) Proporcionar un mecanismo para residualizar las llamadas del programa decompilado (es decir, no desplegarlas y añadirlas sin modificaciones al código residual). Generaremos automáticamente anotaciones en la EP

para este propósito. (iii) Estudiaremos las condiciones que aseguran que la decompilación *separada* de métodos es correcta.

2.4.1. Intérprete con Semántica "Big-step" para habilitar la Modularidad

Tradicionalmente, se han considerado dos enfoques distintos a la hora de definir la semántica de un lenguaje, la semántica "big-step" (o *natural*) y la "small-step" (o *operacional-estructural*), ver por ejemplo [58]). Básicamente, en una semántica "big-step" las transiciones relacionan los estas inicial y final para cada instrucción, mientras que en una "small-step" las transiciones definen el siguiente paso de ejecución para cada sentencia. En el contexto de los intérpretes de bytecode, ocurre que la mayoría de las instrucciones se ejecutan en un solo paso, haciendo que ambos enfoques sean prácticamente equivalentes. Este es el caso de nuestro intérprete de bytecode de la Figura 2.3 para todas las instrucciones excepto para *invoke*. La transición para *invoke* en la semántica "small-step" define el siguiente paso de la computación, es decir, el "frame" actual se apila en la pila de llamadas y se inicializa un nuevo "frame" para la ejecución del método invocado. Nótese que después de dar este paso, no es posible distinguir ya entre el código del método anterior y el llamado. Esto provoca que no podamos obtener modularidad en la decompilación.

En el contexto de la decompilación interpretativa de lenguajes imperativos, tradicionalmente se han utilizado intérpretes con semántica "small-step" (ver por ejemplo [76, 47]). En esta tesis sostenemos que el uso de intérprete con una semántica "big-step" es necesario para poder definir un esquema modular y poder así escalar al considerar lenguajes y programas reales. En la Figura 2.8, mostramos la parte relevante de la versión "big-step" del intérprete de bytecode de la Figura 2.3. Podemos observar que ahora, la instrucción *invoke*, una vez extraídos los parámetros de llamada de la pila de operandos, llama recursivamente al predicado `main/3` para ejecutar el método llamado. Al terminar la ejecución del método, el valor de retorno se apila de vuelta en la pila de operandos del nuevo estado y la ejecución procede normalmente. Por otro lado ya no es necesario llevar en el estado explícitamente la pila de llamadas, sino sólo la información de la ejecución actual, es decir, los estado son ahora de la forma

```
execute(S,S) :-
   S = st(M,PC,[_Top|_],_),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(M,PC,_,_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).
```

```
step(invoke(M'),S,S') :-
   S = st(M,PC,OS,LV),
   next(M,PC,PC'),
   split_OS(M',OS,Args,OSRs),
   main(M',Args,RV),
   S' = st(M,PC',[RV|OSRs],LV).
```

Figura 2.8: Fragmento del intérprete de bytecode "big-step"

`st(M,PC,OStack,LocalV)`. La pila de llamadas la mantendría ahora el propio Prolog por medio de las llamadas recursivas al predicado `main/3`.

El tratamiento composicional en cuanto a las llamadas a métodos no sólo es esencial para permitir la decompilación modular (solucionando así L1, L2 y L3) sino que también resuelve el problema con la recursión de una manera simple y elegante. De hecho, la decompilación usando el intérprete "big-step" ya no presenta la limitación L4. Por ejemplo, la decompilación de un método recursivo $m1$ empezaría por la llamada $\texttt{main}(\texttt{m1}, \_, \_)$ y llegaría entonces a $\texttt{main}(\texttt{m1}, \texttt{args}, \_)$ donde `args` representaría a los argumentos de la llamada recursiva. Esta llamada sería detectada como peligrosa por el control local y por tanto se pararía la derivación. A diferencia de lo que pasaba anteriormente, no es necesario una segunda evaluación pues la segunda llamada es necesariamente una instancia de la primera, y por tanto, no habrá ninguna perdida de información asociada con la generalización de la pila de llamadas.

Nótese que la idea de utilizar una semántica "big-step" para definir el intérprete y así conseguir una especialización modular es igual de necesario en el caso de la especialización de intérpretes por medio de EP offline. Más aún, esta idea es, por lo que sabemos, nueva y no había sido propuesta antes en ningún contexto, ni en la especialización online ni offline de intérpretes.

2.4.2. El Esquema de Decompilación Modular

Además de usar un intérprete "big-step", para poder diseñar un esquema de decompilación modular, es necesario: (1) proporcionar un mecanismo

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(gcd,[B,0],A) :-
    main(abs,[B],A).
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,C) :- main(abs,[A],C).
execute_1(A,B,F) :- B\=0,
                    H is A rem B,
                    execute_1(B,H,F).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[B],A) :- B\=0,
                    C is B-1,
                    main(fact,[C],D),
                    A is B*D.
main(fact,[0],1).
```

Figura 2.9: Código decompilado usando la decompilación modular

para residualizar llamadas en el programa decompilado (es decir, no desplegarlas y añadirlas sin cambios al código residual), y (2) definir la noción de decompilación *separada* y estudiar las condiciones que aseguran su corrección.

El Artículo 6 estudia con detalle estos aspectos y define un esquema de decompilación modular demostrando formalmente su corrección y completitud. También se demuestra que el esquema propuesto satisface el criterio de la *método-optimalidad*, que asegura que cada método es decompilado una sola vez.

La decompilación modular funciona básicamente de la siguiente manera: cuando se va a decompilar una invocación a método, aparece la llamada `step(invoke(m'),_,_)` durante el proceso de desplegado. Utilizando el intérprete "big-step" de la Figura 2.8, se generará una llamada de la forma `main(m',_,_)`. En este punto, habrá una anotación que indica al evaluador parcial que no debe desplegar la llamada y que debe sin embargo dejarla en el código residual sin modificaciones. Si `m'` es *interno* (es decir, está definido en el programa de entrada) se realizará (o ya se habrá realizado) su correspondiente decompilación, pues el esquema de decompilación asegura que la EP se efectúa para todos los métodos del programa.

La Figura 2.9 muestra el programa decompilado que se obtiene utilizan-

do el esquema de decompilación modular sobre nuestro ejemplo motivador. Se puede observar que la estructura del programa original respecto a las llamadas a métodos si se preserva ahora. Por ejemplo, puede verse como en la definición de `gcd` hay una llamada a `abs` como ocurre en el programa bytecode original. Mas aún, ahora si obtenemos una decompilación efectiva para el método recursivo `fact` donde la capa de interpretación se ha eliminado por completo. Concluimos así que todas las limitaciones expuestas anteriormente en esta sección se han resuelto satisfactoriamente.

2.5. Reto III: Un Esquema de Decompilación Óptima

Como mencionamos en la Sección 2.2, y como podemos ver mirando el código de la Figura 2.9, los programas decompilados obtenidos usando el esquema modular no son aún totalmente óptimos pues contienen duplicaciones de código. Ver por ejemplo el código de la parte derecha de las reglas que definen `main(gcd,...)` y `execute_1/3`. Estas duplicaciones normalmente se producen debido a que parte del código se reevalúa durante la fase de EP. Desafortunadamente, como veremos después estas duplicaciones y reevaluaciones crecen exponencialmente con el número de puntos de *divergencia* y *convergencia* respectivamente, y como veremos en la evaluación experimental, degradan mucho la eficiencia del proceso y la calidad del código decompilado. Un aspecto fundamental de esta tesis es estudiar si se puede obtener, por medio de la decompilación interpretativa, programas decompilados cuya calidad sea equivalente a la calidad de los programas obtenidos utilizando decompiladores dedicados (punto (c) de la Figura 2.1). Para poder obtener resultados comparables, tiene sentido que se usen heurísticas similares. El hecho de que los decompiladores habitualmente construyan siempre un *grafo del flujo de control*, "control-flow-graph" (CFG), hace pensar que aplicar una noción similar para controlar la EP de nuestro intérprete pueda resultar útil.

A continuación explicamos el problema a través de un ejemplo. Consideremos el método m_{bl} de la Figura 2.10. El código fuente a la izquierda, el bytecode relevante en el centro y su CFG a la derecha. Como es habitual, el CFG [1] consiste en bloques básicos que contienen una secuencia de

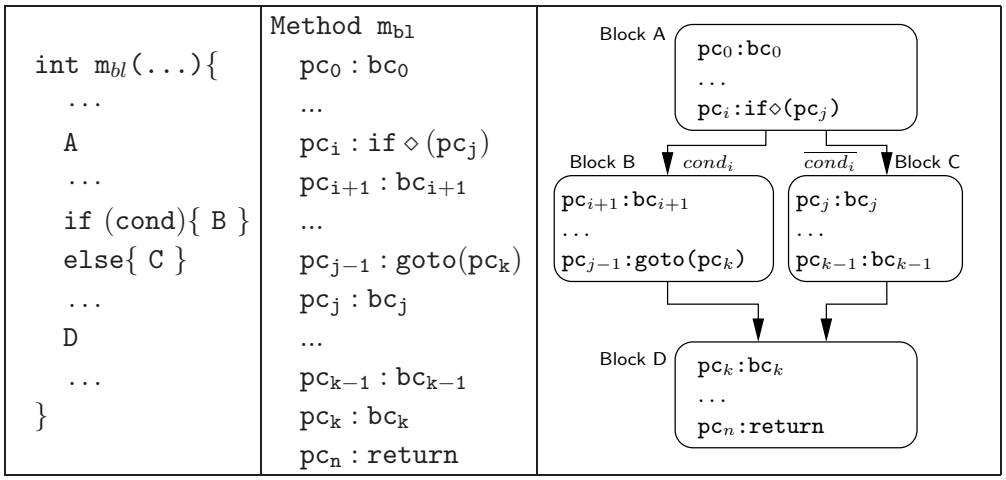


Figura 2.10: Código fuente, bytecode y CFG del método m_{bl}

instrucciones bytecode (sin bifurcaciones), los cuales están conectados por aristas las cuales describen los posibles flujos originados por las distintas instrucciones de bifurcación (como los saltos condicionales, las excepciones, las invocaciones virtuales, etc). En los programas que mostramos, éstas se corresponden simplemente con los saltos condicionales (es decir `if`$\diamond$ y `if0`$\diamond$). Un *punto de divergencia* (punto D) es un punto de programa (índice bytecode) del cual parten más de una rama; de forma similar, un *punto de convergencia* (punto C) es un punto de programa en el cual convergen dos o más ramas. En el CFG de m_{bl}, el único punto de divergencia (resp. convergencia) es pc_i (resp. pc_k).

Utilizando el esquema de decompilación actual se obtendría el árbol SLD de desplegado que aparece en la Figura 2.11, en el que todas las llamadas se han desplegado por completo al no haber ningún riesgo de terminación. El código decompilado correspondiente se muestra en la propia figura bajo el árbol. Usamos $\{\texttt{res}_\texttt{X}\}$ para referirnos al código emitido para el bloque BlockX y $\texttt{cond}_\texttt{i}$ para referirnos a la condición asociada a la instrucción de bifurcación en el índice $\texttt{pc}_\texttt{i}$ ($\overline{\texttt{cond}_\texttt{i}}$ denota su negación). La calidad del código decompilado no es óptima debido a lo siguiente:

D. El código decompilado $\{\texttt{res}_\texttt{A}\}$ para BlockA aparece duplicado en ambas reglas. Durante la EP, este código se ha evaluado sólo una vez pero, debido a la forma en la que se definen los resultantes (ver Sección 2.1), cada regla contiene el código decompilado asociado a la

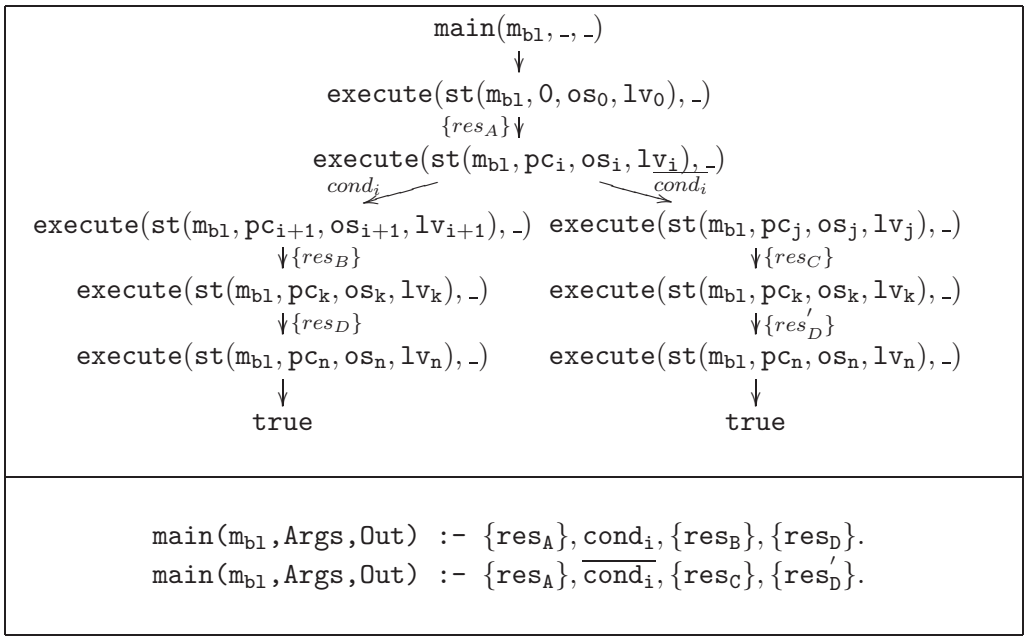


Figura 2.11: Árbol SLD de desplegado y código decompilado para m_{bl}

rama completa correspondiente del árbol. Este tipo de duplicación de código tiene dos consecuencias importantes: aumenta considerablemente el tamaño de los programas decompilados y provoca que su ejecución sea más lenta. Por ejemplo, cuando $\overline{\texttt{cond}_\texttt{i}}$ se cumple, la ejecución habría de pasar necesariamente a través de $\{\texttt{res}_\texttt{A}\}$ de la primera regla, fallaría al evaluar $\texttt{cond}_\texttt{i}$, y probaría después con la segunda regla.

C. El código decompilado para BlockD se emite de nuevo más de una vez. Cada regla del programa contiene ahora una versión (posiblemente diferente), $\{\texttt{res}_\texttt{D}\}$ y $\{\texttt{res}_\texttt{D}'\}$, para el código obtenido al evaluar el bloque BlockD. Ahora, en tiempo de EP, el código de BlockD se evalúa en el contexto de $\{\texttt{cond}_\texttt{i}, \{\texttt{res}_\texttt{B}\}\}$ y se vuelve a evaluar después en el contexto de $\{\overline{\texttt{cond}_\texttt{i}}, \{\texttt{res}_\texttt{C}\}\}$. Por tanto, debido a los puntos de convergencia, tanto la eficiencia del proceso como la calidad de la decompilación se ven seriamente perjudicados.

La cantidad de código residual repetido crece exponencialmente con el número de puntos C y D y la cantidad de código reevaluado crece exponen-

cialmente con el número de puntos C. Tratamos a continuación de definir un esquema de decompilación *óptimo*, y *a nivel de bloques*, que resuelva los problemas D y C. Intuitivamente, una decompilación a nivel de bloques debe producir código residual para cada bloque del CFG. Esto puede conseguirse básicamente haciendo que los árboles SLD de desplegado se correspondan con cada bloque, no expandiéndolos más después de un final de bloque. Nótese que esta idea va en contra de la filosofía habitual de la EP, donde normalmente, para maximizar la cantidad de información estática propagada, se suele tratar de expandir las secuencias lo máximo posible y parar el proceso de desplegado sólo cuando se ponga en peligro la terminación.

Este comportamiento puede conseguirse fácilmente en nuestro esquema simplemente proporcionando anotaciones de forma que se fuerce al proceso de desplegado a parar cuando aparezca en la secuencia un átomo `execute/2` cuyo *PC* se corresponda con un punto D. En el ejemplo, el desplegado debería parar en el punto pc_i. En cuanto al problema C, un requerimiento adicional es que los bloques que comiencen en puntos C deben ser evaluados parcialmente una sola vez. Esto básicamente se puede conseguir de la siguiente manera: (1) parando las derivaciones en las llamadas `execute/2` cuyo *PC* se corresponda con un punto C, y (2) pasando la llamada al control global, y asegurando que ésta se evalúa en un contexto suficientemente generalizado de forma que se cubran todos los posibles contextos en los que se evalúa dicha llamada. El primer punto se asegura proporcionando al evaluador parcial las correspondientes anotaciones, mientras que el segundo se asegura incluyendo en el conjunto inicial de átomos pasado a la EP, una llamada generalizada de la forma $\texttt{execute}(\texttt{st}(\texttt{m}_{\texttt{bl}}, \texttt{pc}_{\texttt{k}}, \_, \_), \_)$ para cada punto C. Obsérvese que, tanto las anotaciones, como el conjunto inicial de llamadas pueden calcularse automáticamente simplemente haciendo dos pasadas sobre el programa bytecode (ver por ejemplo [2, 84]).

El código resultante utilizando el esquema de decompilación óptima para el método m_{bl} se muestra en la Figura 2.12. Ahora, el código residual asociado a cada bloque aparece una sola vez, asegurando que se preserva la forma del CFG, como hacen los decompiladores dedicados. Conseguimos así obtener programas cuya calidad es equivalente a la obtenida usando decompiladores dedicados [2, 69], pero preservando las ventajas de la decompilación interpretativa.

```
main(m_bl,Args,Out) :- {res_A}, execute_1(...).
execute_1(...) :- cond_i, {res_B}, execute_2(...).
execute_1(...) :- ¬cond_i, {res_C}, execute_2(...).
execute_2(...) :- {res_D}.
```

Figura 2.12: Código decompilado óptimo para el método m_{bl}

Estos aspectos se estudian en detalle en el Artículo 6 donde se define formalmente dicho esquema de decompilación. Se demuestra también formalmente que el esquema propuesto satisface el criterio de la *bloque-optimalidad*, que asegura que: (I) el código residual para cada instrucción se emite una sola vez en el código decompilado, (II) cada instrucción bytecode se evalúa como máximo una vez durante la EP, y (III) hay como máximo una regla residual por cada bloque del programa bytecode.

2.5.1. Conclusiones de la Decompilación Óptima

Una vez se tiene en cuenta la observación principal de la Sección 2.4 de que el intérprete se debe escribir usando una semántica "big-step", cada una de las condiciones del criterio de la *bloque-optimalidad* puede resultar más o menos complicada de asegurar dependiendo de la estrategia de control local utilizada. Por ejemplo, si empezamos con un decompilador modular como el expuesto en la Sección 2.4, la condición (III) se cumplirá en general, pero no así las condiciones (I) ni (II) pues la regla de control local tiende a sobreespecializar las llamadas, lo que resulta en duplicaciones y reevaluaciones de código.

Por otro lado, si se utilizara un evaluador parcial offline, la regla de control local natural residualizaría todas las llamadas a `execute`, y filtraría en el control global toda la información del estado excepto la signatura del método y el contador de programa. Esta estrategia de control garantiza trivialmente las condiciones (I) y (II), pues asegura que cada instrucción bytecode se decompila de forma independiente. Sin embargo, tiende a ser demasiado conservadora, y, en particular, no satisface la condición (III), pues tan pronto como se encuentre con un bloque que tenga más de una instrucción bytecode (lo que ocurre casi siempre), el programa especializado

generará una regla para cada instrucción bytecode del bloque. Como resultado, el programa residual obtenido es de *alto nivel* en el sentido de que está escrito en Prolog. No obstante, su estrategia de control está altamente influenciada por el hecho de que estamos decompilando desde un programa bytecode (y no por ejemplo desde un programa fuente Java), y el programa obtenido no se parece para nada al programa Prolog que un programador Prolog podría escribir para realizar la misma tarea. Pues uno de los objetivos importantes de la decompilación es facilitar su comprensión y su análisis, en esta tesis sostenemos que los programas que cumplen el criterio de la *bloque-optimalidad*, y en particular aquellos que cumplen la condición (III), como los que se generan usando nuestro esquema de decompilación óptima, son más fáciles de tratar.

Otra observación importante es que los costosos mecanismos utilizados para controlar la EP, usados anteriormente para obtener los resultados de las Secciones 2.3.3 y 2.4, en particular el TbHEm y el control avanzado de polivarianza del Artículo 2, ya no son necesarios al utilizarse el esquema de decompilación óptima. Se pueden ahora usar los siguiente operadores triviales de control: unfold despliega todas las llamadas excepto aquellas que se correspondan con una anotación, y abstract añade al conjunto S_{i+1} todas las llamadas en L^{pe} que no sean una instancia de ninguna llamada en S_i (ver el algoritmo genérico de la Sección 2.1). Se puede demostrar fácilmente que la terminación está garantizada, tanto a nivel local como global gracias a las anotaciones y al conjunto inicial de átomos proporcionados en la EP.

2.6. Implementación y Resultados Experimentales

El Artículo 6 discute varios detalles de implementación y realiza una evaluación experimental exhaustiva de los diferentes esquemas de decompilación propuestos. Hemos llevado a cabo dos implementaciones distintas de un decompilador de Java Bytecode (secuencial) a Prolog. En la primera, hemos extendido un evaluador parcial online existente, aquel integrado en el sistema `CiaoPP`. Éste es un evaluador parcial muy potente y implementa reglas de desplegado y operadores de abstracción. Esto nos ha permitido comparar los diferentes esquemas de decompilación, y en particular

comparar respecto a los esquemas no óptimos. Sin embargo, la sobrecarga introducida al utilizar una herramienta tan potente y genérica no nos permite competir, respecto a eficiencia, con decompiladores dedicados. Por ello, hemos llevado a cabo una segunda implementación para la cual hemos escrito un evaluador parcial autocontenido que sólo contiene las estrategias de control necesarias para el esquema óptimo. Éste evaluador parcial ha sido integrado en una herramienta de decompilación, llamada jbc2prolog, la cual incluye también un intérprete de Java Bytecode. Esto ha hecho posible obtener decompilaciones óptimas y al mismo tiempo competir en términos de eficiencia respecto a decompiladores dedicados. El Artículo 6 realiza una comparación exhaustiva frente al decompilador del sistema COSTA [5] y al decompilador de Java JDec [14].

Ambas implementaciones consideran el lenguaje Java Bytecode (secuencial) al completo. Las extensiones necesarias para poder tratar los aspectos del lenguaje no tratadas en esta introducción se discuten en el Artículo 6. Éstas incluyen a las excepciones, operaciones del heap, invocaciones virtuales, decompilación al nivel de clases, etc. Todas ellas han sido fácilmente integradas en nuestro esquema de decompilación, en la mayoría de los casos, simplemente las funcionalidades correspondientes en el intérprete de bytecode.

Para la evaluación experimental del Artículo 6, hemos utilizado el conjunto estándar de "benchmarks" JOlden [54]. En particular, nos hemos interesado en: a) demostrar empíricamente la escalabilidad del enfoque, y b) comprobar la eficiencia de la herramienta implementada respecto a otros decompiladores. Concluimos lo siguiente:

- **Escalabilidad:** Mientras que en la decompilación no-óptima los tiempos de decompilación y los tamaños de los programas decompilados crecen de forma muy significativa con el tamaño de los "benchmarks", esto no ocurre en el esquema óptimo. Con la decompilación óptima, estos valores se mantienen totalmente estables. Mostramos que tanto los tiempos de decompilación como los tamaños de los programas decompilados crecen de forma *lineal* con respecto al tamaño de los programas bytecode de entrada, demostrando así la escalabilidad de la decompilación óptima.

- **Eficiencia:** Para demostrar la eficiencia de nuestro esquema, hemos

comparado los tiempos de decompilación obtenidos usando nuestra herramienta jbc2prolog frente a aquellos obtenidos usando el decompilador del sistema COSTA, y , a aquellos obtenidos con el decompilador JDec [14]. Podemos concluir que los resultados son competitivos respecto a aquellos obtenidos con decompiladores dedicados. En particular, observamos que son bastante similares a los obtenidos con COSTA. Mas aún, en la mayoría de ejemplos, podemos observar que jbc2prolog es cerca de diez veces más rápido que JDec. Nuestra conclusión en este sentido es que es muy difícil comparar decompiladores escritos en diferentes lenguajes de programación y más aún que decompilan a diferentes lenguajes.

2.7. Trabajo Relacionado

Los trabajos previos sobre decompilación interpretativa se han centrado básicamente en demostrar que el enfoque es viable para pequeños y medianos intérpretes y lenguajes. Principalmente han tratado de demostrar su *efectividad*, es decir, que la llamada capa de interpretación se puede eliminar de los programas compilados. Para ello se han usado técnicas de EP offline [62], online [47, 76] y híbridas [63]. Esta tesis se ha centrado en, primeramente, demostrar la viabilidad del enfoque para un lenguaje bytecode con orientación a objetos, para después estudiar cuestiones más avanzadas como su escalabilidad y la calidad de las decompilaciones, las cuales no se habían estudiado hasta ahora. Los trabajos sobre decompilación interpretativa ya se han ido comparando en las diferentes secciones del capítulo y en la introducción. Revisamos a continuación el trabajo relacionado en el campo de la decompilación.

Se puede realizar decompilación a diferentes niveles, con sus correspondientes grados de precisión y éxito. El caso más complicado es sin duda la decompilación de *ejecutables binarios*. Hay una serie de complicaciones como por ejemplo la dificultad a la hora de recuperar el flujo de control. Un problema intrínseco es la imposibilidad de distinguir entre el código de los datos de forma estática. Ver por ejemplo [25, 80] y sus referencias donde se discuten los problemas y las técnicas que se aplican en la decompilación de ejecutables binarios. El siguiente nivel es la decompilación de *código ensamblador* [26]. En este contexto, muchas de las complicaciones de la

decompilación de binarios se siguen presentando, aunque al menos se suele poder separar el código de los datos. Un nivel más arriba se encontraría la decompilación de código a ser ejecutado por una máquina virtual, como el bytecode. Esto es en general más sencillo, pues las máquinas virtuales son generalmente más sencillas que las arquitecturas hardware. Además, estos programas suelen cumplir varias restricciones, como que por ejemplo sean *verificables* [59] o que los tipos de las variables estén disponibles. Como resultado, en el caso particular de la decompilación de Java Bytecode, hay un buen número de herramientas de decompilación capaces de tratar una amplia clase de programas bytecode, especialmente aquellos generados por compiladores de Java, por ejemplo `javac`. No obstante, las cosas se pueden complicar bastante cuando el java Bytecode se ha generado con un *obfuscador*, y especialmente cuando se ha utilizado un compilador optimizante o un compilador de un lenguaje distinto de Java como Haskell, ML, Ada, etc. Ver por ejemplo [71] y sus referencias para una discusión más detallada sobre decompiladores de Java Bytecode y las dificultades a las que éstos se enfrentan.

Como hemos mencionado anteriormente, existen varios analizadores de Java Bytecode que transforman el bytecode en algún tipo de representación intermedia de más alto nivel, y por tanto pueden verse como decompiladores dedicados. En particular, los sistemas COSTA [5] y `CiaoPP` [48] convierten bytecode a una representación que es usada después como entrada para la fase de análisis. Aunque en ambos casos la representación utilizada es similar, en el caso de COSTA se formaliza como una representación basada en reglas [2], mientras que en `CiaoPP` se formaliza como cláusulas de Horn, es decir, como un programa lógico [69]. Esto se hace en `CiaoPP` para así poder usar directamente los análisis disponibles para programas CLP de `CiaoPP`.

Hay sin embargo una diferencia crucial entre los programas generados en [69] o [5], y los generados por nuestro decompilador. Mientras que los programas de [69] y [5] están exclusivamente pensados para ser analizados, no siendo por tanto ejecutables, los programas que nosotros generamos pueden tanto ser analizados como ejecutados. La razón de esto es que los primeros, aunque representan el flujo de control de los programas bytecode con reglas, dejan las instrucciones bytecode como "builtins", es decir, predicados predefinidos, que el análisis posteriormente interpreta. Producir programas ejecutables es algo no trivial, pues muchas de las instrucciones

bytecode operan con el heap de una forma u otra. Por tanto, para conseguir una decompilación CLP ejecutable, se debe introducir el heap de la JVM explícitamente en el programa CLP. Esto se consigue de forma automática en nuestro enfoque.

Capítulo 3

Aplicaciones de la Decompilación Interpretativa

Como ya se mencionó en el capítulo anterior, una de las ventajas importantes de la decompilación interpretativa es que los programas que obtenemos son totalmente *ejecutables*, lo que amplia su campo de aplicación. En este capítulo, resumimos dos estudios experimentales llevados a cabo que tratan de aprovechar dicha ventaja de los programas decompilados:

1. Análisis de programas bytecode analizando sus decompilaciones a LP utilizando herramientas de análisis de LP. Este punto se elabora en el Artículo 1.

2. *Generación de datos de prueba* para programas bytecode por medio de evaluación parcial en CLP. Esto se estudia con detalle en el Artículo 7.

3.1. Análisis de Bytecode utilizando Herramientas de Análisis LP

Analizar programas en el paradigma de la programación lógica ofrece una serie de ventajas, quizás la más importante sea la madurez y sofisticación de las herramientas de análisis ya disponibles en dicho contexto. En particular, el sistema `CiaoPP`, aparte de proporcionar el potente evaluador parcial que hemos utilizado para realizar parte de los experimentos en el capítulo anterior, proporciona un motor genérico de análisis con un buen

número de dominios abstractos disponibles. Esto permite inferir una gran cantidad de propiedades de los programas lógicos como *terminación*, cotas en el consumo de recursos, *tipos* y *modos*, *ausencia de errores*, etc.

Uno de los objetivos de esta tesis ha sido investigar la posibilidad de reutilizar herramientas existentes en el paradigma de la CLP, en particular `CiaoPP`, para analizar programas bytecode a base de analizar sus decompilaciones a LP. Esto permitiría diseñar un esquema de análisis y verificación de programas bytecode en el cual la potencia de las herramientas de análisis de CLP se transfiere automáticamente al análisis y verificación de programas bytecode. Esta misma idea había sido aplicada para analizar versiones muy reducidas de lenguajes imperativos de alto nivel [76] y también código ensamblador del procesador PIC [46], un microprocesador de 8 bits. Sin embargo, por lo que conocemos, esta es la primera vez que este enfoque se ha aplicado con éxito para un lenguaje imperativo bytecode, real y de propósito general.

Dicho estudio se presenta en el Artículo 1, donde: (1) proponemos dicho esquema para el análisis y verificación de programas bytecode (en particular para Java Bytecode), y (2) se realizan una serie de experimentos utilizando el sistema `CiaoPP` demostrando así la viabilidad práctica del enfoque propuesto. En resumen, el Artículo 1 muestra como, razonando sobre nuestros programas decompilados, utilizando para ello los análisis disponibles en el sistema `CiaoPP`, podemos demostrar automáticamente propiedades no triviales de los programas bytecode como terminación, ausencia de errores en tiempo de ejecución e inferir cotas en el consumo de recursos. Por ejemplo, para demostrar ausencia de errores en tiempo de ejecución, proponemos instrumentar un intérprete de bytecode, que además de computar el valor de retorno del método llamado, también calcula la *traza de ejecución*, la cual captura la historia de la ejecución. Dichas trazas representan los pasos semánticos dados, y por tanto no sólo representan las instrucciones ejecutadas, sino que representan también cierta información de contexto. Éstas nos permiten distinguir, para una misma instrucción bytecode, si el correspondiente paso lanza una excepción o se ejecuta normalmente. Por ejemplo, `invoke_step_ok` y `invoke_step_NullPointerException` representan respectivamente la llamada a método *normal* y la llamada sobre una referencia *null* que lanza una excepción. Esta flexibilidad adicional de la decompilación interpretativa nos ha permitido demostrar ausencia de

errores en tiempo de ejecución de una forma casi directa, simplemente especificando la propiedad de "error-free" basada en las trazas. Una traza es "error-free" si no contiene ningún paso de excepción, o si no termina lanzando una excepción. Esto, en cualquier caso, dependerá de la *política de seguridad* utilizada para la propiedad "error-freeness". De nuevo, nuestro enfoque demuestra su flexibilidad en este punto, pues se pueden definir fácilmente diferentes políticas, simplemente especificando la propiedad correspondiente en `CiaoPP`.

3.2. Generación de Datos de Prueba por EP en CLP

Una característica especial de nuestros programas decompilados es que éstos representan el estado *completo* del programa, a diferencia de otros enfoques [69, 2, 84] Hasta ahora, la principal motivación de decompilar bytecode a LP había sido el ser capaz de realizar análisis estático sobre los programas decompilados, para poder así obtener propiedades de los programas bytecode. Si la decompilación produce programas LP que son ejecutables, entonces dichos programas decompilados pueden usarse no solo para ser analizados estáticamente, sino también para análisis dinámico y ejecución. Nótese que no siempre ocurre esto, pues en otros enfoques (como [4, 69]) las transformaciones están exclusivamente pensadas para el análisis estático, y por tanto los programas no pueden ejecutarse. Una aplicación novedosa de la decompilación interpretativa que proponemos en esta tesis es la *generación automática de datos de prueba*.

La *generación de datos de prueba*, *test data generation* (TDG), trata de generar automáticamente casos de prueba para un determinado *criterio de recubrimiento*. El criterio de recubrimiento mide lo que se ejercita el programa por el conjunto dado de casos de prueba. Ejemplos de criterios de recubrimiento son: *recubrimiento de instrucciones*, que requiere que cada instrucción del programa se ejecute; *recubrimiento de caminos*, que requiere que cada posible traza del código se ejercite, etc. Existe una gran variedad de enfoques para realizar TDG (ver [91] para un resumen). Nuestro trabajo se centra en TDG de tipo "glass-box", en el que los casos de prueba se generan a partir del programa concreto, en lugar de generarse a partir de

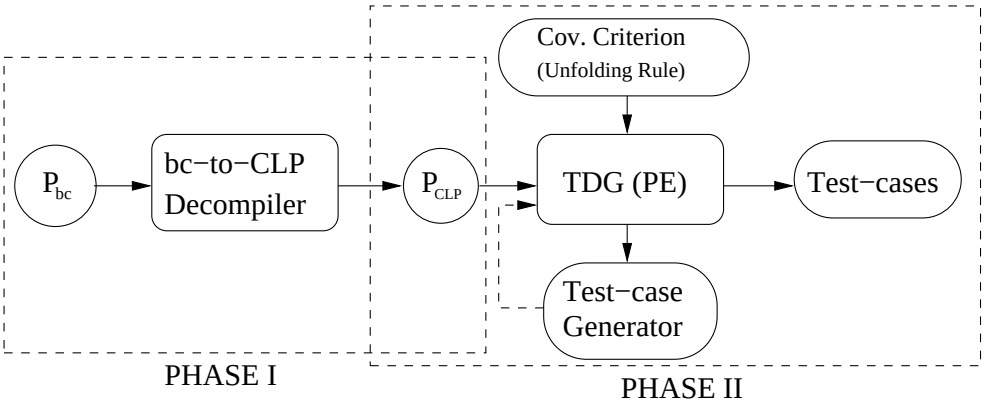


Figura 3.1: Visión general del enfoque de TDG de bytecode por EP en CLP

una especificación de éste. Además, nos centraremos en TDG estático, en el cual no se asume ningún conocimiento de los datos de entrada, a diferencia de los enfoques dinámicos [38, 45], en los cuales el programa es en algún momento ejecutado con valores de entrada concretos.

El enfoque estándar para generar casos de prueba estáticamente consiste en realizar una *ejecución simbólica* del programa (ver por ejemplo [28]), donde los contenidos de las variables son expresiones en lugar de valores concretos. La ejecución simbólica produce finalmente un sistema de *restricciones*, las cuales definen las condiciones bajo las que se ejecutan los distintos caminos. Esto ocurre por ejemplo, en las instrucciones condicionales, como en los "if-then-else", donde se suele requerir generar casos de prueba para las dos alternativas, y por tanto se han de acumular las condiciones de cada camino como restricciones. Para el caso de Java Bytecode, en [72] se diseñó una JVM simbólica (SJVM), la cual integraba varios resolutores de restricciones. La SJVM requiere una serie de extensiones no triviales respecto a la JVM: (1) necesita que el bytecode sea ejecutado simbólicamente como explicamos anteriormente, y (2) debe ser capaz de realizar "backtracking", pues al no conocerse los datos de entrada, el motor de ejecución debe considerar todas las alternativas. El mecanismo de "backtracking" utilizado en [72] es de hecho esencialmente el mismo al utilizado en la programación lógica.

Esta tesis propone un esquema novedoso de TDG de bytecode basado en técnicas de EP en CLP, el cual, a diferencia de trabajos previos, no requiere

el desarrollo de una máquina simbólica virtual. La Figura 3.1 muestra un diagrama con el esquema general. Como podemos ver, el enfoque se basa en dos fases independientes de EP en CLP, que consisten básicamente en lo siguiente:

1. *Decompilación del bytecode en un programa CLP.* Ya hemos explicado en el Capítulo 2 que la decompilación de bytecode a LP se puede obtener automáticamente por medio de la EP de LP, o alternativamente utilizando un decompilador dedicado [69]. Las modificaciones en el esquema para obtener programas CLP en lugar de programas LP son prácticamente triviales. Esto se puede conseguir básicamente transformando los "builtins" aritméticos del intérprete por los correspondientes "builtins" CLP de la librería correspondiente.

2. *Generación de casos de prueba.* Ésta es una aplicación novedosa de la EP que nos permite generar *generadores de casos de prueba* a partir de los programas CLP decompilados. En este caso, utilizaremos un evaluador parcial CLP capaz de propagar restricciones de la misma manera que haría una máquina simbólica. Los operadores de control de la EP juegan un papel esencial: (1) El control local permite capturar fácilmente diferentes criterios de recubrimiento. (2) El control global permite la generación de *generadores de casos de prueba*. Intuitivamente, éstos son programas CLP cuya ejecución en CLP devuelve más casos de prueba bajo demanda, sin la necesidad de empezar el proceso de TDG de nuevo.

Esta tesis sostiene que este enfoque de TDG de bytecode tiene varias ventajas importantes respecto a enfoques previos basados de alguna u otra manera en ejecución simbólica. Éstas incluyen: (i) Es más *genérico*, pues la mismas técnicas se pueden aplicar para otros lenguajes (imperativos) de entrada. En particular, una vez se la realizado la decompilación a CLP, las características del lenguaje las características del lenguaje quedan abstraídas, siendo por tanto la fase de generación de datos de prueba totalmente *independiente del lenguaje*. Esto evita tener que tratar con aspectos como la recursión, las llamadas a procedimientos, la memoria dinámica, etc. (ii) Es más *flexible* pues es muy fácil incorporar diferentes criterios de recubrimiento simplemente proporcionando las correspondientes reglas de control local a la EP. (iii) Es más potente gracias a la característica

de poder generar generadores de casos de prueba. (iv) Es más simple de implementar pues no requiere el desarrollo de ninguna máquina simbólica, asumiendo claro que se dispone de una evaluador parcial.

Como se acaba de mencionar en la ventaja (iv), una de las ventajas de los programas decompilados CLP respecto a sus versiones bytecode es que se puede realizar una ejecución simbólica de éstos sin necesidad de escribir un mecanismo específico de ejecución simbólica. Simplemente podemos ejecutar el programa decompilado usando el mecanismo de ejecución estándar de CLP, poniendo variables en todos los argumentos del correspondiente predicado. Por ejemplo, para nuestro ejemplo motivador de la Figura 2.4, podríamos ejecutar simbólicamente el método `gcd` lanzando el objetivo $\texttt{main}(\texttt{gcd}, [\texttt{X}, \texttt{Y}], \texttt{Z})$ sobre el programa decompilado. Los resultados obtenidos (restricciones sobre las variables) se pueden interpretar como las condiciones que han de cumplir las variables de entrada (en este caso X e Y) para seguir el camino de ejecución correspondiente. La solución de dichas restricciones nos daría por tanto datos de entrada concretos.

Sin embargo, un problema importante de la ejecución simbólica, independientemente de si se realiza en CLP o utilizando una máquina simbólica, es que el árbol de ejecución a ser recorrido, es en la mayoría de los casos infinito, pues los programas suelen contener construcciones iterativas y recursiones las cuales suelen inducir un número infinito de caminos de ejecución al ejecutarse sin valores concretos. Es por tanto esencial establecer un *criterio de terminación*, en este contexto *criterio de recubrimiento*, que garantice que el número de caminos a ser recorrido es finito, y al mismo tiempo que se obtiene un conjunto interesante de casos de valores de entrada.

La mayoría de criterios de recubrimiento están definidos sobre lenguajes de programación estructurados y de alto nivel. Un criterio basado en el flujo de control ampliamente utilizado es el loop-count(k) [52], el cual limita a una cantidad k el número de veces que se puede iterar en los bucles. No obstante, el bytecode tiene un flujo sin estructura, cuyos CFG's pueden variar mucho en forma. Es por ello que en esta tesis hemos introducido el criterio de recubrimiento *block-count*(k) . Éste, en lugar de limitar el número de veces que se itera en bucles, cuenta el número de veces que se visita cada bloque durante cada computación. Básicamente, un conjunto de caminos de computación satisface el criterio *block-count*(k) si éste incluye

todos los caminos de computación terminados en los que el número de veces que se visita cada bloque no excede la k dada.

En el Artículo 7 se discuten los detalles técnicos de dicho enfoque de TDG de bytecode. En particular:

- Se define formalmente el criterio *block-count*(k) .
- Se define una *estrategia de evaluación* que garantiza construir un árbol SLD de forma que se generen suficientes derivaciones para cumplir el criterio block-count(k), asegurando al mismo tiempo la terminación del proceso.
- La fase de TDG se formaliza como una EP en CLP del programa CLP decompilado donde la regla de desplegado juega el papel del criterio de recubrimiento. Definimos además una regla de desplegado que implementa el criterio de recubrimiento block-count(k) y describimos como debe el operador de abstracción tratar con restricciones para poder obtener generadores de casos de prueba efectivos.
- Todos estos aspectos se ilustran a través de un ejemplo que consiste en una serie de métodos que realizan diferentes cálculos aritméticos.

3.2.1. Generando Datos de Prueba para Prolog por EP

Como contribución tangencial de esta tesis, hemos aplicado la idea de utilizar EP para generar automáticamente datos de prueba en el contexto de la LP. Ya mencionamos que nuestro enfoque podría utilizarse en principio para hacer TDG de cualquier lenguaje imperativo. Sin embargo, al tratar de aplicarlo a un lenguaje declarativo como Prolog, encontramos problemas a la hora de generar datos de prueba que cubran ciertos flujos de control de Prolog. Básicamente, el problema es que una característica intrínseca de la EP es que sólo computa derivaciones no fallidas, mientras que en la TDG de Prolog es esencial generar casos de prueba asociados a derivaciones de fallo. En el Artículo 8 hemos realizado un estudio preliminar en este dirección, en el que se propone transformar el programa Prolog original en un programa Prolog equivalente con *fallo explícito*. Esto puede hacerse evaluando parcialmente un intérprete Prolog que captura las derivaciones de fallo del

programa. Otro aspecto importante que se discute en el Artículo es que, mientras que en el caso del lenguaje bytecode considerado anteriormente, el dominio de restricciones sólo necesitaba manipular números enteros, en Prolog éste debe tratar adecuadamente los datos simbólicos que maneja el programa. Nuestros experimentos preliminares sugieren que el enfoque de TDG basado en EP propuesto en la sección puede ser también útil para la generación automática de casos de prueba para Prolog.

3.2.2. Trabajo Relacionado en la Generación de Datos de Prueba

Como hemos mencionado anteriormente, nuestro enfoque se centra en TDG estático, en el que los casos de prueba se generan sin ejecutar realmente el programa con datos particulares. Por el contrario, los enfoques *dinámicos* [38, 45] ejecutan el programa para ciertos valores de entrada concretos hasta conseguir satisfacer el criterio de recubrimiento correspondiente. El enfoque estándar para generar casos de prueba estáticamente es la *ejecución simbólica* [28, 70, 72, 56, 44], Ésta se ha combinado con el uso de *resolutores de restricciones* en [72, 44] para: tratar los sistemas de restricciones resolviendo la viabilidad de los caminos y después instanciar las variables de entrada. Para el caso particular de Java Bytecode, en [72] se propone una JVM simbólica que integra varios resolutores de restricciones.

La TDG para lenguajes declarativos ha recibido comparativamente mucha menos atención que para lenguajes imperativos. La mayoría de las herramientas existentes para lenguajes funcionales son de tipo "black-box", es decir, generan los casos de prueba a partir de la especificación del programa (ver por ejemplo [27]). Una excepción es [39] donde se propone un enfoque de tipo "glass-box" para el lenguaje Curry. En el caso de CLP, se han generado casos de prueba para Prolog en [68, 13, 90]; y más recientemente para Mercury [35]. Básicamente, para obtener los casos de prueba, primeramente calculan las restricciones sobre las variables de entrada asociadas con los diferentes caminos de computo considerados, y después resuelven las restricciones para obtener valores concretos. Por otro lado, se han definido criterios de recubrimiento específicos para lenguajes funcionales [39] en los que se consideran aspectos del lenguaje como la *pereza* ("laziness").

En general, los lenguajes declarativos plantean diferentes problemas re-

lacionados con sus propios modelos de ejecución –como la *pereza* en lenguajes funcionales o el *fallo* en lenguajes lógicos (con restricciones)– los cuales han de ser tratados por los correspondientes criterios de recubrimiento. Una vez dicho esto, pensamos que las ideas relacionadas con el uso de técnicas de EP para generar generadores de casos de prueba, así como el uso de reglas de desplegado para supervisar la evaluación, se pueden adaptar para lenguajes declarativos como hemos mostrado en nuestros resultados preliminares del Artículo 8.

Capítulo 4

Análisis del Consumo del Heap para Bytecode

Predecir la cantidad de memoria que un programa requiere para su ejecución es crucial en muchos contextos, como en aplicaciones *empotradas*, donde suele haber fuertes restricciones de espacio, o en sistemas de tiempo real, que han de responder a eventos tan rápido como sea posible. Se sabe también que la estimación del uso de memoria también es importante para una predicción precisa del tiempo de ejecución, pues los fallos de página y de memoria *cache* contribuyen significativamente al tiempo de ejecución.

El análisis del consumo del heap trata de inferir *cotas* en el consumo del heap de los programas. Como es habitual, éste se ha formulado típicamente al nivel del código fuente (ver por ejemplo[83, 49, 85, 53] en el contexto de la programación funcional y [51, 23] para lenguajes imperativos de alto nivel). Como mencionamos en el Capítulo 1.4, hay sin embargo situaciones en las que no se tiene acceso al código fuente. El análisis del consumo del heap tiene aplicaciones interesantes en este contexto. Por ejemplo, la *certificación de cotas de recursos*, "resource bound certification" [32, 8, 10, 50, 22], propone utilizar propiedades de seguridad incluyendo requerimientos de coste, es decir, el código recibido ha de adherirse a unos requerimientos específicos respecto a su consumo de memoria. También, las cotas en el consumo del heap pueden resultar útiles en sistemas empotrados ("embedded systems"), por ejemplo, en tarjetas inteligentes en las cuales la memoria es limitada y no puede recuperarse de forma sencilla.

Recientemente, en [3] se ha propuesto un análisis de coste para Java

Bytecode secuencial, dando lugar al sistema COSTA [5]. Dicho análisis genera estáticamente *relaciones de coste* (*CRs*) que definen el coste del programa como una función de los tamaños de sus argumentos de entrada. Estas relaciones, se expresan por medio de *ecuaciones recursivas* generadas abstrayendo la estructura recursiva del programa e infiriendo relaciones entre los argumentos. El análisis es paramétrico respecto al *modelo de coste*, el cual define la unidad de coste asociada con cada instrucción bytecode de programa.

Esta tesis desarrolla una aplicación novedosa del análisis de coste propuesto en [3] para inferir cotas en el consumo del heap de programas Java Bytecode:

1. En un primer paso, desarrollamos un modelo de coste que define el coste de cada instrucción de alojamiento de memoria (`new`, `newarray`, etc) en términos del número de unidades del heap que consumen. Por ejemplo, el coste de crear un nuevo objeto será el número de unidades de heap alojadas por el objeto. El resto de las instrucciones bytecode no añaden ningún coste. Con este modelo, generamos *CRs*, y las usamos para inferir cotas del consumo de memoria de los diferentes métodos del programa. Estas cotas, proporcionan información de la cantidad máxima de heap que se requiere para ejecutar cada método.

2. Desafortunadamente, en el caso de lenguajes con *recolección automática de basura*, "garbage collection" (GC), este enfoque, aunque es correcto, produce estimaciones demasiado pesimistas. Es por ello, que en un segundo paso, refinamos el análisis para que se considere el efecto del GC. Proponemos por tanto un *análisis del consumo de la memoria activa*, que aproxima la cantidad máxima de heap usada durante la ejecución de un programa, proporcionando una estimación mucho más precisa en presencia del GC. Para ello, nos basamos en un *análisis de escape* [17] para identificar aquellos objetos creados en un método, que serán recolectados antes de salir de él. Con esta información, inferimos cotas de la *memoria escapada* en la ejecución del método, es decir, la memoria que se aloja durante la ejecución del método, *y que* permanece ocupada tras su finalización. Proponemos entonces una nueva forma de *CRs* del *consumo pico* del heap, que capturan el consumo pico de la ejecución del programa sobre todos

sus posibles estados. Una característica esencial de nuestras *CRs*, es que éstas pueden resolverse utilizando resolutores existentes, en particular el propuesto en [9].

Estos aspectos se introducen respectivamente en las Secciones 4.1 y 4.2, y se estudian en detalle en los Artículos 9 y 10.

Una característica única de los análisis presentados en esta tesis con respecto a trabajos previos (por ejemplo [10, 49, 18, 24]), es que éstos no están limitados a cotas lineales ni polinómicas, pues nuestras *CRs* pueden en principio capturar cualquier clase de complejidad. Más aún, en la mayoría de los casos, utilizando el resolutor de *CRs* del sistema COSTA, nuestras relaciones pueden simplificarse a una forma cerrada, lo que nos da información directa sobre el consumo del código en cuestión.

Una observación importante es que estos análisis se podrían también haber desarrollado de forma similar utilizando nuestros programas decompilados LP. De hecho, COSTA decompila también el bytecode a una representación basada en reglas antes de realizar el análisis propiamente dicho, con el propósito de simplificar el diseño (ver [3] para más detalles). Las representaciones intermedias de COSTA son de hecho muy similares a nuestros programas decompilados, con la diferencia fundamental de que, en COSTA, prácticamente, todas las instrucciones de bytecode quedan residuales en el código como "builtins", es decir predicados predefinidos. Por el contrario, en nuestras decompilaciones, las instrucciones de bytecode se interpretan y evalúan en tiempo de decompilación, y se convierten, en su caso, a instrucciones básicas de Prolog, como unificaciones y operaciones aritméticas. La razón fundamental por la que decidimos no usar nuestras decompilaciones para el análisis de consumo de memoria, es que de esta manera hemos sido capaces de integrar nuestro análisis en el sistema COSTA, aprovechando así toda la maquinaría para el análisis de coste incluida en él (por ejemplo, el *análisis de tamaños* que infiere las relaciones de tamaños entre argumentos, el resolutor de *CRs*, etc).

4.1. Análisis del Consumo Total

Consideremos el programa Java que aparece en la Figura 4.1. Consiste en una serie de clases Java que definen una estructura de datos del tipo

```
abstract class List {
  abstract List copy();
}

class Nil extends List {
  List copy() {
    return new Nil();
  }
}

class Cons extends List {
  int elem;
  List next;

  List copy(){
    Cons aux = new Cons();
    aux.elem = m(this.elem);
    aux.next = this.next.copy();
    return aux;
  }

  static int m(int n) {
    Integer aux = new Integer(n);
    return aux.intValue();
  }
} // class Cons
```

Figura 4.1: Ejemplo de consumo de memoria

lista enlazada, implementada en un estilo fuertemente orientado a objetos. La clase `Cons` se utiliza para los nodos de datos (en este caso números enteros), y la clase `Nil` juega el papel de *null* para indicar el final de la lista. Tanto `Cons` como `Nil` heredan de la clase abstracta `List`. Así, los objetos del tipo `List` pueden ser bien instancias de `Cons` o de `Nil`. Ambas subclases implementan el método `copy`, el cual se utiliza para *clonar* el objeto correspondiente. En el caso de `Nil`, `copy` simplemente devuelve una nueva instancia de sí mismo, pues es el último elemento de la lista. En el caso de `Cons`, se devuelve una instancia clonada donde el dato se clona invocando al método estático `m`, y la continuación se clona llamando recursivamente al método `copy` sobre el objeto `next`.

Nuestro análisis de consumo del heap infiere relaciones de coste (simplificadas) para el método `copy` de la clase `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 12 + C_{copy}(a-1), \quad a > 1$$

las cuales se pueden resolver usando el resolutor de COSTA dando como resultado la siguiente cota en forma cerrada:

$$C_{copy}(a) = 12*nat(a-1) + 12$$

Se puede observar que el consumo de heap es lineal respecto al parámetro de entrada `a`, que se corresponde con el tamaño del objeto *this* del método,

es decir, la longitud de la lista a clonar. Esto ocurre gracias a que la abstracción utilizada por nuestro análisis para referencias a objetos es la *longitud de la cadena de referencias más larga*, que en este caso se corresponde con la longitud de la lista. La constante numérica 12 se obtiene sumando 8 y 4, siendo 8 el número de bytes ocupados por una instancia de la clase `Cons`, y 4 los bytes ocupados por una instancia de `Integer`. Nótese, que estamos aproximando el tamaño de los objetos como la suma de los tamaños de todos sus atributos. En particular, asumimos que, tanto un entero (*integer*) como una referencia ocupan 4 bytes.

El análisis ha sido integrado en el sistema COSTA. En el Artículo 9 se discuten los resultados obtenidos en nuestra evaluación experimental, en la que se estudia el comportamiento del análisis con una serie de aplicaciones, escritas en un estilo fuertemente orientado a objetos, que hacen un uso intensivo del heap, y que ilustran diferentes aspectos relevantes como consumos dependientes de atributos, herencia y polimorfismo, invocaciones virtuales, etc. Estos ejemplos ilustran los aspectos más relevantes de nuestro análisis: inferencia de consumos constantes, consumos proporcionales al tamaño de la entrada, soporte para estructuras de datos como listas, árboles, arrays, etc. Por que sabemos, éste es el primer análisis de consumo capaz de inferir cotas arbitrarias para Java Bytecode.

4.2. Análisis de Consumo del Heap Activo para Lenguajes con GC

Como hemos comentado anteriormente, la recolección de basura (GC) complica mucho el problema de la predicción de memoria. Una primera aproximación es inferir el consumo de memoria *total*, es decir, la cantidad acumulada de memoria alojada por el programa, sin tener en cuenta el GC (como se hicimos en la Sección anterior). Si disponemos de dicha cantidad, está garantizado que el programa podrá ejecutarse, incluso si el GC no se aplica durante la ejecución. No obstante, el consumo inferido es una estimación demasiado pesimista del consumo de memoria real.

Esta tesis presenta un enfoque genérico para inferir el *consumo pico de memoria* durante la ejecución del programa. Nuestro análisis del consumo del heap activo se ha formulado para (una representación intermedia de)

un lenguaje bytecode con orientación a objetos y con gestión automática de memoria (es decir, GC).

El análisis de la memoria activa se diferencia del análisis del consumo total pues requiere considerar el consumo en *todos los estado del programa* durante su ejecución, a diferencia del consumo total, en el que sólo se ha de tener en cuenta el estado *final*. Como consecuencia, el enfoque clásico de análisis de coste estático propuesto por Wegbreit en 1975 [86] sólo se ha aplicado para inferir consumos totales (o acumulativos). La ventaja de este enfoque es que puede obtener información precisa sin estar restringido a ninguna clase de complejidad. Además, en principio, el enfoque es genérico en el sentido de que puede usarse para inferir diferentes nociones de recursos como consumo de memoria, número de instrucciones, número de llamadas a métodos, etc. Desafortunadamente, como discutimos en el Artículo 9, el enfoque no es válido para inferir el consumo pico pues éste no es un recurso acumulativo de la ejecución del programa. Sin embargo, requiere razonar sobre todos los posibles estados para calcular su máximo. Basándonos en distintas técnicas, que no generan *CRs*, el análisis de consumo del heap activo está actualmente limitado a cotas polinómicas y a métodos no recursivos [18] o a cotas lineales con recursión [24].

Inspirándonos en las técnicas básicas usadas en los análisis de coste, en esta tesis, presentamos un enfoque general para inferir cotas precisas en el consumo pico de los programas, mejorando el estado actual del arte al no estar el enfoque restringido a ninguna clase de complejidad, y al ser capaz de tratar la recursión. Para desarrollar nuestro análisis necesitamos primeramente caracterizar el comportamiento del recolector de basura subyacente. Asumiremos un gestor de memoria basado en *entornos* ("scopes"), que reclama memoria sólamente al finalizar los métodos. Nuestras principales contribuciones son:

1. *Análisis de la memoria escapada.* En primer lugar, desarrollamos un análisis para inferir cotas superiores de la *memoria que escapa* de los métodos, es decir, la memoria alojada durante la ejecución del método y que permanece cuando éste finaliza. La idea básica es inferir primero una cota superior del consumo total de memoria del método, como hacemos en la Sección 4.1. Después, dicha cota puede ser manipulada, utilizando la información inferida por un *análisis de escape* [17] para extraer de él una cota superior de la memoria escapada.

2. *Análisis del consumo de memoria activa.* Utilizando las cotas superiores del punto anterior, como nuestra principal contribución, proponemos una nueva forma de *ecuaciones del consumo pico*, que capturan el consumo pico sobre todos los estados del programa para el gestor de memoria considerado. Una característica fundamental de nuestras *CRs* es que aún se pueden resolver usando los resolutores existentes.

3. *Recolector de basura ideal.* Un aspecto muy interesante, y al mismo tiempo novedoso de nuestro enfoque es que podemos refinar fácilmente el análisis para acomodar otros tipos de gestores de memoria, en particular más cercanos al gestor *ideal*, el cual recolectaría los objetos tan pronto como éstos dejan de ser referenciables.

4. *Implementación.* El análisis ha sido implementado e integrado en el sistema COSTA. Hemos realizado además una evaluación experimental usando los "benchmarks" JOlden. Los resultado preliminares demuestran que el sistema obtiene cotas del consumo pico de los programas razonablemente precisas de forma totalmente automática.

Consideremos de nuevo el ejemplo de la sección previa. Nuestro análisis de consumo del heap activo infiere las siguientes *CRs* (simplificadas) para el método `copy` de la clase `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 8 + \max(4, C_{copy}(a-1)), \quad a > 1$$

La intuición de la segunda relación es que el consumo pico del método cuando $a > 1$ es el consumo del método (un objeto `Cons`) más el máximo entre el consumo pico del método `m` y la memoria escapada de `m` más el consumo pico de *copy* con el argumento decrementado. El *CRs* puede de nuevo resolverse utilizando el resolutor de COSTA devolviendo la siguiente cota en forma cerrada:

$$C_{copy}(a) = 8*nat(a-1) + 24$$

Una observación interesante es que el objeto de tipo *Integer* creado dentro del método *m*, no es alcanzable desde fuera y por tanto puede ser recolectado. Nuestro análisis lo tiene en cuenta, y es por ello, que ha borrado el tamaño del objeto *Integer* de la ecuación recursiva, obteniéndose 8 en lugar

de 12 multiplicando a $\texttt{nat}(\texttt{A}-\texttt{1})$. También podemos observar que COSTA no está siendo del todo preciso, pues el consumo pico real del método es $\texttt{8} * \texttt{nat}(\texttt{A} - \texttt{1}) + \texttt{8}$ (el tamaño de la lista clonada). La razón de esta imprecisión es que el resolutor de cotas superiores ha de considerar los casos adicionales introducidos por el análisis del consumo pico de memoria en las expresiones *max* para asegurar su corrección, haciendo que la segunda constante crezca hasta 24.

4.3. Trabajo Relacionado

En la literatura hay una gran cantidad de trabajos sobre el análisis de recursos y de complejidad, aunque la mayoría son sobre análisis de tiempos (ver por ejemplo [88]). El análisis del consumo del heap activo es diferente pues requiere que se consideren todos los estados del programa. La mayoría de los trabajos sobre estimación de memoria, se han realizado en el contexto de lenguajes funcionales. El trabajo en [49] infiere estáticamente, por medio de derivaciones de tipos y programación lineal, expresiones lineales que dependen de los parámetros funcionales. Nótese que con nuestro enfoque se pueden calcular cotas no lineales (polinómicas, logarítmicas, exponenciales, etc). Las técnicas propuestas en [83, 82] consisten en, dada una función, construir una nueva función que representa simbólicamente el coste de la primera. Aunque estas funciones recuerdan a nuestras relaciones de coste, éstas deben ejecutarse sobre unos valores concretos de los parámetros para obtener una cota de memoria para una asignación concreta. A diferencia de nuestras cotas en forma cerrada, la evaluación de su función podría no terminar, incluso aunque el programa original si lo hiciese.

Merece la pena mencionar el trabajo en [21], donde se presenta una análisis del consumo de memoria. A diferencia de nuestro enfoque, su propósito es verificar que el consumo de memoria del programa está acotado. Los autores consiguen esto simplemente comprobando que no se creen objetos dentro de bucles, pero en ningún momento infieren cotas simbólicas como hacemos en nuestro análisis. Nótese que realizar este tipo de comprobación resultaría directo usando nuestras ecuaciones de coste. Otro trabajo relacionado es el realizado en el proyecto MRG ("Mobile Resource Guarantees") [10, 16], en el cual se centran en construir una arquitectura de "Proof-Carrying-Code"[73] en la que se asegura que los programas no

violan las restricciones de consumo de recursos impuestas. El análisis se desarrolla para un lenguaje funcional que se compila posteriormente a un subconjunto de Java Bytecode, y está restringido a cotas lineales.

Para lenguajes del estilo de Java, podemos mencionar el trabajo de [51], donde se presenta un sistema de tipos para realizar análisis de heap sin recolección de basura. El análisis está desarrollado a nivel del código fuente y está basado en técnicas de *análisis amortizado*. Es por tanto técnicamente diferente al nuestro, y no llega a proponer un método de inferencia de consumo del heap.

Se han propuesto también recientemente técnicas que tratan de mejorar nuestro primer enfoque, presentado en el Artículo 9. En particular, [24] considera un lenguaje ensamblador, e infiere cotas del consumo de memoria (tanto de la pila como del heap). El enfoque está, no obstante, restringido a cotas lineales, y se basa en la existencia de comandos explícitos de liberación de memoria en lugar de en un gestor automático de memoria. En su sistema, estos comandos de liberación de memoria pueden generarse automáticamente a partir de unas anotaciones de usuario. En [18] se considera un lenguaje del estilo de Java y se infieren cotas del consumo pico basándose en un gestor automático de memoria como hacemos nosotros. Sin embargo no tratan con métodos recursivos y su enfoque está restringido a cotas polinomiales. Además, nuestro enfoque (Artículo 10) es más flexible en cuanto a su posible adaptación a distintos esquemas de recolección de basura. Pensamos que nuestro sistema es el primero capaz de inferir cotas del consumo pico de los programas no restringidas a ninguna clase de complejidad.

Capítulo 5

Conclusiones y Trabajo Futuro

El principal objetivo de esta tesis ha sido mejorar el estado del arte en la transformación y análisis de lenguajes bytecode. Nuestro primer reto fue proponer un esquema formal para la decompilación automática de programas bytecode (con orientación a objetos) a representaciones intermedias de alto nivel usando LP, por medio de decompilación interpretativa. Este enfoque ofrece una serie de ventajas comparado con el desarrollo de decompiladores dedicados como *flexibilidad*, *mantenibilidad*, *seguridad* y *generalidad*. Aunque es muy atractivo, hasta ahora no se había usado realmente en la práctica exceptuando algunas *pruebas de concepto* en las que simplemente se demuestra su viabilidad [62, 47, 75, 63].. Quedaban por tanto una serie de cuestiones abiertas a la hora de tratar de aplicar el enfoque interpretativo en lenguajes y aplicaciones reales, en particular su *escalabilidad* y *efectividad*. Esta tesis propone soluciones novedosas y finalmente responde afirmativamente a estas cuestiones presentando un esquema de decompilación modular y óptimo que: (1) produce programas decompilados cuya calidad es equivalente a la obtenido utilizando decompiladores dedicados, y (2) demuestra teórica y empíricamente escalar en la práctica. Los resultados experimentales obtenidos muestran que nuestro decompilador es competitivo en la práctica, desde el punto de vista de la eficiencia, con decompiladores dedicados. Creemos por tanto que, las técnicas propuestas, junto con su evaluación experimental, proporcionan por primera vez una prueba tangible de que la teoría interpretativa propuesta por Futamura en los años 70, es de hecho una alternativa viable y realista al desarrollo de decompiladores dedicados de lenguajes modernos a representaciones

intermedias.

Para concretar, nuestro esquema de decompilación interpretativa ha sido formalizado en el contexto de la EP de programas lógicos, e implementado para el lenguaje Java Bytecode. Es sin embargo importante notar que las ideas propuestas para posibilitar la puesta en práctica del enfoque, son por supuesto de interés para la decompilación interpretativa de cualquier par de lenguajes *origen* y *destino*.

Por otro lado, el estudio de una aplicación tan compleja de la EP, nos ha llevado a resolver varios problemas no triviales de ésta en general, como por ejemplo el tratamiento de signaturas infinitas. A este respecto, se ha presentado la relación de la *subsunción homeomórfica basada en tipos*, la cual ha demostrado mejorar el estado del arte en las herramientas de especialización online. Hemos visto también como distintos enfoques existentes que extendían la relación original no tipada para tratar con signaturas infinitas, se pueden reconstruir como instancias de nuestra relación TbHEm. Aunque se han esbozado procedimientos para inferir los tipos en el contexto de la LP, nuestra relación basada en tipos no está atada a ningún paradigma de programación. Más aún, se podría usar en un rango amplio de aplicaciones como en distintas áreas de análisis, síntesis, verificación, especialización y transformación automática de programas, y además se beneficiaría directamente de cualquier progreso en inferencia automática de tipos.

Hemos visto como la representación intermedia resultante utilizando LP, puede simplificar en gran medida el desarrollo de herramientas de análisis, verificación y chequeo de modelos para lenguajes modernos, y, en particular, como pueden ser directamente aplicadas herramientas existentes desarrolladas para LP (probadas correctas y efectivas). Hemos realizado dos estudios experimentales en esta dirección. En el primero de ellos, hemos investigado si es viable analizar programas bytecode a base de analizar sus decompilaciones a LP utilizando herramientas de análisis LP existentes. En este sentido, hemos sido capaces de demostrar automáticamente, usando el sistema `CiaoPP`, algunas propiedades no triviales de programas Java Bytecode (de tamaño pequeño) como, ausencia de errores en tiempo de ejecución, terminación e inferencia de cotas en el uso de recursos.

En nuestro segundo estudio experimental, hemos aprovechado el hecho de que nuestros programas decompilados son totalmente *ejecutables*, pues

a diferencia de otros enfoques [69, 2, 84], representan el estado completo de ejecución (es decir, contienen una representación explícita del heap además de la pila de operandos). En este sentido, hemos propuesto un esquema de generación automática de datos de prueba, basado en ejecución simbólica, para lenguajes bytecode por medio de técnicas de EP en CLP. Nuestro enfoque consiste en dos fases diferenciadas: (1) la (de)compilación del bytecode a un programa CLP, y (2) la generación de datos de prueba a partir del programa CLP. una pregunta que surge inevitablemente es si este enfoque puede utilizarse para otros lenguajes imperativos que no sean necesariamente bytecode. Los enfoques basados en ejecución simbólica existentes para Java [72], y para C [44], presentan problemas a la hora de tratar con aspectos como la recursión, las llamadas a métodos, la memoria dinámica, etc. Hemos visto como estos aspectos se tratan de forma uniforme en nuestro enfoque gracias a la transformación a CLP. En particular, todas las clases de bucles en el bytecode se representan de forma uniforme como predicados recursivos en el programa CLP. Hemos visto también como las llamadas a métodos se tratan de la misma manera que las llamadas a bloques, y por tanto no presentan ninguna dificultad adicional.

Pensamos que el estudio experimental realizado es una prueba de concepto muy prometedora, que muestra que la EP en el contexto de CLP es una técnica muy potente, en particular, para realizar TDG de lenguajes bytecode e imperativos en general. Para poner en práctica nuestras ideas hemos considerado un sencillo lenguaje bytecode imperativo dejando aspectos como la orientación a objetos para trabajo futuro. Hemos restringido también el lenguaje a números enteros quedando también como trabajo futuro la extensión para tratar con diferentes tipos de datos. A corto plazo, planeamos realizar una evaluación experimental con programas Java Bytecode de "benchmarks" existentes. Al considerar aspectos de lenguajes bytecode más realistas como el uso de números reales, llamadas virtuales, etc, seguro que tendremos que tratar como numerosas dificultades. Uno de los puntos prácticos fundamentales es la escalabilidad del enfoque, que suele venir ligada al problema de la *no viabilidad de caminos* [91]. Éste suele ocurrir sobretodo en enfoques que no integran la fase de resolución de restricciones con la de generación de caminos, sino que realizan estas fases de forma independiente. En este sentido, no esperamos tener problemas pues nuestro enfoque integra ambas fases y por tanto detecta y descar-

ta los caminos inviables tan rápido como éstos aparecen. Otro problema interesante es el obtener una representación manejable del heap, lo cual será necesario para poder obtener casos de prueba en programas que manipulen objetos y arrays. Para la evaluación experimental también planeamos extender nuestra técnica para incluir criterios de recubrimiento más avanzados. En particular sería interesante considerar otras clases de criterios que, por ejemplo, nos permitan obtener casos de prueba que recubran una determinada instrucción del programa.

Como hemos visto, en principio nuestro enfoque de TDG puede aplicarse a cualquier lenguaje, tanto de alto como de bajo nivel. En este sentido, esta tesis también ha presentado un estudio experimental en el que tratamos de aplicar la segunda fase para generar casos de prueba de programas CLP, no necesariamente obtenidos por decompilación de programas bytecode o imperativos. Esto introduce algunas dificultades como el tratamiento de las derivaciones de fallo y de los datos simbólicos. Esta tesis ha esbozado las soluciones para superar dichas dificultades. En particular, hemos propuesto una transformación de programa, basada en EP, que hace explícito el fallo en los programas lógicos. Para tratar la negación en los programas transformados, hemos esbozado soluciones basadas en técnicas existentes, que hacen posible transformar la información negativa en positiva. Aunque nuestros primeros experimentos en este sentido ya sugieren que el enfoque puede ser muy útil, por ejemplo para generar casos de prueba para programas Prolog, pensamos realizar aún una evaluación experimental en profundidad comparando con técnicas existentes. Esto requeriría cubrir ciertos aspectos del lenguaje Prolog que aún no hemos considerado, como el corte, el sistema de módulos, etc. Queremos también estudiar la integración de otras clases de criterios de recubrimiento como los basados en el *flujo de datos*. Finalmente, nos gustaría estudiar la integración los análisis estáticos en el contexto de TDG. Por ejemplo, utilizar la información inferida por un *análisis de fallo* podría ser muy útil para poder así podar algunas de las ramas que nuestros programas transformados en principio han de considerar.

Otro de los grandes retos de esta tesis ha sido mejorar el estado del arte en el análisis de consumo del heap de lenguajes bytecode. En este sentido, hemos desarrollado una aplicación novedosa del esquema de análisis de coste de [3] para analizar consumo del heap, el cual ha sido también extendido

para considerar el efecto de la recolección de basura. Hemos presentado por tanto un enfoque genérico para el análisis automático y preciso de consumo del heap activo para lenguajes bytecode con recolección automática de basura. Para ello, primero hemos propuesto como obtener cotas precisas de la memoria que escapa de los métodos, combinando el consumo total inferido por el propio método, junto con la información obtenida por un análisis de escape. Después, hemos introducido una forma novedosa de *relaciones del consumo de memoria pico*, que utilizando las cotas de la memoria escapada, capturan el consumo pico de los programas considerando el efecto de la recolección de basura. Estas relaciones de coste se pueden convertir a forma cerrada utilizando resolutores existentes, en particular el del sistema COSTA. Para concretar, nuestro análisis ha sido desarrollado para un lenguaje bytecode con orientación a objetos, aunque pensamos que las mismas técnicas podrían aplicarse a otros lenguajes con recolección de basura. En primer lugar, el análisis considera un gestor de memoria basado en *entornos*, que sólamente reclama la memoria en la finalización de los métodos. La cantidad de memoria requerida para poder ejecutar un método bajo este modelo, puede usarse como una sobreaproximación de la cantidad requerida en el contexto de una recolector de basura ideal (que reclama la memoria de los objetos tan pronto como estos dejan de ser alcanzables). Hemos mostrado también como aproximar el comportamiento de dicho recolector ideal en nuestro análisis.

Finalmente, es importante notar que este enfoque podría también utilizarse para estimar otros recursos no acumulativos, que requieren maximizar el consumo de varios caminos de ejecución diferentes. Por ejemplo, pensamos que podría utilizarse para estimar la profundidad máxima de la pila de llamadas de la siguiente manera:

Dada una regla $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, donde $b_{i_1} \ldots b_{i_k}$ son las llamadas en r, con $1 \leq i_1 \leq \cdots \leq i_k \leq n$ y $\mathrm{b}_{\mathrm{i_j}} = \mathrm{q}_{\mathrm{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, su ecuación correspondiente sería

$$p(\bar{x}) = \text{máx}(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_k}(\bar{x}_{i_k})) \quad \varphi_r$$

que considera la profundidad máxima de todas las cadenas de llamadas. Cada "1" corresponde a cada "frame" creado para la llamada correspondiente. Ésta es por tanto otra línea de posible trabajo futuro.

Parte II

Versión en Inglés (English Version)

Chapter 6

Introduction: Motivation and Contributions

6.1. Bytecode Languages

Programming languages can in general be categorized by the underlying execution model that runs them. Thus, they typically fall into one of two categories: *compiled* or *interpreted*. In compiled languages the *source code* is first translated, or compiled, into a set of hardware-specific instructions, often called *object code*. The program is then run by executing the object code in the corresponding hardware. In contrast, in interpreted languages the source code is executed by an interpreter. The distinction applied to programming languages is somewhat vague as in theory any language may be compiled or interpreted. The categorization usually reflects the most popular or widespread implementations of languages and not their underlying properties. Each of the alternatives has their own advantages and drawbacks. For instance, the execution of an object program in the corresponding machine tends to be much faster than executing the same source program using an interpreter, even a 10:1 ratio is not uncommon. On the other hand, interpreted languages provide certain extra flexibility over compiled programs, namely, ease of implementation, ease of debugging, and the most important, platform-independence.

A combination of both approaches, known as *bytecode-compilation* or *bytecode-interpretation*, is becoming widely used. In a bytecode-compilation-based execution model, source code is translated to some inter-

```
void foo(int n,int m){
    this.f = n + m;
}
```

```
void foo(int,int)
  0: aload_0        3: iadd
  1: iload_1        4: putfield f
  2: iload_2        7: return
```

Figura 6.1: Simple Java Bytecode program

mediate form, known as *bytecode*. The bytecode is not the machine code for any particular computer, and may be portable among computer architectures. The bytecode may be then interpreted by, or run on, a virtual machine. The name bytecode stems from instruction sets which have one-byte *opcodes* followed by optional parameters. Bytecode instructions are often akin to traditional hardware instructions. For instance, bytecode languages often have an unstructured control flow with several sources of branching (e.g., conditional and unconditional jumps) and use an operand stack to perform intermediate computations. Nevertheless, since bytecode instructions are thought to be processed by software, they may be arbitrarily complex, especially in the case of object-oriented and declarative bytecode languages. To get the picture, Figure 6.1 shows the source code and (Java) bytecode of a method that takes two integer numbers, adds them, and assigns the result to the `f` field of the *this* object. Note that, in Java Bytecode, the *this* object is explicitly passed in local variable 0, thus, the `aload_0` instruction, pushes the *this* reference at the top of the stack.

As regards efficiency, the bytecode-compilation model is typically somewhere in between the pure compilation-based and interpretation-based models; while it keeps the advantages of interpretation-based models, in particular, platform independence. Furthermore, there is nothing about a bytecode language that requires it to be exclusively interpreted. Therefore, *just-in-time* compilation (JIT) can be used to speed up the execution of bytecode. A JIT compiler performs the conversion to native machine code gradually during the program's execution thus obtaining a better performance. Bytecode-compilation together with JIT compilation can therefore combine most advantages of interpretation-based and compilation-based execution models. This is the main reason of success of the runtime envi-

ronments of *Microsoft .NET* and the *Java* frameworks, which are probably the most widely used programming environments nowadays.

Java Bytecode Java Bytecode is the language designed to be executed by the *Java Virtual Machine* (JVM) [65]. It was originally designed by *Sun Microsystems* to be an intermediate language in the Java runtime environment. A Java Bytecode instruction consists of an opcode specifying the operation to be performed, followed by zero or more operands embodying values to be operated upon. The JVM uses a set of structures that the bytecode instructions manipulate. The main ones are: the *program counter* which contains the index of the current instruction, the *operand stack* and *local variables array* in which parameters, variables and intermediate results are stored, the *heap* from which memory for all class instances and arrays is allocated, and the usual *call stack* or *frame stack* to handle method calls and returns. Java Bytecode comprises, on one hand, the classical low-level instructions to transfer values from the operand stack to the local variables and viceversa, to perform arithmetic operations (most of them operate directly on the operand stack), to jump to other part of the code (conditionally or unconditionally), to call (and return from) methods, etc. For instance, the instruction `iload_1` loads the local variable 1 on the top of the operand stack and instruction `iadd` adds (and pops) the top two values of the stack and pushes the result on it. On the other hand, Java Bytecode has object-oriented and concurrency features. Thus it also includes instructions to create objects and arrays, to get and set object fields and array elements, to perform virtual invocations, to enter and exit monitors, etc. E.g., instruction `putfield f` sets with the value on top of the stack, the `f` field of the object referenced by the memory address stored inmediately under the top of the stack.

Although Java is the most common language targeting Java Bytecode, there are nowadays many compilers from different high-level languages to Java Bytecode. Some of the most well known are: *jython* for *Python* programs, *jRuby* for *Ruby* and *jGNAT* for *Ada*.

.NET Common Intermediate Language The *Common Intermediate Language* (CIL) is the bytecode intermediate language used by the *.NET*

programming framework. Thus, source languages targeting the .NET framework compile to CIL. As Java Bytecode, CIL is an object-oriented and stack-based bytecode language. It therefore includes the same kind of bytecode instructions. Unlike Java Bytecode, CIL is not meant to be interpreted. From the beginning, it was rather thought to be compiled into machine code using JIT compilation. The bytecode is even sometimes entirely converted into machine code before runtime using a native image generator to further improve performance. The .NET framework is a key Microsoft offering and is intended to be used by most new applications created for the Windows platform.

There are other well known bytecode languages both imperative, like the *p-code* used in some Pascal implementations, and declarative, like the *WAM* bytecode, used in most Prolog implementations, the *Haskell Hugs'98* bytecode and the *Erlang BEAM* bytecode, to name some.

This thesis is mainly focused on object-oriented imperative bytecode languages. In particular, as we will see, the technical parts of this thesis, as well as the different prototype implementations we have developed, consider representative subsets of Java Bytecode.

6.2. Static Program Analysis

Predicting the behavior of programs before their actual execution becomes more and more relevant as programs increase in complexity and get used in critical situations such as medical operations, flight control or banking cards. Being able to prove, in an automatic way, that programs do adhere to their functional specifications is a basic factor to their success. *Static program analysis* is the process of automatically analyzing the behavior of programs without actually executing the code. In contrast, analysis performed by executing programs is known as *dynamic analysis*. Classical static analyses aim at inferring properties of programs like: *error-freeness*, *termination*, cost or resource consumption (time or memory), *liveness* of variables, *pointer shape*, etc. The usual way to perform static analysis is to use formal methods. Some of the most common are: *abstract interpretation*, *model checking* and *type systems*. This thesis mainly focuses on static analysis based on abstract interpretation.

Abstract Interpretation. The technique of abstract interpretation [29] provides a general formal framework for computing safe approximations (i.e., abstractions) of programs behavior. Its main practical application is formal static analysis. Analyzers based on abstract interpretation infer information on programs by interpreting ("running") them using abstract values rather than concrete ones. These analyzers are parametric w.r.t. the so-called abstract domain, which provides a finite representation of possibly infinite sets of values. Different domains capture different properties of the program with different levels of precision and at a different computational cost.

Abstract interpretation-based static analysis has been studied in the context of declarative languages and also for high-level imperative languages. In what follows we enumerate some analysis systems:

The ASTRÉE analyzer. *ASTRÉE* [30] is a static program analyzer developed at the *École Normale Supérieure* by Cousot *et. al.* aiming at proving the absence of run-time errors of C programs. ASTRÉE was able for example to prove fully automatically the absence of any run-time error in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132.000 lines.

The CiaoPP system. CiaoPP [48] is the abstract interpretation-based preprocessor of the Ciao-Prolog *Contraint Logic Programming* (CLP) system [81]. It can perform a number of program debugging, analysis and source-to-source transformation tasks on Ciao-Prolog programs. Some of the properties it is able to infer are: *types*, *modes* and other variable instantiation properties, *non-failure*, *determinacy*, bounds on computational cost, bounds on sizes of terms in the program, etc. CiaoPP is also able to perform several kinds of source to source program transformations such as program specialization, program parallelization (including granularity control), etc.

Some other well-known (non-commercial and commercial) static analysis systems are: *Lint*, *CCA* and *BOON* for C programs, *CodeSonar* for C++, *Fluid* and *jLint* for Java, and many others. Other static analyzers

have not become self-contained tools but are rather integrated in most compilers. An example of this is the JVM verifier which integrates a data-flow analyzer.

Traditionally, most analyses have been formulated at the source code level. However, it can be the case that the analysis must consider the compiled code, or bytecode, instead. This may happen, in particular, when the code consumer is interested in verifying some properties of 3rd party programs, but has no direct access to the source code, as usual for commercial software and in mobile code. This is the general picture where the idea of *Proof-Carrying code* [73] was born: in order for the code to be verifiable by the user, security properties (possibly inferred by static analysis) must refer to the compiled code (or bytecode) available to the user, so that it is possible to check the provided proof and verify that the program satisfies the requirements (e.g., that the code does not require more than a certain amount of memory, or that it executes in less than a certain amount of time).

Hence, there is a need to develop analysis and verification tools which work directly on bytecode programs. Unfortunately, reasoning about realistic (object-oriented) bytecode programs is rather complicated and time consuming. In addition to the object-oriented features such as inheritance and virtual method invocations, a bytecode analyzer has to deal with several low-level language features like the unstructured control flow, the usage of the operand stack, etc.

6.3. From Bytecode to Intermediate Representations

In the context of analysis of bytecode languages, a usual practice is to approach the problem into two steps: (1) the bytecode program is transformed into a higher-level *intermediate representation* (IR), and (2) the analysis is developed over such IR. This allows abstracting away the particular bytecode language features and developing the analysis tools on much simpler representations. As another advantage, this approach also enables the possibility of reusing the analysis step (step (2) below) for analyzing different bytecode (and not bytecode) languages, provided they are trans-

formed into the same IR. In the rest of the thesis, we will use the term *decompilation* to refer to the transformation of bytecode to an IR, as it translates a low-level language to a higher-level one.

Most of the approaches develop *ad-hoc*, or dedicated, decompilers, i.e., decompilers exclusively designed to carry out a particular decompilation. There is however an alternative to the development of dedicated decompilers which is the so called *interpretive decompilation* by *partial evaluation*. As we will see, it allows decompiling programs by partially evaluating an interpreter w.r.t. them.

Partial Evaluation. *Partial evaluation* (PE for short) [55] is a semantics-based, source to source, program transformation technique, which allows specializing programs w.r.t. part of their input data. Hence it is often called *program specialization*. Consider a program P, and its input which is split in I_{static} and $I_{dynamic}$. I_{static} is the static data, i.e., the input which is known at compile time, and $I_{dynamic}$ is the rest of the input. We can see the program P as a mapping of its input into its output as follows:

$$P : I_{static} \times I_{dynamic} \longrightarrow O$$

The partial evaluator transforms the pair $\langle P, I_{static} \rangle$ into $P' : I_{dynamic} \longrightarrow O$ by performing the computations of P that depend on I_{static} at compile time. P' is called the *residual program* and should run more efficiently than the original program P.

The Interpretive Approach to Compilation. A particularly interesting application of PE, first described in the 1970s by Yoshihiko Futamura [40], is when the program P to be partially evaluated, is an interpreter for a programming language. This is called the *interpretive approach to compilation* or the *first Futamura projection*. Let us assume an interpreter, written in a target language L_T, for programs written in a source language L_S. Then, if I_{static} is a source program, written in L_S, the PE of the interpreter w.r.t. this data/program produces P', a version of the interpreter that only runs that source code, which is written in the implementation language of the interpreter, L_T, and which does not require the source code to be resupplied. P' can be considered as a

compiled version of I_{static} into the target language L_T. The interpretive compilation thus allows compiling programs written in L_S into another language L_T by partially evaluating an interpreter for L_S written in L_T w.r.t. them.

In the particular case of bytecode decompilation, the interpretive approach to compilation allows us to decompile a bytecode program written in some bytecode language BC, into a higher-level representation, written in a high-level language HL, by partially evaluating an interpreter of BC written in HL. This is in principle more generic and flexible, safer and easier to maintain than the development of a dedicated decompiler for the same task. These advantages will be discussed later on in Section 7. The interpretive approach, though very attractive in principle, has not been widely applied in practice mainly because of the difficulty in finding partial evaluation strategies which produce *effective*, i.e., *quality*, and *efficient* decompilations.

6.4. Heap Space Analysis for Bytecode

Research about the resource usage of programs goes back to the seminal work by Wegbreit in 1975 [86], which proposes to analyze the performance of a program by deriving a mathematical expression which represents its runtime behavior. The standard approach to perform static cost analysis is as follows: given an input program, (1) in a first step, the cost analysis generates an associated cost equation system (CES) from the program, which captures the relation between the different parts of the code. CESs are sets of recurrence equations which express the cost of a program in terms of the size of its input arguments. (2) In the second step, CESs can be often solved (or approximated) by typically relying on algebraic techniques, thus obtaining a closed form (e.g., without recurrences) solution or an upper (or lower) bound for it.

Cost analysis has been intensively studied in the context of declarative (see, e.g., [78, 79, 43, 15] for functional programming and [33, 34] for logic programming) and high-level imperative programming languages (mainly focused on the estimation of worst-case execution times and the design of cost models [89]). Traditionally, as most static analyses, cost analysis has

been formulated at the source level. However, as we have seen, there are situations where we do not have access to the source code, but only to the compiled code. Recently, [2] has proposed a cost analysis framework for Java Bytecode which is the formal base of the COSTA system [5].

The COSTA **system.** COSTA [5] is a research prototype which performs automatic **COS**t and **T**ermination **A**nalysis for Java Bytecode programs. The system receives as input a bytecode program and a cost model chosen from a selection of resource descriptions, and tries to bound the resource consumption of the program with respect to the given cost model. COSTA follows the standard approach to perform cost analysis, i.e., it first produces a CES, which is an extended form of recurrence relation. Then, in order to obtain a closed form for such recurrence relations, which represents an upper bound, COSTA includes a dedicated solver [9].

An interesting application of cost analysis which poses new challenges is heap space analysis. It aims at inferring *bounds* on the heap space consumption of programs. Again, heap analysis is more typically formulated at the source level (see, e.g., [83, 49, 85, 53] in the context of functional programming and [51, 23] for high-level imperative programming languages). In the context of bytecode languages, heap space analysis has interesting applications. For instance, *resource bound certification* [32, 8, 10, 50, 22] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered.

Unfortunately, automatic memory management, also known as *garbage collection* (GC), which is increasingly used in bytecode languages like Java Bytecode and the .NET CIL, makes the problem of predicting the memory required to run a program very difficult. A first approximation to this problem is to infer the *total memory allocation*, i.e., the *accumulated* amount of memory allocated by a program ignoring GC. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, this is an overly pessimistic estimation of the actual memory requirement. Recently,

[83, 18, 24] have proposed *live heap space analysis*, which aims at approximating the size of the *live* data on the heap during a program's execution, thus providing a much tighter estimation. These approaches are however currently restricted to polynomial bounds and non-recursive methods [18] or to linear bounds dealing with recursion [24].

6.5. Main Goals and Contributions

The main objective of this thesis is to improve the state-of-the-art in the transformation and analysis of bytecode languages by: (1) providing and implementing a formal framework for the automatic decompilation of (object-oriented) bytecode programs to higher-level intermediate representations, in particular represented using logic programming (LP), by means of the interpretive approach to compilation; (2) study the practical applications that having such IRs based on LP can have; and (3) designing and implementing a live memory consumption analysis for bytecode languages with *garbage collection*. In particular, the main contributions of this thesis are the following:

1. **Interpretive decompilation of bytecode to LP:** There have been several proofs-of-concept showing that the interpretive approach is feasible [62, 47, 75, 63]. However, there remain important open issues when it comes to decompile realistic languages. These include scalability, which in turn depends on compositionality, and effectiveness, i.e., quality of the obtained programs. This thesis presents, to the best of our knowledge, the first scheme to enable interpretive decompilation of a realistic bytecode language to a high-level representation, namely, we decompile Java Bytecode to Prolog.

   a) **Control strategies:** One of the main difficulties of interpretive decompilation and of EP in general, is to properly handle infinite signatures. We have proposed novel techniques which enable the definition of sophisticated control rules. In particular, we have introduced the *Type-based homeomorphic embedding* relation, a generalization of the original *Homeomorphic embedding* relation which provides more precise results in the presence

of infinite signatures. We have shown that this technique, besides being crucial in the specialization of interpreters, improves state-of-the-art (online) specialization tools. This work was first proposed in Paper 3 (see Appendix A) and later extended in Paper 4, which was published in the *Information Processing Letters* journal.

b) **Controlling the polyvariance of EP:** Even after enhancing a partial evaluator with the *Type-based homeomorphic embedding*, the decompiled programs we obtain tend to have too many (redundant) specialized versions of some predicates. This issue is studied in detail in Paper 2, where we propose advanced techniques to control the *polyvariance* of the PE process, i.e., which avoid having such redundant specialized versions.

c) **How to write the bytecode interpreter:** As shown in previous work on interpretive compilation, the characteristics of the interpreter can be crucial for obtaining a successful specialization. We have identified the necessary features in order to obtain a compositional decompilation scheme.

d) **Optimal decompilation:** We ensure the quality of the decompilations, both in terms of effectiveness and efficiency, by providing different *optimality criteria*. They basically require that (1) the decompilation does not generate code more than once for each program point, and (2) there is at most one residual rule for each block in the bytecode. We propose a decompilation scheme which is *optimal* w.r.t. these optimality criteria. This ensures scalability of the process and quality decompilations. This work, together with that described in issue (c), led to Paper 5.

e) **Dealing with object-oriented features:** We show how our scheme can be easily adapted to handle object-oriented features. Namely, we provide the mechanisms to: handle the heap and its associated instructions, represent classes by means of Prolog modules, and represent virtual invocations by means of module-qualified calls.

f) **Implementation and experimental evaluation:** All the techniques above have been implemented and integrated in a pro-

totype decompiler of full sequential Java Bytecode to Prolog, called jbc2prolog. Experimental results are reported using our prototype (and other systems). In particular, both the scalability and efficiency of our approach are assessed using the JOlden suite of benchmarks [54]. The work described in issues (b), (c), (d), (e) and (f), led to Paper 6 which has been recently published by the *Journal of Information and Software Technology*. This paper thus achieves the objective (1) above.

2. **Applications of interpretive decompilation:** Using a declarative language for defining our IR offers important advantages. In particular, existing advanced analysis and transformation tools for declarative languages (already proven correct and effective) could be then re-used for the analysis and transformation of bytecode programs

   a) **Re-using LP analysis tools:** Using the CiaoPP system with our decompiled programs we have been able to prove some non-trivial properties of Java Bytecode programs such as *termination* and run-time *error-freeness*, as well as, for some simple programs, to infer bounds on their resource consumption. This work is presented in Paper 1.

   b) **Test data generation:** A standard approach to the automatic generation of test data is to perform *symbolic* execution of the program [28, 56], where the contents of variables are expressions rather than concrete values. The fact that our decompiled programs are executable Prolog programs allows us to directly rely on available techniques for CLP (where backtracking is inherent to the language) to carry out such symbolic execution. We have therefore developed a novel framework for *test-case generation* of bytecode by relying on our decompiled (C)LP programs. Interestingly, we show that the generation of test-cases in CLP, can be seen as another PE, which allows us obtaining not only test-cases but *test-case generators*. This work led to Paper 7. As a tangential contribution, we have applied this idea to automatically generate test-cases for Prolog. A preliminary study in this direction is found in Paper 8.

3. **Heap and Live heap space analysis:**

   a) **Total heap space analysis:** We have first developed a novel application of the cost analysis framework of [3] to infer upper bounds on the heap space consumption of sequential Java Bytecode programs. To do that, we have just provided a cost model that defines the cost of memory allocation instructions in terms of the number of heap (memory) units they consume. We can then generate *heap space cost relations* which are directly used to infer upper bounds on the heap space usage of Java Bytecode programs.

   b) **Live heap space analysis for languages with garbage collection:** In presence of garbage collection, the proposed approach is a too pessimistic estimation of the actual memory requirement. This thesis presents a general approach for inferring the *peak heap consumption* of a bytecode program's execution, i.e., the maximum of the live heap usage along its execution which, unlike previous works, is not restricted to any complexity class.

   c) **Implementation and experimental evaluation:** The analyses have been implemented and integrated in the COSTA system. We experimentally evaluate them with a series of example applications which make intensive use of the heap, including the JOlden benchmark suite [54]. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way. All this work on heap space analysis led to papers 9 and 10, thus achieving objective (3) above.

6.6. Organization of this Thesis

This thesis is a "thesis by articles" and therefore it consists of an introduction describing its main objectives, contributions and conclusions, which is presented in chapters 6, 7, 8, 9 and 10, and, the set of papers which led to the thesis presented as they appear on the corresponding formal proceedings as an appendix.

The rest of the thesis is thus organized as follows: Chapter 7 overviews the work covering the contribution (1) above. In particular, Section 7.1 provides some background on PE of logic programs, then the challenges that specializing a bytecode interpreter are presented in Section 7.2, Section 7.3 introduces the *Type-based homeomorphic embedding* relation, Sections 7.4 and 7.5 summarize the technical details of our modular and optimal decompilation schemes, Section 7.6 summarizes the implementation and experimental evaluation, and finally Section 7.7 overviews related work on (interpretive) decompilation.

Chapter 8 overviews our work on the applications of using our interpretive decompilation technique to analyze bytecode programs (Section 8.1), and to perform test data generation (Section 8.2). Then, Chapter 9 introduces our work on heap space analysis (Section 9) and its extension to consider the effect of garbage collection (Section 9.2), and discusses related work (Section 9.3). Finally, Chapter 10 presents the conclusions of the thesis and discusses ongoing and future work.

The technical details are presented in the papers which led to this thesis, which can be found in Appendix A.

Chapter 7

Interpretive Decompilation of Bytecode to LP

Decompiling bytecode languages to an intermediate representation has become a usual practice nowadays within the development of analyzers, verifiers, model checkers, etc. For instance, in the context of *mobile* code, as the source code is not available, decompilation facilitates the reuse of existing analysis and model checking tools. In general, high-level intermediate representations allow abstracting away the particular language features and developing the tools on simpler representations. In particular, Java Bytecode is decompiled to a rule-based representation in [2], to clause-based programs in [69], to a three-address code representation in Soot [84] and to the typed procedural language BoogiePL in [36]. Also, analysis of Java programs is formalized and performed using Datalog in [87] and in [47] PIC assembly is transformed into logic programs. This shows that the rule-based representations used in declarative programming in general—and in LP in particular—provide a convenient formalism to define such intermediate representations. For instance, as it can be seen in [2, 69, 47], the operand stack used in a bytecode language can be represented by means of explicit logic variables and its unstructured control flow can be transformed into recursion.

The resulting intermediate representation greatly simplifies the development of the above tools for modern languages and, interestingly, existing advanced tools developed for declarative programs (already proven correct and effective) can be directly applied on it.

All above cited approaches (except [47]) develop *ad-hoc*, or dedicated, decompilers to carry out the particular decompilations. As we pointed out in Section 6.3, an appealing alternative to the development of dedicated decompilers is interpretive decompilation by partial evaluation. The advantages of interpretive (de)compilation w.r.t. dedicated (de)compilers are well-known and discussed in the PE literature. Very briefly, they include:

1. *Flexibility*: it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest). As an interesting example, in Paper 1, a Java Bytecode interpreter is instrumented with an additional argument which computes the *trace* of bytecode instructions in order to collect the computation history. A program decompiled by using this interpreter contains an additional argument with the execution trace at the level of Java Bytecode. This trace will allow observing a good number of interesting properties about the program, e.g., runtime *error-freeness* can be ensured when the trace does not contain instructions which issue any kind of run-time error.

2. *Easier to trust*: it is more difficult to prove (or trust) that ad-hoc decompilers preserve the program semantics. For example, the formal specification chosen for defining our bytecode interpreter is Bicolano [77], which is written with the Coq Proof Assistant [11]. This allows checking that the specification is consistent and also proving properties on the behavior of some programs.

3. *Easier to maintain*: new changes in the language semantics can be easily reflected in the interpreter. This will become apparent later when we see that defining a bytecode interpreter in Prolog is a rather easy task and, hence, also maintaining it.

The challenge now is in defining a practical, scalable scheme to interpretive decompilation which achieves quality decompiled programs and, provided this is feasible, we will be able to take advantage of the above features.

There have been several proofs-of-concept of interpretive (de)compilation (e.g., [47, 62]), but there remain interesting open issues when it comes to assess its power and/or limitations to decompile a real language. Such issues are enumerated in Figure 7.1 to facilitate further referencing. This thesis answers these questions positively by proposing

a) *does the approach scale?*

b) *do decompiled programs preserve the structure of the original ones?*

c) *is the "quality" of decompiled programs comparable to that obtained by dedicated decompilers?*

Figura 7.1: Open Issues of Interpretive Decompilation

a modular decompilation scheme which can be steered to control the structure of decompiled code and ensure quality decompilations which preserve the original program's structure.

The rest of the chapter is organized as follows:

- We first provide in Section 7.1 an informal background on PE of logic programs in order to enable the reader to understand the details explained throughout the chapter. A more formal background is given in Section 2 of Paper 6.
- Section 7.2 introduces the challenges behind the specialization a bytecode interpreter through a representative example.
- Section 7.3.3 informally introduces the *Type-based homeomorphic embedding*, an extension of the original *homeomorphic embedding* relation which, by taking information about the behavior of the computation into account, provides more precise results in the presence of infinite signatures. The formal details and an experimental evaluation is presented in Papers 4 and 3.
- Sections 7.4 and 7.5 introduce the necessary ingredients to develop a *modular* and *optimal* decompilation scheme addressing issues *a)*, *b)* and *c)*. The notion of *optimality* is first defined by means of a series of *optimality criteria*. Then, the problems of *non-modular* decompilation are presented and the components needed to enable a *modular* scheme are identified. This includes how to write an interpreter and how to control an *online* partial evaluator in order to preserve the structure of the original program w.r.t. method invocations. We finally introduce an interpretive decompilation scheme which answers

issues *(a)*, *(b)* and *(c)* by producing decompiled programs whose *quality* is equivalent to that of dedicated decompilers. This requires a *block-level* decompilation scheme which avoids code duplication and code re-evaluation.

- Section 7.6 summarizes our experimental results on a prototype implementation of a decompiler of Java bytecode to Prolog which incorporates the above techniques, and demonstrates its scalability and efficiency on an set of realistic Java Bytecode programs.

- Finally, Section 7.7 discusses related work on interpretive decompilation of bytecode languages.

For the sake of concreteness, our interpretive decompilation scheme is formalized in the context of PE of logic programs but the ideas we propose for enabling the practicality of the approach are also of interest for the interpretive (de)compilation of any pair of source and target languages.

7.1. Basics of Partial Evaluation of Logic Programs

We assume familiarity with basic notions of logic programming [67]. Executing a logic program P for an atom A consists in building a so-called SLD tree for $P \cup \{A\}$ and then extracting the computed answer substitutions from every non-failing branch of the tree. Partial evaluation builds upon the execution approach of logic programs with two main differences:

- In order to guarantee termination of the *unfolding* process, when building the SLD-trees, it is possible to choose *not* to further unfold a goal, and rather leave a leaf in the tree with a non-empty, possibly non-failing, goal. The resulting SLD tree is called a *partial* SLD tree. Note that even if the SLD trees for all possible queries are finite, the SLD tree to be built during partial evaluation may be infinite. The reason for this is that since dynamic values are not known at specialization time, the specialization SLD tree can have more branches (in particular, infinite branches) than the actual SLD tree at run-time.

```
1: function EP (P, A, S)
2:     S_0 := S;  i := 0;
3:     repeat
4:         L^pe := unfold(S_i, P, A);
5:         S_{i+1} := abstract(S_i, L^pe, A);
6:         i := i + 1;
7:     until S_i = S_{i-1}      % (modulo renaming)
8:     return codegen(L^pe, unfold);
```

Figura 7.2: A generic PE algorithm for logic programs

Which atom to select from each resolvent and when to stop unfolding is determined by the *unfolding rule*.

- The partial evaluator may have to build several SLD-trees to ensure that all atoms left in the leaves are "covered" by the root of some tree (this is known as the *closedness* condition of EP [66]). The so-called *abstraction operator* performs "generalizations" on the atoms that have to be partially evaluated in order to avoid computing partial SLD trees for an infinite number of atoms. When all atoms are covered, then there is no need to build more trees and the process finishes.

The essence of most algorithms for partial evaluation of logic programs (see e.g. [41]) can be viewed in the algorithm shown in Figure 7.2, which is parametric w.r.t. the unfolding rule, unfold, and the abstraction operator, abstract. EP starts from a program P, a (possibly empty) set of annotations $\mathcal{A}$ and an initial set of calls S. At each iteration, the so-called *local control* is performed by the unfolding rule unfold (Line 4), which takes the current set of atoms S_i, the program and the annotations and constructs a *partial* SLD tree for each call in S_i. In the *global control*, which is performed by the abstraction operator abstract, when some calls in the leaves of the trees are not properly *covered*, the operator abstract adds them to the new set of atoms to be partially evaluated in a proper "generalized" form such that termination is ensured (i.e., the condition $S_i = S_{i-1}$ is reached).

A partial evaluation of P w.r.t. S is then systematically extracted from the resulting set of calls L^{pe} in the final phase, codegen in L8. The notion of *resultant* is used to generate a program rule associated to each root-

to-leaf derivation of the SLD-trees for the final set of atoms L^{pe}. Given an SLD derivation of $P \cup \{A\}$ with $A \in L^{pe}$ ending in B and θ being the composition of the mgu's in the derivation steps, the rule $\theta(A) : -B$ is called the *resultant* of the derivation. A PE is defined as the set of resultants (clauses) associated to the derivations of the constructed partial SLD trees for all $P \cup L^{pe}$. The resulting program is often referred to as the *specialized program* or *residual program*. A more formal background is given in Section 2 of Paper 6.

7.1.1. Online vs. Offline Partial Evaluation

It is well-known that both the quality of the specialized programs and the time required for the PE process greatly vary with the control strategies used. Traditionally, two approaches to PE have been considered, *online* and *offline* PE. In online PE, all control decisions are taken on the fly during the specialization phase by keeping track of the specialization history. In the offline approach, all control decisions are taken before the proper specialization phase. These control decisions are based on abstract descriptions of the data instead of the actual data. The control strategy is usually represented as program annotations which are the sole decision criteria for control of the partial evaluator. For instance, in the local control, an annotation can explicitly indicate that an atom should not be unfolded. In the global control, annotations typically specify for each call which arguments have to be generalized away (i.e. replaced by variables). Such annotations are in some partial evaluators automatically generated by a *binding-time analysis* and in other partial evaluators they are manually provided by the user, either in part or in full.

Under this classification, the PE algorithm we propose can be considered a generic or a hybrid approach since the $\mathcal{A}$ annotations can provide information to the control operators, as in offline PE, and the algorithm can include control rules based on the actual specialization history, as in online PE. The advantages of the offline approach are that, once all control annotations are available, PE is quite simple and efficient. On the other hand, online PE, though less efficient, has a strictly more powerful control strategy since control decisions are based on actual data instead of abstract descriptions of data. Therefore, though all offline PEs can be re-

plicated using online techniques, many online PEs cannot be reproduced using offline techniques.

In this work we are interested in investigating how far we can go with the more powerful but less efficient online PE approach. The motivation for this is that this way we may obtain decompilations of higher quality than those achievable using offline PE. Thus, our challenges are both in terms of quality of the decompiled programs and in terms of efficiency of the decompilation process. As we will see later, many of the lessons learned in this thesis are of interest both to the online and offline approaches to the PE of interpreters.

7.2. Challenges in the Specialization of Bytecode Interpreters

This section illustrates the challenges which appear in the specialization of a bytecode interpreter by means of an example. Fig. 7.3 shows a fragment of a bytecode interpreter implemented in Prolog. We assume that the code for every method in the bytecode program is represented as a set of facts `bytecode/3` such that, for every pair $pc_i{:}bc_i$ in the code for method m, we have a fact `bytecode(`m`,`pc_i`,`bc_i`)`. The state carried around by the interpreter is of the form `st(Fr,FrStack)` where `Fr` represents the current frame (environment) and `FrStack` the stack of frames (call-stack) implemented as a list. Frames are of the form `fr(M,PC,OStack,LocalV)`, where `M` represents the current method, `PC` the program counter, `OStack` the operand stack and `LocalV` the list of local variables. Predicate `main/3`, given the method to be interpreted `Method` and its input method arguments `InArgs`, first builds the initial state by means of predicate `build_s0/3` and then calls predicate `execute/2`, returning `Res`, which is the top of the operand stack at the end of the computation. In turn, `execute/2` calls predicate `step/3`, which produces `S'`, the state after executing the bytecode, and then calls predicate `execute/2` recursively with `S'` until we reach a `return` instruction with the empty stack. For brevity, we only show the definition of `step/3` for a selected set of instructions and omit the code of some auxiliary predicates. Namely `build_s0/3`, which was explained above, `next/3`, which produces the next program counter given the current one, and `split_OS/4`, which

```
main(Method,InArgs,Res) :-
   build_s0(Method,InArgs,S0),
   execute(S0,Sf),
   Sf = st(fr(_,_,[Res|_],_),_)).

execute(S,S) :-
   S = st(fr(M,PC,_,_),[]),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(fr(M,PC,_,_),_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).

step(goto(PC),S,S') :-
   S = st(fr(M,_,OS,LV),FrS),
   S' = st(fr(M,PC,OS,LV),FrS).
```

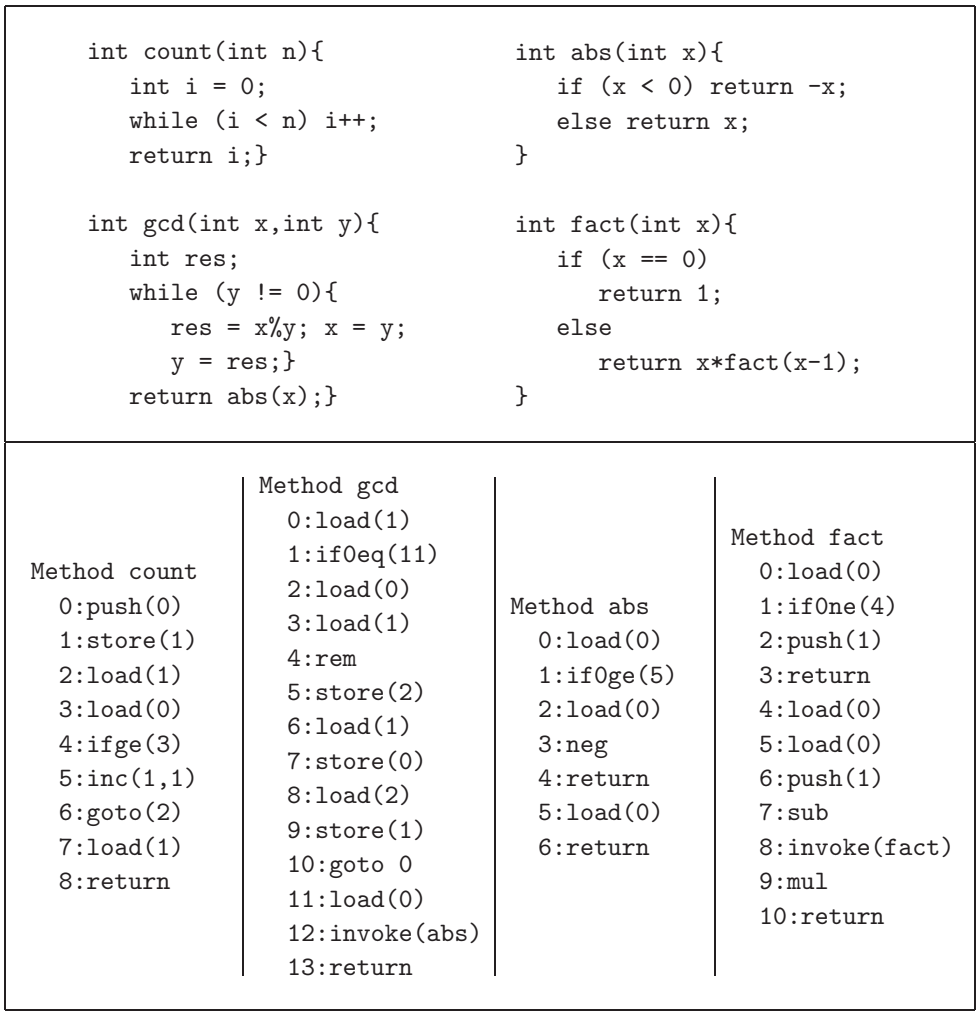
```
step(push(X),S,S') :-
   S = st(fr(M,PC,OS,L),FrS),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[X|OS],L),FrS).
...
step(invoke(M'),S,S') :-
   S = st(fr(M,PC,OS,LV),FrS),
   split_OS(M',OS,Args,OS''),
   build_s0(M',Args,
            st(fr(M',PC',OS',LV'),_)),
   S' = st(fr(M',PC',OS',LV'),
           [fr(M,PC,OS'',LV)|FrS]).
step(return,S,S') :-
   S = st(fr(_,_,[RV|_],_),
          [fr(M,PC,OS,LV)|FrS]),
   next(M,PC,PC'),
   S' = st(fr(M,PC',[RV|OS],LV),FrS).
```

Figura 7.3: Fragment of a bytecode interpreter

splits the current operand stack into the parameters list to be used in the called method and the rest.

Figure 7.4 depicts the bytecode program that we will use as our working example. On the top of the figure we depict the Java source code for clarity. Note that the decompiler works directly on the bytecode which is shown at the bottom. Our working example consists of a set of methods that carry out different arithmetic operations. Method `gcd` computes the greatest-common divisor, `abs` the absolute value and `fact` the factorial recursively. Method `count` has no particular meaning, it just increments a counter initialized to 0 until its value reaches the value of the given argument.

In order to achieve an *effective* decompilation, one of the crucial requirements is to have available control strategies (i.e., unfold and abstract operators) which are powerful enough to remove the interpreter overhead. For this reason, our first experiments in Paper 1 were performed using "aggressive" control strategies based on *homeomorphic embedding* [57, 61]. In local control, by aggressiveness we mean unfolding rules which compute derivations as long as possible provided there are no termination problems.



```
int count(int n){
   int i = 0;
   while (i < n) i++;
   return i;}

int gcd(int x,int y){
   int res;
   while (y != 0){
      res = x%y; x = y;
      y = res;}
   return abs(x);}
```

```
int abs(int x){
   if (x < 0) return -x;
   else return x;
}

int fact(int x){
   if (x == 0)
      return 1;
   else
      return x*fact(x-1);
}
```

```
Method count
  0:push(0)
  1:store(1)
  2:load(1)
  3:load(0)
  4:ifge(3)
  5:inc(1,1)
  6:goto(2)
  7:load(1)
  8:return
```

```
Method gcd
  0:load(1)
  1:if0eq(11)
  2:load(0)
  3:load(1)
  4:rem
  5:store(2)
  6:load(1)
  7:store(0)
  8:load(2)
  9:store(1)
  10:goto 0
  11:load(0)
  12:invoke(abs)
  13:return
```

```
Method abs
  0:load(0)
  1:if0ge(5)
  2:load(0)
  3:neg
  4:return
  5:load(0)
  6:return
```

```
Method fact
  0:load(0)
  1:if0ne(4)
  2:push(1)
  3:return
  4:load(0)
  5:load(0)
  6:push(1)
  7:sub
  8:invoke(fact)
  9:mul
  10:return
```

Figura 7.4: Source code and bytecode for working example

In global control, it denotes abstraction operators which generalize in as few situations as possible without endangering termination.

Figure 7.5 depicts the decompiled program we obtain using the state-of-the-art partial evaluator available in the `CiaoPP` system [48]. For this preliminary experiments the *homeomorphic embedding* was used both for the local and global control levels. By looking at the code we observe the following:

1. The partial evaluator has not been able to successfully decompile the

```
main(count,[N],A) :-
    % out of memory error

main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% Full interpreter
```

Figura 7.5: Decompiled code for working example. First attempt.

`count` method. It runs out of memory because there are termination problems both at the local and at the global control. The reason for this is that the control rules based on the *homeomorphic embedding* do not ensure termination of PE in programs which can potentially generate infinite values, as it is the case of the bytecode interpreter (in particular because of the `is/3` predicate).

2. The method compositionality of the original program has been lost in the decompiled program. This can be seen by looking at the decompiled code of the `gcd` method. Note that it does not call `abs` but instead it has inlined its code. Though this might be seen as positive from the point of view of the specialization, the consequences can highly degrade the efficiency and quality of the process and makes impossible to scale up when considering realistic bytecode programs with calls to libraries.

3. The decompiled code we obtain for the `fact` method basically contains the full interpreter and is not shown in the figure due to space limitations. The problem was first detected in [42] and arises when the method to be decompiled is recursive. As we will see, this problem (and also its solution) is very related to the composisionality problem explained in the previous item.

4. If we look at the code of the `main(gcd,...)` and `execute/3` predicates, we can see that there are code duplications. Even more, the

partial evaluator has produced such duplications because some part of the bytecode program has been re-evaluated during EP. Our experimental evaluation demonstrates that having or not these duplications (re-evaluations) makes the difference between being or not being able to scale up in practice.

The solutions to the above problems are summarized in the following three challenges:

- **Challenge I. Handling infinite signatures in PE:** We first review existing solutions identifying their flaws and then introduce the *type-based homeomorphic embedding*. This issue is further discussed in Section 7.3 and elaborated in more detail in Papers 3 and 4.
- **Challenge II: A modular decompilation scheme.** Even after achieving *Challenge I*, we will see that it is a necessity to design a modular decompilation scheme which preserves the method compositionality of the original programs and, besides, solves the problem with recursive programs. Such a decompilation scheme is introduced in Section 7.4 and further studied in detail in Paper 6.
- **Challenge III: Optimal decompilation.** Preliminary experiments performed using the modular decompilation scheme with realistic programs show that it is yet not possible to successfully scale up. We therefore introduce an *optimal* decompilation scheme which ensures that the decompilation times and decompiled program sizes grow linearly w.r.t. to the size of the input programs by avoiding code duplications and re-evaluations. This issue is introduced in Section 7.5 and further studied in depth in Paper 6.

7.3. Challenge I: Handling Infinite Signatures

7.3.1. The Homemorphic Embedding

The *homeomorphic embedding* (HEm) relation [57, 60, 61] has become very popular to ensure online termination of *symbolic* transformation and

specialization methods and it is essential to obtain powerful optimizations, for instance, in the context of online PE. Intuitively, HEm is a structural ordering under which an expression t_1 *embeds* expression t_2, written as $t_2 \trianglelefteq t_1$, if t_2 can be obtained from t_1 by deleting some operators, e.g., $\underline{\mathtt{s}}(\mathtt{s}(\underline{\mathtt{U}}+\mathtt{W})\underline{\times}(\underline{\mathtt{U}}+\underline{\mathtt{s}}(\underline{\mathtt{V}})))$ embeds $\mathtt{s}(\mathtt{U} \times (\mathtt{U} + \mathtt{V}))$.

The HEm relation can be used to guarantee termination because, assuming that the set of constants and functors is finite, every infinite sequence of expressions $t_1, t_2, \ldots$, contains at least a pair of elements t_i and t_j with $i < j$ s.t. $t_i \trianglelefteq t_j$. Therefore, when iteratively computing a sequence $t_1, t_2, \ldots, t_n$, finiteness of the sequence can be guaranteed by using HEm as a "whistle". Whenever a new expression t_{n+1} is to be added to the sequence, we first check whether $t_i \not\trianglelefteq t_{n+1}$ for all i s.t. $1 \leq i \leq n$. If that is the case, finiteness is guaranteed and computation can proceed. Otherwise, HEm is not capable of guaranteeing finiteness and the computation has to be stopped. The intuition is that computation can proceed as long as the new expression is not larger than any of the previously computed ones since that is a sign of potential non-termination. The success of HEm is due to the fact that sequences can usually grow considerably large before the whistle blows, when compared to other online approaches for guaranteeing termination.

While HEm has been proved very powerful for symbolic computations, some difficulties remain in the presence of infinite signatures such as the numbers. In the case of logic programs, infinite signatures appear as soon as certain Prolog built-ins such `is/2`, `functor/3` and `name/2` are used. HEm relations over infinite signatures have been defined (e.g. [60, 6]), but they tend to be too conservative in practice ("whistling" too early).

7.3.2. A Challenging Example

Consider the `count` method which appears in the left hand side of Figure 7.4. It can be seen that `count` receives an integer and executes a loop where a counter initialized to "0" (in bytecodes `0` and `1`) is incremented by one at each iteration (bytecode `5`) until the counter reaches the value of the input parameter (checking the condition comprises bytecodes `2`, `3` and `4`). The method returns the value of the counter in bytecodes `7` and `8`. In order to decompile the `count` method, we partially evaluate the interpreter

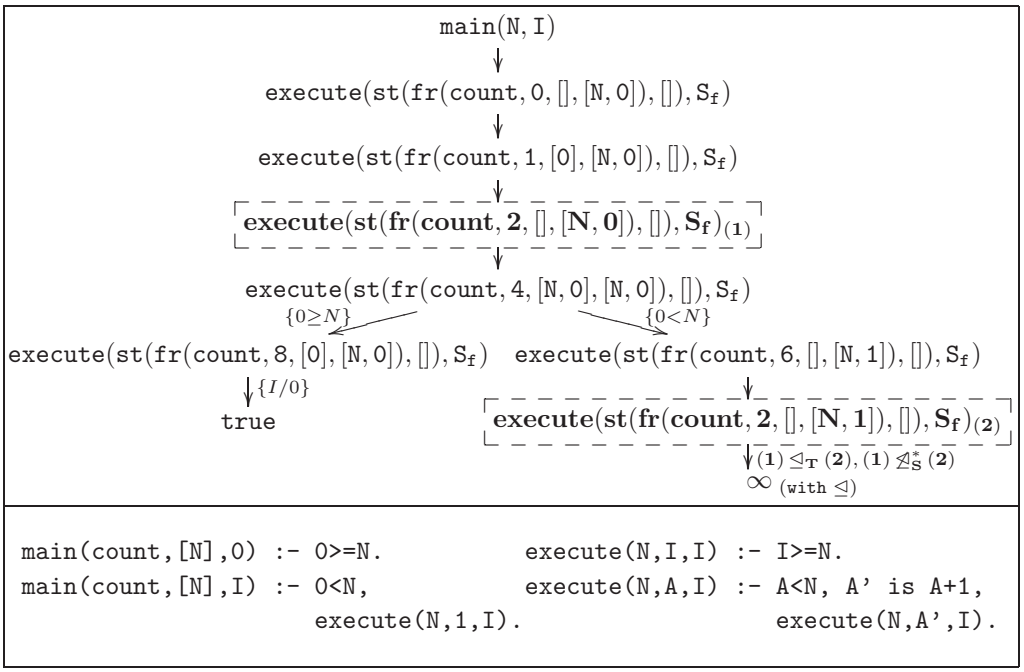


Figura 7.6: Partial unfolding SLD tree and residual code of working example

in Figure 7.3 w.r.t. the `count` bytecode method by specializing the atom `main(count,[N],I)`, where `N` is the input parameter and `I` represents the returned value (i.e., the top of the stack at the end of the computation).

In Figure 7.6, we depict (a reduced version of) one of the SLD trees that leads to an effective decompilation of the `count` method and that we will refer to below. For simplicity, apart from the entry atom `main/3`, we only show atoms for `execute/2`, as it is the only recursive predicate in the program. Thus, each arrow in the tree involves the application of several unfolding steps. Note that some of the statements within the body of each `step` operation can remain residual when they involve data which is not known at specialization time. The computation rule used in the unfolding operator is able to residualize calls which are not sufficiently instantiated and select non-leftmost atoms in a safe way [7], in particular, further calls to `execute` can be selected. We represent such residual calls as labels in the arrows of the tree.

Using the Original HE

Let us first consider an online partial evaluator which uses HEm to control termination both at the local and global control levels. As it can be seen in the figure, the PC value "2" corresponds to the loop entry. By applying HEm, the evaluation contains a subsequence of atoms of the form: execute(st(fr(count, 2, [], [N, 0]), []), S_f), execute(st(fr(count, 2, [], [N, 1]), []), S_f), execute(st(fr(count, 2, [], [N, 2]), []), S_f), ... marked within dashed frames in the figure, which correspond to consecutive iterations of the loop in which the control returns to the loop head (PC value 2 in the first position of the state) with a value for the loop counter (local variable at the second position in the resulting state) increased by one. This sequence can grow infinitely, as the HEm does not flag it as potentially dangerous, which is marked by "∞ (with $\trianglelefteq$)" in the figure. This is because the interpreter uses Prolog's arithmetic (i.e., the is/2 predicate), which breaks the finite signature property featured by pure logic programs.

In order to get a quality decompilation, we need to filter out the value of the counter (local variable 1) but not that of the PC. As shown in the figure, this requires stopping the derivation when we hit the atom execute(st(fr(count, 2, [], [N, 1]), []), S_f) (marked as $(\mathbf{1}) \trianglelefteq_{\mathbf{T}} (\mathbf{2})$) and generalize it w.r.t. the above atom within a dashed frame, resulting in execute(st(fr(count, 2, [], [N, X]), []), S_f).

Recovering Termination: Embedding with Number Filtering

In programs which contain Prolog arithmetic but do not generate an infinite number of functors via functor/3, =../2, etc., a relatively straightforward solution in order to recover termination is to use the $\trianglelefteq_{num}$ relation, which is an adaptation of HEm which filters out numeric values, i.e., any number embeds any other number. The atom execute(st(fr(count, 2, [], [N, 1]), []), S_f) embeds execute(st(fr(count, 2, [], [N, 0]), []), S_f) under $\trianglelefteq_{num}$ and therefore we avoid non-termination. Unfortunately, this modification to HEm, is far too conservative, and leads to excessive precision loss. For instance, in the specialization of main(count, [N], I), the first two atoms for execute/2 are execute(st(fr(count, 0, [], [N, 0]), []), S_f) and execute(st(fr(count, 1, [0], [N, 0]), []), S_f). By using $\trianglelefteq_{num}$, the whistle

blows at this point and unfolding has to stop. Furthermore, the latter atom is generalized at the global control level into $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{X},\texttt{Y},[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$ before proceeding with the specialization. This turns out not to be acceptable for the specialization of our interpreter, since we lose track of which the next instruction to execute is—which prevents us from eliminating the interpretation layer—and in many cases the residual program ends up containing the whole original interpreter.

Increasing Accuracy: Static Symbols in the Program

A simple syntactic way of increasing the accuracy while preserving termination, as proposed in [60], consists in considering two sets of symbols: those which appear explicitly in the program and goal, which is obviously finite, and another infinite set which contains all other symbols. In the following, this relation is denoted as $\trianglelefteq_S^*$. When comparing two terms we keep those symbols which belong to the finite set and filter out all other ones. Under this relation, the atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{1},[\texttt{0}],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$ does not embed the atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{0},[],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$ in the figure, as the numbers `0` and `1` are different static symbols which occur in the program. Hence, we are not forced to generalize them and we can keep the `PC` value.

Unfortunately, the $\trianglelefteq_S^*$ relation turns out not to be optimal in our case either since $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{1}]),[]),\texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$. This means that unfolding proceeds with a second iteration of the loop. The process is guaranteed to terminate, we will unfold at most as many iterations of the loop as distinct numbers appear in the program. However, we are not able to achieve the quality decompilation which appears at the bottom of Figure 7.6. For obtaining such good decompilation, we need to generalize the loop counter, i.e., the atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{1}]),[]),\texttt{S}_\texttt{f})$ has to embed $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count},\texttt{2},[],[\texttt{N},\texttt{0}]),[]),\texttt{S}_\texttt{f})$. Intuitively, the reason why this relation does not behave optimally is because many symbols which appear explicitly in the program for one argument (in our case the PC counter) should not affect the set of symbols which we should consider as static for other arguments (the list of local variables).

In conclusion, this example suggests that embeddings that take context

information into account are needed: a context-sensitive embedding should handle in a different way the PC values and the numeric values in program variables such as the loop counter.

7.3.3. Type-based Homeomorphic Embedding

In the presence of infinite signatures, a general method of defining homeomorphic embedding relations exists; an *extended homeomorphic embedding relation* is defined in [60] based on previous results by Kruskal [57] and by Dershowitz [37]. This solution defines a family of embedding relations, where a subsidiary ordering on function symbols plays an essential role. However, we argue that this does not really solve the practical problem of finding an effective embedding relation, since there is no automated mechanism for finding the "right" ordering relation on the function symbols in the signature.

In this thesis, we propose the *type-based homeomorphic embedding* (TbHEm for short), a relation which improves HEm by making use of additional information provided in the form of types. We outline how this approach can be seen as a way of generating instances of extended HEm as defined by Leuschel, including the possibility of taking into account the program semantics. The types required for guiding TbHEm can be provided manually or, interestingly, be automatically inferred by program analysis, as we discuss in Paper 3.

A starting point of TbHEm is the observation that, even if an expression is defined over an infinite signature, it might only take a finite set of values over such domain for each computation. To perform such a distinction our typed relation is defined on types which are structured into a (possibly empty) finite part and a (possibly empty) infinite partition. Intuitively, TbHEm allows expanding sequences as long as, whenever we compare sub-terms from an infinite type, the concrete values which appear in the expression remain within the finite part of the type.

Using the TbHEm to control the PE of the bytecode interpreter

In the case of our bytecode interpreter, the PC argument can be defined by a structured type such that the bounded interval in which it ranges constitutes its finite partition and the remaining integers form its infinite

part. This way, the TbHEm will not generalize the `PC` as long as its value remains within the bounded interval.

In order to infer such type, let us rely on existing analysis techniques, namely on the inference of well-typings described by Bruynooghe *et al.* [20][1]. The following type τ_{PC} for the program counter argument is inferred for the interpreter of Figure 7.3, together with the particular bytecode program of Figure 7.4:

```
τPC --> -4; 0; 1; 2; 3; 4; 5; 6; 7; 8; num
```

Type τ_{PC} can be naturally interpreted as consisting of a finite part (the named constants) and an infinite part (the numbers other than the named constants). In other words, the partition F of the rule is $\{-4, 0, 1, 2, \ldots, 8\}$ and $I = \texttt{num} \setminus F$. Using the rule structured in this way, TbHEm ensures that the program counter is never abstracted away during partial evaluation, so long as its value remains in the expected range (the named constants). The atom $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, \texttt{1}, [\texttt{0}], [\texttt{N}, \texttt{0}]), []), \texttt{S}_\texttt{f})$ does not embed $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, \texttt{0}, [], [\texttt{N}, \texttt{0}]), []), \texttt{S}_\texttt{f})$ by using the type definition above, thus, the derivation can proceed. This avoids the need for generalizing the `PC` what would prevent us from having a quality specialization (decompilation) as explained above. The derivation will either eventually end or the `PC` value will be repeated due to a backwards jump in the code (loops). In this case, the TbHEm, also written $\trianglelefteq_T$, will flag the relevant atom as dangerous, e.g., $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, \texttt{2}, [], [\texttt{N}, \texttt{0}]), []), \texttt{S}_\texttt{f}) \trianglelefteq_\texttt{T} \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{count}, \texttt{2}, [], [\texttt{N}, \texttt{1}]), []), \texttt{S}_\texttt{f})$, as can be seen in Figure 7.6.

The decompiled program that we obtain using the inferred typings and combined with TbHEm is shown at the bottom of Figure 7.6. We can observe that the decompilation is optimal[2] in the sense that the interpretation layer has been completely removed and there is no superfluous residual code.

Besides the inference of well-typings we saw above, Paper 3 also outlines how analysis of numeric bounds can be used to infer useful information for TbHEm. Such analysis makes over-approximations of the set of values that

[1] Available on-line at `http://saft.ruc.dk/Tattoo/`

[2] We will see later that this can be further improved

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[N],A) :- ...% full int.
```

```
main(gcd,[A,0],A) :- A>=0.
main(gcd,[B,0],A) :-
    B<0, A is -B.
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).

execute_1(A,0,A) :- A>=0.
execute_1(A,0,C) :- A<0, C is -A.
execute_1(A,B,G) :-
    B\=0, I is A rem B,
    execute_1(B,I,G).
```

Figura 7.7: Decompiled code for working example after overcoming Chall. I

the program arguments can have. Intuitively, when we can prove that such set of values is bounded, then we know that the infinite partition of the type is empty and, hence, we can safely apply traditional HEm (and improve the effectiveness of PE).

Note that, determining the exact set of symbols that can appear at runtime at a specific program point, and in particular determining whether the set is finite, is closely related to termination detection and is thus undecidable. However, the better the derived types are, the more aggressive partial evaluation can be without risking non-termination. If the derived types have finite components that are too small, then over-generalization is likely to result; if they are too large, then specialization might be over-aggressive, producing unnecessary versions.

7.4. Challenge II: Modular Decompilation

Once we have overcome the problem of handling infinite signatures, the class of bytecode programs which can be successfully decompiled is significantly wider. Another issue, which is not further discussed in this introduction, is that using the classical online abstraction operator which is simply based on the original HEm, or even enhancing it by using the TbHEm, the decompiled programs we obtain tend to have too many (re-

dundant) specialized versions of some predicates. This is studied in detail in Paper 2 where we propose an advanced abstraction operator which is able to control the *polyvariance* of the PE process, i.e., which is able to avoid having such redundant specialized versions. As it is shown in Paper 2, this allows obtaining better decompiled programs, in a more efficient way, which also widens a bit more the class of programs which we can successfully decompile. Nevertheless, even after enhancing the partial evaluator so that it integrates both the TbHEm and such advanced abstraction operator, the current scheme can still be rather unsatisfactory with realistic programs, since the compositionality of the original programs as regards method calls is lost.

Let us consider again our bytecode example in Figure 7.4. The decompiled code we obtain using the enhanced partial evaluator is shown in Figure 7.7. It can be noted that it is the same as the one in Figure 7.5 except for the method `count` for which the code at the bottom of Figure 7.6 is now obtained. Note that, this example is not complex enough to expose the problem of polyvariance that the advanced abstraction operator solves. We refer the reader again to Paper 2 where a representative example is presented.

We now identify four limitations, which we name as **(L1)**... **(L4)**, of the current decompilation (from now on *non-modular* decompilation). It is important to note that such limitations, and the way to avoid them which we introduce later, are also relevant to the case of offline PE.

(L1) Calls to methods are *inlined* within their calling contexts and, as a consequence, the structure of the original code is lost. For example, the method invocation from `gcd` to `abs` (index 12 of `gcd`) does not appear in the decompiled code. As a result, the decompiled code for `gcd` has two base cases in which the builtins of `abs` are inlined, namely, `A>=0`, `B<0` and `A is -B`. This happens because calls to methods are dealt with in a *small-step* fashion within the interpreter, i.e., the code of invoked methods is unfolded as if it was inlined inside the "caller" method.

(L2) As a consequence, decompilation becomes very inefficient. E.g., if n calls to the same method appear within a code, such method will be decompiled n times. Even worse, if there is a method invocation inside a loop, its code will be evaluated twice in the best case, as we have to perform the corresponding generalizations in the global control before reaching a

fixpoint. This can be even worse in the case of nested loops.

(L3) The non-modular approach does not work incrementally, in the sense that it does not support *separate* decompilation of methods but rather has to (re)decompile all method calls. Thus, decompiling a real language becomes unfeasible, as one needs to consider system libraries, whose code might not be available. Limitation L2 together with L3 answer issue *(a)* of Figure 7.1 negatively.

(L4) The decompiled code contains basically the whole interpreter when there are recursive methods. This is why the decompiled program in Figure 7.7 does not contain the code corresponding to the recursive `fact` method. The problem with recursion is as follows. Assume we want to decompile method $m1$ whose code is $\langle pc_0 : bc_0, \ldots, pc_j : invoke(m1), \ldots, pc_n : return\rangle$. There is an initial decompilation for $A_k = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv}), []), \texttt{S}_\texttt{f})$ in which the call-stack is empty. During its decompilation, a call of the form $A_l = \texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}', \texttt{lv}'), [\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \texttt{os}, \texttt{lv})]), \texttt{S}_\texttt{f})$ with the call-stack containing the previous frame appears when we arrive to the recursive call. At this point, the derivation must be stopped as $A_k \trianglelefteq_T A_l$. In order to ensure termination, global control generalizes the above calls into $\texttt{execute}(\texttt{st}(\texttt{fr}(\texttt{m1}, \texttt{pc}_\texttt{j}, \_, \_), \_), \texttt{S}_\texttt{f})$, where $\_$ denotes a fresh variable and thus the call-stack has become unknown. As a consequence, after evaluating the *return* statement, the continuation obtained from the call-stack is unknown and we produce the call $\texttt{execute}(\texttt{st}(\texttt{fr}(\_, \_, \_, \_), \_), \texttt{S}_\texttt{f})$ to be decompiled. Here, the fact that the method and the program counter are unknown prevents us from any chance of removing the interpretation layer, i.e., the decompiled code will potentially contain the whole interpreter. This indeed happens during the decompilation of `fact`. Limitations L1 and L4 answer issue *(b)* (see Figure 7.1) negatively.

We now identify the ingredients which are necessary in order to achieve a *modular* decompilation scheme. By *modular* decompilation, we refer to a decompilation technique whose decompilation unit is the method, i.e., we decompile a method at a time. We show that this approach overcomes the four limitations of non-modular decompilation described above and answers issues *(a)* and *(b)* of Figure 7.1 positively. In essence, we need to: (i) Give a compositional treatment to method invocations. We show that this can be achieved by considering an interpreter implemented using a *big-step* semantic. (ii) Provide a mechanism to residualize calls in the decompiled

program (i.e., do not unfold them and add them without modifications to the residual code). We automatically generate program annotations for this purpose. (iii) Study the conditions which ensure that *separate* decompilation of methods is sound.

7.4.1. Big-step Semantics Interpreter to Enable Modularity

Traditionally, two different approaches have been considered to define language semantics, *big-step* (or *natural*) semantics and *small-step* (or *structural operational*) semantics (see, e.g., [58]). Essentially, in big-step semantics, transitions relate the initial and final states for each statement, while in small-step semantics transitions define the *next* step of the execution for each statement. In the context of bytecode interpreters, it turns out that most of the statements execute in a single step, hence making both approaches equivalent for such statements. This is the case for our bytecode interpreter in Figure 7.3 for all statements except for *invoke*. The transition for *invoke* in small-step defines the next step of the computation, i.e., the current frame is pushed on the call-stack and a new environment is initialized for the execution of the invoked method. Note that, after performing this step, we do not distinguish anymore between the code of the caller method and that of the callee. This prevents us from having modularity in decompilation.

In the context of interpretive (de)compilation of imperative languages, small-step interpreters are commonly used (see e.g. [76, 47]). We argue that the use of a big-step interpreter is a necessity to enable modular decompilation which scales to realistic languages. In Fig. 7.8, we depict the relevant part of a big-step version for our bytecode interpreter. We can see that the *invoke* statement, after extracting the method parameters from the operand stack, calls recursively predicate `main/3` in order to execute the callee. Upon return from the method execution, the return value is pushed on the operand stack of the new state and execution proceeds normally. Also, we do not need to carry the call-stack explicitly within the state, but only the information for the current environment, i.e., states are of the form `st(M,PC,OStack,LocalV)`. This is because the call-stack is already available by means of the calls for predicate `main/3`.

The compositional treatment of methods within the interpreter is not only essential to enable modular decompilation (overcome L1, L2 and L3) but also to solve the recursion problem in a simple and elegant way. Indeed, the decompilation based on the big-step interpreter does not present L4. E.g., the decompilation of a recursive method $m1$ starts from the call $\texttt{main}(\texttt{m1}, \_, \_)$ and then reaches a call $\texttt{main}(\texttt{m1}, \texttt{args}, \_)$ where `args` represents the particular arguments in the recursive call. This call is flagged as dangerous by local control and the derivation is stopped. The important points are that, unlike before, no re-computation is needed as the second call is necessarily an instance of the first one and, besides, there is no information loss associated to the generalization of the call-stack, as there is no stack.

Partial solutions to the recursion problem exist and are discussed in the following. The problem was first detected in [42] and a solution based on computing regular approximations during PE was proposed. Although feasible in theory, such technique might be too inefficient in practice and problematic to scale it up to realistic applications due to the overhead introduced by the underlying analysis. Another solution is proposed in [47], where a simpler control-flow analysis is performed before PE in order to collect all possible instructions which might follow the *return*. Such information may then be used to recover information lost by the generalization. This solution turns out to be also impractical for our purposes when considering realistic programs that make intensive use of library code (e.g. Java Bytecode) as many continuations can follow. Our solution does not require the use of static analysis and, as our experiments show, does not pose scalability problems.

It is important to note that the idea of using a big-step semantics for describing the interpreter in order to achieve modular (de)compilation is equally useful in the offline approach to interpretive decompilation. Furthermore, to the best of our knowledge, our idea is novel and has not been proposed before, neither in online nor in offline PE of interpreters.

7.4.2. The Modular Decompilation Scheme

In addition to use a big-step interpreter, it is necessary in order to design a modular decompilation scheme to: 1) provide a mechanism to

```
execute(S,S) :-
   S = st(M,PC,[_Top|_],_),
   bytecode(M,PC,return).
execute(S,Sf) :-
   S = st(M,PC,_,_),
   bytecode(M,PC,Inst),
   step(Inst,S,S'),
   execute(S',Sf).
```

```
step(invoke(M'),S,S') :-
   S = st(M,PC,OS,LV),
   next(M,PC,PC'),
   split_OS(M',OS,Args,OSRs),
   main(M',Args,RV),
   S' = st(M,PC',[RV|OSRs],LV).
```

Figura 7.8: Fragment of big-step bytecode interpreter

residualize calls in the decompiled program (i.e., do not unfold them and add them without modifications to the residual code), and 2) define the notion of *separate* decompilation and study the conditions which ensure its soundness.

Paper 6 studies in detail these issues and defines a modular decompilation scheme whose correctness and completeness is formally proven. It is also proven that the proposed scheme satisfies the *method-optimality* criterion, which ensures that each method is decompiled only once.

Modular decompilation basically works as follows: when a method invocation is to be decompiled, the call `step(invoke(m'),_,_)` occurs during unfolding. We can see that, by using the big-step interpreter in Fig. 7.8, a subsequent call `main(m',_,_)` will be generated. At this point, there will be an annotation indicating to the partial evaluator to not to unfold this call and rather add it without modifications to the residual code. If `m'` is internal (i.e., it is defined in the input program), a corresponding decompilation from the call `main(m',_,_)` will be, or has already been, performed since modular decompilation ensures that the PE is executed for every method in the bytecode program.

Figure 7.9 shows the decompiled program we obtain using the modular decompilation scheme with our working example. It can be observed that the structure of the original program w.r.t. method calls is now preserved, as the residual predicate for `gcd` contains an invocation to the definition of `abs`, as it happens in the original bytecode. Moreover, we now obtain an effective decompilation for the recursive method `fact` where the interpretive layer is completely removed. Thus, L1 and L4 have been successfully

```
main(count,[N],0) :- 0>=N.
main(count,[N],I) :-
    0<N, execute_2(N,1,I).

execute_2(N,I,I) :- I>=N.
execute_2(N,A,I) :-
    A<N, A' is A+1,
    execute_2(N,A',I).

main(gcd,[B,0],A) :-
    main(abs,[B],A).
main(gcd,[B,C],A) :-
    C\=0, D is B rem C,
    execute_1(C,D,A).
```

```
execute_1(A,0,C) :- main(abs,[A],C).
execute_1(A,B,F) :- B\=0,
                    H is A rem B,
                    execute_1(B,H,F).

main(abs,[A],A) :- A>=0.
main(abs,[B],A) :- B<0, A is -B.

main(fact,[B],A) :- B\=0,
                    C is B-1,
                    main(fact,[C],D),
                    A is B*D.
main(fact,[0],1).
```

Figura 7.9: Decompiled code obtained using modular decompilation

solved.

Note that modular decompilation gives a monovariant treatment to methods in the sense that it does not allow creating specialized versions of method definitions. This is against the usual spirit in PE, where polyvariance is a main goal to achieve further specialization. However, in the context of decompilation, we have shown that a monovariant treatment is necessary to enable scalability and to preserve program structure. It naturally raises the question whether a polyvariant treatment could achieve, even if at the cost of efficiency and loss of structure, a better quality decompilation. Note that enabling polyvariant specialization in the modular setting can be trivially done by not introducing the corresponding annotations for certain selected methods which should be treated in a polyvariant manner. Our experience indicates that there is often a small quality gain at the price of a highly inefficient decompilation.

7.5. Challenge III: An Optimal Decompilation Scheme

As we already mentioned in Section 7.2, and as we can see by looking at Figure 7.9, the decompiled programs we obtain using the modular scheme

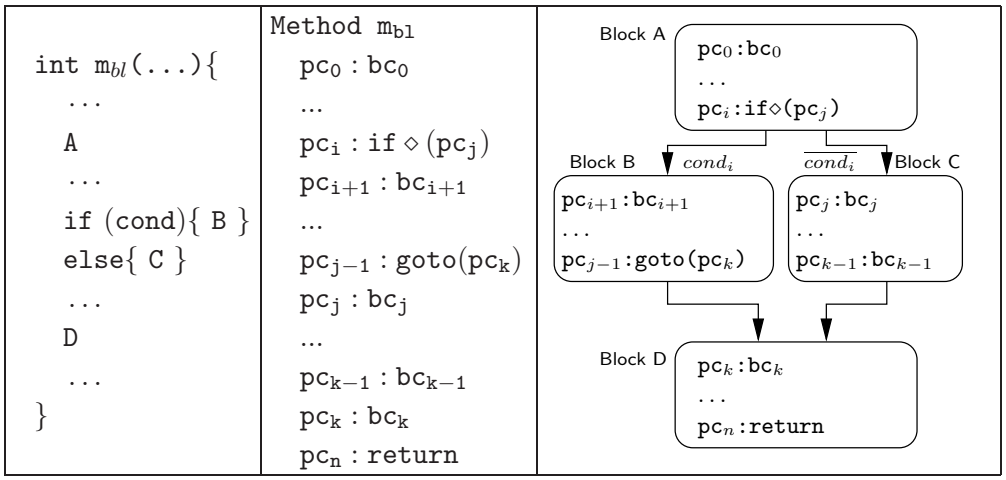


Figura 7.10: Source code, bytecode and CFG of m_{bl} method

are still not optimal as they can contain code duplications. See for example the code on the right-hand side of the rules defining `main(gcd,...)` and `execute_1/3`. Duplications are (very often) produced because part of the code is re-evaluated during PE. Unfortunately, as we will see later, such duplications and re-evaluations grow exponentially with the number of branching and merging points respectively, and as our experiments show, highly degrade the effiency of the process and the quality of the decompiled code. The main issue is whether it is possible to obtain, by means of interpretive decompilation, programs whose quality is equivalent to that obtained by dedicated decompilers; issue *(c)* in Figure 7.1. In order to obtain comparable results, it makes sense to use similar heuristics. Since decompilers first build a *control flow graph* (CFG) for the method, which guides the decompilation process, we now study how a similar notion can be used for controlling PE of the interpreter.

Let us explain the problem by means of an example. Consider the method m_{bl} in Fig. 7.10. The source code is shown to the left, the relevant bytecode in the center and its CFG to the right. As customary, the CFG [1] consists of basic blocks which contain a sequence of non-branching bytecode instructions and which are connected by edges which describe the possible flows originated from the branching instructions (like conditional jumps, exceptions, virtual method invocation, etc.). In our small bytecode programs, they correspond with conditional jumps (i.e. `if`$\diamond$ and `if0`$\diamond$). A

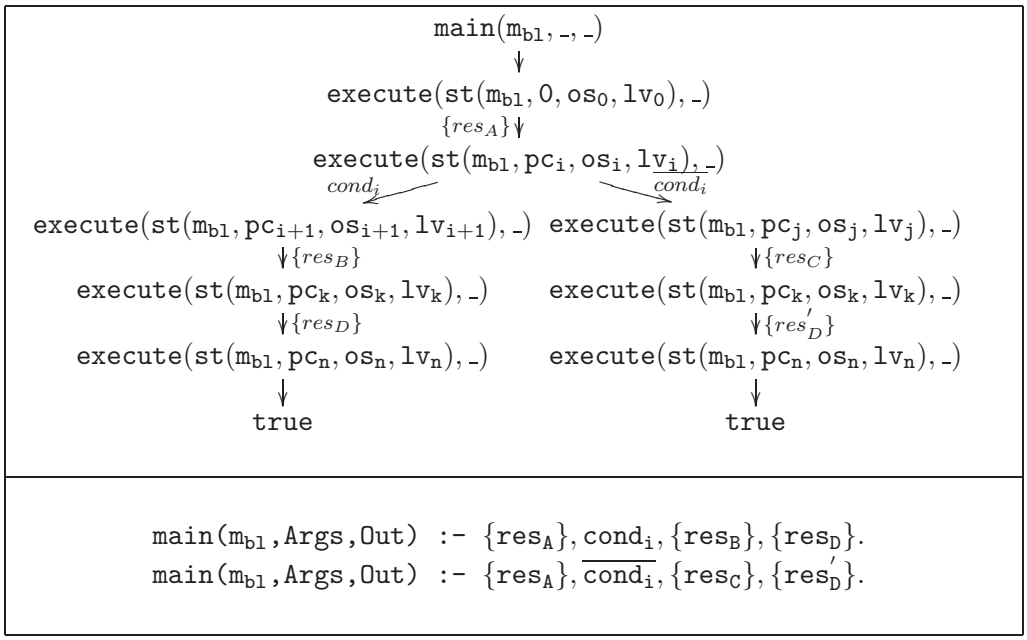


Figura 7.11: Unfolding SLD-tree and decompiled code of m_{bl} method

divergence point (D point) is a program point (bytecode index) from which more than one branch originates; likewise, a *convergence point* (C point) is a program point where two or more branches merge. In the CFG of m_{bl}, the only divergence (resp. convergence) point is pc_i (resp. pc_k).

By using the decompilation scheme presented so far, we obtain the SLD-tree shown in Fig. 7.11, in which all calls are completely unfolded as there is no termination risk. The decompiled code is shown under the tree. We use $\{\mathtt{res_X}\}$ to refer to the residual code emitted for BlockX and $\mathtt{cond_i}$ to refer to the condition associated to the branching instruction at $\mathtt{pc_i}$ ($\overline{\mathtt{cond_i}}$ denotes its negation). The quality of the decompiled code is not optimal due to:

D. Decompiled code $\{\mathtt{res_A}\}$ for BlockA is duplicated in both rules. During PE, this code is evaluated once but, due to the way resultants are defined (see Section 7.1), each rule contains the decompiled code associated to the whole branch of the tree. This code duplication brings in two problems: it increases considerably the size of decompiled programs and also makes their execution slower. For instance, when $\overline{\mathtt{cond_i}}$ holds, the execution goes unnecessarily through $\{\mathtt{res_A}\}$

in the first rule, fails to prove $\texttt{cond}_\texttt{i}$ and, then, attempts the second rule.

C. Decompiled code of BlockD is again emitted more than once. Each rule for the decompiled code contains a (possibly different) version, $\{\texttt{res}_\texttt{D}\}$ and $\{\texttt{res}'_\texttt{D}\}$, of the code of BlockD. Unlike above, at PE time, the code of BlockD is actually evaluated in the context of $\{\texttt{cond}_\texttt{i}, \{\texttt{res}_\texttt{B}\}\}$ and then re-evaluated in the context of $\{\overline{\texttt{cond}_\texttt{i}}, \{\texttt{res}_\texttt{C}\}\}$. Convergence points thus might degrade both efficiency (and endanger scalability) and quality of decompilation (due to larger residual code).

The amount of repeated residual code grows exponentially with the number of C and D points and the amount of re-evaluated code grows exponentially with the number of C points. Thus, we now aim at designing an *optimal*, *block-level* decompilation that helps overcome problems D and C above. Intuitively, a block-level decompilation must produce a residual rule for each block in the CFG. This can be achieved by building SLD-trees which correspond to each single block, rather than expanding them further. Note that this idea is against the typical spirit of PE which, in order to maximize the propagation of static information, tries to build SLD-trees as large as possible and only stops unfolding when there is termination risk.

This can be easily done in our setting by providing annotations that force the unfolding process to stop when an `execute/2` atom whose *PC* corresponds to a D point appears in the sequence. In the example, unfolding should stop at pc_i. Regarding C points, an additional requirement is to partially evaluate the code on blocks starting at these points at most once. The problem is similar to the polyvariant vs. monovariant treatment in the decompilation of methods in the previous section, by viewing entries to blocks as method calls. Not surprisingly, the solution can be achieved similarly in our setting by: (1) stopping the derivation at `execute/2` calls whose *PC* corresponds to C points and (2) passing the call to the global control, and ensuring that it is evaluated in a sufficiently generalized context which covers all incoming contexts. The former point is ensured by the use of the corresponding annotations and the latter by including in the initial set of atoms a generalized call of the form $\texttt{execute}(\texttt{st}(\texttt{m}_\texttt{bl}, \texttt{pc}_\texttt{k}, \_, \_), \_)$ for all C points, which forces such generalization.

```
main(m_bl,Args,Out) :- {res_A}, execute_1(...).
execute_1(...) :- cond_i, {res_B}, execute_2(...).
execute_1(...) :- ¬cond_i, {res_C}, execute_2(...).
execute_2(...) :- {res_D}.
```

Figura 7.12: Optimal decompiled code for m_{bl} method

An important point is that, unlike annotations used in offline PE [62] which are generated by only taking the interpreter into account, our annotations for the optimal decompilation are generated by taking into account the particular program to be decompiled. Importantly, both the annotations and the initial set of calls can be computed automatically by performing two passes on the bytecode (see, e.g., [2, 84]).

The result of performing an optimal decompilation on m_{bl} is shown in Figure 7.12. Now, the residual code associated to each block appears once in the code. This ensures that the optimal decompilation preserves the CFG shape as dedicated decompilers do. Thus, the quality of our decompiled code is as good as that obtained by state-of-the-art decompilers [2, 69] but with the advantages of interpretive decompilation.

Paper 6 studies in detail these issues and defines a block-level, optimal decompilation scheme overcoming the problems above. It is also formally proved that the proposed scheme satisfies the *block-optimality* criterion, which ensures that: (I) residual code for each bytecode instruction in the program is emitted once in the decompiled program, (II) each bytecode instruction is evaluated at most once during PE, and (III) there is at most one residual rule for each block in the bytecode program.

7.5.1. Conclusions of Optimal Decompilation

After taking into account the central observation from Section 7.4 that the interpreter should be written in big-step semantics, each condition of the block-optimality criterion above is simpler or more complicated to achieve depending on the local control strategy we use. For example, if we start from a modular decompiler as discussed in Section 7.4 above, condition (III) will in general be satisfied, but not condition (I) nor (II) since the

local control rule tends to over-specialize calls which results in re-evaluating expressions and emitting code multiple times.

Conversely, if we use an offline partial evaluator, the natural local control is to residualize all calls to `execute` and, then, filter out all information other than the method signature and program counter when transferring the atom to the global control. This control strategy trivially guarantees conditions (I) and (II) of the block-optimality criterion since it guarantees that each bytecode instruction is decompiled independently of the others. However, it tends to under-specialize and namely it does not satisfy the condition (III): as soon as there is a block with more than one bytecode instruction, which is almost always the case, the specialized program will contain a separate rule for each and every bytecode instruction in the block. As a result, the residual program thus obtained is high-level in the sense that it is written in Prolog. However, its control strategy is heavily influenced by the fact that we decompile bytecode (instead of converting, e.g. from Java source) and the decompiled program is not at all similar to the Prolog program which a Prolog programmer would write for performing the same task. Since an important objective of decompilation is to enable program understanding and analysis, we argue that programs which satisfy this optimality criterion, in particular meeting condition (III), like the ones we generate, are easier to reason about.

Another important observation is that the costly mechanisms, namely the TbHEm and the advanced polyvariance control from Paper 2, used for controlling the PE that were used earlier to achieve the results in Sections 7.3.3 and 7.4, are not needed anymore using the optimal decompilation scheme. Instead, the following trivial control operators can be used: unfold unfolds all calls except those matching an annotation, and abstract adds to the set S_{i+1} every call in L^{pe} which is not an instance of any call in S_i (see the generic algorithm in Section 7.1). It can be easily proved that termination is ensured both at the local and at the global control level thanks to the annotations and the initial set of atoms provided to the PE.

7.6. Implementation and Experimental Results

Paper 6 discusses several implementation details and performs a thorough experimental evaluation of the different decompilation schemes proposed in this chapter. We have reported on two different implementations of a decompiler for full (sequential) Java Bytecode into Prolog. For the first one we have extended an already existing powerful online PE, the one integrated in the CiaoPP system. This partial evaluator implements several unfolding rules and abstraction operators. This has allowed us to compare the different decompilation schemes, in particular, to compare against the non-optimal ones. However, the overhead introduced by using such generic and powerful tool prevents us from competing with ad-hoc decompilers as regards efficiency (decompilation times). For this reason, we have carried out a second implementation for which we have written a stand-alone PE which only contains the local and global strategies required by an optimal decompilation. This partial evaluator is integrated into a decompilation tool called jbc2prolog which also includes a Java Bytecode interpreter. This makes it possible to both obtain optimal decompilations and be competitive in terms of efficiency with ad-hoc decompilers. Paper 6 performs a thorough comparison against the decompiler in the COSTA [5] system and against the JDec [14] decompiler.

Both implementations consider full sequential Java Bytecode. The extensions needed to handle the features not considered in this introduction are further discussed in Paper 6. These include exceptions, heap operations, virtual invocations, decompilation at the level of classes, etc. It is important to note that all of them have been easily accommodated in our decompilation scheme, most of the times, simply by providing the corresponding support within the bytecode interpreter.

For the experimental evaluation in Paper 6, we have used the standardized set of benchmarks in the JOlden suite [54]. In particular, we are interested in: a) empirically demonstrating the scalability of the approach, and b) assessing the efficiency of the implemented tool by comparing it against other decompilers. We conclude the following:

- **Scalability:** While in the non-optimal decompilation both the de-

compilation times and the decompiled program sizes greatly increase with the size of the benchmarks, this does not happen in the optimal scheme. In the optimal decompilation, these figures are totally stable. We show that both the decompilation times and the decompiled program sizes are *linear* with the size of the input bytecode program, thus demonstrating the scalability of our optimal decompilation.

- **Efficiency:** To assess the efficiency of our approach we have compared the decompilation times we get using our tool jbc2prolog w.r.t. those obtained using the decompiler in the COSTA system and those obtained using the well-known Java decompiler JDec [14]. It can be concluded that our results our competitive with those of an ad-hoc decompiler. In particular, we see that they are similar to those obtained in COSTA. Furthermore, in most examples, jbc2prolog is slightly more efficient. On the other hand we can see that jbc2prolog is about ten times faster than JDec. Our conclusion in this regard is that it is very difficult to compare with decompilers written in other programming languages, since the performance of the implementation language heavily influences the decompilation time.

7.7. Related Work on Interpretive Decompilation

Previous work in *interpretative* (de)compilation has mainly focused on proving that the approach is feasible for small interpreters and medium-sized programs. The focus has been on demonstrating its *effectiveness*, i.e., that the so-called interpretation layer can be removed from the compiled programs. To achieve effectiveness, offline [62], online [47, 76] and hybrid [63] PE techniques have been assessed. This thesis has firstly focused on demonstrating that interpretive decompilation is feasible (as shown in previous work) and has studied further issues which had not been explored yet. Let us review now related work both in the field of decompilation of low-level code. Related work on on the PE of interpreters has been already compared in the introduction of this chapter and in several places throughout the paper.

The work by Breuer and Bowen [19] is only tangentially related to

ours. They propose a general method for compiling decompilers from the specifications of (non-optimizing) compilers. The main idea is that a data type specification for a programming-language grammar can be remolded into a functional program that enumerates all of the abstract syntax trees of the grammar. It is showed that by relying on this technique a decompiler can be generated from a simple Occam-like compiler specification. The only similarity with our work is that decompiled programs are somehow obtained from specifications (in our case of the interpreter and in their case of the compiler). However, the underlying methods are technically different and also they do not provide a practical solution for ensuring applicable conditions for their technique.

As regards (direct) decompilation of low-level back to source code, it has been the subject of a good amount of research. Decompilation can be attempted at different levels, with different levels of success. The most complicated case is when decompiling binary executables. There are a good number of associated complications, such as recovering the control flow. One intrinsic problem in this approach is that it is not possible in general to distinguish code from data statically. See e.g. [25, 80] and their references for a discussion on the problems and techniques for binary decompilation. The next level is decompilation of assembly, see e.g. [26]. This shares many of the complications associated to the decompilation of binaries, since current hardware architectures are rather complex, but at least it is possible to separate code from data. The following level is decompilation of code to be run on a virtual machine. This is in general easier to perform since virtual machines are usually simpler than current hardware architectures and because often the code for this virtual machines (bytecode) must satisfy certain behavior restrictions (must be *verifiable* [59]) and types of variables are available. As a result, in the particular case of decompilation of Java Bytecode back to Java source, a number of successful commercial and free software decompilers exist which are able to handle a large class of bytecode programs, especially those generated by common Java compilers, i.e., `javac`. Nevertheless, things become more complicated when the Java Bytecode has been generated by an obfuscator, and especially when an optimizing compiler, or a compiler from other programming languages such as Haskell, Eiffel, ML, Ada, and Fortran is used. See e.g. [71] and its references for a good account on the existing Java Bytecode decompilers

and the difficulties associated to its decompilation.

As already mentioned, there exist several analyzers for Java Bytecode which use a higher-level intermediate representation and which can be seen as ad-hoc decompilers. In particular, both the COSTA [5] and `CiaoPP` [48] systems have a front-end which converts bytecode into an intermediate representation which is then the input to the subsequent analysis. Though in both cases the intermediate representation is similar, in the case of COSTA it is formalized as a rule-based representation [2], whereas in `CiaoPP` it is formalized as Horn clauses, i.e., a logic program [69]. The reason for doing that in `CiaoPP` is that, at least in principle, that allows using the analysis which are already available in `CiaoPP`. However, there is a crucial difference between the logic programs generated in [69] and those generated by our decompiler. Whereas the programs generated by [69] are only meant to be the subject of static analysis and are not executable, the programs we generate can both be subject to analysis or be executed. The reason why the programs in [69] nor those in [2] are executable is because they basically capture the control-flow of the bytecode program, but the basic bytecode instructions themselves remain as *builtins*, i.e., predefined predicates, to the analysis. Analysis results are correct as long as the behavior of such bytecode instructions is safely approximated by the analysis. Producing fully executable logic programs as the result of decompilation is not trivial since many of the bytecode instructions operate on the heap in a way or another. Thus, in order to make an executable decompiled program we need to introduce the JVM heap explicitly in the logic program. All this is done automatically in our approach.

Chapter 8

Applications of Interpretive Decompilation

As already mentioned in the previous chapter, an important advantage is that the decompiled programs obtained by interpretive decompilation are fully *executable*, which in turn broadens their application field. In this chapter, we summarize two different experimentations we have performed which take advantage of such feature of our decompiled programs:

1. Analysis of bytecode programs by analyzing its decompilations to LP using LP analysis tools. This is further elaborated in Paper 1.

2. *Test data generation* of bytecode programs by CLP partial evaluation. This issue is studied in detail in Paper 7.

8.1. Analysis of Bytecode using LP Analysis Tools

Analyzing programs in the CLP paradigm offers a good number of advantages, an important one being the maturity and sophistication of the analysis tools available for it. In particular, the `CiaoPP` system, besides providing a very powerful partial evaluator which we have used to perform part of our experimentation in the previous chapter, also provides a generic analysis engine with a good number of abstract domains available. This

allows inferring a good number of properties of logic programs like *termination*, bounds on resource consumption, *types* and *modes*, *error-freeness*, etc.

One of the objectives of this thesis has been to investigate whether it is feasible to reuse existing analysis tools already available in the CLP paradigm, in particular `CiaoPP`, to analyze bytecode programs by analyzing their decompilations to LP. This allows devising a generic framework for the analysis and verification of bytecode programs in which the power of the analysis tools for CLP is automatically transferred to the analysis and verification of bytecode programs. The same idea had been applied to analyze rather restricted versions of high-level imperative languages [76] and also assembly code for PIC [46], an 8-bit microprocessor. However, to the best of our knowledge, this is the first time this approach has been successfully applied to a general purpose, realistic, imperative programming language.

These issues are further elaborated in Paper 1, where: 1) we propose such a framework for the analysis and verification of bytecode programs (in particular for Java Bytecode), and 2) we perform a series of experiments using the `CiaoPP` system demonstrating the feasibility of the proposed approach. In summary, Paper 1 shows how, by reasoning on our decompiled programs, we can automatically prove, by relying on the analyses available in the `CiaoPP` system, some non-trivial properties of bytecode programs such as termination, run-time *error-freeness* and infer bounds on its resource consumption. For instance, in order to prove run-time *error-freeness*, we propose an enhanced bytecode interpreter which computes, in addition to the return value of the method called, also the trace which captures the computation history. Such traces represent the semantic steps used, and therefore do not only represent instructions, as the context has also some importance. They have allowed us to distinguish, for example, for a same instruction, the step that throws an exception from the normal behavior. E.g., `invoke_step_ok` and `invoke_step_NullPointerException` represent, respectively, a normal method call and a method call on a null reference that throws an exception. Such additional flexibility of interpretive decompilation has allowed to prove run-time *error-freeness* in a straightforward way by simply specifying the property of being *error-free* as verifying that the corresponding trace in the decompiled program does not contain an exceptional step, or that it does not end raising an exception, depending on

the particular policy for the *error-freeness* property. Again, our approach demonstrates its flexibility here, as different policies can be easily defined simply by specifying the corresponding property in `CiaoPP`.

8.2. Test Data Generation by CLP PE

A unique feature of our decompiled programs is that they represent the whole program state, in contrast to [69, 2, 84]. In particular, they contain a representation of the heap explicitly in addition to the operand stack. Up to now, the main motivation for decompiling bytecode to LP had been to be able to perform static analysis on the decompiled programs in order to infer properties about the original bytecode. If the decompilation approach produces LP programs which are executable, then such decompiled programs can be used not only for static analysis, but also for dynamic analysis and execution. Note that this is not always the case, since there are approaches (like [4, 69]) which are aimed at producing static analysis targets only and their decompiled programs cannot be executed. A novel interesting application of interpretive decompilation which we propose in this thesis is the automatic generation of *test data*.

Test data generation (TDG) aims at automatically generating test-cases for interesting test *coverage criteria*. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed; etc. There are a wide variety of approaches to TDG (see [91] for a survey). Our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [38, 45] which execute the program to be tested for concrete input values.

The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program (see e.g. [28]), where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching ins-

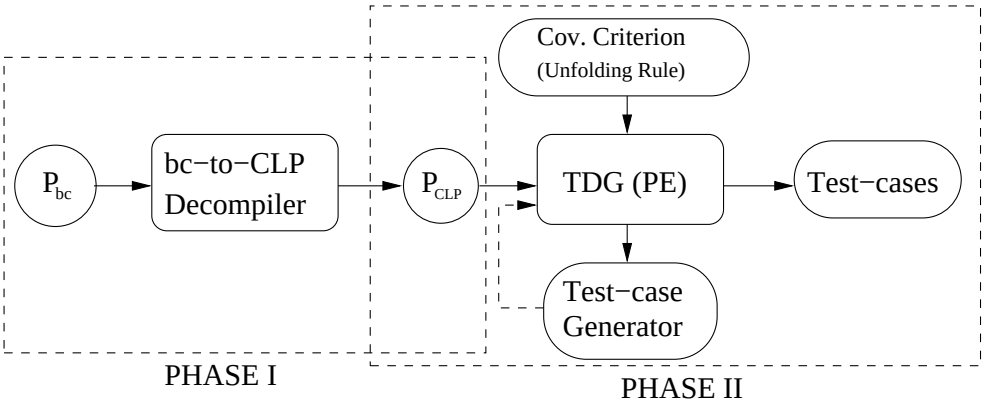


Figura 8.1: Overview of our approach for TDG of bytecode by CLP PE

tructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. For the particular case of Java Bytecode, a symbolic JVM (SJVM) which integrates several constraint solvers has been designed in [72]. A SJVM requires non-trivial extensions w.r.t. a JVM: (1) it needs to execute the bytecode symbolically as explained above, (2) it must be able to backtrack, as without knowledge about the input data, the execution engine might need to execute more than one path. The backtracking mechanism used in [72] is essentially the same as in logic programming.

In this thesis we propose a novel approach to TDG of bytecode which is based on PE techniques developed for CLP and which, in contrast to previous work, does not require devising a dedicated symbolic virtual machine. Figure 8.1 depicts a diagram with an overview of the framework. As can be seen, it comprises two independent, CLP PE phases, which basically consist in the following:

1. *The decompilation of bytecode into a CLP program.* We already discussed in Chapter 7 that the decompilation of bytecode to LP can be achieved automatically by means of partial evaluation of LP, or alternatively by means of an ad-hoc decompiler [69]. The modification to obtain CLP instead of LP programs is straightforward, e.g. by means of a trivial transformation of the arithmetic builtins into their CLP counterparts.

2. *The generation of test-cases.* This is a novel application of PE which

allows generating test-case generators from the CLP decompiled bytecode. In this case, we rely on a CLP partial evaluator which is able to solve the constraint system, in much the same way as a symbolic abstract machine would do. The two control operators of a CLP partial evaluator play an essential role: (1) The local control applied to the decompiled code will allow capturing interesting coverage criteria for TDG of the bytecode. (2) The global control will enable the generation of *test-case generators*. Intuitively, the test-case generators we produce are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch.

We argue that our CLP PE based approach to TDG of bytecode has several advantages w.r.t. existing approaches based on symbolic execution: (i) It is more *generic*, as the same techniques can be applied to other both low and high-level imperative languages. In particular, once the CLP decompilation is done, the language features are abstracted away and, the whole part related to the generation of test data is totally *language independent*. This avoids the difficulties of dealing with recursion, procedure calls, dynamic memory, etc. that symbolic abstract machines typically face. (ii) It is more *flexible*, as different coverage criteria can be easily incorporated to our framework just by adding the appropriate local control to the partial evaluator. (iii) It is more *powerful* as we can generate test-case generators. (iv) It is *simpler* to implement compared to the development of a dedicated symbolic virtual machine, as long as a CLP partial evaluator is available.

As noted in point (iv) above, an important advantage of CLP decompiled programs w.r.t. their bytecode counterparts is that symbolic execution does not require, at least in principle, to build a dedicated symbolic execution mechanism. Instead, we can simply run the decompiled program by using the standard CLP execution mechanism with all arguments being distinct free variables. E.g., for our working example of Figure 7.4, we could perform symbolic execution of the `gcd` method by running the query $\texttt{main}(\texttt{gcd}, [\texttt{X}, \texttt{Y}], \texttt{Z})$ on the decompiled program. Note that as we are not providing input values, each successful execution corresponds to a different computation path in the bytecode. Furthermore, along the execution, a constraint store on the program's variables is obtained which can be used for inferring the conditions that the input values (in our case `X` and `Y`) must

satisfy for the execution to follow the corresponding computation path.

However, an important problem with symbolic execution, regardless of whether it is performed using CLP or a dedicated execution engine, is that the execution tree to be traversed is in most cases infinite, since programs usually contain iterative constructs such as loops and recursion which induce an infinite number of execution paths when executed without input values. Therefore, it is essential to establish a *termination criterion* (in this context coverage criterion) which guarantees that the number of paths traversed remains finite, while at the same time an interesting set of test data is generated.

Another issue is that depending on the particular type of decompilation –and even on the options used within a particular method– we can obtain different correct decompilations which are valid for the purpose of execution. However, for the purpose of generating useful test-cases, additional requirements are needed: we must be able to define coverage criteria on the CLP decompilation which produce test-cases which cover the *equivalent* coverage criteria for the bytecode. Fortunately, our notion of *block-level* decompilation, introduced in 7.5, provides a sufficient condition for ensuring that equivalent coverage criteria can be defined. According to this definition, there is a one to one correspondence between blocks in the CFG of the bytecode program and rules in the decompiled one.

Most existing coverage criteria are defined on high-level, structured programming languages. A widely used control-flow based coverage criterion is loop-count(k), which dates back to 1977 [52], and limits the number of times we iterate on loops to a threshold k. However, bytecode has an unstructured control flow: CFGs can contain multiple different shapes, some of which do not correspond to any of the loops available in high-level, structured programming languages.

In this thesis, we introduce the *block-count*(k) coverage criterion which is not explicitly based on limiting the number of times we iterate on loops, but rather on counting how many times we visit each block in the CFG within each computation. Basically, a set of computation paths satisfies the *block-count*(k) *criterion* if the set includes all finished computation paths which can be built such that the number of times each block is visited within each computation does not exceed a given k.

Paper 7 discusses the technical details of such an approach to TDG of

bytecode. In particular:

- The *block-count*(k) coverage criterion is formally defined.
- We define an *evaluation strategy* which guarantees constructing an SLD tree so that we generate sufficiently many derivations so as to satisfy the block-count(k) criterion while, at the same time, guaranteeing termination.
- The TDG phase is formalized as a CLP PE of the CLP decompiled program where the unfolding rule plays the role of the coverage criterion. We hence provide an unfolding rule which implements the block-count(k) coverage criterion and outline how the abstraction operator must deal with constraints so that we get effective test-case generators.
- All such issues are illustrated through a working example which comprises a set of methods performing different arithmetic computations.

8.2.1. On the Generation of Test Data for Prolog by EP

As a tangential contribution of the thesis, we have applied the idea of using PE to automatically generate test data in the context of LP. We argue that our approach to TDG can in principle be directly applied to any imperative language. However, when one tries to apply it to a declarative language like Prolog, we have found as a main difficulty the generation of test-cases which cover the more complex control flow of Prolog. Essentially, the problem is that an intrinsic feature of PE is that it only computes non-failing derivations while in TDG for Prolog it is essential to generate test-cases associated to failing computations. Paper 8 performs a preliminary study in this direction. Basically, it proposes to transform the original Prolog program into an equivalent Prolog program with *explicit failure* by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. Another issue that we have discussed in the paper is that, while in the case of bytecode the underlying constraint domain only manipulates integers, in Prolog it should properly handle the symbolic data

manipulated by the program. Our preliminary experiments already suggest that the approach can be very useful to generate test-cases for Prolog.

8.2.2. Related work on Test Data Generation

As mentioned before, our approach is focused on static TDG, in which test-cases are obtained without running the program with particular input values. In contrast, *dynamic* approaches [38, 45] execute the program to be tested for concrete input values until achieving the particular coverage of the program. The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [28, 70, 72, 56, 44]. The symbolic execution approach has been combined with the use of *constraint solvers* [72, 44] in order to: handle the constraint systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For the particular case of Java Bytecode, a symbolic JVM machine (SJVM) which integrates several constraint solvers has been designed in [72].

TDG for declarative languages has received comparatively less attention than for imperative languages. The majority of existing tools for functional programs are based on black-box testing (see e.g. [27]). An exception is [39] where a glass-box testing approach is proposed to generate test-cases for Curry. In the case of CLP, test-cases are obtained for Prolog in [68, 13, 90]; and very recently for Mercury in [35]. Basically the test-cases are obtained by first computing constraints on the input arguments that correspond to execution paths of logic programs and then solving these constraints to obtain test inputs for such paths. For functional logic languages, specific coverage criteria are defined in [39] which capture the control flow of these languages as well as new language features are considered, namely laziness.

In general, declarative languages pose different problems to testing related to their own execution models –like laziness in functional languages and failing derivations in (C)LP– which need to be captured by appropriate coverage criteria. Having said this, we believe our ideas related to the use of PE techniques to generate test data generators and the use of unfolding rules to supervise the evaluation could be adapted to declarative programs as our preliminary experiments in Paper 8 show.

Chapter 9

Heap Space Analysis of Bytecode Programs

Predicting the memory required to run a program is crucial in many contexts like in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals as fast as possible. It is widely recognized also that memory usage estimation is important for an accurate prediction of running time, as cache misses and page faults contribute directly to the runtime.

Heap space analysis aims at inferring *bounds* on the heap space consumption of programs. Heap analysis is more typically formulated at the source level (see, e.g., [83, 49, 85, 53] in the context of functional programming and [51, 23] for high-level imperative programming languages). As mentioned in Chapter 6.4, there are however situations where one has only access to the compiled code and not to the source code. Automatic heap space analysis has interesting applications in this context. For instance, *resource bound certification* [32, 10, 50, 22] proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on the resource consumption. Also, heap bounds are useful on embedded systems, e.g., smart cards in which memory is limited and cannot easily be recovered.

A general framework for the cost analysis of sequential Java Bytecode has been proposed in [3] which led to the COSTA system [5]. Such analysis statically generates *cost relations* (*CRs*) which define the cost of a program as a function of its input data size. The *CRs* are expressed by means of

recursive equations generated by abstracting the recursive structure of the program and by inferring size relations between arguments. The analysis is parametric w.r.t. a *cost model* which defines the cost unit associated to each bytecode.

This thesis develops a novel application of the cost analysis framework of [3] to infer bounds on the heap space consumption of sequential Java Bytecode programs:

1. In a first step, we develop a cost model that defines the cost of memory allocation instructions (e.g., `new` and `newarray`) in terms of the number of heap (memory) units they consume. E.g., the cost of creating a new object is the number of heap units allocated to that object. The remaining bytecode instructions do not add any cost. With this cost model, we generate heap space *CRs* which are then used to infer upper bounds on the heap space usage of the different methods. These upper bounds provide information on the maximal heap space required for executing each method in the program.

2. Unfortunately, in the case of languages with automatic memory management (*garbage collection*), the above approach, though still correct, can produce too pessimistic estimations. Therefore, in a second step, we refine the analysis to consider the effect of garbage collection. We propose a *live heap space analysis*, which aims at approximating the maximum of the live heap usage along the execution of a program, thus providing a much tighter estimation in presence of garbage collection. This is done by relying on escape analysis [17] to identify those memory allocation instructions which create objects that will be garbage collected upon exit from the corresponding method. With this information available, we can infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that is allocated during the execution of the method *and* which remains upon exit. We then propose a novel form of *peak consumption CRs* which capture the peak memory consumption over all program states along its execution. An essential feature of our *CRs* is that they can be solved by using existing tools for solving standard *CRs*.

These issues are respectively introduced and summarized in Sections 9.1 and 9.2 and studied in detail in Papers 9 and 10.

A distinguishing feature of the analyses presented in this chapter w.r.t. previous approaches (e.g., [10, 49, 18, 24]) is that they are not restricted to linear bounds since the generated *CRs* can in principle capture any complexity class. Moreover, in many cases, using the upper bound solver of the COSTA system, the relations can be simplified to a *closed form* solution from which one can glean immediate information about the expected consumption of the code to be run.

It is important to note that the analysis could have been developed on the decompiled LP programs in a similar way. In fact, COSTA performs a decompilation of the bytecode into a rule-based representation before the actual analysis phase with the aim of making the analysis design simpler (see [3] for details). The IRs of COSTA are actually very similar to our LP decompiled programs with the main difference that in the COSTA IRs, all bytecode instructions remain residual and have to be taken as builtins, i.e., predefined procedures. In contrast, in our decompilations, bytecode instructions are interpreted at decompilation time and converted into basic Prolog instructions such as unifications and arithmetic operations. The reason why we have not used our interpretive decompilations for the analysis is that this way we have been able to integrate our analysis in COSTA and thus take advantage of all the machinery for the cost analysis which is included in it, like e.g., the *size analysis* for inferring the size relations among arguments, the upper bound solver, etc.

9.1. Total Heap Space Analysis of Bytecode

Let us consider the Java program depicted in Figure 9.1. It consists of a set of Java classes which define a linked-list data structure in an object-oriented style. The class `Cons` is used for data nodes (in this case integer numbers) and the class `Nil` plays the role of *null* to indicate the end of a list. Both `Cons` and `Nil` extend the abstract class `List`. Thus, a `List` object can be either a `Cons` or a `Nil` instance. Both subclasses implement a `copy` method which is used to clone the corresponding object. In the case of `Nil`, `copy` just returns a new instance of itself since it is the last element of the list. In the case of `Cons`, it returns a cloned instance where the data is cloned by calling the static method `m`, and the continuation is cloned by calling recursively the `copy` method on `next`.

```
abstract class List {
  abstract List copy();
}

class Nil extends List {
  List copy() {
    return new Nil();
  }
}

class Cons extends List {
  int elem;
  List next;

  List copy(){
    Cons aux = new Cons();
    aux.elem = m(this.elem);
    aux.next = this.next.copy();
    return aux;
  }

  static int m(int n) {
    Integer aux = new Integer(n);
    return aux.intValue();
  }
} // class Cons
```

Figura 9.1: Example for memory consumption

Our heap space analysis infers the following simplified *CRs* for the `copy` method of class `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 12 + C_{copy}(a-1), \quad a > 1$$

which can then be solved using the upper-bound solver of COSTA yielding the following upper-bound in closed-form:

$$C_{copy}(a) = 12*nat(a-1) + 12$$

It can be observed that the heap consumption is linear w.r.t. the input parameter `a`, which corresponds to the size of the *this* object of the method, i.e., the length of the list which is being cloned. This is because the abstraction being used by our analysis for object references is the *length of the longest reference chain*, which in this case corresponds to the length of the list. The numeric constant 12 is obtained by adding 8 and 4, being 8 the bytes taken by an instance of class `Cons`, and 4 the bytes taken by an `Integer` instance. Note that we are approximating the size of an object by the sum of the sizes of all of its fields. In particular, both an integer and a reference are assumed to consume 4 bytes.

The analysis has been integrated in the COSTA system. Paper 9 performs an experimental evaluation by means of a series of example applications written in an object-oriented style which make intensive use of the

heap and which present novel features like heap consumption that depends on the class fields, multiple inheritance, virtual invocation, etc. These examples allow us to illustrate the most salient features of our analysis: inference of constant heap usage, heap usage proportional to input size, support of standard data-structures like lists, trees, arrays, etc. To the best of our knowledge, this is the first analysis able to infer arbitrary heap usage bounds for Java Bytecode.

9.2. Live Heap Space Analysis for Languages with GC

As mentioned earlier, garbage collection (GC) makes the problem of predicting the memory required to run a program difficult. A first approximation is to infer the total memory allocation, i.e., the *accumulated* amount of memory allocated by a program ignoring GC, as done in the previous section. If such amount is available it is ensured that the program can be executed without exhausting the memory, even if no GC is performed during its execution. However, it is an overly pessimistic estimation of the actual memory requirement.

This thesis presents a general approach for inferring the *peak heap consumption* of a program's execution, i.e., the maximum of the live heap usage along its execution. Our live heap space analysis is developed for (an intermediate representation of) an object-oriented *bytecode* language with automatic memory management.

Analysis of live heap usage is different from total memory allocation because it involves reasoning on the memory consumed at *all program states* along an execution, while total allocation needs to observe the consumption at the *final* state only. As a consequence, the classical approach to static cost analysis proposed by Wegbreit in 1975 [86] has been applied only to infer total allocation. Intuitively, given a program, this approach produces a *CR* system which is a set of recursive equations that capture the cost *accumulated* along the program's execution. Symbolic closed-form solutions (i.e., without recursion) are found then from the *CR*. This approach leads to very accurate cost bounds as it is not limited to any complexity class (infers polynomial, logarithmic, exponential consumption, etc.) and, besides, it can

be used to infer different notions of resources (total memory allocation, number of executed instructions, number of calls to specific methods, etc.). Unfortunately, as argued in Paper 9, it is not suitable to infer peak heap consumption because it is not an accumulative resource of a program's execution as *CR* capture. Instead, it requires to reason on all possible states to obtain their maximum. By relying on different techniques which do not generate *CR*, live heap space analysis is currently restricted to polynomial bounds and non-recursive methods [18] or to linear bounds dealing with recursion [24].

Inspired by the basic techniques used in cost analysis, in this thesis, we present a general framework to infer accurate bounds on the peak heap consumption of programs which improves the state-of-the-art in that it is not restricted to any complexity class and deals with all bytecode language features including recursion. To pursue our analysis, we need to characterize the behavior of the underlying garbage collector. We assume a standard *scoped-memory* manager that reclaims memory when methods return. In this setting, our main contributions are:

1. *Escaped Memory Analysis.* We first develop an analysis to infer upper bounds on the *escaped memory* of method's execution, i.e., the memory that it is allocated during the execution of the method *and* which remains upon exit. The key idea is to infer first an upper bound for the total memory allocation of the method, as done in Section 9.1. Then, such bound can be manipulated, by relying on information computed by *escape analysis* [17], to extract from it an upper bound on its escaped memory.

2. *Live Heap Space Analysis.* By relying on the upper bounds on the escaped memory, as our main contribution, we propose a novel form of *peak consumption CR* which captures the peak memory consumption over all program states along the execution for the considered scoped-memory manager. An essential feature of our *CRs* is that they can be solved by using existing tools for solving standard *CRs*.

3. *Ideal Garbage Collection.* An interesting, novel feature of our approach is that we can refine the analysis to accommodate other kinds of scope-based managers which are closer to an *ideal* garbage collector which collects objects as soon as they become unreachable.

4. *Implementation.* We report on a prototype implementation which is integrated in COSTA and experimentally evaluate it on the JOlden benchmark suite. Preliminary results demonstrate that our system obtains reasonably accurate live heap space upper bounds in a fully automatic way.

Let us consider again the example of the previous section. Our live heap analysis now infers the following simplified *CRs* for the `copy` method of class `Cons`:

$$C_{copy}(a) = 12, \quad a = 1$$
$$C_{copy}(a) = 8 + \max(4, C_{copy}(a-1)), \quad a > 1$$

The intuition of the second *CR* is that the peak consumption of the method when $a > 1$ is the consumption of the method (a `Cons` object) plus the maximum between the peak consumption of method `m` and the escaped memory from `m` plus the peak consumption of *copy* with the decremented argument. The *CRs* can again be solved using the upper-bound solver of COSTA yielding the following upper-bound in closed-form:

$$C_{copy}(a) = 8*nat(a-1) + 24$$

An interesting observation is that the *Integer* object which is created inside the *m* method is not reachable from outside and thus can be garbage collected. The peak heap analyzer accounts for this and therefore has deleted the size of the *Integer* object from the recursive equation, thus obtaining 8 instead of 12 multiplying $\mathtt{nat}(\mathtt{A}-1)$. It can also be observed that COSTA is not being fully precise, as the actual peak consumption of this method is $8*\mathtt{nat}(\mathtt{A}-1)+8$ (i.e. the size of the cloned list). The reason for this is that the upper bound solver has to consider the additional cases introduced by the peak heap analysis in the *max* expressions to ensure soundness, hence making the second constant increase to 24.

9.3. Related Work on Heap Space Analysis

There has been much work on analyzing program cost or resource complexities, but the majority of it is on time analysis (see, e.g., [88]). Analysis of live heap space is different because it involves explicit analysis of all

program states. Most of the work on memory estimation has been studied for functional languages. The work in [49] statically infers, by typing derivations and linear programming, linear expressions that depend on functional parameters while we are able to compute non-linear bounds (exponential, logarithmic, polynomial). The technique is developed for functional programs with an explicit deallocation mechanism while our technique is meant for imperative bytecode programs which are better suited for an automatic memory manager. The techniques proposed in [83, 82] consist in, given a function, constructing a new function that symbolically mimics the memory consumption of the former. Although these functions resemble our cost equations, their computed function has to be executed over a concrete valuation of parameters to obtain a memory bound for that assignment. Unlike our closed-form upper bounds, the evaluation of that function might not terminate, even if the original program does. Other differences with the work by Unnikrishnan et al. are that their analysis is developed for a functional language by relying on *reference counts* for the functional data constructed, which basically count the number of pointers to data and that they focus on particular aspects of functional languages such as tail call optimizations.

It is worth mentioning also the work in [21], where a memory consumption analysis is presented. In contrast to ours, their aim is to verify that the program executes in bounded memory by simply checking that the program does not create new objects inside loops, but they do not infer bounds as our analysis does. Moreover, it is straightforward to check that new objects are not created inside loops from our cost relations. Another related work includes research in the MRG project [10, 16], which focuses on building a proof-carrying code [74] architecture for ensuring that bytecode programs are free from run-time violations of resource bounds. In contrast to ours, the analysis is developed for a functional language which then compiles to a (subset of) Java Bytecode and it is restricted to linear bounds. In [12] the Bytecode Specification Language is used to annotate Java Bytecode programs with memory consumption behavior and policies, and then verification tools can be used to verify those policies.

For Java-like languages, the work of [51] presents a type system for heap analysis without garbage collection, it is developed at the level of the source code and based on amortized analysis (hence it is technically quite different

to our work) and, unlike us, they do not present an inference method for heap consumption.

Related techniques have been also recently proposed to improve our first proposal of Paper 9. In particular, for an assembly language, [24] infers memory resource bounds (both stack usage and heap usage) for low-level programs (assembly). The approach is limited to linear bounds, they rely on explicit disposal commands rather than on automatic memory management. In their system, dispose commands can be automatically generated only if alias annotations are provided. For a Java-like language, the approach of [18] infers upper bounds of the peak consumption by relying on an automatic memory manager as we do. They do not deal with recursive methods and are restricted to polynomial bounds. Besides, our approach in Paper 10 is more flexible as regards its adaptation to other GC schemes. We believe that our system is the first one to infer upper bounds on the live heap consumption which are not restricted to simple complexity classes.

Chapter 10

Conclusions and Future Work

The main objective of this thesis has been to improve the state-of-the-art in the transformation and analysis of bytecode languages. Our first challenge was to provide a formal framework for the automatic decompilation of (object-oriented) bytecode programs into higher level intermediate representations using LP by means of interpretive decompilation. Compared to the development of a dedicated decompiler, interpretive decompilation has important advantages like *flexibility*, *maintainability*, *security* and *genericity*. Though very attractive, up to now it had not been widely applied in practice except for some proofs-of-concept showing the feasibility of the approach [62, 47, 75, 63]. Hence, there remained important open issues when it came to decompile realistic languages, namely, *scalability* and *effectiveness*. The thesis proposed novel solutions and finally answered them positively by presenting a modular, optimal decompilation scheme which: 1) produces decompiled programs whose quality is equivalent to that of dedicated decompilers, and 2) is demonstrated (theoretically and empirically) to scale up in practice. Our experimental results show that our decompiler is competitive, from the point of view of efficiency, with dedicated decompilers. We thus believe that the proposed techniques, together with their experimental evaluation, provide for the first time actual evidence that the interpretive theory proposed by Futamura in the 70s is indeed an appealing and feasible alternative to the development of dedicated decompilers from modern languages to intermediate representations.

For the sake of concreteness, our interpretive decompilation scheme has been formalized in the context of PE of logic programs, and implemented

for the Java Bytecode language. It is however important to note that the ideas we propose for enabling the practicality of the approach are also of interest for the interpretive (de)compilation of any pair of source and target languages.

On the other hand, the study of such a complex application of EP has led us to solve some non-trivial problems of EP in general, like the handling of infinite signatures. In this regard, we have come out with the *type-based homeomorphic embedding* relation, which has been demonstrated to improve the state-of-the-art of (online) specialization tools. We have shown that existing approaches which extend the untyped embedding relation to handle infinite signatures can be reconstructed as instances of our TbHEm relation. Though we have outlined procedures to infer the types in the context of LP, our type-based relation is not tied to any programming paradigm. Moreover, it can be used for a wide range of applications, namely in all areas of automatic program analysis, synthesis, verification, specialization and transformation; and will directly benefit from any progress on automatic type inference.

We have seen that the resulting intermediate representation, using LP, can greatly simplify the development of analysis, verification and model-checking tools for modern languages and, more interestingly, existing advanced tools developed for declarative languages (already proven correct and effective) can be directly applied on it. We have performed two experimentations in this direction. In the first one we have investigated whether it is feasible to analyze bytecode programs by analyzing its decompilations to LP using existing LP analysis tools. In this sense, we have been able to automatically prove in the `CiaoPP` system some non-trivial properties of Java Bytecode programs such as termination, run-time error freeness and infer bounds on its resource consumption for some simple programs.

For our second experimentation we have taken advantage of the fact that our decompiled programs are fully *executable* since, in contrast to other approaches [69, 2, 84], they represent the whole program state (i.e. they contain a representation of the heap in addition to the operand stack). We have therefore proposed a methodology for test data generation of bytecode by means of existing EP techniques developed for CLP. Our approach consists of two separate phases: (1) the compilation of the bytecode to a CLP program, and (2) the generation of test-cases from the CLP program.

It naturally raises the question whether this approach can be applied to other imperative languages in addition to bytecode. This is interesting as existing approaches for Java [72], and for C [44], struggle for dealing with features like recursion, method calls, dynamic memory, etc. during symbolic execution. We have shown that these features can be uniformly handled in our approach after the transformation to CLP. In particular, all kinds of loops in the bytecode become uniformly represented by recursive predicates in the CLP program. Also, we have seen that method calls are treated in the same way as calls to blocks.

We believe this experimentation is a, very promising, proof-of-concept that partial evaluation of CLP is a powerful technique for carrying out TDG in bytecode languages. To develop our ideas, we have considered a simple imperative bytecode language and left out object-oriented features which require a further study. Also, our language is restricted to integer numbers and the extension to deal with real numbers is subject of future work. We plan to carry out an experimental evaluation by transforming Java Bytecode programs from existing test suites to CLP programs and then trying to obtain useful test-cases. When considering realistic programs with object-oriented features and real numbers, we will surely face additional difficulties. One of the main practical issues is related to the scalability of our approach. An important threaten to scalability in TDG is the so-called infeasibility problem [91]. It happens in approaches that do not handle constraints along the construction of execution paths but rather perform two independent phases (1) path selection and (2) constraint solving our approach integrates both parts in a single phase, we do not expect scalability limitations in this regard. Also, a challenging problem is to obtain a decompilation which achieves a manageable representation of the heap. This will be necessary to obtain test-cases which involve data for objects stored in the heap. For the practical assessment, we also plan to extend our technique to include further coverage criteria. We want to consider other classes of coverage criteria which, for instance, generate test-cases which cover a certain statement in the program.

In principle, such an approach to TDG can be applied to any language, both to high-level and low-level. In this direction, this thesis has performed a preliminary experimentation in which we study whether the second phase can be useful for test-case generation of CLP programs, which are

not necessarily obtained from a decompilation of an imperative code. This introduces some difficulties like the handling of failing derivations and of symbolic data. In this thesis, we have sketched solutions to overcome such difficulties. In particular, we have proposed a program transformation, based on PE, to make failure explicit in the Prolog programs. To handle Prolog's negation in the transformed programs, we have outlined existing solutions that make it possible to turn the negative information into positive information. Though our preliminary experiments already suggest that the approach can be very useful to generate test-cases for Prolog, we plan to carry out a thorough practical assessment. This requires to cover additional Prolog features which have not been handled yet, and, also to compare the results with other TDG systems. We also want to study the integration of other kinds of coverage criteria like *data-flow* based criteria. Finally, we would like to explore the use of static analyses in the context of TDG. For instance, the information inferred by a *failure analysis* can be very useful to prune some of the branches that our transformed programs have to consider.

Another big challenge of this thesis has been to improve the state-of-the-art of heap space analysis of bytecode languages. In this regard, we have developed a novel application of the cost analysis framework of [3] which has been further extended to consider the effect of garbage collection. We have therefore presented a general approach to the automatic and accurate live heap space analysis for bytecode languages with garbage collection. First, we have proposed how to obtain accurate bounds on the memory *escaped* from a method's execution by combining the total allocation performed by the method together with information obtained by means of escape analysis. Then, we have introduced a novel form of *peak consumption cost relation* which uses the computed escaped memory bounds and precisely captures the actual heap consumption of programséxecution for garbage-collected languages. Such cost relations can be converted into closed-form upper bounds by relying on standard upper bound solvers, in particular the one in COSTA. For the sake of concreteness, our analysis has been developed for object-oriented bytecode, though the same techniques can be applied to other languages with garbage collection. We have first developed our analysis under a scoped-memory management which reclaims memory on method's return. The amount of memory required to run a method under

such model can be used as an over-approximation of the amount required to run it in the context of an ideal garbage collection which frees objects as soon as they become dead. We have also shown how to approximate such ideal behavior with our analysis.

Finally, it is important to note that this approach could be used to estimate other (non accumulative) resources which require to consider the maximal consumption of several execution paths. For example, it can be used to estimate the maximal height of the frames stack as follows. Given a rule $\mathrm{r} \equiv \mathrm{p}(\langle \bar{\mathrm{x}} \rangle, \langle \bar{\mathrm{y}} \rangle) ::= \mathrm{g}, \mathrm{b}_1, \ldots, \mathrm{b}_\mathrm{n}$, where $b_{i_1} \ldots b_{i_k}$ are the calls in r, with $1 \leq i_1 \leq \cdots \leq i_k \leq n$ and $\mathrm{b}_{\mathrm{i_j}} = \mathrm{q}_{\mathrm{i_j}}(\langle \bar{\mathrm{x}}_{\mathrm{i_j}} \rangle, \langle \bar{\mathrm{y}}_{\mathrm{i_j}} \rangle)$, its corresponding equation would be

$$p(\bar{x}) = \mathrm{máx}(1 + q_{i_1}(\bar{x}_{i_j}), \ldots, 1 + q_{i_k}(\bar{x}_{i_k})) \quad \varphi_r$$

which takes the maximal height from all possible call chains. Each "1" corresponds to a single frame created for the corresponding call. Note that in this setting, tail call optimization can be also supported, by using an analysis that detects calls in tail position, and then replace their corresponding 1's by 0's. This is also a subject for future work.

Bibliografía

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

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *16th European Symposium on Programming, ESOP'07*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, March 2007.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *16th European Symposium on Programming, ESOP'07*, Lecture Notes in Computer Science. Springer, March 2007. Available online http://www.clip.dia.fi.upm.es/papers/jvm-cost-esop.pdf.

[4] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *ESOP*, LNCS 4421, pages 157–172. Springer, 2007.

[5] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *6th International Symposioum on Formal Methods for Components and Objects (FMCO'08)*, number 5382 in Lecture Notes in Computer Science, pages 113–133. Springer, 2007.

[6] E. Albert, M. Hanus, and G. Vidal. A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. *Journal of Functional and Logic Programming*, 2002(1), 2002.

[7] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In

15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05), number 3901 in LNCS, pages 115–132. Springer-Verlag, April 2006.

[8] E. Albert, G. Puebla, and M. Hermenegildo. Abstraction-Carrying Code. In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

[9] Elvira Albert, Puri Arenas, Samir Genaim, and German Puebla. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 15-17, 2008, Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, July 2008.

[10] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04*, LNCS 3362, pages 1–27. Springer-Verlag, 2005.

[11] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Technical Report RT-0203, 1997. citeseer.ist.psu.edu/barras97coq.html.

[12] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. In *SEFM*, pages 86–95, 2005.

[13] F. Belli and O. Jack. Implementation-based analysis and testing of prolog programs. In *ISSTA*, pages 70–80, 1993.

[14] S. Belur and K. Bettadapura. Jdec: Java Decompiler. http://jdec.sourceforge.net/.

[15] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theoretical Computer Science*, 318(1-2), 2004.

[16] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic Certification of Heap Consumption. In *Proc. of LPAR'04*, LNCS 3452, pages 347–362. Springer, 2004.

[17] Bruno Blanchet. Escape Analysis for Object Oriented Languages. Application to Java(TM). In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, pages 20–34. ACM, November 1999.

[18] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM*. ACM Press, 2008.

[19] Peter T. Breuer and Jonathan P. Bowen. Decompilation: The enumeration of types and grammars. *ACM Trans. Program. Lang. Syst.*, 16(5):1613–1647, 1994.

[20] M. Bruynooghe, J. Gallagher, and W. Humbeeck. Inference of Well-typings for Logic Programs with Application to Termination Analysis. In *12th International Static Analysis Symposium (SAS'05)*, volume 3672 of *LNCS*, pages 35–51. Springer-Verlag, 2005.

[21] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. In *FM'05*, number 3582 in LNCS. Springer, 2005.

[22] A. Chander, D. Espinosa, N. Islam, P. Lee, and G.Ñecula. Enforcing Resource Bounds via Static Verification of Dynamic Checks. In *14th European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2005.

[23] W.-N. Chin, H. H. Nguyen, S. Qin, and M. C. Rinard. Memory Usage Verification for OO Programs. In *Proc. of SAS'05*, volume 3672 of *LNCS*, pages 70–86, 2005.

[24] W-N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM*. ACM Press, 2008.

[25] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[26] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *ICSM*, pages 228–237, 1998.

[27] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[28] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[29] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[30] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *The European Symposium on Programming (ESOP 2005)*, number 3444 in LNCS, pages 21–30. Springer-Verlag, 2005.

[31] Stephen-John Craig, John P. Gallagher, Michael Leuschel, and Kim S. Henriksen. Fully automatic binding-time analysis for prolog. In *LOPSTR*, pages 53–68, 2004.

[32] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00*, pages 184–198. ACM, 2000.

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

[34] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[35] F. Degrave, T. Schrijvers, and W. Vanhoof. Automatic generation of test inputs for mercury. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, LNCS. Springer-Verlag, 2009.

[36] R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[37] N. Dershowitz and J. P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. Elsevier, 1990.

[38] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.

[39] S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *PPDP*, pages 63–74, 2007.

[40] Y. Futamura. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[41] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[42] J.P. Gallagher and J.C. Peralta. Using regular approximations for generalisation during partial evaluation. In *Proc. of the SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 44–51. ACM Press, 2000.

[43] G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the ACM SIGPLAN 2002 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 75–88. ACM Press, 2002.

[44] A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.

[45] N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Automated Software Engineering*, pages 219–228, 2000.

[46] Kim S. Henriksen and John P. Gallagher. Analysis and specialisation of a pic processor. In *SMC (2)*, pages 1131–1135. IEEE, 2004.

[47] Kim S. Henriksen and John P. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE Computer Society, 2006.

[48] 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(1–2):115–140, October 2005.

[49] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[50] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.

[51] M. Hofmann and S. Jost. Type-Based Amortised Heap-Space Analysis. In *15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006.

[52] W.E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[53] J. Hughes and L. Pareto. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proc. of ICFP'99*, pages 70–81. ACM Press, 1999.

[54] JOlden Suite Collection. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.

[55] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.

[56] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[57] J.B. Kruskal. Well-quasi-ordering, the Tree Theorem, and Vazsonyi's Conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[58] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *POPL*, pages 144–154, 1993.

[59] Xavier Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[60] M. Leuschel. Homeomorphic Embedding for Online Termination of Symbolic Methods. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.

[61] M. Leuschel and M. Bruynooghe. Logic Program Specialisation through Partial Deduction: Control Issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[62] M. Leuschel, S. Craig, M. Bruynooghe, and W. Vanhoof. Specialising interpreters using offline partial deduction. In *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 340–375. Springer, 2004.

[63] M. Leuschel, S. Craig, and D. Elphick. Supervising offline partial evaluation of logic programs using online techniques. In *LOPSTR*, volume 4407 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2006.

[64] Michael Leuschel and Germán Vidal. Fast offline partial evaluation of large logic programs. In *LOPSTR*, pages 119–134, 2008.

[65] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[66] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[67] J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.

[68] G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In *In Proc. of the 3rd International Symposium on Software Reliability Engineering*, pages 104–113, 1992.

[69] M. Méndez-Lojo, J.Ñavas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

[70] C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.

[71] Jerome Miecznikowski and Laurie J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In R.Ñigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer, 2002.

[72] R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.

[73] G. Necula. Proof-Carrying Code. In *ACM Symposium on Principles of programming languages (POPL 1997)*, pages 106–119. ACM Press, 1997.

[74] G. Necula. Proof-Carrying Code. In *POPL'97*, pages 106–119. ACM Press, 1997.

[75] J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In G. Levi, editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *LNCS*, pages 246–261, 1998.

[76] J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS'98*, volume 1503 of *LNCS*, pages 246–261, 1998.

[77] D. Pichardie. Bicolano (Byte Code Language in cOq). http://www-sop.inria.fr/everest/personnel/David.Pichardie/bicolano/main.html.

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

[79] D. Sands. A Naïve Time Analysis and its Theory of Cost Equivalence. *Journal of Logic and Computation*, 5(4), 1995.

[80] Benjamin Schwarz, Saumya K. Debray, and Gregory R. Andrews. Disassembly of executable code revisited. In Arie van Deursen and Elizabeth Burd, editors, *WCRE*, pages 45–54. IEEE Computer Society, 2002.

[81] The Ciao Development Team. The Ciao Multiparadigm Language and Program Development Environment, November 2006. The ALP Newsletter 19(3). The Association for Logic Programming.

[82] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Automatic Accurate Live Memory Analysis for Garbage-Collected Languages. In *Proc. of LCTES/OM*, pages 102–111. ACM, 2001.

[83] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu. Optimized Live Heap Bound Analysis. In *Proc. of VMCAI'03*, volume 2575 of *LNCS*, pages 70–85, 2003.

[84] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proc. of Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 125–135, 1999.

[85] P. Vasconcelos and K. Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *IFL*, volume 3145 of *LNCS*. Springer, 2003.

[86] B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9), 1975.

[87] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[88] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008.

[89] Reinhard Wilhelm. Timing Analysis and Timing Predictability. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, Third International Symposium (FMCO)*, volume 3657 of *LNCS, Revised Lectures*, pages 317–323. Springer, 2004.

[90] L. Zhao, T. Gu, J. Qian, and G. Cai. A novel test case generation method for prolog programs based on call patterns semantics. In *APLAS*, pages 105–121, 2007.

[91] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

Apéndice A

Artículos de la Tesis (Papers of the Thesis)

Lista de artículos (List of papers):

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Ninth International Symposium on Practical Aspects of Declarative Languages*, number 4354 in LNCS, pages 124–139. Springer-Verlag, January 2007.

2. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Improving the Decompilation of Java Bytecode to Prolog by Partial Evaluation. In M. Huisman and F. Spoto, editors, *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, volume 190, Issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 85–101. Elsevier - North Holland, July 2007.

3. E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding and its Applications to Online Partial Evaluation. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, volume 4915 of *LNCS*, pages 23–42. Springer-Verlag, February 2008.

4. E. Albert, J. Gallagher, M. Gómez-Zamalloa, and G. Puebla. Type-based Homeomorphic Embedding for Online Termination. *Information Processing Letters*, 2009.

5. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Modular Decompilation of Low-Level Code by Partial Evaluation. In *8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 239–248. IEEE Computer Society, September 2008.

6. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Journal of Information and Software Technology*, 2009. To appear.

7. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by clp Partial Evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, number 5438 in LNCS, pages 4–23. Springer-Verlag, March 2009.

8. M. Gómez-Zamalloa, E. Albert, and G. Puebla. On the Generation of Test Data for Prolog by Partial Evaluation. In *Workshop on Logic-based methods in Programming Environments (WLPE'08)*, volume WLPE/2008/06, pages 26–43, 2008.

9. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Heap Space Analysis for Java Bytecode. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 105–116, New York, NY, USA, October 2007. ACM Press.

10. E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live Heap Space Analysis for Languages with Garbage Collection. In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.