Universidad Politécnica de Madrid

E.T.S.I. Informáticos

Master in Software and Systems

MASTER THESIS

Improved Static Analysis and Verification of Energy Consumption and other Resources via Abstract Interpretation

Author: Maximiliano Klemen
Supervisor: Manuel Carro Liñares
Co-Supervisor: Pedro López García

MADRID, JULY, 2015

To Yami

Acknowledgments

I would like to thank my advisor and thesis co-supervisor Pedro López García, for his support, guidance and encouragement throughout the development of this thesis, and for all his time and energy, without which reaching this goal would not have been possible. I would also like to express my gratitude to the Scientific Director of the IMDEA Software Institute, Manuel Hermenegildo, for all his advice, guidance and support.

I also want to thank my ENTRA Project's fellows, Umer Liqat and Remy Haemmerlè, and José Morales, for all their help, advice, criticism and, most importantly, understanding. I take this oportunity to thank all the members of the ENTRA Project, especially John Gallagher, for his uninterested help. Thank you also to Manuel Carro, the official thesis supervisor, for all his help on administrative issues.

Finally, thank you very much to my parents, for making all this possible, and to my beloved lifemate Yamila, for all her support and sacrifice, and for sharing this adventure with me.

Abstract

Resource analysis aims at inferring the cost of executing programs for any possible input, in terms of a given resource, such as the traditional execution steps, time or memory, and, more recently energy consumption or user defined resources (e.g., number of bits sent over a socket, number of database accesses, number of calls to particular procedures, etc.). This is performed *statically*, i.e., without actually running the programs.

Resource usage information is useful for a variety of optimization and verification applications, as well as for guiding software design. For example, programmers can use such information to choose different algorithmic solutions to a problem; program transformation systems can use cost information to choose between alternative transformations; parallelizing compilers can use cost estimates for *granularity control*, which tries to balance the overheads of task creation and manipulation against the benefits of parallelization.

In this thesis we have significatively improved an existing prototype implementation for resource usage analysis based on abstract interpretation, addressing a number of relevant challenges and overcoming many limitations it presented. The goal of that prototype was to show the viability of casting the resource analysis as an abstract domain, and how it could overcome important limitations of the state-of-the-art resource usage analysis tools. For this purpose, it was implemented as an abstract domain in the abstract interpretation framework of the CiaoPP system, PLAI. We have improved both the design and implementation of the prototype, for eventually allowing an evolution of the tool to the industrial application level.

The abstract operations of such tool heavily depend on the setting up and finding closed-form solutions of recurrence relations representing the resource usage behavior of program components and the whole program as well. While there exist many tools, such as Computer Algebra Systems (CAS) and libraries able to find closed-form solutions for some types of recurrences, none of them alone is able to handle all the types of recurrences arising during program analysis. In addition, there are some types of recurrences that cannot be solved by any existing tool. This clearly constitutes a bottleneck for this kind of resource usage analysis. Thus, one of the major challenges we have addressed in this thesis is the design and development of a novel modular framework for solving recurrence relations, able to combine and take advantage of the results of existing solvers. Additionally, we have developed and integrated into our novel solver a technique for finding upper-bound closed-form solutions of a special class of recurrence relations that arise during the analysis of programs with accumulating parameters.

Finally, we have integrated the improved resource analysis into the CiaoPP general framework for resource usage verification, and specialized the framework for verifying energy consumption specifications of embedded imperative programs in a real application, showing the usefulness and practicality of the resulting tool.

Resumen

El Análisis de recursos tiene como objetivo inferir el coste de la ejecución de programas para cualquier entrada posible, en términos de algún recurso determinado, como pasos de ejecución, tiempo o memoria, y, más recientemente, el consumo de energía o recursos definidos por el usuario (por ejemplo, número de bits enviados a través de un socket, el número de accesos a una base de datos, cantidad de llamadas a determinados procedimientos, etc.). Ello se realiza *estáticamente*, es decir, sin necesidad de ejecutar los programas.

La información sobre el uso de recursos resulta muy útil para una gran variedad de aplicaciones de optimización y verificación de programas, así como para asistir en el diseño de los mismos. Por ejemplo, los programadores pueden utilizar dicha información para elegir diferentes soluciones algorítmicas a un problema; los sistemas de transformación de programas pueden utilizar la información de coste para elegir entre transformaciones alternativas; los compiladores paralelizantes pueden utilizar las estimaciones de coste para realizar *control de granularidad*, el cual trata de equilibrar el coste debido a la creación y gestión de tareas, con los beneficios de la paralelización.

En esta tesis hemos mejorado de manera significativa la implementación de un prototipo existente para el análisis del uso de recursos basado en interpretación abstracta, abordando diversos desafíos relevantes y superando numerosas limitaciones que éste presentaba. El objetivo de dicho prototipo era mostrar la viabilidad de definir el análisis de recursos como un dominio abstracto, y cómo se podían superar las limitaciones de otras herramientas similares que constituyen el estado del arte. Para ello, se implementó como un dominio abstracto en el marco de interpretación abstracta presente en el sistema CiaoPP, PLAI. Hemos mejorado tanto el diseño como la implementación del mencionado prototipo para posibilitar su evolución hacia una herramienta utilizable en el ámbito industrial.

Las operaciones abstractas de dicha herramienta dependen en gran medida de la generación, y posterior búsqueda de soluciones en forma cerrada, de relaciones recurrentes, las cuales modelizan el comportamiento, respecto al consumo de recursos, de los componentes del programa y del programa completo. Si bien existen actualmente muchas herramientas capaces de encontrar soluciones en forma cerrada para ciertos tipos de recurrencias, tales como Sistemas de Computación Algebraicos (CAS) y librerías de programación, ninguna de dichas herramientas es capaz de tratar, por sí sola, todos los tipos de recurrencias que surgen durante el análisis de recursos. Existen incluso recurrencias que no las puede resolver ninguna herramienta actual. Esto constituye claramente un cuello de botella para este tipo de análisis del uso de recursos. Por

lo tanto, uno de los principales desafíos que hemos abordado en esta tesis es el diseño y desarrollo de un novedoso marco modular para la resolución de relaciones recurrentes, combinando y aprovechando los resultados de resolutores existentes. Además de ello, hemos desarrollado e integrado en nuestro nuevo resolutor una técnica para la obtención de cotas superiores en forma cerrada de una clase característica de relaciones recurrentes que surgen durante el análisis de programas lógicos con parámetros de acumulación.

Finalmente, hemos integrado el nuevo análisis de recursos con el marco general para verificación de recursos de CiaoPP, y hemos instanciado dicho marco para la verificación de especificaciones sobre el consumo de energía de programas imperativas embarcados, mostrando la viabilidad y utilidad de la herramienta resultante en una aplicación real.

INDEX

Chapter 1

Introduction

Static resource analysis aims at inferring the cost of executing programs for any possible input (without actually running them), in terms of a given resource, such as the traditional execution steps, time or memory, and, more recently energy consumption or user defined resources (e.g., number of bits sent over a socket, number of database accesses, number of calls to particular procedures, etc.).

A very interesting resource, which is getting increasing attention in the last few years, is energy. Energy consumption and the environmental impact of computing technologies have become a major worldwide concern. Energy consumption is now a major issue in high-performance computing, distributed applications, and data centers. The growth of cloud computing-related energy consumption and Internet traffic is not sustainable with the current energy efficiency levels. There is also an increasing demand for small complex computing systems which have to operate on batteries, such as implantable/portable medical devices, smartphones or smart watches.

Resource usage information is useful for a variety of purposes. For example, programmers can use it to choose different algorithmic solutions to a problem; program transformation systems can use cost information to choose between alternative transformations; parallelizing compilers can use cost estimates for *granularity control*, which tries to balance the overheads of task creation and manipulation against the benefits of parallelization.

Also, the specification of a system could impose, for example, bounds on the amount of energy consumed, the execution time or the memory allocated for the execution of a program for arbitrary input data. If a system does not meet these *non-functional* requirements, it will not be considered correct at all. This becomes crucial in the embedded systems area, where meeting real-time constraints is critical. In order to automatically check this kind of specifications, a resource analysis tool must infer statically an approximation of the resource usage of the program, and then this information must be compared against the specification. In this verification application, it is particularly

important that the analysis be performed statically, because it is necessary to obtain *sound* information regarding *all possible inputs* in order to prove that a given specification is met.

The resource consumption behavior of a program depends in practice mainly on the sizes or values of certain input arguments. For example, for a program that performs some particular operation on the elements of a list, its resource consumption will be a function on the length of the list (and possibly the size of the elements of the list, as we will see later). In this sense, one of the objectives of this thesis is to come up with an effective and practical tool that statically infers *functions* representing upper and lower bounds on the amount of resources that each of the components of the program will consume (and the whole program as well). Such functions will be parameterized by the sizes of the input data to the topmost piece of code being analyzed. The input to the tool will be the source code, as well as basic information about the resource consumed by elementary operations (given once and for all for a particular hardware platform and language, by using assertions).

1.1 State of the Art and Challenges

The approach to cost analysis based on setting up and solving recurrence relations was proposed in [Wegbreit, 1975] and has been developed significantly in subsequent work. For example, in [Rosendahl, 1989] an automatic upper-bound analysis was presented based on an abstract interpretation of a step-counting version of a functional program, in order to infer both execution time and execution steps. However, size measures could not automatically be inferred and the experimental section showed few details about the practicality of the analysis. The cost analysis in [Vasconcelos and Hammond, 2003] deals with recursive, polymorphic and higher-order functional programs. In the context of Logic Programming, a semi-automatic analysis was presented in [Debray et al., 1990, Debray and Lin, 1993] that inferred upper-bounds on the number of execution steps, given as functions on the input data sizes. It also proposed techniques to address the additional challenges posed by the Logic Programming paradigm, as for example, dealing with the generation of multiple solutions via backtracking. However, a shortcoming of the approach was its loss in precision in the presence of divide-and-conquer programs in which the sizes of the output arguments of the "divide" predicates are dependent. This approach was later fully automated (by integrating it into the CiaoPP system and automatically providing *modes and size measures*) and extended to inferring both upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [Debray et al., 1997, Hermenegildo et al., 2005b]. In addition, [Debray et al., 1997] introduced the setting up of non-deterministic recurrence relations for the class of divide-and-conquer programs mentioned above, and proposed a technique for computing approximated closed form bound functions for some of them. Such a technique was based on bound-

ing the number of terminal and non-terminal nodes in the set of computation trees corresponding to the evaluation of the non-deterministic recurrence relations, and bounding the cost of such nodes. Non-deterministic recurrence relations were also used and further developed in [Albert et al., 2008, Albert et al., 2011a] (named Cost Relations).

The approach in [Debray et al., 1990, Debray and Lin, 1993, Debray et al., 1997] was generalized in [Navas et al., 2007a] to infer *user-defined resources* (by using an extension of the Ciao assertion language [Hermenegildo et al., 2012a]), and was further improved in [Serrano et al., 2014a], which is our starting point.

The novel general resource analysis for logic programs presented in [Serrano et al., 2014a] was entirely based on abstract interpretation (see Chapter 2 for an introduction to the abstract interpretation technique). The abstract operations of such approach heavily depend on setting up and finding closed-form solutions of recurrence relations representing the resource usage behavior of program components and the whole program as well. Nowadays, there exist many tools, such as Computer Algebra Systems (CAS) and libraries able to find closed-form solutions for some types of recurrences (e.g., Mathematica® or Maxima). However, none of those tools alone is able to solve or bound all the types of recurrences that can arise during analysis. In addition, there are some types of recurrences that cannot be solved by any existing tool. As a result, recurrence relation solving has become a bottleneck in [Serrano et al., 2014a] and other resource analysis systems. Thus, a major challenge that we have addressed in this thesis is the design and development of a component for solving recurrence relations, overcoming important limitations of existing resource analysis, and offering good quality features such as robustness and extensibility. This is crucial for obtaining a practical resource analysis tool that can be used in real industrial applications.

Moreover, the goal of [Serrano et al., 2014a] was to show the viability of casting the resource analysis as an abstract interpretation, the benefits of using such approach, and the important limitations of state-of-the-art resource usage analysis tools that can be overcome. To achieve this, an initial design of the abstract domain for resource analysis, as well as a prototype implementation, was developed and integrated into the CiaoPP abstract interpretation framework, PLAI [Hermenegildo et al., 2005a, Hermenegildo et al., 2012b]. As the prototype implementation was a proof-of-concept, its design was not completely appropriate and its functionality not completely implemented. For example, the integration with other abstract domains and with the resource usage verification framework present in CiaoPP was not totally implemented and well-defined. Thus, a challenge that we address in this thesis is to come up with a better design and integration of such resource analysis into the CiaoPP system, in order to enhance its effectiveness, practicality, maintainability, extensibility and allow an easier combination with other supporting analysis present in the CiaoPP system. This will also eventually allow an evolution of the tool the industrial application level.

Finally, as mentioned before, an important application of static analysis is resource usage verification. Although CiaoPP provides a general framework for resource usage verification [Lopez-Garcia et al., 2010b, Lopez-Garcia et al., 2012], it is a challenge, which we also have addressed in this thesis, to use it for *verifying* energy consumption specifications of embedded imperative programs in a real world scenario, where energy models of the particular hardware platform have to be used and translations from the source language to the (block-based) internal representation language (Horn Clauses), including assertions representing specifications, have to be developed. In addition, it is also necessary to reflect the verification results up to the source code.

1.2 Contributions

The main contributions of this thesis, in order to address the challenges mentioned before, are the following:

- Improvements to the design and implementation of the resource usage analysis based on abstract interpretation present in the CiaoPP preprocessor. The goal of these improvements is a better integration of the analysis with the rest of the system, in order to make it more effective and practical, and allow its evolution to industrial level applicability. This contribution is explained in Chapter 3.

- Design and implementation of a modular component in charge of solving recurrence relations. It is a modular solver framework offering a well-defined interface to the analyzer, and providing all the algebraic-related services, being the most important one, the finding of closed-form functions for recurrences. In turn, our solver communicates with a set of *external* solvers using a common interface that we have also defined. Our proposed architecture has two main advantages. Firstly, it establishes a good and clear separation between the analysis and the mathematical machinery. Secondly, results from different external solvers can be combined in order to obtain better solutions. Finally, it makes easier to add new external solvers in order to handle more classes of recurrences and solving other mathematical problems that can arise during analysis. The use of our new solver component has resulted in a significant improvement of the whole resource analysis. This contribution is detailed in Chapter 4.

- Design and implementation of a specialized solver for a common class of recurrence relations that arise in the analysis of programs with accumulating parameters. This solver has been integrated into the modular framework mentioned above. The details of this contribution can be found in Chapter 4.

- Integration of the new abstract-interpretation based resource analysis into the CiaoPP general framework for resource verification, so that the resource usage information inferred by the new analysis can be compared with program specifications (Chapter 5).

- Finally, we have leveraged the CiaoPP resource verification framework and specialized it for *verifying* energy consumption specifications of embedded imperative programs in a concrete industry case study: the energy verification of embedded programs written in the XC language [Watt, 2009] and running on the XMOS XS1-L architecture (XC is a high-level C-based programming language that includes extensions for communication, input/output operations, real-time behavior, and concurrency). The specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The verification process produces preconditions (related to input data sizes) under which a given specification is met or not. The use of the improved resource analysis has also resulted in an improvement of the verification application. This contribution has produced a publication [Lopez-Garcia et al., 2015] and is explained in Chapter 5.

1.3 Structure of the Thesis

The rest of this thesis is organized as follows. In Chapter 2 we recall some concepts and results about abstract interpretation, and give a detailed description of the PLAI abstract interpretation framework, which is the basis for our resource analysis implementation. In Chapter 3 we present the resource analysis for logic programs (Horn Clauses) implemented as an abstract domain in the PLAI framework, showing the improvements we have incorporated to its design and implementation. In Chapter 4 we describe the architecture of our modular solver, which improves the solving of recurrences, and hence the applicability, extensibility and practicability of the whole resource analysis. In Chapter 5 we show the application of our improved resource analysis tool to the *verification* of energy consumption specifications of embedded imperative programs in a real industrial scenario. Finally, in Chapter 6 we give some final conclusions and mention directions for future work.

Chapter 2

Abstract Interpretation and the PLAI Framework

In this chapter we give a detailed description of the PLAI abstract interpretation framework, upon which our resource analysis is implemented as an abstract domain. In Section 2.1, we enumerate the main concepts of the theory of abstract interpretation developed by [Cousot and Cousot, 1977]. In Section 2.2, we explain in depth the top-down algorithm of abstract interpretation for logic programs that is implemented in PLAI. This algorithm is based on the framework of abstract interpretation for logic programs proposed in [Bruynooghe, 1987], with the optimizations proposed in [Muthukumar and Hermenegildo, 1990] for an efficient fixpoint computation.

2.1 Abstract Interpretation

Abstract interpretation [Cousot and Cousot, 1977] is a theory that allows approximating program semantics by abstracting its concrete semantics in a sound and systematic way. This technique has allowed the development of sophisticated program analyses which are at the same time provably correct and practical. It provides a framework for deriving an approximation of the properties of programs, and a static analyzer based on that approximation that is sound by construction.

Abstract interpretation often represents semantics as fixpoints of operators. It provides a systematic method to derive computable abstract approximate semantics which we can summarize in the following general steps [Miné, 2012]:

- A first step is to choose a level of abstraction. The set of objects of the concrete domain (e.g., the set of integers, in the case of a numerical domain), is replaced

by a partially ordered set of elements that abstract the concrete ones, although loosing information (e.g., the set of integer intervals, ordered with set inclusion).

- Operators on the concrete domain are abstracted as operators on the abstract domain (e.g., $+ : \mathbb{Z} \to \mathbb{Z}$ is abstracted as $+\sharp : \mathbb{I}(\mathbb{Z}) \to \mathbb{I}(\mathbb{Z})$, where $\mathbb{I}(\mathbb{Z})$ represents the set of integer intervals). Sometimes, it is necessary to over-approximate this abstract operators for the sake of efficiency.
- Fixpoints of concrete operators are replaced with fixpoints of abstract operators, using extrapolation operators to ensure termination, in those cases where the partial order in the abstract set has infinite chains.

The semantic approximations produced by the abstract-interpretation based analyses have been traditionally applied to high- and low-level *optimizations* during program compilation, including *program transformation*. More recently, novel and promising applications of semantic approximations have been proposed in the more general context of program development, such as *verification* and *debugging*.

In the following, we start by presenting some basic concepts commonly used in abstract interpretation. Then, we briefly present the main definitions and results of abstract interpretation. We refer the reader to [Miné, 2013] for a more extensive introduction on this topic, and to [Cousot and Cousot, 1992, Cousot and Cousot, 1977] for an in-depth explanation.

2.1.1 Lattices

Definition 1 (*Pre-order* and *partial order*)**.** *A* pre-order *on a set* S *is a binary relation* $\sqsubseteq$ *that is:*

1. Reflexive: $\forall x \in X : (x \sqsubseteq x)$*; and*
2. Transitive: $\forall x, y, z \in X : (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow x \sqsubseteq z$.

A partial order *is a pre-order that is* antisymmetric*, i.e.,*

$$\forall x, y \in S : (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y.$$

Definition 2 (Least upper bound and greatest lower bound)**.** *Let* $\sqsubseteq$ *be a partial order on a set* S*. We say that* u *is an* upper bound *of a subset* X *of* S *if and only if* $\forall x \in X : x \sqsubseteq u$*. We say that* u *is the* least upper bound *of* X *if and only if* u *is an* upper bound *of* X*, and for any* u' *that is an* upper bound *of* X*, then* $u \sqsubseteq u'$*. If it exists, the least upper bound of X is represented as* $\sqcup X$*. The definition of* lower bound *and* greatest lower bound *of X (represented as* $\sqcap X$*) are obtained by using the dual version of* $\sqsubseteq$ *(i.e.,* $\sqsupseteq$*) in the definitions of upper bounds and least upper bounds.*

Definition 3 (Poset and complete lattice)**.** *A partially ordered set (poset for short)* $(A, \sqsubseteq)$ *is a set equipped with a partial order relation. A tuple* $(L, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ *is a complete lattice if and only if* $(L, \sqsubseteq)$ *is a poset such that any subset* X *of* L *has a least upper bound* $\sqcup X$ *and a greatest lower bound* $\sqcap X$*, where* $\bot = \sqcup\emptyset = \sqcap L$ *and* $\top = \sqcap\emptyset = \sqcup L$ *are the smallest and greatest element of* L *respectively.*

Definition 4 (Linear order, increasing chains, and complete partial order)**.** *A linear order* $\sqsubseteq$ *on a set* A *is a partial order on* A *such that* $\forall x, y \in A : (x \sqsubseteq y \vee y \sqsubseteq x)$*. An increasing chain is a set* $X \subseteq A$ *such that* $\sqsubseteq$ *is a linear order on* X*. A complete partial order is a poset such that every increasing chain has a least upper bound.*

2.1.2 Abstraction and Concretization

In the context of abstract interpretation, posets are used as sets of elements carrying information about programs. The order relation quantifies the relative amount of information of those elements, i.e., given the poset $(A, \sqsubseteq)$, $a \sqsubseteq b$ means that b carries *less* information than a, with $a, b \in A$.

Having the previous observation in mind, let us assume in the following that the concrete program properties are described by elements of a given poset $(D, \sqsubseteq)$, and that the abstract program properties are represented by elements of a poset $(D^\sharp, \sqsubseteq^\sharp)$.

Definition 5 (Abstraction and concretization functions)**.** *A concretization function is a monotonic function* $\gamma \in D^\sharp \to D$ *that maps abstract descriptions of program properties into their corresponding concrete properties. It is monotonic because it has to respect the information order, i.e.,* $\forall a, b \in D^\sharp : a \sqsubseteq^\sharp b \implies \gamma(a) \sqsubseteq \gamma(b)$*.*

The approximation of a property is formalized by an abstraction function $\alpha \in D \to D^\sharp$ *that gives the best abstraction for a concrete property.*

If $a^\sharp = \alpha(a)$ and $a^\sharp \sqsubseteq b^\sharp$, then $b^\sharp$ is also a correct abstract approximation of a, although less precise. This soundness can be expressed by $\alpha(a) \sqsubseteq b^\sharp$. If $p = \gamma(p^\sharp)$ and $q \sqsubseteq p$ then $p^\sharp$ is also a correct approximation of the concrete property q although this concrete property q provides more accurate information about the program than p. This soundness condition can be expressed by $q \sqsubseteq \gamma(p^\sharp)$. When these two soundness conditions are equivalent, α and γ form a *Galois connection.*

Definition 6 (Galois connection)**.** *Given the posets* $(P, \sqsubseteq)$ *and* $(P^\sharp, \sqsubseteq^\sharp)$*, a Galois connection is a pair of maps such that*

$$\alpha \in P \to P^\sharp$$

$$\gamma \in P^\sharp \to P$$

$$\forall a \in P : \forall b \in P^\sharp : \alpha(a) \sqsubseteq^\sharp b \iff a \sqsubseteq \gamma(b)$$

in which case we write

$$(P, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (P^\sharp, \sqsubseteq^\sharp)$$

2.1.3 Fixpoints and Operation Abstraction

The operations on the concrete domain have to be abstracted as operations on the abstract domain in a sound way.

Definition 7. *Given a concrete operation $f \in A \to A$, an abstraction of f is a function $f^\sharp \in A^\sharp \to A^\sharp$ such that*

$$\forall a^\sharp \in A^\sharp : f(\gamma(a^\sharp)) \sqsubseteq \gamma(f^\sharp(a^\sharp))$$

Many objects or properties in a concrete domain needs to be expressed as fixpoints of operators. We recall the definition of this concept and two important results for abstract interpretation.

Definition 8 (Fixpoint)**.** *A fixpoint of a function $f \in A \to A$ is any element $a \in A$ such that $f(a) = a$. If $f(a) \sqsubseteq a$, we say that a is a post-fixpoint. Analogously, if $a \sqsubseteq f(a)$, we say that a is a pre-fixpoint. When it exists, we denote the least fixpoint of f as lfp f. Moreover, $lfp_a f$ denotes the least fixpoint of f greater than a.*

Theorem 1 (Tarski's theorem)**.** *The set of fixpoints of a monotonic function $f \in A \to A$ in a complete lattice A is a non-empty complete lattice.*

Theorem 2 (Cousot and Cousot)**.** *If $f \in A \to A$ is a monotonic function in a complete partial order A, and a is a pre-fixpoint of A, then the following sequence:*

$$f(n) = \begin{cases} a & \text{if } \delta = 0 \\ f(x_\beta) & \text{if } \delta = \beta + 1 \\ \sqcup\{x_\beta | \beta < \delta\} & \text{if } \delta \text{ is a limit ordinal} \end{cases}$$

converges towards $lfp_a f$. Moreover, if f is a join-morphism, then $lfp_a f = x_\omega$ (i.e., the iteration converges after a countable number of steps).

This theorem suggests an iterative way of computing fixpoints, being necessary to ensure that the involved sequences converge in finite time.

Once we have expressed a property in the concrete domain as a fixpoint of a particular function, the natural step is to abstract it as a least fixpoint of an abstract version of the function. The following theorem shows us that we can abstract a concrete least fixpoint as an abstract post-fixpoint.

Theorem 3. *If $f^\sharp$ is a sound abstraction of f, $a \sqsubseteq \gamma(a^\sharp)$, and $x^\sharp$ satisfies $f^\sharp(x^\sharp) \sqsubseteq^\sharp x^\sharp$ and $a^\sharp \sqsubseteq^\sharp x^\sharp$, then $lfp_a f \sqsubseteq \gamma(x^\sharp)$.*

In order to enforce termination of a fixpoint computation, when there exist infinite increasing chains in the lattice, Cousot and Cousot proposed the use of *widening* operators that perform an extrapolation to accelerate the computation.

Definition 9 (Widening operator)**.** *A widening operator is a binary operator* $\triangledown \in (D^\sharp \times D^\sharp) \to D^\sharp$ *satisfying:*

- $\forall x^\sharp, y^\sharp : x^\sharp \sqsubseteq^\sharp x^\sharp \triangledown y^\sharp$ *and* $y^\sharp \sqsubseteq^\sharp x^\sharp \triangledown y^\sharp$*;*
- *for any sequence* $(y_i^\sharp)_{i\in\mathbb{N}}$*, the sequence defined as* $x_0^\sharp \stackrel{def}{=} y_0^\sharp$ *and* $x_{i+1}^\sharp \stackrel{def}{=} x_i^\sharp \triangledown y_i^\sharp$ *is not strictly increasing.*

With this operators, the following theorem shows us that we can approximate $\mathrm{lfp}_a f$ with a finite number of iterations.

Theorem 4. *If* $f^\sharp$ *is a sound abstraction of* f *and* $a \sqsubseteq \gamma(a^\sharp)$*, then the sequence* $x_0^\sharp \stackrel{def}{=} a^\sharp, x_{i+1}^\sharp \stackrel{def}{=} x_i^\sharp \triangledown f^\sharp(x_i^\sharp)$ *reaches a stable iterate* $x_\beta^\sharp = x_{\beta+1}^\sharp$ *for some* $\beta < \omega$*. Moreover,* $lfp_a f \sqsubseteq \gamma(x_\beta^\sharp)$*.*

2.2 Top-down Abstract Interpretation of Logic Programs in PLAI

PLAI is an abstract interpretation-based framework for static analysis that implements the algorithms proposed by [Bruynooghe, 1991a] with the optimizations described in [Muthukumar and Hermenegildo, 1990]. In logic programming, all possible program states can be represented by an infinite set of proof trees (*AND* trees, *SLD* derivations). The core idea proposed in [Bruynooghe, 1991a] is to represent this infinite set of *AND* trees by a finite abstract *AND-OR* graph, constructed by following an abstract interpretation-based procedure. PLAI is domain-independent, in the sense that new abstract domains can be easily implemented and integrated, by using a particular interface. In this section we present an overview of the core algorithms of PLAI, and how a new abstract domain can be integrated into the framework. For a complete description of this framework and abstract interpretation of logic programs in general, we refer the reader to [Bruynooghe, 1991a, Muthukumar and Hermenegildo, 1990]. For good examples of abstract domains integrated into PLAI, we refer the reader to [Bueno et al., 2004, Muthukumar and Hermenegildo, 1989].

2.2.1 Algorithm

Goal dependent abstract interpretation takes as input a program P, an abstract domain $D^\sharp$, and a description $Q^\sharp$ of the possible initial queries to P, given as a set of abstract queries. An *abstract query* is a pair (L, λ), where L is an atom (corresponding to one of the exported predicates) and $\lambda \in D^\sharp$ describes the initial calls to L. A set $Q^\sharp$ represents

the set of queries $\gamma(Q^\sharp)$, which is defined as $\gamma(Q^\sharp) = \{(L,\theta)|(L,\lambda) \in Q^\sharp \wedge \theta \in \gamma(\lambda)\}$. The result of the analysis is a set of triples $Analysis(P,Q^\sharp,D^\sharp) = \{\langle L_p, \lambda^c, \lambda^s\rangle |\ p \text{ is a predicate of } P\}$, where L_p is a (program) atom for predicate p.

Consider a clause $h : -p_1,\cdots,p_n$. Let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal $p_i, 1 \leq i \leq n$ in this clause. Then λ_i and λ_{i+1} are, respectively, the *abstract call substitution* and the *abstract success substitution* for the subgoal p_i. For this same clause, λ_1 is the *abstract entry substitution* (denoted β_{entry} when it is projected onto the variables of h) and λ_{n+1} is the *abstract exit substitution* (denoted β_{exit} when it is projected onto the variables of h).

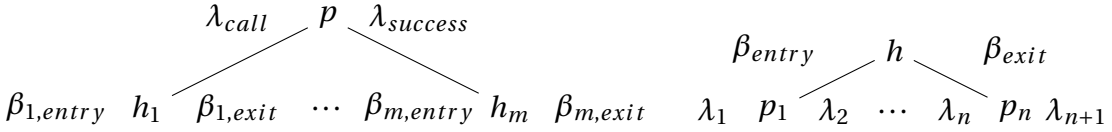

Figure 2.1: The top-down abstract interpretation process of PLAI.

Computing the *success* substitution ($\lambda_{success}$) from the *call* substitution (λ_{call}) is done as follows (see Figure 2.1, left-hand side). Given a call substitution λ_{call} for a subgoal p, let $h_1,\cdots,h_m$ be the heads of the clauses defining p. First, the entry substitutions $\beta^1_{entry},\cdots,\beta^m_{entry}$ for these clauses are computed. Then, we compute their success substitutions $\lambda^1_{success},\cdots,\lambda^m_{success}$ from their corresponding exit substitutions (the algorithm for computing the exit substitutions is explained later in this section). At this point, all different success substitutions can be considered for the rest of the analysis, or a single success substitution $\lambda_{success}$ for subgoal p can be computed by using the LUB (*least upper bound*) operation of the abstract domain. In the first case, we say that the analysis is *multi-variant* on successes[1], in the second case it is not.

Computing the exit substitution from the entry substitution of a clause is straightforward (see Figure 2.1, right-hand side). Given a clause h:-$p_1,\cdots,p_n$ and an entry substitution β_{entry} for the clause head h, first, the call substitution for p_1, λ_1, is obtained by adding to β_{entry} an abstraction for the free variables in the clause. Then, the success substitution λ_2 for p_1 is computed as explained above (essentially, by repeating this same process for the clauses whose heads unify with p_1). In the same way, $\lambda_3,\cdots,\lambda_{n+1}$ are computed. The exit substitution β_{exit} for this clause is λ_{n+1} projected onto the variables of the head h.

It is possible to compute different entry substitutions for the same clause, which are originated from different subgoals.

Thus, different call substitutions for the subgoals in the body of the clause can appear. These substitutions can be collapsed using the LUB operation, or, alternatively a different node in the graph can be created. In the latter solution, different nodes exist

[1]This behavior is configurable with a *flag* in CiaoPP.

in the graph for each call substitution and subgoal, and the analysis becomes *multi-variant* on calls.

The framework is domain-independent. To use it for a particular domain, we need to define the abstract elements and some operations that are called from the general analysis. These operations implement the abstract unification, the order relation $\sqsubseteq$, and the *least upper bound* ($\sqcup$). The domain dependent functions are:

- $call\_to\_entry(p(\overline{u}), C, \lambda)$ which gives an abstract substitution describing the effects on $vars(C)$ of unifying $p(\overline{u})$ with $head(C)$ given an abstract substitution λ describing $\overline{u}$,
- $exit\_to\_success(\lambda, p(\overline{u}), C, \beta)$ which gives an abstract substitution describing $\overline{u}$ accordingly to β (which describes $vars(head(C))$) and the effects of unifying $p(\overline{u})$ with $head(C)$ under the abstract substitution λ describing $\overline{u}$,
- $extend(\lambda, \lambda')$ which extends abstract substitution λ to incorporate the information in λ' in a consistent way,
- $project\_in(\overline{v}, \lambda)$ which extends λ so that it refers to all of the variables $\overline{v}$,
- $project\_out(\overline{v}, \lambda)$ which restricts λ to only the variables $\overline{v}$.

The process described so far is schematized in algorithm 1 and 2, assuming the *mono-variant on success* version, i.e., by combining all the success substitutions using the LUB operation.

Algorithm 1: *entry_to_exit*: Computes the exit substitution for a clause.

Data: A clause C $\equiv h : -q_1(\overline{u}_1), \cdots, q_m(\overline{u}_m)$; an entry substitution β_{entry}
Result: An exit substitution β_{exit}
A_1 := project_in(vars(C),β_{entry}) ;
for *i:= 1 to m* **do**
  A_{i+1} := call_to_success($q_i(\overline{u}_i)$, A_i)
return project_out(vars(h), A_{n+1}) ;

Recursion

The algorithm described so far generates an infinite AND-OR graph in the presence of recursive predicates. In order to handle recursion, a fixpoint computation is required. In [Muthukumar and Hermenegildo, 1990] a fixpoint algorithm was proposed. This algorithm localizes fixpoint computations to only strongly connected components of (mutually) recursive predicates. Additionally, an initial approximation to the fixpoint is

Algorithm 2: *call_to_success*: Computes the success substitution for a particular call.

Data: A goal $p(\overline{u})$; an abstract call substitution λ_{call}
Result: A success substitution $\lambda_{success}$
λ := project_out($\overline{u}$,λ_{call}) ;
λ' := $\bot$;
for *each clause C which matches* $p(\overline{u})$ **do**
  β_{exit} := entry_to_exit(C, call_to_entry($p(\overline{u})$,C,λ)) ;
  λ':=$\lambda' \sqcup$ exit_to_success($\lambda, p(\overline{u}), C, \beta_{exit}$) ;
return extend(λ_{call},λ') ;

computed from the non-recursive clauses of the recursive predicate. The convergence of the fixpoint is accelerated by updating this value with the information obtained just after each clause is analyzed. The algorithms 3 and 4 show a high-level description of this fixpoint computation.

Algorithm 3: *call_to_success_recursive*: Computes the success substitution for a particular call, handling recursive predicates.

Data: A goal $p(\overline{u})$; an abstract call substitution λ_{call}
Result: A success substitution $\lambda_{success}$
λ := project_out($\overline{u}$,λ_{call}) ;
λ' := $\bot$;
for *each non-recursive clause C which matches* $p(\overline{u})$ **do**
  β_{exit} := entry_to_exit(C, call_to_entry($p(\overline{u})$,C,λ)) ;
  λ':=$\lambda' \sqcup$ exit_to_success($\lambda, p(\overline{u}), C, \beta_{exit}$) ;
λ'' := fixpoint($p(\overline{u})$,λ,λ') ;
return extend(λ_{call},λ'') ;

Algorithm 4: *fixpoint*: Computes the success substitution for a call to a recursive predicate.

Data: A goal $p(\overline{u})$; an initial abstract call substitution λ; a call substitution λ'
Result: A success substitution $\lambda_{success}$
λ'' := λ'; **for** *each recursive clause C which matches* $p(\overline{u})$ **do**
  β_{exit} := entry_to_exit(C, call_to_entry($p(\overline{u})$,C,λ)) ;
  λ'':=$\lambda'' \sqcup$ exit_to_success($\lambda, p(\overline{u}), C, \beta_{exit}$) ;
if $\lambda'' = \lambda'$ **then**
  return λ''
else
  return extend(λ_{call},λ')
;

Chapter 3

Resource Analysis by Abstract Interpretation

In this chapter we present the resource analysis for logic programs implemented as an abstract domain in the PLAI framework, together with the modifications, extensions and improvements we have performed to it in this thesis. The analysis is based on the sized types abstract domain. Sized types are regular types extended with structural (shape) information that express both lower and upper bounds on the size of a set of terms and their subterms at any position and depth. They also allow relating the sizes of terms and subterms occurring at different argument positions in logic predicates. With this information, the resource analysis can infer both lower and upper bounds on the resources used by all the procedures in a program as functions on the sizes of input terms (and subterms). The abstract domain operations are based on the setting up and solving of recurrence equations for inferring both size and resource usage functions.

3.1 Introduction

Resource usage analysis infers the aggregation of some numerical properties (named *resources*), like memory usage, time spent in computation, or bytes sent over a wire, throughout the execution of a piece of code. This information is returned in the form of functions of the sizes of some input arguments.

The starting point of this approach is the methodology outlined by [Debray et al., 1990, Debray and Lin, 1993, Debray et al., 1997], characterized by the setting up of recurrence equations. This methodology basically performs, previous to the resource usage analysis, a size and cardinality analysis for inferring size relations and bounds on the number of solutions computed by a predicate, respectively. This information is ultimately used to obtain the resource usage bounds. The main limitation of this ap-

proach is that it is able to cope with size information about subterms in a very limited way. For example, consider a predicate which computes the factorials of a list:

```
1 % listfact(+L, -FL).
2 listfact([],    []).
3 listfact([E|R],[F|FR]) :-
4  fact(E, F),
5  listfact(R, FR).
```

```
1 % fact(+N, -F).
2 fact(0,1).
3 fact(N,M) :- N1 is N - 1,
4              fact(N1, M1),
5              M is N * M1.
```

Intuitively, the best bound for the running time of this program for a list L is $c_1 + \sum_{e \in L}\big(c_2 + time_{fact}(e)\big)$, where c_1 and c_2 are constants related to unification and calling costs. But with no further information, the upper bound for the elements of L must be ∞ to be on the safe side, and then the returned overall time bound must also be ∞.

The sized types analysis proposed in [Serrano et al., 2013] solve this problem by giving information for all the *dimensions* of terms. Thus, the use of this analysis as a basis of resource analysis is a clear advantage over the previous proposals. The ease of the integration with this analysis is possible due to the use of a common abstract interpretation framework (PLAI).

In this framework, the resource analysis is implemented as an *abstract domain*, obtaining features such as *multivariance*, efficient fixpoints, and assertion-based verification and user interaction for free.

The rest of this chapter is organized as follows. First, in Section 3.2 we give a brief state of the art of resource usage analysis. In Section 5.2 we present an overview of the resource analysis. In section 3.4 we explain the sized type analysis which is a fundamental part of the resource analysis. Afterwards, Section 3.5 gives the details of the resource usage analysis. Finally, in Section 3.6 we draw the conclusions of the chapter.

3.2 State of the Art in Resource Analysis

The approach to cost analysis based on setting up and solving recurrence equations was proposed in [Wegbreit, 1975] and has been developed significantly in subsequent work. For example, in [Rosendahl, 1989] an automatic upper-bound analysis was presented based on an abstract interpretation of a step-counting version of a functional program, in order to infer both execution time and execution steps. However, size measures could not automatically be inferred and the experimental section showed few details about the practicality of the analysis. The cost analysis in [Vasconcelos and Hammond, 2003] deals with recursive, polymorphic and higher-order functional programs. In the context of Logic Programming, a semi-automatic analysis was presented in [Debray et al., 1990, Debray and Lin, 1993] that inferred upper-bounds on the number of execution steps, given as functions on the input data sizes. It also pro-

posed techniques to address the additional challenges posed by the Logic Programming paradigm, as for example, dealing with the generation of multiple solutions via backtracking. However, a shortcoming of the approach was its loss in precision in the presence of divide-and-conquer programs in which the sizes of the output arguments of the "divide" predicates are dependent. This approach was later fully automated (by integrating it into the CiaoPP system and automatically providing *modes and size measures*) and extended to inferring both upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [Debray et al., 1997, Hermenegildo et al., 2005b]. In addition, [Debray et al., 1997] introduced the setting up of non-deterministic recurrence relations for the class of divide-and-conquer programs mentioned above, and proposed a technique for computing approximated closed form bound functions for some of them. Such a technique was based on bounding the number of terminal and non-terminal nodes in the set of computation trees corresponding to the evaluation of the non-deterministic recurrence relations, and bounding the cost of such nodes. Non-deterministic recurrence relations were also used and further developed in [Albert et al., 2008, Albert et al., 2011a] (named Cost Relations). The approach in [Debray et al., 1990, Debray and Lin, 1993, Debray et al., 1997] was generalized in [Navas et al., 2007a] to infer *user-defined resources* (by using an extension of the Ciao assertion language [Hermenegildo et al., 2012a]), and was further improved in [Serrano et al., 2014a] by defining the resource analysis itself as an *abstract domain* that is integrated into the PLAI abstract interpretation framework [Muthukumar and Hermenegildo, 1992, Puebla and Hermenegildo, 1996] of CiaoPP.

Other approaches to static analysis based on the transformation of the analyzed code into an intermediate representation have been proposed for analyzing low-level languages [Henriksen and Gallagher, 2006] and Java (by means of a transformation into Java bytecode) [Albert et al., 2007]. In [Albert et al., 2007], a cost relation system is obtained directly for these bytecode programs and solve it using a specialized solver, finding upper bounds for such cost relations. In [Navas et al., 2008] the bytecode is first transformed into Horn clauses.

The general resource analyzer in [Navas et al., 2007a] was also instantiated in [Mera et al., 2008] for the estimation of execution times of logic programs running on a bytecode-based abstract machine.

A number of static analyses are also aimed at worst case execution time (WCET), usually for imperative languages in different application domains (see e.g., [Wilhelm et al., 2008] and its references). The worst-case analysis presented in [Jayaseelan et al., 2006], which is not based on recurrence equation solving, distinguishes instruction-specific (not proportional to time, but to data) from pipeline-specific (roughly proportional to time) energy consumption. However, these worst case analysis methods do not obtain functions on input data sizes as result, but rather absolute maximum execution times, in general requiring annotations from the user indicating upper bounds for the numbers of iterations of each loop.

Abstract interpretation has been applied successfully for inferring useful information about programs, such as in [Hermenegildo et al., 2005b], and also for run-time error detection in sequential and concurrent embedded avionic systems [Miné, 2012]. It has also been proved useful for the implementation of resource consumption analysis on sequential logic programs [Serrano et al., 2014a]. In [Miné, 2012], abstract interpretation is combined with Rely-guarantee proof methods to implement a thread-modular static analyzer for run-time error detection on concurrent embedded systems. However, there is very few work done on static analysis of the resource usage of concurrent programs using abstract interpretation.

3.3 Overview

We give now an overview of the approach for resource usage analysis by abstract interpretation present in CiaoPP, showing the main ideas by using the classical `append/3` predicate as a running example:

```
1 append([],    S, S).
2 append([E|R], S, [E|T]) :- append(R, S, T).
```

The process starts by performing the regular type analysis present in the CiaoPP system [Vaucheret and Bueno, 2002]. In our example, the system infers that for any call to the predicate `append(X, Y, Z)` with `X` and `Y` bound to lists of numbers and `Z` a free variable, if the call succeeds, then `Z` also gets bound to a list of numbers. The set of "list of numbers" is represented by the regular type $listnum$, defined as follows:

```
1 listnum := [] | [num | listnum].
```

From this regular type definition, sized type schemes are derived. The sized type schema $listnum\text{-}s$ is derived from $listnum$. This schema corresponds to a list whose length is between α and β, containing numbers between γ and δ.

$$listnum\text{-}s \rightarrow listnum^{(\alpha,\beta)}(num^{(\gamma,\delta)})$$

From now on, in the examples we will use ln and n instead of $listnum$ and num for the sake of conciseness. The next phase involves relating the sized types of the different arguments to the `append/3` predicate using recurrence (in)equations. Let $size_X$ denote the sized type schema for argument `X` in a call `append(X, Y, Z)` (from the regular type inferred by a previous analysis). We have that $size_X$ denotes $ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)})$. Similarly, the sized type schema for the output argument `Z` is $ln^{(\alpha_Z,\beta_Z)}(n^{(\gamma_Z,\delta_Z)})$, denoted by $size_Z$. We are interested in expressing bounds on the length of the output list `Z` and the values of its elements as a function of size bounds for the input lists `X` and `Y` (and their elements). For this, we set up a system of inequations. For instance, the

inequations that are set up to express a lower bound on the length of the output argument Z, denoted α_Z, as a function on the size bounds of the input arguments X and Y, and their subarguments (α_X, β_X, γ_X, δ_X, α_Y, β_Y, γ_Y, and δ_Y) are:

$$\alpha_Z\begin{pmatrix}\alpha_X,\beta_X,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} \geq \begin{cases}\alpha_Y & \text{if } \alpha_X = 0\\ 1+\alpha_Z\begin{pmatrix}\alpha_X-1,\beta_X-1,\gamma_X,\delta_X,\\ \alpha_Y,\beta_Y,\gamma_Y,\delta_Y\end{pmatrix} & \text{if } \alpha_X > 0\end{cases}$$

Note that in the recurrence inequation set up for the second clause of append/3, the expression $\alpha_X - 1$ (respectively $\beta_X - 1$) represents the size relationship that a lower (respectively upper) bound on the length of the list in the first argument of the recursive call to append/3 is one unit less than the length of the first argument in the clause head.

As the number of size variables grows, the set of inequations becomes too large. Thus, in [Serrano et al., 2014a] the authors propose a compact representation, which allows us to grasp all the relations in one view. The first change in this representation is to write the parameters to size functions directly as sized types. Now, the parameters to the α_Z function are the sized type schemas corresponding to the arguments X and Y of the append/3 predicate:

$$\alpha_Z\begin{pmatrix}ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)})\\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\end{pmatrix} \geq \begin{cases}\alpha_Y & \text{if } \alpha_X = 0\\ 1+\alpha_Z\begin{pmatrix}ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)})\\ ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\end{pmatrix} & \text{if } \alpha_X > 0\end{cases}$$

In a second step, all the relations of a single sized type are grouped together. Throughout this chapter we use a representation using $\lesseqgtr$ for the symbols $\geq$ and $\leq$ that are always paired, as the authors of [Serrano et al., 2014a] proposed. In the implementation, constraints for each variable are kept apart and solved separatedly.

After setting up the corresponding system of inequations for the output argument Z of append/3, and solving it, we obtain the following expression:

$$size_Z(size_X, size_Y) \lesseqgtr ln^{(\alpha_X+\alpha_Y,\beta_X+\beta_Y)}(n^{(\min(\gamma_X,\gamma_Y),\max(\delta_X,\delta_Y))})$$

that represents, among others, the relation $\alpha_z \geq \alpha_X + \alpha_Y$ (resp. $\beta_z \leq \beta_X + \beta_Y$), expressing that a lower (resp. upper) bound on the length of the output list Z, denoted α_z (resp. β_z), is the addition of the lower (resp. upper) bounds on the lengths of X and Y. It also represents the relation $\gamma_Z \geq \min(\gamma_X, \gamma_Y)$ (resp. $\delta_Z \leq \max(\delta_X, \delta_Y)$), which expresses that a lower (resp. upper) bound on the size of the elements of the list Z, denoted γ_z (resp. δ_z), is the minimum (resp. maximum) of the lower (resp. upper) bounds on the sizes of the elements of the input lists X and Y.

Resource analysis builds upon the sized type analysis and adds recurrence equations for each resource we want to analyze. Apart from that, when considering logic programs, we have to take into account that they can fail or have multiple solutions when executed, so we need an auxiliary *cardinality analysis* to get correct results.

Let s_L and s_U denote lower and upper bounds on the number of solutions for `append/3`. Following the program structure we can infer:

$$\begin{array}{rcl} s_L\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \geq & 1 \\ s_L\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \geq & s_L\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{array}$$

$$\begin{array}{rcl} s_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & 1 \\ s_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{array}$$

Since $s_L \leq s_U$, the solution to these inequations must be $(s_L, s_U) = (1, 1)$. Thus, we have inferred that `append/3` has at least (and at most) one solution: it behaves like a function. When setting up the equations, the analysis uses the result of the non-failure analysis to see that `append/3` cannot fail when given lists as arguments. If not, the lower bound is 0.

Now we move forward to the resource usage approximation. We are considering the number of resolution steps performed by a call to `append/3` (we will only focus on upper bounds, r_U, for brevity). For the first clause, it is clear that only one resolution step is needed, so:

$$r_U\left(ln^{(0,0)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq 1$$

The second clause performs one resolution step plus all the resolution steps performed by all possible backtrackings over the call in the body of the clause. This number can be bounded as a function of the number of solutions. Thus, the equation reads:

$$\begin{array}{rcl} r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), size_Y\right) & \leq & 1 + s_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\ & & \times\ r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \\ & = & 1 + r_U\left(ln^{(\alpha_X-1,\beta_X-1)}(n^{(\gamma_X,\delta_X)}), size_Y\right) \end{array}$$

Solving these equations the analysis infers that an upper bound on the number of resolution steps is the (upper bound on) the length of the input list `X` plus one. This is expressed as:

$$r_U\left(ln^{(\alpha_X,\beta_X)}(n^{(\gamma_X,\delta_X)}), ln^{(\alpha_Y,\beta_Y)}(n^{(\gamma_Y,\delta_Y)})\right) \leq \beta_X + 1$$

3.4 Sized Types

Sized types are symbolic representations that summarize the size of a set of terms, similar to those found in [Hughes et al., 1996] for functional languages. In CiaoPP's size analysis, the *sized types* schemas are automatically built from the regular types inferred by a previous analysis [Vaucheret and Bueno, 2002]. In this section we show an overview of size analysis, which is the base of the resource analysis based on abstract interpretation of CiaoPP. We refer the readers to [Serrano et al., 2013] for a detailed explanation.

The sized types analysis is based, as we already mentioned, on *regular types*. Among several representations of regular types used in the literature, this analysis is using one based on *regular term grammars*, equivalent to [Dart and Zobel, 1992] but with some adaptations. A *type term* is either a *base type* η_i (taken from a finite set), a *type symbol* τ_i (taken from an infinite set), or a term of the form $f(\phi_1,\ldots,\phi_n)$, where f is a n-ary function symbol (taken from an infinite set) and $\phi_1,\ldots,\phi_n$ are *type terms*. A *type rule* has the form $\tau \rightarrow \phi$, where τ is a *type symbol* and ϕ a *type term*. A *regular term grammar* Υ is a set of *type rules*.

Sized types analysis is built as an abstract domain in the PLAI abstract interpretation framework of CiaoPP. The PLAI algorithm abstracts execution AND-OR trees similarly to [Bruynooghe, 1991b] but represents the abstract executions *implicitly* and computes fixpoints efficiently using memo tables, dependency tracking, etc. It takes as input a pair (L,λ_c) representing an entry point (predicate) along with an abstraction of the call patterns (in the chosen *abstract domain*) and produces an abstraction λ_o which over-approximates information at all program points (for all procedure versions).

Formally speaking, a *sized type* is an abstraction of a set of Herbrand terms which are a subset of some regular type τ and meet some lower- and upper-bound size constraints on the number of *type rule applications* needed to generate the terms. The grammar for sized types is the following:

sized-type	::=	η^{bounds}	η base type
	|	τ^{bounds}(*sized-args*)	τ recursive type symbol
	|	τ(*sized-args*)	τ non-recursive type symbol
bounds	::=	$nob \mid (n,m)$	$n,m \in \mathbb{N}, m \geq n$
sized-args	::=	$\epsilon \mid$ *sized-arg, sized-args*	
sized-arg	::=	*sized-type*$_{position}$	
position	::=	$\epsilon \mid \langle f,n \rangle$	f functor, $0 \leq n \leq$ arity of f

However, in the abstract domain it is necessary to refer to sets of sized types which satisfy certain constraints on their bounds, i.e. it is necessary to have symbolic expressions as bounds instead of concrete ground values. For that purpose, the con-

cept of *sized type schemas* is introduced: a schema is just a sized type with variables in bound positions, i.e., where n and m in the pair (n,m) defining the symbol *bounds* in the grammar above are variables (called bound variables), along with a set of constraints over those variables. Such variables are called *bound variables*. We will denote $sized(\tau)$ the sized type schema corresponding to a regular type τ where all the bound variables are fresh.

The full abstract domain of sized types is an extension of sized type schemas to several predicate variables. Each abstract element is a triple $\langle t,d,r\rangle$ such that:

1. t is a set of $v \rightarrow (sized(\tau),c)$, where v is a variable, τ its regular type and c is its classification. Subgoal variables can be classified as *output*, *relevant*, or *irrelevant*. Variables appearing in the clause body but not in the head are classified as *clausal*;
2. d (the *domain*) is a set of constraints over the relevant variables;
3. r (the *relations*) is a set of relations among bound variables.

For example, the final abstract elements corresponding to the clauses of the `listfact` example can be found below. The equations have already been normalized into their simplest form, and the variables refer to the predicate arguments are in normal form. $listfact$ refers implicitly to the solution of the joint equations: it is the recurrence we need to solve. The position element $\langle .,1\rangle$ has been dropped for readability from ln.

$$\lambda_1' = \left\langle \begin{array}{c} \{L \rightarrow (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \rightarrow (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.)\} \\ \{\alpha_1 = 1, \beta_1 = 1\}, \{ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(1,1)}(n^{nob})\} \end{array} \right\rangle$$

$$\lambda_2' = \left\langle \begin{array}{c} \left\{ \begin{array}{c} L \rightarrow (ln^{(\alpha_1,\beta_1)}(n^{(\gamma_1,\delta_1)}), rel.), FL \rightarrow (ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}), out.), \\ E \rightarrow (n^{(\gamma_3,\delta_3)}, cl.), R \rightarrow (ln^{(\alpha_4,\beta_4)}(n^{(\gamma_4,\delta_4)}), cl.), \\ F \rightarrow (n^{(\gamma_5,\delta_5)}, cl.), FR \rightarrow (ln^{(\alpha_6,\beta_6)}(n^{(\gamma_6,\delta_6)}), cl.) \end{array} \right\} \\ \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} ln^{(\alpha_2,\beta_2)}(n^{(\gamma_2,\delta_2)}) \lesseqgtr ln^{(\alpha'+1,\beta'+1)}(n^{(\min(\gamma_1!,\gamma'),\max(\delta_1!,\delta'))}) \\ ln^{(\alpha',\beta')}(n^{(\gamma',\delta')}) \lesseqgtr listfact\left(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})\right) \end{array} \right\} \end{array} \right\rangle$$

3.5 The Resources Abstract Domain

The main advantage that exhibits the resource analysis based on abstract interpretation, implemented in CiaoPP, is that it uses the added power of sized types to develop a better resource analysis which infers upper and lower bounds on the amount of resources used by each predicate as a function of the sized type schemas of the input arguments. For this reason, the abstract domain for resource analysis is in fact an extension of the sized types abstract domain. Regarding the resource consumption itself,

there are two places that need to be taken into account in order to abstract the resource consumption of a logic program (see [Navas et al., 2007a]):

- When entering a clause: some resources may be needed during unification of the call (subgoal) and the clause head, the preparation of entering that clause, and any work done when all the literals of the clause have been processed. This cost, dependent on the head h, is called *head cost*, $\varphi(h)$.
- Before calling a literal q: some resources may be used to prepare a call to a body literal (e.g., constructing the actual arguments). The amount of these resources is known as *literal cost* and is represented by $\omega(q)$.

Lets first make focus on the case of estimating upper bounds on resource usages. For simplicity, lets assume first that the predicates we are dealing with are deterministic and do not fail. Then, we can bound the resource consumption of a clause $C \equiv p(\bar{x}) :- q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)$, denoted $r_{U,clause}$:

$$r_{U,clause}(C) \leq \varphi(p(\bar{x})) + \sum_{i=1}^{n} \left(\omega(q_i(\bar{x}_i)) + r_{U,pred}(q_i(\bar{x}_i))\right)$$

As in sized type analysis, the sizes of some input arguments may be explicitly computed, or, otherwise, they are represented with a generic expression, giving rise (in the case of recursive clauses) to a recurrence equation that it is necessary to solve in order to find closed-form resource usage functions.

The resource usage of a predicate, $r_{U,pred}$, depending on its input data sizes, is obtained from the resource usage of the clauses defining it, by taking the maximum of the equation expressions that meet the constraints on the input data sizes (i.e., have the same domain).

Now, as we are analyzing logic programs, it is necessary to consider two extra features that this paradigm presents:

- We may execute a literal more than once on backtracking. To bound the number of times a literal is executed, we need to know the *number of solutions* each literal (to its left) can generate. Using the information provided by cardinality analysis, the number of times a literal is executed is at most the product of the upper bound on the number of solutions, s_U, of all the previous literals in the clause. We get:

$$\begin{aligned} &r_{U,clause}\left(p(\bar{x}) :- q_1(\bar{x}_1), \ldots, q_n(\bar{x}_n)\right) \\ &\quad \leq \varphi(p(\bar{x})) + \sum_{i=1}^{n} \left(\prod_{j=1}^{i-1} s_{pred}(q_j(\bar{x}_j))\right) \left(\omega(q_i(\bar{x}_i)) + r_{U,pred}(q_i(\bar{x}_i))\right) \end{aligned}$$

- Also, in logic programming more than one clause may unify with a given subgoal. In that case it is incorrect to take the maximum of the resource usages of each clause when setting up the recurrence equations. A correct solution is to take the sum of every set of equations with a common domain, but the bound becomes then very imprecise. Finer-grained possibilities can be considered by using different *aggregation* procedures per resource. It is also possible to use the information from the *determinacy* analysis [Lopez-Garcia et al., 2010a],also included in CiaoPP as an abstract domain, to take the maximum of the resource of each clause in the case that the predicate is deterministic.

Lower bounds analysis is similar, but needs to take into account the possibility of failure, which stops clause execution and forces backtracking. Basically, no resource usage should be added beyond the point where failure may happen. For this reason, the resource analysis use the non-failure analysis already present in CiaoPP. Also, the aggregation of clauses with a common domain must be different to that used in the upper bounds case. The simplest solution is to just take the minimum of the clauses. However, this again leads to very rough bounds.

3.5.1 Cardinality Analysis

We have already discussed why cardinality analysis (which estimates bounds on the number of solutions) is instrumental in resource analysis of logic programs. We can consider the number of solutions as another resource, but, due to its importance, we treat it separately.

An upper bound on the number of solutions of a single clause could be gathered by multiplying the number of solutions of its body literals:

$$s_{U,clause}\left(p(\bar{x}) :- q_1(\bar{x}_1),\ldots,q_n(\bar{x}_n)\right) \leq \prod_{i=1}^{n} s_{U,pred}(q_i(\bar{x}_i))$$

For aggregation we need to add the equations with a common domain, to get a recurrence equation system. These equations will be solved later to get a closed-form function giving an upper-bound on the number of solutions.

It is important to remark that many improvements can be added to this simple cardinality analysis to make it more precise. Some of them are discussed in [Debray and Lin, 1993], like maintaining separate bounds for the relation defined by the predicate and the number of solutions for a particular input, or dealing with mutually exclusive clauses by performing the max operation, instead of the addition operation when aggregating. However, our focus here is the definition of an abstract domain, and see whether a simple definition produces comparable results for the resource usage analysis.

One improvement in the precision of cardinality analysis included in the resource analysis is the use of the determinacy analysis present in CiaoPP [Lopez-Garcia et al., 2010a]. If such analysis infers that a predicate is deterministic, the analysis set the upper bound for the number of solutions to 1.

In the case of lower bounds, it is necessary to know for each clause whether it may fail or not. For that reason the resources domain uses the non-failure analysis already present in CiaoPP [Bueno et al., 2004]. In case of a possible failure, the lower bound on cardinality is set to 0.

3.5.2 The Abstract Elements

Within the PLAI abstract interpretation framework [Muthukumar and Hermenegildo, 1992, Puebla and Hermenegildo, 1996] an analysis is defined by the abstract elements involved in it and a set of operations. We refer the reader to the Section 2.2 for an overview of the overall framework.

The abstract elements of the resource usage domain are derived from sized type analysis by adding some extra components. In particular:

1. The *current variable for solutions*, and *current variable for each resource*.
2. A boolean element for telling whether we have already found a failing literal.
3. An abstract element from the non-failure domain.
4. An abstract element encoding information about determinacy.

The abstract elements are denoted by $\langle (s_L, s_U), v_{resources}, failed?, d, r, nf, det \rangle$ where (s_L, s_U) are the lower and upper bound variables for the number of solutions, $v_{resources}$ is a set of pairs (r_L, r_U) giving the lower and upper bound variables for each resource, $failed?$ is a boolean element (`true` or `false`), d and r are defined as in the sized type abstract domain, and nf and det can take values `not_fails/fails` and `non_det/is_det` respectively, as explained in [Lopez-Garcia et al., 2010a, Bueno et al., 2004].

Lets assume that we are given the definition of a set of resources, which are fixed throughout the whole analysis process. For each resource r we have: its head cost, φ_r, which takes a clause head as parameter; its literal cost, ω_r, which takes a literal as parameter; its aggregation procedure, Γ_r, which takes the equations for each of the clauses and creates a new set of recurrence equations from them; and the default upper $\bot_{r,U}$ and lower $\bot_{r,L}$ bound on resource usage.

We will continue with the analysis of `listfact` as a running example. We assume that the only resource to be analyzed is the "number of resolution steps," which uses the following parameters:

$$\varphi = 1, \quad \omega = 0, \quad \Gamma_r = +, \quad (\bot_L, \bot_U) = (0,0)$$

In what follows, we explain informally the operations that form the resource usage abstract domain.

The $\sqsubseteq$, $\sqcup$ Operations and the $\bot$ Element.

There is no decidable definition for $\sqsubseteq$ or $\sqcup$, because there is no general algorithm for checking the inclusion or union of sets of integers defined by recurrence relations. Instead, for the inequation components it is just checked whether one is a subset of another one, up to variable renaming, or it is performed a syntactic union of the inequations. The ordering is finished by taking the product order with the non-failure and determinacy parts. This is enough for having a correct analysis. For the bottom element, $\bot$, the analysis generates new variables for each of the resources and the cardinality. Then, new relations are added between them and the default cost for each resource. For an unknown predicate, the cardinality should be $[0,\infty)$ and it may fail. For example, the bottom element for the "number of resolution steps" resource will be:

$$\langle (s_L, s_U), \{(n_L, n_U)\}, \texttt{true}, \emptyset, \{(s_L, s_U) \lesseqgtr (0,\infty), (n_L, n_U) \lesseqgtr (0,0)\}, \texttt{fails}, \texttt{non\_det} \rangle$$

where `fails` and `non_det` are the bottom elements of their respective domains.

The λ_{call} to β_{entry} Operation

In this operation it needs to be created the initial structures for handling the bounds on cardinality and resources. This implies the generation of fresh variables for each of them, and setting them to their initial values. In the case of cardinality, the initial value is 1 (which is the number of solutions generated by a fact). For a resource r, the initial value is exactly φ_r. We will name new fresh variables by adding an integer subscript. For example, $s_{L,1,1}$ will be the first fresh variable related to the *l*ower bound on *s*olutions on *first* clause.

The addition of constraints over sized types when the head arguments are partially instantiated is inherited from the sized types domain.

Attaching Constraints to Recurrence Relations In this stage, we have introduced a modification to the design of the abstract substitution, and extended the representation of recurrence relations in order to incorporate constraints on the input data to clauses (referred as *tests* in determinacy and non-failure analysis). We take advantage of the information inferred during determinacy analysis, and perform a query to the

PLAI database through a well-defined interface implemented in CiaoPP, passing the *clause id* as argument, and recovering the set of constraints of such clause from determinacy information. Finally, we add such constraints to the *domain* component. These constraints are attached to the relation that represent the cost of a clause, so that it can be used by the solver and obtain solutions for many recurrences for which the previous implementation did not find any solution. As a technical implementation issue, in order to perform such modification, we have changed the PLAI common interface, which has required a lot of effort, given that it is used in may parts of the system.

Finally, the $failed?$ component is initialized with value `false`, as no literal has been executed yet, so it cannot fail.

In the `listfact` example, the entry substitutions are:

$$\beta_{entry,1} = \left\langle \begin{array}{c} (s_{L,1,1}, s_{U,1,1}), \{(n_{L,1,1}, n_{U,1,1})\}, \texttt{false}, \{\alpha_1 = 0, \beta_1 = 0\}, \\ \{(s_{L,1,1}, s_{U,1,1}) \lesseqgtr (1,1), (n_{L,1,1}, n_{U,1,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

$$\beta_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,1}, s_{U,2,1}), \{(n_{L,2,1}, n_{U,2,1})\}, \texttt{false}, \{\alpha_1 > 0, \beta_1 > 0\}, \\ \{(s_{L,2,1}, s_{U,2,1}) \lesseqgtr (1,1), (n_{L,2,1}, n_{U,2,1}) \lesseqgtr (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The Extend Operation

operation, the current abstract substitution needs to be *extended* with the information of the success substitution from the literal call. In this way, several components of the abstract element have to be updated. First of all, a call to the function giving the number of solutions and the resource usage from the called literal has to be added.

Afterwards, new variables need to be generated for the number of solutions and resources, which will hold the bounds for the clause up to that point. New relations must be added to the abstract element to give a value to those new variables:

- For the number of solutions, let $s_{U,c}$ be the new upper bound variable, $s_{U,p}$ the previous variable defining an upper bound on the number of solutions, and $s_{U,\lambda}$ an upper bound on the number of solutions for the subgoal. Then we need to include a constraint: $s_{U,c} \leq s_{U,p} \times s_{U,\lambda}$.

  In the case of lower bound analysis, there are two phases. First of all, it is checked whether the called literal can fail, looking at the output of the non-failure analysis. If it is possible for it to fail, the $failed?$ component is updated to `true`. If after this checking the $failed?$ component is still `false` (meaning that neither this literal nor any of the previous ones may fail) a relation is included, similar to the one for the upper bound case: $s_{L,c} \geq s_{L,p} \times s_{L,\lambda}$. Otherwise, the relation $s_{L,c} \geq 0$ is included, because failing predicates produce no solutions.

- The approach for resources is similar. Let $r_{U,c}$ be the new upper bound variable, $r_{U,p}$ the previous variable defining an upper bound on that resource and $r_{U,\lambda}$ an upper bound on resources from the analysis of the literal. The relation added in this case is $r_{U,c} \leq r_{U,p} + s_{U,p} \times \left(\omega + r_{U,\lambda}\right)$.

  For lower bounds, the $failed?$ component is already updated. Then, if the component is still `false`, a new relation similar to the one for upper bounds is added. If it is `true`, it means that failure may happen at some point, so we do not have to add that resource any more. Thus the relation to be included is $r_{L,c} \geq r_{L,p}$.

In our example, consider the extension of `listfact` after performing the analysis of the `fact` literal, whose resource components of the abstract element will be:

$$\left\langle \begin{array}{c} (s_L, s_U), \{(n_L, n_U)\}, \texttt{false}, \{\alpha, \beta \geq 0\} \\ \{(s_L, s_U) \lessapprox (1,1), (n_L, n_U) \lessapprox (\alpha, \beta)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

This literal is known not to fail, so the value of $failed?$ is not changed in the abstract element for the second clause. That means that it is still `false`, so complete calls are added:

$$\beta_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,2}, s_{U,2,2}), \{(n_{L,2,2}, n_{U,2,2})\}, \texttt{false}, \{\ldots\} \\ \left\{ \begin{array}{c} \ldots, \\ (s_{L,2,2}, s_{U,2,2}) \lessapprox (1 \times s_{L,2,1}, 1 \times s_{U,2,1}), \\ (n_{L,2,2}, n_{U,2,2}) \lessapprox (\gamma_1 + n_{L,2,1}, \delta_1 + n_{U,2,1}) \end{array} \right\}, \\ \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

The β_{exit} to λ' Operation

After all the extend operations, the variables appearing in the number of solutions and resources positions will hold the correct value for their properties. As in the case of sized types, at this point a normalization step takes place, based on [Debray and Lin, 1993]: replace each variable appearing in an expression with its definition in terms of other variables, in reverse topological order. Following this process, we should reach the variables in the sized types of the input parameters in the head.

Going back to `listfact`, the final substitutions are as follows. s'_L, s'_U, n'_L and n'_U refer to number of solutions and resolution steps from the recursive call to `listfact`.

$$\lambda'_1 = \left\langle \begin{array}{c} (s_{L,1,1}, s_{U,1,1}), \{(n_{L,1,1}, n_{U,1,1})\}, \texttt{false}, \{\alpha_1 = 0, \beta_1 = 0\}, \\ \{(s_{L,1,1}, s_{U,1,1}) \lessapprox (1,1), (n_{L,1,1}, n_{U,1,1}) \lessapprox (1,1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

$$\lambda'_{entry,2} = \left\langle \begin{array}{c} (s_{L,2,3}, s_{U,2,3}), \{(n_{L,2,3}, n_{U,2,3})\}, \texttt{false}, \{\alpha_1 > 0, \beta_1 > 0\}, \\ \left\{ \begin{array}{c} s_{L,2,3} \geq 1 \times s'_L(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})), \\ s_{U,2,3} \leq 1 \times s'_U(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})), \\ n_{L,2,3} \geq \gamma_1 + n'_L(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})), \\ n_{U,2,3} \leq \delta_1 + n'_U(ln^{(\alpha_1-1,\beta_1-1)}(n^{(\gamma_1,\delta_1)})) \end{array} \right\}, \\ \texttt{not\_fails,is\_det} \end{array} \right\rangle$$

The Widening Operator ∇ and Closed Forms

As mentioned before, at this point there exists the possibility of having different aggregation operators for different resources. Thus, when the analysis has the equations, it needs to pass them to each of the corresponding Γ_r per each resource r to get the final equations.

This process can be further refined in the case of solution analysis, using the information from the non-failure and determinacy analyses. If the final output of the non-failure analysis is `fails`, we know that the only correct lower bound is 0. So the analysis can just assign the relation $s_L \geq 0$ without further relations. Conversely, if the final output of the determinacy analysis is `is_det`, it can be safely set the relation $s_U \leq 1$, because at most one solution will be produced in each case. Furthermore, we can refine the lower bound on the number of solutions with the minimum between the current bound and 1.

In the example analyzed above there was an implicit assumption while setting up the relations: that the recursive call in the body of `listfact` refers to the same predicate call, so we can set up a recurrence. This fact is implicitly assumed in Hindley-Milner type systems. But in logic programming it is usual for a predicate to be called with different patterns (for example, modes). Fortunately, the CiaoPP framework allows multivariance (support for different call patterns of the same predicate). For the analysis to handle it, it is not enough just add calls with the bare name of the predicate, because it will conflate all the versions. The solution is to add a new component to the abstract element: a random name given to the specific instance of the predicate, and generated in the λ_{call} to β_{entry}. In the widening step, all different versions of the same predicate are conflated.

Even though the analysis works with relations, these are not as useful as functions defined without recursion or calls to other functions. First of all, developers will get a better idea of the sizes presented in such a closed form. Second, functions are amenable to comparison as outlined in [Lopez-Garcia et al., 2010c], which is essential in verification. There are several packages able to get bounds for recurrence equations: computer algebra systems, such as Mathematica (which has been used in our experiments) or Maxima; and specialized solvers such as PURRS [Bagnara et al., 2005] or PUBS [Albert et al., 2011b]. In this implementation, this operation is applied after each widening. For our example, the final abstract substitution is:

$$\lambda_1' \nabla \lambda_2' = \left\langle \begin{array}{c} (s_L, s_U), \{(n_L, n_U)\}, \texttt{false}, \{\alpha_1, \beta_1 \geq 0\}, \\ \{(s_L, s_U) \lesseqgtr (1,1), (n_L, n_U) \lesseqgtr (\alpha_1\gamma_1, \beta_1\delta_1)\}, \texttt{not\_fails}, \texttt{is\_det} \end{array} \right\rangle$$

3.5.3 Improvements in Design and Implementation

We have taken the initial prototype implementation of the analysis performed in [Serrano et al., 2014a] as a starting point, and have applied multiple code refactorizations and design changes on it, in order to achieve a better integration into the CiaoPP system, and also to prepare the analysis for its integration with the modular solver that we have proposed in Chapter 4.

One of the first improvements we have made has been to *decouple* the solver-related functionality from the resource abstract domain, in order to make the later independent of the former. This allowed us to give the option to the user of choosing between two different back-end algebraic solvers. Thus, the user can choose between an off-the-shelf CAS system, and a specialized recurrence solver implemented in Ciao, which we call *built-in solver*. This change has represented an important improvement. On one hand, it was the first step towards the integration of a new, modular solver. On the other hand, the resource analysis itself has resulted in a better modularization, with all the advantages that entails (maintainability, extensibility, readability, etc.).

Another improvement was to implement a better way of communicating with other abstract domains. Instead of including the information of determinacy and non-failure analysis in the abstract substitution of the resource abstract domain, which requires calling the abstract operations of these domains from each abstract operation of the resource domain (as a sort of *Cartesian product*), we now use an interface, provided by CiaoPP, to query the PLAI database and access all the information inferred by the rest of the analysis. We can access such information both at predicate and program-point level. For the later case, we have modified the interface between the abstract domains and the PLAI main fixpoint algorithm in order to access, from the abstract domains, the program-point key. This allows us, for example, to reuse the tests gathered for determinacy analysis at each program point.

Regarding the solving of equations, we have modified the structure of the equations in order to include the constraints (which we can call *applicability conditions*) associated to each equation. In the previous implementation these constraints were used by the analyzer in order to set up the equations in a format suitable for the external CAS. Thus, the CAS received no information regarding applicability conditions. With the new design we have proposed, where each equation/recurrence relation has a set of associated *applicability conditions*, it is now possible to take advantage of that information and apply more and better techniques for solving the equations (such as the ones proposed in Chapter 4). In addition, this brings the possibility of performing

transformations and operations at the equation level, independently of the analysis itself (such as checking mutual exclusion, inferring loop invariants, etc.).

3.6 Conclusions

In this chapter we have given an overview of the resource analysis based on abstract interpretation implemented in the PLAI abstract interpretation framework, which is included in CiaoPP. This analysis has been defined as an abstract domain and uses the sized types that have been developed in [Serrano et al., 2013]. This brings in useful features such as *multivariance* for free. The approach overcomes important limitations of other existing resource analyses and enhances their precision. We have taken the initial prototype implementation of the analysis performed in [Serrano et al., 2014a] as starting point, and have improved both, the design and implementation. In Chapter 4 we take a deeper look at the improvements of the operation for finding closed-form functions for recurrences. The objective of these improvements is a better integration of the analysis with the rest of the system, in order to enhance its effectiveness, practicality, maintainability, and extensibility, and allow an easier combination with other supporting analysis present in the CiaoPP system. This will also eventually allow an evolution of the tool the industrial application level.

Chapter 4

A Modular Solver Framework

This chapter describes the work we have done in this thesis to significantly improve the usefulness and applicability of the resource usage analysis, by defining a proper architecture for the component in charge of mathematical operations. The most important of these operations is the finding of closed-form expressions for recurrence (in-)equations or relations, either exact solutions or over-approximations (upper or lower bounds) of them. Firstly, we give an overview of the problem of finding closed-form expressions for the recurrence relations that arise in resource analysis, and why this is important. Then, we describe how this problem is addressed in the current implementation of the resource analysis within CiaoPP based on abstract interpretation, pointing out its main limitations. Then, we present our proposal for a new component in charge of this part of the analysis, including a modular architecture that brings in many interesting features in terms of maintainability and extensibility. Finally, we present the implementation of a technique we have developed for upper-bounding a special class of recurrence relations that arise usually in the analysis of programs with accumulating parameters.

4.1 The Need for Finding Closed-form Expressions

The traditional approach for static resource usage analysis dates back to the seminal work by Wegbreit [Wegbreit, 1975], which consists of two general steps:

- Given the source code of the program and a cost model for basic operations, the analysis sets up a system of equations that describe the cost of the program in terms of the sizes of the input. In the presence of loops or recursion, these equations have the form of *recurrences*.
- As recurrences are not useful in most situations, the second step is to replace

these recurrences by equivalent *closed-form* functions.

Finding closed-form functions for recurrences is crucial for the usefulness of the results of the analysis. Developers will get a better idea if this result is expressed with a closed-form expression. Second, and most important, closed-form expressions are appropiate for comparison of resource related information, as outlined in [Lopez-Garcia et al., 2010c], which is essential for verification purposes. There are several systems to get exact or over-approximated closed-forms for recurrence relations: Computer Algebra Systems (CAS), such as Mathematica® or Maxima; and specialized solvers such as PURRS [Bagnara et al., 2005] or PUBS [Albert et al., 2011a]. However, none of these systems alone is able to solve or bound all the recurrences that can appear during analysis. Thus, it is very important to identify different classes of recurrence equations and choose the right solver to handle them.

Algorithms for finding closed-form solutions for recurrence relations have been studied in numerous previous work, such as [Ivie, 1978, Petkovšek, 1990, Cohen and Katcoff, 1977]. It is always possible to reduce a system of linear recurrence relations to a single linear recurrence relation in one variable, so it suffices to consider the solution of a single linear recurrence relation in one variable. The algorithms presented in [Ivie, 1978, Petkovšek, 1990, Cohen and Katcoff, 1977] solve linear recurrence relations with constant coefficients using either characteristic equations or generating functions. However, nonlinear recurrences may arise in size or resource analysis, for example from recursive programs with more than one recursive call in the body of a clause, or from a divide-and-conquer program. The solution for this kind of recurrences is generally much more difficult than the solution of linear recurrences. A large class of nonlinear recurrence relations can be transformed into linear equations by variable transformations. There are also many well-known special nonlinear recurrence relations which have known solutions.

In order to automatize the whole resource usage analysis, it is necessary to have a solver capable of finding closed-form solutions for *any* recurrence relation that could arise from program analysis. As we already explained, there is no general method for solving recurrence relations. At a first glance, this might be hopeless. However, we can take some advantages of the specific requirements and features of resource analysis. First of all, since we are interested in over-approximations of resource usage functions, it is not absolutely necessary to find exact solutions for recurrences. It is enough to find closed-form expressions that represent upper or lower bounds on the recurrences, for any input. This can be done by simplifying the equations using transformations in a way that a solution to the transformed equation is guaranteed to be an upper (or lower) bound on the solution to the original equation.

4.2 Finding Closed-form Solutions in the Previous Approach

In the previous implementation of the abstract interpretation-based resource usage analysis of CiaoPP, the operation of finding a closed-form solution for each recurrence relation is performed after every call to the *widening* operation. This process broadly consists in the following steps:

1. The first step sets up the recurrences directly as functions on the size variables, instead of on the size types schemas.
2. The second step consists on removing unnecessary arguments, taking into account how they are used in the equations.
3. The third step uses the constraints contained in the *domain* component of the abstract substitution to finally set up the equations in an appropriate format for the solver. This step strongly depends on which back-end solver is used.
4. The third step simplifies the expressions appearing in the right hand of the equations.
5. Finally, the equations are sent to the back-end solver, and the solutions returned back replace the recurrences in the abstract substitution.

This previous implementation presents many problems and limitations. It is highly dependent on the back-end CAS chosen for solving recurrences. The solver used in the current implementation is Mathematica®, as already mentioned. The function *RSolve*, that tries to solve recurrence equations, does not accept as a valid input, any recurrence relation where there is an argument not used in the right hand side of an equation. For example, the following recurrence relation:

$$\begin{aligned} f(1,D) &= 1, \\ f(X,D) &= f(X-1,D) + D \end{aligned}$$

is not a valid input for *RSolve* because in the first equation the second argument is not used in its right side.

Another important limitation of *RSolve* is that it does not deal with constraints on the variables of equations. For example, the following recurrence relation:

$$\begin{aligned} f(X) &= 1, && \text{if } 0 \leq X \leq 2 \\ f(X) &= f(X-1)+1, && \text{if } X > 2 \end{aligned}$$

needs to be transformed into

$$f(0) = 1, \qquad f(1) = 1, \qquad f(2) = 1,$$
$$f(X) = f(X-1)+1$$

before sending it to the back-end solver.

Another limitation we can mention, is the fact that Mathematica® (or any other CAS) tries to find exact solutions for the recurrences, while over-approximations would be enough for our resource analysis application. Thus, by using the existing CASs, we are limited to exact solutions only. Finally, the most important limitation of this implementation is that currently, there is no CAS or specialized solver capable of solving any of the recurrence equations/relations that might arise in resource analysis. However, in the previous implementation changing the solver was quite difficult.

4.3 Goals of the Modular Solver Framework

Given the above considerations, the main goals of the new solver component are the following:

- We should be able to integrate easily new back-end solvers, such as existing CAS, existing specialized solvers (e.g., PURRS or PUBS), or even new specialized solvers for a limited set of special recurrences.
- We should be able to combine easily the results from different back-end solvers.

Integrating new off-the-shelf systems to solve a larger set of recurrences is crucial for the extensibility and applicability of the whole resource analysis. The ability of combining results from different back-end solvers allows to increase their power, and analyzing (increasingly) more complex programs.

4.4 The Modular Solver Architecture

We propose a new modular architecture for the resource analysis, defining a new component in charge of the algebraic operations of the analysis. In particular, this component will be in charge of finding closed-form functions that over-approximate the recurrences set up during the analysis. This architecture is shown in Figure 4.1. The main modules are:

- *Solver_Strategies*: This module defines the common interface of the different strategies to solve recurrence relations.
- $Strat_i$: These are the different strategies to solve or over-approximate recurrences, using the services of the back-end solvers and the *classifier* in order to identify different characteristics of the recurrences.

- *Rec_Classifier*: It associates a *label* to each input recurrence relation that identifies the class of recurrence.
- *Solver_Utils*: Defines the common interface of the different *back-end solvers*.
- BS_i: These are the modules that implement the interface defined by **Solver_Utils**, connecting directly with the particular back-end solvers, such as Mathematica®, CiaoPP's built-in Solver, etc.

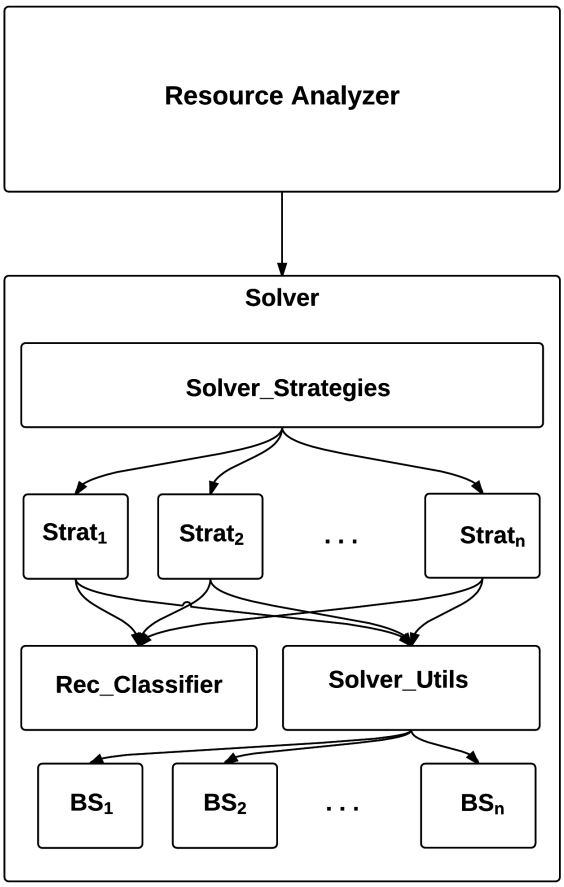

Figure 4.1: Architecture of the Modular Solver Framework.

4.4.1 Algebraic Expression Syntax

In order to use different back-end solvers from our architecture, and, more importantly, combine the results coming from different back-end solvers, it is necessary to define a *common expression syntax* for the inputs and outputs of our solver component, delegating the responsibility of translating this syntax, forth and back to the internal syntax of the back-end solvers, to each particular back-end solver interface. In

Figure 4.2 we show this common syntax, both for arbitrary expressions and recurrence relations, which we call *Algebraic Expression Syntax*.

⟨*expr*⟩ ::= '-'⟨*expr*⟩ | 'fact('⟨*expr*⟩')' | 'fibo('⟨*expr*⟩')' | 'lucasl('⟨*expr*⟩')'
| ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
| 'fun('⟨*name*⟩',['⟨*exprs*⟩'])'
| 'exp('⟨*expr*⟩')' | 'log('⟨*num*⟩ ',' ⟨*expr*⟩')'
| 'max('⟨*expr*⟩','⟨*expr*⟩')' | 'min('⟨*expr*⟩','⟨*expr*⟩')'
| ⟨*num*⟩ | ⟨*var*⟩
| 'sum('⟨*var*⟩',' ⟨*expr*⟩',' ⟨*expr*⟩',' ⟨*expr*⟩')'
| 'prod('⟨*var*⟩',' ⟨*expr*⟩',' ⟨*expr*⟩',' ⟨*expr*⟩')'

⟨*exprs*⟩ ::= ϵ | ⟨*expr*⟩ | ⟨*expr*⟩',' ⟨*exprs*⟩

⟨*binop*⟩ ::= '+' | '-' | '*' | '/' | '**'

⟨*recrels*⟩ ::= ⟨*recrel*⟩ | ⟨*recrel*⟩',' ⟨*recrels*⟩

⟨*recrel*⟩ ::= 'equation('⟨*name*⟩',' ⟨*vars*⟩ ',' ⟨*expr*⟩',' '['⟨*tests*⟩'])'

⟨*tests*⟩ ::= ϵ | ⟨*test*⟩ | ⟨*test*⟩',' ⟨*tests*⟩

⟨*test*⟩ ::= ⟨*expr*⟩ ⟨*bincomp*⟩ ⟨*expr*⟩

⟨*bincomp*⟩ ::= '>' | '<' | '>=' | '=<'

Figure 4.2: Algebraic Expression Syntax

4.4.2 Interface to Back-End Solvers

The common interface that the different *back-end_solvers* have to implement is the following (defined by *Solver_Utils*):

- `translate_from_common_syntax(+Exp, -TExp)`: Translates any expression or recurrence relation received as input, in the *common algebraic expression* syntax, to the specific syntax of the corresponding back-end solver.
- `translate_to_common_syntax(+TExp, -Exp)`: Translates expressions or recurrences from the syntax of the back-end solver to the common algebraic expression syntax.
- `simplify(+Exp, -Simp)`: Simplifies the expression or recurrence relation `Exp`. If the input is a recurrence relation, it simplifies the right hand side of the equations.

- `expand(+Exp, -Simp)`: Expands out products and positive integer powers in `Exp`.
- `greater_than(+A,+B)`,`equal_than(+A,+B)`, `less_than(+A,+B)`,`greater_or_equal_than(+A,+B)`,`less_or_equal_than(+A,+B)`: Implements the comparison operators $>, <, =, >=$ and $=<$, respectively.
- `solve_rec_rel(+RecRel,+Bound,-ClosedForm)`: Tries to find an exact closed-form solution for `RecRel`, or a correct approximation with respect to `Bound`, whose value could be `upper`, if we want an upper bound for the recurrence, or `lower` if we want a lower bound.

The translations forth and back to the common expression syntax are performed from `Solver_Utils`, by calling the corresponding predicates of the back-end solvers. Thus, from the point of view of an user of the solver, the solver receives and returns elements in the common syntax, abstracting away all the translation details.

4.5 Using Ranking Functions for Upper-bounding Special Recurrences

In order to show the advantages of the new architecture proposed for dealing with recurrences, we present in this section how we can implement and integrate an specialized back-end solver for dealing with a special class of logic programs, that commonly appear when the logic program is a translation, more or less direct, of an imperative program with simple loops.

To illustrate the problem we are going to address, we will use a very naive running example. Let us consider an imperative function that calculates the addition of the first N natural numbers.

```
1  int nats(int N){
2    int add = 0;
3    for(int i=1;i=<N;i++)
4       add += i;
5    return add;
6  }
```

In order to analyze this program with our resource analysis, an automatic translation into a Horn clause representation is performed. Assume that this step has already been performed, and the loop for this function is translated into the following logic program (Horn clauses):

```
1 nats(I, N, I):-
2     I >= N.
3 nats(I, N, S):-
4     I < N,
5     I1 is I + 1,
6     nats(I1, N, S1),
7     S is S1 + I.
```

Assuming *execution steps* as the resource to measure, our resource analysis based on abstract interpretation (presented in Chapter 3) will set up following recurrence relation representing an upper bound on the resource usage of a call to that program (predicate):

```
1 equation(g, [I,N], fun(g,[I + 1,N])+1,[I < N]),
2 equation(g, [I,N], 1                  ,[I >= N]),
```

As we can see, this recurrence relation has not a proper form to be handled by a CAS. Mathematica®, for example, can not deal with it properly because it does not support the constraints associated to each equation, and moreover, the second argument only appears in constraints. However, we can transform this recurrence into a common, one variable recurrence equation by performing some variable substitution and using the information present in the constraints. Let us define $Y = N - I + 1$. If we rewrite the previous recurrence in terms of Y, we obtain:

```
1 equation(g, [Y], fun(g,[Y - 1])+1,[Y > 1]),
2 equation(g, [Y], 1                ,[Y =< 1]),
```

which can be easily solved now by almost any recurrence solver. Its solution, Y, needs to be replaced by its definition, $N - I + 1$. This is a valid upper-bound for the recurrence if $I \leq N$. If $I > N$ the result should be C, because the base case condition applies. For that reason, as a last step, we need to use the max operator in order to obtain a valid upper bound for all the possible values of I and N. Therefore, we obtain the following closed-form upper-bound expression for g:

$$max(C, N - I + 1) \tag{4.1}$$

We just have shown how to solve, in a very intuitive way, a tricky but simple kind of recurrence relation. As programs with this characteristics are very common, it is important to automatize this technique. If we pay attention to the reasoning we just followed, we can notice that we looked for a variable replacement that transform the

recurrence in a form amenable for traditional solvers. That means we need the recurrence in a decremental way, expressing the n-th term as a function of some i-th term, obtaining $i < n$. Therefore, what we need for the variable replacement is an expression that we know *decrease* in each step of the recurrence. The automatic termination community have studied in depth this particular kind of expressions for *loops* (we can see a recurrence as a loop), because they are instrumental for proving termination. They usually prove that there exists a function that maps the arguments of the loop to a *well-founded* partial order, such that this function decreases in each step. This kind of functions are called *ranking functions* [Floyd, 1967]. In [Podelski and Rybalchenko, 2004] a complete method for obtaining linear ranking functions is presented.

The use of ranking functions for finding closed-form upper bounds for recursive resource usage expressions (called *cost relations*) was presented in [Albert et al., 2011a]. In such work, the authors first define a set of *evaluation trees* for a system of cost equations and a given initial call. Then, they try to find an upper bound on the number of nodes, both internal nodes and leaves. Each of this kind of node have a *local cost* that need to be multiplied by the corresponding bound on the number of nodes in order to obtain a closed-form expression. These local costs are also over-approximated by using linear programming techniques, and the result is finally obtained. This idea of over-approximating the number of nodes of any possible evaluation tree was previously given in [Debray et al., 1997], but for inferring both lower and upper bounds for *divide-and-conquer* predicates.

In our case, we are going to limit this technique to simple cases, as the one that we showed above. By simple, we mean recurrences with one single recursive call, possibly multiplied by a constant coefficient, and no other equation call. Even more, we are going to support recurrences (under this technique) with mutually exclusive equations, and only one base case. As we are going to use the algorithm proposed in [Podelski and Rybalchenko, 2004], we also need to limit ourselves to linear arithmetic expressions as input.

In this thesis we follow an approach similar to the ones in [Albert et al., 2011a, Debray et al., 1997].

Definition 10 (Simple recurrence relation)**.** *We call simple recurrence relation to a recurrence of the form:*

$$\begin{aligned} f(\overline{u}) &= C, && \text{if } \varphi(\overline{u}) \\ f(\overline{u}) &= K * f(\overline{u}') + g(\overline{u}), && \text{if } \varphi'(\overline{u}) \end{aligned}$$

where

- $C, K \in \mathbb{Z}^+$ *are constants,*
- g *is an expression that does not contain any call to* f *or any other functions except built-ins.*

- $\varphi(\overline{u}) \wedge \varphi'(\overline{u})$ *is unsatisfiable (mutual exclusion),*
- $\overline{u}$ *is a sequence of arguments, and* $\overline{u}'$ *is a sequence of linear arithmetic expressions over* $\overline{u}$*, such that* $|\overline{u}| = |\overline{u}'|$

Definition 11 (Ranking function for a simple recurrence relation)**.** *A ranking function for the following simple recurrence relation*

$$\begin{aligned} f(\overline{u}) &= C, && \text{if } \varphi(\overline{u}) \\ f(\overline{u}) &= K * f(\overline{u}') + g(\overline{u}), && \text{if } \varphi'(\overline{u}) \end{aligned}$$

is a function $h : \mathbb{Z}^{|\overline{u}|} \to \mathbb{Z}^+$ *such that* $\varphi'(\overline{u}) \models h(\overline{u}) > h(\overline{u}')$.

Now that we have already defined the class of recurrences we are going to handle, and what a ranking function is for those recurrences, we can show how to obtain a safe closed-form over-approximation for them. Given an initial input $\overline{u}_0$, let us first observe what is the result of applying the recursive case several times, before reaching the base case:

$$\begin{aligned} f(\overline{u}_0) = \\ Kf(\overline{u}_1) + g(\overline{u}_0) = \\ K(Kf(\overline{u}_2) + g(\overline{u}_1)) + g(\overline{u}_0) = K^2 f(\overline{u}_2) + Kg(\overline{u}_1) + g(\overline{u}_0) = \\ \cdots \\ K^{i-1} f(\overline{u}_{i-1}) + K^{i-2} g(\overline{u}_{i-2}) + \cdots + g(\overline{u}_0) = \\ K^i C + K^{i-1} g(\overline{u}_{i-1}) + \cdots + g(\overline{u}_0) \end{aligned}$$

As we can see, the last expression, where the base case is finally reached, is in closed-form, with i being the number of applications of the recursive case. We can replace i by a ranking function $h(\overline{u})$ for f, because it is also an upper bound of the number of times a recursive case is applied (see for example [Albert et al., 2011a]). We also need to find a general form for all the expressions $g(\overline{u}_j)$. In order to do that, and to ensure we are going to obtain a correct upper-bound, we can replace each of such subexpressions by the result of maximizing $g(\overline{u})$ under the constraints $\varphi'(\overline{u})$. Let M be the result of this operation. Finally, as we explained before, it is necessary to *wrap* the resulting expression with the max operator and the base case constant C as argument. In conclusion, we obtain the following closed-form expression:

$$\hat{f}(\overline{u}) = max(C, (\sum_{i=0}^{h(\overline{u})-1} K^i M) + K^{h(\overline{u})} C) \geq f(\overline{u}) \tag{4.2}$$

Comming back to our running example, we can see that the recurrence is a simple recurrence relation with $K = 1, C = 1$ and $\forall \overline{u} : g(\overline{u}) = 1$. If we apply the algorithm

in [Podelski and Rybalchenko, 2004] (interpreting the recurrence as a single loop), we obtain the ranking function $h(I,N) = N - I + 1$. Finally, by applying the formula we have just derived, we obtain the following upper-bound closed-form expression:

$$max(1,(N-I+1)*1+1 = max(1,N-I+2) \tag{4.3}$$

which is a correct upper-bound and is consistent with our first intuition (equation 4.1).

4.6 Implementation

We have implemented a prototype of the proposed architecture taking advantage of the module system of Ciao. We have also performed some *refactorizations* on the abstract interpretation-based resource analysis in order to integrate our prototype of the solver into it. In this prototype, we have developed a *strategy* called *chain*, as a proof of concept. This strategy simply tries to solve a recurrence relation by calling in sequence each available back-end solver. The first solution found is the one that is returned, obtained from one of the back-end solvers. A simple but significant improvement of this strategy would be to compare all of these results and get the maximum or minimum of them, depending on which kind of approximation we are looking for.

The implementation of the method showed in this section mainly requires a procedure for finding (linear) ranking functions. The Parma Polyhedra Library (PPL) provides numerical abstractions especially targeted at applications in the field of analysis and verification of complex systems, including implementations of methods for the synthesis of linear ranking functions [Bagnara et al., 2002, Bagnara et al., 2010]. Thus, we have integrated PPL as a back-end solver in our proposed architecture, implementing the operation `solve_rec_rel` in such a way that it: (a) identifies the different parts of the recurrence relation; (b) obtains a ranking function for the given recurrence; and (c) sets up the closed-form expression of Equation (4.2), using the ranking function obtained in (b) and the components of the recurrence obtained in (a). We also require an operation that performs the maximization of an expression under a set of constraints. Fortunately, we can use any CAS (in our case Mathematica®), or even PPL to perform such operation.

4.7 Conclusions and Future Work

In this chapter we have presented a new modular architecture for performing all of the algebraic operations required by the resource usage analysis. The most important one among these operations is the obtention of closed-form solutions or over-approximations of recurrence relations. This architecture is based on the integration of many back-end solvers, such as CAS and specialized solvers, and in the definition of

a common algebraic expression syntax, allowing the combination of results from different back-end solvers. As for the combination of the results, the architecture allows to easily incorporate different *strategies*, i.e., modules with an algorithm for solving recurrences, but in a higher level, using the services provided by the set of back-end solvers.

In order to show the advantages of the new architecture proposed for dealing with recurrences, we have presented a technique for solving a special class of recurrence relations that can appear during the analysis of some programs. This technique is based on finding a closed-form ranking function for the recurrence relation. In turn, the technique presented in this case is also a contribution of this thesis.

As future work, we can mention the integration of more specialized back-end solvers, such as PURRS or PUBS, in order to take advantage of the architecture and improve the applicability of the resource usage analysis. We also define new and smarter strategies for solving recurrence relations.

Chapter 5

Applying the Improved Analysis to Energy Consumption Verification

In this chapter we describe how we have applied our improved resource analysis tool for *verifying* energy consumption specifications of imperative embedded programs. We first have integrated it into the CiaoPP resource verification framework, so that the framework can access the information inferred by our improved resource analysis, and use it for comparison with specifications. Then, we have instantiated the resulting verification framework for *verifying* specifications about the energy consumed by embedded imperative programs.

Such specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The bounds of the intervals can be given in general as functions on input data sizes. Our verification system can prove whether such energy usage specifications are met or not. It can also infer the particular conditions under which the specifications hold. To this end, these conditions are also expressed as intervals of functions of input data sizes, such that a given specification can be proved for some intervals but disproved for others. The specifications themselves can also include preconditions expressing intervals for input data sizes.

Finally, we report on the prototype implementation of our approach that we have performed within the CiaoPP system for an industrial case study, the XC language and XS1-L architecture, and illustrate with a real example how embedded software developers can use this tool, in different scenarios, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service. This work has been published in [Lopez-Garcia et al., 2015].

5.1 Introduction

In an increasing number of applications, particularly those running on devices with limited resources, it is very important and sometimes essential to ensure conformance with respect to specifications expressing non-functional global properties such as energy consumption, maximum execution time, memory usage, or user-defined resources. For example, in a real-time application, a program completing an action later than required is as erroneous as a program not computing the correct answer. The same applies to an embedded application in a battery-operated device (e.g., a portable or implantable medical device, an autonomous space vehicle, or even a mobile phone) if the application makes the device run out of batteries earlier than required, making the whole system useless in practice.

In general, high performance embedded systems must control, react to, and survive in a given environment, and this in turn establishes constraints about the system's performance parameters including energy consumption and reaction times. Therefore, a mechanism is necessary in these systems in order to prove correctness with respect to specifications about such non-functional global properties.

To address this problem we leverage our improved resource usage analysis framework, integrate it with the verification framework of CiaoPP [Lopez-Garcia et al., 2010b, Lopez-Garcia et al., 2012], and specialize it for *verifying* energy consumption specifications of embedded programs. As a case study, we focus on the energy verification of embedded programs written in the XC language [Watt, 2009] and running on the XMOS XS1-L architecture (XC is a high-level C-based programming language that includes extensions for communication, input/output operations, real-time behavior, and concurrency). However, the approach presented here can also be applied to the analysis of other programming languages and architectures. We will illustrate with an example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

5.2 Overview of the Energy Verification Tool

In this section we give an overview of the prototype tool for *energy consumption verification* of XC programs running on the XMOS XS1-L architecture, which we have implemented within the CiaoPP system [Hermenegildo et al., 2005a]. As in previous work [Liqat et al., 2014b, Méndez-Lojo et al., 2007], we differentiate between the *input language*, which can be XC source, LLVM IR [Lattner and Adve, 2004], or Instruction Set Architecture (ISA) code, and the *intermediate semantic program representation* that the CiaoPP core components (e.g., the analyzer) take as input. The latter is a series of connected code blocks, represented by Horn Clauses, that we will refer to as "HC IR" from

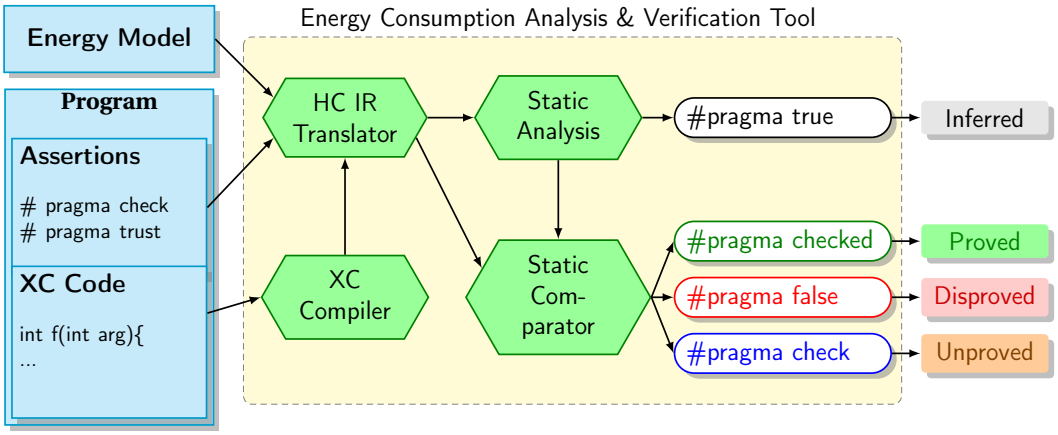

Figure 5.1: Energy consumption verification tool using CiaoPP.

.

now on. We perform a transformation from each *input language* into the HC IR and pass it to the corresponding CiaoPP component. The main reason for choosing Horn Clauses as the intermediate representation is that it offers a good number of features that make it very convenient for the analysis [Méndez-Lojo et al., 2007]. For instance, it supports naturally Static Single Assignment (SSA) and recursive forms, as will be explained later. In fact, there is a current trend favoring the use of Horn Clause programs as intermediate representations in analysis and verification tools [Bjørner et al., 2014].

Figure 5.1 shows an overview diagram of the architecture of the prototype tool we have developed. Hexagons represent different tool components and arrows indicate the communication paths among them.

The tool takes as input an XC source program (left part of Figure 5.1) that can optionally contain assertions in a C-style syntax. As we will see later, such assertions are translated into Ciao assertions, the internal representation used in the Ciao/CiaoPP system.

The energy specifications that the tool will try to prove or disprove are expressed by means of assertions with `check` *status*. These specifications can include both lower and upper bounds on energy usage, and they can express intervals within which energy usage is to be certified to be within such bounds. The bounds of the intervals can be given in general as functions on input data sizes. Our tool can prove whether such energy usage specifications are met or not. It can also infer the particular conditions under which the specifications hold. To this end, these conditions are also expressed as intervals of functions of input data sizes, such that a given specification can be proved for some intervals but disproved for others.

In addition, assertions can also express trusted information such as the energy usage of procedures that are not developed yet, or useful hints and information to the

tool. In general, assertions with status `trust` can be used to provide information about the program and its constituent parts (e.g., individual instructions or whole procedures or functions) to be trusted by the analysis system, i.e., they provide base information assumed to be true by the inference mechanism of the analysis in order to propagate it throughout the program and obtain information for the rest of its constituent parts.

In our tool the user can choose between performing the analysis at the ISA or LLVM IR levels (or both). We refer the reader to [Liqat et al., 2014a] for an experimental study that sheds light on the trade-offs implied by performing the analysis at each of these two levels, which can help the user to choose the level that fits the problem best.

The associated ISA and/or LLVM IR representations of the XC program are generated using the xcc compiler. Such representations include useful metadata. The *HC IR translator* component (described in Section 5.4) produces the internal representation used by the tool, HC IR, which includes the program and possibly specifications and/or trusted information (expressed in the Ciao assertion language [Puebla et al., 2000, Hermenegildo et al., 2012b]).

The tool performs several tasks:

1. Transforming the ISA and/or LLVM IR into HC IR. Such transformation preserves the resource consumption semantics, in the sense that the resource usage information inferred by the tool is applicable to the original XC program.

2. Transforming specifications (and trusted information) written as C-like assertions into the Ciao assertion language.

3. Transforming the energy model at the ISA level [Kerrison and Eder, 2015], expressed in JSON format, into the Ciao assertion language. Such assertions express the energy consumed by individual ISA instruction representations, information which is required by the analyzer in order to propagate it during the static analysis of a program through code segments, conditionals, loops, recursions, etc., in order to infer analysis information (energy consumption functions) for higher-level entities such as procedures, functions, or loops in the program.

4. In the case that the analysis is performed at the LLVM IR level, the *HC IR translator* component produces a set of Ciao assertions expressing the energy consumption corresponding to LLVM IR block representations in HC IR. Such information is produced from a mapping of LLVM IR instructions with sequences of ISA instructions and the ISA-level energy model. The mapping information is produced by the *mapping tool* that was first outlined in [Lopez-Garcia, 2014] (Section 2 and Attachments D3.2.4 and D3.2.5) and is described in detail in [Georgiou et al., 2014].

Then, our improved resource analyzer (presented in Chapter 3), specialized for energy consumption based on the approach described in [Liqat et al., 2014b], takes the HC IR, together with the assertions which express the energy consumed by LLVM IR blocks and/or individual ISA instructions, and possibly some additional (trusted) information, and processes them, producing the analysis results, which are expressed also using Ciao assertions. Such results include energy usage functions (which depend on input data sizes) for each block in the HC IR (i.e., for the whole program and for all the procedures and functions in it.). The analysis can infer different types of energy functions (e.g., polynomial, exponential, or logarithmic). The procedural interpretation of the HC IR programs, coupled with the resource-related information contained in the (Ciao) assertions, together allow the resource analysis to infer static bounds on the energy consumption of the HC IR programs that are applicable to the original LLVM IR and, hence, to their corresponding XC programs. Analysis results are given using the assertion language, to ensure interoperability and make them understandable by the programmer.

The verification of energy specifications is performed by a specialized component which compares the energy specifications with the (safe) approximated information inferred by the static resource analysis. Such component is based on our previous work on general resource usage verification presented in [Lopez-Garcia et al., 2010b, Lopez-Garcia et al., 2012], where we extended the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes. We also defined an abstract semantics for resource usage properties and operations to compare the (approximated) intended semantics of a program (i.e., the specification) with approximated semantics inferred by static analysis. These operations include the comparison of arithmetic functions (e.g., polynomial, exponential, or logarithmic functions) that may come from the specifications or from the analysis results. As a possible result of the comparison in the output of the tool, either:

1. The original (specification) assertion (i.e., with status `check`) is included with status `checked` (resp. `false`), meaning that the assertion is correct (resp. incorrect) for all input data meeting the precondition of the assertion,
2. the assertion is "split" into two or three assertions with different status (`checked`, `false`, or `check`) whose preconditions include a conjunct expressing that the size of the input data belongs to the interval(s) for which the assertion is correct (status `checked`), incorrect (status `false`), or the tool is not able to determine whether the assertion is correct or incorrect (status `check`), or
3. in the worst case, the assertion is included with status `check`, meaning that the tool is not able to prove nor to disprove (any part of) it.

If all assertions are `checked` then the program is *verified*. Otherwise, for assertions

(or parts of them) that get `false` status, a *compile-time error* is reported. Even if a program contains no assertions, it can be checked against the assertions contained in the libraries used by the program, potentially catching bugs at compile time. Finally, and most importantly, for assertion (or parts of them) left with status `check`, the tool can optionally produce a *verification warning* (also referred to as an "alarm"). In addition, optional run-time checks can also be generated.

5.3 The Assertion Language

Two aspects of the assertion language are described here: the front-end language in which assertions are written and included in the XC programs to be verified, and the internal language in which such assertions are translated into and passed, together with the HC IR program representation, to the core analysis and verification tools, the Ciao assertion language.

5.3.1 The Ciao Assertion Language

We describe here the subset of the Ciao assertion language which allows expressing global "computational" properties and, in particular, resource usage. We refer the reader to [Puebla et al., 2000, Hermenegildo et al., 2005a, Hermenegildo et al., 2012b] and their references for a full description of this assertion language.

For brevity, we only introduce here the class of `pred` **assertions**, which describes a particular predicate and, in general, follows the schema:

`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-Props*] `.`

where *Pred* is a predicate symbol applied to distinct free variables while *Precond* and *Postcond* are logic formulae about execution states. An execution state is defined by variable/value bindings in a given execution step. The assertion indicates that in any call to *Pred*, if *Precond* holds in the calling state and the computation of the call succeeds, then *Postcond* also holds in the success state. Finally, the *Comp-Props* field is used to describe properties of the whole computation for calls to predicate *Pred* that meet *Precond*. In our application *Comp-Props* are precisely the resource usage properties.

For example, the following assertion for a typical `append/3` predicate:

```
1 :- pred append(A,B,C)
2    : (list(A,num),list(B,num),var(C))
3   => (list(C,num),
4       rsize(A,list(ALb,AUb,num(ANl,ANu))),
```

```
5       rsize(B,list(BLb,BUb,num(BNl,BNu))),
6       rsize(C,
7        list(ALb+BLb,AUb+BUb,
8          num(min(ANl,BNl),max(ANu,BNu)))))
9    + resource(steps,ALb+1, AUb+1).
```

states that for any call to predicate `append/3` with the first and second arguments bound to lists of numbers, and the third one unbound, if the call succeeds, then the third argument will also be bound to a list of numbers. It also states that an upper bound on the number of resolution (execution) steps required to execute any of such calls is $AUb+1$, a function on the length of list A. The `rsize` terms are the *sized types* derived from the regular types, containing variables that represent explicitly lower and upper bounds on the size of terms and subterms appearing in arguments. See Chapter 3 for an overview of the general resource analysis framework and how sized types are used.

The global non-functional property `resource/3` (appearing in the "+" field), is used for expressing resource usages and follows the schema:

resource(*Res_Name***,** *Low_Arith_Expr, Upp_Arith_Expr)*

where *Res_Name* is a user-provided identifier for the resource the assertion refers to, *Low_Arith_Expr* and *Upp_Arith_Expr* are arithmetic functions that map input data sizes to resource usage, representing respectively lower and upper bounds on the resource consumption.

Each assertion can be in a particular *status*, marked with the following prefixes, placed just before the `pred` keyword: `check` (indicating the assertion needs to be checked), `checked` (it has been checked and proved correct by the system), `false` (it has been checked and proved incorrect by the system; a compile-time error is reported in this case), `trust` (it provides information coming from the programmer and needs to be trusted), or `true` (it is the result of static analysis and thus correct, i.e., safely approximated). The default status (i.e., if no status appears before `pred`) is `check`.

5.3.2 The XC Assertion Language

The assertions within XC files use instead a different syntax that is closer to the standard C notation and friendlier for C developers. These assertions are transparently translated into Ciao assertions when XC files are loaded into the tool. The Ciao assertions output by the analysis are also translated back into XC assertions and added inline to a copy of the original XC file.

More concretely, the syntax of the XC assertions accepted by our tool is given by the following grammar, where the non-terminal ⟨*identifier*⟩ stands for a standard C iden-

tifier, ⟨*integer*⟩ stands for a standard C integer, and the non-terminal ⟨*ground-expr*⟩ for a ground expression, i.e., an expression of type ⟨*expr*⟩ that does not contain any C identifiers that appear in the assertion scope (the non-terminal ⟨*scope*⟩).

⟨*assertion*⟩ ::= '`#pragma`' ⟨*status*⟩ ⟨*scope*⟩ '`:`' ⟨*body*⟩

⟨*status*⟩ ::= '`check`' | '`trust`' | '`true`' | '`checked`' | '`false`'

⟨*scope*⟩ ::= ⟨*identifier*⟩ '`(`' '`)`'
| ⟨*identifier*⟩ '`(`' ⟨*arguments*⟩ '`)`'

⟨*arguments*⟩ ::= ⟨*identifier*⟩ | ⟨*arguments*⟩ '`,`' ⟨*identifier*⟩

⟨*body*⟩ ::= ⟨*precond*⟩ '`==>`' ⟨*cost-bounds*⟩ | ⟨*cost-bounds*⟩

⟨*precond*⟩ ::= ⟨*upper-cond*⟩ | ⟨*lower-cond*⟩
| ⟨*lower-cond*⟩ '`&&`' ⟨*upper-cond*⟩

⟨*lower-cond*⟩ ::= ⟨*ground-expr*⟩ '`<=`' ⟨*identifier*⟩

⟨*upper-cond*⟩ ::= ⟨*identifier*⟩ '`<=`' ⟨*ground-expr*⟩

⟨*cost-bounds*⟩ ::= ⟨*lower-bound*⟩ | ⟨*upper-bound*⟩
| ⟨*lower-bound*⟩ '`&&`' ⟨*upper-bound*⟩

⟨*lower-bound*⟩ := ⟨*expr*⟩ '`<=`' '`energy`'

⟨*upper-bound*⟩ := '`energy`' '`<=`' ⟨*expr*⟩

⟨*expr*⟩ := ⟨*expr*⟩ '`+`' ⟨*mult-expr*⟩
| ⟨*expr*⟩ '`-`' ⟨*mult-expr*⟩

⟨*mult-expr*⟩ := ⟨*mult-expr*⟩ '`*`' ⟨*unary-expr*⟩
| ⟨*mult-expr*⟩ '`/`' ⟨*unary-expr*⟩

⟨*unary-expr*⟩ := ⟨*identifier*⟩
| ⟨*integer*⟩
| '`sum`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`prod`' '`(`' ⟨*identifier*⟩ '`,`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩'`,`' ⟨*expr*⟩ '`)`'
| '`power`' '`(`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`log`' '`(`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ '`)`'
| '`(`' ⟨*expr*⟩ '`)`'
| '`+`' ⟨*unary-expr*⟩
| '`-`' ⟨*unary-expr*⟩
| '`min`' '`(`' ⟨*identifier*⟩ '`)`'
| '`max`' '`(`' ⟨*identifier*⟩ '`)`'

XC assertions are directives starting with the token `#pragma` followed by the assertion *status*, the assertion *scope*, and the assertion *body*. The assertion *status* can take several values, including `check`, `checked`, `false`, `trust` or `true`, with the same meaning as in the Ciao assertions. Again, the default status is `check`.

The assertion scope identifies the function the assertion is referring to, and provides the local names for the arguments of the function to be used in the body of the assertion. For instance, the scope `biquadCascade(state, xn, N)` refers to the function `biquadCascade` and binds the arguments within the body of the assertion to the respective identifiers `state`, `xn`, `N`. While the arguments do not need to be named in a consistent way w.r.t. the function definition, it is highly recommended for the sake of clarity. The *body* of the assertion expresses bounds on the energy consumed by the function and optionally contains preconditions (the left-hand side of the `==>` arrow) that constrain the argument sizes.

Within the body, expressions of type ⟨*expr*⟩ are built from standard integer arithmetic functions (i.e., `+`, `-`, `*`, `/`) plus the following extra functions:

- `power(base, exp)` is the exponentiation of `base` by `exp`;
- `log(base, expr)` is the logarithm of `expr` in base `base`;
- `sum(id, lower, upper, expr)` is the summation of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `prod(id, lower, upper, expr)` is the product of the sequence of the values of `expr` for `id` ranging from `lower` to `upper`;
- `min(arr)` is the minimal value of the array `arr`;
- `max(arr)` is the maximal value of the array `arr`.

Note that the argument of `min` and `max` must be an identifier appearing in the assertion scope that corresponds to an array of integers (of arbitrary dimension).

5.4 ISA/LLVM IR to HC IR Transformation

In this section we describe briefly the HC IR representation and the transformations into it that we have developed in order to achieve the verification tool presented in Section 5.2 and depicted in Figure 5.1. The transformation of ISA code into HC IR was described in [Liqat et al., 2014b]. We provide herein an overview of the LLVM IR to HC IR transformation.

The HC IR representation consists of a sequence of *blocks* where each block is represented as a *Horn clause*:

$$< block\_id > (< params >) :- \ S_1, \ldots, S_n.$$

Each block has an entry point, that we call the *head* of the block (to the left of the $:-$ symbol), with a number of parameters $< params >$, and a sequence of steps (the *body*, to the right of the $:-$ symbol). Each of these S_i steps (or *literals*) is either (the representation of) an LLVM IR *instruction*, or a *call* to another (or the same) block. The analyzer deals with the HC IR always in the same way, independent of its origin.

LLVM IR programs are expressed using typed assembly-like instructions. Each function is in SSA form, represented as a sequence of basic blocks. Each basic block is a sequence of LLVM IR instructions that are guaranteed to be executed in the same order. Each block ends in either a branching or a return instruction. In order to represent each of the basic blocks of the LLVM IR in the HC IR, we follow a similar approach as in the ISA-level transformation [Liqat et al., 2014b]. However, the LLVM IR includes an additional type transformation as well as better memory modelling. It is explained in detail in [Liqat et al., 2014a]. The main aspects of this process, are the following:

1. Infer input/output parameters to each block.
2. Transform LLVM IR types into HC IR types.
3. Represent each LLVM IR block as a HC IR block and each instruction in the LLVM IR block as a literal (S_i).
4. Resolve branching to multiple blocks by creating clauses with the same signature (i.e., the same name and arguments in the head), where each clause denotes one of the blocks the branch may jump to.

The translator component is also in charge of translating the XC assertions to Ciao assertions and back. Assuming the Ciao type of the input and output of the function is known, the translation of assertions from Ciao to XC (and back) is relatively straightforward. The *Pred* field of the Ciao assertion is obtained from the scope of the XC assertion to which an extra argument is added representing the output of the function. The *Precond* fields are produced directly from the type of the input arguments: to each input variable, its regular type and its regular type size are added to the precondition, while the added output argument is declared as a free variable. Finally the *Comp-Props* field is set to the usage of the resource `energy`, i.e., a literal of the form `resource(energy, Lower, Upper)` where `Lower` and `Upper` are the lower and upper bounds from the energy consumption specification.

5.5 The General Resource Usage Verification Framework

In this section we describe the general framework for (static) resource usage *verification* [Lopez-Garcia et al., 2010b, Lopez-Garcia et al., 2012] that we have specialized for verifying energy consumption specifications of XC programs.

The framework, that we introduced in [Lopez-Garcia et al., 2010b], extends the criteria of correctness as the conformance of a program to a specification expressing non-functional global properties, such as upper and lower bounds on execution time, memory, energy, or user defined resources, given as functions on input data sizes.

Both program verification and debugging compare the *actual semantics* $[\![P]\!]$ of a program P with an *intended semantics* for the same program, which we will denote by I. This intended semantics embodies the user's requirements, i.e., it is an expression of the user's expectations. In the framework, both semantics are given in the form of (*safe*) approximations. The abstract (safe) approximation $[\![P]\!]_\alpha$ of the concrete semantics $[\![P]\!]$ of the program is actually computed by (abstract interpretation-based) *static analyses*, and compared directly to the (also approximate) specification, which is safely assumed to be also given as an abstract value I_α. Such approximated specification is expressed by *assertions* in the program. Program verification is then performed by comparing I_α and $[\![P]\!]_\alpha$.

In this chapter, we assume that the program P is in HC IR form (i.e., a logic program), which is the result of the transformation of the ISA or LLVM IR code corresponding to an XC program. As already said, such transformation preserves the resource consumption semantics, in the sense that the resource usage information inferred by the static analysis (and hence the result of the verification process) is applicable to the original XC program.

Resource usage semantics Given a program p, let $\mathscr{C}_p$ be the set of all calls to p. The concrete resource usage semantics of a program p, for a particular resource of interest, $[\![P]\!]$, is a set of pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of data (either simple data such as numbers, or compound data structures), $p(\bar{t}) \in \mathscr{C}_p$ is a call to procedure[1] p with actual parameters $\bar{t}$, and r is a number expressing the amount of resource usage of the computation of the call $p(\bar{t})$. The concrete resource usage semantics can be defined as a function $[\![P]\!] : \mathscr{C}_p \mapsto \mathbb{R}$ where $\mathbb{R}$ is the set of real numbers (note that depending on the type of resource we can take other set of numbers, e.g., the set of natural numbers).

The abstract resource usage semantics is a set of 4-tuples:

$$(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$$

where $p(\bar{v}) : c(\bar{v})$ is an abstraction of a set of calls. $\bar{v}$ is a tuple of variables and $c(\bar{v})$ is an abstraction representing a set of tuples of data which are instances of $\bar{v}$. $c(\bar{v})$ is an element of some abstract domain expressing instantiation states. Φ is an abstraction of the resource usage of the calls represented by $p(\bar{v}) : c(\bar{v})$. We refer to it as a *resource usage interval function* for p, defined as follows:

- A *resource usage bound function* for p is a monotonic arithmetic function, $\Psi : S \mapsto \mathbb{R}_\infty$, for a given subset $S \subseteq \mathbb{R}^k$, where $\mathbb{R}$ is the set of real numbers, k is the

[1] Also called *predicate* in the HC IR.

number of input arguments to procedure p and $\mathbb{R}_\infty$ is the set of real numbers augmented with the special symbols ∞ and $-\infty$. We use such functions to express lower and upper bounds on the resource usage of procedure p depending on input data sizes.

- A *resource usage interval function* for p is an arithmetic function, $\Phi : S \mapsto \mathbb{I}_\mathbb{R}$, where S is defined as before and $\mathbb{I}_\mathbb{R}$ is the set of intervals of real numbers, such that $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ for all $\bar{n} \in S$, where $\Phi^l(\bar{n})$ and $\Phi^u(\bar{n})$ are *resource usage bound functions* that denote the lower and upper endpoints of the interval $\Phi(\bar{n})$ respectively for the tuple of input data sizes $\bar{n}$. Although $\bar{n}$ is typically a tuple of natural numbers, we do not want to restrict our framework. We require that Φ be well defined so that $\forall \bar{n}\ (\Phi^l(\bar{n}) \leq \Phi^u(\bar{n}))$.

$input_p$ is a function that takes a tuple of data $\bar{t}$ and returns a tuple with the input arguments to p. This function can be inferred by using the existing mode analysis or be given by the user by means of assertions. $size_p(\bar{t})$ is a function that takes a tuple of terms $\bar{t}$ and returns a tuple with the sizes of those data under the size measure used by our improved analysis described in Chapter 3).

In order to make the presentation simpler, we will omit the $input_p$ and $size_p$ functions in abstract tuples, with the understanding that they are present in all such tuples.

Intended meaning The intended approximated meaning I_α of a program is an abstract semantic object with the same kind of tuples: $(p(\bar{v}) : c(\bar{v}), \Phi, input_p, size_p)$, represented by using Ciao assertions (which are part of the HC IR) of the form:

`:- check` *Pred* [`:` *Precond*] + *ResUsage*.

where $p(\bar{v}) : c(\bar{v})$ is defined by *Pred* and *Precond*, and Φ is defined by *ResUsage*. The information about $input_p$ and $size_p$ is implicit in *Precond* and *ResUsage*. The concretization of I_α, $\gamma(I_\alpha)$, is the set of all pairs $(p(\bar{t}), r)$ such that $\bar{t}$ is a tuple of terms and $p(\bar{t})$ is an instance of *Pred* that meets precondition *Precond*, and r is a number that meets the condition expressed by *ResUsage* (i.e., r lies in the interval defined by *ResUsage*) for some assertion.

Example 5.5.1. Consider the following HC IR program that computes the factorial of an integer.

```
1 fact(N,Fact) :- N=<0, Fact=1.
2 fact(N,Fact) :- N>0, N1 is N-1,
3         fact(N1,Fact1), Fact is N*Fact1.
```

One could use the assertion:

```
1 :- check pred fact(N,F)
2    : (num(N), var(F))
3   => (num(N), num(F),
```

```
4         rsize(N, num(Nmin, Nmax)),
5         rsize(F, num(Fmin, Fmax)))
6     + resource(steps, Nmin+1, Nmax+1).
```

to express that for any call to `fact(N,F)` with the first argument bound to a number and the second one a free variable, the number of resolution steps performed by the computation is always between `Nmin` + 1 and `Nmax` + 1, where `Nmin` and `Nmax` respectively stand for a lower and an upper bound of `N`. In this concrete example, the lower and upper bounds are the same, i.e., the number of resolution steps is exactly `N`+1, but note that they could be different. □

Example 5.5.2. The assertion in Example 5.5.1 captures the following concrete semantic tuples:

```
( fact(0, Y), 1 )    ( fact(8, Y), 9 )
```

but it does not capture the following ones:

```
( fact(N, Y), 1 )    ( fact(1, Y), 35 )
```

the left one in the first line above because it is outside the scope of the assertion (i.e., since N is a variable, it does not meet the precondition *Precond*), and the right one because it violates the assertion (i.e., it meets the precondition *Precond*, but does not meet the condition expressed by *ResUsage*). □

Partial correctness: comparing to the abstract semantics Given a program p and an intended resource usage semantics I, where $I : \mathscr{C}_p \mapsto \mathbb{R}$, we say that p is partially correct w.r.t. I if for all $p(\bar{t}) \in \mathscr{C}_p$ we have that $(p(\bar{t}), r) \in I$, where r is precisely the amount of resource usage of the computation of the call $p(\bar{t})$. We say that p is partially correct with respect to a tuple of the form $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ if for all $p(\bar{t}) \in \mathscr{C}_p$ such that r is the amount of resource usage of the computation of the call $p(\bar{t})$, it holds that: if $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$ then $r \in \Phi_I(\bar{s})$, where $\bar{s} = size_p(input_p(\bar{t}))$. Finally, we say that p is partially correct with respect to I_α if:

- For all $p(\bar{t}) \in \mathscr{C}_p$, there is a tuple $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ in I_α such that $p(\bar{t}) \in \gamma(p(\bar{v}) : c_I(\bar{v}))$, and
- p is partially correct with respect to every tuple in I_α.

Let $(p(\bar{v}) : c(\bar{v}), \Phi)$ and $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$ be tuples expressing an abstract semantics $[\![P]\!]_\alpha$ inferred by analysis and an intended abstract semantics I_α, respectively,

such that $c_I(\bar{v}) \sqsubseteq c(\bar{v})$,[2] and for all $\bar{n} \in S$ ($S \subseteq \mathbb{R}^k$), $\Phi(\bar{n}) = [\Phi^l(\bar{n}), \Phi^u(\bar{n})]$ and $\Phi_I(\bar{n}) = [\Phi^l_I(\bar{n}), \Phi^u_I(\bar{n})]$. We have that:

(1) If for all $\bar{n} \in S$, $\Phi^l_I(\bar{n}) \leq \Phi^l(\bar{n})$ and $\Phi^u(\bar{n}) \leq \Phi^u_I(\bar{n})$, then p is partially correct with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

(2) If for all $\bar{n} \in S$, $\Phi^u(\bar{n}) < \Phi^l_I(\bar{n})$ or $\Phi^u_I(\bar{n}) < \Phi^l(\bar{n})$, then p is incorrect with respect to $(p(\bar{v}) : c_I(\bar{v}), \Phi_I)$.

Checking the two conditions above requires the comparison of resource usage bound functions.

Resource Usage Bound Function Comparison Since the resource analysis we use is able to infer different types of functions (e.g., polynomial, exponential, and logarithmic), it is also desirable to be able to compare all of these functions.

For simplicity of exposition, consider first the case where resource usage bound functions depend on one argument. Given two resource usage bound functions (one of them inferred by the static analysis and the other one given in an assertion/specification present in the program), $\Psi_1(n)$ and $\Psi_2(n)$, $n \in \mathbb{R}$, the objective of the comparison operation is to determine intervals for n in which $\Psi_1(n) > \Psi_2(n)$, $\Psi_1(n) = \Psi_2(n)$, or $\Psi_1(n) < \Psi_2(n)$. For this, we define $f(n) = \Psi_1(n) - \Psi_2(n)$ and find the roots of the equation $f(n) = 0$. Assume that the equation has m roots, $n_1, \ldots, n_m$. These roots are intersection points of $\Psi_1(n)$ and $\Psi_2(n)$. We consider the intervals $S_1 = [0, n_1)$, $S_2 = (n_1, n_2)$, $S_m = \ldots (n_{m-1}, n_m)$, $S_{m+1} = (n_m, \infty)$. For each interval S_i, $1 \leq i \leq m$, we select a value v_i in the interval. If $f(v_i) > 0$ (respectively $f(v_i) < 0$), then $\Psi_1(n) > \Psi_2(n)$ (respectively $\Psi_1(n) < \Psi_2(n)$) for all $n \in S_i$.

There exist powerful algorithms for obtaining roots of polynomial functions. In our implementation we have used the GNU Scientific Library [Galassi et al., 2009], which offers a specific polynomial function library that uses analytical methods for finding roots of polynomials up to order four, and uses numerical methods for higher order polynomials.

We approximate exponential and logarithmic resource usage functions using Taylor series. In particular, for exponential functions we use the following formulae:

$$e^x \approx \Sigma_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \qquad for\ all\ x$$

$$a^x = e^{x\ ln\ a} \approx 1 + x\ ln\ a + \frac{(x\ ln\ a)^2}{2!} + \frac{(x\ ln\ a)^3}{3!} + \ldots$$

[2] Note that the condition $c_I(\bar{v}) \sqsubseteq c(\bar{v})$ can be checked using the CiaoPP capabilities for comparing program state properties such as types.

In our implementation these series are limited up to order 8. This decision has been taken based on experiments we have carried out that show that higher orders do not bring a significant difference in practice. Also, in our implementation, the computation of the factorials is done separately and the results are kept in a table in order to reuse them.

Dealing with logarithmic functions is more complex, as Taylor series for such functions can only be defined for the interval $(-1, 1)$.

For resource usage functions depending on more than one variable, the comparison is performed using constraint solving techniques.

Safety of the Approximations When the roots obtained for function comparison are approximations of the actual roots, we must guarantee that their values are safe, i.e., that they can be used for verification purposes, in particular, for safely checking the conditions presented above. In other words, we should guarantee that the error falls on the safe side when comparing the corresponding resource usage bound functions. For this purpose we developed an algorithm for detecting whether the approximated root falls on the safe side or not, and in the case it does not fall on the safe side, performing an iterative process to increment (or decrement) it by a small value until the approximated root falls on the safe side.

5.6 Using the Tool: Example

As an illustrative example of a scenario where the embedded software developer has to decide values for program parameters that meet an energy budget, we consider the development of an equaliser (XC) program using a biquad filter. In Figure 5.2 we can see what the graphical user interface of our prototype looks like, with the code of this biquad example ready to be verified. The purpose of an equaliser is to take a signal, and to attenuate / amplify different frequency bands. For example, in the case of an audio signal, this can be used to correct for a speaker or microphone frequency response. The energy consumed by such a program directly depends on several parameters, such as the sample rate of the signal, and the number of banks (typically between 3 and 30 for an audio equaliser). A higher number of banks enables the designer to create more precise frequency response curves.

Assume that the developer has to decide how many banks to use in order to meet an energy budget while maximizing the precision of frequency response curves at the same time. In this example, the developer writes an XC program where the number of banks is a variable, say `N`. Assume also that the energy constraint to be met is that an application of the biquad program should consume less than 125 milli-joules (i.e., 125000000 nano-joules). This constraint is expressed by the following check assertion:

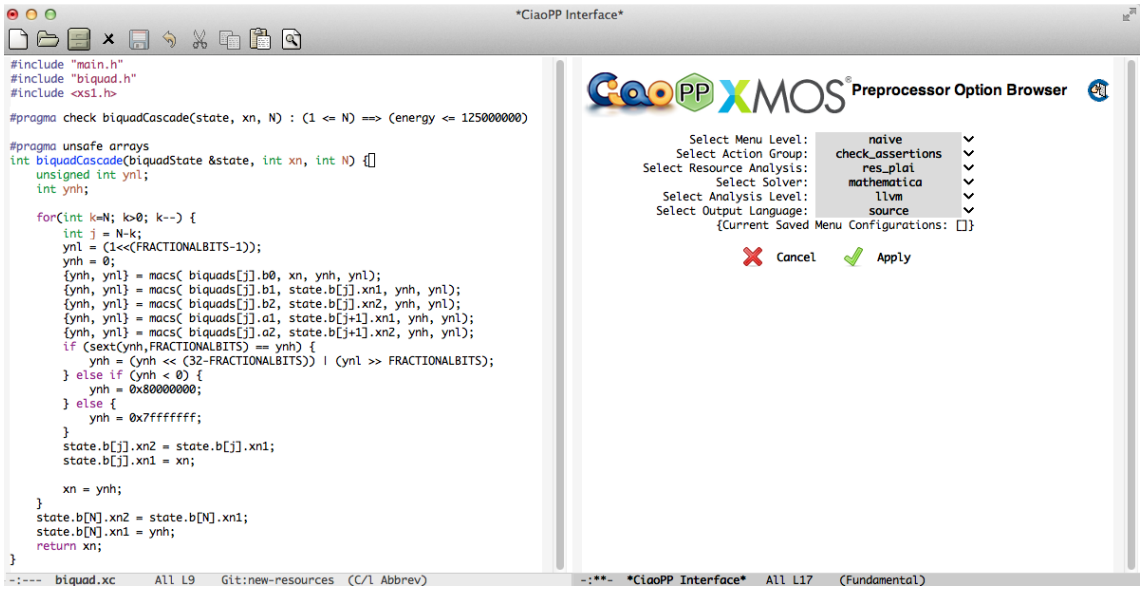

Figure 5.2: Graphical User Interface of the prototype with the XC biquad program.

```
#pragma check biquadCascade(state,xn,N):
              (1 <= N) ==> (energy <= 125000000)
```

where the precondition `1 <= N` in the assertion (left-hand side of ==>) expresses that the number of banks should be at least 1.

Then, the developer makes use of the tool, by selecting the following menu options, as shown in the right-hand side of Figure 5.2: `check_assertions`, for `Action Group`, `res_plai`, for `Resource Analysis`, `mathematica`, for `Solver`, `llvm`, for `Analysis Level` (which will tell the analysis to take the LLVM IR option by compiling the source code into LLVM IR and transforming it into HC IR for analysis) and finally `source`, for `Output Language` (the language in which the analysis / verification results are shown). After clicking on the `Apply` button below the menu options, the analysis is performed, which infers a lower and an upper bound function for the energy consumption of the program. Concretely those bounds are represented by the following assertion, which is included in the output of the tool:

```
#pragma true biquadCascade(state,xn,N):
             (16502087*N + 5445103 <= energy &&
              energy <= 16502087*N + 5445103)
```

In this particular case, both bounds are identical. In other words, the energy consumed by the program is exactly characterized by the following function, depending on `N` only:

$$E_{\text{biquad}}(\mathtt{N}) = 16502087 \times \mathtt{N} + 5445103 \text{ nJ}$$

Then, the verification of the specification (check assertion) is performed by comparing the energy bound functions above with the upper bound expressed in the specification, i.e., 125000000, a constant value in this case. As a result, the two following assertions are produced (and included in the output file of the tool):

```
#pragma checked biquadCascade(state,xn,N):
                (1 <= N && N <= 7) ==> (energy <= 125000000)
#pragma false biquadCascade(state,xn,N):
                (8 <= N) ==> (energy <= 125000000)
```

The first one expresses that the original assertion holds subject to a precondition on the parameter `N`, i.e., in order to meet the energy budget of 125 milli-joules, the number of banks `N` should be a natural number in the interval [1, 7] (precondition `1 <= N && N <= 7`). The second one expresses that the original specification is not met (status `false`) if the number of banks is greater or equal to 8.

Since the goal is to maximize the precision of frequency response curves and to meet the energy budget at the same time, the number of banks should be set to 7. The developer could also be interested in meeting an energy budget but this time ensuring a lower bound on the precision of frequency response curves. For example by ensuring that $N \geq 3$, the acceptable values for `N` would be in the range [3, 7].

In the more general case where the energy function inferred by the tool depends on more than one parameter, the determination of the values for such parameters is reduced to a constraint solving problem. The advantage of this approach is that the parameters can be determined analytically at the program development phase, without the need of determining them experimentally by measuring the energy of expensive program runs with different input parameters.

5.7 Related Work

As mentioned before, this work adds verification capabilities to the improved resource analysis developed in this thesis, which we have specialized for energy consumption of XC programs running on the XS1-L architecture, based on previous work on energy consumption [Liqat et al., 2014b] and on a general framework for resource usage analysis [Navas et al., 2007b, Navas et al., 2008, Serrano et al., 2014b, Hermenegildo et al., 2005a, Méndez-Lojo et al., 2007] and its support for resource verification [Lopez-Garcia et al., 2010b, Lopez-Garcia et al., 2012], and the energy models of [Kerrison and Eder, 2015].

Regarding the support for verification of properties expressed as functions, the closest related work we are aware of presents a method for comparison of cost func-

tions inferred by the COSTA system for Java bytecode [Albert et al., 2010]. The method proves whether a cost function is smaller than another one *for all the values* of a given initial set of input data sizes. The result of this comparison is a Boolean value. However, as mentioned before, in our approach [Lopez-Garcia et al., 2010b, Lopez-Garcia et al., 2012] the result is in general a set of subsets (intervals) in which the initial set of input data sizes is partitioned, so that the result of the comparison is different for each subset. Also, [Albert et al., 2010] differs in that comparison is syntactic, using a method similar to what was already being done in the CiaoPP system: performing a function normalization and then using some syntactic comparison rules. Our technique goes beyond these syntactic comparison rules. Moreover, [Albert et al., 2010] only covers (generic) cost function comparisons while we have addressed the whole process for the case of energy consumption *verification*. Note also that, although we have presented our work applied to XC programs, the CiaoPP system can also deal with other high- and low-level languages, including, e.g., Java bytecode [Navas et al., 2009, Méndez-Lojo et al., 2007].

In a more general context, using abstract interpretation in debugging and/or verification tasks has now become well established. To cite some early work, abstractions were used in the context of algorithmic debugging in [Lichtenstein and Shapiro, 1988]. Abstract interpretation has been applied by Bourdoncle [Bourdoncle, 1993] to debugging of imperative programs and by Comini et al. to the algorithmic debugging of logic programs [Comini et al., 1995] (making use of partial specifications in [Comini et al., 1999]), and by P. Cousot [Cousot, 2003] to verification, among others. The CiaoPP framework [Bueno et al., 1997, Hermenegildo et al., 1999, Hermenegildo et al., 2005a] was pioneering in many aspects, offering an integrated approach combining abstraction-based verification, debugging, and run-time checking with an assertion language.

5.8 Conclusions

We have specialized an existing general framework for resource usage verification for verifying energy consumption specifications of embedded programs. These specifications can include both lower and upper bounds on energy usage, expressed as intervals within which the energy usage is supposed to be included, the bounds (end points of the intervals) being expressed as functions on input data sizes. Our tool can deal with different types of energy functions (e.g., polynomial, exponential or logarithmic functions), in the sense that the analysis can infer them, and the specifications can involve them. We have shown through an example, and using the prototype implementation of our approach within the Ciao/CiaoPP system and for the XC language and XS1-L

architecture, how our verification system can prove whether such energy usage specifications are met or not, or infer particular conditions under which the specifications hold. These conditions are expressed as intervals of input data sizes such that a given specification can be proved for some intervals but disproved for others. The specifications themselves can also include preconditions expressing intervals for input data sizes. We have illustrated through this example how embedded software developers can use this tool, and in particular for determining values for program parameters that ensure meeting a given energy budget while minimizing the loss in quality of service.

Chapter 6

Conclusions and Future Work

In this thesis we have presented improvements and extensions to a prototype implementation of general resource usage analysis for logic programs, based on the framework of abstract interpretation. This analysis statically infers upper and lower bounds on the usage that a logic program makes of a set of predefined or user defined *resources*. The inferred bounds are in general functions of input data sizes. The analysis is completely automatic, without requiring annotations from the user, more than (optionally) the specification of the entry call of the program we want to analyze. This specification, as well as the extra information we (optionally) can provide to the analysis, and the results of the analysis, are in the form of *assertions* in the source code.

The analysis is based on the setting up of recurrence relations that describe size relations between input and output arguments, cardinality and resource consumption of a set of predicates. As recurrence relations are not as useful as closed-form functions, a key operation of the analysis is to *solve* or *over-approximate* these recurrences with closed-form functions. While there exist several software tools able to solve or bound recurrence relations, such as general Computer Algebra Systems (CAS), such as Mathematica®, Maxima, etc., or specialized solvers (e.g., PURRS or PUBS), none of these tools alone is able to handle *all* of the relations that could arise during the analysis of a program. In this thesis we have designed and implemented a modular solver architecture, in charge of all the algebraic operations, and, in particular of solving or bounding recurrence relations. This architecture overcomes many limitations and problems of the current solvers. Firstly, it allows to integrate many existing solvers, such as well-known CASs, and specialized solvers, deciding which one to use based on a previous classification of recurrence relations. By using a common expression syntax (called *algebraic expression syntax*), our architecture also allows to easily combine results from different back-end solvers. Secondly, it can be easily extended, both by integrating a new back-end solver to the framework, or by defining a *strategy* for solving relations. Strategies can be seen as *meta-solvers*, in the sense that they define particular algorithms for solving recurrence relations, by making use of the services provided

by back-end solvers, through the common interface defined in the architecture.

Another contribution of this thesis has been the implementation of a specialized solver for a particular class of recurrence relations that are very common in practice (in the presence of programs with accumulating parameters), but cannot be handled directly by most of the solvers. Our approach is based on the synthesis of linear ranking functions for bounding the number of times the recursive case of a recurrence can be applied, before reaching a base case.

We have also improved the design and implementation of the mentioned prototype for resource usage analysis based on abstract interpretation, integrated in the CiaoPP abstract interpretation framework, PLAI. The main benefits of the work we have done in this thesis, have been a better integration of the analysis with the rest of the system, and making it more effective, practical and maintainable, representing a step forward to industrial level applicability.

Finally, we have leveraged the CiaoPP resource verification framework and specialized it for verifying energy consumption specifications of embedded imperative programs written in the XC language (the work was published in [Lopez-Garcia et al., 2015]), and tested it in a real industrial case study. To achieve this, we also have integrated the new abstract interpretation-based resource analysis into the CiaoPP general framework for resource verification, so that the resource usage information inferred by the new analysis can be compared against specifications.

Regarding future directions of this research, we can mention resource usage analysis for concurrent programs. Concurrent programming is currently the mainstream technique to improve system performance. Many chip manufacturers are turning to multi-core processor designs as a way of increasing performance. Concurrency is inherently present in data centers applications, distributed systems and cloud services. The amount of energy consumed by these kind of architectures, not only for computation and communication but also for cooling, is impressively high. Therefore, the use of a resource analysis tool for estimating the energy consumption of concurrent programs will potentially contribute to reducing the footprint of energy usage worldwide, and will have an important impact on the industry in terms of energy and economic savings.

However, while resource usage analysis for sequential programming languages has received considerable attention, in concurrent programming there are comparatively much less results, due in part to the complexity that this paradigm adds to the problem. The execution of a concurrent program is achieved by interleaving the execution of its threads, according to some scheduler algorithm. This results in a combinatorial explosion of possible execution states, making the problem of exhaustive analysis intractable. Thus, developing effective tools for inferring resource usage of concurrent systems is an important challenge that will contribute to the applicability of the tool in the industry. In this sense, as a future work we will study the use of abstract interpreta-

tion as the theoretical framework for the development of a resource usage analysis for concurrent programs, because of its successful application to the sequential case and to the inference of some safety properties for concurrent programs. In order to tackle down the problem of thread interaction, we plan to study the use of Rely-guarantee proof methods, such as the ones described in in [Miné, 2012], for the representation of such interactions.

Bibliography

[Albert et al., 2010] Albert, E., Arenas, P., Genaim, S., Herraiz, I., and Puebla, G. (2010). Comparing cost functions in resource analysis. In *1st International Workshop on Foundational and Practical Aspects of Resource Analysis (FOPARA'09)*, volume 6234 of *Lecture Notes in Computer Science*, pages 1–17. Springer.

[Albert et al., 2008] Albert, E., Arenas, P., Genaim, S., and Puebla, G. (2008). Cost Relation Systems: a Language–Independent Target Language for Cost Analysis. In *8th Spanish Conference on Programming and Computer Languages (PROLE'08)*, volume 17615 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

[Albert et al., 2011a] Albert, E., Arenas, P., Genaim, S., and Puebla, G. (2011a). Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203.

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

[Albert et al., 2011b] Albert, E., Genaim, S., and Masud, A. N. (2011b). More Precise yet Widely Applicable Cost Analysis. In *12th Verification, Model Checking, and Abstract Interpretation (VMCAI'11)*, volume 6538 of *Lecture Notes in Computer Science*, pages 38–53. Springer Verlag.

[Bagnara et al., 2010] Bagnara, R., Mesnard, F., Pescetti, A., and Zaffanella, E. (2010). The automatic synthesis of linear ranking functions: The complete unabridged version. Quaderno 498, Dipartimento di Matematica, Università di Parma, Italy. Available at `http://www.cs.unipr.it/Publications/`.

[Bagnara et al., 2005] Bagnara, R., Pescetti, A., Zaccagnini, A., and Zaffanella, E. (2005). PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. Technical report. `arXiv:cs/0512056`.

[Bagnara et al., 2002] Bagnara, R., Ricci, E., Zaffanella, E., and Hill, P. M. (2002). Possibly not closed convex polyhedra and the Parma Polyhedra Library. In Hermenegildo, M. V. and Puebla, G., editors, *Static Analysis: Proceedings of the 9th International*

Symposium, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229, Madrid, Spain. Springer-Verlag, Berlin.

[Bjørner et al., 2014] Bjørner, N., Fioravanti, F., Rybalchenko, A., and Senni, V., editors (2014). *Workshop on Horn Clauses for Verification and Synthesis*. Electronic Proceedings in Theoretical Computer Science.

[Bourdoncle, 1993] Bourdoncle, F. (1993). Abstract debugging of higher-order imperative languages. In *Programming Languages Design and Implementation'93*, pages 46–55.

[Bruynooghe, 1987] Bruynooghe, M. (1987). A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven.

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

[Bruynooghe, 1991b] Bruynooghe, M. (1991b). A practical framework for the abstract interpretation of logic programs. *J. Log. Program.*, 10(2):91–124.

[Bueno et al., 1997] Bueno, F., Deransart, P., Drabent, W., Ferrand, G., Hermenegildo, M., Maluszynski, J., and Puebla, G. (1997). On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. WS on Automated Debugging–AADEBUG*, pages 155–170. U. Linköping Press.

[Bueno et al., 2004] Bueno, F., Lopez-Garcia, P., and Hermenegildo, M. (2004). Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany. Springer-Verlag.

[Cohen and Katcoff, 1977] Cohen, J. and Katcoff, J. (1977). Symbolic solution of finite-difference equations. *ACM Trans. Math. Softw.*, 3(3):261–271.

[Comini et al., 1999] Comini, M., Levi, G., Meo, M. C., and Vitiello, G. (1999). Abstract diagnosis. *Journal of Logic Programming*, 39(1–3):43–93.

[Comini et al., 1995] Comini, M., Levi, G., and Vitiello, G. (1995). Declarative diagnosis revisited. In *1995 International Logic Programming Symposium*, pages 275–287, Portland, Oregon. MIT Press, Cambridge, MA.

[Cousot, 2003] Cousot, P. (2003). Automatic Verification by Abstract Interpretation, Invited Tutorial. In *Fourth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, number 2575 in LNCS, pages 20–24. Springer.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). 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.

[Cousot and Cousot, 1992] Cousot, P. and Cousot, R. (1992). Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179.

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

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

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

[Debray et al., 1997] Debray, S. K., Lopez-Garcia, P., Hermenegildo, M., and Lin, N.-W. (1997). Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA.

[Floyd, 1967] Floyd, R. W. (1967). Assigning Meanings to Programs. In Schwartz, J., editor, *Proceedings of Symposium in Applied Mathematics*, volume 19, Mathematical Aspects of Computer Science, pages 19–32. American Mathematical Society, Providence, RI.

[Galassi et al., 2009] Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., and Rossi, F. (2009). *GNU Scientific Library Reference Manual*. Network Theory Ltd. Available at `http://www.gnu.org/software/gsl/`.

[Georgiou et al., 2014] Georgiou, K., Kerrison, S., and Eder, K. (2014). A Multi-level Worst Case Energy Consumption Static Analysis for Single and Multi-threaded Embedded Programs. Technical Report CSTR-14-003, University of Bristol.

[Henriksen and Gallagher, 2006] Henriksen, K. S. and Gallagher, J. P. (2006). 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.

[Hermenegildo et al., 1999] Hermenegildo, M., Puebla, G., and Bueno, F. (1999). Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In Apt, K. R., Marek, V., Truszczynski, M., and Warren, D. S., editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag.

[Hermenegildo et al., 2005a] Hermenegildo, M., Puebla, G., Bueno, F., and Garcia, P. L. (2005a). Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140.

[Hermenegildo et al., 2005b] Hermenegildo, M., Puebla, G., Bueno, F., and Lopez-Garcia, P. (2005b). Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140.

[Hermenegildo et al., 2012a] Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., and Puebla, G. (2012a). An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252.

[Hermenegildo et al., 2012b] Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., and Puebla, G. (2012b). An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252.

[Hughes et al., 1996] Hughes, J., Pareto, L., and Sabry, A. (1996). Proving the correctness of reactive systems using sized types. In *POPL*, pages 410–423.

[Ivie, 1978] Ivie, J. (1978). Some macsyma programs for solving recurrence relations. *ACM Trans. Math. Softw.*, 4(1):24–33.

[Jayaseelan et al., 2006] Jayaseelan, R., Mitra, T., and Li, X. (2006). Estimating the worst-case energy consumption of embedded software. In *IEEE Real Time Technology and Applications Symposium*, pages 81–90. IEEE Computer Society.

[Kerrison and Eder, 2015] Kerrison, S. and Eder, K. (2015). Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–88. IEEE Computer Society.

[Lichtenstein and Shapiro, 1988] Lichtenstein, Y. and Shapiro, E. Y. (1988). Abstract algorithmic debugging. In Kowalski, R. A. and Bowen, K. A., editors, *Fifth International Conference and Symposium on Logic Programming*, pages 512–531, Seattle, Washington. MIT.

[Liqat et al., 2014a] Liqat, U., Georgiou, K., Kerrison, S., Lopez-Garcia, P., Hermenegildo, M., Gallagher, J. P., and Eder, K. (2014a). Inferring Energy Consumption at Different Software Levels: ISA vs. LLVM IR. Technical report, ENTRA Project. Appendix D3.2.4 of Deliverable D3.2. Available at `https://entraproject.eu`.

[Liqat et al., 2014b] Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M., and Eder, K. (2014b). Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In *Proceedings of LOPSTR'13*, volume 8901 of *LNCS*, pages 72–90. Springer.

[Lopez-Garcia, 2014] Lopez-Garcia, P., editor (2014). *Initial Energy Consumption Analysis*. ENTRA Project: Whole-Systems Energy Transparency (FET project 318337). Deliverable 3.2, https://entraproject.eu.

[Lopez-Garcia et al., 2010a] Lopez-Garcia, P., Bueno, F., and Hermenegildo, M. (2010a). Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Analyses. *New Generation Computing*, 28(2):117–206.

[Lopez-Garcia et al., 2010b] Lopez-Garcia, P., Darmawan, L., and Bueno, F. (2010b). A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification. In *Technical Communications of ICLP*, volume 7 of *LIPIcs*, pages 104–113. Schloss Dagstuhl.

[Lopez-Garcia et al., 2010c] Lopez-Garcia, P., Darmawan, L., and Bueno, F. (2010c). A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification. In Hermenegildo, M. and Schaub, T., editors, *Technical Communications of the 26th Int'l. Conference on Logic Programming (ICLP'10)*, volume 7 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 104–113, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Lopez-Garcia et al., 2012] Lopez-Garcia, P., Darmawan, L., Bueno, F., and Hermenegildo, M. (2012). Interval-Based Resource Usage Verification: Formalization and Prototype. In *Proc. of FOPARA*, volume 7177 of *LNCS*, pages 54–71. Springer-Verlag.

[Lopez-Garcia et al., 2015] Lopez-Garcia, P., Haemmerlé, R., Klemen, M., Liqat, U., and Hermenegildo, M. (2015). Towards Energy Consumption Verification via Static Analysis. In *Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES), arXiv:1501.03064*. arXiv:1512.09369.

[Méndez-Lojo et al., 2007] Méndez-Lojo, M., Navas, J., and Hermenegildo, M. (2007). A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *LOPSTR*, volume 4915 of *LNCS*, pages 154–168. Springer-Verlag.

[Mera et al., 2008] Mera, E., Lopez-Garcia, P., Carro, M., and Hermenegildo, M. (2008). Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press.

[Miné, 2012] Miné, A. (2012). Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63. `http://www.di.ens.fr/~mine/publi/article-mine-LMCS12.pdf`.

[Miné, 2013] Miné, A. (2013). Static analysis by abstract interpretation of concurrent programs. Technical report, École normale supérieure. `http://www.di.ens.fr/~mine/hdr/hdr-compact-col.pdf`.

[Muthukumar and Hermenegildo, 1989] Muthukumar, K. and Hermenegildo, M. (1989). Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press.

[Muthukumar and Hermenegildo, 1990] Muthukumar, K. and Hermenegildo, M. (1990). Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759.

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

[Navas et al., 2008] Navas, J., Méndez-Lojo, M., and Hermenegildo, M. (2008). Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, pages 29–32. Extended Abstract.

[Navas et al., 2009] Navas, J., Méndez-Lojo, M., and Hermenegildo, M. (2009). User-Definable Resource Usage Bounds Analysis for Java Bytecode. *ENTCS - BYTECODE'09*, 253(5):6–86.

[Navas et al., 2007a] Navas, J., Mera, E., Lopez-Garcia, P., and Hermenegildo, M. (2007a). User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*, pages 348–363. Springer. 10-year Test of Time Award.

[Navas et al., 2007b] Navas, J., Mera, E., Lopez-Garcia, P., and Hermenegildo, M. (2007b). User-Definable Resource Bounds Analysis for Logic Programs. In *Proc. of ICLP'07*, volume 4670 of *LNCS*, pages 348–363. Springer.

[Petkovšek, 1990] Petkovšek, M. (1990). *Finding Closed-form Solutions of Difference Equations by Symbolic Methods*. Number v. 91-103 in Finding closed-form solutions of difference equations by symbolic methods. School of Computer Science, Carnegie Mellon University.

[Podelski and Rybalchenko, 2004] Podelski, A. and Rybalchenko, A. (2004). A Complete Method for the Synthesis of Linear Ranking Functions. In Steffen, B. and Levi, G., editors, *5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer.

[Puebla et al., 2000] Puebla, G., Bueno, F., and Hermenegildo, M. (2000). An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag.

[Puebla and Hermenegildo, 1996] Puebla, G. and Hermenegildo, M. (1996). Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag.

[Rosendahl, 1989] Rosendahl, M. (1989). Automatic Complexity Analysis. In *4th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'89)*, pages 144–156. ACM Press.

[Serrano et al., 2013] Serrano, A., Lopez-Garcia, P., Bueno, F., and Hermenegildo, M. (2013). Sized Type Analysis for Logic Programs. *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, Online Supplement (technical communication)*, 13(4-5):1–14.

[Serrano et al., 2014a] Serrano, A., Lopez-Garcia, P., and Hermenegildo, M. (2014a). Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754.

[Serrano et al., 2014b] Serrano, A., Lopez-Garcia, P., and Hermenegildo, M. (2014b). Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP, ICLP'14 Special Issue*, 14(4-5):739–754.

[Vasconcelos and Hammond, 2003] Vasconcelos, P. and Hammond, K. (2003). Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proceedings of the International Workshop on Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*, pages 86–101. Springer-Verlag.

[Vaucheret and Bueno, 2002] Vaucheret, C. and Bueno, F. (2002). More Precise yet Efficient Type Inference for Logic Programs. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag.

[Watt, 2009] Watt, D. (2009). *Programming XC on XMOS Devices*. XMOS Limited.

[Wegbreit, 1975] Wegbreit, B. (1975). Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539.

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