Analysis and Inference of Resource Usage Information

by

Jorge A. Navas Laserna

B.S., Computer Science, Technical Univ. of Madrid, 2003
M.S., Computer Science, Univ. of New Mexico, 2006
M.B.A., Business Administration, Univ. of New Mexico, 2008

Advisor: **Manuel V. Hermenegildo**

DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Dedication

A mis padres y Yosune

Acknowledgments

First, I would like to thank my parents who gave me what they did not have to reach this goal. I do not have words to express my gratitude to them and I hope to make them happy and proud of me despite the long distance which separates us. In Spanish:

"Gracias a mis padres que me dieron todo lo que no tenían para alcanzar este objetivo. No tengo palabras para expresarles mi gratitud y espero haberles hecho feliz y orgullosos de mi a pesar de la gran distancia que nos separa."

Also, nothing of this could have been possible without Yosune. She always supported me in my few good days and, more important, in my numerous bad days. Thank you for loving and respecting me. To them I dedicate this thesis.

I want to thank Manuel Hermenegildo my mentor, collaborator, and friend, for all his time, energy and resources who taught me a lot and allowed me to finish this thesis. He showed me this amazing world of research supporting me during these years in very different ways. I also gratefully acknowledge the support of the Prince of Asturias Chair in Information Science and Technology at UNM funded by Iberdrola.

I would like to thank specially Mario Méndez for his friendship and fruitful series of common works that are a fundamental part of this thesis, and also Amadeo Casas for his interesting scientific discussions. They became my best friends during these years and we shared great moments in Albuquerque.

Throughout these years, I would like also to thank the rest of my co-authors of the papers that are part of this thesis: Elena Ackley, Francisco Bueno, Stephanie Forrest, Pedro López-García, Edison Mera, and Eric Trias. I also want to thank all members of the CLIP Group who allowed me to take advantage of many existing tools and analyses used in this thesis. I would like also to thank specially UNM Professors Deepak Kapur and George Luger, members of my committee, who have provided me with many helpful comments about this thesis and contributed to its improvement.

Finally, I cannot forget my brother Kike and all my friends who might not have contributed to this thesis directly, but without their participation it would have been impossible to make it: Anick, Basam, Gabriel, Jose, Lili, Manoito, Natalia, Raquel, Roberto, Salvador, Sandra, Myriam, ..., and of course, the wonderful land of New Mexico and its people.

Acknowledgments are hard to write since, inevitably, someone important is left out by mistake. I apologize to you.

Jorge Navas
August 2008

Analysis and Inference of Resource Usage Information

by

Jorge A. Navas Laserna

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

Analysis and Inference of Resource Usage Information

by

Jorge A. Navas Laserna

B.S., Computer Science, Technical Univ. of Madrid, 2003

M.S., Computer Science, Univ. of New Mexico, 2006

M.B.A., Business Administration, Univ. of New Mexico, 2008

Ph.D., Computer Science, University of New Mexico, 2008

Abstract

Static analysis is a powerful technique used traditionally for optimizing programs, and, more recently, in tools to aid the software development process, and in particular, in finding bugs and security vulnerabilities. More concretely, the importance of static analyses that can infer information about the costs of computations is well recognized since such information is very useful in a large number of applications. Furthermore, the increasing relevance of analysis applications such as static debugging, resource bounds certification of mobile code, and granularity control in parallel computing makes it interesting to develop analyses for resource notions that are actually application-dependent. This may include, for example, bytes sent or received by an application, number of files left open, number of SMSs sent or received, energy consumption, number of accesses to a database, etc.

In this thesis, we present a resource usage analysis that aims at inferring upper and lower bounds on the cost of programs as a function of its data size for a given set of user-definable resources of interest. We use logic programming as our basic paradigm since it subsumes many others and this allows us treating the problem at a considerable level of generality.

Resource usage analysis requires various pre-analysis steps. An important one is Set-Sharing analysis which attempts to detect mode information and which variables do not point to the same memory location, providing essential information to the resource usage analysis. Hence, this thesis also investigates the problem of efficient Set-Sharing analyses presenting two different alternatives: (1) via widening operators, and (2) defining compact and effective encodings.

Moving to the area of applications, a very interesting class involves certification of the resources used by mobile code. In this context, Java bytecode is widely used, mainly due to its security features and the fact that it is platform independent. Therefore, this thesis finally presents a resource usage analysis tool for Java bytecode that includes also a transformation which provides a block-level view of the bytecode, and can be used as a basis for developing analyses. We have also developed for this purpose, a generic, abstract interpretation-based fixpoint algorithm which is parametric in the abstract domain. By plugging appropriate abstract domains into it, the framework provides useful information that can improve the accuracy of the resource usage information.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Static program analysis is the process of inferring information at compile-time on the runtime behavior of the program. Static program analysis has many important applications. It has traditionally been used primarily for optimizing programs so that they will run faster. More recently, program analysis is increasingly being used in tools to aid the software development process and it is extremely useful in finding bugs and security vulnerabilities in software.

Although many promising advancements have been achieved, there are still many challenges in software reliability and development in the real world. One of the biggest issues is the lack of more practical tools but the fact is that static program analysis is a very hard task. The current complexity of software makes sometimes existing static analysis tools unable to infer information in a reasonable amount of time. On the other hand, most problems in static program analysis are undecidable, which means the use of approximate algorithms is needed. To make these approximated algorithms work well on real programs is also a challenge.

The importance of static analyses that can infer information about the costs of computations is well recognized since such information is useful in a large number of applications. The kinds of costs which have received most attention so far are related to a fixed set of resources such as execution steps as well as, sometimes, execution time or memory (see, e.g., [69, 104, 101, 108, 45, 15, 47] for functional languages, [111, 14, 40, 119] for imperative languages, and [37, 36, 38] for logic languages).

These and other types of cost analyses have been used in the context of applications such as granularity control in parallel and distributed computing (e.g., [80, 24]), resource-oriented specialization (e.g., [32, 100]), or, more recently, certification of the resources used by mobile code (e.g., [8, 25, 4, 51]). Specially in these more recent applications, the properties of interest are often higher-level, user-oriented, and application-dependent. Examples of such programmer-definable resources are bits sent or received by an application over a socket, number of files left open, number of SMSs sent or received, number of accesses to a database, energy consumption, monetary units spent, disk space used, etc.

Some recent work does deal with less restricted sets of resources ([116], [1]). However, while the approaches proposed can conceptually be adapted to infer some application-dependent resources in addition to the more traditional costs, the number of resources of interest may be unbounded since it depends on each application. Therefore, for each analysis developed the set of measured resource is fixed and changes in their implementations are needed to develop analyses for other resources.

Among existing programing paradigms, in an important part of this thesis we use logic programming [65] as our basic paradigm since it subsumes many others and this allows us treating the problem at a considerable level of generality. Moreover, languages based on logic programming are considered well suited for program analysis due to their very high abstraction level and higher separation of control issues from the logical specification of the problem. This observation was expressed by Kowalsky in the following equation [66]:

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

Regarding the object of certification, in the case of mobile code we do not have access to the source code, but only to compiled code. Therefore, the certification and checking processes are often performed at the bytecode level, since, in addition to other reasons of syntactic convenience, bytecode is what is most often available at the receiving (checker) end. In this context, Java bytecode [75] is widely used because of its security features and its condition of platform independent. Due to these reasons, we also cover in this thesis analysis of Java bytecode.

1.1 Thesis Objectives

The final objective of the work presented in this thesis is the development, implementation, and experimental evaluation of a set of advanced static analysis-based compilation techniques with special emphasis on resource-related properties. We believe that they contribute to the state of the art of this research area and can potentially improve the program development process.

In order to achieve the objectives mentioned above, the thesis develops a fully automated resource bounds analysis for logic programs which is quite independent of the particular resource of interest, based on the philosophy: *"write once run for any resource"*. To do this, the analysis uses a resource notion that is actually application-dependent. In our context a resource is easily defined by the user through a flexible and powerful assertion language which, for each external or built-in called by a procedure in the program to be analyzed, provides its cost in terms of that particular resource. The objective of our method is to statically derive from these elementary assertions upper and lower bounds on the amount of those resources that each of the procedures in the program (and the program as a whole) will consume or provide. This approach allows us to infer almost any kind of resource without changes in the implementation.

The automatic inference of resource usage information relies on other analyses and the accuracy and efficiency of these analyses have a deep impact on it. An important issue in resource usage analysis of logic programs is that certain program information must be first automatically inferred by other (abstract interpretation-based) analyzers. Such analyses must, for example:

- Determine which argument is input or output. This is called the *mode* of an argument.
- Infer the type of each argument, since Prolog is an untyped (or, rather, dynamically typed) programming language.
- Optionally, if lower bounds are to be inferred, detect which procedures fail or not, which can improve considerably the precision of the results.

Set-Sharing analyses aim to detect which variables do not point transitively to the same memory location. This information can provide very accurate input/output modes to the resource usage analysis. However, traditional Set-Sharing analyses can also be quite inefficient and they are not good choices when analyzing large programs. Because of this, this thesis also aims at the development of efficient Set-Sharing analyses presenting two practical solutions.

We first present a more efficient Set-Sharing analysis via *widening* operators. Traditionally, widening operators are used in the context of abstract interpretation [31] to ensure, in some cases, the termination of the analysis. In our context, we use widening to speed up the fixpoint computation. We define the abstract functions required by standard analysis frameworks, and also define several widening operators.

We then evaluate the efficiency and precision of the resulting analyses, and discuss the interactions between thresholds, precision, efficiency and cost of the widenings.

In our second proposal, we present an alternative approach: we define a new representation that leverages the complement (or negative) sharing relationships of the original sharing relationships, without loss of accuracy. The intuition is that switching to the complement representation can dramatically reduce the number of elements that need to be represented during the fixpoint computation when the number of relationships becomes large.

Since this thesis is focused on the design and implementation of tools that help the development of real-life programs, it cannot omit the fact that in numerous real applications the source code is not available, but only its bytecode version. In this context, Java bytecode has become very popular because of the security features and its platform-independent nature. Therefore, the thesis also develops a tool for the inference of resource usage information for Java bytecode. It should be also noticed that, although this work is clearly inspired by the proposal for logic programs, an adaptation from logic programs to Java programs is required because of issues such as virtual method invocation, exceptions, unstructured control flow, assignment, etc.

To fill all these gaps, the thesis presents the architecture of an analysis tool which takes a Java bytecode program and, a set of resources of interest and attempts to compute an upper bound of its resource usage as a (closed form) expression depending on the input data size. The inference of resource usage information requires first the construction, from the program, of an intermediate representation representing the Control Flow Graph (CFG) of the original program. This provides a uniform high-level representation which allows us to reason compositionally about the cost. The tool also includes an abstract interpretation-based fixpoint algorithm for analysis of Java bytecode which is parametric on the abstract domain. The fixpoint algorithm receives a CFG of the original program, and computes a safe approximation of the state of the program in terms of the abstraction chosen. The fixpoint is efficient (due to memoization techniques and dependency tracking) and precise (because of the top-down, context sensitive approach adopted). By plugging appropriate abstract domains such as class hierarchy analysis and nullity into the fixpoint algorithm, our framework can provide useful information that may improve the accuracy of the resource usage information.

1.2 Main Contributions

We now enumerate the main contributions of this thesis. Since we have completed some parts of the work in collaboration with other researchers, we also mention their names and institutions to which they belong to, and the level of our contribution. We also mention the publications resulting from each part of the work.

Regarding the problem of resource usage inference for logic programs:

- We propose a novel resource bounds analysis for logic programs which allows automatically inferring both upper and lower bounds on the usage that a logic program makes of a set of application programmer-definable resources of interest. This work has been done in collaboration with Prof. Pedro López-García (Technical University of Madrid) and Edison Mera (Complutense University of Madrid) and has been published at the 23rd International Conference on Logic Programming (ICLP) in 2007 [95]. I am the main contributor to this work.

- We have studied the problem of scalable Set-Sharing analyses which play a pivotal role in resource bounds analysis of logic programs proposing:

  - A Set-Sharing analysis via widening that accelerates the fixpoint computation. This work has been done in collaboration with Prof. Francisco

  Bueno (Technical University of Madrid) and has been published at the 8th International Symposium on Practical Aspects of Declarative Languages (PADL) in 2006 [93]. I am the main contributor to this work.

  – A Set-Sharing analysis using a novel encoding that compacts the representation of sharing relationships among variables by working with the complement (or negative) set of the original relationships. This work has been done in collaboration with Eric Trias and Elena S. Ackley (Univ. of New Mexico) and Prof. Stephanie Forrest (Univ. of New Mexico) and has been published at the 24th International Conference on Logic Programming (ICLP) in 2008 [112]. I am the main contributor to this work. Eric Trias and Elena S. Ackley contributed to the formalization and implementation.

Regarding the problem of inference of resource usage information for Java bytecode:

- We present the architecture of a tool for inference of resource usage information for Java bytecode. The tool includes:
  - An intermediate language representing the Control Flow Graph (CFG) of the original Java bytecode program.
  - An abstract interpretation-based fixpoint algorithm which is parametric on the abstract domain.
  - A resource usage analysis that takes the CFG of a Java bytecode program and a set of resources of interest and tries to compute an upper bound of its resource usage. In addition, by plugging appropriate abstract domains into the fixpoint algorithm, the tool may improve the accuracy of the resource usage functions.

This tool has been developed in collaboration with Mario Méndez (Univ. of New Mexico). The intermediate language has been published at 17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR) in 2007 [84]. The fixpoint algorithm has been published at the ETAPS 2nd Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07) in 2007 [85] and at the ECOOP 9th Workshop on Formal Techniques for Java-like Programs (FTfJP) also in 2007 [94]. Finally, the resource usage analysis is submitted for publication at the time of writting this thesis. Mario Méndez and I have contributed in approximately equal amounts to this work.

1.3 Structure of the Work

The structure of the rest of this thesis is as follows:

- Chapter 2 provides background about logic programming required to understand an important part of this thesis.
- Chapter 3 gives background about abstract interpretation that will be required to understand several chapters of this thesis.
- Chapter 4 describes the resource bounds analysis for logic programs and presents some experimental results of the implementation completed of such analysis.
- Chapter 5 provides background information regarding the Set-Sharing analysis that will be required to understand Chapters 6 and 7.
- Chapter 6 describes our first approach to Set-Sharing analysis using widening operators. It also shows experimental results which allow evaluating the im-

provement obtained with respect to the original Set-Sharing analysis and other efficient approaches.

- Chapter 7 presents the alternative approach to Set-Sharing working with the negative of the sharing relationships. The chapter also shows an initial experimental evaluation of the approach and compares it with respect to the original Set-Sharing analysis.

- Chapter 8 describes the intermediate language and the abstract interpretation-based fixpoint algorithm that we propose. Both components will be used by the analyzer shown in Chapter 9. We also show some experimental results for standard benchmarks, which further support the feasibility of the solution adopted.

- Chapter 9 presents the generic resource usage analysis for Java bytecode proposed. This chapter also presents experimental results that support the practicability of the approach.

- Finally, Chapter 10 presents our main conclusions and proposed directions for future work.

Chapter 2

Logic Programming

This chapter gives a brief review of basic notions of logic programming based mainly on [71], [48], [49], and [35]. For a more extensive introduction to general aspects of logic programming, the reader is referred to VanEmdem and Kowalski [115], Kowalski [66], Lloyd[76], and Apt [6].

2.1 Definitions: First-Order Logic and Syntax of Logic Programs

Definition 2.1.1. (Alphabet). An *alphabet* consists of:

- a possible empty set of *function symbols*. They are denoted by lower case letters starting from the letter f, for example f,g,h, ...
- *predicate symbols*. They are denoted by lower case letters beginning with p, for instance p,q,r, ...

- *variables*. They are denoted by upper case letters selected from the end of the alphabet, such as X,Y,Z, . . .
- *connectives*: $\neg$,$\vee$,$\wedge$,$\leftarrow$,$\leftrightarrow$.
- *quantifiers*: $\forall$, $\exists$.
- punctuation symbols, such as brackets and comma.

Functions and predicate symbols have an associated *arity* that represents the number of arguments. *Constant* symbols are a special case of functions when the arity is zero. They are denoted by lower case letters starting from a,b,c, . . . On the other hand, a predicate with arity of zero is called a *proposition*.

■

In the case of logic programs, the following notation is used:

- All variables are quantified universally. Therefore, quantifiers are omitted.
- The conjunction operator $\wedge$ is replaced by a comma.
- Lists are represented as in Prolog such that $[H|T]$ denotes the distinguished functor $.(H,T)$ where H is the head element of the list and T is the tail. The empty list, *nil*, is denoted by [ ].
- A don't care variable is denoted by the symbol $'\_'$.

We can refer to functions or predicates using their functor or predicate symbols and arity. For instance, the predicate $p(X,Y)$ is denoted by $p/2$.

Definition 2.1.2. (Term). The set of *terms* over some alphabet is defined recursively as:

- a variable is a term.
- a constant is a term.
- a function symbol f of arity $n > 0$ applied to the sequence $t_1, \dots, t_n$ of n terms, $f(t_1, \dots, t_n)$ is a term.

■

Definition 2.1.3. (Atom). The set of *atoms* over some alphabet is defined as:

- a proposition is an atom.
- a predicate symbol p of arity $n > 0$ applied to the sequence $t_1, \dots, t_n$ of n terms, denoted by $p(t_1, \dots, t_n)$ is an atom.

■

Definition 2.1.4. (Ground). A term or atom is *ground* if it does not contain any variable. ■

Definition 2.1.5. (Formula). A *formula* over some alphabet is recursively defined as:

- an atom is a formula.
- if A and B are formulas then $\neg A$, $A \wedge B$, $A \vee B$, $A \leftarrow B$, and $A \leftrightarrow B$ are also formulas.
- if X is a variable and A is a formula, then $\forall X A$ and $\exists X B$ are also formulas.

■

Definition 2.1.6. (Literal) If A is an atom then the formulas A and $\neg A$ are called *literals.* A is called a *positive literal* and $\neg A$ is called *negative literal.* In this thesis, we will only use positive literals. ∎

Definition 2.1.7. (Clause). A *clause* is a formula of the form $\forall(H_1 \vee \ldots, H_m \vee \leftarrow B_1 \wedge \ldots \wedge B_n)$ were $m \geq 0$, $n \geq 0$ and $H_1, \ldots, H_m, B_1, \ldots, B_n$ are all literals.

The left hand side of the formula $H_1, \ldots, H_m$ is called the *head* of the clause, and the right hand side $B_1, \ldots, B_n$ is called the *body* of the clause. ∎

Definition 2.1.8. (Horn Clause). A *Horn clause* is clause in which there is at most one positive literal in the head of the clause.

- A *fact* is a clause with an empty body.
- A *goal* is a clause with an empty head and a non-empty body.
- A clause with only atoms containing no variables is called a *ground instance.*

∎

Definition 2.1.9. (Substitution). Let $X_i \mapsto t_i$ be a *binding* between a variable X_i and a term t_i such that $X_i \neq t_i$. A *substitution* θ is a finite set of bindings, $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ were $\{X_1, \ldots, X_n\}$ are distinct.

∎

Definition 2.1.10. (Interpretation). Given a first-order language $\mathcal{L}$, an *interpretation* I for $\mathcal{L}$ consists of:

- a non-empty set D called the domain of interpretation D.
- an assignment for each constant in L of an element in D.

- an assignment for each n-ary function in L of a mapping from $D^n \to D$.
- an assignment for each n-ary predicate in L of a subset of D^n.

∎

Definition 2.1.11. (Herbrand Universe). The *Herbrand Universe* for the language $\mathcal{L}$ is the set of all ground terms that can be formed from the function symbols including constants. ∎

Definition 2.1.12. (Herbrand Base). The *Herbrand Base* for the language $\mathcal{L}$ is the set of all ground atoms that can be formed using predicate symbols and where their arguments are in the Herbrand Universe. ∎

If $\mathcal{L}$ is associated with a program P, we denote the Herbrand universe and base as U_P and B_P, respectively. For instance, let $P = \{p(f(X)) \leftarrow p(X).\ p(a).\ q(a).\ q(b).\}$ be a program, then for the language $\mathcal{L}$ associated with P, we can define

- $U_P = \{a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots\}$
- $B_P = \{p(a), p(b), q(a), q(b), p(f(a)), p(f(b)), q(f(a)), \ldots\}$

Definition 2.1.13. (Herbrand interpretation). The *Herbrand interpretation* for a language $\mathcal{L}$ is the interpretation defined by:

- The domain of the interpretation is U_P.
- Constants in $\mathcal{L}$ are mapped to themselves in U_P.
- For all function symbols f/n in $\mathcal{L}$ and all terms $t_1, \ldots, t_n \in U_P$, f applied to $t_1, \ldots, t_n$ is mapped to $f(t_1, \ldots, t_n)$.
- For all n-ary predicate p/n in $\mathcal{L}$ and all terms $t_1, \ldots, t_n \in U_P$, p applied to $t_1, \ldots, t_n$ is mapped to an element in $\wp((U_P)^n)$.

Thus, a Herbrand interpretation is uniquely determined by a subset of B_P.

■

Definition 2.1.14. (Model). A *model* of a formula (over a domain D) is an interpretation in which the formula has the value `true` assigned to it.

The concept of a model of a formula can be extended to sets of formulas. A model of a set S of formulas is an interpretation which is a model of all formulas in S. Two formulas are *logically equivalent* if they have the same set of models. A formula Q is a *logical consequence* of a set S of formulas if Q is assigned the value `true` in all models of S and it is denoted by $S \models Q$.

■

Definition 2.1.15. (Herbrand model). A *Herbrand model* of a program P of the language $\mathcal{L}$ is any Herbrand interpretation of $\mathcal{L}$ that is also a model of P. A Herbrand model $M \subseteq B_P$ for a program P is a *least Herbrand model* if no other $H' \subset H$ is also a Herbrand model of P.

The least Herbrand model captures the meaning of a program. It contains all the atomic logical consequences of the program. A formula that is `true` in the least Herbrand model is true in all Herbrand models. ■

2.2 Semantics of Logic Programs

The semantics of a program is the meaning assigned to this program. For a logic program P, its semantics is equivalent to the least Herbrand model of P, and it defines the set of atomic logical consequences of P.

2.2.1 Declarative Semantics

The least Herbrand model can be obtained as the least fixpoint of the function T_P. The theoretical foundation of this semantics is based on among other things, *complete lattices* and *monotone functions* over complete lattices. We postpone the definitions of these concepts to the next chapter.

Definition 2.2.1. (T_P). Let P be a program, the *immediate consequence operator* $T_P : 2^{B_P} \leftarrow 2^{B_P}$ is defined as follows:

$$T_P(I) = \{H \in B_P \mid \exists C \in ground(P), C = H \leftarrow B_1, ..., B_n \text{ and } B_1, ..., B_n \in I\}$$

where $ground(C) = \{C\theta \mid \theta \text{ is a valid substitution for } C \text{ and } var(C\theta) = \emptyset\}$

∎

Definition 2.2.2. (Transfer function). Let T be a mapping $2^D \to 2^D$, we define $T \uparrow 0 = \bot$ and $T \uparrow i + 1 = T(T \uparrow Ti)$. We also define $T \uparrow \infty$ as $\bigcup_{i<\infty} T \uparrow i$.

∎

Theorem 2.2.1. ***(Fixpoint characterization of the least Herbrand model).*** *Let P be a program, then $lfp(T_P) = T_P \uparrow \infty = H_P$ where lfp is the least fixpoint and H_P is the least Herbrand model of P.*

Proof. Proved by Van Emdem and Kowalski in [115].

2.2.2 Operational Semantics

The *operational semantics* of a logic program is based on a *top-down* (or 'goal oriented') resolution and is named *SLD*-resolution. This SLD-resolution can be defined by the following algorithm where P is a logic program and Q is a goal:

```
SLD(P,Q)
1: Initialize the set $R$ to be $\{Q\}$
2: while $R \neq \emptyset$
3:     Take a literal $A$ in $R$
4:     Choose a renamed clause $A' \leftarrow B_1, ..., B_n$ from $P$, such that $A$ and $A'$
       unify with unifier $\theta$.
5:     if no such clause can be found then
           return FAIL; explore another branch.
6:     else
7:         Remove $A$ from $R$, add $B_1, ..., Bn$ to $R$
8:         $R \leftarrow R\theta$
9:         $Q_\theta \leftarrow Q\theta$
10: if $R = \emptyset$ then
11:     return $Q_\theta$ and SUCCEED
```

Note that lines 3 and 4 do not specify the ordering of clauses within the program, and the ordering of the goals in the bodies of the clauses, respectively. Different logic programming systems may define different strategies for each case. In this thesis, we concentrate on *Prolog* (PROgramming in LOGic). It was the first practical logic programming language and it still is the most widely used and efficiently implemented today. It was devised by the group led by A. Colmenauer at the U. of Marseille. They chose for Prolog an extremely simple *implicit control strategy.* The following two rules determine Prolog's *control strategy*:

- *Search rule*, line 3: given a goal, the first clause whose head unifies with the goal, scanning from top to bottom of the program, is selected. Then the goals in the body of the clause are executed in the order determined by the *computation rule* below. If the choice does not lead to a solution (i.e. it leads to FAIL), all resolution steps and variable substitutions (i.e. all 'bindings') done since the last such choice are undone, the next clause whose head matches with the goal is selected, and execution continues from there. This technique is called *backtracking.*

- *Computation rule*, line 4: once a clause is selected (using the *search rule* above), the goals in the body of the clause are executed one by one in left-to-right order.

Definition 2.2.3. (Success set). The *success set* is the set of all grounds atoms $Succ_P = \{b \mid \mathsf{SLD}(P, Q) = b,\ Q\ is\ a\ goal\}$. Then, $Succ_P$ corresponds to the least Herbrand model of P.

∎

2.3 Unification

In the SLD-resolution algorithm explained above, we omitted deliberately one of the its basic mechanisms called *unification* and defined by Alan Robinson [103] (line 4). Two atoms $p_a(ta_1, ..., ta_m)$ and $p_b(tb_1, ..., tb_n)$ are said to be *unifiable*, if they have identical predicate symbols (i.e., $p_a = p_b$), they have the same arity (i.e., $n = m$), and all their terms are pairwise (i.e., ta_1 vs. tb_1, ta_2 vs. tb_2 etc.) unifiable. Two terms, ta and tb are unifiable if the following recursive algorithm succeeds for them:

1. **if** ta is a variable which appears in tb FAIL[1]; **else**
2. **if** ta is a variable, and tb is not, then SUCCEED, and substitute tb for all occurrences of ta; **else**

[1]This "check" (referred to as the *occurs check*) is sometimes omitted in practical implementations because of the overhead involved in performing it.

3. **if** both *ta* and *tb* are variables, then SUCCEED, keeping them as variables, but giving them the same name. These variables are said to *share*: if a substitution is done for one of them it will also be done for the other; **else**

4. **if** *ta* is a constant then, **if** *tb* is a constant and both constants are identical, SUCCEED, **else** FAIL; **else**

5. *ta* is a structure (compound term); then, **if** *tb* is also a structure, they have identical functors and arity, and all their respective terms are unifiable (using this algorithm recursively), SUCCEED; **else**

6. FAIL.

2.4 Non-Determinism

At this point, it should be fairly clear from all descriptions given above that there are two distinct components during the execution of a logic program:

1. The program P, i.e., the set of rules and facts, provided by the user (including the *query goal*, Q).

2. An evaluator of the program, which is in charge of answering the query using the SLD-resolution algorithm given above.

It should also be clear from that description that there are two occasions, lines 3 and 4 in the SLD-resolution algorithm, in which the next step to be taken by the program evaluator is not uniquely determined. This is the origin of the two basic types of non-determinism present in Logic programs [66]:

- *non-determinism*$_1$: if several clause heads unify with the selected goal, the *policy* used by the program evaluator for performing this selection is called the search rule. The search rule also determines whether the remaining choices will also be eventually tried or not. This results in two subtypes of *nondeterminism*$_1$:

  - *'Don't care' non-determinism*$_1$: once a choice is made the system *commits* to that choice.
  - *'Don't know' non-determinism*$_1$: more than one of the possible choices may eventually be tried in the search for a solution.

- *non-determinism*$_2$: if the current query goal contains several goals (procedure calls) the policy used by the program evaluator for performing this selection is called the computation rule.

It is important to note that modifying the search rule affects the order and number of *solutions* which can be obtained from the system: although SLD-resolution does not impose a particular order in the choices made by the search rule, *completeness* (i.e., the guarantee of finding all possible solutions) is only preserved if a *fair rule* is chosen, i.e., one which will ensure that all possible paths in the *search space* will eventually be explored. Systems which use only "don't care" non-determinism$_1$ are therefore *incomplete* (also, they can only provide at most one solution path for a given query goal). Systems which use 'don't know' non-determinism$_1$ can provide more than one solution to a given query. Their degree of completeness depends on the type of search rule being used. Since most computation rules are *exhaustive* (i.e., they will eventually invoke all goals in the body of a clause) the choice of one or another will only affect the behavior of the system, but not the number of solutions found.

2.5 Modes in Logic Programming

One of the distinctive features of logic programs is that predicates can run in different modes, i.e., there is no priori notion of input and output. This allows a form of code reuse that is not available (or supported) in other programming languages. For instance, consider the quicksort algorithm[2] implemented by the following `qsort/2` program:

```
qsort([],[]).
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).

partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E .<. C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]):- append(X,Y,Z).
```

This program can be used to answer questions of different kinds:

- Given an arbitrary list, return all its elements sorted:

[2]It is written using *Constraint Logic Programming* [62] to avoid an instantiation error during the execution of `</2` when its arguments are not instantiated.

```
| ?- qsort([2,1,4,3],L).
  L = [1,2,3,4] ?
  yes
```

- but also, given a sorted list, return all its possible permutations:

```
| ?- qsort(L,[1,2,3,4]).
  L = [1,2,3,4] ;
  L = [1,2,4,3] ;
  ...
```

These different ways of using a logic program are usually referred to by saying that the programs is used in different *modes*. Mode information is important mainly for compiler optimizations. *Mode analysis* deals with analyzing the possible modes in which a predicate may be called within a particular program in order to obtain information that may be useful for specializing the predicate and thus helping the compiler to implement it more efficiently.

In this thesis, mode information is also essential since it has a deep impact on the correctness of the resource usage analysis for logic programs described in Chapter 4. In particular, the same predicate with different modes may have different complexities. For instance, suppose we would like to infer an upper bound of the number of resolution steps during the execution of `qsort/2`. If we run the program with the first argument instantiated to a list, then the upper bound on the number of steps is $O(n^2)$ where n is the length of the list. However, if the first argument is free (not instantiated) and the second argument is a sorted list, then the upper bound on the number of resolution steps is factorial.

Therefore, a precise inference of predicate modes is very important for the resource usage analysis. In Chapters 6 and 7, we will propose two different analyses that can be used for inferring precise mode information with special emphasis on efficiency.

2.6 The Ciao Prolog System

In this section, we provide a brief description of the Prolog system used in this thesis, `Ciao`, and its preprocessor, `CiaoPP`, that contains a number of program analyzers, and into which all the analyses presented in this thesis have been integrated. Finally, we also describe a subset of the assertion language used in `CiaoPP` that will be necessary to understand the Prolog examples shown in this thesis.

`Ciao` [21, 27, 53] is a multiparadigm programming language with an advanced programming environment that relies on a high-performance Prolog-based engine. Its modular approach allows both restricting and augmenting the language through libraries in a well-controlled fashion. This allows providing significant extensions which make `Ciao` a next-generation logic-programming language as well as a multiparadigm programming system. These advanced features together with the capabilities already known of standard Prolog engines persuaded us to choose `Ciao` as the main program development system in this thesis.

`CiaoPP` [52, 54], the preprocessor of the `Ciao` system, is a novel programming framework which uses extensively abstract interpretation as a fundamental tool in the program development process to obtain information about the program. Then, this information is used to verify programs, to detect bugs with respect to partial specifications written using assertions (in the program itself and/or in system libraries), to generate run-time tests for properties which cannot be checked completely at compile-time and simplify them, and to perform high-level program transformations such as multiple abstract specialization, parallelization, and resource usage control, all in a provably correct way. The usage of `CiaoPP` in this thesis is twofold. On one hand, the tools available in `CiaoPP` such as efficient and precise fixpoint algorithms, static analysis algorithms, abstract verification code, etc. are taken as starting point for the analyses developed in this thesis; on the other hand, in this thesis we provide new analyses which have been integrated into the preprocessor.

One of these advanced features is the assertion language [22, 98] in which (partial) specifications are written for program validation and debugging. Such assertions are simply linguistic constructions which allow expressing properties of programs. One of the most useful characteristics of the assertions used in `CiaoPP` is that they may be used in different contexts and for many different purposes. First, any assertions present in programs can be processed by an autodocumenter (`lpdoc` [50]) in order to generate useful documentation. Also, assertions are used as *specifications* which are then compared by `CiaoPP` interactively during program development with the results of analysis in order to *find bugs* statically, *verify* that the program complies with the assertions, or even generate automatically proofs of correctness that can be shipped with programs and checked easily at the receiving end (using the *proof/abstraction carrying code* approach [4]). Even if a program contains no user-provided assertions, `CiaoPP` can check the program against the assertions contained in the libraries used by the program, thus potentially catching additional bugs at compile time. For homogeneity, and to ease information exchange among the autodocumenter and the different checkers and analyzers, analysis results are reported using also the assertion language —which, since it is readable by humans, can be inspected by a programmer, for example to make sure that the results of the analyses agree with the intended meaning of the program.

Assertions also allow programmers to describe the relevant properties of modules or classes which are not yet written or are written in other languages. This is also done in other languages but often using different types of assertions for each purpose. In contrast in Ciao the same assertion language is used again for this task. This, interestingly, makes it possible to run checkers / verifiers / documenters against code which is only partially developed: the traditional "stubs", which have to be changed later on for a working version, can be replaced by an assertion declaring how the predicate should behave, with the advantage that this declared behavior can effectively be checked against its uses. Finally, assertions can be used to guide analysis when precision is lost.

It is beyond the scope of this thesis to present the complete assertion language. Instead, we concentrate on a subset of it which suffices for illustrating the main concepts involved in further chapters. The assertions that we will use adhere to the following schema:

' `:- pred` *Pred* `:` *PreCond* `=>` *PostCond* `+` *Comp_prop* `.`'

which should be interpreted as

> "for any call of the form *Pred* for which *PreCond* holds, if the call succeeds then on success *PostCond* should also hold."

Properties which refer to the whole computation of the predicate, rather than the input-output behavior can also be expressed by means of the *Comp_prop* field. These properties should be interpreted as

> "for any call of the form *Pred* for which *PreCond* holds, *Comp_prop* should also hold for the computation of *Pred*."

We will illustrate this subset of the assertion language with the following example that presents (part of) the previously introduced `Ciao` *quicksort* program implementing the algorithm for sorting lists in ascending order. Predicate `qsort/2` is annotated with a predicate assertion which expresses properties that the user expects should hold for the program.

```
:- pred qsort(A,B) : list(A) * var(B)
                  => list(A) * list(B)
                   + not_fail.
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).
```

In `qsort/2`, examples of *PreCond* and *PostCond* are type and instantiation declarations such as, e.g., `list(A)`, `list(B)`, and `var(B)`. `list(A)` denotes the variable `A` is instantiated to a polymorphic list. `var(B)` expresses that B is a free variable, i.e., unbounded variable. An example of a *Comp_prop* property is `not_fail` which expresses that if the predicate is called with the first argument instantiated to a list then the predicate should not fail. Another example of a *Comp_prop* property is the resource usage of a predicate which we will describe in Chapter 4. Note also that a property may be one out of a predefined set, including extra-logical properties such as, e.g., `var`, `gnd` (ground term), `atm` (atoms), etc, or, in principle, predicates defined by the user, using the full underlying logic programming language (but which must satisfy some properties such as, e.g., terminating for any possible call).

Finally, a predicate assertion may be extended to the following schema:

' `:- pred` *Tag Pred* `:` *PreCond* `=>` *PostCond* `+` *Comp_prop* `.`'

Having at most one of the following tags in front of the assertion:

- `check` used to mark the corresponding assertion as expressing an expected property of the final program (intended property).
- `true` indicates that the property holds for the program at hand (actual property).
- `trust`. The property holds for the program at hand. The difference with the above is that this information is given by the user and it may not be possible to infer it automatically.
- `checked`. A `check` assertion which expresses an intended property is rewritten with the status `checked` during compile-time checking when such property is proved to actually hold in the current version of the program for any valid initial query.
- `false`. Similarly, a `check` assertion is rewritten with the status `false` during compile-time checking when such property is proved not to hold in the current version of the program for some valid initial query. In addition, an error message will be issued by the preprocessor.

Chapter 3

Abstract Interpretation

Most program models are infinite such as e.g., the T_P semantics in Section 2.2.1. Thus, the least fixed point of T_P cannot be computed in general in a finite amount of time. Fortunately, there are some formal techniques that provide *safe approximations* (i.e., so that the success set of the program is included in the approximation) which are computable in finite time. In this section, we provide some background on *abstract interpretation*, a technique used for approximating the concrete semantics of programs in this thesis.

Abstract interpretation [31] is a theory of approximation of mathematical structures, in particular those involved in the semantic models of programs. The idea behind abstract interpretation is to *"mimic"* the execution of a program using an abstraction of the concrete semantics of the program to approximate undecidable or very complex properties. The abstraction of the semantics equipped with a structure (e.g., ordering) may involve a simpler abstraction of the data values that variables may take.

3.1 Definitions

Definition 3.1.1. (Partial ordered set, poset). A *partial order set* is a set and a binary relation such that the relation is:

1. *Reflexive*: $x \sqsubseteq x$ is in the relation.
2. *Transitive*: if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$.
3. *Antisymmetric*: if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$.

■

Definition 3.1.2. (Lower bound). Given a set S and T, a subset of S, $z \in S$ is a *lower bound* of T if and only if for all $x \in T$, $z \sqsubseteq x$. ■

Definition 3.1.3. (Upper bound). Given a set S and T, a subset of S, $z \in S$ is a *upper bound* of T if and only if for all $x \in T$, $x \sqsubseteq z$. ■

Definition 3.1.4. (Greatest Lower bound). Given a set S and T, a subset of S, z is the *greatest lower bound* of T if and only if:

1. z is a lower bound of T, and
2. for all x a lower bound of T, $x \sqsubseteq z$.

■

Definition 3.1.5. (Least Upper bound). Given a set S and T, a subset of S, z is the *least upper bound* of T if and only if:

1. z is an upper bound of T, and
2. for all x an upper bound of T, $z \sqsubseteq x$.

■

Definition 3.1.6. (Chain). For a poset T, a subset $S \subseteq T$ is a *chain* if for all $s_1, s_2 \in S$ then $s_1 \sqsubseteq s_2$ or $s_2 \sqsubseteq s_1$. ■

Definition 3.1.7. (Ascending Chain Condition). A poset S has the *Ascending Chain Condition* if every ascending chain $s_1 \sqsubseteq s_2 \sqsubseteq \ldots$ of elements in S is eventually stationary. A chain is *stationary* if there exists an $n \in \mathbb{N}$ such that $s_m = s_n$ for all $m > n$. ■

Definition 3.1.8. (Completely partially ordered set, cpo). A completely partially ordered set S, also called a *complete lattice*, is a poset with the further restrictions that:

1. Every subset of S has a unique greatest lower bound.
2. Every chain of S has a unique least upper bound.

■

3.2 Galois Connections

An abstract semantic object is a finite representation of a, possibly infinite, set of actual semantic objects in the concrete domain (D). The set of all possible abstract semantic values represents an *abstract domain* (D_α) which is usually a complete lattice or cpo which is ascending chain finite. In this thesis, we restrict ourselves to complete lattices over sets both for the concrete $\langle D, \sqsubseteq \rangle$ and abstract $\langle D_\alpha, \sqsubseteq \rangle$ domains. An *abstraction function* describes elements of D in terms of elements in D_α:

$$\alpha : D \mapsto D_\alpha$$

Similarly, a *concretization function* defines the mapping from elements of D_α to D:

$$\gamma : D_\alpha \mapsto D$$

The concrete and abstract domains are related by *Galois Connections*.

Definition 3.2.1. (Galois Connection). $\langle D, \alpha, \gamma, D_\alpha \rangle$ is a *Galois Connection* between the lattices $\langle D, \sqsubseteq \rangle$ and $\langle D_\alpha, \sqsubseteq \rangle$ if and only if:

1. α and γ are monotonic.
2. $\forall x \in D : \ \gamma(\alpha(x)) \sqsupseteq x$
3. $\forall y \in D_\alpha : \ \alpha(\gamma(y)) \sqsubseteq y$

■

Definition 3.2.2. (Galois Insertion). A *Galois Insertion* is a Galois Connection satisfying: $\forall y \in D_\alpha : \ \alpha(\gamma(y)) = y$. Therefore, $\langle D, \alpha, \gamma, D_\alpha \rangle$ is a Galois Insertion between the lattices $\langle D, \sqsubseteq \rangle$ and $\langle D_\alpha, \sqsubseteq \rangle$ if and only if:

$$\forall x \in D : \ \gamma(\alpha(x)) \sqsupseteq x \quad \text{and} \quad \forall y \in D_\alpha : \ \alpha(\gamma(y)) = y. \tag{3.1}$$

■

The abstract domain D_α is usually constructed with the objective of computing approximations of the semantics of a given program. Thus, all operations in the abstract domain also have to abstract their concrete counterparts. In particular, if the semantic operator S_P can be decomposed in lower level operations, and their abstract counterparts are locally correct w.r.t. them, then an abstract semantic operator S_P^α can be defined which is correct w.r.t. S_P. This means that $\gamma(S_P^\alpha(\alpha(x))$ is an approximation of $S_P(x)$ in D, and consequently, $\gamma(lfp(S_P^\alpha))$ is an approximation of the meaning of the program P, denoted by $[\![P]\!]$. We will denote $lfp(S_P^\alpha)$ as $[\![P]\!]_\alpha$. The fundamental theorem of abstract interpretation provides the following result:

Theorem 3.2.1. *Let $\langle D, \alpha, \gamma, D_\alpha \rangle$ be a Galois Insertion and let $S_P : D \mapsto D$ and $S_P^\alpha : D_\alpha \mapsto D_\alpha$ be monotonic functions such that $\forall x \in D : \ \gamma(S_P^\alpha(\alpha(x)) \sqsupseteq S_P(x)$, i.e., S_P^α approximates S_P. Then:*

$$\gamma([\![P]\!]_\alpha) \sqsupseteq [\![P]\!] \ \ \textit{equivalently} \ \ [\![P]\!]_\alpha \sqsupseteq \alpha([\![P]\!])$$

i.e., $[\![P]\!]_\alpha$ approximates $[\![P]\!]$.

Proof. Proved by P.Cousot and R.Cousot in [31].

Therefore, the art of abstract interpretation can be described as involving the following steps:

1. Choose an appropriate concrete semantics.
2. Provide good approximations of the basic operations in the concrete semantics.
3. Compute the abstract least fixed point.

In practice, the abstract domains should be sufficiently simple to allow effective computation of semantic approximations of programs. For example, Herbrand interpretations of some alphabet may be mapped into an abstract domain where each element represents a typing of predicates in some type system. For a given program P the abstract operator S_P^α would allow then to compute a typing of the predicates in the least Herbrand model of P.

Example 3.2.1. A simple example of abstract interpretation in logic programming can be constructed as follows. The concrete semantics (least Herbrand model) of a program P is $[\![P]\!] = lfp(T_P)$. So the concrete domain is $D = \wp(B_P)$ (where B_P is the Herbrand base of the program).

We consider over-approximating the set of "succeeding predicates", i.e., those whose predicate symbols appear in $[\![P]\!]$. A possible abstraction is as follows. The abstract domain is $D_\alpha = \wp(B_P^\alpha)$, where B_P^α is the set of predicate symbols of P. Let $pred(A)$ denote the predicate symbol for an atom A. We define the abstraction function:

$$\alpha : D \rightarrow D_\alpha \text{ such that } \alpha(I) = \{pred(A) \mid A \in I\}.$$

Similarly, the concretization function is defined as:

$$\gamma : D_\alpha \rightarrow D \text{ such that } \gamma(I_\alpha) = \{A \in B_P \mid pred(A) \in I_\alpha\}.$$

For example,

$$\alpha(\{p(a,b), p(c,d), q(a), r(a)\}) = \{p/2, q/1, r/1\}$$
$$\gamma(\{p/2, q/1\}) = \{p(a,a), p(a,b), p(a,c), \ldots, q(a), q(b), \ldots\}.$$

Note that $\langle D_\alpha, \alpha, \gamma, D\rangle$ is a Galois Insertion. The abstract semantic operator $T_P^\alpha : D_\alpha \rightarrow D_\alpha$ is defined as:

$$T_P^\alpha(I_\alpha) = \{pred(A) \mid \exists\, (A \leftarrow B_1, \ldots, B_n) \in P\ \forall i \in [1,n] : pred(B_i) \in I_\alpha\}.$$

Since D_α is finite and T_P^α is monotonic, the analysis (applying T_P^α repeatedly until fixpoint, starting from $\emptyset$) will terminate in a finite number of steps n and $[\![P]\!]_\alpha = T_P^\alpha \uparrow n$ approximates $[\![P]\!]$. For example, for the following program P,

```
p(X,Y) :- q(X), r(Y).
t(X) :- l(X).
m(X) :- s(X).
q(a). q(b).
r(a). r(c). r(X).
```

we have $B_P^\alpha = \{p/2, q/1, r/1, s/1, t/1, l/1, m/1\}$, and:

$$\begin{aligned}
&T_P^\alpha \uparrow 0 = \bot \\
&T_P^\alpha \uparrow 1 = T_P^\alpha(\bot) = \{q/1, r/1\} \\
&T_P^\alpha \uparrow 2 = T_P^\alpha(\{q/1, r/1\}) = \{q/1, r/1, p/2\} \\
&T_P^\alpha \uparrow 3 = T_P^\alpha(\{q/1, r/1, p/2\}) = \{q/1, r/1, p/2\}
\end{aligned}$$

So $T_P^\alpha \uparrow 2 = T_P^\alpha \uparrow 3 = \{q/1, r/1, p/2\} = [\![P]\!]_\alpha$

3.3 Widening

Widening is a technique often used to approximate the least fixed point of a program. Widening can be used to construct chains that converge to a fixed point much faster than the direct application of a monotonic operator over complete lattices (such as the transfer function in Definition 2.2.2).

Widening is implemented through *widening operators*. How these operators are constructed will affect the precision of the approximated fixed point, and the computational cost of finding the approximation. In general, there is a trade-off between precision and efficiency in the process of accelerating the convergence of the fixpoint computations.

Definition 3.3.1. (Widening operator).

Let $l_1, l_2 \in \mathcal{L}$ be lattices, then an operator $\nabla : \mathcal{L} \times \mathcal{L} \to \mathcal{L}$ is a widening operator $l_1 \nabla l_2$ if:

1. **(soundness)** It is an upper bound operator: $l_1 \sqsubseteq l_1 \nabla l_2$ and $l_2 \sqsubseteq l_1 \nabla l_2$.
2. **(stationary)** For any increasing chain $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ the chain $b_0 = a_0, b_1 = b_0 \nabla a_1, \ldots, b_{i+1} = b_i \nabla a_{i+1}, \ldots$ is not strictly increasing for $\sqsubseteq$, that is, it should be a stationary sequence.

∎

3.4 Abstract Functions Required by Logic Programming-based Analysis Frameworks

In this section, we outline the two main approaches for analysis of logic programs based on abstract interpretation, and we also describe their core abstract operations. This background is required to understand Chapters 5, 6, and 7.

Different semantics definition styles lead to different approaches to program analysis. There are mainly two approaches in the analysis of logic programs: *top-down* and *bottom-up* analysis. The top-down (e.g., [117, 18, 88]) approach propagates the information in the same direction as *SLD*-resolution does. Alternatively, bottom-up (e.g., [81]) analyses propagate the information as in the computation of the least fixpoint of the immediate-consequences operator T_p. The main difference between the top-down and bottom-up approaches is related to *goal dependence*. The top-down analyses start with a particular (abstract) goal, and they are able to determine call pattern information, i.e., information about specific procedure calls. On the other hand, bottom-up analyses determine an approximation of the success set, which is goal independent.

In top-down frameworks, the analysis of a clause $Head$`:-`$Body$ proceeds as follows (we follow the description and –at a high level– the algorithm of [89, 88, 91]). There is a goal $Goal$ for the predicate of $Head$, which is called in a context represented by abstract substitution $Call$ on a set of variables (distinct from $vars(Head) \cup vars(Body)$) which contains those of $Goal$. Then the success of $Goal$ by executing the above clause is represented by abstract substitution $Succ$ given in Figure 3.1.

$$\begin{array}{lcl}
Succ & = & extend(Call, Goal, Prime) \\
Prime & = & exitToSucc(project(Head, Exit), Goal, Head) \\
Exit & = & entryToExit(Body, Entry) \\
Entry & = & augment(F, callToEntry(Proj, Goal, Head)) \\
Proj & = & project(Goal, Call)
\end{array}$$

Figure 3.1: Abstract functions required by top-down analyses

In Figure 3.1, F is any term with the variables $vars(Body) \setminus vars(Head)$. $Call$, $Prime$, and $Success$ are abstract substitutions (θ_α) of the form $\theta_\alpha = \{x_1 \mapsto d_1^\alpha \in D_\alpha, \ldots, x_n \mapsto d_n^\alpha \in D_\alpha\}$. The abstract functions in Figure 3.1 are described as:

- $extend(Call, Goal, Prime)$ makes the framework inter-procedural updating the caller's context, $Call$, with the callee's context, $Prime$, yielding a substitution for the success of $Goal$ when it is called in a context represented by substitution $Call$ on a set of variables which contains those of $Goal$, given that in such context the success of $Goal$ is already represented by substitution $Prime$ on the variables of $Goal$. The domain of the resulting substitution is the same as the domain of $Call$.

- $project(Goal, Call)$ removes all bindings ($x_i \mapsto d_i^\alpha$) of a substitution $Call = \{x_1 \mapsto d_1^\alpha, \ldots x_n \mapsto d_n^\alpha\}$ such that the variable $x_i, 1 \leq i \leq n$ does not appear in $Goal$.

- $augment(Goal, Call)$ extends the domain of an abstract substitution $Call$ to the variables of a given term $Goal$. Thus, for each variable $x_i \in vars(Goal)$, the binding $x_i \mapsto \top$ is added into $Call$, where $\top$ is the top abstract value.

- $entryToExit(Body, Entry)$ is given by the framework, and basically traverses the body of a clause, analyzing each atom in turn. The exit abstract substitution is the abstract substitution after the last literal in $Body$, starting the first literal with $Entry$.

- $callToEntry(Proj, Goal, Head)$ (*"procedure entry"*) yields a substitution on the variables of $Head$ which represents the effects of unification $Goal = Head$ in a context represented by substitution $Proj$ on the variables of $Goal$.
- $exitToSucc(Exit', Goal, Head)$ (*"procedure exit"*) yields a substitution on the variables of $Goal$ which represents the effects of unification $Goal = Head$ in a context represented by substitution $Exit'$ on the variables of $Head$.

All these operations need to be defined specifically for a given abstract domain. However, $callToEntry$ and $exitToSucc$ can be defined from the abstract unification operation $(unify)$[1] as follows:

$$\begin{array}{lcl} callToEntry(ASub, Goal, Head) & = & unify(ASub, Head, Goal) \\ exitToSucc(ASub, Goal, Head) & = & unify(ASub, Goal, Head) \end{array}$$

Given an operation $amgu(x, t, ASub)$ of abstract unification for equation $x = t$, where x is a variable, $t, t_1,$ *and* t_2 are terms, and $ASub$ an abstract substitution (the domain of which contains variables $vars(t) \cup \{x\}$), then abstract unification, $unify$, for equation $t_1 = t_2$, is given by:

$$unify(ASub, t_1, t_2) = project(t_1, Amgu(solve(t_1 = t_2), augment(t_1, ASub)))$$

$$Amgu(Eq, ASub) = \begin{cases} ASub & \text{if } Eq = \emptyset \\ Amgu(Eq', amgu(x, t, ASub)) & \text{if } Eq = Eq' \cup \{x = t\} \end{cases}$$

such that $solve(t_1 = t_2)$ denotes the solved form of unification equation $t_1 = t_2$, i.e., the left hand side of the equation is a variable.

Finally, in addition to these operations, top-down frameworks also require the definition of:

[1] $unify$ is also domain-dependent.

- $init(Goal)$ generates an initial abstract substitution, $Call = \{x_1 \mapsto \top, \ldots x_n \mapsto \top\}$, for all $x_i \in vars(Goal)$.
- $equivalence(ASub_1, ASub_2)$ succeeds if and only if $ASub_1$ is equivalent to $ASub_2$.
- $join(ASub_1, ASub_2)$ merges the two abstract substitutions $ASub_1$ and $ASub_2$.

In the case of bottom-up frameworks the analysis is simpler than top-down analyses since only the following operations are required: *init*, *equivalence*, *join*, *project*, *augment*, and *amgu*, and they can be defined as above.

Chapter 4

Resource Usage Analysis for Logic Programs

This chapter presents the foundations of one of the major components and main motivation of this thesis, a static analysis that infers both upper and lower bounds on the usage that a logic program makes of a set of user-definable resources. The inferred bounds will in general be functions of input data sizes. A resource in our approach is a quite general, user-defined notion which associates a basic cost function with elementary operations. The analysis then derives the related upper- and lower-bound resource usage functions for all predicates in the program. This chapter also presents an assertion language which is used to define both such resources and resource-related properties that the system can then check based on the results of the analysis. Finally, this chapter also shows some experimental evaluation with some concrete resources such as execution steps, bytes sent or received by an application, number of files left open, number of accesses to a database, number of calls to a predicate, heap memory usage, etc.

4.1 Motivation

The importance of inferring information about the costs of computations for a variety of applications is well recognized. These costs are usually related to execution steps and, sometimes, time or memory. We propose an analyzer which allows automatically inferring both upper and lower bounds on the usage that a logic program makes of *user-definable resources*. Examples of such user-definable resources are bits sent or received by an application over a socket, number of calls to a predicate, number of files left open, number of accesses to a database, energy consumption, monetary units spent, disk space used, etc., as well as the more traditional execution steps, execution time, or memory. We expect the inference of this kind of information to be instrumental in a variety of applications, such as resource usage verification and debugging, certification of resource consumption in mobile code, resource/granularity control in parallel/distributed computing, or resource-oriented specialization.

In our approach a *resource* is a user-defined, application-dependent notion which associates a basic cost function with elementary operations in the base language and/or to some predicates in libraries. In this sense, each resource is essentially a user-defined *counter*. The user gives a name (such as, e.g., `bits`) to the counter and then defines via assertions how each elementary operation in the program (e.g., unifications, calls to builtins, external calls, etc.) increments or decrements that counter. The use of resources obviously depends in practice on the sizes or values of certain inputs to programs or predicates. Thus, in the assertions describing elementary operations the counters may be incremented or decremented not only by constants but also by amounts that are *functions* of input data sizes or values. The objective of our approach is to statically derive from these elementary assertions and the program text functions that yield upper and lower bounds on the amount of those resources that each of the predicates in the program (and the program as a whole) will consume or provide. The input to these functions will also be the sizes or value ranges of the topmost input data to the program or predicate being analyzed.

The structure of the rest of this chapter is as follows. Section 4.2 provides more details about the size and resource usage functions inferred through a worked example. In the following, Section 4.3.1 first presents in details the assertion language proposed for defining resources and annotating elementary operations. Section 4.3.2 shows how size relationships among program variables are determined, and Sections 4.3.3 and 4.3.4 describe how the resource usage functions are inferred. Section 4.4 shows some experimental results with concrete resources. Section 4.5 compares our approach with respect to the current state of the art, and finally, Section 4.6 summarizes our conclusions.

```
:- pred client(Opts, IBuf, OBuf)
   : list(gnd) * list(byte) * var.

client([Host,Port],IBuf,OBuf) :-
   connect(Host,Port,Stream),
   exch_buffer(IBuf,Stream,OBuf),
   close(Stream).

exch_buffer([],_,[]).
exch_buffer([B|Bs],Id,[B0|Bs0]) :-
   exch_byte(B,Id,B0),
   exch_buffer(Bs,Id,Bs0).

:- head_cost(ub,bits,0).
:- literal_cost(ub,bits,0).
```

```
/* SOCKET LIBRARY */
:- trust pred connect(Host,Port,S)
    :  atm * num * var
    => atm * num * atm
    +  cost(ub,bits,0).

:- trust pred close(Stream)
    :  atm => atm
    +  cost(ub,bits,0).

:- trust pred exch_byte(B,Id,B0)
    :  byte * atm * var
    => byte * atm * byte.
    +  cost(ub,bits,8).
```

Figure 4.1: A simple client application.

4.2 Worked Example

Consider a client application written in `Ciao` in Figure 4.1 that sends a data buffer (a list of bytes) through a socket and receives another (possibly transformed) data buffer. In this section, we will provide an overview of our approach with that program.

A *resource* is a user-defined, application dependent notion which associates a basic cost function with some user-selected predicates in the program. This is expressed by adding annotations using our assertion language (Section 4.3.1) to the code. The objective of the analysis is to safely approximate the usage that the program makes of the resource. In the example, assume that we would like to obtain an upper bound on the number of bits received by the application that we will call `bits`. We assume that the program receives 8 bits each time that `exch_byte/3` is called. This fact is reflected by the user by adding the *Comp_prop* assertion `'cost(ub,bits,8)'` which will increment the counter associated with the upper-bound on the number of bits received by 8. Similarly, we assume that open and close socket connections (`connect/3` and `close/1`) do not imply any exchange of bits, as indicated by `'cost(ub,bits,0)'`. In addition, the types and modes of the socket operations must be given to the analysis by other analyses or by user-provided assertions. In this example, we assume that the analysis does not have access to the code of the socket operations and hence, the user provides this information using `trust` assertions. For now, we will omit deliberately the directives `':- head_cost(ub,bits,0)'` and `':- literal_cost(ub,bits,0)'`. The rest of this section describes the main steps applied by the analyzer to approximate the number of bits received of the program depicted in Figure 4.1.

Step 1: Size metrics and mode inference. In the first step, the approach needs to infer for each argument in the program the notion of size metrics. For instance, the length of a list, the depth of a term, the size of a term, etc. In addition, the analysis also needs to infer if each argument is input or output (i.e., the modes) in order to perform properly the size and resource usage analyses described in Sections 4.3.2, 4.3.3, and 4.3.4. Input/output and size metrics information can be required by the language (typed language), given by the user (via assertions), or, as in our implementation, inferred automatically via analysis. In the example this information is asserted by the user in case of the socket library (`connect/3`, `close/1`, and `exch_byte/3`) and inferred automatically from the program for the predicates `client/3` and `exch_buffer/3`.

Step 2: Inference of data dependencies and size relationships. In the second step, the analysis firstly yields *argument dependency graph*s for the clauses within a strongly connected component, through a dataflow analysis. These graphs will be used for inferring *size relationships* for each literal argument between the input and output head arguments of every clause. The goal of this phase is then to describe the sizes of each body or head argument in terms of the size of some input head argument. In the example, assume the `exch_buffer/3` predicate. The analysis will infer from the first clause that the size of the third argument is 0, i.e. empty list, if the first argument is also an empty list. We denote this size relationship by the equation:

$$\Psi^3_{exch\_buffer}(0, \_) = 0 \tag{4.1}$$

where $\Psi^3_{exch\_buffer}$ describes the size of the third argument for the predicate `exch_buffer/3`. The idea is that for each k-output argument of a predicate p, the analysis defines its size as a function Ψ^k_p which takes as arguments the sizes of the input head arguments of p. Note that the size of the third argument does not depend on the second argument. We denote this by using the don't care symbol '_'. Similarly, the analysis will infer from the second clause the following equation:

$$\Psi^3_{exch\_buffer}(x, \_) = \Psi^3_{exch\_buffer}(x-1, \_) + 1 \tag{4.2}$$

Since the clause is recursive the analysis describes the sizes using a symbolic expression (a recurrence equation). This symbolic expression means that the size of the third argument is one plus the resulting size of calling recursively the predicate `exch_buffer/3` where the size of the first input argument has been decreased by one. Finally, the recurrence equation system shown above (Equations 4.1 and 4.2) must be approximated by a recurrence solver in order to obtain a closed form solution. In this case, our analysis yields the solution:

$$\Psi^3_{exch\_buffer}(x, \_) = x$$

i.e. the size of the third argument is proportional to the size of the first argument.

Step 3: Resource usage analysis. In this step, the analysis will use the size metrics, modes, the data dependencies, and the size relationships inferred in previous steps, and also the user-defined resource-related assertions in order to infer a resource usage equation for each clause and further simplify the resulting obtaining upper/lower bound closed form solutions. The resource analysis will statically derive safe upper/lower bounds on the amount of resources that each of the predicates consumes or provides. The result given by our analysis for an upper bound on the number of bits received by `exch_buffer/3` in the case of the first clause is:

$$\mathsf{Cost}(exch\_buffer, ub, bits, \langle 0, \_\rangle) = 0$$

that is interpreted as "the upper bound of the number of bits received when the size of the first input argument is zero results zero". Note that the sizes of the input arguments is given by the tuple $\langle 0, \_\rangle$ where the size of second input argument is irrelevant since it does not affect the resource usage of the clause. Similarly, the analysis infers a resource usage equation for the second clause:

$$\mathsf{Cost}(exch\_buffer, ub, bits, \langle x, \_\rangle) = 8 + \mathsf{Cost}(exch\_buffer, ub, bits, \langle x-1, \_\rangle)$$

that is interpreted as "the upper bound of the number of bits received when the size of the first input argument is x results in a symbolic expression formed by 8 (i.e., the resource usage of `exch_byte/3`) plus the resource usage of the recursive call to `exch_buffer/3` in which the size of the first input argument has been decreased by one. Again, the size of the second input argument of `exch_buffer/3` is irrelevant. Finally, this equation system is solved by a recurrence solver, resulting in the closed form:

$$\mathsf{Cost}(exch\_buffer, ub, bits, \langle x, \_\rangle) = 8 \times x$$

Note that since we know from the user-defined assertions that `connect/3` and `close/1` receive no bits, then

$$\mathsf{Cost}(client, ub, bits, \langle \_, n\rangle) = 8 \times n$$

4.3 A Framework for Inference of Resource Usage

We can now describe in detail our framework, outlined in Section 4.2, for inferring upper and lower bounds on the usage that a program makes of a set of user-definable resources. Our basic approach is as follows. Given a predicate call p, let $\Phi(p, r, \overline{n})$ denote the exact units of resource r consumed or produced during the computation of p for a tuple of argument sizes $\overline{n}$. Note that, in general, the computation of $\Phi(p, r, \overline{n})$ will be undecidable or very complex. Therefore, an expression $\mathsf{Cost}(p,ap,r,\overline{n})$ is determined at compile-time that approximates $\Phi(p, r, \overline{n})$ with approximation ap (i.e., upper-bound or lower-bound). For assuring the correctness of our approach, we must always generate resource usage bounds functions such as $\mathsf{Cost}(p,ap,r,\overline{n})$ that hold the following conditions:

- If the analysis computes an upper-bound approximation, i.e., $ap = \texttt{ub}$, then:

$$\Phi(p, r, \overline{n}) \leq \mathsf{Cost}(p,\texttt{ub},r,\overline{n}) \tag{4.3}$$

- Conversely, if the analysis computes a lower-bound $ap = \texttt{lb}$, then:

$$\mathsf{Cost}(p,\texttt{lb},r,\overline{n}) \leq \Phi(p, r, \overline{n}) \tag{4.4}$$

Note that the analysis can always generate trivial upper and lower bounds, ∞ and $-\infty$, in those cases where it cannot infer resource equations or find a closed form. Of course, the analysis should infer bounds as precise as possible.

Certain program information is first automatically inferred by other abstract interpretation-based analyzers included in `CiaoPP` and then provided as input to the size and resource analysis:

1. Inference of modes, i.e., determine which arguments are input or output.
2. Inference of types for each predicate argument.
3. Inference of size metrics for predicate arguments based on the type information.
4. Inference of non-failure information, i.e., determine which predicates should fail.

The techniques involved in inferring this information are beyond the scope of this thesis —see, e.g., [54] and its references for some examples.

The size of an output argument in a predicate p call depends in general on the size of the input arguments in that call. For this reason, for each k-output argument we infer an expression which yields its size as a function of the input data sizes (i.e., Ψ_p^k). Argument sizes are described in terms of size metrics. Typical size metrics are the actual value of a number, the length of a list, the size (number of constant and function symbols) of a term, etc. To this end, and using the input-output argument information, argument dependency graphs are used to set up recurrence equations whose solution yields size relationships between input and output arguments of predicate calls. This information regarding argument sizes and other such as resource-related assertions are then used to set up another set of recurrence equations whose solution provides resource usage bound functions. Both the size and resource usage recurrence equations must be solved by a recurrence equation solver. Although the operation of such solvers is beyond the scope of the thesis our implementation does provide a table-based solver (an evolution of the solver of the Caslog system[36]) which covers a reasonable set of recurrence equations such as first-order and higher-order linear recurrence equations in one variable with constant and polynomial coefficients,divide and conquer recurrence equations, etc. In addition, the system allows the use of external solvers (such as, e.g. [12], Mathematica, Matlab, etc.). Note also that, since we are computing upper/lower bounds, it suffices to compute upper/lower bounds on the solution of a set of recurrence equations, rather than an exact solution. This allows obtaining an *approximate* closed form when the exact solution is not possible.

In further sections, we will describe each main component of our framework. In Section 4.3.1 we will first present the assertion language proposed for defining resources and annotating elementary operations. Section 4.3.2 shows how size relationships among program variables are determined, Section 4.3.3 describes how the resource usage bound functions are inferred, and finally, Section 4.3.4 shows how users can define resources using our assertion language.

4.3.1 The Resource Assertion Language

We start by describing the assertion schema. This language is used for describing resources and providing other input to the resource analysis, and is also the language in which the resource analysis produces its output. This assertion language is used additionally to state resource-related specifications which can then be proved or disproved based on the results of analysis following the scheme of [54], as already mentioned in Chapter 2, allowing finding bugs, verifying the program, etc.

The rules for the assertion language grammar are listed in Figure 4.2. In this grammar Var corresponds to variables written in the syntax for variables of the underlying logic programming language (i.e., normally non-empty strings of characters which start with a capital letter or underscore). Similarly, Num is any valid number and $Pred\_name$ any valid name for a predicate in the underlying programming language, normally non-empty strings of characters which start with a lower-case letter or are quoted. $State\_prop$ corresponds to other *state properties* such as modes and types, and $Comp\_prop$ stands for any other valid *computational property*, see [54] and its references.

Predicates can be annotated with zero or more assertions. These assertions can refer to properties of the execution states when the predicate is called (*PreCond*), properties of the execution states when the predicate terminates execution (*PostCond*), and properties which refer to the whole computation of the predicate (*Comp_prop*), rather than the input-output behavior, which herein will be used only for resource-related properties). The assertion schema that allows defining the *PreCond*, *PostCond*, and *Comp_prop* parts together in a compact way via `pred` assertions which was already described in Section 2.6. In addition, there may be a set of global `head_cost` and `literal_cost` declarations (i.e., directives), one for each resource and approximation direction. The *Res_name* fields determine which resource the assertion refers to. These *Res_name*s are user-provided identifiers which give a name to each particular resource that needs to be tracked. Resources do not need to be declared in any other way, i.e., the set of resources that the system is aware of is simply the set of such names that appear in assertions which are in the scope. The

$\langle \mathit{program\_assrt} \rangle$	::=	`:-` $\langle \mathit{status\_flag} \rangle$ $\langle \mathit{pred\_assrt} \rangle$.
	$\mid$	`:- head_cost(`$\langle \mathit{approx} \rangle$,Res_name,$\Delta^H$`).`
	$\mid$	`:- literal_cost(`$\langle \mathit{approx} \rangle$,Res_name,$\Delta^L$`).`
$\langle \mathit{status\_flag} \rangle$	::=	`trust` $\mid$ `check` $\mid$ `true` $\mid$ `checked` $\mid$ `false` $\mid$ ϵ
$\langle \mathit{pred\_assrt} \rangle$	::=	`pred` $\langle \mathit{pred\_desc} \rangle$ $\langle \mathit{pre\_cond} \rangle$ $\langle \mathit{post\_cond} \rangle$ $\langle \mathit{comp\_cond} \rangle$.
$\langle \mathit{pred\_desc} \rangle$	::=	Pred_name $\mid$ Pred_name($\langle \mathit{args} \rangle$)
$\langle \mathit{args} \rangle$	::=	Var $\mid$ Var, $\langle \mathit{args} \rangle$
$\langle \mathit{pre\_cond} \rangle$	::=	`:` $\langle \mathit{state\_props} \rangle$ $\mid$ ϵ
$\langle \mathit{post\_cond} \rangle$	::=	=> $\langle \mathit{state\_props} \rangle$ $\mid$ ϵ
$\langle \mathit{comp\_cond} \rangle$	::=	`+` $\langle \mathit{comp\_props} \rangle$ $\mid$ ϵ
$\langle \mathit{state\_prop} \rangle$	::=	`size`(Var,$\langle \mathit{approx} \rangle$,$\langle \mathit{sz\_metric} \rangle$,$\langle \mathit{arith\_expr} \rangle$) $\mid$ State_prop
$\langle \mathit{state\_props} \rangle$	::=	$\langle \mathit{state\_prop} \rangle$ $\mid$ $\langle \mathit{state\_prop} \rangle$, $\langle \mathit{state\_props} \rangle$
$\langle \mathit{comp\_prop} \rangle$	::=	`size_metric`(Var,$\langle \mathit{sz\_metric} \rangle$) $\mid$ $\langle \mathit{cost} \rangle$ $\mid$ Comp_prop
$\langle \mathit{comp\_props} \rangle$	::=	$\langle \mathit{comp\_prop} \rangle$ $\mid$ $\langle \mathit{comp\_prop} \rangle$, $\langle \mathit{comp\_props} \rangle$
$\langle \mathit{cost} \rangle$	::=	`cost`($\langle \mathit{approx} \rangle$,Res_name,$\langle \mathit{arith\_expr} \rangle$)
$\langle \mathit{approx} \rangle$	::=	`ub` $\mid$ `lb` $\mid$ `oub` $\mid$ `olb`
$\langle \mathit{sz\_metric} \rangle$	::=	`value` $\mid$ `length` $\mid$ `term_size` $\mid$ `depth` $\mid$ `void`
$\langle \mathit{arith\_expr} \rangle$	::=	$-$ $\langle \mathit{arith\_expr} \rangle$ $\mid$ $\langle \mathit{arith\_expr} \rangle$ `!` $\mid$ $\langle \mathit{quantifier} \rangle$ $\langle \mathit{arith\_expr} \rangle$
	$\mid$	$\langle \mathit{arith\_expr} \rangle$ $\langle \mathit{bin\_op} \rangle$ $\langle \mathit{arith\_expr} \rangle$
	$\mid$	`exp`($\langle \mathit{arith\_expr} \rangle$,$Num$) $\mid$ `log`(Num, $\langle \mathit{arith\_expr} \rangle$)
	$\mid$	Num $\mid$ $\langle \mathit{sz\_metric} \rangle$(Var)
$\langle \mathit{bin\_op} \rangle$	::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/`
$\langle \mathit{quantifier} \rangle$	::=	$\sum$ $\mid$ $\prod$

Figure 4.2: Syntax of the resource assertion language

$\langle \mathit{approx} \rangle$ fields state whether $\langle \mathit{arith\_expr} \rangle$ is providing an upper bound or a lower bound (with `oub` meaning it is a "big O" expression, i.e., with only the order information, and `olb` meaning it is an Ω asymptotic lower bound). For instance, given the upper and lower bounds $UB = 2 \times n + 5$ and $LB = 2^n + n^2 + 1$, the $O(UB) = n$ (`oub`) and $\Omega(LB) = 2^n$ (`olb`).

The first and most fundamental use of assertions in our context is to describe how the execution of some predicates increments or decrements the usage of the resources defined in the program. The purpose of analysis is then to infer the resource usage of all predicates in the program. The "`head_cost(`$\langle \mathit{approx} \rangle$, Res_name, Δ^H)" declarations are used to describe how predicates in general update the value for those resources that are applicable to predicate heads such as counting the number of arguments passed or total execution steps –see Section 4.3.3. The definition of $\Delta^{H}(cl\_head, arith\_expr) \rightarrow \mathcal{B}$ is provided by means a user-defined (or imported) predicate, written in the source language, and which will be called by the analyzer when the clause head is analyzed. This code gets loaded into the compiler in a similar way to, e.g., macro expansion code. The "`literal_cost`($\langle approx \rangle$, Res_name, Δ^{L})" declarations describe how predicate bodies update the value of certain resources which are applicable to body literals such as, for example, number of unifications. In this case, $\Delta^{L}(body\_lit, arith\_expr) \rightarrow \mathcal{B}$ is also user- (or library-)provided code which will be executed when the body literals of different predicates are analyzed. The actual resource usage bound functions for each builtin or external (e.g., defined in another language) predicate used in the program are provided by a kind of *Comp_prop* property expressed by "`cost`($\langle approx \rangle$, Res_name, $\langle arith\_expr \rangle$)". Additionally, size metrics ("`size_metric`(Var,$\langle sz\_metric \rangle$)") information can be provided by users if needed, but note that in practice size metrics can often be derived automatically from the inferred types. Finally, assertions can also be used, via the *PreCond* and *PostCond* fields, to declare relationships between the data sizes of the inputs and outputs of predicates ("`size`(Var,$\langle approx \rangle$,$\langle sz\_metric \rangle$,$\langle arith\_expr \rangle$)"), which may be needed by our analysis in case of external or builtins predicates.

Therefore, as mentioned in Section 4.1, users should describe how each predicate increments or decrements the counters associated with each resource by defining two assertions: `head_cost` and `literal_cost`. Additionally, for each builtin or external predicate *p* an assertion of type '`:- trust pred` *p* `:` *PreCond* `=>` *PostCond* `+` *Comp_prop* `.`' should be also defined. *PreCond* will contain typically type declarations, *PostCond* also type declarations and properties such as `size`. Finally, *Comp_prop* will include properties such as e.g., `size_metric` and `cost`. Optionally, users also can guide the analysis (i.e., improve its precision) by defining a similar schema for other predicates. In this case, the analysis will compute the most precise approximation between the information provided by the assertion and the information inferred by analysis.

4.3.2 Size Analysis

We will now explain the foundations behind the argument dependency-based method for inferring bounds on the sizes of output arguments in the head of a predicate as a function of the sizes of input arguments to the predicate. Besides this, as a result of the size analysis, we have bounds on the size of each input argument to body literals in a clause as a function of the size of the input arguments to the head of that clause. The size of the input arguments to body literals will be used later to infer functions which give bounds on the resource usage of body literals in terms of the sizes of the input arguments to the head. We adopt the approach of Debray et al. [36, 37] for the inference of upper bounds on argument sizes and [38] for lower bounds.

The size of an input is defined in terms of metrics. By *size metrics* we refer to a total function that, given a term, returns an arithmetic expression or an undefined value $\perp$, possibly in terms of other input argument sizes. One of the difference with respect to Debray's approach is that our analysis is parametric on size metrics, which can be defined by the user through `size_metric` and `size` assertions. For concreteness, several size metrics are defined in our system. We define here a $\mathsf{size}(\langle sz\_metric\rangle, t)$ operation which returns the size of a term t under the metric $\langle sz\_metric\rangle$ for those predefined metrics:

- If size metrics is the integer value and let $\ominus$ be an arithmetic operator ($+, -, \times, \ldots$) then:

$$\mathsf{size}(\mathit{value}, t) = \begin{cases} t & \text{if } t \text{ is an integer} \\ \ominus(\mathsf{size}(\mathit{value}, t_1), \ldots, \mathsf{size}(\mathit{value}, t_n)) & \text{if } t = \ominus(t_1, \ldots, t_n) \\ \perp & \text{otherwise.} \end{cases}$$

- If size metrics is the length of a list, then:

$$\mathsf{size}(\mathit{length}, t) = \begin{cases} 0 & \text{if t} = [\,] \\ 1 + \mathsf{size}(\mathit{length}, T) & \text{if t} = [H \mid T] \\ \perp & \text{otherwise.} \end{cases}$$

- If size metrics is the size of a term, then:

$$\mathsf{size}(\mathit{term\_size}, t) = \begin{cases} 1 & \text{if } t \text{ is a constant} \\ 1 + \sum_{i=1}^{n} \mathsf{size}(\mathit{term\_size}, t_i) & \text{if } t = f(t_1, \ldots, t_n) \\ \perp & \text{otherwise.} \end{cases}$$

- If size metrics is the depth of a term, then:

$$\mathsf{size}(\mathit{depth}, t) = \begin{cases} 0 & \text{if } t \text{ is a constant} \\ 1 + \mathsf{max}\{\mathsf{size}(\mathit{depth}, t_i)\} & \text{if } t = f(t_1, \ldots, t_n) \\ \perp & \text{otherwise.} \end{cases}$$

Some examples: $\mathsf{size}(\mathit{length}, [X, Y]) = 2$, $\mathsf{size}(\mathit{length}, [X|Y]) = \perp$, $\mathsf{size}(\mathit{value}, 3 + 7) = 10$, $\mathsf{size}(\mathit{term\_size}, f(g(a), b) = 4$, and $\mathsf{size}(\mathit{depth}, f(2, f(3, \mathit{nil}, \mathit{nil}), \mathit{nil}) = 2$.

Since our approach assumes the general case in which the input program is not normalized (i.e., functor and predicate symbols may have arguments that are not atoms or variables), sometimes we need to establish size relationships as the difference between the sizes of two terms. This relationship is provided by the function $\mathsf{diff}(\langle sz\_metric\rangle, t_1, t_2)$ operation, which returns an approximation of the difference between the size of t_1 and the size of t_2 under the metric $\langle sz\_metric\rangle$. We define it again for our predefined metrics:

- If size metrics is the integer value, then:

$$\mathsf{diff}(value, t_1, t_2) = \begin{cases} t_2 - t_1 & \text{if } t_1 \text{ and } t_2 \text{ are integers} \\ \perp & \text{otherwise.} \end{cases}$$

- If size metrics is the length of a list, then:

$$\mathsf{diff}(length, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \mathsf{diff}(length, t, t_2) - 1 & \text{if } t_1 = [\_|t] \text{ for some term t} \\ \mathsf{diff}(length, t_1, t) + 1 & \text{if } t_2 = [\_|t] \text{ for some term t} \\ \perp & \text{otherwise.} \end{cases}$$

  For instance, $\mathsf{diff}(length, [X|Xs], Xs) = -1$ and $\mathsf{diff}(length, Ys, [Y|Ys]) = 1$.

- If size metrics is the size of a term, then:

$$\mathsf{diff}(term\_size, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ (\mathsf{sz}(t_2(i)) - \mathsf{size}(term\_size, t_1)) + 1 & \text{if } t_1 = f(s_1, \ldots, s_n) \\ & s_i \equiv t_2, \exists i, 1 \leq i \leq n \\ (\mathsf{sz}(t_1(i)) + \mathsf{size}(term\_size, t_2)) - 1 & \text{if } t_2 = f(s_1, \ldots, s_n) \\ & s_i \equiv t_1, \exists i, 1 \leq i \leq n \\ \perp & \text{otherwise.} \end{cases}$$

  where $\mathsf{sz}(t(i))$ is a symbolic expression that represents the size of the i-th argument position of the t term. For example, $\mathsf{diff}(term\_size, g(X,Y), Y) = \mathsf{sz}(g(2)) - 2$, and similarly, $\mathsf{diff}(term\_size, B, h(A,B,C)) = \mathsf{sz}(h(2)) + 3$.

- If size metrics is the depth of a term, then:

$$\mathsf{diff}(depth, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \mathsf{max}\{\mathsf{diff}(depth, s_i, t_2)\} - 1 & \text{if } t_1 = f(s_1, \ldots, s_n) \\ \mathsf{max}\{\mathsf{diff}(depth, s_i, t_1)\} + 1 & \text{if } t_2 = f(s_1, \ldots, s_n) \\ \perp & \text{otherwise.} \end{cases}$$

  where the maximum between $\perp$ and a number n is n. For example, $\mathsf{diff}(depth, tree\text{-}(X, Left, Right), Right) = \mathsf{max}\{\perp, \perp, 0\} - 1 = -1$, and conversely, $\mathsf{diff}(depth, Left\text{-}, tree(X, Left, Right)) = \mathsf{max}\{\perp, 0, \perp\} + 1 = 1$.

Definition 4.3.1. (Argument dependency graph).. An argument dependency graph $G = (V, E)$ is a directed acyclic graph such that V denotes argument positions of a clause, and there is an edge from a node n_1 to a node n_2, $(n_1, n_2) \in E$ if the variable bindings generated by n_1 are used to construct the term occurring at n_2. ■

Argument dependency graphs are used to represent the data dependency between argument positions in a clause body, and between them and those in the clause head.

Definition 4.3.2. (Predecessor, predec). Let $G = (V, E)$ be an argument dependency graph, and (n_1, n_2) and edge of E, then the node n_1 is said to be a *predecessor* of the node n_2. We will assume a predec function that takes an argument dependency graph, a literal, and a parameter position and returns its nearest predecessor in the graph. ■

Figure 4.3 shows the argument dependency graph of `exch_buffer/3` for its recursive clause. Solid rectangles and arrows denote input arguments and dependencies between input arguments, respectively. Similarly, dashed rectangles and arrows represent output arguments and dependencies between input and output arguments.

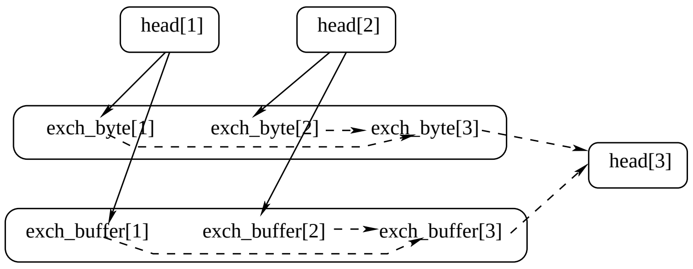


Figure 4.3: Argument dependency graph for the recursive clause of `exch_buffer/3`

Using the size and diff functions and the argument dependency graph for every clause, the analysis will traverse each strongly-connected component in reverse topological order in order to set up size relations for expressing the size of each argument position in terms of the sizes of its predecessors for every clause. Let $\mathsf{sz}(i)$ denote the size of the term occurring at an argument position i. For convenience, we will omit the argument $\langle sz\_metric \rangle$ in the size and diff functions in the rest of the chapter. Then, the size relationships can be obtained as follows:

- *Output arguments.* Let $i_1, \ldots, i_n$ denote the input argument positions of the predicate p, and let Ψ_p^k be a function that represents the size of the k-th (output) argument position of the predicate p in terms of the size of its input argument positions $i_1, \ldots, i_n$. Then the following size relation is set up:

$$\mathsf{sz}(k) \leq \Psi_p^k(\mathsf{sz}(i_1), \ldots, \mathsf{sz}(i_n)) \tag{4.5}$$

  1. If p is a recursive literal defined in the body of a clause, then $\Psi_p^k(\mathsf{sz}(i_1), \ldots, \mathsf{sz}(i_n))$ is a symbolic expression. For instance, the size of the output argument of the recursive call of `exch_buffer/3` in the second clause should be defined as $\Psi^3_{exch\_buffer}(\mathsf{sz}(exch\_buffer_1), \mathsf{sz}(exch\_buffer_2))$.
  2. Otherwise, if p is a non-recursive literal in the body of a clause then the function Ψ_p^k has been recursively computed, and thus we replace

$\Psi_p^k(\mathsf{sz}(i_1), \ldots, \mathsf{sz}(i_n))$ by the (explicit) expression resulting from the application of the function Ψ_p^k to $\mathsf{sz}(i_1), \ldots, \mathsf{sz}(i_n)$. For example, the size of the output argument of the call to `exch_byte/3` should be defined as $\Psi^3_{exch\_byte}(\mathsf{sz}(exch\_byte_1), \mathsf{sz}(exch\_byte_2))$, and this equation can be simplified to 1 (given by user assertion). Therefore, we can obtain a closed form solution in that case.

- *Input arguments.* Assume now that i is an input argument position in a body literal l in a clause C, and i' the term occurring at an argument position i. Let G be also the argument dependency graph of C. We have the following possibilities:

  1. Compute $\mathsf{size}(i')$. If $\mathsf{size}(i') \neq \perp$ then set up the size relation:

     $$\mathsf{sz}(i) \leq \mathsf{size}(i') \tag{4.6}$$

  2. Let $r = \mathsf{predec}(G, l, i)$, if the size metrics corresponding to r and i are the same and $d = \mathsf{diff}(r, i) \neq \perp$, then set up the size relation

     $$\mathsf{sz}(i) \leq \mathsf{sz}(r) + d \tag{4.7}$$

  3. Otherwise, $\mathsf{sz}(i) = \perp$.

Size relations can be propagated to transform a size relation corresponding to an input argument in a body literal or an output argument in the clause head into a function in terms of the sizes of the input arguments of the head. The basic idea here is to repeatedly substitute size relations for body literals into size relations for head arguments. This is the purpose of the normalization algorithm described in [36].

Example 4.3.1. Consider again the program described in Figure 4.1. We will denote by *pred_name* the name of a predicate, and by $pred\_name_i$ the i-th argument position

Size relation equations for first clause of `exch_buffer/3`:			
$1:$	$\mathsf{sz}(head_1)$	$\leq$	$\mathsf{size}([\,])\ and\ \mathsf{size}([\,]) = 0$
$2:$	$\mathsf{sz}(head_2)$	$\leq$	∞
$3:$	$\mathsf{sz}(head_3)$	$\leq$	$\mathsf{size}([\,])\ and\ \mathsf{size}([\,]) = 0$
Size relation equations for second clause of `exch_buffer/3`:			
$4:$	$\mathsf{sz}(exch\_byte_1)$	$\leq$	$\mathsf{size}(B)\ and\ \mathsf{size}(B) = 1$
$5:$	$\mathsf{sz}(exch\_byte_2)$	$\leq$	$\mathsf{sz}(head_2) + \mathsf{diff}(Id, Id)$
		$\leq$	$\mathsf{sz}(head_2)$
$6:$	$\mathsf{sz}(exch\_byte_3)$	$\leq$	$\Psi^3_{exch\_byte}(\mathsf{sz}(exch\_byte_1), \mathsf{sz}(exch\_byte_2))$
		$\leq$	$\Psi^3_{exch\_byte}(1, \mathsf{sz}(head_2))$
		$\leq$	1
$7:$	$\mathsf{sz}(exch\_buffer_1)$	$\leq$	$\mathsf{sz}(head_1) + \mathsf{diff}([B\|Bs], Bs)$
		$\leq$	$\mathsf{sz}(head_1) - 1$
$8:$	$\mathsf{sz}(exch\_buffer_2)$	$\leq$	$\mathsf{sz}(head_2) + \mathsf{diff}(Id, Id)$
		$\leq$	$\mathsf{sz}(head_2)$
$9:$	$\mathsf{sz}(exch\_buffer_3)$	$\leq$	$\Psi^3_{exch\_buffer}(\mathsf{sz}(exch\_buffer_1), \mathsf{sz}(exch\_buffer_2))$
		$\leq$	$\Psi^3_{exch\_buffer}(\mathsf{sz}(head_1) - 1, \mathsf{sz}(head_2))$
$10:$	$\mathsf{sz}(head_3)$	$\leq$	$\mathsf{sz}(exch\_buffer_3) + \mathsf{diff}(Bs0, [B0\|Bs0])$
		$\leq$	$\Psi^3_{exch\_buffer}(\mathsf{sz}(head_1) - 1, \mathsf{sz}(head_2)) + 1$
Closed form for the output argument of the head:			
$11:$	$\Psi^3_{exch\_buffer}(0, y)$	$=$	0
$12:$	$\Psi^3_{exch\_buffer}(x, y)$	$=$	$\Psi^3_{exch\_buffer}(x - 1, y) + 1$
$13:$	$\Psi^3_{exch\_buffer}(x, y)$	$=$	x

Figure 4.4: Size relation equations for `exch_buffer/3`

in literal with predicate name *pred_name* in the body of a clause. Let $head_i$ denote the i-th argument position in the clause head.

The Figure 4.4 shows all equations needed to establish the size of the output argument in `exch_buffer/3`. First, the system sets up its size in the first clause. Since this argument (third) is an empty list, its size based on the *length* metrics is zero. Note also that it is straightforward for the analysis to infer the sizes of the first and second input arguments. The size of the first argument is zero because it is also an empty list, and the size of the second argument is infinite since the variable is unbounded. The next step is the size inference of the output argument for `exch_buffer/3` in the second clause. In this case, the size of the third argument of the head depends on the size of some body literal argument. Thus, the system needs first to determine the size relations for each input and output argument of the body literals (inequalities $4 - 10$):

4 By Equation 4.6.

5 By Equation 4.7.

6 By Equation 4.5. Note that `exch_byte/3` is not recursive and has been previously computed. In particular, the size of its third argument has been given by the user through assertions.

7 By Equation 4.7.

8 By Equation 4.7.

9 By Equation 4.5. In this case, there is a recursive call to `exchange_buffer/3`. Thus, the size of the third argument is a symbolic expression which will be further solved by the recurrence solver.

10 By Equation 4.7.

Note that the size of the output argument of `exch_buffer/3` can be denoted by $\mathtt{sz}(head_3)$. Note also that since it is an output argument, that expression is equivalent to $\Psi^3_{exch\_buffer}$. Therefore, the system can then establish the recurrence system formed by inequalities 11 and 12, where x and y represent the sizes of the first and second input argument, respectively. In the next step, the system obtains a closed form function (inequality 13) by calling the recurrence equation solver.

Finally, it is important to notice that although the main objective of this method is to infer the sizes of each output argument in a clause head. By construction, the method can also infer the sizes of each body literal argument, i.e., size relationships at each program point. This information will be also required by the resource usage analysis in the next section.

4.3.3 Resource Usage Analysis

In order to infer the resource usage functions all predicates in the program are processed in a single traversal of the call graph in reverse topological order. Consider such a predicate p defined by clauses $C_1, \ldots, C_m$. Assume that $\overline{n}$ is a tuple such that each element corresponds to the size of an input argument position to predicate p. Then, the resource usage expressed in units of resource r with approximation ap of a call to p, for an input of size $\overline{n}$, can be expressed as:

$$\mathsf{Cost}_{pred}(p, ap, r, \overline{n}) = \bigodot(ap)_{1 \leq i \leq m}\{\mathsf{Cost}_{clause}(C_i, p, ap, r, \overline{n})\} \tag{4.8}$$

where $\bigodot(ap)$ is a function that takes an approximation identifier ap and returns a function which applies over all $\mathsf{Cost}_{clause}(C_i, p, ap, r, \overline{n})$, for $1 \leq i \leq m$. For example, if ap is the identifier for approximation "upper bound" (`ub`), then a possible conservative definition for $\bigodot(ap)$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will be demanded is generally not known in advance, a conservative upper bound on the computational cost of a predicate is obtained by assuming that all solutions are needed, and that all clauses are executed, thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses. However, the analysis can take mutual exclusion into account, which is inferred by `CiaoPP` and is available to our analysis, to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses.[1]. If *ap* is the identifier for approximation "lower bounds" (`lb`), then $\bigodot(ap)$ is the *min* function.

Let us see now how to compute the resource usage of clauses. Consider a clause C of predicate p of the form $H\ :-\ L_1, \ldots, L_k$ where L_j, $1 \leq j \leq k$, is a literal (either a predicate call, or an external or builtin predicate), and H is the clause head. Assume that $\overline{n}_j$ is a tuple with the sizes of all the input arguments to literal L_j, given as functions of the sizes of the input arguments to the clause head. Note that these $\overline{n}_j$ size relations have previously been computed during size analysis for all input arguments to literals in the bodies of all clauses.

Then, $\mathsf{Cost}_{clause}(C, ap, r, \overline{n})$, the resource usage expressed in units of resource r with approximation ap of clause C of predicate p, is given by the expression $\mathsf{Cost}_{clause}(C, ap, r, \overline{n}) = \mathsf{solver}(\mathsf{Cost}(C, ap, r, \overline{n}))$. That is, it is expressed as the solved form function of the following expression which, in general, for recursive clauses yields a recurrence equation:

$$\begin{aligned}\mathsf{Cost}(C, ap, r, \overline{n}) =\ & \delta(ap, r)(head(C)) + \\ & \sum_{j=1}^{lim(ap,C)} \Big(\prod_{l \prec j} \mathsf{Sols}_{\mathsf{L}_l}(\overline{n}_l)\Big)(\beta(ap, r)(L_j) + \mathsf{Cost}_{lit}(L_j, ap, r, \overline{n}_j))\end{aligned} \tag{4.9}$$

where $lim(ap, C)$ is a function that takes an approximation identifier ap and a clause C and returns the index of a literal in the clause body. For example, if ap is the identifier for approximation "upper bound" (`ub`), then $lim(ap, C) = k$ (the index of the last body literal). If ap is the identifier for approximation "lower bounds" (`lb`), then $lim(ap, C)$ is the index for the rightmost body literal that is guaranteed not to fail. $\delta(ap, r)$ is a function that takes an approximation identifier ap and a

[1]Note that the problem of detecting predicates whose clause tests are mutually exclusive is far from being trivial. Since the inference of mutual exclusion among predicate clauses is external to our analysis, it is beyond the scope of this thesis to explain it (see [79] for details).

resource identifier r and returns a function $\Delta^H(cl\_head, arith\_expr) \rightarrow \mathcal{B}$ which takes a clause head and returns an arithmetic resource usage expression $< arith\_expr >$ as defined in Figure 4.2. Thus, $\delta(ap, r)(head(C))$ represents $\Delta^H(head(C))$. On the other hand, $\beta(ap, r)$ is a function that takes an approximation identifier ap and a resource identifier r and returns a function $\Delta^L(body\_lit, arith\_expr) \rightarrow \mathcal{B}$ which takes a body literal and returns also an arithmetic resource usage expression $< arith\_expr >$. In this case, $\beta(ap, r)(L_j)$ represents $\Delta^L(L_j)$. Section 4.3.4 illustrates different definitions of the functions $\delta(ap, r)$ and $\beta(ap, r)$ in order to infer different resources. Sols_{L_l} is the number of solutions that literal $\mathtt{L}_l$ can generate, where $l \prec j$ denotes that $\mathtt{L}_l$ precedes $\mathtt{L}_j$ in the literal dependency graph for the clause. The inference of upper bounds on the number of solutions given a literal is far from being trivial. We take the approach of [36].

Finally, $\mathsf{Cost}_{lit}(\mathtt{L_j}, ap, r, \overline{n}_j)$ is:

- If L_j is recursive, i.e., calls a predicate q which is in the strongly-connected component of the call graph being analyzed, then $\mathsf{Cost}_{lit}(L_j, ap, r, \overline{n}_j)$ is replaced by a symbolic expression $\mathsf{Cost}(q, ap, r, \overline{n}_j)$.
- If L_j is not recursive, assume that it is a call to q (where q can be either a predicate call, or an external or builtin predicate), then q has been already analyzed, i.e., the (closed form) resource usage function for q has been recursively computed as γ and $\mathsf{Cost}_{lit}(L_j, ap, r, \overline{n}_j)$ can be expressed explicitly in terms of the function γ, and it is thus replaced with $\gamma(\overline{n}_j)$, i.e., the resource usage function γ is updated with the sizes at that particular program point which is given by $\overline{n}_j$.

Note that in both cases, if there is a resource usage assertion for q, `'cost(ap,-r,`$\langle arith\_expr\rangle$`)'`, then $\mathsf{Cost}_{lit}(L_j, ap, r, \overline{n}_j)$ is replaced by the most precise between the arithmetic resource usage expression in closed form and its closed form resource usage function inferred previously by the analysis, provided they are not incompatible, in which case an error is flagged.

It can be proved by induction on the number of literals in the body of clause C that:

1. If clause C is not recursive, then, expression (4.9) results in a closed form function of the sizes of the input argument positions in the clause head;

2. If clause C is simply recursive, then, expression (4.9) results in a recurrence equation in terms of the sizes of the input argument positions in the clause head;

3. If clause C is mutually recursive, then expression (4.9) results in a recurrence equation which is part of a system of equations for mutually recursive clauses in terms of the sizes of the input argument positions in the clause head.

If these recurrence equations can be solved, including approximating the solution in the direction of ap, then $\mathsf{Cost}(p, ap, r, \overline{n})$ can be expressed in a closed form, which is a function of the sizes of the input argument positions in the head of predicate p (and hence $\mathsf{Cost}_{clause}(C, ap, r, \overline{n}) = \mathsf{solver}(\mathsf{Cost}(p, ap, r, \overline{n}))$). Thus, after the strongly-connected component to which p belongs in the call graph has been analyzed, we have that expression (4.8) results in a closed form function of the sizes of the input argument positions in the clause head.

Finally, note that our analysis is parameterized by the functions $\delta(ap, r)$ and $\beta(ap, r)$ whose definitions can be given by means of assertions of type `head_cost` and `literal_cost`, respectively. These functions make our analysis parametric with respect to any resource of interest defined by users.

4.3.4 Defining the Parameters (Functions) of the Analysis

In this section we explain and illustrate with examples how the functions that make our resource analysis parametric, namely, δ (which includes the definition of Δ^H), and β (which includes the definition of Δ^L) are written in practice in our system. Both $\Delta^H(cl\_head, arith\_expr) \rightarrow \mathcal{B}$ and $\Delta^L(body\_lit, arith\_expr) \rightarrow \mathcal{B}$ will be implemented as predicates of two arguments. The first one takes the clause head if Δ^H or the body literal, otherwise. The second argument is the resource usage function.

Assume for example that the resource we want to measure is an upper bound on the number of resolution steps (`steps`) performed by a program. This can be achieved by adding one unit each time a clause head is traversed. Since the assertion `head_cost` is applied each time a clause head is analyzed, it is straightforward to measure the number of resolution steps by providing the following assertion and definition of the `delta_one/2` predicate:

```
:- head_cost(ub,steps,delta_one).
delta_one(_,1).
```

Note that the predicate `delta_one/2` is the definition of Δ^H and it will return one for any value in its first argument. If the resource usage function is a constant expression and it does not depend on the clause head or body literal, the system also allows writing the following shortcut:[2]

```
:- head_cost(ub,steps,1).
```

In order to simplify the process of defining interesting and useful Δ^H and Δ^L functions, our implementation provides a library with predicates that perform syntactic

[2]In the worked example in Figure 4.1 the `head_cost` and `literal_cost` assertions are written following this style.

operations on clauses, such as, for example, getting the number of arguments in a clause head or body literal, getting a clause head, getting a clause body, accessing an argument of a clause head or body literal, getting the main functor and arity of a term in a certain position, etc. In this context it is important to remember that the different Δ^H and Δ^L function definitions perform syntactic matching on the program text.

Assume now that the resource we want to measure is the number of argument passings (`num_args`) that occur during clause head matching in a program. This is achieved by the following code:

```
:- head_cost(ub,num_args,delta_num_args).
delta_num_args(H,N) :-  functor(H,_,N).
```

`functor/3` is a predicate defined in any Prolog system and it receives a term and returns the functor symbol and the arity in the second and third argument, respectively.

As another example, if we are interested in decomposing arbitrary unifications performed while unifying a clause head with the literal being solved into simpler steps, we can define a resource `num_unifs`, and a `head_cost` assertion which counts the number of function symbols, constants, and variables in each clause head as follows:

```
:- head_cost(ub,num_unifs,
             delta_num_unifs).

delta_num_unifs(H,S) :-
   functor(H,_,N),
   num_fun_vars(N,H,S).

num_fun_vars(0,_H,0).
num_fun_vars(N,H,S) :-
   N > 0,
   arg(N,H,Arg),
   nfun_vars(Arg,S1),
   N1 is N-1,
   num_fun_vars(N1,H,S2),
   S is S1 + S2.
```

```
nfun_vars(Arg,1) :-
   var(Arg).
nfun_vars(Arg,1) :-
   atomic(Arg).
nfun_vars(Arg,S) :-
   nonvar(Arg),
   functor(Arg,_, N),
   num_fun_vars(N,Arg,S1),
   S is S1 + 1.
```

`var/1`, `atomic/1`, `nonvar/1`, `arg/3` are additional built-in predicates in ISO-Prolog similarly to `functor/3`. `var/1` succeeds if the input argument is a free variable. `atomic/1` succeeds if the input argument is instantiated to an atom. `nonvar/1` succeeds if the input argument is a term which is not a free variable. Finally, `arg(Index,Term,Arg)` returns in `Arg` argument number `Index` from `Term`.

If in addition to the number of unifications performed while unifying a clause head we are also interested in the cost of term creation for the literals in the body of clauses, we can define a resource `terms_created`, and include a `literal_cost` (Δ^L) assertion which keeps track of the number of function symbols and constants in body literals:

```
:- literal_cost(ub,
   terms_created,
   beta_terms_created).
beta_terms_created(L,S) :-
    functor(L,_,N),
    num_fun(N,L,S).

num_fun(0,_L,0).
num_fun(N,L,S) :-
    N > 0,
    arg(N,L,Arg),
    nfun(Arg,S1),
    N1 is N-1,
    num_fun(N1,L,S2),
    S is S1 + S2.
```

```
nfun(Arg,0) :-
    var(Arg).
nfun(Arg,1) :-
    atomic(Arg).
nfun(Arg,S) :-
    nonvar(Arg),
    functor(Arg,_,N),
    num_fun(N,Arg,S1),
    S is S1 + 1.

:- head_cost(ub,
   terms_created,
   delta_terms_created).

delta_terms_created(_L,0).
```

Note that in this case we also define a `head_cost` assertion which returns 0 for every clause head.

More interestingly, our implementation provides a library with predicates that perform semantic checks of properties. These properties are inferred by the available analyzers. Some of the analyses are always performed as part of the resource analysis, such as mode and type analysis, and others are performed on demand, depending on the properties that need to be checked in the Δ^H and Δ^L function definitions or depending on the type of approximation to be performed by the resource analysis.

For instance, suppose that for debugging purposes we would like to generate heap space cost relations to define an upper bound on the heap consumption of the program as a function of its input data sizes. In order to infer the heap consumption of the program, we will assume for example purposes a simple memory model. We define a resource model, $\mathcal{M}_{heap}$, that counts the number of bytes allocated in the heap as follows. We assume that input arguments are ground and hence, no heap allocation is required. Therefore, we only consider the heap usage of the output arguments using the following formula:

$$\mathcal{M}_{heap}(t) = \begin{cases} 4 & \text{if } t \text{ is output and constant or variable} \\ 4 + \sum_{i=1}^{i=N} \mathcal{M}_{heap}(t_i) & \text{if } t \text{ is output and } t = f(t_1, \ldots, t_N) \\ 0 & \text{otherwise} \end{cases} \qquad (4.10)$$

Then, we can implement Equation 4.10 through a `heap_usage_function/2` predicate, defined as:

```
heap_usage_function(LitInfo,Cost) :-
    get_literal(LitInfo,Head),
    get_modes(LitInfo,Modes),
    usage_func(Modes,Head,1,0,Cost).

usage_func([],_Head,_Ind,Cost,Cost).
usage_func([in|Modes],Head,Ind,Acc,Cost):-
    NInd is Ind + 1,
    usage_func(Modes, Head,NInd,Acc,Cost).
usage_func([out|Modes],Head,Ind,Acc,Cost):-
    arg(Index,Head,Term),
    term_heap_usage(Term,Cost),
    NAcc is Acc + Cost,
    NInd is Ind + 1,
    usage_func(Modes, Head,NInd,NAcc,Cost).

term_heap_usage(Term,4):- var(Term).
term_heap_usage(Term,4):- atm(Term).
term_heap_usage(Term,N):-
    functor(Term,F,_A),
    Term =.. [F|Ts],
    term_heap_usage_(Ts,N1),
    N is N1 + 4.

term_heap_usage_([],0).
term_heap_usage_([T|Ts],N):-
    term_heap_usage(T,N1),
    term_heap_usage_(Ts,N2),
    N is N1 + N2.
```

where *Term* `=..` *List* means that the functor and arguments of the term Term comprise the list List. For instance, `f(a,b) =.. [f,a,b]`.

It is important to notice that `heap_usage_function/2` not only operates *syntactically* on the program text but also *semantically* since the argument modes are considered. Further, since the creation of terms can occur both in the clause head and in the body literals, we need to apply `heap_usage_function/2` to both cases. The user makes this explicit through the assertions:

```
:- head_cost(ub,heap_usage,heap_usage_function).
:- literal_cost(ub,heap_usage,heap_usage_function).
```

Assume now that we want to separate the counting of unifications where one of the terms being unified is a variable and thus behaves as an "assignment," and the counting of full unifications, i.e., when both terms being unified are not variables, and thus unification performs a "test" or produces new terms, etc.

For this purpose, we can define a resource, as for example `vo_unif`, which counts the number of variables in the clause head which correspond to "output" argument positions through `head_cost` assertions. This describes a component of the execution time that is directly proportional to the number of cases where both a goal argument and the corresponding head argument are variables. This should boil down to assignment (maybe with trailing). This is achieved by the following code:

```
:- head_cost(ub,vo_unif,
             delta_vo_unif).
delta_vo_unif(H,S) :-
    functor(H,_,N),
    num_vo_unif(N, H, S).

num_vo_unif(0,_H,0) :- !.
num_vo_unif(N,H,S) :-
    arg(N,H,Arg),
    free(Arg),
    !,
    nvo_unif(Arg,S1),
    N1 is N-1,
    num_vo_unif(N1, H, S2),
    S is S1 + S2.
```

```
num_vo_unif(N,H,S) :-
    N1 is N-1,
    num_vo_unif(N1,H,S).

nvo_unif(Arg,1) :-
    var(Arg).
nvo_unif(Arg,0) :-
    atomic(Arg).
nvo_unif(Arg,S) :-
    nonvar(Arg),
    functor(Arg,_, N),
    num_vo_unif(N,Arg,S1),
    S is S1 + 1.
```

Similarly, we could define resources for counting:

- The number of variables in the clause head which correspond to input argument positions
- The number of function symbols and constants in the clause head which appear in output arguments.
- The number of function symbols and constants in the clause head which appear

in input arguments.

Example 4.3.2. Consider the same program defined in Figure 4.1 and the size relations computed in Example 4.3.1. We now show the corresponding resource usage equations for each clause for the resource `bits`. Although the functions $\delta(ap, r)(H)$ and $\beta(ap, r)(L)$ take as arguments a clause head H and a body literal L respectively, in our examples we will only write the predicate name of H and L for the sake of simplicity. Since the program is analyzed in a single traversal of the call graph in reverse topological order, the system starts by analyzing the predicate `exch_buffer/3`. Note that the resource usage for external predicates (whose code is not available) `connect/3`, `exch_byte/3` and `close/1` is already given by user assertions:

```
:- pred connect(Host,Port,S) ...   + cost(ub,bits,0)
:- pred close(Stream) ...          + cost(ub,bits,0)
:- pred exch_byte(B,Id,B0) ...     + cost(ub,bits,8)
```

which express that:

$$\begin{array}{lcl}
\mathsf{Cost}(\mathit{connect}, \mathtt{ub}, \mathtt{bits}, \langle \_, \_ \rangle) & = & 0 \\
\mathsf{Cost}(\mathit{close}, \mathtt{ub}, \mathtt{bits}, \langle \_ \rangle) & = & 0 \\
\mathsf{Cost}(\mathit{exch\_byt}, \mathtt{ub}, \mathtt{bits}, \langle \_, \_ \rangle) & = & 8
\end{array}$$

For simplicity, we have omitted the sizes of the input arguments (don't care symbols) since the resource usage functions do not depend on the sizes of the input arguments.

The system first infers the resource usage of the first clause of `exch_buffer/3` applying formula 4.9. Recall that the δ and β functions applied to each clause head and body literal respectively return zero, as provided by the user through the `head_cost` and `literal_cost` assertions. Additionally, the body of the clause is empty. Hence, no resource usage is provided by the clause. Then, the system yields the following resource usage equation:[3]

$$\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle 0, \_\rangle) = 0$$

For the recursive clause of `exch_buffer/3`, the system follows the same formula 4.9. For this clause, there is a call to a predicate (`exch_byte/3`) that receives 8 bits, and also a recursive call to `exch_buffer/3`. In this case, the system establishes a symbolic expression of the form $\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n-1, \_\rangle)$ that expresses the resource usage of the recursive call. Note that the size of the first input argument has been updated at this particular program point, $n-1$, where n represents the size of the first argument to this predicate. Therefore, the analysis sets up the following recurrence equation:

$$\begin{aligned}\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n, \_\rangle) = \;& \overbrace{\delta(\mathtt{ub}, \mathtt{bits})(exch\_buffer)}^{0} + \\ & \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(exch\_byte)}^{0} + \overbrace{\mathsf{Cost}(exch\_byte, \mathtt{ub}, \mathtt{bits}, \langle \_, \_\rangle)}^{8} + \\ & \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(exch\_buffer)}^{0} + \mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n-1, \_\rangle) \\ & = 8 + \mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n-1, \_\rangle)\end{aligned}$$

Then, the analysis calls a recurrence solver with the recurrence equation system inferred:

$$\begin{aligned}\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle 0, \_\rangle) &= 0 \\ \mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n, \_\rangle) &= 8 + \mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle n-1, \_\rangle)\end{aligned}$$

yielding the following closed form resource usage function:

$$\mathsf{Cost}(exch\_buf, \mathtt{ub}, \mathtt{bits}, \langle n, \_\rangle) = 8 \times n$$

Finally, the system analyzes the main predicate of the program (i.e., `client/3`).

[3]Again the size of the second input argument is omitted since it is irrelevant for the resource usage of the predicate.

This predicate has only one clause which is not recursive. Moreover, the resource usage functions of all body literals have been previously inferred by the analysis (e.g., `exch_buffer/3`) or given by the user through assertions (e.g., `connect/3` and `close/1`). Then, the system sets up the following expression where k expresses the size of the input buffer, i.e., the second argument to this predicate:

Resource usage equations for `client/3`:

$$\begin{aligned}\mathsf{Cost}(client, \mathtt{ub}, \mathtt{bits}, \langle \_, k\rangle) = \;& \overbrace{\delta(\mathtt{ub}, \mathtt{bits})(client)}^{0} + \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(connect)}^{0} + \\ & \overbrace{\mathsf{Cost}(connect, \mathtt{ub}, \mathtt{bits}, \langle \_, \_\rangle)}^{0} + \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(exch\_buffer)}^{0} + \\ & \overbrace{\mathsf{Cost}(exch\_buffer, \mathtt{ub}, \mathtt{bits}, \langle k, \_\rangle)}^{8\times k} + \overbrace{\beta(\mathtt{ub}, \mathtt{bits})(close)}^{0} + \\ & \overbrace{\mathsf{Cost}(close, \mathtt{ub}, \mathtt{bits}, \langle \_\rangle)}^{0} = 8 \times k\end{aligned}$$

i.e., the result of the analysis is that an upper bound of the bits received by the client application is eight times the size of the second input argument, which is a buffer of bytes.

4.4 Experimental results

In this section we study the feasibility of the approach analyzing a set of representative benchmarks which include definitions of resources using this language and used the system to infer the resource usage bound functions. In order to do this, we have completed a prototype implementation of the analyzer, written in the `Ciao` language, using a number of modules and facilities from `CiaoPP`, including recurrence equation processing. We have also written a `Ciao` language extension (a "package" in `Ciao` terminology) which when loaded into a module allows writing the resource-related assertions and declarations proposed herein.[4]

First, we show the actual resource for which bounds are being inferred by the analysis for a given benchmark together with a brief description. In addition, we also show the size metric used for the relevant arguments. While any of the resources defined in a given benchmark could then be used in any of the others we show only the results for the most natural or interesting resource for each one of them. We have tried to use a relatively wide range of resources: number of bytes sent by an application, number of calls to a particular predicate, robot arm movements, number of files left open in a kernel code, number of accesses to a database, heap memory usage, etc. We also cover a significant set of complexity functions such as constant, polynomial, and exponential using relevant data structures in Prolog programs such as lists, trees, etc.

- `bst` is a program that illustrates a typical operation, insertion, over binary search trees, and we measure the heap usage in terms of number of bytes as a function on the depth of the input argument.
- `client` is the program depicted in Fig. 4.1 and we measure the number of bits received by the application as a function on the length of the input argument.
- `color_map`: performs map coloring and we measure the number of unifications as a function that depends on the term size of one of the input arguments.
- `fib`: computes the fibonacci function and infers the number of arithmetic operations in terms of the integer value of the input argument.
- `hanoi`: is the Towers of Hanoi program and we assume that this program, after computing the movements, sends these to a robotic arm that will actually be

[4]The system also supports adding resource assertions specifying expected resource usages which the system will then verify or falsify using the results of the implemented analysis.

moving the disks. We want to measure the energy consumption of the robot movements as a function in terms of the integer value of the input argument.

- `eight_queen`: plays the 8-queens game and we measure the number of queens movements as a function in terms of the length of the input argument.

- `eval_polynom`: evaluates a polynomial function and we measure the floating point unit time usage as a function in terms of the length of the the list of coefficients.

- `grammar`: represents a simple sentence parser and we measure the number of phrases generated by the parser as a function in terms of the term size of the input argument.

- `insert_stores`: is a database transaction that adds a new entry into the `STORE` relation. We measure the number of updates as a function in terms of the relation size, i.e. number of records.

- `merge`: is a program that merges the content of a set of input files into an output file, and we measure the number of files left open as a function in terms of the length of the list of files.

- `power_set`: generates the powerset of a list and we measure the number of output elements as a function in terms of the input list length.

- `qsort`: implements the quicksort algorithm and we measure the number of lists parallelized as a function in terms of the input list length.

- `send_files`: is a program that sends the content of a set of files through a stream. We measure the number of bytes read as a function in terms of the input list length.

Program	Usage Function	Exact Function	Time
`bst`	$\lambda x.20 \cdot x + 16$	$\equiv$	184
`client`	$\lambda x.8 \cdot x$	$\equiv$	186
`color_map`	104691	31686	176
`eight_queen`	19173961	$\equiv$	304
`eval_polynom`	$\lambda x.2.5x$	$\equiv$	44
`fib`	$\lambda x.2.17 \cdot 1.61^x +$ $0.82 \cdot (-0.61)^x - 3$	$\equiv$	116
`grammar`	24	16	227
`hanoi`	$\lambda x.2^x - 1$	$\equiv$	100
`insert_stores`	$\lambda n, m.n + k$ $\lambda n, m.n$	$\equiv$ $\equiv$	292
`merge`	$\lambda x.x$	$\equiv$	180
`power_set`	$\lambda x.\frac{1}{2} \cdot 2^{x+1}$	$\equiv$	119
`qsort`	$\lambda x.4 \cdot 2^x - 2x - 4$	$\lambda x.2 \cdot x^2$	144
`send_files`	$\lambda x, y.x \cdot y$	$\equiv$	179
`subst_exp`	$\lambda x, y.2xy + 2y$	$\lambda x, y.x \cdot y$	153
`zebra`	30232844295713061	6869	292

Table 4.1: Accuracy and efficiency in milliseconds of the analysis.

- `subst_exp`: substitutes a list of variables in a mathematical expression. We measure the number of replacements as a function in terms of the list length and also the term size of the input arguments.
- `zebra`: based on the classic zebra puzzle we measure the number of resolution steps as a function in terms of the term size of the input.[5]

The results from the analysis of these benchmarks are shown in Table 4.1. For brevity, we report only results for upper-bounds analysis. The column Usage Function shows the actual resource usage function (which depends on the size of the input arguments) inferred by the analysis, given as a lambda term. The column Exact Function shows the exact resource usage function, given also as a lambda term. Finally,

[5]The system infers a resource usage function considering all possible solutions. The exact function shown in Table 4.1 considers only one solution.

the column labeled Time shows the resource analysis times in milliseconds, on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 5.0. Note that these times do not include other analyses such as types, modes, etc.

4.5 Related Work

As mentioned previously, most previous work is specific to the analysis of execution steps and, sometimes, time or memory. The ACE system [69] can automatically extract upper bounds on execution steps for a subset of functional programming. The system is based on program transformation. The original program is transformed into a step-counting version and then into a composition of a cost bound and a measure function. Rosendahl defines in [104] another automatic upper-bound analysis based on an abstract interpretation of a step-counting version. The analysis measures both execution time and execution steps. However, size measures cannot automatically be inferred and the experimental section shows few details about the practicality of the analysis. Debray et al. presents in [36, 37] a semi-automatic analysis which infers upper-bounds on the number of execution steps. These bounds are functions on the sizes or value ranges of input data. This seminal work applies to a large class of logic programs and presents techniques in order to deal with the generation of multiple solutions via backtracking. The authors also show how other specific analyses could be developed, such as for, e.g., time or memory. This approach was later fully automated and extended to inferring upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [38, 54]. Our method builds on this work but generalizes it in order to deal with a much more general class of user-defined resources, allowing thus the coverage of different analyses within a single implementation. Grobauer presents in [47] a method for automatically extracting cost recurrences from first-order DML (dependent ML) programs, a conservative extension of ML. The main feature is the use of dependent types to describe a size measure that abstracts from data to data size. In [96], and inspired by [13] and [82], a complexity analysis is presented for Horn clauses, also fully automating the necessary calculations. In [60], Igarashi et al. presents a method for modeling problems such as memory management, lock primitive usage, etc., and a type-based method is proposed as solution to the inference problem. In [116] a cost model is presented for inferring cost equations for recursive, polymorphic, and higher-order functional programs. While it is claimed that the approach can be modified in order to infer a reduced set of resources such as execution time, execution steps, or memory, no details are given. Worst case execution time (WCET) estimation has been studied for imperative languages and for different application domains (see, e.g., [111, 14, 40] and its references). However these and related methods again concentrate only on execution time. Also, they do not infer cost functions of input data sizes but rather absolute maximum execution times, and they generally require the manual annotation of loop iteration bounds. In [25] a method is presented for reserving resources before their actual use. However, the programmer (or program optimizer) needs to annotate the program with “acquire” and “consume” primitives, as well as provide loop invariants and function pre- and post-conditions. Interesting type-based related work has also been performed in the GRAIL system [9], also oriented towards resource analysis, but it has concentrated mainly on ensuring memory bounds.

In comparison with previous work our approach allows dealing with a class of resources which is open, in the ample sense that such resources are in fact defined by programmers using an assertion language, which we also consider itself an important contribution of our work. Another important contribution of our work because of its impact in the scalability and automation of the analysis is that our approach allows defining the resource usage of external predicates, which can be used for modular composition. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating.

4.6 Summary

Research about resource usage analysis goes back to the seminal work by Wegbreit in 1975 [118], which proposed to analyze the performance of a program by deriving closed form expressions for its execution behavior. Since then, there has been a good number of cost analysis frameworks for a wide variety of programming languages, including functional [69, 104, 101, 108, 45, 15, 47], imperative [111, 14, 40, 119], and logic languages [37, 36, 38].

In spite of such large amount of work in the area, there is a lack of resource usage analysis tools that:

- deal with a generic user-definable notion of resources allowing thus the coverage of different analyses within a single implementation.
- analyze programs with a realistic size and complexity in a fully automated way.

In this chapter, we have presented a resource bounds analysis framework that infers upper and lower bounds on the usage that a logic program makes of a quite general notion of user-definable resources. The inferred bounds are in general functions of input data sizes. We have also presented the assertion language which is used to define such resources. The analysis then derives the related (upper- and lower-bound) resource usage functions for all predicates in the program. Our experimental evaluation is encouraging because it shows that interesting resource bound functions can be obtained automatically and in reasonable time, for a representative set of benchmarks with a good variety of resources such as bits sent or received by an application over a socket, number of files left open, number of accesses to a database, energy consumption, etc., as well as the more traditional execution steps, execution time, or heap memory. While clearly further work is needed to assess scalability we are cautiously hopeful in the sense that our approach allows defining via assertions the resource usage of external predicates, which can then be used for modular composition. These includes also predicates for which the code is not available or which are written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. Our expectation is that the automatic analysis will be able to do the bulk of the work for large applications, even if the cost of some specially complex predicates may still need to be given by the user. Finally, we expect the applications of our analysis to be rather interesting, including resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization.

Chapter 5

Set-Sharing Analysis

In the automatic inference of resource bounds functions certain program information is first automatically inferred by other analyzers. In particular, the input/output modes of the predicate arguments represent essential information for the resource usage analysis. *Set-Sharing* analyses aim to detect which variables do not point transitively to the same memory location. This information can provide very accurate input/output modes to the resource usage analysis. While other techniques exist for inferring modes (such as, e.g., *def* analysis [7]) we choose the sharing domain because it is also useful for many other optimizations in compilers (in a similar way to points-to and shape-sharing analyses in imperative languages). However, traditional Set-Sharing analyses can also be quite inefficient and they are traditionally not considered good choices when analyzing large programs.

In this chapter, we provide the background information about the Set-Sharing abstract domain introduced by Jacobs and Langen in 1990 [61, 68] that will be necessary to understand our two practical Set-Sharing solutions presented in this thesis, in Chapters 6 and 7, respectively.

5.1 Overview

Definition 5.1.1. (Sharing). Two or more variables in a logic program are said to *share* if in some possible execution of the program they are bound to terms that contain a common variable. ∎

Recall that a variable in a logic program is said to be *ground* if it is bound to a term that does not contain free variables. *Set-Sharing* is an important type of combined sharing and groundness analysis. It was originally introduced by Jacobs and Langen [61, 68] and its abstract values are sets of sets of variables that keep track of the sharing relationships among variables.

Example 5.1.1. (Set-Sharing abstraction). Let $\mathcal{V} = \{X, Y, Z\}$ be a set of variables of interest. A substitution $\theta = \{X \mapsto f(U_1, U_2, V_1, V_2, W), Y \mapsto g(V_1, V_2, W), Z \mapsto g(W)\}$, depicted in Figure 5.1, will be abstracted in Sharing as $\{\{X\}, \{X, Y\}, \{X, Y, Z\}\}$. Sharing group $\{X\}$ in the abstraction represents the occurrence (i.e., the possible occurrences of run-time variables within the terms to which program variables will be bound) of run-time variables U_1 and U_2 in the concrete substitution, $\{X, Y\}$ represents V_1 and V_2, and $\{X, Y, Z\}$ represents W. Note that the number of (occurrences of) run-time variables shared is abstracted away.

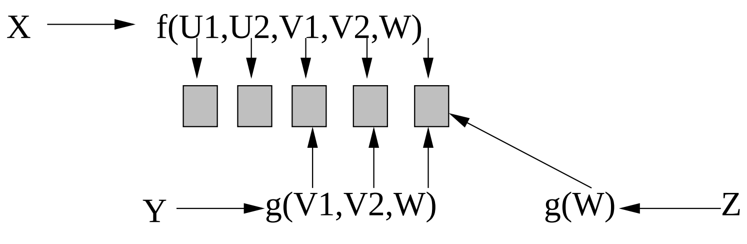


Figure 5.1: Memory Representation for θ

Definition 5.1.2. (Independence). Several program variables are said to be *independent* if the terms they are bound to do not have (run-time) variables in common.

■

Variable independence is the counterpart of sharing: program variables share when the terms they are bound to do have run-time variables in common. When we are talking of only two variables then we refer to *Pair-Sharing* [109], and when we track relations among more than two variables we refer to *Set-Sharing*. Sharing abstract domains are used to infer *may sharing*, i.e., the possibility that shared variables exist, and thus, in the absence of such possibility, *definite* information about independence.

Example 5.1.2. (Notion of independence). Let $\mathcal{V} = \{X, Y, Z\}$ be the variables of interest. A Set-Sharing abstract substitution such as $\{\{X\}, \{Y\}, \{Z\}\}$ (which denotes the set of the singleton sets containing each variable) represents that all three variables are independent.

Sharing analysis has been used to infer several interesting properties and perform optimization and verification of programs at compile-time, most notably but not limited to: occurs-check reduction (e.g., [109]), automatic parallelization (e.g., [92, 91, 23]), and finite-tree analysis (e.g., [11]). In addition, and as mentioned before, the resource usage analysis described in Chapter 4 requires that certain program information (such as, for example, input/output modes, types, non-failure information, etc.) be first automatically inferred by other (abstract interpretation-based) analyzers. Set-Sharing analyses can provide very accurate input/output modes to the resource usage analysis and improve the accuracy of others such as e.g., types and non-failure which are also required by the resource usage analysis.

5.2 Preliminaries

Let $\wp(S)$ denote the powerset of set S, and $\wp^0(S)$ denote the *proper powerset* of set S, i.e., $\wp^0(S) = \wp(S) \setminus \{\emptyset\}$. Let also $|S|$ denote the cardinality of a set S. Let $\mathcal{V}$ be a set of variables of interest; e.g., the variables of a program.

Let F and P be sets of ranked (i.e., with a given arity) functors of interest; e.g., the function symbols and the predicate symbols of a program. We will use $Term$ to denote the set of terms constructed from $\mathcal{V}$ and $F \cup P$. Although somehow unorthodox, this will allow us to simply write $g \in Term$ whether g is a term or a predicate atom, since all our operations apply equally well to both classes of syntactic objects. We will denote $\hat{t}$ the set of variables of $t \in Term$. For two elements $s \in Term$ and $t \in Term$, $\hat{st} = \hat{s} \cup \hat{t}$. We will also denote by $[t]_y$ the number of occurrences of the variable y in the term t.

Analysis of a program proceeds by abstractly solving unification equations of the form $t_1 = t_2$, $t_1 \in Term$, $t_2 \in Term$. Let $solve(t_1 = t_2)$ denote the solved form of unification equation $t_1 = t_2$. The results of analysis are abstract substitutions which approximate the concrete substitutions that may occur during execution of the program. Let U be a denumerable set of variables (e.g., the variables that may occur during execution of a program). Concrete substitutions that occur during execution are mappings from $\mathcal{V}$ to the set of terms constructed from $U \cup \mathcal{V}$ and F.

5.3 The Set-Sharing Domain

The Set-Sharing abstract domain was first presented in [61]. Abstract unification for bottom-up analyses was first defined and proved correct in [68]. The presentation here and in Chapters 6 and 7 follows that of [120, 30], since the notation used and the abstract unification operation obtained are rather intuitive. A complete set of abstract functions for top-down analysis (as well as a top-down analysis framework) was defined and proved correct in [92], and presented in [91, 90].

A *sharing group* is a set containing one or more of the variables of interest, ant it represents a possible sharing among them (i.e., that they might be bound to terms which have a common variable).

Definition 5.3.1. (Set-Sharing abstract domain, SH). Let $SG = \wp^0(\mathcal{V})$ be the set of all sharing groups. A *sharing set* is a set of sharing groups. The Sharing domain is $SH = \wp(SG)$, the set of all sharing sets, ordered by $\subseteq$. ∎

Definition 5.3.2. (Occur). Let θ be a substitution and $V \in \mathcal{V}$ a variable of interest, the sharing group $occur(\theta, V)$ is defined as:

$$occur(\theta, V) = \{X \in \mathcal{V} \mid V \in var(\theta(X))\}$$

∎

For instance, if $\theta = \{X \mapsto f(V, U), y \mapsto g(V), Z \mapsto h(U, W)\}$ then:

- $occur(\theta, U) = \{X, Z\}$
- $occur(\theta, V) = \{X, Y\}$
- $occur(\theta, W) = \{Z\}$

The abstract function α_{SH} is defined as $\alpha_{SH}(\theta) = \{occurs(\theta, V) \mid V \in range(\theta)\}$. Jacobs and Langen proved that Set-Sharing enjoys a Galois Insertion into the domain of concrete substitutions, and in particular, a concretization function γ_{SH} exists. We now show the definition of the abstract unification which also was proved by Jacobs and Langen as a safe approximation of the concrete unification.

Definition 5.3.3. (Relevant sharing $rel(sh, t)$ and irrelevant sharing $irrel(sh, t)$). Given terms s and t, and $sh \in SH$, we denote by $rel : SH \times Term \rightarrow SH$ the set of sets in sh which have non-empty intersection with $\hat{t}$, the set of variables of t.

$$rel(sh,t) = \{s \mid s \in sh, (s \cap \hat{t}) \neq \emptyset\}$$

Also, $irrel(sh,t)$ is the complement of $rel(sh,t)$, i.e., $sh \setminus rel(sh,t)$. ■

Definition 5.3.4. (Cross-union $sh_1 \bowtie sh_2$). For two elements $sh_1 \in SH$, $sh_2 \in SH$, let $sh_1 \bowtie sh_2 : SH \times SH \to SH$ be their *cross-union*, i.e., the result of applying union to each pair in their Cartesian product $sh_1 \times sh_2$.

$$sh_1 \bowtie sh_2 = \{s \mid s = s_1 \cup s_2, s_1 \in sh_1, s_2 \in sh_2\}$$

■

Definition 5.3.5. (Up-closure, sh^*). Let $sh \in SH$ be a sharing set, then the *up-closure* $(.)^* : SH \to SH$ is defined as its closure under union that represents the smallest superset of sh such that $s_1 \cup s_2 \in sh^*$ whenever $s_1, s_2 \in sh^*$:

$$sh^* = \{s \mid \exists n \geq 1\ \exists t_1, \ldots, t_n \in sh,\ s = t_1 \cup \ldots \cup t_n\}$$

■

Definition 5.3.6. (Abstract unification, $amgu$). The abstract unification is a function $amgu : \mathcal{V} \times Term \times SH \to SH$ defined as:

$$amgu(x,t,sh) = irrel(sh, x = t) \cup (rel(sh,x) \bowtie rel(sh,t))^*$$

■

Example 5.3.1. (Abstract unification, *amgu*). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ be the set of variables of interest and let $sh = \{\{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}\}$ be a sharing set. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$\begin{array}{lcl}
A = rel(sh, X_1) & = & \{\{X_1\}\} \\
B = rel(sh, f(X_2, X_3)) & = & \{\{X_2\}, \{X_3\}\} \\
A \bowtie B & = & \{\{X_1, X_2\}, \{X_1, X_3\}\} \\
(A \bowtie B)^* & = & \{\{X_1, X_2\}, \{X_1, X_3\}, \\
 & & \{X_1, X_2, X_3\}\} \\
C = irrel(sh, X_1 = f(X_2, X_3)) & = & \{\{X_4\}\} \\
amgu(X_1, f(X_2, X_3), sh) = C \ \cup \ (A \bowtie B)^* & = & \{\{X_1, X_2\}, \{X_1, X_3\}, \\
 & & \{X_1, X_2, X_3\}\}, \{X_4\}\}
\end{array}$$

Finally, we define the rest of the abstract operations required by a top-down Set-Sharing analysis which have been proved sound in [91, 90]:

Definition 5.3.7. (Extend, *extend***).** Let $sh_1, sh_2 \in SH$ be two abstract substitutions and $t \in Term$ then *extend* updates all sharing groups in sh_1 relevant to t that appear in sh_2 and it is defined as follows:

$$extend(sh_1, t, sh_2) = irrel(sh_1, t) \cup \{ \ s \mid s \in rel(sh_1, t)^*, \ (s \ \cap \ \hat{t}) \in sh_2 \ \}$$

■

Definition 5.3.8. (Projection, *project***).** Let $sh \in SH$ be an abstract substitution and $t \in Term$, the projection of sh onto the variables of t is defined as:

$$project(t, sh) = \{s \cap \hat{t} \mid s \in sh\} \setminus \{\emptyset\}$$

■

Definition 5.3.9. (Augment, *augment***).** Let $sh \in SH$ be an abstract substitution and $t \in Term$, sh can be augmented with the variables of t as follows:

$$augment(t, sh) = sh \cup \{\{x\} \mid x \in \hat{t}\}$$

■

5.4 The Sharing+Freeness Domain

The inclusion of freeness information, i.e., which variables are free, into the Sharing domain and the benefits it could report to sharing analysis was already discussed in [68] but the first proposal of a domain was in [90]. The presentation here follows that of [57].

Example 5.4.1. Let $sh \in SH$ be an abstract substitution defined as $sh = \{\{X\},$-$\{W,Y\},\{Y,Z\}\}$. Assume the following abstract unification $amgu(X = f(W,Y), sh)$ which returns (by Definition 5.3.6) the new abstract substitution $sh_1 = \{\{X,W,Y\},$-$\{X,Y,Z\},\{X,W,Y,Z\}\}$.

Suppose now that X is a free variable. Then, it is not possible that W and Y can share through X since X is a free variable. In this case, the Up-closure operation can be avoided on the relevant sharing groups to W and Y. Thus, the result would be $sh_2 = \{\{X,W,Y\},\{X,Y,Z\}\}$. Since $sh_2 \subset sh_1$ it is shown that the inclusion of freeness can improve the original Set-Sharing.

Definition 5.4.1. (Sharing+Freeness domain, SHF). The Sharing+Freeness domain is $SHF = SH \times \mathcal{V}$, i.e., the Sharing domain, SH, augmented with a new component which tracks the variables which are free. ∎

Definition 5.4.2. (Abstract unification, $amgu^f$). The abstract unification defined as $\mathcal{V} \times Term \times SHF \to SHF$ and it is given by $amgu^f(x,t,(sh,f)) = (sh',f')$, where:[1]

$$sh' = \begin{cases} irrel(sh, x = t) \cup (rel(sh, x) \bowtie rel(sh, t)) & \text{if } x \in f \text{ or } t \in f \\ irrel(sh, x = t) \cup (rel(sh, x) \bowtie rel(sh, t)^*) & \text{if } x \notin f,\ t \notin f,\ \text{but } \hat{t} \subseteq f \\ & \text{and } lin(t) \\ amgu(x, t, sh) & \text{otherwise} \end{cases}$$

and $lin(t)$ holds if for all $y \in \hat{t}$: $[t]_y = 1$ and for all $z \in \hat{t}$ such that $y \neq z$, $rel(sh, y) \cap rel(sh, z) = \emptyset$.

$$f' = \begin{cases} f & \text{if } x \in f, t \in f \\ f \setminus (\cup\, rel(sh, x)) & \text{if } x \in f, t \notin f \\ f \setminus (\cup\, rel(sh, t)) & \text{if } x \notin f, t \in f \\ f \setminus (\cup\, (rel(sh, x) \cup rel(sh, t))) & \text{if } x \notin f, t \notin f \end{cases}$$

■

Note that, for implementation, the second condition in the direct definition of $lin(t)$ might be rather expensive to compute: $rel(sh, y)$ has to be calculated for every $y \in \hat{t}$ to check that each pairwise intersection is empty. Instead, an equivalent condition to checking pairwise intersections, which is more efficient, can be used: for all $s \in rel(sh, t)$ $|s \cap \hat{t}| = 1$.

Finally, we show how the functions *extend*, *project*, and *augment* are lifted for the inclusion of freeness information:

Definition 5.4.3. (Extend, $extend^f$). Let $(sh_1, f_1), (sh_2, f_2) \in SHF$ be two abstract substitutions and $t \in Term$ then $extend^f$ is defined as follows:

$$extend^f((sh_1, f_1), t, (sh_2, f_2)) = (sh', f')$$
$$sh' = extend(sh_1, t, sh_2)$$
$$f' = f_2 \cup \{x \mid x \in (f_1 \setminus \hat{t}), ((\cup rel(sh', x)) \cap \hat{t}) \subseteq f_2\}$$

■

[1]Note that t is not necessarily a variable: $t \in f$ means "t *is* a variable and is known to be free".

Definition 5.4.4. (Projection, $\mathit{project}^f$). Let $(sh, f) \in SHF$ be an abstract substitution and $t \in Term$, the projection of (sh, f) onto the variables of t is defined as:

$$\mathit{project}^f(t, (sh, f)) = (\mathit{project}(t, sh), f \ \cap \ \hat{t})$$

■

Definition 5.4.5. (Augment, $\mathit{augment}^f$). Let $(sh, f) \in SHF$ be an abstract substitution and $t \in Term$, (sh, f) can be augmented with the variables of t as follows:

$$\mathit{augment}^f(t, (sh, f)) = (\mathit{augment}(g, sh), f \ \cup \ \hat{t})$$

■

5.5 Previous Work

Due to all applications described in Section 5.1, the accuracy of the Set-Sharing domain has received a lot of attention in the literature in the past. In particular, it has been improved by extending it with other kinds of information, the most relevant being:

- Extension with *linearity* was first proposed by Jacobs and Langen [61]. Extension with *freeness* was proposed by Muthukumar and Hermenegildo [90, 91]. These extensions have been further studied by Codish et al. [29], and Hill, Zaffanella, and Bagnara [57].
- Combination with *term structure* information such as depth-k was proposed preliminarily in in [88, 90] and developed fully as depth-k sharing by King and Soper [64], in the abstract equation systems by Mulkers et al. [87], and in the composite domains for deriving sharing by Bruynooghe et al. [19].

- Finally, combination with other abstract domains has been proposed in [28, 43, 30].

However, Set-Sharing has a key computational disadvantage: the *abstract unification* implies potentially exponential growth in the number of sharing groups due to the *closure* operation which is the heart of that operation. Therefore, the study of reducing the impact of the complexity of this operation has been also essential:

- In [92, 91], Muthukumar and Hermenegildo presented the first comparatively efficient algorithms for performing the basic operations needed for implementing set sharing-based analyses.

- In [30], Codish, Søndergaard, and Stuckey showed that the Jacobs and Langen's sharing domain is isomorphic to the dual negative of *Pos* [7], denoted by $\overline{coPos}$. This insight improved the understanding of sharing analysis, and led to an elegant expression of the combination with groundness dependency analysis based on the reduced product of Sharing and Pos. In addition, this work pointed out the possible implementation of $\overline{coPos}$ through *Reduced Ordered Binary Decision Diagrams* (*ROBDDs*) [20], although this point was not investigated further therein.

- In [120], Zaffanella et al. extended the Set-Sharing representation for inferring pair-sharing from a set of sets of variables to a pair of sets of sets of variables in order to support widening. A new component is added to abstract substitutions that represents sets of variables, the powerset of which would have been part of the original abstract substitution. Such sets are called *cliques*. The precision and efficiency of using cliques for the case of inferring pair-sharing were reported in [120]. In [121], cliques were incorporated into the original Sharing domain, but precision and efficiency are again studied for the case of inferring pair-sharing. Although significant efficiency gains were achieved, this approach loses precision with respect to the original Set-Sharing.

- Other relevant work was presented in [74] in which the closure operation was delayed and full sharing information was recovered lazily. However, this interesting approach shares some of the disadvantages of Zaffanella's widening. Therefore, the authors refined the idea in [73] reformulating the amgu in terms of the *closure under union* operation, collapsing those closures to reduce the total number of closures and applying them to smaller descriptions without loss of accuracy.

In the next two chapters, we will present two new, alternative, and practical solutions to the problem of Set-Sharing analysis. The first approach, in Chapter 6, is inspired by Zaffanella et al. [120] and the idea behind this approach is to define different widening operators to accelerate the fixpoint computation. Although, as we will show, relevant efficiency gains are achieved, this is achieved at the expense of losses in accuracy. Our second approach, in Chapter 6, is based on representing the complement of the sharing relationships. This alternative representation may imply important efficiency gains when the number of relationships is relatively large, and has the advantage of doing so without any loss of accuracy.

Chapter 6

Widening Set-Sharing Analysis

In this chapter, we study the problem of improving the efficiency and scalability of Set-Sharing analysis of logic programs for top-down analyses using a form of *cliques*. We provide a brief overview of the approach in Section 6.1. The representation based on cliques and the clique-domains for set-sharing and set-sharing with freeness are presented in Sections 6.2 and Section 6.3, respectively. In Section 6.4 the required functions for top-down analysis are defined. In Section 6.5 an algorithm for detecting cliques is presented and, in Section 6.6 the use of the representation based on cliques as widening is shown. Section 6.7 shows an experimental evaluation of the proposed analyses. Finally, Section 6.8 summarizes this chapter.

6.1 Overview

Our starting point is Zaffanella's idea [120] of representing sharing information as a pair of abstract substitutions, one of which is a worst-case sharing representation called a clique set, which as mentioned previously, was proposed for the case of inferring pair-sharing. Our other starting point is the original set-sharing. The main goal is to reduce the running time and memory consumption of the traditional Set-Sharing domain.

We use the clique-set representation for:

1. Inferring actual set-sharing information, and
2. Analysis within a top-down framework.

In particular, we define the new abstract functions required by standard top-down analyses, both for sharing alone and also for the case of including freeness in addition to sharing. Such functions were not defined in [120, 121], since bottom-up analyses were used there. The analysis uses the PLAI/CiaoPP framework [52], which, as mentioned before, includes an efficient implementation of a top-down analyzer using the fixpoint algorithms and optimizations described in [89, 91, 55]. We use cliques both as an alternative representation and as a widening, defining several widening operators.

6.2 The Clique-Sharing Domain

When a sharing set $sh \in SH$ over a set of variables of interest $\mathcal{V}$ includes the proper powerset of some subset $C \subseteq \mathcal{V}$ of variables, the representation can be made more compact since the powerset of C does not provide any useful information, i.e., all variables of C may share each other. This situation is illustrated in the following example:

Example 6.2.1. (Useless sharing groups). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ the set of variables of interest. Let $sh \in SH$ be an abstract substitution $\{\{X_1\}, \{X_1, X_2\}, \{X_1, X_2, X_3\},$- $\{X_1, X_3\}, \{X_2\}, \{X_2, X_3\}, \{X_1, X_2, X_3\}, \{X_4\}\}$. A key observation is that nothing is known of the subset of variables $C = \{X_1, X_2, X_3\}$ since any aliasing may be possible in C. Therefore, we may define a more compact representation to group the powerset of C.

Definition 6.2.1. (Clique). A *clique* is a set of variables of interest, much the same as a sharing group, but a clique C represents all the sharing groups in $\wp^0(C)$. For a clique C, we will use $\downarrow C = \wp^0(C)$. Note that $\downarrow C$ denotes all the sharing that is implicitly represented in a clique C. ■

Definition 6.2.2. (Clique set). A *clique set* is a set of cliques. Let $CL = SH$ denote the set of all clique sets. For a clique set $cl \in CL$ we define $\Downarrow cl = \cup\{\downarrow C \mid C \in cl\}$. Note that $\Downarrow cl$ denotes all the sharing that is implicitly represented in a clique set cl. For a pair (cl, sh) of a clique set cl and a sharing set sh, the sharing that the pair represents is $\Downarrow cl \cup sh$. ■

Example 6.2.2. (Clique-Sharing representation). Assume the same set of variables of interest as in Example 6.2.1. Assume also the same set-sharing $sh = \{\{X_1\},$-$\{X_1, X_2\}, \{X_1, X_2, X_3\}, \{X_1, X_3\}, \{X_2\}, \{X_2, X_3\}, \{X_1, X_2, X_3\}, \{X_4\}\}$. Then, we can represent sh as a pair (cl, sh') where $cl = \{\{X_1, X_2, X_3\}\}$ and $sh' = \{\{X_4\}\}$.

Definition 6.2.3. (The Clique-Sharing Domain,SH^W). The Clique-Sharing domain is $SH^W = \{(cl, sh) \mid cl \in CL, sh \in SH\}$, i.e., the set of pairs of a clique set and a sharing set [120]. ■

An abstract unification operation $amgu^W$ is defined in [121] which uses a function $irrel : CL \times Term \times \longrightarrow CL$ (complement of rel), defined as:

$$irrel(cl, t) = \{ \ C \setminus \hat{t} \mid C \in cl \ \} \setminus \{\emptyset\}$$

which approximates the sharing not related to variables in t.

In [121], the following operations[1] are defined as counterparts in SH^W of the corresponding ones in Sharing, and proved correct with respect to their corresponding counterparts (Theorem 9.8, page 239). Let $(cl, sh) \in SH^W$, $(cl_1, sh_1) \in SH^W$, $(cl_2, sh_2) \in SH^W$:

$$rel((cl, sh), t) = (rel(cl, t), rel(sh, t))$$

$$irrel((cl, sh), t) = (irrel(cl, t), irrel(sh, t))$$

$$(cl_1, sh_1) \cup^W (cl_2, sh_2) = (cl_1 \cup cl_2, sh_1 \cup sh_2)$$

$$(cl_1, sh_1) \uplus (cl_2, sh_2) = (\ (cl_1 \uplus cl_2) \cup (cl_1 \uplus sh_2) \cup (sh_1 \uplus cl_2), sh_1 \uplus sh_2\ )$$

$$(cl, sh)^* = \begin{cases} (\emptyset, sh^*) & \text{if } cl = \emptyset \\ (\{\ \cup(cl \cup sh)\ \}, \emptyset) & \text{otherwise} \end{cases}$$

Definition 6.2.4. (Abstract unification, *amgu*^{W}). The abstract unification is a function $amgu^W : \mathcal{V} \times Term \times SH^W \to SH^W$ defined in [121] as:

$$amgu^W(x, t, clsh) = irrel(clsh, x = t)\ \cup^W\ (rel(clsh, x)\ \uplus rel(clsh, t))^*$$

■

In [121] the correctness of $amgu^W$ is also shown, which is reproduced here.

Theorem 6.2.1. *Let $(cl, ss) \in SH^W$, $sh \in SH$, equation $x = t$, $x \in V$ and $t \in Term$, and $amgu^W(x, t, (cl, ss)) = (cl^o, ss^o)$. If $\Downarrow cl \cup ss \supseteq sh$ then:*

$$\Downarrow cl^o \cup ss^o \supseteq amgu(x = t, sh)$$

Proof. See Appendix A.

By using the above definitions of the operations and a case analysis, $amgu^W$ can be also defined as:

[1]Note that the operations lifted to SH^W are named with the same symbol as their counterparts in Sharing, and also the same name *irrel* as defined before is used. Thus, we are overloading all symbols except $\cup$.

$$amgu^{W}(x,t,(cl,sh)) = \begin{cases} (\ cl\ , amgu(x,t,sh)) & \text{if } rel(cl,x) = rel(cl,t) = \emptyset \\ (\ irrel(cl, x=t))\ , & \text{if } rel(cl,x) = rel(sh,x) = \emptyset \\ \ irrel(sh, x=t)\ ) & \text{or } rel(cl,t) = rel(sh,t) = \emptyset \\ (\ irrel(cl, x=t) \cup & \text{otherwise} \\ \{\cup(rel(cl,x)\ \cup rel(cl,t) \cup & \\ \quad rel(sh,x)\ \cup\ rel(sh,t))\} & \\ \ ,\ irrel(sh, x=t)\ ) & \end{cases}$$

which is the abstract unification operation implemented and it is of course equivalent to the one in [121] as proved also in Appendix A.

6.3 The Clique-Sharing+Freeness Domain

Similarly to the Set-Sharing domain described in Section 5, the Clique-Sharing domain can also improve its accuracy by adding some freeness information about the set of variables of interest. Freeness can be introduced to the Clique-Sharing domain in the usual way [90], by including a component which tracks the variables which are known to be free.

Definition 6.3.1. (Clique-Sharing+Freeness domain, SHF^{W}). The Clique-Sharing+Freeness domain is $SHF^{W} = SH^{W} \times \mathcal{V}$, i.e., the Clique-Sharing domain, SH^{W}, augmented with a new component which tracks the variables which are free.

∎

A method to define an abstract unification function for SH^{W} with freeness and linearity is outlined in [121]. We have used an abstract unification operation $amgu^{sf}$ for SH^{W} with freeness which is a simplification of the corresponding operation which results from the application of such method.

The method in [121] is basically the one used above for $amgu^{W}$: define the counterparts for the basic operations and prove them correct. For freeness we will need the following: Let $clsh \in SH^{W}$, $clsh = (cl, sh)$, $t \in Term$,

$$\begin{aligned} \mathcal{W}(clsh) &= \cup(cl \cup sh) \\ lin^{s}(t) &\Leftrightarrow \forall y \in \hat{t} : [t]_y = 1 \qquad\qquad\qquad\qquad\qquad\qquad \wedge \\ &\qquad \forall z \in \hat{t} : y \neq z \rightarrow rel(cl, y) \cap rel(cl, z) = \emptyset \quad \wedge \\ &\qquad rel(sh, y) \cap rel(sh, z) = \emptyset \end{aligned}$$

Note again that checking emptiness of each pairwise intersection in the definition of $lin^{s}(t)$ (as in $lin(t)$) can be reduced to a more efficient equivalent condition: given $rel(clsh, t) = (rel(cl, t), rel(sh, t))$, for all $s \in rel(cl, t) \cup rel(sh, t)$ $|s \cap \hat{t}| = 1$.

Now $amgu^{sf}$ is defined simply by lifting $amgu^{f}$ by substituting each original operation by its counterpart.

Definition 6.3.2. (Abstract unification,$amgu^{sf}$). Abstract unification is a function $amgu^{sf} : \mathcal{V} \times Term \times SHF^{W} \rightarrow SHF^{W}$ given by $amgu^{sf}(x, t, (clsh, f)) = (clsh', f')$, where:

$$clsh' = \begin{cases} amgu^{sff}(x, t, clsh) & \text{if } x \in f \text{ or } t \in f \\ amgu^{sfl}(x, t, clsh) & \text{if } x \notin f,\ t \notin f \text{ but } \hat{t} \subseteq f \text{ and } lin^{s}(t) \\ amgu^{W}(x, t, clsh) & \text{otherwise} \end{cases}$$

$$amgu^{sff}(x, t, clsh) = irrel(clsh, x = t) \cup^{W} (rel(clsh, x) \bowtie rel(clsh, t))$$

$$amgu^{sfl}(x, t, clsh) = irrel(clsh, x = t) \cup^{W} (rel(clsh, x) \bowtie rel(clsh, t)^{*})$$

$$f' = \begin{cases} f & \text{if } x \in f, t \in f \\ f \setminus (\mathcal{W}(rel(clsh, x))) & \text{if } x \in f, t \notin f \\ f \setminus (\mathcal{W}(rel(clsh, t))) & \text{if } x \notin f, t \in f \\ f \setminus (\mathcal{W}(rel(clsh, x) \cup^{W} rel(clsh, t))) & \text{if } x \notin f, t \notin f \end{cases}$$

■

so that by using the above definitions of the operations the following is obtained:

$$\begin{aligned} amgu^{sff}(x,t,(cl,sh)) = \; (\;\; & irrel(cl, x=t) \cup \\ & ((rel(cl,x) \;\cup\; rel(sh,x)) \bowtie rel(cl,t)) \cup \\ & (rel(cl,x) \;\bowtie rel(sh,t)) \\ , \;\; & irrel(sh, x=t) \;\cup\; (rel(sh,x) \;\bowtie rel(sh,t)) \;\;) \end{aligned}$$

$$amgu^{sfl}(x,t,(cl,sh)) = \begin{cases} (\;\; irrel(cl,x=t) \;\cup\; (rel(cl,x) \;\bowtie rel(sh,t)^*) & \text{if } cl_t = \emptyset \\ , \;\; irrel(sh,x=t) \;\cup\; (rel(sh,x) \;\bowtie rel(sh,t)^*) \;\;) & \\ (\;\; irrel(cl,x=t) \cup & \text{otherwise} \\ \quad ((rel(cl,x) \;\cup\; rel(sh,x)) \bowtie & \\ \quad \{\cup(rel(cl,t) \;\cup\; rel(sh,t))\}) & \\ \quad \cup(rel(cl,x) \;\bowtie rel(sh,t)^*) & \\ , \;\; irrel(sh,x=t) \;\;) & \end{cases}$$

Theorem 6.3.1. *Let $((cl,ss),f) \in SHF^{W}$, $(sh,e) \in SHF$, and equation $x = t$, $x \in \mathcal{V}$, $t \in Term$. Let also $amgu^{sf}(x,t,((cl,ss),f)) = ((cl^o,ss^o),f^o)$ and $amgu^{f}(x,t,(sh,e)) = (sh',f')$. If $\Downarrow cl \cup ss \supseteq sh$ and $f \subseteq e$ then:*

$$\Downarrow cl^o \cup ss^o \supseteq sh' \text{ and } f^o \subseteq f'$$

Proof. See Appendix A.

6.4 Abstract Functions for Top-Down Analysis in the Clique Domains

Functions *call2entry* and *exit2succ* have usually been defined in a way which is specific to the domain and for top-down analysis (see, as mentioned before, [91] for a definition for Set-Sharing). We have chosen instead to present here a formalization of a way to use the *amgu* in top-down frameworks. Thus, the definitions of *call2entry* and *exit2succ* based on *amgu* given above. Our intuition in doing this is that the results should be (more) comparable to goal-dependent bottom-up analyses, where *amgu* is used directly.

Note, however, that such definitions imply a possible loss of precision. Using *amgu* in the way explained above does not allow to take advantage of the fact that all variables in the head of the clause being entered during analysis are free. Alternative definitions of *call2entry* can be obtained that improve precision from this observation.[2] The overall effect would be equivalent to using the *amgu* function for the Sharing domain coupled with freeness, with the head variables as free variables, and then throwing out the freeness component of the result. For example, for the Clique-Sharing domain a function $call2entry^s$ that takes advantage of freeness information can be defined as follows, where $unify^{sf}$ is the version of $unify$ that uses $amgu^{sf}$:

$$\begin{array}{lrcl} & call2entry^s(ASub, Goal, Head) & = & ASub' \\ \text{where} & (ASub', Free) & = & unify^f((ASub, \emptyset), Head, Goal) \end{array}$$

However, for the reasons mentioned above, we have used the definitions of *call2entry* and *exit2succ* based on *amgu*. The rest of the top-down functions are defined below. For the Clique-Sharing domain, let $g \in Term$, and $(cl, sh) \in SH^W$. Functions $project^s$ and $augment^s$ are defined as follows:

$$project^s(g, (cl, sh)) = (project(g, cl), project(g, sh))$$
$$augment^s(g, (cl, sh)) = (cl, augment(g, sh))$$

Function $extend^s(Call, g, Prime)$ is defined as follows. Let $Call = (cl_1, sh_1)$ and $Prime = (cl_2, sh_2)$. Let *normalize* be a function which normalizes a pair (cl, sh)

[2]For example, one such definition (developed independently) can be found in [5].

so that no powersets occur in sh (all are "transferred" to cliques in cl; Section 6.5 presents a possible implementation of such a function). Let *Prime* be already normalized, and:

$$(cl', sh') = normalize((rel(cl_1, g)^* \cup (rel(cl_1, g)^* \bowtie rel(sh_1, g)^*), rel(sh_1, g)^*))$$

The following two functions lift the classical *extend* [91] respectively to the cases of the two clique sets (clique groups of the *Call* allowed by the clique component of the *Prime*) and of the two sharing sets (sharing groups belonging to the *Call* allowed by the sharing part of the *Prime*):

$$extsh(sh_1, g, sh_2) = irrel(sh_1, g) \cup \{ s \mid s \in sh',\ (s \cap \hat{g}) \in sh_2 \}$$
$$extcl(cl_1, g, cl_2) = irrel(cl_1, g) \cup \{ (s' \cap s) \cup (s' \setminus \hat{g}) \mid s' \in cl',\ s \in cl_2 \}$$

The following two functions account respectively for the sharing sets belonging to the clique component of the *Call* allowed by the sharing part of the *Prime*, and the sharing sets of the sharing component of the *Call* allowed by the clique part of the *Prime*:

$$clsh(cl', g, sh_2) = \{ s \mid s \subseteq c \in cl',\ (s \cap \hat{g}) \in sh_2 \}$$
$$shcl(sh', g, cl_2) = \{ s \mid s \in sh',\ (s \cap \hat{g}) \subseteq c \in cl_2 \}$$

The *extend* function for the Clique-Sharing domain is thus:

$$\begin{array}{l} extend^s((cl_1, sh_1), g, (cl_2, sh_2)) \ = \\ (\ \ extcl(cl_1, g, cl_2) \\ ,\ \ extsh(sh_1, g, sh_2) \cup clsh(cl', g, sh_2) \cup shcl(sh', g, cl_2)\ \ ) \end{array}$$

Example 6.4.1. (Extend for the Clique-Sharing domain). Let $Call = (cl_1, sh_1) = (\{\{X, Y, Z\}\}, \{\{U, V\}\})$, $Prime = (cl_2, sh_2) = (\{\{X\}\}, \{\{U, V\}\})$, and $\hat{g} = \{X, U, V\}$.

Then we have $(cl', sh') = (\{\{X,Y,Z,U,V\}\}, \emptyset)$. The $extend^s$ function is computed as follows:

$$\begin{array}{lcllcl}
extsh(sh_1, g, sh_2) & = & extsh(\{\{U,V\}\}, g, \{\{U,V\}\}) & = & \emptyset \\
extcl(cl_1, g, cl_2) & = & extcl(\{\{X,Y,Z\}\}, g, \{\{X\}\}) & = & \{\{X,Y,Z\},\{Y,Z\}\} \\
clsh(cl', g, sh_2) & = & clsh(\{\{X,Y,Z,U,V\}\}, g, \{\{U,V\}\}) & = & \{\{Y,Z,U,V\}, \\
 & & & & \{Y,U,V\}, \\
 & & & & \{Z,U,V\},\{U,V\}\} \\
shcl(sh', g, cl_2) & = & shcl(\emptyset, g, \{\{X\}\}) & = & \emptyset
\end{array}$$

Thus, $extend^s(Call, g, Prime) = (\{\{X,Y,Z\},\{Y,Z\}\}, \{\{Y,Z,U,V\},\{Y,U,V\},\{Z,U,V\},\{U,V\}\})$, which after normalization yields $(\{\{X,Y,Z\}\}, \{\{Y,Z,U,V\},\{Y,U,V\},\{Z,U,V\},\{U,V\}\})$.

Note how the result is less precise than the exact result $(\{\{X,Y,Z\}\}, \{\{U,V\}\})$. This is due to the overestimation of sharing implied by the cliques; in particular, for the case of *extend*, overestimations stem mainly from the necessary worst-case assumption given by (cl', sh'), which is then "pruned" as much as possible by the functions defined above. The resulting operation, however, is correct: the sharing implied by $extend^s$ on two abstract substitutions $Call$ and $Prime$ is an over-approximation of that given by *extend* on the sharing set substitutions corresponding to $Call$ and $Prime$.

Theorem 6.4.1. *Let $Call \in SH^W$, $Prime \in SH^W$, and $g \in Term$, such that the conditions for the extend function are satisfied. Let $Call = (cl_1, ss_1)$, $Prime = (cl_2, ss_2)$, $extend^s(Call, g, Prime) = (cl, ss)$, $\Downarrow cl_1 \cup ss_1 \supseteq sh_1$, and $\Downarrow cl_2 \cup ss_2 \supseteq sh_2$ then:*

$$\Downarrow cl \ \cup \ ss \ \supseteq \ extend(sh_1, g, sh_2)$$

Proof. See Appendix A.

For the Clique-Sharing+Freeness domain, let $g \in Term$, and $s \in SHF^W$, $s = ((cl, sh), f)$. Functions $project^{sf}$ and $augment^{sf}$ are defined as follows:

$$\begin{aligned}
&project^{sf}(g, s) = (project^{s}(g, (cl, sh)), f \ \cap \ \hat{g}) \\
&augment^{sf}(g, s) = (augment^{s}(g, (cl, sh)), f \ \cup \ \hat{g})
\end{aligned}$$

Function $extend^{sf}(Call, g, Prime)$ is defined as follows. Let $Call = ((cl_1, sh_1), f_1)$ and $Prime = ((cl_2, sh_2), f_2)$, $extend^{sf}(Call, g, Prime) = ((cl', sh'), f')$, where:

$$\begin{aligned}
&(cl', sh') = extend^{s}((cl_1, sh_1), g, (cl_2, sh_2)) \\
&f' = f_2 \ \cup \ \{x \mid x \ \in (f_1 \ \setminus \ \hat{g}), \ ((\cup(rel(sh', x) \ \cup \ rel(cl', x))) \ \cap \ \hat{g}) \ \subseteq f_2\}
\end{aligned}$$

Theorem 6.4.2. *Let $Call \in SHF^W$, $Prime \in SHF^W$, and $g \in Term$, such that the conditions for the extend function are satisfied. Let $Call = ((cl_1, sh_1), f_1)$, $Prime = ((cl_2, sh_2), f_2)$, and $extend^{sf}(Call, g, Prime) = ((cl', sh'), f')$. Let also $s_1 = \Downarrow cl_1 \cup sh_1$, $s_2 = \Downarrow cl_2 \cup sh_2$, and $extend^{f}((s_1, f_1), g, (s_2, f_2)) = (sh, f)$. Then $(\Downarrow cl' \cup sh') \supseteq sh$ and $f' \subseteq f$.*

Proof. See Appendix A.

Therefore, the operation $extend^{sf}$ is correct: it gives safe approximations. The resulting sharing it implies when applied on two abstract substitutions $Call$ and $Prime$ is no less than that given by $extend^{f}$ on the sharing set substitutions corresponding to $Call$ and $Prime$; and the freeness is no more than what $extend^{f}$ would have computed.

6.5 Detecting Cliques

Obviously, to minimize the representation in SH^W it pays off to replace any set S of sharing groups which is the proper powerset of some set of variables C by including C as a clique. Once this is done, the set S can be eliminated from the sharing set, since the presence of C in the clique set makes S redundant. This is the normalization mentioned in Section 6.4 when defining *extend* for the Clique-Sharing domain, and denoted there by a *normalize* function. In this section we present an algorithm for such a normalization.

Given an element $(cl, sh) \in SH^W$, sharing groups might occur in sh which are already implicit in cl. Such groups are redundant with respect to the sharing represented by the pair. We say that an element $(cl, sh) \in SH^W$ is *minimal* if $\Downarrow cl \cap sh = \emptyset$. An algorithm for minimization is straightforward: it should delete from sh all sharing groups which are a subset of an existing clique in cl. But normalization goes a step further by "moving sharing" from the sharing set of a pair to the clique set, thus forcing redundancy of some sharing groups (which can therefore be eliminated).

While normalizing, it turns out that powersets may exist which can be obtained from sharing groups in the sharing set plus sharing groups implied by existing cliques in the clique set. The representation can be minimized further if such sharing groups are also "transferred" to the clique set by adding the adequate clique. We say that an element $(cl, sh) \in SH^W$ is *normalized* if whenever there is an $s \subseteq (\Downarrow cl \cup sh)$ such that $s = \downarrow c$ for some set c then $s \cap sh = \emptyset$.

Our normalization algorithm is presented in Figure 6.1. It starts with an element $(cl, sh) \in SH^W$, which is already minimal, and obtains an equivalent element (w.r.t. the sharing represented) which is normalized and minimal. First, the number m is computed, which is the length of the longest possible clique. Then the sharing set sh is traversed to obtain candidate cliques of the greatest possible length i (which starts in m and is iteratively decremented). Existing subsets of a candidate clique S of length i are extracted from sh. If there are $2^i - 1 - [S]$ subsets of S in sh then S is a clique: it is added to cl and its subsets deleted from sh. Note that the test is performed on the number of existing subsets, and requires the computation of a number $[S]$, which is crucial for the correctness of the test.

```
1:   Let $n = |sh|$; if $n < 3$, stop
2:   Compute the maximum $m$ such that $n \geq 2^m - 1$
3:   Let $i = m$
4:   **if** $i = 1$, stop
5:   Let $C = \{s \mid s \in sh, |s| = i\}$
6:   **if** $C = \emptyset$ **then** decrement $i$ and **goto** 4
7:   Take $S \in C$ and delete it from $C$
8:   Let $SS = \{s \mid s \in sh, s \subseteq S\}$
9:   Compute $[S]$
10:  **if** $|SS| = 2^i - 1 - [S]$ **then**
         Add $S$ to $cl$ (regularize $cl$)
         Subtract $SS$ from $sh$
11:      **goto** 6
```

Figure 6.1: Algorithm for detecting cliques

The number $[S]$ stands for the number of subsets of S which may not appear in sh because they are already represented in cl (i.e., they are already subsets of an existing clique). In order to correctly compute this number it is essential that the input to the algorithm be already minimal; otherwise, redundant sharing groups might bias the calculation: the formula below may count as not present in sh a (redundant) group which is in fact present. The computation of $[S]$ is as follows. Let $I = \{S \cap C \mid C \in cl\} \setminus \{\emptyset\}$ and $A_i = \{\cap A \mid A \subseteq I,\ |A| = i\}$. Then:

$$[S] = \sum_{1 \leq i \leq |I|} (-1)^{i-1} \sum_{A \in A_i} (2^{|A|} - 1)$$

Note that the representation can be minimized further by eliminating cliques which are redundant with other cliques. This is the regularization mentioned in step 10 of the algorithm. We say that a clique set cl is *regular* if there are no two cliques $c_1 \in cl$, $c_2 \in cl$, such that $c_1 \subset c_2$. This can be tested while adding cliques in step 10 above.

Finally, there is a chance for further minimization by considering as cliques candidate sets of variables such that not all of their subsets exist in the given element of SH^W. Note that the algorithm preserves precision, since the sharing represented by the element of SH^W input to the algorithm is the same as that represented by the element which is output. However, we could set up a threshold for the number of subsets of the candidate clique that need be detected, and in this case the output element may in general represent more sharing. This might in fact be useful in practice in order to use the normalization algorithm as a widening operation. Note that, although the complexity of this algorithm is exponential since it is actually the problem of solving all the maximal cliques of an undirected graph (NP-complete), it is not a practical problem due to the small size of these graphs.

6.6 Widening Set-Sharing

A *widen* function for SH^W is based on an unary widening operator $\bigtriangledown : SH^W \rightarrow SH^W$, which must guarantee that for each $clsh \in SH^W$, $\bigtriangledown clsh \supseteq clsh$[3]. The following theorem is necessary to establish the correctness of the widenings used:

Theorem 6.6.1. *Let $clsh \in SH^W$ and equation $x = t$, $x \in V$, $t \in Term$, we have*

$$amgu^W(x, t, \bigtriangledown clsh) \supseteq amgu^W(x, t, clsh)$$

For our experiments we start using two widenings already defined. The first of them, by [43], is of an intermediate precision and it is as follows:

$$\overset{F}{\bigtriangledown}(cl, sh) = (cl \cup sh, \emptyset)$$

[3]Note that this definition of widening for sharing is slightly different from original Definition 3.3.1.

The second widening was defined in [120] as a cautious widening (because it did not introduce new sharing sets, although obviously information was lost as soon as the operations for the Clique-Sharing domain were used) and the idea was to define an undirected graph from an element $clsh \in SH^W$ and compute the maximal cliques of that graph:

$$\overset{G}{\bigtriangledown}(cl, sh) = (\{C_1, \ldots, C_k\}, sh)$$

where $C_1, \ldots, C_k$ are all the maximal cliques of the induced graph from (cl, sh). For the experimental evaluation in [120] a version of this cautious widening $\bigtriangledown^g$ was used which is equivalent to the previous one but disregarding the singletons. It is easy to see that our normalization process is totally equivalent to the computation of the maximal cliques of a graph and thus we will use the normalization process as a cautious widening $\bigtriangledown^N$. In the same way as [120], we use a more precise version of $\bigtriangledown^N$ which is based on disregarding the singletons called $\bigtriangledown^n$.

Since cliques should only be used when it is strictly necessary to keep the analysis from running out of memory, its application is guarded by a condition. We use the simplest possible condition based on cardinality of the sets in SH^W, imposing a threshold n on cardinality which triggers the widening. We have tuned the threshold in order to be able to achieve a reasonable trade-off between the objective of triggering widening only when strictly required and preventing running out of memory in all cases. For each widening, the triggering condition is defined as follows:

$$widen(cl, sh) = \begin{cases} \bigtriangledown(cl, sh) & \text{if } (\sum_{s \in sh} |s|) > n \\ (cl, sh) & \text{otherwise} \end{cases}$$

6.7 Experimental Results

We have measured experimentally the relative efficiency and precision obtained with the inclusion of cliques both as an alternative representation in the Set-Sharing and Set-Sharing+Freeness domains and as a widening in the Set-Sharing+Freeness domain. Our first objective is to study the implications of the change in representation for analysis: although the introduction of cliques does not by itself imply a loss of precision, the abstract operations for cliques are not precise. We first want to measure such loss in practice. Second, to minimize precision loss, the clique representation should ideally be used only whenever necessary, i.e., when the classical representation cannot deal with the analysis of the program at hand. In this case, we will be using the clique representation as a widening to guarantee (smooth) termination of the analysis, i.e., that analysis does not abort because of running out of memory. It turns out that this is not a trivial task: it is not easy to determine beforehand when analysis will need more memory than is available.

Benchmarks are divided into three groups.

- The first group, append (app in the tables) through serialize (serial), is a set of simple programs, used as a testbed for an analysis: they have only direct recursion and make a straightforward use of unification (basically, for input/output of arguments i.e., they are moded).
- The second group, aiakl through zebra, are more involved: they make use of mutual recursion and of elaborate aliasing between arguments to some extent; some of them are parts of "real" programs (aiakl is part of an analyzer of the AKL language; prolog_read (plread) and rdtok are Prolog parsers).
- The benchmarks in the third group are all (parts of) "real" programs: ann is the &-prolog parallelizer, peephole (peep) is the peephole optimizer of the

SB-Prolog compiler, qplan is the core of the Chat-80 application, and witt is a conceptual clustering application.

Our results are shown in Tables 6.1, 6.2 and 6.3. Columns labeled **T** show analysis times in milliseconds, on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 2.0, and averaging several runs after eliminating the best and worst values. Ciao version 1.11#326 and CiaoPP 1.0#2292 were used. Columns labeled **P** (precision) show the number of sharing groups in the information inferred and, between parenthesis, the number of sharing groups for the worst-case sharing. Columns labeled **#W** show the number of widenings performed and columns labeled **#C** show the number of clique groups. Since our top-down framework infers information at all program points (before and after calling each clause body atom), and also several variants for each program point, it is not trivial to provide a good absolute measure of precision: changes in precision may cause more variants during analysis, which in turn affect the precision measure. Instead, we have chosen to provide the accumulated number of sharing groups in all variants for all program points, in order to be able to compare results in different situations.

6.7.1 Cliques as Alternative Representation

Tables 6.1 and 6.2 show the results for Set-Sharing, Clique-Sharing and Sharing+Freeness, and Clique-Sharing+Freeness, respectively for the cases in which cliques are used as an alternative representation.

In order to understand the results it is important to note an existing synergy between normalization, efficiency, and precision when cliques are used as an alternative representation. If normalization causes no change in the sharing representation (i.e., sharing groups are not moved to cliques), usually because powersets do not

	Sh		SH^W		
	T	**P**	**T**	**P**	$\#C$
app	11	29 (60)	8	44 (60)	4
deriv	35	27 (546)	27	27 (546)	0
mmat	13	14 (694)	11	14 (694)	0
qsort	24	30 (1716)	25	30 (1716)	0
query	11	35 (501)	13	35 (501)	5
serial	306	1734 (10531)	90	2443 (10531)	88
aiakl	35	145 (13238)	42	145 (13238)	0
boyer	369	1688 (4631)	267	1997 (4631)	158
brow	30	69 (776)	29	69 (776)	0
plread	400	1080 (408755)	465	1080 (408755)	10
rdtok	325	1350 (11513)	344	1391 (11513)	182
wplan	3261	8207 (42089)	1430	8191 (26857)	420
zebra	25	280 ($67{\cdot}10^7$)	34	280 ($67{\cdot}10^7$)	0
ann	2382	10000 ($31{\cdot}10^4$)	802	19544 ($31{\cdot}10^4$)	700
peep	831	2210 (12148)	435	2920 (12118)	171
qplan	∞		860	$42{\cdot}10^4$ ($38{\cdot}10^5$)	747
witt	405	858 ($45{\cdot}10^5$)	437	858 ($45{\cdot}10^5$)	25

Table 6.1: Precision and Time-efficiency for Sharing and Clique-Sharing

really occur during analysis, then the clique part is empty. Analysis is the same as without cliques, but with the extra overhead due to the use of the normalization process. Then precision is the same but the time spent in analyzing the program is a little longer. This also occurs often if the use of normalization is kept to a minimum: only for correctness (in our implementation, normalization is required for correctness at least for the *extend* function and other functions used for comparing abstract substitutions). This should not be surprising, since the fact that powersets occur during analysis at a given time does not necessarily mean that they keep on occurring afterward: they can disappear because of groundness or other precision improvements during subsequent analysis (of, e.g., builtins).

When the normalization process is used more often (like for example at every

	Shfr		SH^W**fr**		
	T	**P**	**T**	**P**	$\#C$
app	6	7 (30)	6	7 (30)	0
deriv	27	21 (546)	27	21 (546)	0
mmat	9	12 (694)	11	12 (694)	0
qsort	25	30 (1716)	27	30 (1716)	0
query	12	22 (501)	14	22 (501)	0
serial	61	545 (5264)	55	736 (5264)	41
aiakl	37	145 (13238)	43	145 (13238)	0
boyer	373	1739 (5036)	278	2074 (5036)	163
brow	29	69 (776)	31	69 (776)	0
plread	425	1050 (408634)	481	1050 (408634)	0
rdtok	335	1047 (11513)	357	1053 (11513)	2
wplan	1320	3068 (23501)	1264	5705 (25345)	209
zebra	41	280 ($67{\cdot}10^7$)	42	280 ($67{\cdot}10^7$)	0
ann	1791	7811 (401220)	968	14108 ($39{\cdot}10^4$)	510
peep	508	1475 (9941)	403	2825 (12410)	135
qplan	∞	-	2181	$23{\cdot}10^4$ ($31{\cdot}10^5$)	529
witt	484	813 (4545594)	451	813 ($45{\cdot}10^5$)	0

Table 6.2: Precision and Time-efficiency for Sharing+freeness and Clique-Sharing+-freeness

call to *call2entry* as we have done), then sharing groups are moved more often to cliques. Thus, the use of the operations that compute on clique sets produces efficiency gains, and also precision losses, as it was expected. However, precision losses are not high. Finally, if normalization is used too often, then the analysis process suffers from heavy overhead, causing too high penalty in efficiency that it makes the analysis intractable. Therefore it is very clear that a thorough tuning of the use of the normalization process is crucial to lead analysis to good results in terms of both precision and efficiency.

As usual in top-down analysis, the *extend* function plays a crucial role. In our case, this function is a very important bottleneck for the use of normalization. As we have said, we use the normalization for correctness at the beginning of the *extend* function. Additionally, it would be convenient to use it also at the end of such function, since the number of sharing groups can grow too much. However, this is not possible in practice due to the *clsh* function, which can generate so many sharing groups that, at the limit, the normalization process itself cannot run. Alternative definitions of *clsh* have been studied, but because of the precision losses incurred, they have been found impractical.

Tables 6.1 and 6.2 shows that there are always programs whose analysis of which does not produce cliques. This occurs in some of the benchmarks (like all of the first group but serialize and some of the second one such as aiakl, browse (brow), prolog_read, and zebra). In this case, precision is maintained as expected but there is a small loss of efficiency due to the extra overhead discussed above. The same thing happens with benchmarks which produce cliques (append, query, prolog_read, and witt, in the case of Sharing without freeness), but this does not affect precision.

On the other hand, for those benchmarks which do generate cliques (like serialize, boyer, warplan (wplan), ann, and peephole) the gain in efficiency is considerable at the cost of a small precision loss. As usual, efficiency and precision correlate inversely: if precision increases then efficiency decreases and vice versa. A special case is, to some extent, that of rdtok, since precision losses are not coupled with efficiency gains. The reason is that for this benchmark there are extra success substitutions (which do not convey extra precision and, in fact, the result is less precise) that make the analysis times larger.

In general, the same effects are maintained with the addition of freeness, although the efficiency gains are lower whereas the precision gains are a little higher. The reason is that the $amgu^{sf}$ function is less efficient than $amgu^{s}$ (but more precise). Overall, however, the trade-off between precision and efficiency is beneficial. Moreover, the more compact representation of the clique domain makes possible to analyze benchmarks (e.g., qplan) which ran out of memory with the standard representation.

6.7.2 Widening Set-Sharing via Cliques

As mentioned before, the intention of the widening operator is to limit the use of cliques only to the cases where it is necessary in order to avoid analysis running out of memory. This is not a trivial task, as explained below. Table 6.3 shows results from our experiments for Sharing+Freeness, Clique-Sharing+Freeness using widening. The widenings have been applied before each abstract unification and at the end of the *extend* function, and they are guarded by the condition discussed in Section 6.6.

The choice of a suitable value of the threshold is a key issue, since this threshold is responsible for triggering widening only for the cases where it is needed. In a top-down framework the choice of threshold is further complicated by the *extend* function. As commented above, this function and, in particular, the *clsh* function defined in Section 6.4 can make the number of sharing groups grow excessively after every call, since that function generates powersets of the given cliques. In order to solve this problem we studied two different alternatives.

First, we tried a more efficient version of the *clsh* function, which moved some extra sharing groups to cliques. This, however, resulted in excessive precision losses which reduced the usefulness of the analysis. Given this, we also developed a hybrid approach for the case of ∇^n, where ∇^n is used in unifications but the more aggressive ∇^F is used after calling *clsh*. We call this version ∇^{nF}.

As for practical thresholds, we have concluded experimentally that an appropriate value for the guard for the widenings in our test platform is 250. This is the highest value that prevents analysis from running out of memory. However, as we will see, it also triggers widening for a few cases where it is not needed. For the additional threshold used in the ∇^{nF} operations (Section 6.5) we have determined that 40% is an appropriate value since, although low, it gives surprisingly good results.The results in Table 6.3 thus correspond to ∇^{F}_{250} and ∇^{nF}_{250-40}.

	SH^W**fr**$+\nabla^{F}_{250}$			SH^W**fr**$+\nabla^{nF}_{250-40}$		
	T	**P**	$\#W$	**T**	**P**	$\#W$
app	11	7 (30)	0	10	7 (30)	0
deriv	48	21 (546)	0	35	21 (546)	0
mmat	16	12 (694)	0	16	12 (694)	0
qsort	40	30 (1716)	0	43	30 (1716)	0
query	23	22 (501)	0	25	22 (501)	0
serial	74	722 (5264)	6	70	703 (5264)	10
aiakl	63	145 (13238)	6	61	145 (13238)	33
boyer	561	1744 (5036)	2	536	1743 (5036)	4
brow	44	69 (776)	0	42	69 (776)	0
plread	3419	24856 (1754310)	198	593	1050 (408634)	103
rdtok	472	1047 (11513)	0	466	1047 (11513)	0
wplan	1878	5376 (21586)	42	1394	5121 (20894)	60
zebra	42	280 ($67{\cdot}10^7$)	1	56	280 ($67{\cdot}10^7$)	48
ann	751	16122 (394800)	17	726	16122 (394800)	34
peep	453	2827 (12410)	8	512	2815 (12410)	16
qplan	1722	238426 (3141556)	26	1897	233070 (3126973)	55
witt	2333	259366 (23378597)	110	736	813 (4545594)	140

Table 6.3: Precision and Time-efficiency with Widening

As expected, the use of widening allows executing programs which the Shfr domain could not due to exceeded memory capacity. However, as mentioned in the discussion of the threshold, we do also widen for some benchmarks which the original domain could handle. Fortunately, the precision losses are limited.

Widening operator ∇^{nF}_{250-40} results at least as precise as ∇^{F}_{250} and, for most of the cases, better. In fact, the results obtained for prolog_read and witt using ∇^{F}_{250} are remarkable since the information obtained is very poor. The difference in time efficiency between ∇^{F}_{250} and ∇^{nF}_{250-40} is acceptable, and in fact for some programs

∇^{nF}_{250-40} is more efficient than ∇^{F}_{250}. Note that for prolog_read and witt the difference is considerable in favor of ∇^{nF}_{250-40}. There appears to be a clear correspondence between number of widenings and efficiency gains. This holds even if the widening operations are expensive, such as with ∇^{nF}_{250-40}, because the widening expense is offset by efficiency gains in the abstract operations due to the reduction in the size of the abstract substitutions being processed.

6.8 Summary

We have studied the problem of efficient, scalable Set-Sharing analysis of logic programs using cliques both as alternative representation and as widenings. We have concentrated on the previously unexplored case of inferring set-sharing information in the context of top-down analyses. To this end, we have proposed all the operations required for top-down analyses for the cases of combining cliques with both Sharing and Sharing+Freeness. We have also proposed and studied several widenings, providing different levels of precision and efficiency tradeoff.

Our experimental evaluation supports the conclusion that, for inferring set-sharing, the use of cliques as an alternative representation results in limited precision losses (due to normalizations) while efficiency gains are obtained. We have also derived interesting conclusions regarding the interactions between thresholds, precision, efficiency and cost of widening which have resulted in the proposal of a hybrid widening which resulted quite useful in practice. In fact, the new representations allowed analyzing some programs that exceeded memory capacity using classical sharing representations. Thus, we believe the results of this chapter contribute to the practical application of top-down analysis of Set-Sharing.

Chapter 7

Negative Set-Sharing Analysis

In Chapter 6, a new approach for improving the efficiency (in terms of memory and running time) of the process of inferring set-sharing information in top-down frameworks was presented. The technique relied on the use of several widenings which provided different levels of precision and efficiency tradeoff. However, sometimes there are situations where the loss of accuracy is not allowed by the application and/or more substantial efficiency gains are required.

In this chapter we introduce another novel approach to improving the efficiency of Set-Sharing, both in terms of memory and running time, in this case without any loss of accuracy. In the remainder of this chapter we first introduce the basis of the new approach in Section 7.1, redefine the Set-Sharing domain, described in Chapter 5, adapting it to a binary string representation (Section 7.2) which we then extend in Section 7.3 to use a more compact representation through a ternary encoding. In Section 7.4, we use the encoding of the complement (or negative) of the original Set-Sharing. Finally, results from an experimental evaluation of these representations are reported in Section 7.5 and summary in Section 7.6.

7.1 Introduction

The new approach for inferring set-sharing information efficiently without loss of accuracy is described as follows. We define a new representation that leverages the complement (or negative) sharing relationships of the original sharing set, and allows more compact representations for cases where there are many sharing sets without loss of accuracy. Intuitively, given an abstract state $sh_{\mathcal{V}}$ over the finite set of variables of interest $\mathcal{V}$, its negative representation is $\wp(\mathcal{V}) \setminus sh_{\mathcal{V}}$. Using this encoding during analysis dramatically reduces the number of elements that need to be represented in the abstract states and during abstract unification as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. To further compress the number of elements, we express the set-sharing relationships through a set of ternary strings that compacts the representation by eliminating redundancies among the sharing sets.

It is important to notice that our work is not based on [30]. Although they define the dual negated positive Boolean functions, $\overline{coPos}$ does not represent the entire complement of the positive set. Moreover, they do not use $\overline{coPos}$ as a means of compressing relationships but as a way of representing Sharing through Boolean functions. We also represent Sharing through Boolean functions, but that is where the similarity ends.

Example 7.1.1. (Negative sharing relationships). Let $\mathcal{V} = \{X_1, X_2, X_3\}$ be a set of variables of interest. Let $sh \in SH$ be an abstract substitution such that $sh = \{\{X_1\}, \{X_1, X_2, X_3\}, \{X_1, X_3\}, \{X_2\}, \{X_1, X_2, X_3\}\}$, $|sh| = 5$. Since that the set of variables $\mathcal{V}$ is finite, the computation of the set complement, i.e. $\wp(\mathcal{V}) \setminus sh$, is always possible. Therefore, the negative (or complement) image of sh, $\overline{sh}$, will be $\{\{X_1, X_2\}, \{X_2, X_3\}\}$ and its cardinality $|\overline{sh}| = 2$. Then, it is easy to see that, in certain cases, the size of the sharing relationships can be reduced by encoding their complement.

7.2 Set-Sharing Encoded by Binary Strings

In this section, we adapt the Set-Sharing abstract domain described in Chapter 5 for handling binary strings rather than sets of variables. Unless otherwise stated, here and in the rest of this chapter we will represent the Set-Sharing domain using a set of strings rather than a set of sets of variables.

Definition 7.2.1. (Binary sharing domain, bSH). Let alphabet $\Sigma = \{0, 1\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in arbitrary order, and Σ^l the finite set of all strings over Σ with length l, $0 \leq l \leq |\mathcal{V}|$. Let $bSH^l = \wp^0(\Sigma^l)$ be the *proper power set* (i.e., $\wp(\Sigma^l) \setminus \{\emptyset\}$) that contains all possible combinations over Σ with length l. Then, the *binary sharing domain* is defined as $bSH = \bigcup_{0 \leq l \leq |\mathcal{V}|} bSH^l$. ■

Example 7.2.1. (Binary encoding of sharing relationships). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ be the set of variables of interest and let $sh = \{\{X_1\}, \{X_1, X_2\}, \{X_1, X_2, X_3\}\}$ be a sharing set. Assume the following order among variables: $X_1 \prec X_2 \prec X_3 \prec X_4$. Then, we can encode each sharing group into a binary string using the algorithm described in Figure 7.1. In this example, the result of mapping sh into a set of binary strings is $bsh = \{1000, 1100, 1110\}$.

```
BinaryEncoding($sh, \mathcal{V}$)
    $bsh \leftarrow \emptyset$
    **foreach** $sg \in sh$
        **foreach** $i$-th variable of $\mathcal{V}$
            **if** the $i$-th variable of $\mathcal{V}$ appears in $sg$ **then**
                $s[i] \leftarrow 1$
            **else**
                $s[i] \leftarrow 0$
        $bsh \leftarrow bsh \cup \{s\}$
    **return** $bsh$
```

Figure 7.1: Simple algorithm for encoding binary sharing relationships

In this chapter, we will denote by $\hat{t}$ the binary representation of the set of variables of $t \in Term$ according to a particular order among variables. Since $\hat{t}$ will be always used through a bitwise operation with some string of length l, the length of $\hat{t}$ must be l. If not, $\hat{t}$ is adjusted with 0's in those positions associated with variables represented in the string but not in t. For instance, if $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ and t contains $\{X_2, X_3\}$, then $\hat{t} = 0110$.

The following definitions are an adaptation for the binary representation of the standard definitions for the Set-Sharing domain:

Definition 7.2.2. (Binary relevant sharing $rel(bsh, t)$ and irrelevant sharing $irrel(bsh, t)$). Given $t \in Term$, the set of binary strings in $bsh \in bSH^l$ of length l that are relevant with respect to t is obtained by a function $rel(bsh, t) : bSH^l \times Term \to bSH^l$ defined as:

$$rel(bsh, t) = \{s \mid s \in bsh, (s \bigwedge \hat{t}) \neq 0^l\}$$

where $\bigwedge$ represents the bitwise AND operation and 0^l is the all-zeros string of length l. Consequently, the set of binary strings in $bsh \in bSH^l$ that are *irrelevant with respect to t is a function* $irrel(bsh, t) : bSH^l \times Term \to bSH^l$ *where* $irrel(bsh, t)$ *is the complement of* $rel(bsh, t)$, *i.e.,* $bsh \setminus rel(bsh, t)$. ■

Definition 7.2.3. (Binary cross-union, $\bowtie$). Given $bsh_1, bsh_2 \in bSH^l$, their *cross-union* is a function $\bowtie : bSH^l \times bSH^l \to bSH^l$ defined as

$$bsh_1 \bowtie bsh_2 = \{s \mid s = s_1 \bigvee s_2, s_1 \in bsh_1, s_2 \in bsh_2\}$$

where $\bigvee$ represents the bitwise OR operation. ■

Definition 7.2.4. (Binary up-closure, $(.)^*$). Let l be the length of strings in $bsh \in bSH^l$, then the *up-closure* of bsh, denoted bsh^* is a function $(.)^* : bSH^l \to$

bSH^l that represents the smallest superset of bsh such that $s_1 \bigvee s_2 \in bsh^*$ whenever $s_1, s_2 \in bsh^*$:

$$bsh^* = \{s \mid \exists n \geq 1 \; \exists t_1, \ldots, t_n \in bsh, \; s = t_1 \bigvee \ldots \bigvee t_n\}$$

■

Definition 7.2.5. (Binary abstract unification, *amgu*). The abstract unification is a function $amgu : \mathcal{V} \times Term \times bSH^l \rightarrow bSH^l$ defined as

$$amgu(x, t, bsh) = irrel(bsh, x = t) \;\cup\; (rel(bsh, x) \bowtie rel(bsh, t))^*$$

■

Example 7.2.2. (Binary abstract unification). Let $\mathcal{V} = \{X_1, X_2, X_3, X_4\}$ be the set of variables of interest and let $sh = \{\{X_1\}, \{X_2\}, \{X_3\}, \{X_4\}\}$ be a sharing set. Assume the following order among variables: $X_1 \prec X_2 \prec X_3 \prec X_4$. Then, sh is encoded as the following set of binary strings $bsh = \{1000, 0100, 0010, 0001\}$. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$\begin{array}{lcl}
A = rel(bsh, X_1) & = & \{1000\} \\
B = rel(bsh, f(X_2, X_3)) & = & \{0100, 0010\} \\
A \bowtie B & = & \{1100, 1010\} \\
(A \bowtie B)^* & = & \{1100, 1010, 1110\} \\
C = irrel(bsh, X_1 = f(X_2, X_3)) & = & \{0001\} \\
amgu(X_1, f(X_2, X_3), bsh) = C \;\cup\; (A \bowtie B)^* & = & \{0001, 1100, 1010, 1110\}
\end{array}$$

As described in Sec. 3.4 in Chapter 3, the design of a bottom-up analysis must be completed by defining the following abstract operations: *init* (initial abstract state), *equivalence* (between two abstract substitutions), *join* (defined as the union), and *project*.

Definition 7.2.6. (Binary initial state, $init_{bSH}$). The *initial state init* : $\mathcal{V} \rightarrow bSH$ describes an initial substitution given a set of variables. Assume that an initial substitution $sh \in SH$ is given by $init_{SH} : \mathcal{V} \rightarrow SH$, defined in [61]. Then, the binary initial state can be defined using the algorithm shown in Fig. 7.1 as:

$$init_{bSH}(\mathcal{V}) = \mathsf{BinaryEncoding}(init_{SH}(\mathcal{V}), \mathcal{V})$$

■

Definition 7.2.7. (Binary equivalence, $\equiv$). Given $bsh_1, bsh_2 \in bSH$, they are *equivalent* (i.e., $bsh_1 \equiv bsh_2$) if and only if $\forall s_1 \in bsh_1, \forall s_2 \in bsh_2, s_1 = s_2$ (syntactic equivalence). ■

Definition 7.2.8. (Binary join, $\sqcup$). Given $bsh_1, bsh_2 \in bSH$, the *join* function $\sqcup : bSH \times bSH \rightarrow \wp(bSH)$ is defined as union (i.e., $bsh_1 \sqcup bsh_2 = bsh_1 \cup bsh_2$). ■

Definition 7.2.9. (Binary projection, $bsh|_t$). The *binary projection* is a function $bsh|_t$: $bSH^l \times Term \rightarrow bSH^k$ $(k \leq l)$ that removes the i-th positions from all strings (of length l) in $bsh \in bSH^l$, if and only if the i-th positions of $\hat{t}$ (denoted by $\hat{t}[i]$) is 0, and it is defined as

$$bsh|_t = \{s' \mid s \in bsh, s' = \pi(s,t)\}$$

where $\pi(s,t)$ is the binary string projection defined as

$$\pi(s,t) = \begin{cases} \epsilon, & \text{if } \ s = \epsilon, \text{the empty string} \\ \pi(s',t), & \text{if } \ s = s'a_i \ and \ \hat{t}[i] = 0 \\ \pi(s',t)a_i, & \text{if } \ s = s'a_i \ and \ \hat{t}[i] = 1 \end{cases}$$

and $s'a_i$ is the concatenation of character a to string s' at position i.

■

7.3 Ternary Set-Sharing

In this section, we introduce a more efficient representation for the Set-Sharing domain defined in Sec. 7.2 to accommodate a larger number of variables for analysis. We extend the binary string encoding discussed above to the ternary alphabet $\Sigma_* = \{0, 1, *\}$, where the $*$ symbol denotes both 0 and 1 bit values. This representation effectively compresses the number of elements in the set into fewer strings without changing what is represented (i.e., without loss of accuracy). To handle the ternary alphabet, we redefine the binary operations covered in Sec. 7.2.

Definition 7.3.1. (Ternary sharing domain, tSH). Let alphabet $\Sigma_* = \{0, 1, *\}$, $\mathcal{V}$ be a fixed and finite set of variables of interest in an arbitrary order as in Def. 7.2.1, and Σ_*^l the finite set of all strings over Σ_* with length l, $0 \le l \le |\mathcal{V}|$. Then, $tSH^l = \wp^0(\Sigma_*^l)$ and hence, the *ternary sharing domain* is defined as $tSH = \bigcup_{0 \le l \le |\mathcal{V}|} tSH^l$.

■

Prior to defining how to transform the binary string representation into the corresponding ternary string representation, we introduce two core definitions, Def. 7.3.2 and Def. 7.3.3, for comparing ternary strings. These operations are essential for the conversion and set operations. In addition, they are used to eliminate redundant strings within a set and to check for equivalence of two ternary sets containing different strings.

Definition 7.3.2. (Match, $\mathcal{M}$). Given two ternary strings, $x, y \in \Sigma_*^l$, of length l, *match* is a function $\mathcal{M} : \Sigma_*^l \times \Sigma_*^l \to \mathcal{B}$, such that $\forall i \; 1 \le i \le l$,

$$x \mathcal{M} y = \begin{cases} \mathsf{true}, \text{if } \; (x[i] = y[i]) \vee (x[i] = *) \vee (y[i] = *) \\ \mathsf{false}, \text{otherwise} \end{cases}$$

■

```
Convert($bsh, k$)
$tsh \leftarrow \emptyset$
foreach $s \in bsh$
    $y \leftarrow$ PatternGenerate($tsh, s, k$)
    $tsh \leftarrow$ ManagedGrowth($tsh, y$)
return $tsh$
```

```
PatternGenerate($tsh, x, k$)
$m \leftarrow$ Specified($x$)
$i \leftarrow 0$
$x' \leftarrow x$
$l \leftarrow length(x)$
while $m > k$ and $i < l$
    Let $b_i$ be the value of $x'$ at position $i$
    if $b_i = 0$ or $b_i = 1$ then
        $x' \leftarrow x' \cdot \overline{b_i}$
        if $x' \preceqq tsh$ then
            $x' \leftarrow x' \cdot *_i$
        else
            $x' \leftarrow x' \cdot b_i$
    $m \leftarrow$ Specified($x'$)
    $i \leftarrow i + 1$
return $x'$
```

```
ManagedGrowth($tsh, y$)
$S_y = \{s \mid s \in tsh, s \preceq y\}$
if $S_y = \emptyset$ then
    if $y \not\preceqq tsh$ then
        append $y$ to $tsh$
else
    remove $S_y$ from $tsh$
    append $y$ to $tsh$
return $tsh$
```

Figure 7.2: A deterministic algorithm for converting a set of binary strings bsh into a set of ternary strings tsh, where k is the desired minimum number of specified bits (non-$*$) to remain.

Definition 7.3.3. (Subsumed_By $\preceq$ and Subsumed_In $\preceqq$). Given two ternary strings $s_1, s_2 \in \Sigma_*^l$, $\preceq : \Sigma_*^l \times \Sigma_*^l \to \mathcal{B}$ is a function such that $s_1 \preceq s_2$ if and only if every string matched by s_1 is also matched by s_2. More formally, $s_1 \preceq s_2 \iff \forall s \in tSH^l$, $if\ s_1 \mathcal{M} s\ then\ s_2 \mathcal{M} s$. For convenience, we augment this definition to deal with sets of strings. Given a ternary string $s \in \Sigma_*^l$ and a ternary sharing set, $tsh \in tSH^l$, $\preceqq : \Sigma_*^l \times tSH^l \to \mathcal{B}$ is a function such that $s \preceqq tsh$ if and only if there exists some element $s' \in tsh$ such that $s \preceq s'$. ∎

Figure 7.2 gives the pseudo code for an algorithm which converts a set of binary strings into a set of ternary strings. The function Convert evaluates each string of the input and attempts to introduce $*$ symbols using PatternGenerate, while eliminating redundant strings using ManagedGrowth.

PatternGenerate evaluates the input string bit-by-bit to determine where the $*$ symbol can be introduced. The number of $*$ symbols introduced depends on the sharing set represented and k, the desired minimum number of specified bits, where $1 \leq k \leq l$ (the string length). For a given set of strings of length l, parameter k controls the compression of the set. For $k = l$ (all bits specified), there is no compression and $tsh = bsh$. For $k = 1$, the maximum number of $*$ symbols is introduced. For now, we will assume that $k = 1$, and some experimental results in Section 7.5 will show the best overall k value for a given l. The Specified function returns the number of specified bits (0 or 1) in x.

ManagedGrowth checks if the input string y subsumes other strings from tsh. If no redundant string exists, then y is appended to tsh only if y itself is not redundant to an existing string in tsh. Otherwise, y replaces all the redundant strings.

Example 7.3.1. (Conversion from bSH to tSH). Let $\mathcal{V}$ be the set of variables of interest with the same order as Example 7.2.2. Assume the following sharing set of binary strings bsh = {1000, 1001, 0100, 0101, 0010, 0001}. Then, a ternary string representation produced by applying Convert is tsh ={100*, 0010, 010*, *001}. There can be a certain level of redundancy in the representation, a subject that will be discussed further in Section 7.5.

The example above begins with Convert(bsh,$k = 1$).

1. Since $tsh = \emptyset$ initially (line 1), the first string 1000 is appended to tsh, so $tsh = \{1000\}$.
2. Next, 1001 from bsh is evaluated. In PatternGenerate, with x' at iteration i

(denoted as x'_i), $i = 3$ and $b_3 = 1$, we test $x'_3 = 1000$ if the i^{th} position of x can be replaced with a $*$ (line 15-24). In this case, since $x'_3 \subseteqq tsh$ (line 19), $x'_3 =$ 100* is returned (line 25). Next, ManagedGrowth evaluates 100* and since it subsumes 1000 ($S_y = \{1000\}$), 100* replaces 1000 leaving $tsh = \{100*\}$ (line 38).

3. The process continues with PatternGenerate({100*},0100) (line 3). In PatternGenerate, since $x'_0 \not\subseteqq tsh$, $x'_1 \not\subseteqq tsh$, $x'_2 \not\subseteqq tsh$, and $x'_3 \not\subseteqq tsh$, we reset each i^{th} bit to its original value (line 22) and $x' = x = 0100$ is returned. Next, ManagedGrowth({100*},0100) is called and since 0100 is not redundant to any string in tsh, it is appended to tsh resulting in $tsh = \{100*,0100\}$.

4. The process continues with PatternGenerate({100*,0100},0101). In PatternGenerate, when $x'_3 = 0100$ and since $x'_3 \subseteqq tsh$, then $x'_3 = 010*$ is returned. ManagedGrowth({100*, 0100}, 010*) is called next and since 010* subsumes 0100 in tsh, it is replaced leaving tsh ={100*,010*} (line 38).

5. The process continues similarly, for the remaining input strings in bsh obtaining the final result of $tsh = \{100*, 0010, 010*, *001\}$.

Next, we redefine the binary string operations to account for the $*$ symbol in a ternary string. Note that since the ternary representation extends the binary alphabet (i.e., binary is a subset of the ternary alphabet), ternary operations can also operate over strictly binary strings. For simplicity, we will overload certain operators to denote operations involving both binary and ternary strings.

Definition 7.3.4. (Ternary-or $\bigvee$ and Ternary-and $\bigwedge$). Given two ternary strings, $x, y \in \Sigma_*^l$ of length l, *ternary-or* and *ternary-and* are two bitwise-or functions defined as $\bigvee, \bigwedge : \Sigma_*^l \times \Sigma_*^l \rightarrow \Sigma_*^l$ such that $z = x \bigvee y$ and $w = x \bigwedge y$, $\forall i \; 1 \leq i \leq l$, where:

$$z[i] = \begin{cases} * & \text{if } (x[i] = * \wedge y[i] = *) \\ 0 & \text{if } (x[i] = 0 \wedge y[i] = 0) \\ 1 & otherwise \end{cases} \qquad w[i] = \begin{cases} * & \text{if } (x[i] = * \wedge y[i] = *) \\ 1 & \text{if } (x[i] = 1 \wedge y[i] = 1) \\ & \quad \vee\ (x[i] = 1 \wedge y[i] = *) \\ & \quad \vee\ (x[i] = * \wedge y[i] = 1) \\ 0 & otherwise \end{cases}$$

■

Definition 7.3.5. (Ternary set intersection, $\cap$). Given $tsh_1,\ tsh_2 \in tSH^l$, $\cap : tSH^l \times tSH^l \to tSH^l$ is defined as

$$tsh_1\ \cap\ tsh_2 = \{r \mid r = s1\ \bigwedge\ s2, s1 \mathcal{M} s2, s1 \in tsh1, s2 \in tsh2\}$$

■

For convenience, we define two binary patterns, 0-mask and 1-mask, in order to simplify further operations. The former takes an l-length binary string s and returns a set with a single string having a 0 where $s[i] = 1$ and $*$'s elsewhere, $\forall i\ 1 \leq i \leq l$. The latter takes also an l-length binary string s, but returns a set of strings with a 1 where $s[i] = 1$ and $*$'s elsewhere, $\forall i\ 1 \leq i \leq l$. For instance, 0-mask(0110) and 1-mask(0110) return $\{*00*\}$ and $\{*1**, **1*\}$, respectively.

Definition 7.3.6. (Ternary relevant sharing $rel(tsh, t)$ and irrelevant sharing $irrel(tsh, t)$). Given $t \in Term$ with length l and $tsh \in tSH^l$ with strings of length l, the set of strings in tsh that are *relevant* with respect to t is obtained by a function $rel(tsh, t) : tSH^l \times Term \to tSH^l$ defined as

$$rel(tsh, t) = tsh \cap \text{1-mask}(\hat{t})$$

In addition, $irrel(tsh, t)$ is defined as

$$irrel(tsh, t) = (tsh \cap \mathsf{1\text{-}mask}(\bar{\hat{t}})) \cap \mathsf{0\text{-}mask}(\hat{t})$$

■

Ternary cross-union, $\bowtie$, and ternary up-closure, $(.)^*$, operations are as defined in Def. 7.2.3 and in Def. 7.2.4, respectively, except the binary version of the bitwise OR operator is replaced with its ternary counterpart defined in Def. 7.3.4 in order to account for the $*$ symbol. In addition, the ternary abstract unification (*amgu*) is defined exactly as the binary version, Def.7.2.5, using the corresponding ternary definitions.

Example 7.3.2. (Ternary abstract unification). Let $tsh = \{100{*}, 010{*}, 0010, {*}001\}$ as in Example 7.3.1. Consider again the analysis of $X_1 = f(X_2, X_3)$, the result is:

$$\begin{array}{lcl}
A = rel(tsh, X_1) & = & \{100{*}\} \\
B = rel(tsh, f(X_2, X_3)) & = & \{010{*}, 0010\} \\
A \bowtie B & = & \{110{*}, 101{*}\} \\
(A \bowtie B)^* & = & \{110{*}, 101{*}, 111{*}\} \\
C = irrel(tsh, X_1 = f(X_2, X_3)) & = & \{0001\} \\
amgu(X_1, f(X_2, X_3), tsh) = C \ \cup \ (A \bowtie B)^* & = & \{0001, 110{*}, 101{*}, 111{*}\}
\end{array}$$

Definition 7.3.7. (Ternary initial state, *init*). The *initial state init* : $\mathcal{V} \times \mathcal{I}^+ \rightarrow tSH^{|\mathcal{V}|}$ describes an initial substitution given a set of variables of interest. Assuming the binary initial state operation defined as $init_{bSH} : \mathcal{V} \rightarrow bSH^{|\mathcal{V}|}$, the ternary initial state can be defined using the Convert algorithm in Fig. 7.2 as:

$$init(\mathcal{V}, k) = \mathsf{Convert}(init_{bSH}(\mathcal{V}), k)$$

■

Definition 7.3.8. (Ternary equivalence, $\equiv$). Given tsh_1, $tsh_2 \in tSH^l$, the sets are *equivalent* if and only $(\forall t_1 \in tsh_1, \forall s_1 \lesseqqgtr t_1, \ s_1 \lesseqqgtr tsh_2) \ \wedge \ (\forall t_2 \in tsh_2, \forall s_2 \lesseqqgtr t_2,$

$s_2 \lesseqqgtr tsh_1$).

∎

The ternary join is defined as its binary counterpart, i.e., union. Finally, the ternary projection, $tsh|_t$, is defined similarly as binary projection, see Def. 7.2.9. However, the projection domain and range is extended to accommodate the $*$ symbol. So, the function definition remains the same except that *ternary* string projection is now defined as a function $\pi(s,t)$: $\Sigma_*^l \times Term \rightarrow \Sigma_*^k$, $k \leq l$. For example, let $tsh = \{$100*, 010*, 0010, *001$\}$ as in Example 7.3.1. Then, the projection of tsh over the term $t = f(X_1, X_2, X_3)$ is $tsh|_t = \{$100, 010, 001$\}$. Note that since all zeros is meaningless in a set-sharing representation, it is not included here.

7.4 Negative Ternary Set-Sharing

In this section, we describe a further step using the ternary representation discussed in the previous section. In certain cases, a more compact representation of sharing relationships among variables can be captured equivalently by working with the complement (or negative) set of the original sharing set. A ternary string t can either be *in* or *not in* the set $tsh \in tSH$. This mutual exclusivity together with the finiteness of $\mathcal{V}$ allows for checking t's membership in tsh by asking if t is in tsh, or, equivalently, if t is *not* in its complement, $\overline{tsh}$. Given a set of l-bit binary strings, its complement or negative set contains *all* the l-bit ternary strings *not* in the original set. Therefore, if the cardinality of a set is greater than half of the maximum size (i.e., $2^{|\mathcal{V}|-1}$), then the size of its complement will not be greater than $2^{|\mathcal{V}|-1}$. It is this size differential that we exploit. In Set-Sharing analysis, as we consider programs with larger numbers of variables of interest, the potential number of sharing groups grows exponentially, toward $2^{|\mathcal{V}|}$, whereas the number of sharing groups not in the

```
0 NegConvert($sh, k$)
1 $tnsh \leftarrow \mathcal{U}$
2 foreach $t \in sh$
3     $tnsh \leftarrow$ Delete($tnsh, t, k$)
4 return $tnsh$
```

```
0 NegConvertMissing($bsh, k$)
1 $tnsh \leftarrow \emptyset$
2 $bnsh \leftarrow \mathcal{U} \setminus bsh$
3 foreach $t \in bnsh$
4     $tnsh \leftarrow$ Insert($tnsh, t, k$)
5 return $tnsh$
```

```
10 Delete($tnsh, x, k$)
11 $D_x \leftarrow \{t \mid t \in tnsh, x \mathcal{M} t\}$
12 $tnsh' \leftarrow tnsh$ with $D_x$ removed
13 foreach $y \in D_x$
14     foreach unspecified bit position $q_i$ of $y$
15         if $b_i$ (the $i^{th}$ bit of $x$) is specified, then
16             $y' \leftarrow y \cdot (q_i = \overline{b_i})$
17             $tnsh' \leftarrow$ Insert($tnsh', y', k$)
18 return $tnsh'$
```

```
20 Insert($tnsh, x, k$)
21 $m \leftarrow$ Specified($x$)
22 if $m < k$ then
23     $P \leftarrow$ select $(k - m)$ unspecified bit positions in $x$
24     foreach possible bit assignment $V_P$ of the selected positions
25         $y \leftarrow x \cdot V_P$
26         $tnsh' \leftarrow$ ManagedGrowth($tnsh, y$)
27 else
28     $y \leftarrow$ PatternGenerate($tnsh, x, k$)
29     $tnsh' \leftarrow$ ManagedGrowth($tnsh, y$)
30 return $tnsh'$
```

Figure 7.3: NegConvert, NegConvertMissing, Delete and Insert algorithms used to transform positive to negative representation; k is the desired number of specified bits (non-*'s) to remain.

sharing set decreases toward 0.

The idea of a negative set representation and its associated algorithms extends the work by Esponda et al. in [41, 42]. In that work, a negative set is generated from the original set in a similar manner to the conversion algorithms shown in Figs. 7.2 and 7.3. However, they produce a negative set with unspecified bits in random positions and with less emphasis on managing the growth of the resulting set. The technique was originally introduced as a means of generating Boolean satisfiability (SAT) formulas where, by leveraging the difficulty of finding solutions to hard SAT instances, the contents of the original set are obscured without using encryption [41]. In addition, these hard-to-reverse negative sets are still able to answer membership queries efficiently while remaining intractable to reverse (i.e., to obtain the contents of the original set). In this paper, we are not interested in this security property, however, and use the negative approach simply to address the efficiency issues faced by the traditional Set-Sharing domain.

The conversion to the negative set can be accomplished using the two algorithms shown in Figure 7.3. NegConvert uses the Delete operation to remove input strings of the set sh from $\mathcal{U}$, the set of all l-bit strings $\mathcal{U} = \{*^l\}$, and then, the Insert operation to return $\mathcal{U} \setminus sh$ which represents all strings *not* in the original input. Alternatively, NegConvertMissing uses the Insert operation directly to append each string *missing* from the input set to an empty set resulting in a representation of all strings *not* in the original input. Although as shown in Table 7.1 both algorithms have similar complexities, depending on the size of the original input it may be more efficient to find all the strings missing from the input and transform them with NegConvertMissing, rather than applying NegConvert to the input directly. Note that the resulting negative set will use the same ternary alphabet described in Def. 7.3.1. For clarity, we will define it as:

Definition 7.4.1. (Ternary Negative Sharing Domain, $tNSH$). The ternary negative sharing domain is defined as its positive counterpart in Def. 7.3.1, i.e. $tNSH \equiv tSH$.

■

We describe only NegConvert since NegConvertMissing uses the same machinery.

Assume a transformation from bsh to $tnsh$ calling NegConvert with $k = 1$. We begin with $tnsh = \mathcal{U} = \{* * **\}$ (line 1), then incrementally Delete each element of bsh from $tnsh$ (line 2-3). Delete removes all strings matched by x from $tnsh$ (line 11-12). If the set of matched strings, D_x, contains unspecified bit values (* symbols), then all string combinations *not* matching x must be re-inserted back into $tnsh$ (line 13-17). Each string y' not matching x is found by setting the unspecified bit to the opposite bit value found in $x[i]$ (line 16). Then, Insert ensures string y' has at least k specified bits (line 22-26). This is done by specifying $k - m$ unspecified bits (line 23) and appending each to the result using ManagedGrowth (line 24-26). If string x already has at least k specified bits, then the algorithm attempts to introduce more $*$ symbols using PatternGenerate (line 28) and appends it while removing any redundancy in the resulting set using ManagedGrowth (line 29).

Example 7.4.1. (Conversion from bSH to tNSH). Consider the same sharing set as in Example 7.3.1: bsh = {1000, 1001, 0100, 0010, 0101, 0001}. A negative ternary string representation is generated by applying the NegConvert algorithm to obtain {0000, 11**, 1*1*, *11*, **11}. Since a string of all 0's is meaningless in a set-sharing representation, it is removed from the set. Thus, $tnsh$ = {11**, 1*1*, *11*, **11}.

1. The first string 1000 is deleted from $\mathcal{U} = \{* * **\}$. So, $D_x = \{* * **\}$ (line 11) and $tnsh' = \emptyset$ (line 12). For each i^{th} bit of x, a new $y'_i \not\mathcal{M} x$ is evaluated for insertion into the result set. So, Insert ($\emptyset$, y'_0 = 0***, $k = 1$) is called (line 17). Since Specified$(y') \geq k$ and $tnsh' = \emptyset$, the result returned is $tnsh'$ ={0***} (line 27-30). For all other unspecified positions (line 14) of y, a new string is created with a bit value opposite to x_i's value, $(\overline{b_i})$. So, Insert ({0***}, y'_1 = *1**, $k = 1$) is called next and y'_1 is appended to $tnsh'$. The process continues with y'_2 and y'_3 resulting in $tnsh$ = {0***, *1**, **1*, ***1}.

2. Next, 1001 from bsh is deleted (line 2) resulting in D_x ={***1} and $tnsh' =$ {0***, *1**, **1*} (line 11,12). Then, Insert ({0***, *1**, **1*}, $y' =$ 0**1, $k = 1$) is called. Since 0**1 $\leqq\!\!\!\!\!\!\!<$ $tnsh'$, then $tnsh'$ remains unchanged. The process continues with y'_1 =*1*1, y'_2 =**11 being subsumed by $tnsh'$; so the result returned is $tnsh =$ {0***, *1**, **1*}.

3. Next, 0100 is deleted resulting in $tnsh =$ {00**, 0**1, 11**, *1*1, **1*}.

4. Next, 0010 is deleted resulting in $tnsh =$ {000*, 0**1, 11**, 1*1*, *11*, *1*1, **11}.

5. Next, 0101 is deleted resulting in $tnsh =$ {000*, 00*1, 11**, 1*1*, *11*, **11}.

6. Finally, 0001 is deleted resulting in $tnsh =$ {0000, 11**, 1*1*, *11*, **11}.

7. Removing the string with all 0s, we get the final $tnsh =$ {11**, 1*1*, *11*, **11}.[1]

An alternative conversion algorithm uses NegConvertMissing. But, first the missing strings must be calculated from the given set. For Example 7.4.1, the missing strings are {0011, 0110, 0111, 1010, 1011, 1100, 1101, 1110, 1111}.

1. The NegConvertMissing begins with the first string 0011 and $tnsh = \emptyset$ resulting in $tnsh$ ={0011}.

2. Then, Insert ({0011}, $y' =$ 0110, $k = 1$) resulting in $tnsh$ ={0011, 0110}.

3. Next, Insert ({0011, 0110}, $y' =$ 0111, $k = 1$) resulting in $tnsh$ ={011*, 0*11}.

4. Next, Insert ({011*, 0*11}, $y' =$ 1010, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 1010}.

[1]Notice that $tnsh = \mathcal{U} \setminus (bsh \cup \{0000\})$.

Transformation	Time Complexity	Size Complexity
$bSH \rightarrow tSH$	$O(\|bsh\|\alpha l)$	$O(\|bsh\|)$
$bSH/tSH \rightarrow tNSH$	$O(\|bsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tnsh\|(l-m)2^{\delta})$
$tNSH \rightarrow tSH$	$O(\|tnsh\|\alpha(\alpha 2^{\delta}+1))$	$O(\|tsh\|(l-m)2^{\delta})$
$bSH \rightarrow tNSH$	$O(\beta + \|bnsh\|(\alpha 2^{\delta}+1))$	$O(\|bnsh\|2^{\delta})$

Table 7.1: Summary of conversions: l-length strings; $\alpha = |Result| \cdot l$; if $m < k$ then $\delta = k - m$ else $\delta = 0$, where m = minimum specified bits in entire set, k = number of specified bits desired; $bnsh = \mathcal{U} \setminus bsh$; $\beta = O(2^l)$ time to find $bnsh$.

5. Next, Insert ({011*, 0*11, 1010}, $y' = 1011$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, *011}.

6. Next, Insert ({011*, 0*11, 101*, *011}, $y' = 1100$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, 1100, *011}.

7. Next, Insert ({011*, 0*11, 101*, 1100, *011}, $y' = 1101$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, 110*, *011}.

8. Next, Insert ({011*, 0*11, 101*, 110*, *011}, $y' = 1110$, $k = 1$) resulting in $tnsh$ ={011*, 0*11, 101*, 110*, *011, *110}.

9. Finally, Insert ({011*, 0*11, 101*, 110*, *011, *110}, $y' = 1111$, $k = 1$) resulting in $tnsh$ ={11**, 1*1*, *11*, **11}.

Notice that NegConvertMissing would return the same result for Example 7.4.1, and, in general, an equivalent negative representation.

Table 7.1 illustrates the different transformation functions and their complexities for a given input. Transformation $bSH \rightarrow tSH$ can be performed by the Convert algorithm described in Fig. 7.2. Transformations $bSH/tSH \rightarrow tNSH$ and $bSH \rightarrow tNSH$ are done by NegConvert and NegConvertMissing, respectively. Both transformations show that we can convert a positive representation into negative with corresponding difference in time and memory complexity. Depending on the size of the original input we may prefer one transformation over another. If the input size is relatively small, less than 50% of the maximum size, then NegConvert is often more efficient than NegConvertMissing. Otherwise, we may prefer to insert those strings missing in the input set. In our implementation, we continuously track the size of the relationships to choose the most efficient transformation. Finally, transformation $tNSH \rightarrow tSH$ is performed by NegConvert allowing comming back to the positive from a negative representation.

Consider now the same set of variables and order among them as in Example 7.4.1 but with a slightly different set of sharing groups encoded as $bsh = \{1000, 1100, 1110\}$ or $tsh = \{1*00, 1110\}$. Then, a negative ternary string representation produced by NegConvert is $tnsh = \{00**, 01**, 0*1*, 0**1, 1**1, *01*\}$. This example shows that the number of elements, or size, of the negative result, $|tnsh| = 6 > |bsh| = 3$ and $|tsh| = 2$. However, in Example 7.4.1 when $|bsh| = 6$, $|tnsh| = 4 < |bsh|$. This is because when $|bsh|$ is less than $2^{|\mathcal{V}|-1}$, i.e., $|bsh| = 3 < 2^3$, then its complement set must represent $(2^{|\mathcal{V}|} - |bsh|) = 13$ elements. Depending on the strings in the positive set, the size of the negative result may indeed be greater. This is a good illustration of how selecting the appropriate set-sharing representation will affect the size of the converted result. Thus, the size of the original sharing set at specific program points will be used by the analysis to produce the most compact working set. The negative sharing set representation allows us to represent more variables of interest enabling larger problem instances to be evaluated.

We now define certain operations on the negative representation in order to perform abstract unification and the other abstract operations required by our engine to use the negative representation.

Definition 7.4.2. (Negative intersection, $\overline{\cap}$). Given two negative sets with same length strings, ns_1 and ns_2, the *Negative Intersection* returns a negative set representing the set intersection of $ns_1 \,\overline{\sqcap}\, ns_2$, and is defined in [42] as:

$$ns_1 \,\overline{\sqcap}\, ns_2 = \{x|x \in ns_1\} \cup \{y|y \in ns_2\}.$$

■

Definition 7.4.3. (Negative relevant sharing $\overline{rel}(tnsh, t)$ and irrelevant sharing $\overline{irrel}(tnsh, t)$) Given $t \in Term$ and $tnsh \in tNSH^l$ with strings of length l, the set of strings in $tnsh$ that are *negative relevant* with respect to t is obtained by a function $\overline{rel}(tnsh, t) : tNSH^l \times Term \to tNSH^l$ defined as:

$$\overline{rel}(tnsh, t) = tnsh \,\overline{\sqcap}\, \textsf{0-mask}(\hat{t}),$$

In addition, $\overline{irrel}(tnsh, t)$ is defined as:

$$\overline{irrel}(tnsh, t) = tnsh \,\overline{\sqcap}\, \textsf{1-mask}(\hat{t}).$$

■

Because the negative representation is the complement, it is not only more compact for large positive set-sharing instances, but also, and perhaps more importantly, it enables us to use inverse operations that are more memory- and computationally efficient than in the positive representation. However, the negative representation does have its limitations. Certain operations that are straightforward in the positive representation are $\mathcal{NP}$-Hard in the negative representation [41, 42]. A key observation given in [41] is that there is a mapping from Boolean formulae to the negative set-sharing domain such that finding which strings are not represented is equivalent to finding satisfying assignments to the corresponding Boolean formula. This is known to be an $\mathcal{NP}$-Hard problem. As mentioned before, this fact is exploited in [41] for privacy enhancing applications. The mapping is defined as follows.

Let $tnsh$ = {11**, 1*1*, *11*, **11} be the same sharing set as in Example 7.4.1. Its equivalent Boolean formula $\phi \equiv$ not [(x_1 and x_2) or (x_1 and x_3) or (x_2 and x_3) or (x_3 and x_4)] is defined over the set of variables $\{x_1, x_2, x_3, x_4\}$. The formula ϕ is mapped into a negative set-sharing instance where each clause corresponds to a string and each variable in the clause is represented as a 0 if it appears negated, as a 1 if it appears un-negated, and as a * if it does not appear in the clause. By applying DeMorgan's law, we can convert ϕ to an equivalent formula in conjunctive normal form. Then, it is easy to see that a satisfying assignment of the formula such as $\{x_1 = true, x_2 = false, x_3 = false, x_4 = true\}$ corresponding to the string 1001 is not represented in the negative set-sharing instance.

Theorem 7.4.1. *A polynomial time algorithm for computing negative cross-union,* $\overline{\boxtimes}$*, implies* $\mathcal{P}$=$\mathcal{NP}$*.*

Proof. See Appendix A.

Due to the interdependent nature of the relationship between the elements of a negative set, it is unclear how a precise negative cross-union can be accomplished without going through a positive representation. Therefore, we accomplish the negative cross-union by first identifying the represented positive strings and then applying cross-union accordingly.

Rather than iterating through all possible strings in $\mathcal{U}$ and performing cross-union on strings not in $tnsh$, we achieve a more efficient negative cross-union, $\overline{\boxtimes}$, by converting $tnsh$ to tsh first, i.e., using NegConvert from Table 7.1 and performing ternary cross-union on strings $t \in tsh$. In this way, the ternary representation continues to provide a compressed representation of the sharing set. Note that the negative up-closure operation, $\bar{*}$, suffers the same drawback as cross-union. Therefore, we handle it in the same way as the negative cross-union.

Definition 7.4.4. (Negative union, $\overline{\cup}$). Given two negative sets with same length strings, ns_1 and ns_2, the *Negative Union* returns a negative set representing the set union of $ns_1 \overline{\cup}\, ns_2$, and is defined in [42] as:

$$ns_1 \overline{\cup}\, ns_2 = \{z|(x\mathcal{M}y) \Rightarrow z = x \bigwedge y, x \in ns_1, y \in ns_2\},$$

where $\bigwedge$ is the ternary AND operator.

■

Definition 7.4.5. (Negative abstract unification, $\overline{amgu}$). The *negative abstract unification* is a function $\overline{amgu} : \mathcal{V} \times Term \times tNSH^l \to tNSH^l$ defined as

$$\overline{amgu}(x, t, tnsh) = \overline{irrel}(tnsh, x = t) \overline{\cup} (\overline{rel}(tnsh, x) \;\overline{\boxtimes}\, \overline{rel}(tnsh, t))^{\overline{*}},$$

■

Example 7.4.2. (Negative abstract unification). Let $tnsh$ = {11**, 1*1*, *11*, **11} be the same sharing set as in Example 7.4.1. Consider the analysis of $X_1 = f(X_2, X_3)$:

$$\begin{array}{lcl}
A = \overline{rel}(tnsh, X_1) & = & \{11**, 1*1*, *11*, **11, 0***\} \\
B = \overline{rel}(tnsh, f(X_2, X_3)) & = & \{11**, 1*1*, *11*, **11, *00*\} \\
A \,\overline{\boxtimes}\, B & = & \{00**, 01**, 0*0*, *00*\} \\
(A \,\overline{\boxtimes}\, B)^{\overline{*}} & = & \{01**, 0*1*, 100*\} \\
C = \overline{irrel}(tnsh, X_1 = f(X_2, X_3)) & = & \{11**, 1*1*, *11*, **11, 1***, \\
 & & *1**, **1*\} \\
 & = & \{1***, *1**, **1*\} \\
\overline{amgu}(X_1, f(X_2, X_3), tnsh) = C \,\overline{\cup}\, (A \,\overline{\boxtimes}\, B)^{\overline{*}} & = & \{01**, 0*1*, 0**0, 100*\}
\end{array}$$

Definition 7.4.6. (Negative initial state, $\overline{init}$). The *negative initial state* $\overline{init} : \mathcal{V} \times \mathcal{I}^+ \to tNSH^{|\mathcal{V}|}$ describes an initial substitution given a set of variables of interest. Assuming as in Def. 7.3.7 the binary initial state operation $init_{bSH} : \mathcal{V} \to bSH^{|\mathcal{V}|}$, the negative initial state can be defined using either NegConvert or NegConvertMissing described in Fig. 7.3 and denoted both by $\overline{\mathsf{Convert}}$ as follows:

$$\overline{init}(\mathcal{V}, k) = \overline{\mathsf{Convert}}(init_{bSH}(\mathcal{V}), k)$$

∎

Definition 7.4.7. (Negative set equivalence, $\equiv$). Given $tnsh_1, tnsh_2 \in tNSH^l$, they are *equivalent* if and only if $(\forall t_1 \in tnsh_1, \forall s_1 \lesseqqgtr t_1, s_1 \not\lesseqqgtr tnsh_2) \wedge (\forall t_2 \in tnsh_2, \forall s_2 \lesseqqgtr t_2, s_2 \not\lesseqqgtr tnsh_1)$. ∎

Definition 7.4.8. (Negative join, $\overline{\sqcup}$). Given $tnsh_1, tnsh_2 \in tNSH^l$, the *negative join* function $\overline{\sqcup} : tNSH^l \times tNSH^l \to \wp^0(tNSH^l)$ is defined as the negative set union of the two sets, i.e., $tnsh_1 \mathbin{\overline{\cup}} tnsh_2$. ∎

Definition 7.4.9. (Negative project, $\overline{\pi}$). Given a negative set ns and the desired bit positions to project Υ, *Negative Project* is defined in [42] as

$$\overline{\pi}_{\Upsilon}(ns) = \{x | (x\mathcal{M}w) \wedge (\forall w \in \mathcal{U}_{\Upsilon}, \forall z \in \mathcal{U}_{\overline{\Upsilon}},\ \exists y \in ns(y[\Upsilon]\mathcal{M}w \ \wedge \ y[\overline{\Upsilon}]\mathcal{M}z))\},$$

e.g., the resulting negative set will contains strings that has a bit value projected in column(s) specified by Υ if and only if all possible binary combination of all strings created with the projected column(s) appear in the negative set. For example, given $ns = \{000,\ 011,\ 10*,\ 11*\}$, the $\overline{\pi}_{\Upsilon=1,2}(ns) = \{10,\ 11\}$.

∎

Definition 7.4.10. (Negative projection, $\overline{tnsh|_t}$). The *negative projection* is a function $\overline{tnsh|_t}$: $tNSH^l \times Term \to tNSH^k$ $(k \leq l)$ that selects elements of $tnsh$ projected onto the binary representation of $t \in Term$ and is defined as

$$\overline{tnsh|_t} = \overline{\pi}(tnsh, \Upsilon_t),$$

where Υ_t is equal to all i^{th}-bit positions of $\hat{t}$ where $\hat{t}[i] = 1$.

■

Example 7.4.3. (Negative projection). Let $tnsh$ = {11**, 1*1*, *11*, **11} be the same sharing set as in Example 7.4.1. The negative projection of $tnsh$ over the term $t = f(X_1, X_2, X_3)$ is $\overline{tnsh|_t}$ = {11*, 1*1, *11}. String **1 is not in the result because it represents the following strings when fully specified {001, 011, 101, 111} and not all these strings are in the complement, e.g., 001 is in the positive result of the same projection over bsh.

7.5 Experimental Results

We developed a proof-of-concept implementation in order to measure experimentally the relative efficiency in terms of running time and memory usage obtained with the two new representations described earlier, tSH and $tNSH$. The prototype uses *tries* [86] to handle efficiently binary and ternary strings, and is connected to a naive *bottom-up* fixpoint analyzer.

Our first objective is to study the implications of the conversions in the representation for analysis. Note that although both tSH and $tNSH$ do not imply a loss of precision, the sizes of the resulting representations and their conversion times can vary significantly from one to another. An essential issue is to determine experimentally the best overall k parameter for the conversion algorithms. Second, we study the core abstract operation of the traditional set-sharing, *amgu*, under two different metrics. One is the running time to perform the abstract unification. The other metric expresses the memory usage through the size of the representation in terms

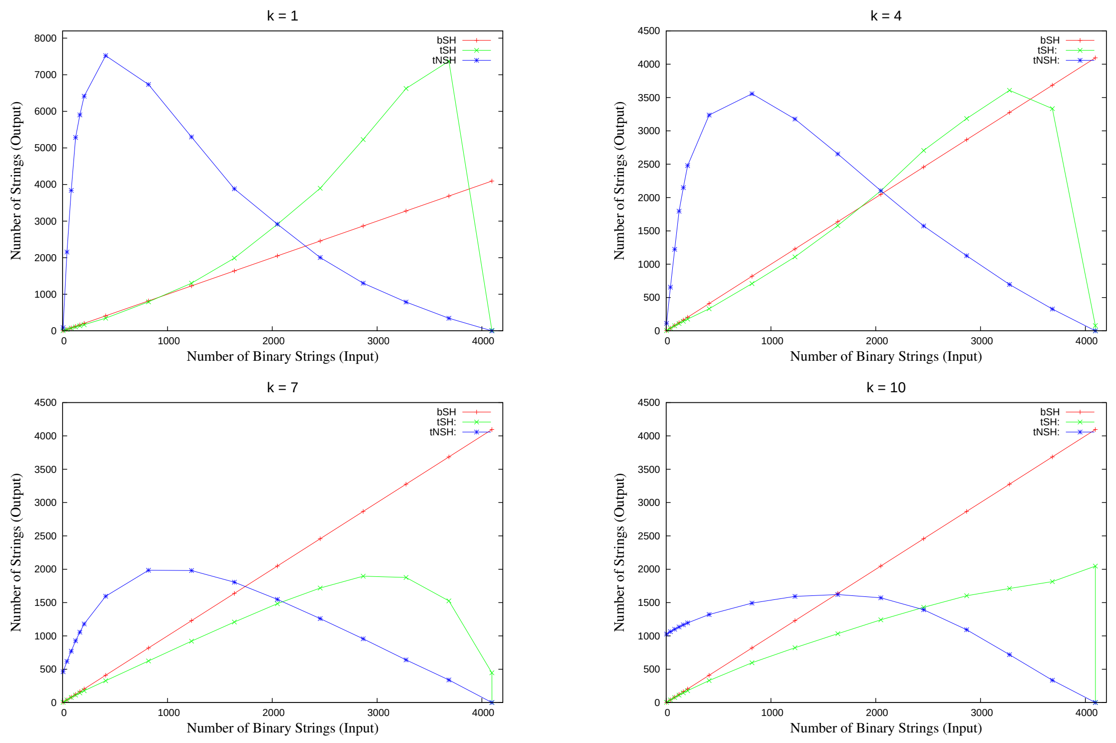
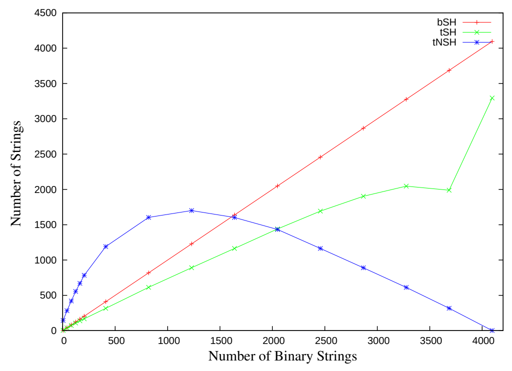


Figure 7.4: Level of compression after conversions from bSH to tSH and $tNSH$ for k = 1, 4, 7, and 10.

of number of strings during key steps in the unification. All experiments have been conducted on an Intel^{R} Core^{TM} Duo CPU T2350 at 1.86GHz with 1GB of RAM running Ubuntu 7.04, and were performed with 12-bit strings since we consider this value large enough to show all the relevant features of our approach. In general, within some upper bound, the more variables considered the better the expected efficiency.

The first experiment determines the best k value suitable for the conversion algorithms, shown in Figs. 7.2 and 7.3. We proceed by submitting a set of 12-bit strings in random order using different k values. We evaluate size for the smallest output (see Fig. 7.4) for a given k value. As expected, bSH ($x = y$ line) results in no compression; tSH slowly increases with increasing input size, remaining below bSH (for $k = 7$ and $k = 10$) due to the compression provided by the $*$ symbol and by having little redundancy; $tNSH$, the complement set, starts larger than bSH but quickly tapers off as the input size increases past 50% of $|\mathcal{U}|$. Since the k parameter helps determine the minimum number of specified bits in the set, there is a direct relationship between the k parameter and the size of the output due to compression by the $*$ symbol. A smaller k value, i.e., $k = 1$, introduces the maximum number of $*$ symbols in the set. However, for a given input, a small k value does not necessarily result in the best compression factor (see $k = 1$ of Fig. 7.4). This result may be counter-intuitive, but it is due to the potentially larger number of unmatched strings that must be re-inserted back into the set determined by all the strings that must be represented by the converted result, see line 13-17 of Fig. 7.3. In addition, a small k value may result in a set with more ternary strings than the number of binary strings represented. This occurs when multiple ternary strings, none of which subsumes any other, represent the same binary string. This redundancy in the ternary representation is not prevented by ManagedGrowth, and is apparent in Fig. 7.4 when $|tSH|$ and $|tNSH|$ exceed the maximum size of binary sharing relationships (i.e., 4096). One way to reduce the number of redundant strings is to sort the binary input by *Hamming distance* before conversion. In the subsequent tests, sorting was performed to maximize compression. We have found empirically that a k setting near (or slightly larger than) $l/2$ is the best overall value considering both the result size and time complexity. We use $k = 7$ in the following experiments. It is interesting to note that a k value of $log_2(l)$ results in polynomial time conversion of the input (see the Complexity column of Table 7.1) but it may not result in the maximum compression of the set (see $k = 4$ of Fig. 7.4). Therefore, k may be adjusted to produce results based on acceptable performance level depending on which parameter is more important to the user, the level of compression (memory constraints) or execution time.

Our second experiment shows the comparison in terms of memory usage (Fig. 7.5, left) and running time (Fig. 7.5, right) of the conversion algorithms for transforming an initial set of binary strings, bSH, into its corresponding set of ternary strings,



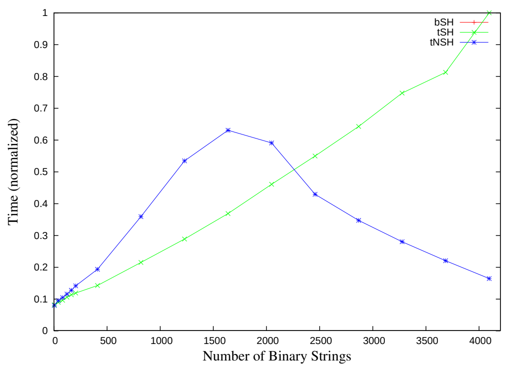


Figure 7.5: Memory usage (avg. # of strings) and time normalized for conversions with $k = 7$.

tSH, or its complement (negative), $tNSH$. We generated random sets of binary strings (over 30 runs) using $k = 7$ and we converted the set of binary strings using the Convert algorithm described in Fig. 7.2 for tSH, and NegConvertMissing in Fig. 7.3 for $tNSH$. We also reduced the number of redundant strings by sorting them using the Hamming distance before conversion. The plot on the left shows that the number of positive ternary strings,$|tSH|$, used for encoding the input binary strings always remains below $|bSH|$, and this number increases slowly with increasing input size. It important to notice that for large values of $|bSH|$, tSH compacts worse than expected and the compression factor is lower. The main cause is the use of the parameter $k = 7$ that implies only the use of 5 or less $*$ symbols for compression. Conversely, the number of negative sharing relationships, $|tNSH|$, is greater than $|bSH|$ and $|tSH|$ up to between 40% and 50%, respectively. However, when the load exceeds those thresholds $tNSH$ compresses much better than its alternatives. For instance, for the maximum number of binary sharing relationships, $tNSH$ compresses them to only one negative string. On the other hand, the rightmost plot shows the average time consumed over 30 runs for both conversion algorithms. Again, $tNSH$ scales better than the positive ternary solution, tSH, after a threshold established around 50% of the maximum number of binary sharing relationships. Our proof-of-concept

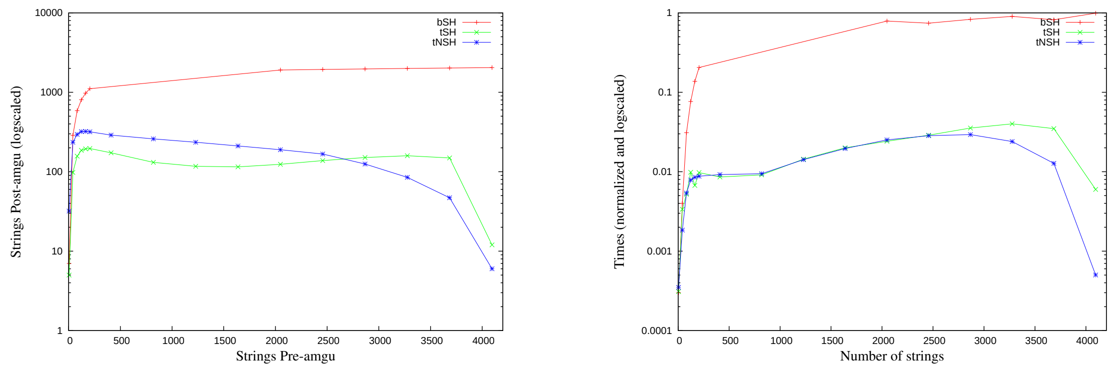


Figure 7.6: Memory usage (avg. # of strings) and time normalized for amgu over 30 runs with $k = 7$.

implementation is not really optimized, since our objective is to study the *relative* performance between the three representations, and thus times are normalized to the range $[0, 1]$. We argue that comparisons that we report between representations are fair since the three cases have been implemented with similar efficiency, and useful since the absolute performance of the base representation is well understood.

Finally, our third experiment shows also the efficiency in terms of the memory usage (in Fig. 7.6, left) and running time (in Fig. 7.6, right) when performing the abstract unification for $k = 7$. Several characteristics of the abstract unification influence the memory usage and its performance. Given an arbitrary set of variables of interest $\mathcal{V}$ ($|\mathcal{V}| = 12$), we constructed $x \in \mathcal{V}$ by selecting one variable and $t \in Term$ as a term consisting of a subset of the remaining variables, i.e., $\mathcal{V} \setminus \{x\}$. We tested with different values of t. Another important aspect is the input sharing set, bSH. Again, we reduced the influence of this factor by generating randomly 30 different sets. In the leftmost plot, the x-axis illustrates the number of input binary strings considered during the *amgu*. In the case of the positive and negative ternary *amgu*, the input binary strings were first converted to their corresponding compressed representations. The y-axis shows the number of strings after the unification. The plot shows that exceeding a threshold lower than 500 in the number of input binary sharing relationships, both tSH and $tNSH$ yield a significant smaller number of strings than the binary solution after unification. Moreover, when the number of the input binary strings is smaller than 50% of its maximum value, tSH compresses more efficiently than $tNSH$. However, if this value is exceeded then this trend is reversed: the negative encoding yields a better compression as the cardinality of the original set grows toward $2^{|\mathcal{V}|}$. The rightmost plot shows the size of the random binary input sets in the x-axis, and the average time consumed for performing the abstract unification in its y-axis, normalized again from 0 to 1. This graph shows that the execution times behave similarly to the memory usage during abstract unification. Both tSH and $tNSH$ run much faster than bSH. The differences are significant (a factor of 10) for most x-values, reaching a factor of 1000 for large values of $|bSH|$. When the load exceeds a $50-60\%$-threshold, $tNSH$ scales better than tSH by a factor of 10. The main difference with respect to the memory usage depicted in the leftmost plot is that for a smaller load, tSH runs as fast as $tNSH$ during unification. The main reason is that the ternary relevant and irrelevant sharing operations are less efficient than their negative counterparts: intersection is an expensive operation in the positive ternary representation whereas the negative intersection is very efficient (positive union).

7.6 Summary

We have presented a novel approach to Set-Sharing that leverages the complement or negative sharing relationships of the original sharing set, without any loss of accuracy. In this work, we based the negative representation on ternary strings. We also showed that the same ternary representation can be used as a positive encoding to efficiently compact the original binary sharing set. This provides the user or the analyzer the option of working with whichever set sharing representation is more efficient for a given problem instance. The capabilities of our negative approach to compress sharing relationships are orthogonal to the use of the ternary representation. Hence, the negative relationships may be encoded by any other representation such as, e.g., Binary Decision Diagrams. Concretely, *Zero-suppressed Binary Decision Diagrams* (ZBDDs) [59] are particularly interesting because ZBDDs were designed to represent sets of combinations (i.e., sets of sets). In addition, this approach may be also applicable to similar sharing-related analyses in object-oriented languages (e.g., [83]).

Our experimental evaluation has shown that our approach may reduce significantly the memory usage of the sharing relationships and the running time of the abstract operations, including the abstract unification. Our experiments also show how to set up key parameters in our algorithms in order to control the desired compression and time complexities. We have shown that we can obtain a reasonable compression in polynomial time by tuning appropriately those parameters. Thus, we believe our results show another approach that can contribute to the practical, scalable application of Set-Sharing.

Chapter 8

A Generic Analysis Framework for Java Bytecode

Chapter 4 presented a practical resource usage analysis for logic programs. However, there are situations, e.g., mobile code, where the source code is not accessible but only compiled code. For example, the receiver of the code may want to infer resource information in order to decide whether to reject code which has too large cost requirements in terms of computing resources (number of bytes sent or received, number of SMSs, energy consumption, heap usage, time, etc.), and to accept code which meets the established requirements. In this context, Java bytecode [75] is widely used, mainly due to its security features and the fact that it is platform-independent.

This chapter presents a generic framework for analysis of Java bytecode programs based on abstract interpretation which can improve the accuracy of the resource usage analysis further shown in Chapter 9. In Section 8.2, we introduce an intermediate representation which generates a Control Flow Graph from the bytecode of each method. In Section 8.3, a generic fixpoint algorithm based on abstract interpretation is described. Section 8.4 shows the feasibility of the framework. Section 8.5

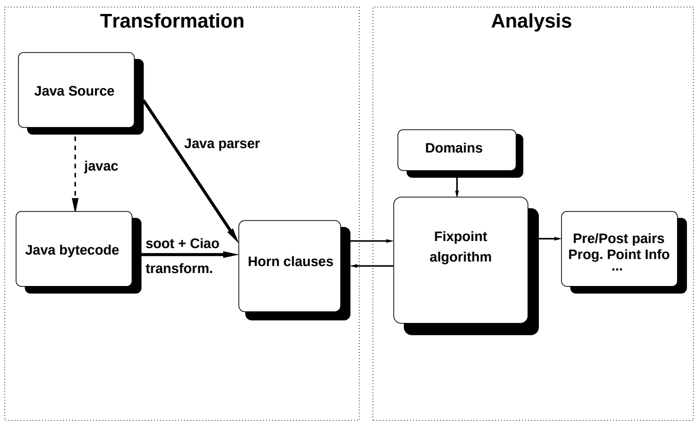


Figure 8.1: Pipeline of transformation and analysis

reviews the state of the art in abstract interpretation-based frameworks, and finally, Section 8.6 summarizes this chapter.

8.1 Motivation and Proposal

Analysis of the Java language (either in its source version or its compiled bytecode [75]) using the framework of abstract interpretation has been the subject of significant research in the last decade (see, e.g., [78] and its references). Most of this research concentrates on finding new abstract domains that better approximate a particular concrete property of the program analyzed in order to optimize compilation (e.g., [17, 105]) or statically verify certain properties about the run-time behavior of the code (e.g., [44, 70]). In contrast to this concentration and progress on the development of new, refined domains there has been comparatively little work on the underlying fixpoint algorithms. In fact, many existing abstract interpretation-based analyses use relatively inefficient fixpoint algorithms. In other cases, the fixpoint algorithms are specific to a particular source language or analysis and cannot easily be reused in other contexts.

The proposed framework (see Figure 8.1) is generic in terms of the abstract domain, and analysis is a two-step process that starts with a program transformation; this phase is language dependent and results in a control flow graph (CFG)-style representation where the operational semantics is made explicit. For example, a virtual call is replaced by a non-deterministic call to all the possible implementations it can be resolved to. This encoding allows transforming different related idioms of a given language (or from several languages) into a highly uniform representation. We argue that this preliminary (de)compilation process greatly simplifies the burden of designing new analyses and abstract operations.

A second, pivotal piece of the framework is an efficient fixpoint algorithm. The efficiency of the algorithm relies on keeping dependencies between different methods during analysis so that only the really affected parts need to be revisited after a change during the convergence process. The algorithm deals thus efficiently with mutually recursive call graphs. In addition, recomputation is avoided using *memoization* which remembers the results corresponding to some set of specific inputs. The proposed algorithm is also *parametric* with respect to the abstract domain, specifying a reduced number of basic operations that it must implement. Another characteristic is that it is *context sensitive* –abstract calls to a given method that represent different input patterns are automatically analyzed separately – and follows a top-down approach, in order to allow modeling properties that depend on the data flow characteristics of the program.

8.2 Intermediate Program Representation

Analysis of a Java bytecode program P normally requires its translation into an intermediate representation that is easier to manipulate. In particular, our decompilation assisted by the Soot [114] tool involves elimination of stack variables, conversion to three-address statements, static single assignment (SSA) transformation, and generation of a Control Flow Graph (CFG) that is ultimately the subject of analysis. In the framework, the decompilation process from Java bytecode to the final CFG involves two steps. The first one is based on the Soot tool that returns a Shimple[1] $CFG(P)$, which has all the described characteristics. In a second phase, the compiler maps that graph into another one, $CFG'(P)$, which represents the same information in a format that is more suitable for analysis.

The following grammar describes the intermediate representation:

CFG	$::=$	$BlockMethod^+$
$BlockMethod$	$::=$	$(\mathsf{id}{:}\mathbb{N}, \mathsf{sig}{:}Sig, \mathsf{fpars}{:}Id^+, \mathsf{annot}{:}expr^*, \mathsf{body}{:}Stmt^*)$
Sig	$::=$	$(\mathsf{class}{:}Type, \mathsf{name}{:}Id, \mathsf{pars}{:}Type^+)$
$Stmt$	$::=$	$(\mathsf{id}{:}\mathbb{N}, \mathsf{sig}{:}Sig, \mathsf{apars}{:}(Id \mid Ct)^+)$
Var	$::=$	$(\mathsf{name}{:}Id, \mathsf{type}{:}Type)$

The Control Flow Graph is formed by *block methods*. A block method is similar to a Java method, except that:

1. If the program flow reaches it, every statement in it will be executed, i.e, it contains no branching;

2. Its signature might not be unique: the CFG might contain several block methods in the same class sharing the same name and formal parameter types;

3. It always includes as formal parameters the returned value *ret* and, unless it is static, the instance self-reference *this*;

4. For every formal parameter (*input* formal parameter) of the original Java method that might be modified, there is an extra formal parameter in the

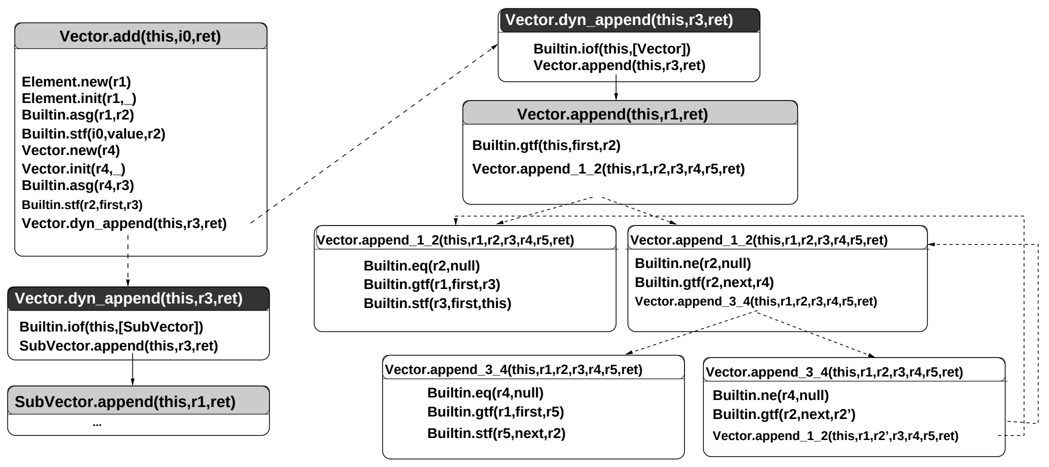
[1]Shimple is an SSA variant of Soot's Jimple internal representation which is a 3-address code, and is the representation of choice for Java analyses.

```
public class Vector {
    Element first;
    public void add(int value){
        Element e = new Element();
        e.value = value;
        Vector v = new Vector();
        v.first = e;
        append(v);
    }
}

class SubVector extends Vector{
    public void append(Vector v){
        //...
    }
}
```

```
public void append(Vector v){
    Element e = first;
    if (e == null)
        first = v.first;
    else{
        while (e.next != null)
            e = e.next;
        e.next = v.first;
    }
}
```

(a) Source code of the Vector class


Vector.add(this,i0,ret)
Element.new(r1)
Element.init(r1,_)
Builtin.asg(r1,r2)
Builtin.stf(i0,value,r2)
Vector.new(r4)
Vector.init(r4,_)
Builtin.asg(r4,r3)
Builtin.stf(r2,first,r3)
Vector.dyn_append(this,r3,ret)
Vector.dyn_append(this,r3,ret)
Builtin.iof(this,[SubVector])
SubVector.append(this,r3,ret)
SubVector.append(this,r1,ret)
...
Vector.dyn_append(this,r3,ret)
Builtin.iof(this,[Vector])
Vector.append(this,r3,ret)
Vector.append(this,r1,ret)
Builtin.gtf(this,first,r2)
Vector.append_1_2(this,r1,r2,r3,r4,r5,ret)
Vector.append_1_2(this,r1,r2,r3,r4,r5,ret)
Builtin.eq(r2,null)
Builtin.gtf(r1,first,r3)
Builtin.stf(r3,first,this)
Vector.append_1_2(this,r1,r2,r3,r4,r5,ret)
Builtin.ne(r2,null)
Builtin.gtf(r2,next,r4)
Vector.append_3_4(this,r1,r2,r3,r4,r5,ret)
Vector.append_3_4(this,r1,r2,r3,r4,r5,ret)
Builtin.eq(r4,null)
Builtin.gtf(r1,first,r5)
Builtin.stf(r5,next,r2)
Vector.append_3_4(this,r1,r2,r3,r4,r5,ret)
Builtin.ne(r4,null)
Builtin.gtf(r2,next,r2')
Vector.append_1_2(this,r1,r2',r3,r4,r5,ret)

(b) Control Flow Graph

block method that contains its final version in the SSA transformation (*output* formal parameter);

5. Every statement in a block method is an invocation, including builtins (assignment asg, field dereference gtf, field access stf, etc.), which are understood as block methods of the class Builtin.

As mentioned before, there is no branching within a block method. Instead, each conditional **if** *cond* $stmt_1$ **else** $stmt_2$ in the original program is replaced with an invocation and two block methods which uniquely match its signature: the first block corresponds to the $stmt_1$ branch, and the second one to $stmt_2$. To respect the semantics of the language, we decorate the first block method with the result of compiling *cond*, while we attach $\overline{cond}$ to its sibling. A similar approach is used in virtual invocations, for which we introduce as many block methods in the graph as possible receivers of the call were in the original program.

Example 8.2.1. (CFG transformation). Figure 8.2(a) shows an alternative version of the JDK Vector class, and Figure 8.2(b) depicts its corresponding Control Flow Graph. An *entry* method corresponds in the original program to the first clause [46] of the Java method of the same name and shares its signature, except for two extra parameter that represents the the instance self-reference, *this*, and the value returned, *ret*. The other clauses present in the Java method are compiled into (components of) *internal* methods which share the same set of variables: all the formal parameters and local variables they reference. Examples of constructions converted into internal clauses are **if**, **while** or **for** loops. In the example, we can see how the `if (e==null)...else` conditional in the Vector implementation of append is converted into two different clauses, one for each branch, which actually share the same name Vector.append_1_2. In this case, the internal method is composed of two clauses which are indistinguishable from the caller's point of view, thus causing invocations to the method to be non-deterministic (i.e., causing the execution of one clause or another). Entry clauses are marked in grey, internal ones in white; dotted arrows denote non-deterministic flows while the continuous ones symbolize deterministic calls.

Another flow transformation, *extra* clauses, tries to expose the internal structure of some complex Java features, which sometimes encode sophisticated operations. That is the case of the virtual invocations. Note that the call to append within add is polymorphic: it might execute the implementation in Vector or the one in SubVector. We make this semantics explicit by inspecting the application hierarchy and replacing the virtual invocation with a set of resolved calls, one for each possible implementation. The method acting as a "hub" is called an *extra* clause; in the example we have two, Vector.dyn_append, marked in black. They behave in a very similar way to the conditional discussed previously, since the program flow might go through two alternative paths (clauses), one for each implementation of append. Each branch contains a guard, iof, see the first statement in each of the Vector.dyn_append clauses, listing the acceptable types for the callee.

It is interesting how, in an analogous way to the clause case, we introduced *extra* statements to further simplify analysis. For example, the mentioned iof builtin filters the execution of subsequent statements when the class of the instance is not listed in the set of possibilities; guard statements have a similar goal in clauses that come from conditional constructions. In Figure 8.2(b) the eq call at the beginning of the leftmost Vector.append_1_2 clause refers to the condition for executing the first branch, while the ne call contains its negated version, for the second alternative. Also, those methods that are *entry* but not *extra* contain assignments to shadow variables that simulate the call-by-reference semantics. They are omitted in Figure 8.2(b) for clarity.

8.3 The Top-Down Analysis Algorithm

We now describe our top-down analysis algorithm, which calculates the least fixed point given a control flow graph and an initial abstract state. Intermediate results are stored in a memo table, which contains the results of computations already performed and is typically used to avoid needless recomputation. In our context it is used to store results obtained from an earlier round of iteration and also to track whether a certain entry represents final, stable results for the block, or intermediate approximations obtained half way during the convergence of fixpoint computations,

```
topDownAnalyze($CFG, method, dom, in, mt, set$)
    $mflag$:=classify($CFG, method$)
    case $mflag$ of
      $not\_recursive$:
        return analyzeNonRecMethod($CFG, method, dom, in, mt, set$)
      $recursive$:
        return analyzeRecMethod($CFG, method, dom, in, mt, set$)
      $builtin$:
        return $dom$.analyzeBuiltin($method, in, mt$)
      $external$:
        return $dom$.analyzeExternal($method, in, mt$)
```

Figure 8.2: The top-down fixpoint algorithm

and also it keeps track of the implicit abstract *and-or* graph. An entry in the memo table has the following fields: block name, its projected call state (λ), its status, its projected exit state (λ') and a unique identifier. Along with the memo table we assume operations which allow to query the status of an entry, retrieve the projected exit state, and add or update an entry.

The pseudocode for the main procedure of the fixpoint algorithm is shown in Figure 8.2. Builtins are treated directly by each domain; the same happens for external invocations since we are making, in the current implementation, a *worst-case assumption* in which any reference to an external method returns the top-most element in the domain for all the variables involved in the call.

Invocations of non-recursive methods are handled by analyzeNonRecMethod in Figure 8.3. It first checks if there is an entry in the memo table for the name of the invoked method and its λ. In that case, we reuse the previously computed value for λ'. Otherwise, the variables of its λ are renamed to the set of variables $\{res, r_0, \ldots, r_m\}$ (we will assume a standard naming for the formal parameters of the form $res, r_0, \ldots, r_m$) and an exit state is calculated for each block the method is built of. The results are then merged through the lub operation, renamed back

```
analyzeNonRecMethod($CFG, method, dom, in, mt, set$)
    $name$:=getName($method$)
    $actPars$:=getActualParams($method$)
    $\lambda$:=$dom$.project($in, actPars$)
    if $mt$.isComplete($\langle name, \lambda \rangle$) then
        $\lambda'$:=$mt$.getOutput($\langle name, \lambda \rangle$)
    else
        $\langle \lambda', mt, set \rangle$:= analyzeNonRecBlocks($CFG, name, dom, actPars,$
                                                              $\lambda$, complete, $mt, set$)
    $out$:=$dom$.extend($in, actPars, \lambda'$)
    return $\langle out, mt, set \rangle$
analyzeNonRecBlocks($CFG, name, dom, actPars, \lambda, st, mt, set$)
    $\lambda$:=$\lambda|^{\{res, r_0, \ldots, r_m\}}_{\{actPar_0, \ldots, actPar_m\}}$
    $blocks$:=getNonRecBlocks($name$)
    $\lambda'$:=$\bot$
    foreach $block \in blocks$
        $body$:=getBody($block$)
        $\langle \beta', mt, set \rangle$:=analyzeBody($CFG, \beta, dom, body, mt, set$)
        $\lambda'_b$:=$dom$.project($\beta', \{res, r_0, \ldots, r_m\}$)
        $\lambda'$:=$\lambda' \sqcup \lambda'_b$
    $\lambda'$:=$\lambda'|^{\{actPar_0, \ldots, actPar_m\}}_{\{res, r_0, \ldots, r_m\}}$
    $mt$.insert($\langle name, \lambda, \lambda', st \rangle$)
    return $\langle \lambda', mt, set \rangle$
analyzeBody($CFG, \beta, body, dom, mt, set$)
    $in$:=$\beta$
    foreach $stmt \in body$
        $\langle out, mt, set \rangle$:=topDownAnalyze($CFG, stmt, dom, in, mt, set$)
        $in$:=$out$
    $\beta'$:=$out$
    return $\langle \beta', mt, set \rangle$
```

Figure 8.3: The top-down fixpoint algorithm: non-recursive methods

to the scope of the callee, and inserted as an entry in the memo table characterized as complete. Finally, λ' is reconciled with the calling state through the extend [85] operation, yielding the exit state.

```
analyzeRecMethod($CFG, method, dom, in, mt, set$)
    $name$:=getName($method$)
    $actPars$:=getActualParams($method$)
    $\lambda$:=$dom$.project($in, actPars$)
    if $mt$.isComplete($\langle name, \lambda\rangle$) then
        $\lambda'$:=$mt$.getOutput($\langle name, \lambda\rangle$)
    elseif $mt$.isFixpoint($\langle name, \lambda\rangle$) then
        $\lambda'$:=$mt$.getOutput($\langle name, \lambda\rangle$)
        $set$:=$set \cup$ {getUniqueID($name$)}
    elseif $mt$.isApproximate($\langle name, \lambda\rangle$) then
        $mt$.update($\langle name, \lambda\rangle$, fixpoint)
        $\langle\lambda', mt, set\rangle$:=analyzeRecBlocks($CFG, method, dom, \lambda, mt, set$)
    else
        $\langle\lambda', mt, set\rangle$:=analyzeNonRecBlocks($CFG, name, dom, actPars,$
                                                   $\lambda$, fixpoint, $mt, set$)
        $set$:=$set \cup$ {getUniqueID($name$)}
        $\langle\lambda', mt, set\rangle$:=analyzeRecBlocks($CFG, method, dom, \lambda, \lambda', mt, set$)
    $out$:=$dom$.extend($in, actPars, \lambda'$)
    return $\langle out, mt, set\rangle$
```

Figure 8.4: The top-down fixpoint algorithm: recursive methods

When a method is recursive, the analyzeRecMethod procedure in Figure 8.4 repeats analysis until a fixpoint is reached for the abstract execution tree, i.e., until it remains the same before and after one round of iteration. In order to do this, we keep track of a flag to signal the termination of the fixpoint computation. The procedure starts the analysis in the non-recursive blocks of the invoked method, thus accelerating convergence since the initial λ' is different from $\perp$. An entry in the memo table is inserted with that tentative abstract state and characterized as fixpoint. The remaining, recursive blocks are analyzed within analyzeRecBlocks in Figure 8.5, which repeats their analysis until the value of λ' does not change between two consecutive iterations.

This basic scheme requires two extra features in order to work also for mutually recursive calls. One is the addition of new possible values for the *status* field in memo

```
analyzeRecBlocks($CFG, method, dom, \lambda, \lambda', mt, set$)
    $name$:=getName($method$)
    $actPars$:=getActualParams($method$)
    $\lambda$:=$\lambda|^{\{res,r_0,\ldots,r_m\}}_{\{actPar_0,\ldots,actPar_m\}}$
    $blocks$:=getRecBlocks($name$)
    $set_{method}$:=$\emptyset$
    $fixpoint$:=true
    repeat
        foreach $block \in blocks$
            $body$:=getBody($block$)
            $\langle\beta', mt, set_{body}\rangle$:=analyzeBody($CFG, \beta, dom, body, mt, \emptyset$)
            $dom$.project($\beta', actPars$)
            $\lambda'_{old}$:=$\lambda'$
            $\lambda'$:=$\lambda'_{old} \sqcup \beta'|^{\{actPar_0,\ldots,actPar_m\}}_{\{res,r_0,\ldots,r_m\}}$
            if $\lambda'_{old} \neq \lambda'$ then
                $fixpoint$:=false
                $mt$.update($\langle N, \lambda\rangle, \lambda'$)
            $set_{method}$:=$set_{method} \cup set_{body}$
    until ($fixpoint$ = true)
    $\langle mt, set\rangle$:=updateDeps($method, mt, set_{method}, set$)
    return $\langle\lambda', mt, set\rangle$
```

Figure 8.5: The top-down fixpoint algorithm: recursive methods (continuation)

table entries. If the fixpoint has not been reached yet for a entry (m_1, λ), we saw that it is labeled as fixpoint; if it has been reached, but by using a possibly incomplete value of λ' of some other method m_2 (i.e., a value that does not correspond yet to a fixpoint), we tag that entry as approximate . The second required artifact is a table with dependencies between methods. Note that the fixpoint computation can involve two or more mutually recursive methods, which will indefinetely wait for the other to be complete before reaching that status. This deadlock scenario can be avoided by pausing analysis in method m_2 if it depends of a call to a method m_1 which is already in fixpoint state; we will use the current approximation λ' for m_1 and wait until it reaches complete status and notifies, via updateDeps in Figure 8.6, all the

```
updateDeps($method, mt, set_{method}, set$)
    $id$:=getUniqueID($method$)
    if $set_{method} \setminus \{id\} = \emptyset$ then
        $status$:=complete
        foreach $id'$ such that $id'$ depends on $id$
            remove dependence between $id'$ and $id$
            if $id'$ is independent then
                let $\langle name_{id'}, \lambda'_{id'} \rangle$ be associated with $id'$
                $mt$.update($\langle name_{id'}, \lambda'_{id'} \rangle$, complete)
    else
        $status$:=approximate
        make $id$ dependent from $set_{method} \setminus \{id\}$
    $mt$.update($\langle name, \lambda' \rangle, status$)
    $set$:=$set \cup set_{method} \setminus \{id\}$
    return $\langle mt, set \rangle$
```

Figure 8.6: The top-down fixpoint algorithm: optimization

methods depending on it.

Computation of that fixpoint can be sometimes computationally expensive or even prohibitive, so in order to speed it up we use a combination of techniques. The first is *memoization* [39] since the memo table acts as a cache for already computed tuples. Efficiency of the computation can be further improved by keeping track of the dependencies between methods. In the above scenario, during subsequent iterations for m_1, the subtree for m_2 is explored every time and its entry in the memo table labeled as approximate. After the last round of iteration for m_1, its entry in the memo table will be tagged as complete but the row for m_2 remains as approximate. The subtree for m_2 has to undergo an unnecessary exploration, since it has already used the complete value of the exit state of m_1. In order to avoid this redundant work, after each fixpoint iteration all those methods depending only on another m that just changed its status to complete are automatically tagged with the same status.

Another major feature of our algorithm is its accuracy. Although precision remains in general a domain-related issue, our solution possesses inherent characteristics that help yield more precise results. First, the algorithm offers results of the analysis at each program point due to its top-down condition. Second, and more relevant, the algorithm is fully context sensitive: every new encountered abstract state for the set of formal parameters is independently stored in the memo table. Moreover, different caller contexts will use the same entry as long as the state of their actual parameters is identical.

Although not present in the pseudo-code, our current implementation also supports path-sensitivity [34], which allows independent reasoning about different branches. Since the extend operation is usually computationally expensive and may introduce further imprecision, it is desirable to avoid it whenever possible. For that reason, the analysis can take advantage of some compiler invariants, such as the equal signature shared by all the internal methods contained in the same Java method. Because of having the same number and naming of formal parameters, the extend operation turns out to be unnecessary when the call is invoked from an internal method and targets an internal method.

Example 8.3.1. (Computation of a fixpoint). We show how an example of mutual recursion, Vector.append in Figure 8.2(b), is handled by the fixpoint algorithm defined in Section 8.3. For simplicity, the abstract domain used is nullity, capable of approximating which variables are definitely null and which ones definitely point to a non-null location. The objective is not to fully understand each of the entries of the memo table in Figure 8.7, which would require a complementary explanation of the domain transfer functions and going through a vast amount of intermediate states, but to illustrate how some interesting dependencies and status change in a very specific subset of those states. The method names have been shortened to fit into the tables.

In step 1 it is assumed that the non-recursive blocks for app_{34} and app_{12} have al-

step	method	λ	λ'	state	dependencies
1	app_{12}	λ_1	λ'_{11}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{21}	fix	$\{app_{34}\}$
	app_{12}	λ_3	λ'_{31}	fix	$\{app_{12}\}$
2	app_{12}	λ_1	λ'_{11}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{21}	fix	$\{app_{34}\}$
	app_{12}	λ_3	λ'_{32}	**app**	$\{app_{12}, app_{34}\}$
3	app_{12}	λ_1	λ'_{11}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{22}	**com**	$\emptyset$
	app_{12}	λ_3	λ'_{32}	app	$\{app_{12}\}$
4	app_{12}	λ_1	λ'_{12}	fix	$\{app_{12}\}$
	app_{34}	λ_2	λ'_{22}	com	$\emptyset$
	app_{12}	λ_3	λ'_{32}	**com**	$\emptyset$
5	app	λ_0	λ'_0	**com**	$\emptyset$
	app_{12}	λ_1	λ'_{12}	com	$\emptyset$
	app_{34}	λ_2	λ'_{22}	com	$\emptyset$
	app_{12}	λ_3	λ'_{32}	**com**	$\emptyset$

Figure 8.7: Fixpoint calculation for Vector.append

ready been analyzed. Both entries for these blocks are marked as *fixpoint* since they correspond to recursive methods whose analyses have not converged to a fixpoint yet. Note that there exist two different entries corresponding to method app_{12} which has been analyzed *twice* with different abstract call patterns: one when called from *app* and another when called from app_{34} yielding $\langle app_{12}, \lambda_1, \lambda'_{11}\rangle$ and $\langle app_{12}, \lambda_3, \lambda'_{31}\rangle$, respectively. In step 2, the analysis corresponding to the entry $\langle app_{12}, \lambda_3, \lambda'_{31}\rangle$ has converged to a fixpoint but using the incomplete value of $\langle app_{34}, \lambda_2, \lambda'_{21}\rangle$. Therefore, the entry is forced to *approximate* changing its exit state to λ'_{32}. In step 3, the analysis for the method app_{34} reaches a fixpoint and since it does not depend on other methods, the entry $\langle app_{34}, \lambda_2, \lambda'_{21}\rangle$ is marked as *complete* and updated to $\langle app_{34}, \lambda_2, \lambda'_{22}\rangle$. After this step, the algorithm notices that $\langle app_{12}, \lambda_3, \lambda'_{32}\rangle$ is *approximate* and waiting for a complete value of $\langle app_{34}, \lambda_2, \lambda'_{22}\rangle$ which has been already produced. Thus, the entry $\langle app_{12}, \lambda_3, \lambda'_{32}\rangle$ is marked directly as *complete* and no extra iteration is required. This change is illustrated in step 4. Finally, the analysis characterizes also the entry $\langle app_{12}, \lambda_1, \lambda'_{12}\rangle$ as *complete* and terminates the semantics computation of *app*.

8.4 Experimental Results

We have completed a preliminary implementation of the framework, and performed two experiments with the framework using the benchmarks corresponding to the JOlden suite [63]. The first experiment is summarized in Table 8.1 and shows the scalability of the transformation phase. The first three columns contain basic metrics about the application: number of classes (k), methods (m) and instructions (i). Since the latter corresponds to the bytecode representation of the source, we also list how many program points (pp) are present in the Horn clause program analyzed. This metric slightly differs from the number of instructions in the sense that extra clauses and builtins make it somewhat larger; pp also provides a better approximation of the size and complexity of the program analyzed because the semantics of the object-oriented program is made explicit. The fifth column (ct) shows the time invested (given in seconds) in transforming the input program and producing the Horn clause version.

The second experiment shown in Table 8.2 illustrates the scalability, efficiency, and precision of the analysis component of our framework. We first use a simple abstract domain, Nullity, capable of approximating which variables are definitely null and which ones definitely point to a non-null location. The second abstract domain is a Class Hierarchy Analysis [10], which uses the combination of the statically declared type of an object and the class hierarchy of the program to determine the set of possible targets of a virtual invocation. The use of a Class Hierarchy Analysis shows

Program	k	m	i	pp	ct
Health	8	30	637	933	1.1
BH	9	70	1208	1739	3.2
Voronoi	6	73	988	1340	2.2
MST	6	36	445	665	0.1
Power	6	32	1017	1270	2.1
TreeAdd	2	12	193	274	2.0
Em3d	4	22	447	669	0.1
Perimeter	10	45	543	814	0.1
BiSort	2	15	323	476	0.1
All	50	317	5839	7251	11.0

Table 8.1: Statistics of the transformation phase.

the scalability of our framework for a domain with non-linear worst-case complexity in its operations. The columns labeled pp' show the number of program points reachable by the analyses. Therefore, pp' may differ from pp because the number of analyzed program points is not always the total number of program points in the program: some commands are found to be unreachable. Since our framework is multivariant and can thus keep track of different *contexts* at each program point, at the end of analysis there may be more than one abstract state associated with each program point. Thus, the number of abstract states is typically larger than the number of reachable program points. Columns ast provide the total number of these abstract states inferred by analyses. The level of multivariance is the ratio ast/pp', presented in columns st. In general, such a larger number for st tends to indicate more precise results. Running times are listed in columns pt (time invested in preprocessing the program and the construction of the class hierarchy) and at (analysis time); both are also given in seconds.

The benchmarks have been tested in both experiments on a Pentium M 1.73Ghz with 1Gb of RAM , and averaging several runs after eliminating the best and worst values. We chose to show separately the total times of the two phases (transformation

		Nullity				**CHA**			
	pt	*pp′*	*ast*	*st*	*at*	*pp′*	*ast*	*st*	*at*
Health	2.1	921	5836	6.3	9.6	933	3542	3.8	52.1
BH	2.2	1739	12384	7.1	50.1	1739	4757	2.7	59.4
Voronoi	2.2	1277	5492	4.3	11.5	1340	5147	3.8	81.3
MST	2.1	496	1503	3.0	1.1	665	1609	2.4	11.6
Power	2.1	1270	10560	8.3	29.9	1270	2908	2.3	32.7
TreeAdd	2.0	274	880	3.2	0.6	274	729	2.6	6.1
Em3d	2.0	669	5565	8.3	0.9	669	3320	4.9	49.5
Perimeter	2.1	814	2653	3.2	1.7	814	3731	4.5	25.0
BiSort	2.1	476	3353	7.0	5.8	476	1614	3.4	15.6
All	2.6	7188	48476	6.7	145.9	7251	29586	4.1	391.2

Table 8.2: Statistics for the Nullity and Class Hierarchy (CHA) domains.

and analysis) because we expect the transformation process to be fully run only once. Later executions can use incremental compilation for those files that changed, so that the overhead of the preprocessing phase should be almost negligible in medium and large programs. Although the same approach can be taken for the analysis [99], the current implementation is not incremental.

8.5 Related Work

Most published analyses based on abstract interpretation for Java or Java bytecode do not provide much detail regarding the implementation of the fixpoint algorithm. Also, most of the published research (e.g., [17, 26]) focuses on particular properties and therefore their solutions (abstract domains) are tied to them, even when they are explicitly multipurpose, like TVLA [72]. In [97] the authors mention a choice of several context insensitive and sensitive computations, but no further information is given. The more recent and quite interesting Julia framework [110] is intended to be generic and targets bytecode as in our case. Their fixpoint techniques are based on prioritizing analysis of non-recursive components over those requiring fixpoint computations and using abstract compilation [56]. However, few implementation details are provided. Also, this is a *bottom-up* framework, while our objective is to develop a top-down, context sensitive framework. While it is well-known that bottom-up analyses can be adapted to perform top-down analyses by subjecting the program to a "magic-sets"-style transformation [102], the resulting analyzers typically lack some of the characteristics that are the objective of our proposal, and, specially, context sensitive results. Finally, Cibai [77] is another generic static analyzer for the modular analysis and verification of Java classes. The algorithm presented is *top-down*, and only a naive version of it (which is not efficient for mutually recursive call graphs) is presented.

8.6 Summary

This chapter has presented a novel abstract interpretation framework, which is generic in terms of abstract domain in use. The framework makes use of a decompilation phase that results in a control flow graph (CFG) where the operational semantics is made explicit, and an analysis phase based on an efficient, precise fixpoint algorithm which has been concisely described in this chapter. This algorithm benefits from acceleration techniques like memoization or dependency tracking, considerably reducing the number of iterations. We also claim that the analysis has the potential to be very accurate because of the top-down, context sensitive approach adopted. Our experimental evaluation shows the feasibility of the approach with medium-size programs using the benchmarks corresponding to the JOlden suite.

Chapter 9

Resource Usage Analysis for Java Bytecode

This chapter presents a resource usage analyzer for Java bytecode. The starting point of this analysis is the analysis described in Chapter 4. Herein, we develop based on it an analysis suitable for Java bytecode. The resulting tool takes a Java bytecode program, a set of resources of interest given by the user, and computes an upper bound of its resource consumption as a (closed form) expression depending on the input data sizes. Its main components as depicted in Figure 9.1 are as follows:

1. The left side of the figure represents the construction, starting from the input bytecode program, of an intermediate representation, as described in Section 8.2, which provides a uniform high-level encoding which allows us to reason compositionally about the cost.

2. The top right side of the figure shows the various pre-analysis steps which are instrumental for the resource usage analysis. We use the fixpoint algorithm defined in Section 8.3 and "plug" into it two domains which result in two

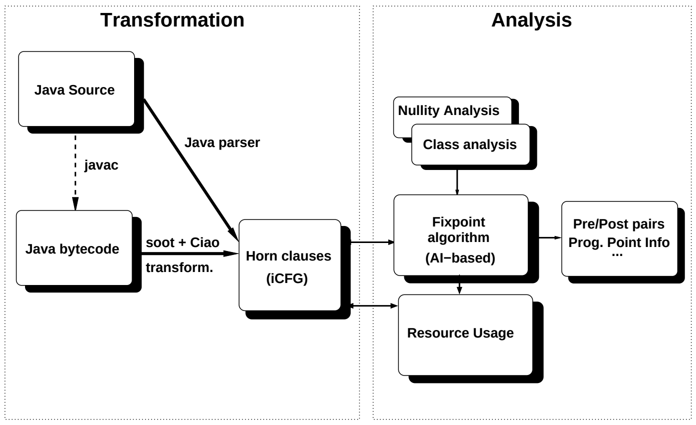


Figure 9.1: Architecture of Resource Usage Analyzer

different analyses: *nullity*, which is aimed at keeping track of null variables, and *class hierarchy analysis (CHA)* [10, 83], which attempts to resolve dynamic dispatching at compile time by transforming dynamic calls into static calls.

3. The bottom right side of the figure shows the resource usage analysis which will be discussed in the rest of this chapter.

Our approach can be used in the context of Java source and Java bytecode in the following fields:

- *Resource Bound Certification [33, 8, 58, 25]:* It proposes the use of safety properties involving cost requirements, i.e., that the untrusted code adheres to specific bounds on resource consumption. Our approach shows, for the first time, that it is possible to automatically generate arbitrary resource bounds certificates for user defined resources in a realistic mobile language. Previous work was restricted to linear bounds [33, 8, 58], to semi-automatic techniques [25], or to source code [54].

- *Performance Debugging and Validation [54]:* This is a direct application of resource analysis, where the analyzer tries to verify or falsify annotations about the efficiency of the program which are written by the programmer. Annotations can possibly refer to the source code level, but it is trivial to translate them to be understandable by the bytecode analyzer.

- *Resource Granularity Control [36]:* Parallel computers have currently become mainstream with multicore processors. In parallel systems, knowledge about the cost of different procedures in the object code can be used in order to guide the partitioning, allocation and scheduling of parallel processes.

In the rest of this chapter, Section 9.1 presents a running example and introduces the basic components of the resource usage analysis. In Section 9.2.1 a practical size analysis and its main sub-components are shown. In Section 9.2.2 the main algorithm for inferring resource usage information is presented, and Section 9.3 shows the feasibility of the approach. Finally, Section 9.5 summarizes our conclusions.

9.1 Overview of the Approach

We start by illustrating the overall approach, whose sub-components are shown in Figure 9.2, through a working example. The Java program in Fig. 9.3 emulates the process of sending of text messages within a cell phone. The source code is provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The phone (class CellPhone) receives a list of packets (SmsPacket), each one containing a single SMS, encodes them (Encoder), and sends them through a stream (Stream). There are two types of encoding: TrimEncoder, which eliminates any leading and trailing white spaces, and UnicodeEncoder, which converts any special character into its Unicode($\backslash uxxxx$) equivalent. The length of the SMS which the cell phone ultimately sends through the stream depends on the size of the encoded

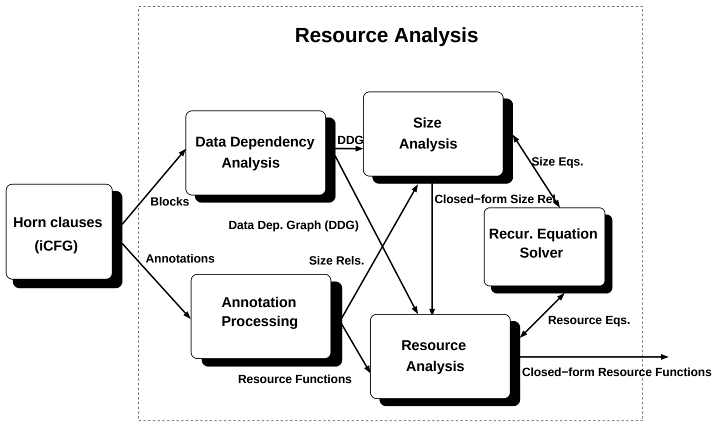


Figure 9.2: Sub-components of the resource usage analysis

message.

A *resource* is a fundamental component in our approach. A resource is a user-defined notion which associates a basic cost function with some user-selected elements (class, method, statement) in the program. This is expressed by adding Java annotations to the code. The objective of the analysis is to approximate the usage that the program makes of the resource. In the example, the resource is the cost in cents of a dollar for sending the list of text messages, since we will assume for simplicity that the carrier charges are proportional (2 cents/character) to the number of characters sent. This domain knowledge is reflected by the user in the method that is ultimately responsible for the communication (Stream.send), by adding the annotation @Cost({"cents","2*size(data)"}). Similarly, the formatting of an SMS done in any implementation of Encoder.format is free, as indicated by the @Cost({"cents","0"})}) annotation. The analysis understands these resource usage expressions and uses them to infer a safe upper bound on the total usage of the program.

Step 1: Constructing the Control Flow Graph. In the first step, the analysis translates the Java bytecode into an intermediate representation building a Control

```
import java.net.URLEncoder;
public class CellPhone {
 SmsPacket sendSms(SmsPacket smsPk,
                   Encoder enc,
                   Stream stm) {
   if (smsPk != null) {
     String newSms = enc.format(smsPk.sms);
     stm.send(newSms);
     smsPk.next=sendSms(smsPk.next,enc,stm);
     smsPk.sms = newSms;
   }
   return smsPk;
 }
}
class SmsPacket{
  String sms;
  SmsPacket next;
}
```

```
interface Encoder{
  String format(String data);
}
class TrimEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=size(s)")
  public String format(String s){
    return s.trim();
  }
}
class UnicodeEncoder implements Encoder{
  @Cost({"cents","0"})
  @Size("size(ret)<=6*size(s)")
  public String format(String s){
    return URLEncoder.encode(s);
  }
}
abstract class Stream{
  @Cost({"cents","2*size(data)"})
  native void send(String data);
}
```

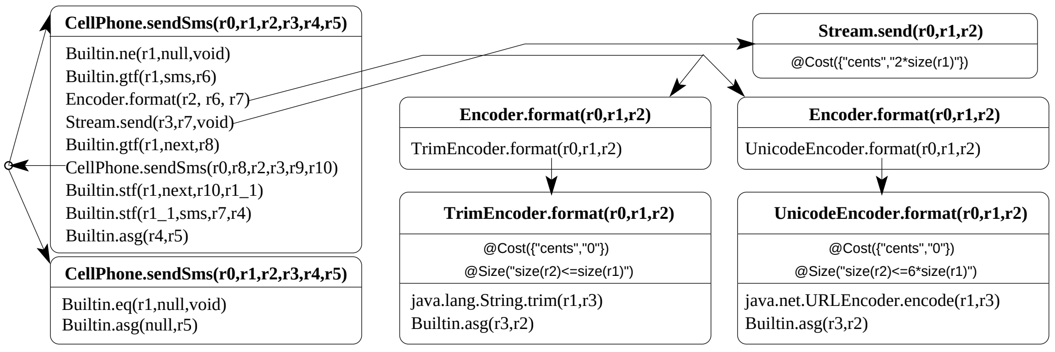

CellPhone.sendSms(r0,r1,r2,r3,r4,r5)
Builtin.ne(r1,null,void)
Builtin.gtf(r1,sms,r6)
Encoder.format(r2, r6, r7)
Stream.send(r3,r7,void)
Builtin.gtf(r1,next,r8)
CellPhone.sendSms(r0,r8,r2,r3,r9,r10)
Builtin.stf(r1,next,r10,r1_1)
Builtin.stf(r1_1,sms,r7,r4)
Builtin.asg(r4,r5)
CellPhone.sendSms(r0,r1,r2,r3,r4,r5)
Builtin.eq(r1,null,void)
Builtin.asg(null,r5)
Stream.send(r0,r1,r2)
@Cost({"cents","2*size(r1)"})
Encoder.format(r0,r1,r2)
TrimEncoder.format(r0,r1,r2)
Encoder.format(r0,r1,r2)
UnicodeEncoder.format(r0,r1,r2)
TrimEncoder.format(r0,r1,r2)
@Cost({"cents","0"})
@Size("size(r2)<=size(r1)")
java.lang.String.trim(r1,r3)
Builtin.asg(r3,r2)
UnicodeEncoder.format(r0,r1,r2)
@Cost({"cents","0"})
@Size("size(r2)<=6*size(r1)")
java.net.URLEncoder.encode(r1,r3)
Builtin.asg(r3,r2)

Figure 9.3: Motivating example: Java source code and Control Flow Graph

Flow Graph (CFG). Edges in the CFG connect *block methods* and describe the possible flows originated from conditional jumps, exception handling, virtual invocations, etc. A (simplified) version of the CFG corresponding to our code example is also shown in Fig. 9.3.

The original sendSms method has been compiled into two block methods that share the same signature: class where declared, name (CellPhone.sendSms), and number and type of the formal parameters. The bottom-most box represents the base case, in which we return null, here represented as an assignment of `null` to the return variable r_5; the sibling corresponds to the recursive case. The virtual invocation of format has been transformed into a static call to a block method named Encoder.format. There are two block methods which are compatible in signature with that invocation, and which serve as proxies for the intermediate representations of the interface implementations in TrimEncoder.format and UnicodeEncoder.format. Note that the resource-related annotations have been carried through the CFG and are thus available to the analysis.

Step 2: Inference of Data Dependencies and Size Relationships. The algorithm infers in this phase *size relationships* between the input and the output formal parameters of every block method. For now, we can assume that size of (the contents of) a variable is the maximum number of pointers we need to traverse, starting at the variable, until `null` is found. The following equations are inferred by the analysis for the two CellPhone.sendSms block methods :

$$\begin{array}{lcl} \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) & \leq & 0 \\ \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) & \leq & 7 \times s_{r_1} - 6 + \mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3}) \end{array}$$

The size of the returned value r_5 is independent of the sizes of the input parameters *this*, *enc*, and *stm* (s_{r_0}, s_{r_2} and s_{r_3} respectively) but not of the size s_{r_1} of the list of text messages *smsPk* (r_1 in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. In the example in Fig. 9.3, the user indicates that the formatting in UnicodeEncoder results in strings that are at most six times longer than the ones received as input @Size("size(ret) $\leq$ 6*size(s)"), while the trimming in TrimEncoder returns strings that are equal or shorter than the input (@Size("size(ret) $\leq$ size(s)")). The equation system shown above must be approximated by a recurrence solver in order to obtain a closed form solution. In this case, our analysis yields the solution $\mathcal{S}ize^{r_5}_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 3.5 \times s_{r_1}^2 - 2.5 \times s_{r_1}$.

Step 3: Resource Usage Analysis. In the this phase, the analysis uses the CFG, the data dependencies, and the size relationships inferred in previous steps in order to infer a resource usage equation for each block method in the CFG and further simplify the resulting obtaining closed form solutions (in general, approximated – upper bounds). Therefore, the objective of the resource analysis is to statically derive safe upper bounds on the amount of resources that each of the block methods in the CFG consumes or provides. The result given by our analysis for the monetary cost of sending the messages (CellPhone.sendSms) is

$$\begin{array}{lll} \mathcal{C}ost_{sendSms}(s_{r_0}, 0, s_{r_2}, s_{r_3}) & \leq & 0 \\ \mathcal{C}ost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) & \leq & 12 \times s_{r_1} - 12 + \mathcal{C}ost_{sendSms}(s_{r_0}, s_{r_1} - 1, s_{r_2}, s_{r_3}) \end{array}$$

i.e., the cost is proportional to the size of the message list (smsPk in the source, r_1 in the CFG). Again, this equation system is solved by a recurrence solver, resulting in the closed formula $\mathcal{C}ost_{sendSms}(s_{r_0}, s_{r_1}, s_{r_2}, s_{r_3}) \leq 6 \times s_{r_1}^2 - 6 \times s_{r_1}$.

9.2 A Framework for Resource Usage Analysis

We now describe our framework for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. The algorithm in Fig 9.4 takes as input a Control Flow Graph in the format described in the previous section, including the user annotations that assign elementary costs to certain graph elements for a particular resource. The user also indicates the set of resources to be tracked by the analysis. Without loss of generality we assume for conciseness in our presentation a single resource.

A preliminary step in our approach is a nullity and class hierarchy analysis, aimed at simplifying the CFG and therefore improving overall precision. Then, another analysis is performed over the CFG to extract data dependencies, as described

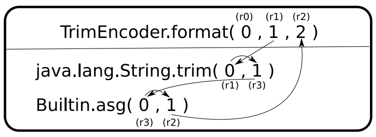
```
resourceAnalysis($CFG, res$)
    $CFG$   $\leftarrow$ classAnalysis($CFG$)
    $mt$    $\leftarrow$ initialize($CFG$)
    $SCCs$ $\leftarrow$ stronglyConnectedComponents($CFG$)
    $dg$    $\leftarrow$ dataDependencyAnalysis($CFG, mt$)
    **foreach** $SCC \in SCCs$ in reverse topological order
        $mt \leftarrow$ sizeAnalysis($SCC, mt, CFG, dg$)
        $mt \leftarrow$ resourceAnalysis($SCC, res, mt, CFG$)
    **return** $mt$
**end**
```

Figure 9.4: Generic Resource Analysis Algorithm

below. The next step is the decomposition of the CFG into its strongly-connected components. After these steps, two different analyses are run separately on each strongly connected component: a) the size analysis, which estimates parameter size relationships for each statement and output formal parameters as a function of the input formal parameter sizes (Sec. 9.2.1); and b) the actual resource analysis, which computes the resource usage of each block method in terms also of the input data sizes (Sec. 9.2.2). Each phase is dependent on the previous one.

The *data dependency analysis* is a dataflow analysis[1] that yields *position dependency graph*s for the block methods within a strongly connected component. Each graph $G = (V, E)$ represents data dependencies between positions corresponding to statements in the same block method, including its formal parameters. Vertexes in V denote positions, and edges $(s_1, s_2) \in E$ denote that s_2 is dependent on s_1. We say that s_1 is a *predecessor* of s_2. We will assume a predec function that takes a position dependency graph, a statement, and a parameter position and returns its nearest predecessor in the graph. The following figure shows the position dependency graph of the TrimEncoder.format block method:

[1]This analysis is similar to the one explained in Chapter 4.


TrimEncoder.format(0 , 1 , 2)
(r0) (r1) (r2)
java.lang.String.trim(0 , 1)
(r1) (r3)
Builtin.asg(0 , 1)
(r3) (r2)

9.2.1 Size Analysis

We now show our algorithm for estimating parameter size relations based on the data dependency analysis. This method is inspired by the ideas of [37, 36] but adapting them to the case of Java bytecode. Our goal is to represent input and output size relationships for each statement as a function in terms of the formal parameter sizes. Unless otherwise stated, whenever we refer to a parameter we mean its position.

The size of an input is defined in terms of measures. By *measure* we mean a function that, given a data structure, returns a number. Our method is parametric on measures, which can be defined by the user and attached via annotations to parameters or classes. For concreteness, we have defined herein two measures, int for integer variables, and the *longest path-length* [1] ref for reference variables. The longest path-length of a variable is the cardinality of the longest chain of pointers than can be followed from it. More complex measures can be defined to handle other datatypes such as cyclic structures, arrays, etc. The set of measures will be denoted by $\mathcal{M}$.

The size analysis algorithm is given in pseudo-code in Figures 9.5, 9.6, and 9.7; its main steps are:

1. Assign an upper bound to the size of every parameter position of all statements, including formal parameters, for all the block methods with the same signature (genBlockSizeRel, Figures 9.6 and 9.7).

```
sizeAnalysis($SCC, mt, CFG, dg$)
  $Eqs \leftarrow \emptyset^{|SCC|}$
  **foreach** $sig \in SCC$
    $Eqs[sig] \leftarrow$ genBlockSizeRel($sig, mt, SCC, CFG, dg$)
  $Sols \leftarrow$ recEqsSolver(simplifyEqs($Eqs$))
  **foreach** $sig \in SCC$
    insert($mt$, size, $sig, Sols[sig]$)
  **return** $mt$
**end**

genBlocksSizeRel($sig, mt, SCC, CFG, dg$)
  $Eqs \leftarrow \emptyset$
  $BMs \leftarrow$ getBlocks($CFG, sig$)
  **foreach** $bm \in BMs$
     $Eqs \leftarrow Eqs \cup$ genBlockSizeRel($bm, mt, SCC, dg$)
  **return** normalize($Eqs$)
**end**
```

Figure 9.5: The size analysis algorithm

2. For a given signature, take the set of size inequations returned by (1) and rename each size relation in terms of the sizes of input formal parameters (normalization, Figure 9.7).

3. Repeat steps (1) and (2) for every signature corresponding to the same strongly-connected component (sizeAnalysis, Figure 9.5).

4. Simplify size relationships by resolving mutually recursive functions, and find closed form solutions for the output formal parameters (sizeAnalysis, Figure 9.5).

Intermediate results are cashed in a memo table mt, which stores measures, sizes, and resource usage expressions for every parameter position. Both size and resource usage expressions are defined in the $\mathcal{L}$ language:

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle expr \rangle \langle bin\_op \rangle \langle expr \rangle \mid \langle quantifier \rangle \langle expr \rangle \\
 & \mid & \langle expr \rangle^{\langle expr \rangle} \mid log_{num} \langle expr \rangle \mid -\langle expr \rangle \\
 & \mid & \langle expr \rangle ! \mid \infty \mid \text{num} \\
 & \mid & \text{size}([\langle measure \rangle,]\text{arg}((\text{r}|\ \text{i}|\ \text{c})\ \text{num})) \\
\langle bin\_op \rangle & ::= & + \mid - \mid \times \mid / \mid \% \\
\langle quantifier \rangle & ::= & \sum \mid \prod \\
\langle measure \rangle & ::= & \textbf{int} \mid \textbf{ref} \mid \ldots
\end{array}
$$

The size of the parameter at position i in statement $stmt$, under measure m, is referred to as $\texttt{size}(m, stmt, i)$. We consider a parameter position to be *input* if it is bound to some data when the statement is invoked. Otherwise, it is considered an *output parameter position*. In the case of input parameter and output formal parameter positions, an upper bound on that size is returned by getSize (Figure 9.6). The upper bound can be a concrete value when there is a constant in the referred position, i.e., when the `val` function returns a non-infinite value:

Definition 9.2.1. (Concrete Size, `val`) The concrete size value for a parameter position under a particular measure is returned by $\texttt{val} : \mathcal{M} \times \mathcal{S}tmt \times \mathbb{N} \rightarrow \mathcal{L}$, which evaluates the *syntactic* content of the actual parameter in that position:

$$
\texttt{val}(m, stmt, i) = \begin{cases} n & \textit{if } stmt.\mathsf{apars}_i \textit{ is an integer } n \\ & \textit{and } m{=}\boldsymbol{int} \\ 0 & \textit{if } stmt.\mathsf{apars}_i \textit{ is } \texttt{null} \textit{ and } m{=}\boldsymbol{ref} \\ \infty & \textit{otherwise} \end{cases}
$$

■

If the content of that input parameter position is a variable, the algorithm searches the data dependency graph for its immediate predecessor. Since the intermediate representation is in SSA form, the only possible scenarios are that either

```
genBlockSizeRel($bm, mt, SCC, dg$)
    $body \leftarrow bm$.body
    $Eqs \leftarrow \emptyset$
    **foreach** $stmt \in body$
        Let $I$ be the input parameter positions in $stmt$
        $Eqs \leftarrow Eqs \cup$ genSizeRel($stmt, I, mt, dg$)
        $Eqs \leftarrow Eqs \cup$ genOutSizeRel($stmt, mt, SCC$)
    Let $K$ be $bm$ output formal parameter positions
    $Eqs \leftarrow Eqs \cup$ genSizeRel($bm, K, mt, dg$)
    **return** $Eqs$
**end**

genSizeRel($elem, Pos, mt, dg$)
    $Eqs \leftarrow \emptyset$
    **foreach** $pos \in Pos$
        $m$   $\leftarrow$ lookup($mt$, measure, $elem$.sig, $pos$)
        $s$   $\leftarrow$ getSize($m$, $elem$.id, $pos$, $dg$)
        $Eqs \leftarrow Eqs \cup \{\texttt{size}(m, elem.\text{id}, pos) \leq s\}$
    **return** $Eqs$
**end**

getSize($m, id, pos, dg$)
    $result \leftarrow \texttt{val}(m, id, i)$
    **if** $result \neq \infty$ **then**
        **return** $result$
    **elseif** $\exists\ (elem, pos_p) \in$ predec($dg, id, pos$) **then**
        $m_p \leftarrow$ lookup($mt$, measure, $elem$.sig, $pos_p$)
        **if** $(m = m_p)$ **then**
            **return** $\texttt{size}(m_p, elem.\text{id}, pos_p)$
    **return** $\infty$
**end**
```

Figure 9.6: The size analysis algorithm: input arguments

there is a unique predecessor whose size is assigned to that input parameter position, or there is none, causing the input parameter size to be unbounded (∞).

Consider now an output parameter position within a block method, case covered

```
genOutSizeRel($stmt, mt, SCC$)
    Let $I = \{i_1, \ldots, i_l\}$ be the input positions in $stmt$
    $sig \leftarrow stmt$.sig
    $\{m_{i_1}, \ldots, m_{i_l}\} \leftarrow \{$lookup$(mt,$ measure$, sig, i_1)$, ..., lookup$(mt,$ measure$, sig, i_l)\}$
    $\{s_{i_1}, \ldots, s_{i_l}\} \leftarrow \{$size$(m_{i_1}, stmt.$id$, i_1)$, ..., size$(m_{i_l}, stmt.$id$, i_l)\}$
    $Eqs \leftarrow \emptyset$
    Let $O$ be the output parameter positions in $stmt$
    foreach $o \in O$
      $m_o \leftarrow$ lookup$(mt,$ measure$, sig, o)$
      if $sig \notin SCC$ then
        $Size_{user} \leftarrow \mathcal{A}^o_{sig}(s_{i_1}, \ldots, s_{i_l})$
        $Size_{alg'} \leftarrow$ max(lookup$(mt,$ size$, sig, o)$)
        $Size_{alg} \leftarrow Size_{alg'}(s_{i_1}, \ldots, s_{i_l})$
        $Size_o \leftarrow$ min$(Size_{user}, Size_{alg})$
      else
        $Size_o \leftarrow \mathcal{S}ize^o_{sig}(m_o, s_{i_1}, \ldots, s_{i_l})$
      $Eqs \leftarrow Eqs \cup \{$size$(m_o, stmt.$id$, o) \leq Size_o\}$
    return $Eqs$
end

normalize($Eqs$)
    foreach size relation $p \leq e_1 \in Eqs$
      repeat
        if subexpression $s$ appears in $e_1$
            and $s \leq e_2 \in Eqs$ then
            replace each occurrence of $s$ in $e_1$ with $e_2$
      until there is no change
    return $Eqs$
end
```

Figure 9.7: The size analysis algorithm: output arguments and normalization

in genOutSizeRel (Figure 9.7). If the output parameter position corresponds to a non-recursive invoke statement, either a size relationship function has already been computed recursively (since the analysis traverses each strongly-connected component in reverse topological order), or it is provided by the user through size annotations. In the first case, the size function of the output parameter position can be retrieved from the memo table by using the lookup operation, taking the maximum in case of several size relationship functions, and then passing the input parameter size relationships to this function to evaluate it. In the second scenario, the size function of the output parameter position is provided by the user through size annotations, denoted by the $\mathcal{A}$ function in the algorithm. In both cases, it will able to return an explicit size relation function.

Example 9.2.1. (Builtin class). We have already shown in the CellPhone example how a class can be annotated. The Builtin class includes the assignment method asg, annotated as follows:

```
public class Builtin {
  @Size{"size(ret)<=size(o)"}
  public static native Object asg(Object o);
  // ...rest of annotated builtins
}
```

which results in equation:

$$\mathcal{A}^{1}_{\mathsf{asg}}(\mathsf{ref}, \mathtt{size}(\mathsf{ref}, asg, 0)) \leq \mathtt{size}(\mathsf{ref}, asg, 0)$$

.

If the output parameter position corresponds to a recursive invoke statement, the size relationships between the output and input parameters are built as a symbolic size function. Since the input parameter size relations have already been computed, we can establish each output parameter position size as a function described in terms of the input parameter sizes.

At this point, the algorithm has defined size relations for all parameter positions within a block method. However, those relations are either constants or given in terms of the immediate predecessor in the dependency graph. The algorithm rewrites the equation system such that we obtain an equivalent system in which only formal parameter positions are involved. This process is called normalization, shown in Figure 9.7. After normalization, the analysis repeats the same process for all block methods in the same strongly-connected component (SCC). Once every component has been processed, the analysis further simplifies the equations in order to resolve mutually recursive calls among block methods within the same SCC in the simplifyEqs procedure.

In the final step, the analysis submits the simplified system to a recurrence equation solver, recEqsSolver, called from sizeAnalysis) in order to obtain approximated upper-bound closed forms[2].

Example 9.2.2. (Size Relationships). We now illustrate the definitions and algorithm with an example of how the size relations are inferred for the two CellPhone.-sendSms block methods (Fig. 9.3), using the ref measure for reference variables. For simplicity, we omit the measures in the equations. We will refer to the k-th occurrence of a statement *stmt* in a block method as $stmt_k$, and denote CellPhone.sendSms, Encoder.format, and Stream.send by sendSms, format, and send respectively. Finally, we will refer to the size of the input formal parameter position i, corresponding to variable r_i, as s_{r_i}.

The main steps in the process are listed in Figure 9.8. The first block of rows contains the most relevant size parameter relationship equations for the recursive block method, while the second block of rows corresponds to the base case. These size parameter relationship equations are constructed by the analysis by first following the algorithm in Figures 9.6 and 9.7 , and then normalizing them (expressing them in terms of the input formal parameter sizes s_{r_i}). Also, in the first block of rows we observe that the algorithm has returned $6 \times \texttt{size}(\mathsf{ref}, format, 1)$ as upper bound for

[2]The analysis uses the same recurrence solver mentioned in Chapter 4.

Size parameter relationship equations (normalized)

$$
\begin{array}{lll}
\mathtt{size}(gtf_1,0) & \leq & \mathtt{size}(ne,0) \leq s_{r1} \\
\mathtt{size}(gtf_1,2) & \leq & \mathcal{A}^2_{gtf}(\mathtt{size}(gtf_1,0),\_) \leq s_{r1}-1 \\
\mathtt{size}(format,1) & \leq & \mathtt{size}(gtf_1,2) \leq s_{r1}-1 \\
\mathtt{size}(format,2) & \leq & \mathsf{max}(\mathsf{lookup}(mt,\mathsf{size},format,2))(\mathsf{size}(format,2)) \\
 & \leq & \mathsf{max}(s_{r1},6\times s_{r1})(s_{r_1}-1)6\times(s_{r1}-1) \\
\mathtt{size}(send,1) & \leq & \mathtt{size}(format,2) \leq 6\times(s_{r1}-1) \\
\mathtt{size}(gtf_2,0) & \leq & \mathtt{size}(gtf_1,0) \leq s_{r1} \\
\mathtt{size}(gtf_2,2) & \leq & \mathcal{A}^2_{gtf}(\mathtt{size}(gtf_2,0),\_) \leq s_{r1}-1 \\
\mathtt{size}(sendSms,1) & \leq & \mathtt{size}(gtf_2,2) \leq s_{r1}-1 \\
\mathtt{size}(sendSms,5) & \leq & \mathcal{S}ize^5_{sendSms}(\_,\mathtt{size}(sendSms,1),\_,\_) \\
 & \leq & \mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3}) \\
\mathtt{size}(stf_1,0) & \leq & \mathtt{size}(gtf_2,0) \leq s_{r1} \\
\mathtt{size}(stf_1,2) & \leq & \mathtt{size}(sendSms,5) \leq \mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3}) \\
\mathtt{size}(stf_1,3) & \leq & \mathcal{A}^3_{stf}(\mathtt{size}(stf_1,0),\_,\mathtt{size}(stf_1,2)) \\
 & \leq & s_{r1}+\mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3}) \\
\mathtt{size}(stf_2,0) & \leq & \mathtt{size}(stf_1,3) \leq s_{r1}+\mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3}) \\
\mathtt{size}(stf_2,2) & \leq & \mathtt{size}(format,2) \leq 6\times(s_{r1}-1) \\
\mathtt{size}(stf_2,3) & \leq & \mathcal{A}^3_{stf}(\mathtt{size}(stf_2,0),\_,\mathtt{size}(stf_2,2)) \\
 & \leq & 7\times s_{r1}-6+\mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3}) \\
\mathtt{size}(asg,0) & \leq & \mathtt{size}(stf_2,3) \leq 7\times s_{r1}-6+\mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3}) \\
\mathtt{size}(asg,1) & \leq & \mathcal{A}^1_{asg}(\mathtt{size}(asg,0))7\times s_{r1}-6+\mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3})
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{size}(eq,0) & \leq & \mathtt{size}(sendSms,1) \leq s_{r1} \\
\mathtt{size}(eq,1) & \leq & \mathtt{val}(eq,1) \leq 0 \\
\mathtt{size}(asg,0) & \leq & \mathtt{val}(asg,0) \leq 0 \\
\mathtt{size}(asg,1) & \leq & \mathcal{A}^1_{asg}(\mathtt{size}(asg,0)) \leq 0
\end{array}
$$

Output param. size functions for builtins (through annotations)

$$
\begin{array}{rcl}
\mathcal{A}^2_{\mathtt{gtf}}(\mathtt{size}(gtf,0),\_) & \leq & \mathtt{size}(gtf,0)-1 \\
\mathcal{A}^1_{\mathtt{asg}}(\mathtt{size}(asg,0)) & \leq & \mathtt{size}(asg,0) \\
\mathcal{A}^3_{\mathtt{stf}}(\mathtt{size}(stf,0),\_,\mathtt{size}(stf,2)) & \leq & \mathtt{size}(stf,0)+\mathtt{size}(stf,2)
\end{array}
$$

Simplified size equations and closed form solution

$$
\mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1},s_{r2},s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1}=0 \\ 7\times s_{r1}-6+ & \text{if } s_{r1}>0 \\ \mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1}-1,s_{r2},s_{r3}) & \end{cases}
$$

$$
\mathcal{S}ize^5_{sendSms}(s_{r0},s_{r1},s_{r2},s_{r3}) \leq 3.5\times s_{r1}^2-2.5\times s_{r1}
$$

Figure 9.8: Size equations example

the size of the formatted string, $\mathsf{max}(\mathsf{lookup}(mt,\mathsf{size},format,2))$. The result is the maximum of the two upper bounds given by the user for the two implementations for Encoder.format since TrimEncoder.format eliminates any leading and trailing white spaces (thus the output is at most as bigger as the input), whereas UnicodeEncoder.-format converts any special character into its Unicode equivalent (thus the output is at most six times the size of the input), a safe upper bound for the output parameter position size is given by the second annotation.

In the particular case of builtins and methods for which we do not have the code, size relationships are not computed but rather taken from the user @Size annotations. These functions are illustrated in the third block of rows. Finally, in the fourth block of rows we show the recurrence equations built for the output parameter sizes in the block method and in the final row the closed form solution obtained.

9.2.2 Resource Usage Analysis

The core of our framework is the resource usage analysis, whose pseudo code is shown in Figures 9.9 and 9.10. It takes a strongly-connected component of the CFG, including a set of annotations which describe *application programmer-definable* cost functions on a given set of resources, and calculates an expression which is an upper bound on the resource usage made by the program. The algorithm manipulates the same memo table described in Sec. 9.2.1 in order to avoid recomputations and access the size relationships already inferred.

The algorithm is structured in a very similar way to the size analysis (which also allows us to draw from it to keep the explanation within space limits): for each element of the strongly-connected component the algorithm will construct an equation for each block method that shares the same signature representing the resource usage of that block. To do this, the algorithm will visit each invoke statement. There are three possible scenarios, covered by the genStmsRUExpr function. If the signatures of caller and callee(s) belong to the same strongly-connected component, we

```
resourceAnalysis($SCC, res, mt, CFG$)
  $Eqs \leftarrow \emptyset^{|SCC|}$
  foreach $sig$ in $SCC$
    $Eqs[sig] \leftarrow$ genBlocksRUExpr($sig, res, mt, SCC, CFG$)
  $Sols \leftarrow$ recEqsSolver(simplifyEqs($Eqs$))
  foreach $sig$ in $SCC$
    insert($mt$, cost, max($Sols[sig]$))
  return $mt$
end

genBlocksRUExpr($sig, res, mt, SCC, CFG$)
  $Eqs \leftarrow \emptyset$
  $BMs \leftarrow$ getBlocks($CFG, sig$)
  foreach $bm \in BMs$
    $body \leftarrow bm$.body
    $Cost_{body} \leftarrow 0$
    foreach $stmt \in Body$
      $Cost_{stmt} \leftarrow$ genStmtRUExpr($stmt, res, mt, SCC$)
      $Cost_{body} \leftarrow Cost_{body} + Cost_{stmt}$
    $Cost_{bm} \leftarrow$ genBlockRUExpr($bm, res, mt$)
    $Eqs \leftarrow Eqs \cup \{Cost_{bm} \leq Cost_{body}\}$
  return $Eqs$
end
```

Figure 9.9: The resource usage analysis algorithm

are analyzing a recursive invoke statement. Then, we add to the body resource usage a symbolic resource usage function, in an analogous fashion to the case of output parameters in recursive invocations during the size analysis.

The other scenarios occur when the invoke statement is non-recursive. Either a resource usage function $Cost_{alg}$ for the callee has been previously computed, or there is a user annotation $Cost_{usr}$ that matches the given signature, or both. In the latter case, the minimum between these two functions is chosen (i.e., the most precise safe upper bound assigned by the analysis to the resource usage of the non-recursive

```
genStmtRUExpr($stmt, res, mt, SCC$)
  Let $\{i_1, \ldots, i_k\}$ be the input parameter positions in $stmt$
  $\{s_{i_1}, \ldots, s_{i_k}\} \leftarrow \{\mathsf{max}(\mathsf{lookup}(mt, \mathsf{size}, stmt.\mathsf{sig}, i_1))$
                        $, \ldots,$
                        $\mathsf{max}(\mathsf{lookup}(mt, \mathsf{size}, stmt.\mathsf{sig}, i_k))\}$
  if $stmt.\mathsf{sig} \notin SCC$ then
    $Cost_{user} \leftarrow \mathcal{A}_{stmt.\mathsf{sig}}(res, s_{i_1}, \ldots, s_{i_k})$
    $Cost_{alg'} \leftarrow \mathsf{lookup}(mt, \mathsf{cost}, res, stmt.\mathsf{sig})$
    $Cost_{alg} \leftarrow Cost_{alg'}(s_{i_1}, \ldots, s_{i_k})$
    return $\mathsf{min}(Cost_{alg}\ ,\ Cost_{user})$
  else
    return $Cost(stmt.\mathsf{sig}, res, s_{i_1}, \ldots, s_{i_k})$
end

genBlockRUExpr($bm, res, mt$)
  Let $\{i_1, \ldots, i_l\}$ be $bm$ input formal parameter positions
  $\{s_{i_1}, \ldots, s_{i_l}\} \leftarrow \{\mathsf{lookup}(mt, \mathsf{size}, bm.\mathsf{id}, i_1)$
                        $, \ldots,$
                        $\mathtt{lookup}(mt, \mathtt{size}, bm.\mathsf{id}, i_l)\}$
  return $Cost(bm.\mathsf{id}, res, s_{i_1}, \ldots, s_{i_l})$
```

Figure 9.10: The resource usage analysis algorithm (continuation)

invoke statement).

Example 9.2.3. (Resource annotations). Consider the same block method as in the previous example and the invocation of Stream.send. The resource usage expression for the statement is defined by the function $\mathcal{A}_{send}(\$, \_, 6 \times (s_{r1} - 1))$ since the input parameter at position one is at most six times the size of the second input formal parameter, as calculated by the size analysis in Figure 9.8. Note also that there is a resource annotation @Cost({"cents","2*size(r1)"}) attached to the block method describing the behavior of $\mathcal{A}_{send}$ and yielding the expression $Cost_{user} = 12 \times (s_{r1} - 1)$. On the other hand, the absence of any callee code to analyze –the original method is native– results in $Cost_{alg} = \infty$. Then, the upper bound obtained by the analysis

Resource usage equations
$Cost(sendSms, \$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, ne)}^{\infty}, \overbrace{\mathcal{A}_{ne}(\$, s_{r1}, \_)}^{\text{@Cost("cents","0")=0}}) + \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, gtf)}^{\infty}, \overbrace{\mathcal{A}_{gtf}(\$, s_{r1}, \_)}^{\text{@Cost("cents","0")=0}}) + \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, format)(\_, s_{r1}-1)}^{0}, \overbrace{\mathcal{A}_{format}(\$, \_, s_{r1}-1)}^{\infty}) + \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, send)}^{\infty}, \overbrace{\mathcal{A}_{send}(\$, \_, 6 \times (s_{r1}-1))}^{\text{@Cost("cents","2*size(r1)")=}12\times(s_{r1}-1)}) + \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, gtf)}^{\infty}, \overbrace{\mathcal{A}_{gtf}(\$, s_{r1}, \_)}^{\text{@Cost("cents","0")=0}}) + Cost(sendSms, \$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3}) + \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, stf)}^{\infty}, \overbrace{\mathcal{A}_{stf}(\$, s_{r1}, \_, \_)}^{\text{@Cost("cents","0")=0}}) + \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, stf)}^{\infty}, \overbrace{\mathcal{A}_{stf}(\$, s_{r1}, \_, \_)}^{\text{@Cost("cents","0")=0}}) + \min(\overbrace{\mathsf{lookup}(mt, \mathsf{cost}, \$, asg)}^{\infty}, \overbrace{\mathcal{A}_{asg}(\$, \_)}^{\text{@Cost("cents","0")=0}}) \leq 12 \times (s_{r1}-1) + Cost(sendSms, \$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3})$
$Cost(sendSms, \$, s_{r0}, 0, s_{r2}, s_{r3}) \leq \min(\overbrace{\mathtt{lookup}(mt, \mathtt{cost}, \$, eq)}^{\infty}, \overbrace{\mathcal{A}_{eq}(\$, 0, \_)}^{\text{@Cost("cents","0")=0}}) + \min(\underbrace{\mathtt{lookup}(mt, \mathtt{cost}, \$, asg)}_{\infty}, \underbrace{\mathcal{A}_{asg}(\$, 0)}_{\text{@Cost("cents","0")=0}}) \leq 0$
Simplified resource usage equations and closed form solution
$Cost(sendSms, \$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq \begin{cases} 0 & \text{if } s_{r1} = 0 \\ 12 * s_{r1} - 12 + & \text{if } s_{r1} > 0 \\ Cost(sendSms, \$, s_{r0}, s_{r1}-1, s_{r2}, s_{r3}) & \end{cases}$
$Cost(sendSms, \$, s_{r0}, s_{r1}, s_{r2}, s_{r3}) \leq 6 \times s_{r1}^2 - 6 \times s_{r1}$

Figure 9.11: Resource equations example

for the statement is $\min(Cost_{alg}, Cost_{user}) = Cost_{user}$.

At this point, the analysis has built a resource usage function (denoted by

$Cost_{body}$) that reflects the resource usage of the statements within the block. Finally, it yields a resource usage equation of the form $Cost_{block} \leq Cost_{body}$ where $Cost_{block}$ is again a symbolic resource usage function built by replacing each input formal parameter position with its size relations in that block method. These resource usage equations are simplified by calling simplifyEqs and, finally, they are solved calling recEqsSolver, both already defined in Sec. 9.2.1. This process yields an (in general, approximate, but always safe) closed form upper bound on the resource usage of the block methods in each strongly-connected component. Note that given a signature the analysis constructs a closed form solution for every block method that shares that signature. These solutions approximate the resource usage consumed in or provided by each block method. In order to compute the total resource usage of the signature the analysis returns the maximum of these solutions yielding a safe global upper bound.

Example 9.2.4. (Resource usage equations). The resource usage equations generated by our algorithm for the CellPhone.sendSms block methods and the resource denoted by \$ (i.e., monetary total cost of sending the SMSs through a cell phone) are listed in Figure 9.11. The computation is in part based on the size relations for each output parameter position in Figure 9.8. The resource usage of each block method is calculated by building an equation such that the left part is a symbolic function constructed by replacing each parameter position with its size (i.e., $Cost(sendSms, \$, s_{r0}, s_{r1}, s_{r2}, s_{r3})$ and $Cost(sendSms, \$, s_{r0}, 0, s_{r2}, s_{r3})$), and the rest of the equation consists of adding the resource usage of the invoke statements in the block method. These are calculated by computing the minimum between the resource usage function inferred by the analysis and the function provided by the user. The equations corresponding to the recursive and non-recursive block methods are in the first and second row, respectively. They can be simplified (third row) and expressed in closed form (fourth row), obtaining a final upper bound for the charge incurred by sending the list of text messages of $6 \times s_{r1}^2 - 6 \times s_{r1}$.

9.3 Experimental Results

We have completed an implementation of our framework, and tested it for a representative set of benchmarks and resources. Our experimental results are summarized in Tables 9.1 and 9.2. Column Program provides the name of the main class to be analyzed. Column Resource(s) shows the resource(s) defined and tracked. Column Size T. shows the time (in milliseconds) required by the size analysis to construct the size relations (including the data dependency analysis and class hierarchy analysis) and obtain the closed form. Column Res. T. lists the time taken to build the resource usage expressions for all method blocks and obtain their closed form solutions. Total T. provides the total times for the whole analysis process. Finally, column Resource Usage Func. provides the upper bound functions inferred for the resource usage. For simplicity, we only show the most important (asymptotic) component of these functions, but the analysis yields concrete functions with constants.

Regarding the benchmarks we have covered a reasonable set of data-structures used in object-oriented programming and also standard Java libraries used in real applications. We have also covered an ample set of application-dependent resources which we believe can be relevant in those applications. In particular, not only have we represented high-level resources such as cost of SMS, bytes received (including a coarse measure of bandwidth, as a ratio of data per program step), and files left open, but also other low-level (i.e., bytecode level) resources such as stack usage or energy consumption. The resource usage functions obtained can be used for several purposes. In program Files (a fragment characteristic of operating system kernel code) we kept track of the number of file descriptors left open. The data inferred for this resource can be clearly useful, e.g., for debugging: the resource usage function inferred in this case ($O(n)$) denotes that the programmer did not close $O(n)$ file descriptors previously opened. In program Join (a database transaction which carries out accesses to different tables) we decided to measure the number of

Program	Resource(s)	Size T.	Res. T.	Total T.
BST	Heap usage	250	22	367
CellPhone	SMS monetary cost	271	17	386
Client	Bytes received and "Bandwidth" required	391	38	527
Dhrystone	Energy consumption	602	47	759
Divbytwo	Stack usage	142	13	219
Files	Files left open and Data stored	508	53	649
Join	DB accesses	334	19	460
Screen	Screen width	388	38	536

Table 9.1: Times in ms of different phases of the resource analysis on a Pentium M 1.73Ghz with 1Gb of RAM.

accesses to such external tables. This information can be used, e.g., for resource-oriented specialization in order to perform optimized checkpoints in transactional systems. The rest of the benchmarks include other definitions of resources which are also typically useful for verifying application-specific properties: BST (a generic binary search tree, used in [3] where a heap space analysis for Java bytecode is presented), CellPhone (extended version of program in Figure 9.3), Client (a socket-based client application), *Dhrystone* (a modified version of a program from [67] where a general framework is defined for estimating the energy consumption of embedded JVM applications; the complete table with the energy consumption costs that we used can be found there), DivByTwo (a simple arithmetic operation), and Screen (a MIDP application for a cellphone, where the analysis is used to make sure that message lines do not exceed the phone screen width). The benchmarks also cover a good range of complexity functions ($O(1), O(log(n), O(n), O(n^2) \ldots, O(2^n), \ldots$) and different types of structural recursion such as simple, indirect, and mutual.

Program	Resource Usage Func.	
BST	$O(2^n)$	$n \equiv$ tree depth
CellPhone	$O(n^2)$	$n \equiv$ packets length
Client	$O(n)$	$n \equiv$ stream length
	$O(1)$	—
Dhrystone	$O(n)$	$n \equiv$ int value
Divbytwo	$O(log_2(n))$	$n \equiv$ int value
Files	$O(n)$	$n \equiv$ number of files
	$O(n \times m)$	$m \equiv$ stream length
Join	$O(n \times m)$	$n, m \equiv$ records in tables
Screen	$O(n)$	$n \equiv$ stream length

Table 9.2: Resource usage functions for programs described in Table 9.1.

9.4 Related Work

We start by noting that while the analysis described in Chapter 4 was also parametric it was designed for Prolog and works at the source code level, and thus cannot be applied to Java bytecode, at least directly, due to issues like virtual method invocation, unstructured control flow, assignment, the fact that statements are low-level bytecode instructions, etc., as well as the absence of backtracking (which had a significant impact on the method presented in Chapter 4). Also, the presentation of Chapter 4 is descriptive in contrast to the concrete algorithm provided in this chapter. With respect to related work, in [2], a cost analysis is described (developed independently from the one described herein) that does deal with Java bytecode and is capable of deriving cost relations which are functions of input data sizes. However, while the approach proposed can conceptually be adapted to infer different resources, for each analysis developed the measured resource is fixed and changes in the implementation are needed to develop analyses for other resources. In contrast, our approach allows the application programmer to define the resources through annotations in the Java source, and without changing the analyzer in any way. In addition, the presentation in [2] is again descriptive, while herein we provide a concrete, memo table-based analysis algorithm, as well as implementation results.

9.5 Summary

This chapter has presented a fully-automated analysis for inferring upper bounds on the usage that a Java bytecode program makes of a set of application programmer-definable resources. The analysis presented derives a vector of functions, one for each defined resource. Each of these functions returns, for each given set of input data sizes, an upper bound on the usage that the whole program (and each individual method) make of the corresponding resource. Important novel aspects of our approach are the fact that it allows the application programmer to define the resources to be tracked by writing simple resource descriptions via source-level annotations, as well as the fact that we have provided a concrete analysis algorithm and report on an implementation. The current results show that the proposed analysis can obtain non-trivial bounds on a wide range of interesting resources in reasonable time.

Another important aspect, because of its impact on the scalability, precision, and automation of the analysis, is that our approach allows using the annotations also for a number of other purposes such as stating the resource usage of external methods, which is instrumental in allowing modular composition and thus scalability. In addition, our annotations allow stating the resource usage of any method for which the automatic analysis infers a value that is not accurate enough to prevent inaccuracies in the automatic inference from propagating. Annotations are also used by the size and resource usage analysis to express their output. Finally, the annotation language can also be used to state specifications related to resource usage, which can then be proved or disproved based on the results of analysis following, e.g., the `CiaoPP` scheme of [54] thus finding bugs or verifying (the resource usage of) the program.

Chapter 10

Conclusions and Future Work

10.1 Conclusions

Resource usage analysis is increasingly important in the context of applications such as granularity control in parallel and distributed computing, resource-oriented specialization, or, more recently, certification of the resources used by mobile code. Specially in these more recent applications, the properties of interest are often higher-level, user-oriented, and application-dependent rather than (or, better, in addition to) the predefined, more traditional costs. Note that traditional cost analyses with a fixed set of resources are not sufficient.

This thesis has covered two main lines. The first one consists of Chapters 4, 6, and 7, and it has been devoted to several analyses associated with the inference of resource usage information for logic programs, presenting the following contributions:

- Chapter 4 has presented a resource bounds analysis that infers automatically lower- and upper-bounds on the usage that a logic program makes of a set of user-definable resources within a single implementation. The chapter has also presented the assertion language which is used to define such resources. The resource usage functions are, in general, functions in terms of the sizes of the input data. Moreover, the chapter discussed the implementation completed of such analysis and experimental results. The experimental evaluation is encouraging because it shows that interesting resource bound functions can be obtained automatically and in reasonable time, for a representative set of benchmarks with a good variety of resources such as bits sent or received by an application over a socket, number of files left open, number of accesses to a database, energy consumption, etc., as well as the more traditional execution steps, execution time, or heap memory. To the best of our knowledge this is the first user-definable resource analysis proposed in the literature.

- Regarding the automatic inference of resource usage information, Chapter 4 pointed out that the inference at compile-time of which variables do not share provides an invaluable source of information for the resource usage analysis among other things because it implies the determination of input/output modes (Set-Sharing). In Chapters 6 and 7 we have presented two different approaches to mitigating the inefficiencies of the Set-Sharing analysis:

  – Chapter 6 has described a Set-Sharing analysis for top-down frameworks based on the definition of several new widening operators in order to accelerate the fixed point computation to converge, providing different levels of precision and efficiency tradeoff. The approach has also included the case of combining with freeness information in order to increase the precision of the analysis. The analysis has shown, in general, relevant efficiency gains with limited precision losses. More interestingly, some benchmarks that ran out of memory with the original Set-Sharing analysis on our test platform have been analyzed by our approach.

  – Chapter 7 has presented another novel approach to improving the efficiency of Set-Sharing that leverages the complement or negative sharing relationships of the original sharing set. Our experimental evaluation has shown that our approach may reduce significantly the memory usage of the sharing relationships and the running time of the abstract operations, including the abstract unification. Our experiments have also shown how to set up key parameters in our algorithms in order to control the desired compression and time complexities. We have shown that we can obtain a reasonable compression in polynomial time by tuning appropriately those parameters.

We believe that our results have shown that both approaches can contribute to the practical, scalable application of Set-Sharing. Notice that both of the approaches are most advantageous when the size of the sharing relationships is considerably large. Otherwise, the traditional Set-Sharing behaves properly and alternative approaches are not required. Moreover, the choice of one or another will depend on each application. In some cases, some losses of accuracy may be tolerable. If this is the case, our widening-based approach may fit perfectly. In other situations, the lack of accuracy may be unacceptable. Then, our negative approach should be considered rather than the widening-based one.

The fact that the source code of many applications and tools is sometimes not available led us in the second part of this thesis (consisting of Chapters 8 and 9) to concentrate on the development of an analysis tool for the inference of resource usage information for bytecode. This work resulted in the following contributions:

- Chapter 8 presented a novel abstract interpretation framework, which is generic in terms of the abstract domain in use. The framework makes use of a decompilation phase that results in an analysis-friendly intermediate representation which can be also used for other non abstract interpretation-based analyses of Java bytecode, and an analysis phase based upon an efficient and precise fixpoint algorithm. The experimental evaluation has shown the feasibility of the approach with medium-size programs, using the benchmarks in the frequently used JOlden suite.

- Chapter 9 presented a generic resource usage analysis for Java bytecode. This work has been inspired by the analysis for logic programs, but required adaptations from logic programs to Java bytecode related to virtual method invocation, exceptions, unstructured control flow, assignment, etc. Moreover, other pre-analysis steps were required to generate more precise bounds. The analysis framework described in Chapter 8 helped us to solve many of these problems. We have used its intermediate language in order to obtain a uniform representation that is easier for the resource usage analysis to handle. Furthermore, we plugged in some abstract domains into the analysis framework to improve the precision of the bounds. We have also shown some experimental results which show that our technique can obtain non-trivial bounds on a wide range of interesting resources in reasonable time, supporting the practicability of the solution adopted.

10.2 Future Work

Finding an upper bound on the cost of computations is an undecidable [113] problem because it can be reduced to solving the halting problem. Given a Turing-equivalent program, we infer its upper bound cost function. If this cost function is ∞ then the program does not halt. Otherwise, the program halts. A similar reasoning can be established for lower bounds. Therefore, we can only hope to develop resource usage analysis that succeed for larger and larger classes of programs, even if there will always be some resource-usage bounded programs which cannot be proved to be bounded.

The set of programs for which the resource usage analysis can infer non trivial bounds can be extended if more information at compile-time is known. Set-Sharing analysis can only abstract information relative to the variables themselves. A more sophisticated model is to consider the shape of data structures in the memory at a program point. Having a more precise vision of the actual graph structure of the heap, not only the resource usage analysis could detect cyclic structures but also reasoning about other type of data structures, e.g., choosing more appropriate metrics. In Java bytecode, we could do that by applying some of the numerous and successful shape analyses, see [107, 16, 106] and their references. In logic programming, it could be more interesting since we may need to develop more accurate shape-type analyses than the current state of the art. Moreover, analysis of larger programs could raise other issues related to efficiency. Therefore, the study of more efficient shape-type analysis seems to be also a future line of research.

On the other hand, we feel that the solution to efficient and precise Set-Sharing analyses may rely on the definition of more compact and effective encodings rather than the use of widening operators. In particular, the encoding of negative sharing relationships on the top of (or in addition to) other efficient representations such as, e.g., Binary Decision Diagrams seems to be another very promising future investigation.

An important challenge for the future is being able to infer the resource usage information of larger and larger, real programs. We have seen that the undecidability of the problem restricts us to a limited subset of programs. In addition, programmers may use complex, intertwined, non-monotonic loops (i.e., loops in which argument sizes may increase or decrease), and complex data structures which may make it very complex to reason about them in terms of the resource usage consumed or provided.

At first sight this would imply that such future work (and, ultimately, the ultimate objective of this thesis) is unachievable. However, in fact, we are rather optimistic and of the opinion that while analyzing automatically 100% of some programs will obviously always remain impossible, automatic resource analysis can be made to be of great help in practice and in large, real program. Thus, we consider an essential part of future work to work towards the previously stated objective of improving the scalability of the analysis for large programs. Our vision towards reaching this goal is based on a number of ideas:

- We believe, based on our experience, that between a large portion of the programs consists of linear code and relatively simple loops which our resource usage analyses can easily deal with. Therefore, the programmer can be liberated from the painful task of working out the complexity of these parts. While for the remaining percentage we do not expect, in the short-term, an automatic solution, the assertion language defined in Chapter 4 can clearly mitigate the problem. This language, which we used for defining the resource usage of external procedures (i.e., libraries), also allows programmers to describe by hand the resource usage of any procedure for which the automatic analysis infers a value that is not accurate enough. This can be used to prevent inaccuracies in the automatic inference from propagating. Thus, the manual work is reduced to a hopefully small part of the program.
- Moreover, we are also working currently on making our inference of resource usage more modular, taking advantage of the compositional nature of the cost of computations. We think that a modular approach is required to deal with large programs.
- Finally, we also plan to work on improving the intrinsic power of the solver and enlarging the class of loops for which accurate bounds can be obtained, relaxing a number of the requirements of our current analysis.

In conclusion, while it is probably true that totally automatic resource analysis of very large programs may be far in the future, this is not a necessary condition for our analyses to be very useful in practice. The real practical benefit will come from the fact that the tool typically will take care of the a large portion of the analysis tasks required, even if a few of the more complex parts will always still be better analyzed by the user. These results from manual analysis can then be fed to the compiler via assertions, which will then compose them with the automatic analysis results for the other, generally much larger parts of the program, thus relieving the programmer from large amounts of work, and obtaining results from the complete program.

Appendix A

Proofs

Lemma 1. *Let $cl \in CL$, $sh \in SH$, $ss \in SH$, $\Downarrow cl \cup ss \supseteq sh$, and $t \in Term$. Then:*

$$\Downarrow rel(cl,t) \ \cup \ rel(ss,t) \ \supseteq \ rel(sh,t) \tag{A.1}$$

$$irrel(\Downarrow cl,t) \ \cup \ irrel(ss,t) \ \supseteq \ irrel(sh,t) \tag{A.2}$$

$$\cup(rel(cl,t) \ \cup \ rel(ss,t)) \supseteq \cup(rel(sh,t)) \tag{A.3}$$

Proof. Since $\Downarrow cl \cup ss \supseteq sh$, we have that $rel(\Downarrow cl \ \cup \ ss,t) \ \supseteq \ rel(sh,t)$. Also $\Downarrow rel(cl,t) \ \cup \ rel(ss,t) \ \supseteq \ rel(\Downarrow cl \ \cup \ ss,t)$ (A.8). Thus, (A.1) follows.

For (A.2), the following is straightforward:

$$\Downarrow cl \cup ss \supseteq sh \Rightarrow irrel(\Downarrow cl \ \cup \ ss,t) \supseteq irrel(sh,t) \Rightarrow$$
$$irrel(\Downarrow cl,t) \ \cup \ irrel(ss,t) \ \supseteq irrel(sh,t)$$

To see (A.3), note that $\cup(rel(cl,t) \ \cup \ rel(ss,t)) = \cup(\Downarrow rel(cl,t) \ \cup \ rel(ss,t))$, since both expressions represent the same set of variables. But, from (A.1),

$$\Downarrow rel(cl,t) \ \cup \ rel(ss,t) \ \supseteq \ rel(sh,t)$$

so that the result follows directly. □

The following results have been already proved or are straightforward from set theory:

Lemma 2. *Let ss_1, ss_2, and ss_3 be sets of sets:*

$$(ss_1 \bowtie ss_2)^* = ss_1^* \bowtie ss_2^* \tag{A.4}$$

$$(ss_1 \cup \{\emptyset\})^* = ss_1^* \cup \{\emptyset\} \tag{A.5}$$

$$ss_1 \bowtie (ss_2 \cup ss_3) = (ss_1 \bowtie ss_2) \cup (ss_1 \bowtie ss_3) \tag{A.6}$$

If both $ss_1 \neq \emptyset$ and $ss_2 \neq \emptyset$ then:

$$\cup(ss_1 \bowtie ss_2) = \cup(ss_1 \cup ss_2) \tag{A.7}$$

The following result characterizes operation $(cl, sh)^*$ for $(cl, sh) \in SH^W$ with an equivalent expression, which makes more amenable the proof of correctness of the *extend*^{s} function:

Lemma 3. *Let $(cl, sh) \in SH^W$ and $(cl, sh)^* = (cl', sh')$:*

$$\Downarrow cl' = \Downarrow(cl^* \cup (cl^* \bowtie sh^*))$$

Proof. By definition, if $cl = \emptyset$ then $cl' = \emptyset$, otherwise $cl' = \{\cup(cl \cup sh)\}$. If $cl = \emptyset$ the result is trivial, since both expressions in the equality reduce to $\emptyset$. Let then $cl \neq \emptyset$. We now have that $cl' = \{\cup(cl \cup sh)\}$ so that $\Downarrow cl' = \downarrow\cup(cl \cup sh)$. Since $cl \neq \emptyset$ and also $sh \cup \{\emptyset\} \neq \emptyset$, we can apply (A.7), so that we can write:

$$\begin{array}{rclcl}
\Downarrow(cl^* \cup (cl^* \bowtie sh^*)) & \overset{(A.6)}{=} & \Downarrow(cl^* \bowtie (sh^* \cup \{\emptyset\})) & \overset{(A.5)}{=} & \Downarrow(cl^* \bowtie (sh \cup \{\emptyset\})^*) \\
& \overset{(A.4)}{=} & \Downarrow(cl \bowtie (sh \cup \{\emptyset\}))^* & \overset{(A.9)}{=} & \downarrow\cup(cl \bowtie (sh \cup \{\emptyset\})) \\
& \overset{(A.7)}{=} & \downarrow\cup(cl \cup sh \cup \{\emptyset\}) & = & \downarrow\cup(cl \cup sh) \\
& = & \Downarrow cl' & &
\end{array}$$

which proves the result. □

Lemma 4. *Let $\mathcal{I}((cl, sh)) = \Downarrow cl \cup sh$. Let $clsh \in SH^W$, $clsh_1 \in SH^W$, $clsh_2 \in SH^W$. Then:*

$$rel(\mathcal{I}(clsh), t) \subseteq \mathcal{I}(rel(clsh, t)) \tag{A.8}$$

$$irrel(\mathcal{I}(clsh), t) = \mathcal{I}(irrel(clsh, t))$$

$$\mathcal{I}(clsh_1) \cup \mathcal{I}(clsh_2) = \mathcal{I}(clsh_1 \cup^W clsh_2)$$

$$\mathcal{I}(clsh_1) \bowtie \mathcal{I}(clsh_2) \subseteq \mathcal{I}(clsh_1 \bowtie clsh_2)$$

$$(\mathcal{I}(cl, sh))^* \subseteq \mathcal{I}((cl, sh)^*)$$

Theorem 6.2.1 Let $(cl, ss) \in SH^W$, $sh \in SH$, equation $x = t$, $x \in \mathcal{V}$ and $t \in Term$, and $amgu^W(x, t, (cl, ss)) = (cl^o, ss^o)$. If $\Downarrow cl \cup ss \supseteq sh$ then:

$$\Downarrow cl^o \cup ss^o \supseteq amgu(x, t, sh)$$

Proof. Direct from the definition of $amgu^W$ and Lemma 4. □

Note that the previous result holds even for the case in which $\Downarrow cl \cup ss = sh$. That is, $amgu^W$ is neccessarily imprecise.

Proposition 1. *Let $(cl, ss) \in SH^W$, $sh \in SH$, equation $x = t$, $x \in \mathcal{V}$ and $t \in Term$, and $amgu^s(x, t, (cl, ss)) = (cl^o, ss^o)$. If $\Downarrow cl \cup ss = sh$ then:*

$$\Downarrow cl^o \cup ss^o \supseteq amgu(x, t, sh)$$

but not in general $\Downarrow cl^o \cup ss^o = amgu(x, t, sh)$.

Proof. The general statement is a direct corollary of Theorem 6.2.1. To see that equality does not hold in general, take $(cl, ss) = (\{\{X, Y\}\}, \emptyset)$ and $sh = \{\{X\},$-$\{X, Y\}, \{Y\}\}$. We have $\Downarrow cl \cup ss = sh$. Take also $t = y$. Then $(cl^o, ss^o) =$

$(\{\{X,Y\}\},\emptyset)$, so that $\Downarrow cl^o \cup ss^o = \{\{X\},\{X,Y\},\{Y\}\}$. But $amgu(x,t,sh) = irrel(sh, x=t) \cup (rel(sh,x) \bowtie rel(sh,t))^* = \{\{X,Y\}\}$, which is a proper subset of $\Downarrow cl^o \cup ss^o$. $\square$

The optimization of $amgu^W$ that we present here is similar to that presented in [120] for the case of inferring pair-sharing. The following two results will be instrumental. In the proofs the *baseline* of a set of sets SS is denoted by the set of elements in sets belonging to SS, i.e., $\cup SS = \bigcup_{S \in SS} S$. Note that the first one allows to safely replace star-union on clique sets simply by set union (which is precisely the observation behind the definition of $(cl, sh)^*$ in [121]):

Lemma 5. *For every $cl \in CL$:*[1]

$$\Downarrow cl^* = \downarrow \cup cl \tag{A.9}$$

Proof. First, $\Downarrow cl^* \subseteq \downarrow \cup cl$. Note that $\downarrow \cup cl \supseteq \Downarrow cl$, since $\cup cl$ is baseline of cl, and therefore the maximal element that can belong to cl (which gives the maximal powerset possible for $\Downarrow cl$). Also, $\cup cl^* = \cup cl$, since the baseline of cl and of cl^* is the same. Thus, $\Downarrow cl^* \subseteq \downarrow \cup cl^* = \downarrow \cup cl$.

Also, $\downarrow \cup cl \subseteq \Downarrow cl^*$. To see this, take $s \in \downarrow \cup cl$, we also have that, if $cl \neq \emptyset$, $\cup cl \in cl^*$ ($\cup cl$ is the maximal element of cl^*; however, if $cl = \emptyset$, $cl^* = \emptyset$, too). Thus, from the definition, $s \in \Downarrow cl^*$. If $cl = \emptyset$, the result follows directly. $\square$

Lemma 6. *For every $cl \in CL$ and $s \in \wp^0(V)$. If $cl \neq \emptyset$:*

$$\Downarrow(\{s\} \bowtie cl^*) = \Downarrow\{\cup cl \cup s\} \tag{A.10}$$

[1]Note that $\Downarrow cl^* = \Downarrow(cl^*)$.

Proof.

$$\begin{array}{rcll}
\Downarrow(\{s\} \bowtie cl^*) & = & \Downarrow(\{s\}^* \bowtie cl^*) & \text{since } \{s\} \text{ is a singleton set} \\
& = & \Downarrow(\{s\} \bowtie cl)^* & \text{by (A.4)} \\
& = & \downarrow\cup(\{s\} \bowtie cl) & \text{by (A.9)} \\
& = & \downarrow(\cup cl \cup s) & \text{since the baseline of } \{s\} \bowtie cl \text{ is, if } cl \neq \emptyset, \\
& & & \text{that of } cl \text{ plus the elements in } s \\
& = & \Downarrow\{\cup cl \cup s\} & \text{since } \cup cl \cup s \text{ is a singleton set}
\end{array}$$

□

Let $(cl, sh) = clsh \in SH^W$. By definition:

$$amgu^W(x, t, clsh) = irrel(clsh, x = t) \ \cup^W \ (rel(clsh, x)^* \ \bowtie rel(clsh, t)^*)$$

so that using the definitions of $\cup^W$ and $\bowtie$, and considering that there are two cases in $rel(clsh, x)^*$ and another two in $rel(clsh, t)^*$:

$$amgu^W(x, t, clsh) = (irrel(cl, x = t) \ \cup \ cl', irrel(sh, x = t) \ \cup \ sh')$$

$$(cl', sh') = \begin{cases}
(\emptyset, rel(sh, x)^* \bowtie rel(sh, t)^*) & \text{if } rel(cl, x) = rel(cl, t) = \emptyset \\
(\{\cup(rel(cl, t) \cup rel(sh, t))\} \bowtie rel(sh, x)^*, \emptyset) & \text{if } rel(cl, x) = \emptyset, rel(cl, t) \neq \emptyset \\
(\{\cup(rel(cl, x) \cup rel(sh, x))\} \bowtie rel(sh, t)^*, \emptyset) & \text{if } rel(cl, x) \neq \emptyset, rel(cl, t) = \emptyset \\
(\{\cup(rel(cl, x) \cup rel(cl, t) \cup rel(sh, x) & \text{if } rel(cl, x) \neq \emptyset, rel(cl, t) \neq \emptyset \\
\ \cup rel(sh, t))\}, \emptyset) &
\end{cases}$$

However, the second and third cases can be reduced to the last one. Note that, in the second case, if $rel(sh, x) \neq \emptyset$ then, by (A.10), we have $cl' = \{\cup(rel(cl, t) \cup rel(sh, x) \cup rel(sh, t))\}$. Since in this case $rel(cl, x) = \emptyset$, we can write $cl' = \{\cup(rel(cl, x) \cup rel(cl, t) \cup rel(sh, x) \cup rel(sh, t))\}$. However, if $rel(sh, x) = \emptyset$ then $cl' = \emptyset$. The same reasoning can be applied to the third case. Thus:

$$(cl', sh') = \begin{cases} (\emptyset, rel(sh,x)^* \bowtie rel(sh,t)^*) & \text{if } rel(cl,x) = rel(cl,t) = \emptyset \\ (\emptyset, \emptyset) & \text{if } rel(cl,x) = rel(sh,x) = \emptyset \\ & \text{or } rel(cl,t) = rel(sh,t) = \emptyset \\ (\{\cup(rel(cl,x) \cup rel(cl,t) \cup & \text{otherwise} \\ \ rel(sh,x) \cup rel(sh,t)\}, \emptyset) & \end{cases}$$

Finally, note that in the first case, since $rel(cl,x) = rel(cl,t) = \emptyset$, we have that $irrel(cl, x = t) = cl$, which gives the abstract unification operation we have implemented.

Now, it is proved that the lifted linearity "operator" lin^s is correct w.r.t. lin.

Lemma 7. *Let $(cl, ss) \in SH^W$, $sh \in SH$, $sh \subseteq \Downarrow cl \cup ss$, and $t \in Term$. Then:*

$$lin^s(t) \textit{ for } (cl, ss) \Rightarrow lin(t) \textit{ for } sh$$

given that:

$$\begin{aligned} lin^s(t) \;\Leftrightarrow\; & \forall y \in \hat{t} : [t]_y = 1 & \wedge \\ & \forall z \in \hat{t} : y \neq z \rightarrow rel(cl,y) \cap rel(cl,z) = \emptyset & \wedge \\ & rel(sh,y) \cap rel(sh,z) = \emptyset & \end{aligned}$$

$$lin(t) \Leftrightarrow \forall y \in \hat{t} : [t]_y = 1 \wedge \forall z \in \hat{t} : y \neq z \rightarrow rel(sh,y) \cap rel(sh,z) = \emptyset$$

Proof. Let $lin^s(t)$ hold. Then, for all $y \in \hat{t}$, $[t]_y = 1$. Also, for all $z \in \hat{t}$ s.t. $y \neq z$ we have $rel(cl,y) \cap rel(cl,z) = \emptyset$, $rel(ss,y) \cap rel(ss,z) = \emptyset$. Assume in what follows that $y \neq z$.

We also have that $rel(\Downarrow cl, y) \cap rel(\Downarrow cl, z) = \emptyset$, $rel(\Downarrow cl, y) \cap rel(ss, z) = \emptyset$, and $rel(ss,y) \cap rel(\Downarrow cl, z) = \emptyset$ (see below). Hence, $(rel(\Downarrow cl, y) \cup rel(ss,y)) \cap (rel(\Downarrow cl, z) \cup$

$rel(ss, z)) = rel(\Downarrow cl \cup ss, y) \cap rel(\Downarrow cl \cup ss, z) = \emptyset$. But $sh \subseteq \Downarrow cl \cup ss$, so that $rel(sh, y) \cap rel(sh, z) = \emptyset$. Thus, $lin(t)$ holds.

To see that $rel(\Downarrow cl, y) \cap rel(\Downarrow cl, z) = \emptyset$ we reason by contradiction. Let $s \in rel(\Downarrow cl, y) \cap rel(\Downarrow cl, z)$. We have that $s \in \Downarrow cl$, $y \in s$, $z \in s$. Then, there is $c \in cl$ such that $s \subseteq c$, $y \in c$, $z \in c$. Therefore, $c \in rel(cl, y)$ and $c \in rel(cl, z)$, so that $rel(cl, y) \cap rel(cl, z) \neq \emptyset$.

To see that $rel(\Downarrow cl, y) \cap rel(ss, z) = \emptyset$ we also reason by contradiction. Let $s \in rel(\Downarrow cl, y) \cap rel(ss, z)$. We have that $s \in \Downarrow cl$, $s \in ss$, $y \in s$, $z \in s$. Therefore, $s \in rel(ss, y)$ and $s \in rel(ss, z)$, so that $rel(ss, y) \cap rel(ss, z) \neq \emptyset$. The proof of $rel(ss, y) \cap rel(\Downarrow cl, z) = \emptyset$ is the same, exchanging y and z. $\square$

Theorem 6.3.1 Let $((cl, ss), f) \in SHF^W$, $(sh, e) \in SHF$, and equation $x = t$, $x \in \mathcal{V}$, $t \in Term$. Let also $amgu^{sf}(x, t, ((cl, ss), f)) = ((cl^o, ss^o), f^o)$ and $amgu^f(x, t, (sh, e)) = (sh', f')$. If $\Downarrow cl \cup ss \supseteq sh$ and $f \subseteq e$ then:

$$\Downarrow cl^o \cup ss^o \supseteq sh' \text{ and } f^o \subseteq f'$$

Proof. That $\Downarrow cl^o \cup ss^o \supseteq sh'$ follows directly from the definition of $amgu^{sf}$ using Lemma 4 and the following observation based on Lemma 7: If $amgu^{sff}$ is used then $x \in f \subseteq e$ or $t \in f \subseteq e$, so that the first case of $amgu^f$ would have also been used in Sharing+Freeness. Also, if it is $amgu^{sfl}$ that is used then we have that $\hat{t} \subseteq f \subseteq e$ and $lin^s(t)$, which implies $lin(t)$ (Lemma 7); so that the second case of $amgu^f$ would have also been used.

We show that $f^o \subseteq f'$, given that $f \subseteq e$ and the rest of conditions of the theorem, in particular, $\Downarrow cl \cup ss \supseteq sh$. From the definition of amg^{sf} we have four cases. We will also have four more subcases of the last case. Note that in every case $f^o \subseteq f$ (by definition of $amgu^{sf}$). Thus:

- $x \in f$ and $t \in f$

  In this case, since $f \subseteq e$, we have $x \in e$ and $t \in e$, so that $f' = e$ (by definition of $amgu^f$). Also, $f^o = f$ (by definition of $amgu^{sf}$). Thus, the result is straightforward.

- $x \notin f$ and $t \in f$

  Now, we have $t \in e$, but either $x \in e$ or $x \notin e$. If $x \in e$, we have $f' = e$. Thus, the result is straightforward, since $f^o \subseteq f$ and $f \subseteq e$.

  If $x \notin e$, we have $f' = e \setminus \cup rel(sh, t)$. Also, $f^o = f \setminus \cup(rel(cl, t) \cup rel(ss, t))$, so that what we have to prove is:

  $$f \setminus \cup(rel(cl, t) \cup rel(ss, t)) \subseteq e \setminus \cup rel(sh, t)$$

  which holds because $f \subseteq e$, and $\cup(rel(cl, t) \cup rel(ss, t)) \supseteq \cup rel(sh, t)$ (A.3).

- $x \in f$ and $t \notin f$

  This case is symmetric to the previous one, with x for t and vice versa.

- $x \notin f$ and $t \notin f$

  In this case, $f^o = f \setminus \cup(rel(cl, x) \cup rel(cl, t) \cup rel(ss, x) \cup rel(ss, t))$, but we may or may not have $x \in e$ and $t \in e$, so we have four more cases.

- $x \notin f$, $t \notin f$, $x \notin e$, and $t \notin e$

  We now have $f' = f \setminus \cup(rel(sh, x) \cup rel(sh, t))$. Thus what we have to prove is:

  $$f \setminus \cup(rel(cl, x) \cup rel(cl, t) \cup rel(ss, x) \cup rel(ss, t)) \subseteq e \setminus \cup(rel(sh, x) \cup rel(sh, t))$$

  which holds because $f \subseteq e$ and also:

  $$\cup(rel(cl, x) \cup rel(cl, t) \cup rel(ss, x) \cup rel(ss, t)) \supseteq \cup(rel(sh, x) \cup rel(sh, t))$$

since we have $\cup(rel(cl,t) \cup rel(ss,t)) \supseteq \cup rel(sh,t)$ (A.3) and the same for x: $\cup(rel(cl,x) \cup rel(ss,x)) \supseteq \cup rel(sh,x)$.

- $x \notin f$, $t \notin f$, $x \in e$, and $t \notin e$

  In this case, $f' = f \setminus \cup rel(sh,x)$. The result then follows from the previous case, since $\cup rel(sh,x) \subseteq \cup(rel(sh,x) \cup rel(sh,t))$.

- $x \notin f$, $t \notin f$, $x \notin e$, and $t \in e$

  In this case, $f' = f \setminus \cup rel(sh,t)$. As before, the result follows because $\cup rel(sh,t) \subseteq \cup(rel(sh,x) \cup rel(sh,t))$.

- $x \notin f$, $t \notin f$, $x \in e$, and $t \in e$

  Now, $f' = e$, and the result follows because $f^o \subseteq f$ and $f \subseteq e = f'$.

$\square$

Theorem 6.4.1 Let $Call \in SH^W$, $Prime \in SH^W$, and $g \in Term$, such that the conditions for the *extend* function are satisfied. Let $Call = (cl_1, ss_1)$, $Prime = (cl_2, ss_2)$, $extend^s(Call, g, Prime) = (cl, ss)$, $\Downarrow cl_1 \cup ss_1 \supseteq sh_1$, and $\Downarrow cl_2 \cup ss_2 \supseteq sh_2$ then:

$$\Downarrow cl \cup ss \supseteq extend(sh_1, g, sh_2)$$

Proof. The following two results, proved in (9.7) and (9.11) of [121] (page 240), respectively, will be used. For $c \in CL$ and term t:

$$\Downarrow(irrel(c,t)) = irrel(\Downarrow c, t) \tag{A.11}$$

$$\Downarrow c^* = (\Downarrow cl)^* \tag{A.12}$$

Now, we simplify the (notation of the) definitions of $extend^s$ and $extend$:

$$(cl, ss) \ = \ (\ irrel(cl_1, g) \cup extcl\ ,\ irrel(ss_1, g) \cup extsh \cup clsh \cup shcl\ )$$

$$extcl = \{\ (s \cap c) \cup (s \setminus \hat{g}) \mid s \in cl',\ c \in cl_2\ \}$$

$$extsh = \{\ s \mid s \in rel(ss_1, g)^*,\ (s \cap \hat{g}) \in ss_2\ \}$$

$$clsh = \{\ s \mid s \subseteq c \in cl',\ (s \cap \hat{g}) \in ss_2\ \}$$

$$shcl = \{\ s \mid s \in rel(ss_1, g)^*,\ (s \cap \hat{g}) \subseteq c \in cl_2\ \}$$

with $(cl', ss') = (rel(cl_1, g), rel(ss_1, g))^*$ and $cl' = rel(cl_1, g)^* \cup (rel(cl_1, g)^* \bowtie rel(ss_1, g)^*)$ because of Lemma 3. So that:

$$\Downarrow cl \cup ss = \ \Downarrow(irrel(cl_1, g))\ \cup\ irrel(ss_1, g)\ \cup\ \Downarrow extcl \cup extsh \cup clsh \cup shcl \quad \text{(A.13)}$$

Also, let $extend(sh_1, g, sh_2) = irrel(sh_1, g)\ \cup ext$ with:

$$ext = \{\ s \mid s \in rel(sh_1, g)^*,\ (s \cap \hat{g}) \in sh_2\ \}$$

Take $s \in extend(sh_1, g, sh_2)$. Then, either $s \in irrel(sh_1, g)$ or $s \in ext$ (or both, but this is obviously impossible). If $s \in irrel(sh_1, g)$ then we have that $s \in\ \Downarrow cl \cup ss$, since, from the condition that $\Downarrow cl_1 \cup ss_1 \supseteq sh_1$, and using (A.2), (A.11), and (A.13):

$$irrel(sh_1, g) \subseteq irrel(\Downarrow cl_1, g) \cup irrel(ss_1, g) = \ \Downarrow(irrel(cl_1, g)) \cup irrel(ss_1, g) \subseteq\ \Downarrow cl \cup ss$$

If $s \in ext$ then, by definition, $s \in rel(sh_1, g)^*$ and $(s \cap \hat{g}) \in sh_2$. But from the condition that $\Downarrow cl_1 \cup ss_1 \supseteq sh_1$, using (A.1) it follows that:

$$rel(sh_1, g)^* \subseteq (\ \Downarrow(rel(cl_1, g)) \cup rel(ss_1, g))^*$$

so that $s \in (\ \Downarrow(rel(cl_1, g))\ \cup\ rel(ss_1, g))^*$. Thus, we have three possible cases: $s \in (\ \Downarrow(rel(cl_1, g)))^* = \ \Downarrow(rel(cl_1, g)^*)$ (by (A.12)), $s \in rel(ss_1, g)^*$, or $s = \cup_{i=1}^{m} a_i \cup \cup_{j=1}^{n} b_j$, $a_i \in\ \Downarrow(rel(cl_1, g))$ for all $i = 1, \ldots, m$, $b_j \in rel(ss_1, g)$ for all $j = 1, \ldots, n$.

Also $(s \cap \hat{g}) \in sh_2$, so that from the condition that $\Downarrow cl_2 \cup ss_2 \supseteq sh_2$ then $(s \cap \hat{g}) \in \Downarrow cl_2 \cup ss_2$. Thus, either $(s \cap \hat{g}) \in \Downarrow cl_2$ or $(s \cap \hat{g}) \in ss_2$. Overall, we have six possible cases:

- $s \in rel(ss_1, g)^*$, $(s \cap \hat{g}) \in ss_2$
- $s \in rel(ss_1, g)^*$, $(s \cap \hat{g}) \in \Downarrow cl_2$
- $s \in \Downarrow(rel(cl_1, g)^*)$, $(s \cap \hat{g}) \in ss_2$
- $s \in \Downarrow(rel(cl_1, g)^*)$, $(s \cap \hat{g}) \in \Downarrow cl_2$
- $s = \cup_{i=1}^{m} a_i \cup \cup_{j=1}^{n} b_j$, $(s \cap \hat{g}) \in ss_2$
- $s = \cup_{i=1}^{m} a_i \cup \cup_{j=1}^{n} b_j$, $(s \cap \hat{g}) \in \Downarrow cl_2$

In the first case, we have that $s \in extsh$, and thus, by (A.13), $s \in \Downarrow cl \cup ss$.

In the second case, we have that there is $c \in cl_2$ such that $(s \cap \hat{g}) \in \downarrow c$. Thus, $(s \cap \hat{g}) \subseteq c \in cl_2$, so that $s \in shcl$. Hence, by (A.13), $s \in \Downarrow cl \cup ss$.

In the third case, we have that there is $c \in rel(cl_1, g)^*$ such that $s \in \downarrow c$. Thus, $s \subseteq c \in rel(cl_1, g)^* \subseteq cl'$, so that $s \in clsh$. Hence, by (A.13), $s \in \Downarrow cl \cup ss$.

In the fourth case, as in the third, we have $s \in cl'$. Also, $(s \cap \hat{g}) \in \Downarrow cl_2$. Thus, $(s \cap \hat{g}) \subseteq c \in cl_2$, so that $((s \cap c) \cup (s \setminus \hat{g})) = e \in extcl$. Obviously, $(s \cap \hat{g}) \subseteq s$, so that $(s \cap \hat{g}) \subseteq (s \cap c)$. Therefore, $s = (s \cap \hat{g}) \cup (s \setminus \hat{g}) \subseteq (s \cap c) \cup (s \setminus \hat{g}) = e \in extcl$. And then $s \in \downarrow e \subseteq \Downarrow extcl$. Hence, by (A.13), $s \in \Downarrow cl \cup ss$.

In the fifth case, we have that $a_i \in \Downarrow(rel(cl_1, g))$, so that there are $d_i \in rel(cl_1, g)$ such that $a_i \in \downarrow d_i$. Thus, $a_i \subseteq d_i \in rel(cl_1, g)$, so that $\cup_{i=1}^{m} a_i \subseteq \cup_{i=1}^{m} d_i \in rel(cl_1, g)^*$. We also have that $b_j \in rel(ss_1, g)$, so that $\cup_{j=1}^{n} b_j \in rel(ss_1, g)^*$. Therefore, $s = \cup_{i=1}^{m} a_i \cup \cup_{j=1}^{n} b_j \subseteq (\cup_{i=1}^{m} d_i \cup \cup_{j=1}^{n} b_j) = c \in rel(cl_1, g)^* \bowtie rel(ss_1, g)^* \subseteq cl'$. Thus, $s \subseteq c \in cl'$, so that $s \in clsh$. Hence, by (A.13), $s \in \Downarrow cl \cup ss$.

In the sixth case, as in the fifth, we have $s \in cl'$. Also, $(s \cap \hat{g}) \in \Downarrow cl_2$. Thus, following the same reasoning as in the fourth case, we also have that $s \in \Downarrow cl \cup ss$. □

Theorem 6.4.2 Let $Call \in SHF^W$, $Prime \in SHF^W$, and $g \in Term$, such that the conditions for the *extend* function are satisfied. Let $Call = ((cl_1, sh_1), f_1)$, $Prime = ((cl_2, sh_2), f_2)$, and $extend^{sf}(Call, g, Prime) = ((cl', sh'), f')$. Let also $s_1 = \Downarrow cl_1 \cup sh_1$, $s_2 = \Downarrow cl_2 \cup sh_2$, and $extend^f((s_1, f_1), g, (s_2, f_2)) = (sh, f)$. Then $(\Downarrow cl' \cup sh') \supseteq sh$ and $f' \subseteq f$.

Proof. We now prove that $e \subseteq f$, given that $e_1 \subseteq f_1$, $e_2 \subseteq f_2$, and the rest of conditions of the theorem, in particular $\Downarrow cl \cup ss \supseteq sh$. Remember also that:

$$f = f_2 \cup ff \text{ and } e = e_2 \cup ee$$

$$ff = \{x \mid x \in (f_1 \setminus \hat{g}), ((\cup rel(sh, x)) \cap \hat{g}) \subseteq f_2\}$$

$$ee = \{x \mid x \in (e_1 \setminus \hat{g}),\ ((\cup(rel(ss, x) \cup rel(cl, x))) \cap \hat{g}) \subseteq e_2\}$$

Take $x \in e$. Then, either $x \in e_2$ or $x \in ee$. If $x \in e_2$ then, since $e_2 \subseteq f_2$, $x \in f_2$, so that $x \in f$. Let, then, $x \in ee$.

Now, we have $x \in (e_1 \setminus \hat{g})$ and $((\cup(rel(ss, x) \cup rel(cl, x))) \cap \hat{g}) \subseteq e_2$. Since $e_1 \subseteq f_1$, we have $x \in (f_1 \setminus \hat{g})$. We also have $((\cup rel(sh, x)) \cap \hat{g}) \subseteq f_2$ (see below). Thus, $x \in ff$, so that $x \in f$.

To see that $((\cup(rel(ss, x) \cup rel(cl, x))) \cap \hat{g}) \subseteq e_2$ implies $((\cup rel(sh, x)) \cap \hat{g}) \subseteq f_2$, consider that by using (A.3) $\Downarrow cl \cup ss \supseteq sh$ implies $((\cup rel(sh, x)) \cap \hat{g}) \subseteq ((\cup(rel(ss, x) \cup rel(cl, x))) \cap \hat{g})$, so that:

$$((\cup rel(sh, x)) \cap \hat{g}) \subseteq ((\cup(rel(ss, x) \cup rel(cl, x))) \cap \hat{g}) \subseteq e_2 \subseteq f_2$$

□

Theorem 7.4.1 A polynomial time algorithm for computing negative cross-union, $\overline{\boxtimes}$, implies $\mathcal{P}$=$\mathcal{NP}$.

Proof. To show that negative cross-union, $\boxtimes$, is $\mathcal{NP}$-Complete we first restate the definition of Non-Empty Self Recognition ($NESR$) shown to be $\mathcal{NP}$-Complete in [41]. Then, we use $NESR$ to show that there is no polynomial time algorithm for computing negative cross-union unless $\mathcal{P} = \mathcal{NP}$.

(Non-Empty Self Recognition, $NESR$**).**
INPUT: A negative set *tnsh* of length l strings over the alphabet $\{0, 1, *\}$.
QUESTION: Does *tnsh* represent an empty positive set *bsh*? In other words, does there exists a string in $\{0, 1\}^l$ not matched in *tnsh*?

The following is a proof for Theorem 7.4.1:

Given a negative set *tnsh* of length l, assume a polynomial time algorithm $\mathcal{M}$ that takes as input negative sets $tnsh_1$ and $tnsh_2$ and outputs $tnsh' = tnsh_1 \boxtimes tnsh_2$, where $tnsh'$ represents the result of the positive cross-union of the two positive sets represented by $tnsh_1$ and $tnsh_2$.

We construct a polynomial time algorithm for $NESR$: given any instance of $NESR$ with input *tnsh*. First, generate a positive set *sh* with two strings s_1 and s_2 of length l each having alternating 1's and 0's, e.g., if $l = 4$, then $sh = \{0101, 1010\}$. Convert *sh* to its negative set representation, *nsh*, using a polynomial time algorithm, i.e., letting $k = log_2(l)$ or the Prefix algorithm, see [41]. Verify that s_1 and s_2 appear in *tnsh*: if either one is missing from *tnsh*, then answer "No" (*tnsh* is not empty, at a minimum it represents the missing string). Otherwise, both s_1 and s_2 appear in *tnsh*, but there may be some other string(s) missing from *tnsh* (*tnsh* is not empty). Let $\mathcal{M}$ compute $tnsh' = tnsh \boxtimes nsh$. Now, check if both s_1 and s_2 appear in $tnsh'$: if both are missing from $tnsh'$, then answer "Yes" (*tnsh* is empty); otherwise, answer "No".

Note that if *tnsh* represented an empty positive set, then its *negative cross-union* with another set *nsh* will yield a representation of the same set *nsh*. In other words, if $tnsh$ is empty and since $s1$ and $s2$ were missing from nsh, then $s1$ and $s2$ will also be missing from the result $tnsh'$. On the other hand, if $tnsh$ is *not* empty (represents some string(s), other than $s1$ and $s2$, in the positive), then negative cross-union (ternary OR operation) with one of the two strings will produce a different string to s_1 or s_2 resulting in either s_1 or s_2 appearing in $tnsh'$. Thus, $\mathcal{M}$ can be used to solve $NESR$ efficiently. Since $NESR$ is $\mathcal{NP}$-Complete, then $\mathcal{P}$=$\mathcal{NP}$. $\square$

References

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

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

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

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

[5] Gianluca Amato and Francesca Scozzari. Optimality in goal-dependent analysis of sharing. Technical Report TR-05-06, Dipartimento di Informatica, Università di Pisa, 2005.

[6] K.R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Model and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[7] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.

[8] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman,

J.-L. Lanet, and T. Muntean, editors, *Proc. of Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 1–27. Springer, 2005.

[9] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In *TPHOLs2004*, volume 3223 of *LNCS*, pages 34–49, Heidelberg, September 2004. Springer Verlag.

[10] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *Proc. of OOPSLA'96, SIGPLAN Notices*, 31(10):324–341, October 1996.

[11] R. Bagnara, R. Gori, P. M. Hill, and E. Zaffanella. Finite-tree analysis for constraint logic-based languages. *Information and Computation*, 193(2):84–116, 2004.

[12] R. Bagnara, A. Pescetti, A. Zaccagnini, E. Zaffanella, and T. Zolo. Purrs: The Parma University's Recurrence Relation Solver. `http://www.cs.unipr.it/purrs`.

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

[14] I. Bate, G. Bernat, and P. Puschner. Java Virtual-Machine Support for Portable Worst-Case Execution-Time Analysis. In *5th IEEE International Symposium on Object-oriented Real-time Distributed Computing, Washington, DC, USA*, Apr. 2002.

[15] R. Benzinger. Automated Higher-Order Complexity Analysis. *Theor. Comput. Sci.*, 318(1-2), 2004.

[16] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, 2007.

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

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

[19] M. Bruynooghe, M. Codish, and A. Mulkers. Abstract unification for a composite domain deriving sharing and freeness properties of program variables. In F.S. de Boer and M. Gabbrielli, editors, *Verification and Analysis of Logic Languages*, pages 213–230, 1994.

[20] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[21] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Reference Manual (v1.10). The ciao system documentation series–TR, School of Computer Science, Technical University of Madrid (UPM), June 2004. System and on-line version of the manual available at `http://www.ciaohome.org`.

[22] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[23] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[24] A. Casas, M. Carro, and M. Hermenegildo. Towards a High-Level Implementation of Execution Primitives for Non-restricted, Independent And-parallelism. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 230–247. Springer-Verlag, January 2008.

[25] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming (ESOP)*, number 3444 in LNCS, pages 311–325. Springer-Verlag, 2005.

[26] Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI'05*, number 3385 in LNCS, pages 147–163. Srpinger, 2005.

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

[28] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains.

In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

[29] Michael Codish, Dennis Dams, Gilberto Filé, and Maurice Bruynooghe. On the design of a correct freeness analysis for logic programs. *The Journal of Logic Programming*, 28(3):181–206, 1996.

[30] Michael Codish, Harald Søndergaard, and Peter J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Transactions on Programming Languages and Systems*, 21(5):948–976, 1999.

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

[32] Stephen-John Craig and Michael Leuschel. Self-Tuning Resource Aware Specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[33] K. Crary and S. Weirich. Resource bound certification. In *POPL'00*. ACM Press, 2000.

[34] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *PLDI*, pages 57–68, 2002.

[35] M. García de la Banda. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, September 1994.

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

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

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

[39] S. W. Dietrich. Extension Tables: Memo Relations in Logic Programming. In *Fourth IEEE Symposium on Logic Programming*, pages 264–272, September 1987.

[40] Jochen Eisinger, Ilia Polian, Bernd Becker, Alexander Metzner, Stephan Thesing, and Reinhard Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proceedings of IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 15–20. IEEE Computer Society, April 2006.

[41] F. Esponda, E. S. Ackley, S. Forrest, and P. Helman. On-line negative databases (with experimental results). *International Journal of Unconventional Computing*, 1(3):201–220, 2005.

[42] F. Esponda, E. D. Trias, E. S. Ackley, and S. Forrest. A relational algebra for negative databases. Technical Report TR-CS-2007-18, University of New Mexico, 2007.

[43] Christian Fecht. An efficient and precise sharing domain for logic programs. In Herbert Kuchen and S. Doaitse Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 469–470. Springer, 1996.

[44] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In R. Cousot, editor, *Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, Paris, France, January 2005. Springer-Verlag.

[45] G. Gómez and Y. A. Liu. Automatic Time-Bound Analysis for a Higher-Order Language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM).* ACM Press, 2002.

[46] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition).* Addison-Wesley Professional, 2005.

[47] Bernd Grobauer. Cost recurrences for DML programs. In *International Conference on Functional Programming*, pages 253–264, 2001.

[48] Kim S. Henriksen. *A Logic Programming Based Approach to Applying Abstract Interpretation to Embedded Software.* PhD thesis, Roskilde University, Roskilde, Denmark, October 2007. Research Report #117.

[49] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[50] M. Hermenegildo. A Documentation Generator for (C)LP Systems. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 1345–1361. Springer-Verlag, July 2000.

[51] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction Carrying Code and Resource-Awareness. In *PPDP*. ACM Press, 2005.

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

[53] M. Hermenegildo and The Ciao Development Team. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In *ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages MPOOL 2007*, July 2007.

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

[55] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[56] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 13(4):349–367, August 1992.

[57] P. M. Hill, E. Zaffanella, and R. Bagnara. A correct, precise and efficient integration of set-sharing, freeness and linearity for the analysis of finite and rational tree languages. *Theory and Practice of Logic Programming*, 4(3):289–323, 2004.

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

[59] Shin ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In *DAC*, pages 272–277, 1993.

[60] Atsushi Igarashi and Naoki Kobayashi. Resource usage analysis. In *Symposium on Principles of Programming Languages*, pages 331–342, 2002.

[61] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[62] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

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

[64] A. King and P. Soper. Depth-k Sharing and Freeness. In *International Conference on Logic Programming*. MIT Press, June 1994.

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

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

[67] Sébastien Lafond and Johan Lilius. Energy consumption analysis for two embedded java virtual machines. *J. Syst. Archit.*, 53(5-6):328–337, 2007.

[68] A. Langen. *Advanced techniques for approximating variable aliasing in Logic Programs.* PhD thesis, Computer Science Dept., University of Southern California, 1990.

[69] D. Le Metayer. ACE: An Automatic Complexity Evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[70] Xavier Leroy. Java bytecode verification: An overview. In *CAV'01*, number 2102 in LNCS, pages 265–285. Springer, 2001.

[71] Michael Leuschel. *Advanced Techniques for Logic Program Specialisation.* PhD thesis, K.U. Leuven, May 1997.

[72] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS*, number 1824 in LNCS, pages 280–301. Springer, 2000.

[73] Xuan Li, Andy King, and Lunjin Lu. Collapsing Closures. In Sandro Etalle and Mirek Truszczynski, editors, *22nd. Int'l. Conference on Logic Programming*, volume 4079 of *LNCS*, pages 148–162. Springer-Verlag, August 2006. Also see http://www.springer.de/comp/lncs/index.html.

[74] Xuan Li, Andy King, and Lunjin Lu. Lazy Set-Sharing Analysis. In Philip Wadler and Masimi Hagiya, editors, *8th. Int'l. Symp. on Functional and Logic Programming*, LNCS. Springer-Verlag, April 2006.

[75] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[76] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[77] Francesco Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In *VMCAI'07*, number 4349 in LNCS. Springer, Jan 2007.

[78] Francesco Logozzo and Agostino Cortesi. Abstract interpretation and object-oriented languages: quo vadis? In *Proceedings of the 1st International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL'05)*, Electronic Notes in Theoretical Computer Science. Elsevier Science, January 2005.

[79] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[80] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.

[81] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. *Information Processing*, pages 601–606, April 1989.

[82] David A. McAllester. On the complexity analysis of static analyses. In *Static Analysis Symposium*, pages 312–329, 1999.

[83] M. Méndez-Lojo and M. Hermenegildo. Precise Set Sharing Analysis for Java-style Programs. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

[84] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.

[85] M. Méndez-Lojo, J. Navas, and M. Hermenegildo. An Efficient, Parametric Fixpoint Algorithm for Analysis of Java Bytecode. In *ETAPS Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'07)*, Electronic Notes in Theoretical Computer Science. Elsevier - North Holland, March 2007.

[86] Donald R. Morrison. Patricia: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[87] A. Mulkers, W. Simoens, G. Janssens, and M. Bruynooghe. On the Practicality of Abstract Equation Systems. In *International Conference on Logic Programming*. MIT Press, June 1995.

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

[89] K. Muthukumar and M. Hermenegildo. 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, April 1990.

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

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

[92] Kalyan Muthukumar. *Compile-time Algorithms for Efficient Parallel Implementation of Logic Programs.* PhD thesis, University of Texas at Austin, August 1991.

[93] J. Navas, F. Bueno, and M. Hermenegildo. Efficient top-down set-sharing analysis using cliques. In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[94] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. An Efficient, Context and Path Sensitive Analysis Framework for Java Programs. In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

[95] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *International Conference on Logic Programming (ICLP)*, volume 4670 of *LNCS*, pages 348–363. Springer-Verlag, September 2007.

[96] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Automatic complexity analysis. In *European Symposium on Programming*, pages 243–261, 2002.

[97] Isabelle Pollet. *Towards a generic framework for the abstract interpretation of Java.* PhD thesis, Catholic University of Louvain, 2004. Dept. of Computer Science.

[98] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[99] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[100] G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 261–271. ACM Press, July 2006.

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

[102] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *The Journal of Logic Programming*, 11(3 & 4):189–216, October/November 1991.

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

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

[105] Erik Ruf. Effective synchronization removal for java. *PLDI'00, SIGPLAN Notices*, 35(5):208–218, 2000.

[106] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.

[107] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[108] D. Sands. A naïve time analysis and its theory of cost equivalence. *J. Log. Comput.*, 5(4), 1995.

[109] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[110] F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005.

[111] Lothar Thiele and Reinhard Wilhelm. Design for time-predictability. In *Perspectives Workshop: Design of Systems with Predictable Behaviour, 16.-19. November 2003*, volume 03471 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2004.

[112] E. Trias, J. Navas, E. S. Ackley, S. Forrest, and M. Hermenegildo. Negative Ternary Set-Sharing. In *International Conference on Logic Programming, ICLP*, LNCS, Udine (Italy), December 2008. Springer-Verlag.

[113] A. Turing. On computable numbers with an application to the entscheidungs problem. *Proc. London Mathematical Society*, 2(42):230–265, 1936.

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

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

[116] P. Vasconcelos and K. Hammond. 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, September 2003.

[117] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

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

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

[120] E. Zaffanella, R. Bagnara, and P. M. Hill. Widening Sharing. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 414–432. Springer-Verlag, Berlin, 1999.

[121] Enea Zaffanella. *Correctness, Precision and Efficiency in the Sharing Analysis of Real Logic Languages*. PhD thesis, School of Computing, University of Leeds, Leeds, U.K., 2001.