Constraint Solver Techniques for Implementing Precise and Scalable Static Program Analysis

Zhang, Ye

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Zhang, Y. (2009). Constraint Solver Techniques for Implementing Precise and Scalable Static Program Analysis. Kgs. Lyngby, Denmark: Technical University of Denmark (DTU). IMM-PHD-2008-211

Constraint Solver Techniques for Implementing Precise and Scalable Static Program Analysis

Ye Zhang

Kongens Lyngby 2008
IMM-PHD-2008-211

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Summary

As people rely on all kinds of software systems for almost every aspect of their lives, how to ensure the reliability of software is becoming more and more important. Program analysis, therefore, becomes more and more important for the software development process. Static program analysis helps developers to build reliable software systems more quickly and with fewer bugs or security defects. While designing and implementing a program analysis remains a hard work, making it both scalable and precise is even more challenging. In this dissertation, we show that with a general inclusion constraint solver using unification we could make a program analysis easier to design and implement, much more scalable, and still as precise as expected.

We present an inclusion constraint language with the explicit equality constructs for specifying program analysis problems, and a parameterized framework for tuning a constraint system. Implementing an analysis is simplified as generating a set of constraints to be complied with. The equality constraints specifies equivalent analysis variables and thereby could be taken advantage of by a constraint solver to reduce a problem space and improve performance. We show a good balance between performance and precision could be achieved with the framework.

We also introduce off-line optimizations for a general constraint solver. The optimizations automatically efficiently detect (potential) equivalent classes. With our case studies on a C pointer analysis, and two data flow analyses for C language, we demonstrate a large amount of equivalences could be detected by off-line analyses, and they could then be used by a constraint solver to significantly improve the scalability of an analysis without sacrificing any precision.

Resumé

Efterhånden som samfundet i stigende grad bliver afhængigt af softwaresystemer, bliver det stadigt vigtigere at sikre deres pålidelighed. Som følge deraf bliver programanalyse et stadigt vigtigere element i softwareudviklingsprocessen, fordi det hjælper udviklere til at bygge pålidelige softwaresystemer hurtigere og med færre fejl eller sikkerhedsdefekter. Det er en udfordring at designe og implementere nyttige programanalyser og ikke mindst, når de både skal være præcise og skalerbare. I denne afhandling viser vi, hvorledes en generel løsningspakke til programanalyser, dels kan udvikles og dels kan bruges til opnå de beskrevne mål.

Vi præsenterer et metasprog til programanalyse, som både tillader inklusion af information mellem analyse variable og egentlig unifikation af disse, samt en parametriseret ramme for at tilpasse analysernes opførsel. Man kan derfor foretage en programanalyse ved blot at generere et passende udtryk i metasproget, der efterfølgende løses af vores løsningspakke, hvis konstruktion beskrives i detaljer. Brugen af unifikation muliggør en kraftig reduktion i løsningspakkens pladsforbrug og dermed også dens tidsforbrug. Vore studier viser at en god balance mellem præcision of effektivitet kan opnås ved at skrue på disse parametre.

For yderligere at effektivesere programanalyserne præsenterer vi en række såkaldte off-line optimeringer. De tager et udtryk i metasproget og ændrer mellem inklusion og unifikation således at god effektivitet opnås samtidig med en høj grad af præcision. Vi bruger dem på case studies som pointer analyse i C og to klassiske programanalyser for C og viser at de fungerer godt i praksis. Det peger frem mod automatisk brug i løsningspakken for derigennem at forbedre dens skalerbarhedsegenskaber yderligere.

Preface

This thesis was completed at the department of Informatics and Mathematical Modelling, the Technical University of Denmark, in partial fulfillment of the requirements for acquiring the Ph.D. degree in Computer Science. The Ph.D study has been carried out under the supervision of Professor Flemming Nielson in the period of September 2005 to November 2008. The study was supported by the Danish Strategic Research Council (proposed number 2059-03-0011).

Acknowledgements

First of all, I must thank my wife Shanshan, and my parents for their consistent support and patience.

I am grateful to my supervisor Flemming Nielson for teaching me so much, for guiding this work, and for having the courage to let me develop my own directions. He has supported me throughout with excellent advice, comments, and suggestions.

I would like to thank each member of LBT group: Hanne Riis Nielson, Sebastian Nanz, Henrik Pilegaard, Christian Probst, Fan Yang, Christoffer Rosenkilde Nielsen, Han Gao, Ender Yuksel, Nataliya Skrypnyuk, and Matthieu Stéphane Benoit Queva, for creating an inspiring working environment.

A special thanks goes to Torben Amtoft for taking care of me during my visit to Kansas, Manhattan, and for many fruitful discussions and comments on my work.

I would also like to thank the evaluation committee Paul August Fischer, Thomas Jensen, and Chris Hankin, for their helpful comments on this thesis.

Last but not least, I would like to give thanks to all the people at DTU Informatics, who have created a friendly working environment during my Ph.D study.

Lyngby, November 2008

Zhang, Ye

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

With the fast development of IT technology, all kinds of programs are running almost everywhere in our society, from offices to homes, from schools to hospitals, etc. While they improve the productivity of the whole society and save a large amount of human resources in doing the same work, the demand of improving the quality of software is increasing and even becomes critical. For example, recent NASA mission failures, such as Mars Polar Lander and Mars Orbiter, show the importance of having an efficient verification and validation process for such systems. A minor error of programs on medical devices could even be fatal to patients. To tackle the problem, much effort has been invested in doing test manually in industry. It is tedious work and cannot guarantee the completeness of test. This is not only because time and human resources are limited but also because finding all the run-time errors or, more general, any kind of violation of a specification is undecidable. Under most cases only part of the execution scenarios can be covered in manually conducted test. That is why automatic techniques become so attractive.

Static program analysis [NNH99] is the process of automatically analyzing the behavior of computer programs and has been widely applied on the fields of compilers and software engineering tools. It is often conducted in two stages. First a theoretically well-founded analysis is designed and expressed using a collection of constraints. Second the constraints are solved by some constraint solver. This strategy separates analysis specification from implementation and also allows program analysis designers to share the insights and efforts in solver technology: optimizations to the solver can be applied to all analyses using it. Furthermore, compared to manually implemented analysis, fewer errors are introduced by automatically deriving the implementation from the set of constraints generated.

Although much effort has been devoted to proving the correctness of analyses, this is not sufficient for getting a useful analysis. To demonstrate utility, analysis designers need to prototype, test their analysis ideas, and realistically judge the cost/performance trade-offs of different design considerations. This dissertation describes a parameterized framework supporting both set inclusion and unification over analysis variables. With the framework, users can implement efficient or precise analyses, or even both. The intuition of using unification is that analysis variables can be collapsed into one representative and thereby reduces the problem space.

Substituting unification for set-inclusion is based on two insights. First, the analysis result of unification is sound with respect to that of set-inclusion. Second, unification can be solved in almost linear time and reduces memory consumption. Given the knowledge from the client, the parameterized framework in this thesis enables analysis designers to test their ideas and reach a good balance between performance and precision.

In this dissertation the universe of the constraint solver consists of finite atomic values as for Datalog solvers, e.g. the Succinct Solver [NSN02], XSB Prolog system [SSW94]. Many program analyses use a declarative language, such as Datalog. However implementations using general Datalog solvers are often slower than traditional implementations and as a result do not scale to large programs. Emphasizing the use of unification, and developing techniques of making use of it, shows light on cracking the scalability bottleneck.

Precision and performance are often considered as a trade-off relation because a more precise result often implies higher cost. For example, set inclusion is more precise than unification because it maintains the direction of the flow of information. In return it takes cubic time in worst case to solve when a graph problem is concerned. In fact precise analyses often show unscalable to large programs. Although using equality constraints may lead to a loss in precision, a heuristic study of this thesis shows that often there is a partitioning into equivalence classes of analysis variables and they can be unified without losing precision. In order to make program analyses both precise and scalable, this thesis explores two approaches for detecting equivalence between analysis variables, in which case doing unification over these variables will not cause any loss in precision. The first is to review analysis specifications and detect where unification can replace set-inclusion without causing any extra false alarms. As shown in this thesis, the process of the detection could be quite tricky. However the effort pays off in better understanding under which cases a false alarm may be introduced by unification. As a result, analysis designers gain a good control in the level of precision when they tune a system to reach a good balance between precision and performance.

The second approach is to develop techniques for automatically detecting as many equivalence classes of analysis variables as possible. Modifying analysis specifications can be quite costly in terms of both the time taken and the level of skills desired from the person who needs to perform it. To make the solver a handy engine not only for analysis designers but for all users who want to do precise as well as high performance analysis with the help of unification, this thesis also provides an off-line technique to detect equivalence analysis variables automatically. The technique is called off-line since it is performed before the execution of solver. The main idea of the technique is to study the dependency relation between the data fields of analysis variables given by a constraint program and identify (potential) equality relations. Since off-line analyses are conducted on the constraints instead of program analyses, the use of the existing off-line analysis does not have to be restricted to a specific analysis and could be reused on a collection of program analyses that share the same class of constraints.

In summary, this dissertation makes a contribution in the area of inclusion constraint solver technology for implementing high performance or(and) precise program analyses. We shall mainly study how to take advantage of unification to make high-utility program analyses.

1.1 Overview of the Dissertation

The present dissertation consists of seven chapters, of which this introductory chapter is the first.

Chapter 2 presents the overall strategy for implementing program analyses using inclusion constraint solver and explains the advantages of using such a strategy. Some background knowledge about ordered theory and the Flow Logic[NN97, NN98, NNH99, NN02] is introduced in anticipation of the development of the following chapters.

Chapter 3 presents the basic inclusion constraint language and a parameterized framework to tune a constraint program. In order to give the meaning of the language, two versions of semantics are specified: the first is from the analysis designer's point of view, the second is from the solver designer's point of view. The properties of the two semantics are studied and related each other. Especially, based on the second semantics, a series of theoretical development provides a foundation for the correctness of using unification over analysis variables (collapsing analysis variables) for program analyses.

Chapter 4 specifies the design of the solver algorithm and conducts an experimental study on a Reaching Definitions Analysis. The important properties of the algorithm, such as the correctness, termination property, and asymptotic complexity, are shown. For the experimental study, we conduct a heuristic study on where to use unifications and the effect of using them. We measure the performance in terms of time and space consumption before and after using unification on a set of well-designed benchmarks. A comparison study between our solver and the Succinct Solver is also performed.

Chapter 5 describes an extended inclusion constraint language for a C pointer analysis. We presents the perspectives of pointer analyses, motivate the extension to the basic inclusion constraints, and finally describe in detail the formulation of real C programs using inclusion constraints.

Chapter 6 introduces two off-line optimizations for the constraint solver and the strategy of reusing the optimizations for more program analyses. We study the effect of applying the optimizations on a C pointer analysis, a reaching definitions analysis, and a live variable analysis.

Chapter 7 concludes our work by a discussion of related work and re-states our major contribution in this dissertation.

Chapter 2

Setting the Scene

Software programs are typically written in general-purpose languages, e.g. the C programming language, or sometimes in process calculi, e.g. CSP [Low95], CCS [FG95], ambients [CG00, BCC01, ZN06], Klaim [BBN$^+$03, NFP98, NFP00], etc, and they are Turing complete language. Semantically analyzing certain property of such a program is, in general, undecidable, i.e., not solvable with limited time and(or) space, according to Rice's Theorem [Ric53].

Despite the undecidable nature of the problem, several techniques have been introduced in order to automate the analysis. Clarke et al. uses the model checking of temporal logic formulae to exhaustively simulate the execution of a program [CE81]. The approach is efficient for finding flaws in applications whose state space is moderate size. However, it becomes intractable for large state spaces and undecidable for infinite state spaces. As a result, the model checking can not guarantee the termination of an analysis and the absence of the flaw.

The basic theme of static analysis is that the analysis can remain computable by providing a little larger answer than the exact one [NNH99]. That is, static analysis approximates the behavior of a software by taking proper abstraction, which shows respect to the semantics. The Figure 2.1 illustrates the principle. Accordingly static analysis always terminates and guarantees the flawlessness of a system if no violation is reported.

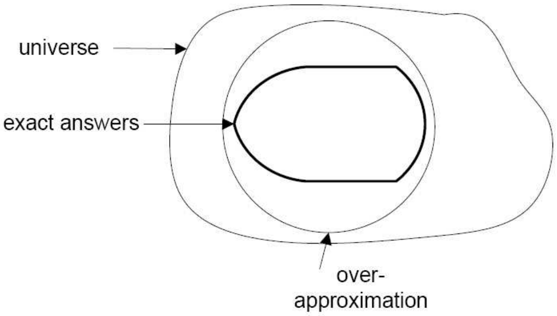


Figure 2.1: Approximate the behavior of a program using static analysis.

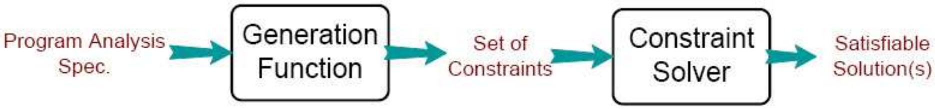


Figure 2.2: Framework of implementing constraint-based program analysis.

Because the nature of static analysis, false positives may appear and have to be manually identified by the users of an analysis tool. Improving the precision of an analysis usually reduces performance. On the other hand, being decidable does not imply being practical: the cubic time complexity of Andersen's pointer analysis [And94] is not scalable in analyzing real C programs. Therefore obtaining precise and high-performance analysis remains a challenge for analysis designers.

The constraint solver technique introduced in this dissertation aims at achieving both precise and high-performance program analysis. The fact that many program analyses can be expressed as a collection of constraints, the approach is known as *constraint based analysis*, allows the development of general constraint solver technique. In return, a general constraint solver simplifies the implementation of program analyses that generate a set of constraints. Figure 2.2 illustrates the whole process: an analysis specification is designed to generate constraints from program code, and these constraints are then solved by a constraint solver; finally the solver outputs a solution for the constraints. This process is demonstrated by all the implementations of the example analyses in this dissertation. Although in Chapter 6 we shall extend the process a little bit, the basic principle remains the same.

2.1 Why constraints

The advantageous of using constraints to specify analyses can be summarized as the following:

- First, the design of program analyses is separated from their implementations. Analysis designers can therefore focus on generating proper constraints. The implementation of an analysis becomes a separate activity that often involves other formalisms and tools.
- Second, inclusion constraints is a natural approach for programmers to specify program analysis. The meaning of the constraint constructs is often straightforward and intuitive, and thus understandable to the average programmer.
- Third, the use of constraints allows analysis designers to choose formalisms or tools for solving the constraints generated. They can choose to either develop their own solver or select one of available solvers.
- Fourth, constraint-based approach also provides a common criteria for comparing between constraint solvers. This is often quite important for acquiring insights about how and why one solver is better than another. Solver designers can accordingly enhance existing algorithms or develop new techniques.
- Fifth, automatically deriving the implementation from an constraint specification minimizes the number of errors that may introduce in the implementation phase.
- Sixth, the optimization of an analysis can be conveniently conducted from analysis designers' side. From analysis designers' side, a good balance between performance and precision can be expected by manually tuning the constraints generated. To improve performance, analysis designers may consider to use fast-solved constraints, e.g. unification on equality constraints, instead of expensive ones, e.g. set-inclusion constraints. As a result, a loss in imprecision may happen by adopting fast-solved constraints. However, one should maintain precision as high as possible since too many false positives often make an analysis tool hard to use in practice. Concerning where to prioritize performance over precision or the opposite, analysis designers often have a better view than solver designers.

  The usability of an analysis is usually required for analyzing real software systems and thus how to obtain a high-performance implementation plays an important role in analysis design. In fact, performance has higher priority than precision before an analysis becomes scalable to a software system. This is because no useful approximation is available at all if an analysis is not scalable.

In this dissertation, the constraint language is designed to support a parameterized framework. Analysis designers can tune the analysis by simply choosing the kinds of constraints to be generated in order to reach a good balance between performance and precision. With the framework, analysis designers can actively involve into the activity of making a better analysis in terms of both performance and precision.

- Seventh, an optimization of an analysis can also be conducted from the solver designers' side. From solver designers' point of view, algorithms can be developed to automatically optimize the given constraints, e.g. detecting equivalent analysis variables and collapsing them. As an extra benefit, these algorithms is reusable in general because the optimization is conducted on constraints, not on a specific analysis specification.
- Eighth, the use of constraints allows a natural way to express equality relation over program analysis variables and thereby enables a constraint solver to take advantage of unification technique to improve the usability of a constraint solver.

2.2 Theoretical Preliminaries

Order and lattice theory are important in program analysis. In this subsection, we briefly review the most important order and lattice notations in preparation for the presentation of the following sections. For a more detailed treatment refer to [NNH99], and for a comprehensive introduction refer to [DP02].

Definition 2.1 Partial Order A partial order on a set P is a binary relation $\sqsubseteq$ on P such that $\sqsubseteq$ is

- reflexive i.e. $\forall x \in P : x \sqsubseteq x$
- antisymmetric i.e. $\forall x, y \in P : (x \sqsubseteq y \wedge y \sqsubseteq x) \Rightarrow x = y$
- transitive i.e. $\forall x, y, z \in P : (x \sqsubseteq y \wedge y \sqsubseteq z) \Rightarrow (x \sqsubseteq z)$

Notation 2.2 *We shall write* $(P, \sqsubseteq)$ *for a set* P *equipped with a partial order* $\sqsubseteq$*.*

Definition 2.3 Preorder A preorder on a set P is a binary relation $\preceq$ on P such that $\preceq$ is

- reflexive i.e. $\forall x \in P : x \preceq x$
- transitive i.e. $\forall x, y, z \in P : (x \preceq y \wedge y \preceq z) \Rightarrow (x \preceq z)$

(but not necessarily antisymmetric)

Notation 2.4 *We write* $(P, \preceq)$ *for a set* P *equipped with a partial order* $\preceq$*.*

Theorem 2.5 *For a preorder* $\preceq$*,* $x \equiv y \overset{\text{def}}{=} (x \preceq y) \wedge (y \preceq x)$ *is a equivalence relation, i.e. reflexive, transitive and symmetric.*

Proof. See, e.g., [DP02]. □

If there is an element $x \in P$ such that $\forall y \in P : x \sqsubseteq y$, then the element x is called the least element of P and is denoted $\bot$. Analogously, the greatest element of P is an element $x \in P$ such that $\forall y \in P : y \sqsubseteq x$ and is denoted $\top$. Generalising this leads to the definition of upper bounds:

Definition 2.6 Upper Bound Let $(P, \sqsubseteq)$ be a partial order and let $S \subseteq P$. The element $u \in P$ is an upper bound of S iff

$$\forall x \in S : x \sqsubseteq u$$

Definition 2.7 Least Upper Bound Let $(P, \sqsubseteq)$ be a partial order and let $S \subseteq P$. The element u is a least upper bound (lub) of S iff

- u is an upper bound of S, and
- for every upper bound u' of S, $u \sqsubseteq u'$

Notation 2.8 *We denote a lub of* S *in* P *by* $\bigsqcup S$ *whenever it exists. The binary least upper bound of* $u, u' \in P$ *is written* $u \sqcup u'$*.*

A lower bound and a greatest lower bound are analogously defined by:

Definition 2.9 Lower Bound Let $(P, \sqsubseteq)$ be a partial order and let $S \subseteq P$. The element $l \in P$ is an lower bound of S iff

$$\forall x \in S : l \sqsubseteq x$$

Definition 2.10 Greatest Lower Bound Let $(P, \sqsubseteq)$ be a partial order and let $S \subseteq P$. The element l is a greatest lower bound (glb) of S iff

- l is an lower bound of S, and
- for every lower bound l' of S, $l' \sqsubseteq l$

Notation 2.11 *We denote a glb of S in P by $\bigsqcap S$ whenever it exists. The binary greatest lower bound of $l, l' \in P$ is written $l \sqcap l'$.*

Definition 2.12 Lattice Let $(P, \sqsubseteq)$ be a non-empty partially ordered set. If $x \sqcup y$ and $x \sqcap y$ exist for all $x, y \in P$, then P is called a lattice.

Definition 2.13 Complete Lattice Let $(P, \sqsubseteq)$ be a non-empty partially ordered set. If $\bigsqcup S$ and $\bigsqcap S$ exist for all $S \subseteq P$, then P is called a complete lattice.

Note that for a complete lattice $(P, \sqsubseteq)$, $\bot = \bigsqcup \emptyset = \bigsqcap P$ and $\top = \bigsqcap \emptyset = \bigsqcup P$.

Lemma 2.14 *For a partially ordered set $P = (P, \sqsubseteq)$ the statements*

(i) P is a complete lattice,

(ii) every subset of P has a least upper bound, and

(iii) every subset of P has a greatest lower bound

are equivalent.

Proof. See, e.g., [NNH99] □

Proposition 2.15 Function Space *For a set S and a complete lattice $(P, \sqsubseteq)$, the function space $(S \to P, \sqsubseteq_{S \to P})$ is a complete lattice in which the pointwise order is defined by*

$$f_1 \sqsubseteq f_2 \text{ iff } \forall e \in S : f_1(e) \sqsubseteq_{S \to P} (e)$$

Proof. See, e.g., [NNH99]. □

Definition 2.16 Moore family For a complete lattice $(P, \sqsubseteq)$, a Moore family is $M \subseteq P$, such that

$$\forall M' \subseteq M : \bigsqcap M' \in M$$

i.e. the set M is closed under greatest lower bound.

2.3 Flow Logic

Flow Logic is a specification oriented framework for constraint based static analysis [NN97, NN98, NNH99, NN02]. In anticipation of the following sections, the most important principles and notions of the Flow Logic framework is reviewed below.

The specification of an analysis in Flow Logic declares a set of constraints that analysis estimation must satisfy in order to be an acceptable estimate of a software program. Thus a Flow Logic specification focuses on not how the analysis is computed but what the analysis does. Because of the constraint based nature, a specification and its implementation give rise to be independent. Analysis designers can therefore concentrate on specifying the analysis and not worry about its implementation at the same time.

Traditionally, Flow Logic is only used for specifying an analysis. However, once an analysis specification is available, analysis designers should also feel free to adjust the constraints generated for achieving a good tradeoff between performance and precision in the precondition of the correctness of an analysis. This is one of benefits acquired from the use of constraint based approach as discussed in the earlier section.

A Flow Logic specification consists of three ingredients as the following.

Analysis Domain. The domain of a Flow Logic specification is the universe of discourse for analysis estimates. Usually the universe is given by a complete lattice L.

Acceptability Judgement. An acceptability judgement relates analysis estimates $\mathcal{A} \in L$ to programs $P \in \mathbf{Prog}$. It has the form

$$\mathcal{A} \models P$$

where the functionality of the relation '$\models$' is

$$\models : (L \times \mathbf{Prog}) \rightarrow \{true, false\}$$

Therefore, intuitively a judgement holds when $\mathcal{A}$ constitutes an acceptable analysis estimate for program P.

Defining Clauses. A judgement is defined by a set of clauses which is typically specified for each syntactic construct φ of a programming language and they has the form

$$\mathcal{A} \models \varphi \text{ iff (some logic formulas } \mathcal{F} \textbf{ and } \mathcal{A} \models \varphi')$$

Here $\mathcal{F}$ states the constraints that analysis estimate $\mathcal{A}$ should follow. If φ' is always a strict sub-programm of φ, then the Flow Logic specification is inductively defined. This kind of specifications are called *syntax directed* or *compositional* since the analysis of a process is only relevant to the analysis of its part. Otherwise, a specification is called *abstract*.

In this dissertation, every example analysis specified in Flow Logic is syntax directed. For more detailed information about abstract specification, see [NNH99].

2.4 Concluding Remarks

In this chapter we presented the overall strategy of using a constraint solver to implement program analysis (which should be constraint-based). We discussed the advantages of using a constraint formulation for implementing a program analysis. In the later chapters, we shall demonstrate these advantages with the techniques developed and all the working examples.

In preparation of the theoretic development in later chapters, we reviewed some important definitions and theorems in the field of ordered theory. They not only form a basement for the proof of some theorems, but also shows to be inspiring in some our definitions presented later, e.g. the complete prelattice and Moore family for complete prelattice.

Finally we introduced the Flow Logic framework that is widely used in specifying constraint-based program analyses. The use of this framework shall be demonstrated in our case study. In fact, the satisfaction relation $\models$ is also used for specifying semantics of the inclusion constraint language and the semantics of the ALFP logic.

Chapter 3

Basic Inclusion Constraint Language

In this chapter we present an inclusion constraint language for implementing program analysis problems. To conveniently specify equivalence relation, we provide the explicit equality constraints besides set-inclusion constraints in the language. Based on this special feature, we describe a parameterized framework, which allows an analysis designer to make use of both equality constraints and inclusion constraints in order to achieve a good tradeoff between performance and precision.

Two versions of the semantics for the language are specified: the first addresses the interest and the needs of analysis designers; the second addresses those of solver designers. A thorough study on their theoretical properties are conducted separately. Finally the relation between the two versions of the semantics is presented.

The content of this chapter builds up a firm theoretic basis for the algorithm design (presented in Chapter 4) and major part of the presentation was previously covered in [ZN08].

$$\begin{array}{rcl} \varphi & ::= & c \subseteq \alpha \mid \alpha \subseteq \beta \mid \alpha = \beta \mid \alpha \cap \beta \subseteq \gamma \mid \\ & & \alpha \setminus c \subseteq \beta \mid \alpha \setminus (D) \subseteq \beta \mid \varphi_1 \wedge \varphi_2 \\ D & ::= & ? \mid ?, D \mid m \mid m, D \end{array}$$

Table 3.1: Syntax of the Basic Constraint Language

3.1 Syntax of Basic Inclusion Constraint

We assume that a countable set of analysis variables is given and a tuple is a sequence of atomic values from the universe $\mathcal{U}$, whose members are left unspecified. A constant is a set of tuples. Formally,

$$\begin{array}{llllll} \alpha, \beta & \in & \mathbf{AVar} & & & \text{analysis variables} \\ t & \in & \mathbf{Tupl} & = & \mathcal{U}^* & \text{tuples} \\ c & \in & \widehat{\mathbf{Const}} & = & \mathcal{P}(\mathbf{Tupl}) & \text{constants} \end{array}$$

For the basic constraint language we use the following syntactic categories:

$$\begin{array}{llll} \varphi & \in & \mathbf{Clause} & \text{clauses} \\ D & \in & \mathbf{Template} & \text{templates} \end{array}$$

The syntax of the inclusion constraint is specified as in Table 3.1. Besides the normal set-inclusion relation between analysis variables, the constraint language can express equivalences between analysis variables with the explicit equality constraint. It is also possible to express the constraints $\alpha \cup \beta \subseteq \gamma$ and $\alpha \subseteq \beta \cap \gamma$ in terms of more primitive operations. For example, $\alpha \cup \beta \subseteq \gamma$ can be expressed as $\alpha \subseteq \gamma \wedge \beta \subseteq \gamma$. The union on the right hand side is dispensed with since it would destroy the Moore family property of a set of satisfiable solutions (presented in the next section). Two kinds of set minus operations are specified: one is standard, another is extended. The extended one uses the template D to represent a set of tuples: the values of some positions of these tuples are fixed and the rest can be any atomic values (represented by '?'). The overloaded set minus operation therefore removes all the tuples matching the template D from a set S, formally

$$\begin{array}{lll} S & \setminus & (m_1, \cdots, m_{i_1-1}, ?, m_{i_1+1}, \cdots, m_{i_k-1}, ?, m_{i_k+1}, \cdots, m_n) = \\ S & \setminus & \{(m_1, \cdots, m_{i_1-1}, \ell_1, m_{i_1+1}, \cdots, m_{i_k-1}, \ell_k, m_{i_k+1}, \cdots, m_n) \,|\, \ell_1, \cdots, \ell_k \in \mathcal{U}\} \end{array}$$

This syntax category is useful to make a concise implementation of program analyses. It also improves the performance of the solver by reducing the number of constraints that would otherwise be required to express the same logic using the standard set minus constraint.

1.	$\hat{\psi} \models c \subseteq \alpha$	iff	$c \subseteq \hat{\psi}(\alpha)$
2.	$\hat{\psi} \models \alpha \subseteq \beta$	iff	$\hat{\psi}(\alpha) \subseteq \hat{\psi}(\beta)$
3.	$\hat{\psi} \models \alpha = \beta$	iff	$\hat{\psi}(\alpha) = \hat{\psi}(\beta)$
4.	$\hat{\psi} \models \alpha \cap \beta \subseteq \gamma$	iff	$\hat{\psi}(\alpha) \cap \hat{\psi}(\beta) \subseteq \psi\hat{(\gamma)}$
5.1	$\hat{\psi} \models \alpha \setminus c \subseteq \beta$	iff	$\hat{\psi}(\alpha) \setminus c \subseteq \hat{\psi}(\beta)$
5.2	$\hat{\psi} \models \alpha \setminus (D) \subseteq \beta$	iff	$\hat{\psi}(\alpha) \setminus (D) \subseteq \hat{\psi}(\beta)$
6.	$\hat{\psi} \models \varphi_1 \ \wedge \ \varphi_2$	iff	$\hat{\psi} \models \varphi_1$ and $\hat{\psi} \models \varphi_2$

Table 3.2: Standard Semantics

3.2 Standard Interpretation

To specify the semantics of the constraint language, an analysis estimate $\hat{\psi}$ is used to associate constants with analysis variables:

$$\hat{\psi} \in \quad \widehat{\mathbf{Env}} \quad = \quad \mathbf{AVar} \to \widehat{\mathbf{Const}} \quad \text{abstract environments}$$

The semantics is defined by a satisfaction relation that has the form

$$\hat{\psi} \models \varphi$$

The judgement is true whenever $\hat{\psi}$ is an acceptable solution of φ. The rules of the semantics of the constraint language is specified in Table 3.2. For the constraint $c \subseteq \alpha$, the estimate $\hat{\psi}$ is valid whenever the data of α, i.e. $\hat{\psi}(\alpha)$, contains the constant c. The second rule demands that any data of β must be included by α. The rule for equality constraints declares that equivalent analysis variables contain a same set of tuples. The rule for the constraint $\alpha \cap \beta \subseteq \gamma$ says that γ should contain the common data shared by α and β. For the constraint using the standard set minus operator $\alpha \setminus c \subseteq \beta$, the set acquired from the operation $\hat{\psi}(\alpha) \setminus c$ is included by β. The rule for the constraint using non-standard set minus operator is very similar except that we remove all the tuples that match the template D from α this time. Finally the sixth rule declares that in order for an estimate $\hat{\psi}$ to be acceptable, it must comply with both the two sub-terms.

Example 3.1 Consider a constraint program

$$\{(a, b)\} \subseteq \alpha \wedge \alpha \subseteq \beta \wedge \{(c, d)\} \subseteq \beta$$

Two estimates are listed as below:

$$
\begin{array}{ll}
(\mathbf{a})\ \hat{\psi}(\alpha) = \{(a,b)\} & (\mathbf{b})\ \hat{\psi}(\alpha) = \{(a,b)\} \\
\quad\ \hat{\psi}(\beta) = \{(a,b),(c,d)\} & \quad\ \hat{\psi}(\beta) = \{(a,b),(c,d),(e,f)\}
\end{array}
$$

Both of them are acceptable to the given constraint program although the solution (b) is more imprecise than the solution (a). There could even be infinite solutions when the universe is infinite. Therefore it is important to study the properties of the abstract environments. □

3.2.1 Properties of Approximations

To better understand the relation between abstract environments, the following partial order is defined.

Definition 3.2 (Relation $\sqsubseteq$) For all $\hat{\psi}, \hat{\psi}' \in \widehat{\mathbf{Env}}$,

$$\hat{\psi} \sqsubseteq \hat{\psi}' \quad \text{iff} \quad \forall \alpha \in \mathbf{AVar} : \hat{\psi}(\alpha) \subseteq \hat{\psi}'(\alpha)$$

where $\subseteq$ is normal set inclusion operator.

Now consider Example 3.1 again, one can verify that the solution $\hat{\psi}$ is a least one: for any estimate $\hat{\psi}'$ that satisfies the constraint program of Example 3.1 we have that $\hat{\psi} \sqsubseteq \hat{\psi}'$. Since in general we are interested in a least solution, the below theorem then guarantees the existence of a unique least solution as desired.

Theorem 3.3 *For each $\varphi \in$ **Clause**, the set $\{\hat{\psi} |\ \hat{\psi} \models \varphi\}$ is a Moore family.*

Proof. First note that $(\mathcal{P}(\mathbf{Tupl}), \subseteq, \cap, \cup, \emptyset, \mathbf{Tupl})$ is a complete lattice. So is the function space $\widehat{\mathbf{Env}}$ according to Proposition 2.15. We prove the theorem by a structural induction on φ.

Case $c \subseteq \alpha$. Assume that

$$\forall i \in I : \hat{\psi}^i \models c \subseteq \alpha$$

for some set I, we show that $\sqcap_i \hat{\psi}^i \models c \subseteq \alpha$. From rule 1 in Table 3.2 we have

$$\forall i \in I : c \subseteq \hat{\psi}^i(\alpha)$$

where $\alpha \in dom(\hat{\psi}^i)$. Thus we have $c \subseteq \cap\{\hat{\psi}^i(\alpha)|\ i \in I\}$, i.e. $\sqcap_i \hat{\psi}^i \models c \subseteq \alpha$ (because of $(\sqcap_i \hat{\psi}^i)(\alpha) = \cap\{\hat{\psi}^i(\alpha)|\ i \in I\}$). This allows us to conclude that the set $\{\hat{\psi}|\ \hat{\psi} \models c \subseteq \alpha\}$ is a Moore family.

Case $\alpha = \beta$. Assume that

$$\forall i \in I : \hat{\psi}^i \models \alpha = \beta$$

for some set I. From rule 3 in Table 3.2 we have

$$\forall i \in I : \hat{\psi}^i(\beta) \subseteq \hat{\psi}^i(\alpha) \wedge \hat{\psi}^i(\alpha) \subseteq \hat{\psi}^i(\beta)$$

where $\alpha, \beta \in dom(\hat{\psi}^i)$. From the first component of the conjunction, we have that $\cap\{\hat{\psi}^i(\beta)|i \in I\} \subseteq \hat{\psi}_j(\alpha)$ for all $j \in I$ whence $\cap\{\hat{\psi}^i(\beta)|i \in I\} \subseteq \cap\{\hat{\psi}_j(\alpha)|j \in I\}$. Together with $(\sqcap_i \hat{\psi}^i)(\beta) = \cap\{\hat{\psi}^i(\beta)|i \in I\}$ and $(\sqcap_i \hat{\psi}^i)(\alpha) = \cap\{\hat{\psi}^i(\alpha)|i \in I\}$, we have that $\sqcap_i \hat{\psi}^i(\beta) \subseteq \sqcap_i \hat{\psi}^i(\alpha)$. Similarly from the second component of the conjunction, we have that $\sqcap_i \hat{\psi}^i(\alpha) \subseteq \sqcap_i \hat{\psi}^i(\beta)$. Then rule 3 in Table 3.2 ensures that $\sqcap_i \hat{\psi}^i \models \alpha = \beta$.

Case $\alpha \subseteq \beta$. Assume that

$$\forall i \in I : \hat{\psi}^i \models \alpha \subseteq \beta$$

for some set I. From rule 2 in Table 3.2 we have

$$\forall i \in I : \hat{\psi}^i(\alpha) \subseteq \hat{\psi}^i(\beta)$$

where $\alpha, \beta \in dom(\hat{\psi}^i)$. We then have that $\cap\{\hat{\psi}^i(\alpha)|\ i \in I\} \subseteq \hat{\psi}_j(\beta)$for all $j \in I$ whence $\cap\{\hat{\psi}^i(\alpha)|\ i \in I\} \subseteq \cap\{\hat{\psi}_j(\beta)|\ j \in I\}$. Together with $(\sqcap_i \hat{\psi}^i)(\alpha) = \cap\{\hat{\psi}^i(\alpha)|\ i \in I\}$ and $(\sqcap_i \hat{\psi}^i)(\beta) = \cap\{\hat{\psi}^i(\beta)|\ i \in I\}$, we have that $\sqcap_i \hat{\psi}^i(\alpha) \subseteq \sqcap_i \hat{\psi}^i(\beta)$. Then rule 2 in Table 3.2 ensures that $\sqcap_i \hat{\psi}^i \models \alpha \subseteq \beta$.

Case $\alpha \cap \beta \subseteq \gamma$. Assume that

$$\forall i \in I : \hat{\psi}^i \models \alpha \cap \beta \subseteq \gamma$$

for some set I. From rule 4 in Table 3.2 we have

$$\forall i \in I : \hat{\psi}^i(\alpha) \cap \hat{\psi}^i(\beta) \subseteq \hat{\psi}^i(\gamma)$$

We then have that $(\cap\{\hat{\psi}^i(\alpha)|\ i \in I\}) \cap (\cap\{\hat{\psi}^i(\beta)|\ i \in I\}) = \cap\{\hat{\psi}^i(\alpha) \cap \hat{\psi}^i(\beta)|\ i \in I\} \subseteq \cap\{\hat{\psi}^i(\gamma)|\ i \in I\}$ and thus $(\sqcap \hat{\psi}^i(\alpha) \cap \sqcap \hat{\psi}^i(\beta) \subseteq \sqcap \hat{\psi}^i(\gamma)$. By rule 4 $\sqcap_i \hat{\psi}^i \models \alpha \cap \beta \subseteq \gamma$.

Case $\alpha \setminus c \subseteq \beta$. Assume that

$$\forall i \in I : \hat{\psi}^i \models \alpha \setminus c \subseteq \beta$$

for some set I. From rule 5.1 in Table 3.2 we have

$$\forall i \in I : \hat{\psi}^i(\alpha) \setminus c \subseteq \hat{\psi}^i(\beta)$$

We show that $(\sqcap_i \psi^i)(\alpha) \setminus c \subseteq (\sqcap_i \psi^i)(\beta)$. First observe $(\cap\{\psi^i(\alpha)|\ i \in I\}) \setminus c = \cap\{\psi^i(\alpha) \setminus c|\ i \in I\} \subseteq \cap\{\psi^i(\beta)|\ i \in I\}$ and therefore $(\sqcap_i \psi^i)(\alpha) \setminus c \subseteq (\sqcap_i \psi^i)(\beta)$. By rule 5.1 in Table 3.2, $\sqcap_i \hat{\psi}^i \models \alpha \setminus c \subseteq \beta$.

Case $\alpha \setminus D \subseteq \beta$. Similar to the standard set minus above. Note that D represents a constant.

Case $\varphi_1 \wedge \varphi_2$. Assume that

$$\forall i \in I : \hat{\psi}^i \models \varphi_1 \ \wedge \ \varphi_2$$

for some set I. From rule 6 in Table 3.2, we immediately get that

$$\forall i \in I : \hat{\psi}^i \models \varphi_1 \ \mathit{and}\ \forall i \in I : \hat{\psi}^i \models \varphi_2$$

The induction hypothesis then gives that

$$\sqcap_i \hat{\psi}^i \models \varphi_1 \ \mathit{and}\ \sqcap_i \hat{\psi}^i \models \varphi_2$$

By rule 6 again, we conclude that $\sqcap_i \hat{\psi}^i \models \varphi_1 \ \wedge \ \varphi_2$. This completes the proof. □

3.2.2 Parameterized Framework

The inclusion constraint language of Section 3.1 provides both set-inclusion constraints and equality constraints. The equality constraint does not increase the expressiveness of the constraint language but functions as a syntactic shortcut for specifying equivalence relation, with which analysis designers can tune a system by syntactically switching between the two kinds of constraints. This is based on the two observations: set-inclusion constraints are quite precise in formulating analyses but takes cubic time to solve; in contrast, equality constraints are, in general, not as precise as set-inclusion constraints but can be solved in almost linear time [Tar83].

Intuitively, using equality constraints instead of set-inclusion constraints is safe because equality relation is more strict than set-inclusion relation. In order to formalize this observation, we define a relation $\leq$ over clauses:

Definition 3.4 (Relation $\leq$) For all $\varphi_1, \varphi_2 \in$ **Clause**, $\varphi_1 \leq \varphi_2$ if and only if φ_2 can be obtained by substituting the equality constraints for some or all set-inclusion constraints (over analysis variables) of φ_1.

It is straightforward to show that $\leq$ is indeed a partial order. We shall say that φ_1 can be lifted to φ_2 if and only if $\varphi_1 \leq \varphi_2$. Now we are ready to present the following two propositions, which state that any acceptable solution to the lifted constraint program is also valid to the original one, and so is the least solution respectively.

Proposition 3.5 *For all $\varphi_1, \varphi_2 \in$ **Clause**, if $\hat{\psi} \models \varphi_2$ and $\varphi_1 \leq \varphi_2$, then $\hat{\psi} \models \varphi_1$.*

Proof. The proof is conducted by an induction on φ_1.

Case $c \subseteq \alpha$. The case is trivial true since $\varphi_1 = \varphi_2$ by Definition 3.4.

Cases $\alpha = \beta, \alpha \setminus c \subseteq \beta, \alpha \setminus D \subseteq \beta,$ and $\alpha \cap \beta \subseteq \gamma$. Similarly.

Case $\alpha \subseteq \beta$. From Definition 3.4 φ_2 could be either $\alpha \subseteq \beta$ or $\alpha = \beta$. The case holds trivially if $\alpha \subseteq \beta$. For $\alpha = \beta$, assume

$$\hat{\psi} \models \alpha = \beta$$

i.e. $\hat{\psi}(\beta) = \hat{\psi}(\alpha)$ from rule 3 of Table 3.2. We immediately have

$$\hat{\psi} \models \alpha \subseteq \beta$$

by the fact $\hat{\psi}(\beta) = \hat{\psi}(\alpha) \Longrightarrow \hat{\psi}(\beta) \subseteq \hat{\psi}(\alpha)$ and rule 2 of Table 3.2.

Case $\varphi_1 \wedge \varphi_2$. According to Definition 3.4 φ_2 must have form $\varphi_1' \wedge \varphi_2'$ where $\varphi_1 \leq \varphi_1'$ and $\varphi_2 \leq \varphi_2'$. Assume that

$$\hat{\psi} \models \varphi_1' \wedge \varphi_2'$$

By rule 4 in Table 3.2 we have

$$\hat{\psi} \models \varphi_1' \text{ and } \hat{\psi} \models \varphi_2'$$

Then apply induction hypothesis on φ_1 and φ_2 and get

$$\hat{\psi} \models \varphi_1 \text{ and } \hat{\psi} \models \varphi_2$$

Finally rule 4 in Table 3.2 allows us to conclude

$$\hat{\psi} \models \varphi_1 \wedge \varphi_2$$

and this completes the whole proof. □

Proposition 3.6 *For all $\varphi_1, \varphi_2 \in$ **Clause**, if $\varphi_1 \leq \varphi_2$, then $\sqcap\{\hat{\psi} \mid \hat{\psi} \models \varphi_1\} \sqsubseteq \sqcap\{\hat{\psi} \mid \hat{\psi} \models \varphi_2\}$.*

Proof. According to Proposition 3.5 we have

$$\hat{\psi} \models \varphi_2 \Longrightarrow \hat{\psi} \models \varphi_1$$

Hence by the properties of greatest lower bound

$$\hat{\psi} \models \varphi_2 \Longrightarrow \sqcap\{\hat{\psi} \,|\, \hat{\psi} \models \varphi_1\} \sqsubseteq \hat{\psi}$$

and thus

$$\sqcap\{\hat{\psi} \,|\, \hat{\psi} \models \varphi_1\} \sqsubseteq \sqcap\{\hat{\psi} \,|\, \hat{\psi} \models \varphi_2\}$$

□

Lifting Strategy. As we shall show in our algorithm for constraint solving in the next chapter, general constraints can be solved in cubic time while unification on equality constraints is nearly linear. Under the framework we present, a general strategy of tuning systems is to try set-inclusion first because analysis designers would always prefer a precise solution if performance is acceptable. If the efficiency of the analysis is unsatisfactory, analysis designers can syntactically adjust the constraint program by lifting some set-inclusion and repeat the process until they achieve a good tradeoff between performance and precision.

The level of precision may be sacrificed by substituting equality for set-inclusion. But the designer of an analysis should be able to know where it may happen and thus has a good control on the loss of precision. For instance, a designer can choose not to use any equality constraints for the analysis of some part of a software system where high precision is desired, while he can use equality constraints more often at the rest part.

On the other hand, equality constraints do not necessarily lead to a loss in precision. Consider the following constraint program for example.

Example 3.7 Consider a constraint program

$$\{a,b,c\} \subseteq \alpha \;\wedge\; \{a,b\} \subseteq \beta \;\wedge \alpha \subseteq^{\dagger} \beta \;\wedge\; \beta \setminus \{b,c\} \subseteq \gamma \;\wedge \{a,c\} \subseteq \eta \;\wedge\; \alpha \subseteq^{\dagger} \eta$$

A least solution of the program is

$$\begin{aligned} \hat{\psi}(\alpha) &= \{a,b,c\} \\ \hat{\psi}(\beta) &= \{a,b,c\} \\ \hat{\psi}(\gamma) &= \{a\} \\ \hat{\psi}(\eta) &= \{a,b,c\} \end{aligned}$$

As the above solution shows, in a least model α has the same data as β and η, and thus substituting equality relations for the set-inclusions (marked with †) would maintain the level of precision. □

1.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} c \subseteq \alpha$	iff	$c \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$
2.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \subseteq \beta$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
3.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha = \beta$	iff	$\hat{\psi}_1(\alpha) = \hat{\psi}_1(\beta)$
4.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \cap \beta \subseteq \gamma$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \cap \hat{\psi}_2(\hat{\psi}_1(\beta)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\gamma))$
5.1	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \setminus c \subseteq \beta$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \setminus c \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
5.2	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \setminus (D) \subseteq \beta$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \setminus (D) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
6.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1 \ \wedge \ \varphi_2$	iff	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1$ and $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_2$

Table 3.3: Semantics Using Type Variables

3.3 Interpretation Using Type Variables

The standard semantics is user-friendly but unclear in specifying how unification benefits our computation. In this section we make it explicit in the semantic specification so that the interesting properties of the new interpretation can be addressed separately from the algorithm of constraint solver. The presentation of this section therefore provides a theoretic foundation for the implementation of the constraint solver.

Specifically we specify a double-layer interpretation using a new category, type variables $i \in \mathbf{TV}$, and enforce that equivalent analysis variables map onto the same type variable explicitly in semantics and hence the corresponding constants are collapsed into one constant. A type-variable solution therefore has two components:

$$\begin{array}{llcll} \hat{\psi}_1 \in & \mathbf{Env_T} & = & \mathbf{AVar} \to \mathbf{TV} & \text{type environment} \\ \hat{\psi}_2 \in & \widehat{\mathbf{Env_{TB}}} & = & \mathbf{TV} \to \widehat{\mathbf{Const}} & \text{type-binding environment} \end{array}$$

The acceptable relation now has the form

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi$$

and the rules are given in Table 3.3. Let $\hat{\psi} = \hat{\psi}_2 \circ \hat{\psi}_1$ then the rules are the same as those in Table 3.2 except for the third one, which enforces that two equivalent analysis variables must be unified onto a same type variable, i.e. $\hat{\psi}_1(\beta) = \hat{\psi}_1(\alpha)$. To further illustrate the difference, consider the below example.

Example 3.8

$$\{(a, b), (c, d)\} \subseteq \alpha \wedge \{(c, d)\} \subseteq \beta \wedge \alpha = \beta \wedge \beta \setminus \{(a, b)\} \subseteq \gamma$$

Two estimates are accordingly specified as the following.

$$\begin{array}{lll@{\quad}lll}
(\mathbf{a}) & \hat{\psi}_1(\alpha) = 1 & \hat{\psi}_2(1) = \{(a,b),(c,d)\} & (\mathbf{b}) & \hat{\psi}_1(\alpha) = 1 & \hat{\psi}_2(1) = \{(a,b),(c,d)\} \\
 & \hat{\psi}_1(\beta) = 1 & & & \hat{\psi}_1(\beta) = 2 & \hat{\psi}_2(2) = \{(a,b),(c,d)\} \\
 & \hat{\psi}_1(\gamma) = 2 & \hat{\psi}_2(2) = \{(c,d)\} & & \hat{\psi}_1(\gamma) = 3 & \hat{\psi}_2(3) = \{(c,d)\}
\end{array}$$

Intuitively they have the same data for each analysis variable. In the sense of the first semantics, both of them are acceptable (suppose $\hat{\psi} = \hat{\psi}_2 \circ \hat{\psi}_1$). However, only **(a)** is valid with respect to the second semantics because $\hat{\psi}_1(\alpha) \neq \hat{\psi}_1(\beta)$. Notice also that the unification coalesces the analysis variables onto one type variable and hence avoids storing redundant information in the environment $\widehat{\mathbf{Env_{TB}}}$. Therefore the use of unification not only reduces the problem space but also saves the space of storing the data fields of each type variable. □

One challenge of adopting the type-variable solution is that the least solution is not unique and may even potentially be infinite. For example, another solution could be acquired for the program in Example 3.8 by reordering the type variables used in **(a)**. In the rest of this section, we study the relation between these least solutions and present the principle of *designated solution* to remove the non-determinism of the choice of type variables. We show further the correctness of the second semantics with respect to the first. Therefore it is sufficient for analysis designers to understand the first semantics and the strategy of tuning systems, and leave the technical details of solving constraints to solver designers.

We start our development with a definition of an ordering relation:

Definition 3.9 (Relation $\preceq$) For $(\hat{\psi}_1, \hat{\psi}_2), (\hat{\psi}'_1, \hat{\psi}'_2) \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$, define

$$(\hat{\psi}_1, \hat{\psi}_2) \preceq (\hat{\psi}'_1, \hat{\psi}'_2) \Longleftrightarrow \exists \pi : \mathbf{TV} \to \mathbf{TV} : \hat{\psi}'_1 = \pi \circ \hat{\psi}_1 \quad \wedge \quad \hat{\psi}_2 \sqsubseteq \hat{\psi}'_2 \circ \pi$$

where π is a total function and $\hat{\psi}_2 \sqsubseteq \hat{\psi}'_2 \circ \pi \Leftrightarrow \forall\, i \in \mathsf{rang}(\psi_1) : \hat{\psi}_2(i) \subseteq \hat{\psi}'_2(\pi(i))$.

Note that it does not matter for the type variables out of the range of ψ_1 in the context of this dissertation. It is straightforward to verify that the relation $\preceq$ is a pre-order, but not a partial order. This definition is not constructive since the definition of the total function π remains unspecified. In preparation for showing the relation between the least solutions of a constraint program, two lemmata, Lemma 3.10 and 3.11 are specified and proved first as follows: the first lemma provides a more constructive way of verifying the relation $\preceq$ than Def. 3.9; the second further generalizes the result onto an equivalence relation $\equiv$.

Lemma 3.10 $(\hat{\psi}_1, \hat{\psi}_2) \preceq (\hat{\psi}'_1, \hat{\psi}'_2)$ *if and only if*

$$\begin{array}{rll} \forall \alpha, \beta \in \mathbf{AVar}: & \hat{\psi}_1(\alpha) = \hat{\psi}_1(\beta) \Rightarrow \hat{\psi}'_1(\alpha) = \hat{\psi}'_1(\beta) \wedge & \text{(i)} \\ \forall \gamma \in \mathbf{AVar}: & \hat{\psi}_2(\hat{\psi}_1(\gamma)) \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\gamma)) & \text{(ii)} \end{array}$$

Proof. Following from the Definition 3.9 the lemma is proved by the observation that every equivalence analysis variable should bind to the same type variable and the functional compositions of the two pairs have the relation $\hat{\psi}_2 \circ \hat{\psi}_1 \subseteq \hat{\psi}_2{}' \circ \hat{\psi}_1{}'$. Formally,

(Only-if.) Suppose $(\hat{\psi}_1, \hat{\psi}_2) \preceq (\hat{\psi}'_1, \hat{\psi}'_2)$, then from Definition 3.9 we are sure that

$$\begin{array}{rll} \exists \pi : & \hat{\psi}'_1 = \pi \circ \hat{\psi}_1 \quad \wedge & \text{(iii)} \\ & \hat{\psi}_2 \sqsubseteq \hat{\psi}'_2 \circ \pi & \text{(iv)} \end{array}$$

We then have that

$$\hat{\psi}_1(\alpha) = \hat{\psi}_1(\beta) \Longrightarrow (\pi \circ \hat{\psi}_1)(\alpha) = (\pi \circ \hat{\psi}_1)(\beta)$$

for some π such that $\hat{\psi}'_1 = \pi \circ \hat{\psi}_1$ from the definition of function composition. This allows us to conclude that

$$\hat{\psi}_1(\alpha) = \hat{\psi}_1(\beta) \Longrightarrow \hat{\psi}'_1(\alpha) = \hat{\psi}'_1(\beta)$$

Since $\hat{\psi}_2 \sqsubseteq \hat{\psi}'_2 \circ \pi$ and $\hat{\psi}_1(\alpha) = \hat{\psi}_1(\alpha)$, we get

$$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq (\hat{\psi}'_2 \circ \pi)(\hat{\psi}_1(\alpha)) \qquad \text{(v)}$$

Also from the condition (iv) and associativity of function composition, we have

$$\begin{array}{rl} & \hat{\psi}'_2(\hat{\psi}'_1(\alpha)) \\ = & \hat{\psi}'_2(\pi \circ \hat{\psi}_1(\alpha)) \\ = & \hat{\psi}'_2 \circ (\pi \circ \hat{\psi}_1)(\alpha) \\ = & \hat{\psi}'_2 \circ \pi \circ \hat{\psi}_1(\alpha) \\ = & (\hat{\psi}'_2 \circ \pi)(\hat{\psi}_1(\alpha)) \end{array}$$

Together with the condition (v), we conclude that $\forall \alpha \in \mathbf{AVar} : \hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$.

(If.) We define function π by

$$\begin{array}{rcl} \forall x \in \mathbf{AVar} & : & \pi(\hat{\psi}_1(\alpha)) = \hat{\psi}'_1(\alpha) \wedge \\ \forall t \in \mathbf{TV} & : & t \notin \mathsf{range}(\hat{\psi}_1) : \pi(t) = t \end{array}$$

The function is well-defined by the condition (i) and it is straightforward to show the function is a total function and $\hat{\psi}'_1 = \pi \circ \hat{\psi}_1 \quad \wedge \quad \hat{\psi}_2 \sqsubseteq \hat{\psi}'_2 \circ \pi$ by (ii) and therefore $(\hat{\psi}_1, \hat{\psi}_2) \preceq (\hat{\psi}'_1, \hat{\psi}'_2)$. □

The formula (i) demonstrates an important property of the relation $\preceq$ when the Moore family property is considered later: if a least solution, say $(\hat{\psi}^l_1, \hat{\psi}^l_2)$, exists for a constraint program φ, then for two analysis variables of φ, say α and β, we have that $\hat{\psi}^l_1(\alpha) = \hat{\psi}^l_1(\beta)$ if and only if $\alpha = \beta$ is specified in φ or can be implied by two or more equality constraints by the transitivity of equivalence relation, e.g. $\alpha = \gamma \wedge \gamma = \beta$. We further define the relation $\equiv$ by

$$(\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2) \stackrel{\text{def}}{=} ((\hat{\psi}_1, \hat{\psi}_2) \preceq (\hat{\psi}'_1, \hat{\psi}'_2)) \wedge ((\hat{\psi}'_1, \hat{\psi}'_2) \preceq (\hat{\psi}_1, \hat{\psi}_2))$$

for $\hat{\psi}_1, \hat{\psi}'_1 \in \mathbf{Env_T}$ $\hat{\psi}_2, \hat{\psi}'_2 \in \widehat{\mathbf{Env_{TB}}}$. It is straightforward to verify that the relation $\equiv$ is an equivalence relation, i.e. it is reflexive, transitive and symmetric.

Lemma 3.11 $(\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$ *iff*

$$\begin{aligned} \forall \alpha, \beta \in \mathbf{AVar} : \quad & \hat{\psi}_1(\alpha) = \hat{\psi}_1(\beta) \Leftrightarrow \hat{\psi}'_1(\alpha) = \hat{\psi}'_1(\beta) \wedge & \text{(vi)} \\ \forall \gamma \in \mathbf{AVar} : \quad & \hat{\psi}_2(\hat{\psi}_1(\gamma)) = \hat{\psi}'_2(\hat{\psi}'_1(\gamma)) & \text{(vii)} \end{aligned}$$

Proof. The result follows directly from the definition of equivalence relation $\equiv$ and Lemma 3.10. □

Lemma 3.11 shows that the equivalence of two pairs amounts to checking the conjuncts (vi) and (vii). We are now ready for specifying the following proposition which states that given a constraint program, if a pair is acceptable then so are all its equivalences.

Proposition 3.12 *If* $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$, *then* $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \varphi$.

Proof. The proof is conducted by induction on φ.

Case $c \subseteq x$**.** Assume that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} c \subseteq \alpha \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$$

From rule 1 in Table 3.3 we have $c \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$. We show that $c \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$. By Lemma 3.11 we have $\hat{\psi}_2(\hat{\psi}_1(\alpha)) = \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$ whence $c \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$ because of the transitivity of inclusion relation. Finally the first rule in Table 3.3 allows us to conclude that $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} c \subseteq \alpha$ as desired.

Case $\alpha \subseteq \beta$**.** Assume that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \subseteq \beta \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$$

By the second rule in Table 3.3 we get $\hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$. The Lemma 3.11 gives $\forall \alpha \in$ **AVar** $: \hat{\psi}_2(\hat{\psi}_1(\alpha)) = \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$ and thus $\hat{\psi}'_2(\hat{\psi}'_1(\alpha)) \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\beta))$. By rule 2 in Table 3.3 we conclude that $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \alpha \subseteq \beta$.

Case $\alpha = \beta$**.** Suppose that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha = \beta \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$$

From the third rule in Table 3.3 we have $\hat{\psi}_1(\alpha) = \hat{\psi}_1(\beta)$. From Lemma 3.11, we immediately get $\hat{\psi}'_1(\alpha) = \hat{\psi}'_1(\beta)$. Finally rule 3 in Table 3.3 allows us to conclude that $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \alpha = \beta$.

Case $\alpha \setminus c \subseteq \beta$**,** $\alpha \setminus (D) \subseteq \beta$**, and** $\alpha \cap \beta \subseteq \gamma$**.** Similarly.

Case $\varphi_1 \wedge \varphi_2$**.** Assume that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1 \wedge \varphi_2$$

The sixth rule gives that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1 \wedge (\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_2$$

The induction hypothesis are applied to two subcomponents and gives that

$$(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \varphi_1 \wedge (\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \varphi_2$$

Then rule 6 in Table 3.3 then allows us to conclude that

$$(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \varphi_1 \wedge \varphi_2$$

This completes the whole proof. □

Even in the setting of pre-ordered sets, a least solution is still our interest. Therefore it is necessary to build up a theoretic result that is similar to the Moore family property. This leads to a series of definitions and theoretical development as the following.

At first, the definitions of the least upper bound and the greatest lower bound for a pre-ordered set are given and they are quite similar to those for a partially ordered set.

Definition 3.13 (Least Upper Bound & Greatest Lower Bound) A least upper bound (*lub*) of a subset S of a pre-ordered set $(P, \preceq)$ is an upper bound u of S such that whenever u' is an upper bound of S we have $u \preceq u'$. A greatest lower bound (*glb*) of a subset S of a pre-ordered set $(P, \preceq)$ is a lower bound ℓ of S such that whenever ℓ' is an lower bound of S we have $\ell' \preceq \ell$.

Fact 3.14 *If x and y are two least upper bounds (greatest lower bounds) of a set $S \subseteq P$ then $x \equiv y$.*

Proof. By Definition 3.13, we have

$$(x \preceq y) \wedge (y \preceq x)$$

and thus $x \equiv y$ from the definition of equivalence $\equiv$. □

Although Fact 3.14 clarifies the relation between least solutions, there is no guarantee that a unique least solution exists. To deal with such a problem there are at least two ways : one way is to introduce the concepts of the designated greatest lower bound and the designated least upper bound of a subset S of a pre-ordered set P denoted by $\hat{\sqcap}\, S$ and $\hat{\sqcup}\, S$ respectively. Suppose there is a choice function which given a set of elements returns a designated one. Another way is to group all equivalent elements into one class such that the relation between groups is a partial order again; then we need to work out how to represent the equivalence class. We take the first approach since it more directly applies to the setting in this dissertation. But clearly the two approaches are connected to each other.

In order to prove the existence of least solution(s) the concepts of the designated greatest lower bound (least upper bound) of a subset S of a pre-ordered set P are introduced and denoted by $\hat{\sqcap}\, S$ ($\hat{\sqcup}\, S$). We assume that there is a choice function that given a set of elements, returns a designated one. The following lemma specifies an important property hold for a greatest lower bound of a pre-ordered set.

Lemma 3.15 *For some set I and a family $(\psi_1^i, \psi_2^i)_{(i \in I)} \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$, let $(\psi_1^{\hat{\sqcap}}, \psi_2^{\hat{\sqcap}}) = \hat{\sqcap}_{i \in I}(\psi_1^i, \psi_2^i)$, then for all $\alpha, \beta, \gamma \in \mathbf{AVar}$:*

$$\begin{aligned}\hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^{\hat{\sqcap}}(\beta) \Leftrightarrow \forall i \in I : \psi_1^i(\alpha) = \psi_1^i(\beta) \wedge \\ \psi_2^{\hat{\sqcap}}(\psi_1^{\hat{\sqcap}}(\gamma)) = \cap_i \hat{\psi}_2^i(\psi_1^i(\gamma))\end{aligned}$$

Proof. Since $(\psi_1^{\hat{\sqcap}}, \psi_2^{\hat{\sqcap}})$ is a lower bound, we have

$$\begin{aligned}&\hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^{\hat{\sqcap}}(\beta) \Rightarrow \forall i \in I : \psi_1^i(\alpha) = \psi_1^i(\beta) \ \wedge \\ &\psi_2^{\hat{\sqcap}}(\psi_1^{\hat{\sqcap}}(\gamma)) \subseteq \cap_i \hat{\psi}_2^i(\psi_1^i(\gamma))\end{aligned}$$

by Lemma 3.10. Suppose a lower bound $(\hat{\psi}_1^\ell, \hat{\psi}_2^\ell)$ such that

$$\begin{aligned}&\forall i \in I : \psi_1^i(\alpha) = \psi_1^i(\beta) \Rightarrow \hat{\psi}_1^\ell(\alpha) = \hat{\psi}_1^\ell(\beta) \ \wedge \\ &\psi_2^\ell(\psi_1^\ell(\gamma)) = \cap_i \hat{\psi}_2^i(\psi_1^i(\gamma))\end{aligned}$$

for all $\alpha, \beta, \gamma \in \mathbf{AVar}$. It is easy to verify that such a lower bound does exist by Lemma 3.10. Considering $(\psi_1^{\hat{\sqcap}}, \psi_2^{\hat{\sqcap}})$ is a greatest lower bound, we immediately have

$$\begin{aligned}&\hat{\psi}_1^{\ell}(\alpha) = \hat{\psi}_1^{\ell}(\beta) \Rightarrow \hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^{\hat{\sqcap}}(\beta) \ \wedge \\ &\psi_2^{\ell}(\psi_1^{\ell}(\gamma)) \subseteq \psi_2^{\hat{\sqcap}}(\psi_1^{\hat{\sqcap}}(\gamma))\end{aligned}$$

for all $\alpha, \beta, \gamma \in \mathbf{AVar}$. This allows us to conclude that

$$\begin{aligned}&\hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^{\hat{\sqcap}}(\beta) \Leftrightarrow \forall i \in I : \psi_1^{i}(\alpha) = \psi_1^{i}(\beta) \ \wedge \\ &\psi_2^{\hat{\sqcap}}(\psi_1^{\hat{\sqcap}}(\gamma)) = \cap_i \hat{\psi}_2^{i}(\psi_1^{i}(\gamma))\end{aligned}$$

as desired. □

With the help of designated function, we define the concepts complete prelattice and Moore family in complete prelattice. Based on these definitions we show that the least models exist for any satisfiable constraint programs.

Definition 3.16 (Complete Prelattice) A complete prelattice $P = (P, \preceq, \hat{\sqcup}, \hat{\sqcap}, \hat{\bot}, \hat{\top})$ is a preordered set such that all its subsets have least upper bounds (with $\hat{\sqcup}S$ a designated least upper bound for S) and greatest lower bounds (with $\hat{\sqcap}S$ a designated greatest lower bound for S). Furthermore, $\hat{\bot} = \hat{\sqcup}\,\emptyset = \hat{\sqcap}\,P$ is a designated least element and $\hat{\top} = \hat{\sqcap}\,\emptyset = \hat{\sqcup}\,P$ is a designated greatest element.

Lemma 3.17 *For a preordered set $(P, \preceq)$ the following statements are equivalent:*
(1) $(P, \preceq)$ can be extended to a complete prelattice $(P, \preceq, \hat{\sqcup}, \hat{\sqcap}, \hat{\bot}, \hat{\top})$;
(2) Every subset of P has a least upper bound;
(3) Every subset of P has a greatest lower bound.

Proof. Clearly (1) implies (2) and (3). For the proof that (2) implies (1), let $S \subseteq P$ and define

$$\hat{\sqcap}\,S = \hat{\sqcup}\,\{\ell \in P | \, \forall \ell' \in S : \ell \preceq \ell'\}$$

We show $\hat{\sqcap}\,S$ defines a greatest lower bound. First, any element of $S \subseteq P$ is an upper bound of $\{\ell \in P | \forall \ell' \in S : \ell \preceq \ell'\}$. From (2) we are sure that $\hat{\sqcap}\,S$ exists and is a lower bound of S.

Second, for any lower bound z of S, we have $\forall \ell \in S : z \preceq \ell$ whence $z \in \{\ell \in P | \forall \ell' \in S : \ell \preceq \ell'\}$. From the definition of least upper bound we get $z \preceq \sqcap S$ proving that $\hat{\sqcap}\,S$ is a greatest lower bound of S.
To show that (3) implies (1) we define

$$\hat{\sqcup}\,S = \hat{\sqcup}\,\{\ell \in P | \, \ell' \in S : \ell' \preceq \ell\}$$

and show this defines a least upper bound. The argument is similar to that above. □

In preparation for proving that $(\mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}, \preceq)$ is a complete prelattice we give a definition for representing a specific greatest lower bound.

Definition 3.18 (Operator $\hat{\sqcap}_T$) For some set $I = \{1, 2, \cdots, n\}$ and a family $(\psi_1^i, \psi_2^i)_{(i \in I)} \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$, let $(\psi_1^{\hat{\sqcap}}, \psi_2^{\hat{\sqcap}}) = \hat{\sqcap}_{T_{i\in I}}(\psi_1^i, \psi_2^i)$ which is given by:

$$\forall \alpha \in \mathbf{AVar} : \psi_1^{\hat{\sqcap}}(x) = \tau_{1_{(\tau_2, \cdots, \tau_n)}}$$

where $\forall i \in I : \tau_i = \psi_1^i(\alpha)$ and

$$\psi_2^{\hat{\sqcap}}(\iota_{1_{(\iota_2, \cdots, \iota_n)}}) = \cap_i \psi_2^i(\iota_i)$$

where $\iota_{1_{(\iota_2, \cdots, \iota_n)}} \in dom(\psi_1^{\hat{\sqcap}})$.

We then show it is indeed a greatest lower bound by Lemma A.2 and A.3 below.

Lemma 3.19 *For some set $I = \{1, 2, \cdots, n\}$ and a family $(\psi_1^i, \psi_2^i)_{(i \in I)} \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$, let $(\psi_1^{\hat{\sqcap}}, \psi_2^{\hat{\sqcap}}) = \hat{\sqcap}_{T_{i\in I}}(\psi_1^i, \psi_2^i)$, then for all $\alpha, \beta, \gamma \in \mathbf{AVar}$ and $i \in I$:*

$$\begin{aligned} &\hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^{\hat{\sqcap}}(\beta) \Leftrightarrow \psi_1^i(\gamma) = \psi_1^i(\beta) \wedge \\ &\psi_2^{\hat{\sqcap}}(\psi_1^{\hat{\sqcap}}(\gamma)) = \cap_i \hat{\psi}_2^i(\psi_1^i(\gamma)) \end{aligned}$$

Proof. It is straightforward to prove the above formulas according to Definition A.1. The proof relies on the fact that for any analysis variable x: $\hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^1(\alpha)_{\hat{\psi}_1^2(\alpha), \cdots, \hat{\psi}_1^n(\alpha)}$. □

Lemma 3.20 *Let $E = \{(\hat{\psi}_1^i, \hat{\psi}_2^i) | \, i \in I \wedge (\hat{\psi}_1^i, \hat{\psi}_2^i) \in \mathbf{Env_T} \times \mathbf{Env_{TB}}\}$ for some set I, and $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) = \hat{\sqcap}_T E$. Then $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}})$ is a greatest lower bound of the set E.*

Proof. It is straightforward to show $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}})$ is a lower bound by Lemmata 3.10 and A.2. We prove it is a greatest one. For any lower bound of E, e.g.

$(\hat{\psi}_1^\ell, \hat{\psi}_2^\ell)$, we have for all $i \in I$ and $\alpha, \beta, \gamma \in \mathbf{AVar}$

$$\hat{\psi}_1^\ell(\alpha) = \hat{\psi}_1^\ell(\beta) \Rightarrow \hat{\psi}_1^i(\alpha) = \hat{\psi}_1^i(\beta) \wedge \\ \hat{\psi}_1^\ell(\hat{\psi}_1^\ell(\gamma)) \subseteq \hat{\psi}_1^i(\hat{\psi}_1^i(\gamma))$$

from Lemma 3.10 and therefore

$$\hat{\psi}_1^\ell(\alpha) = \hat{\psi}_1^\ell(\beta) \Rightarrow \hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^{\hat{\sqcap}}(\beta) \\ \hat{\psi}_1^\ell(\hat{\psi}_1^\ell(\gamma)) \subseteq \hat{\psi}_1^{\hat{\sqcap}}(\hat{\psi}_1^{\hat{\sqcap}}(\gamma))$$

by Lemma A.2. Now the preorder definition of $\preceq$ allows us to conclude that $(\hat{\psi}_1^\ell, \hat{\psi}_2^\ell) \preceq (\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}})$ for any lower bound $(\hat{\psi}_1^\ell, \hat{\psi}_2^\ell)$ of E. □

Following from the above definition and lemma, we have

Fact 3.21 *The pre-ordered set* $(\mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}, \preceq)$ *is a complete prelattice.*

Proof. First note that $\preceq$ is a preorder. To show the existence of a greatest lower bound, we refer to the definition of the operator $\hat{\sqcap}_T$ and the result from Lemma A.3: for any subset O of $E \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$, a greatest lower bound of O is given by $\hat{\sqcap}_T O$. Finally Lemma 3.17 ensures that the least upper bounds of O also exist and thus $(\mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}, \preceq)$ is a complete prelattice. □

The formalisms above are quite similar to their counterparts in the world of partially ordered sets. This is because the designated bounds enable us to work around the randomness of choosing type variables. However, sticking to the designated solutions may lose some generality in defining a Moore family for complete prelattices. For example, consider a subset S of a pre-ordered set R. A straightforward definition of Moore family using the designated bounds could be: $\forall S' \subseteq S : \hat{\sqcap} S' \in S$, i.e. S is closed under the designated greatest lower bound. However since the Moore family property is in fact concerned with the existence of the greatest lower bound, we would like to give a more generic definition that retains the original meaning of Moore family: Instead of enforcing $\hat{\sqcap} S' \in S$ we want to express that $\exists u : u \equiv \hat{\sqcap} S' \wedge u \in S$. This idea is further formalized by a compositional operator $\equiv\in$ (read as *"is represented in"*).

Definition 3.22 (Relation $\equiv\in$) For an element e and a set P, we say that $e \equiv\in P$ if and only if there exists an element e' such that $e \equiv e'$ and $e' \in P$.

Definition 3.23 (Moore Family for a Complete Prelattice) A Moore family for a complete prelattice is a subset M of a complete prelattice $P = (P, \preceq)$ such that it is closed under greatest lower bounds, formally $\forall M' \subseteq M : \hat{\cap} M' \equiv\in M$. Similar to Moore families for partially ordered sets, a Moore family for a complete prelattice always contains at least one least element and one greatest element, formally $\hat{\cap}\emptyset \equiv\in M$ and $\hat{\cap}M \equiv\in M$. Thus it is never empty.

Applying the above definition, we have that the least solution is guaranteed for the set of pairs $(\hat{\psi}_1, \hat{\psi}_2)$ such that $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi$, formally:

Theorem 3.24 *A set of solutions given by* $\{(\hat{\psi}_1, \hat{\psi}_2) | \, (\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi\}$ *is a Moore family for a complete prelattice.*

Proof. Note that $(\mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}, \preceq)$ is a complete prelattice. We then prove the theorem by structural induction on φ.

Case $c \subseteq x$**.** Assume that

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} c \subseteq x$$

for some set I and let $(\hat{\psi}_1^{\hat{\cap}}, \hat{\psi}_2^{\hat{\cap}}) = \hat{\cap}_i(\hat{\psi}_1^i, \hat{\psi}_2^i)$. We show that $(\hat{\psi}_1^{\hat{\cap}}, \hat{\psi}_2^{\hat{\cap}}) \models_{\mathcal{T}} c \subseteq x$. From rule 1 in Table 3.3 we have

$$\forall i \in I : c \subseteq \hat{\psi}_2^i(\hat{\psi}_1^i(x))$$

Thus we have $c \subseteq \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(x))$ and thus $c \subseteq \hat{\psi}_2^{\hat{\cap}}(\hat{\psi}_1^{\hat{\cap}}(x))$ by Lemma 3.15. Finally the fist rule of Table 3.3 allows us to conclude that $(\hat{\psi}_1^{\hat{\cap}}, \hat{\psi}_2^{\hat{\cap}}) \models_{\mathcal{T}} c \subseteq x$.

Case $\alpha \subseteq \beta$**.** Assume that

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} \alpha \subseteq \beta$$

for some set I. From rule 2 in Table 3.3 we have

$$\forall i \in I : \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) \subseteq \hat{\psi}_2^i(\hat{\psi}_1^i(\beta))$$

We then have that $\cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) \subseteq \hat{\psi}_2^j(\hat{\psi}_1^j(\beta))$ for all $j \in I$ whence $\cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) \subseteq \cap_j \hat{\psi}_2^j(\hat{\psi}_1^j(\beta))$. Together with $\hat{\psi}_2^{\hat{\cap}}(\hat{\psi}_1^{\hat{\cap}}(\beta)) = \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\beta))$ for any analysis variable β, we have that $\hat{\psi}_2^{\hat{\cap}}(\hat{\psi}_1^{\hat{\cap}}(\alpha)) \subseteq \hat{\psi}_2^{\hat{\cap}}(\hat{\psi}_1^{\hat{\cap}}(\beta))$. Then rule 2 in Table 3.3 ensures that $(\hat{\psi}_1^{\hat{\cap}}, \hat{\psi}_2^{\hat{\cap}}) \models_{\mathcal{T}} \alpha \subseteq \beta$.

Case $\alpha \setminus c \subseteq \beta, \alpha \setminus (D) \subseteq \beta$, and $\alpha \cap \beta \subseteq \gamma$ are similar.

Case $\alpha = \beta$**.** Assume that

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} \alpha = \beta$$

for some set I. From rule 3 in Table 3.3 we have

$$\forall i \in I : \hat{\psi}_1^i(\alpha) = \hat{\psi}_1^i(\beta)$$

From Lemma 3.15 we have $\hat{\psi}_1^{\hat{\sqcap}}(\alpha) = \hat{\psi}_1^{\hat{\sqcap}}(\beta)$. Then rule 3 in Table 3.3 allows that $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) \models_{\mathcal{T}} \alpha = \beta$.

Case $\varphi_1 \wedge \varphi_2$**.** Assume that

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} \varphi_1 \ \wedge \ \varphi_2$$

for some set I. From rule 6 in Table 3.3, we immediately get that

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} \varphi_1 \text{ and } \forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} \varphi_2$$

The induction hypothesis then gives that

$$(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) \models_{\mathcal{T}} \varphi_1 \textit{ and } (\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) \models_{\mathcal{T}} \varphi_2$$

By rule 6 again, we conclude that $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) \models_{\mathcal{T}} \varphi_1 \ \wedge \ \varphi_2$. This completes the proof. □

Finally we relate the results of the type variable interpretation back to those of the standard one by showing (1) the second semantics complies with the lifting strategy (in Proposition 3.25), (2) a solution (ψ_1, ψ_2) w.r.t. the type variable interpretation implies a solution w.r.t. the standard interpretation (in Proposition 3.26), and (3) the least solution using type variable is as precise as that of the standard one (in Proposition 3.27).

Proposition 3.25 *If* $\varphi_1 \leq \varphi_2$ *and* $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_2$ *then* $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1$*.*

Proof. The proof is a straightforward induction on the clause φ_1. □

Proposition 3.26 *If* $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi$*, then* $\hat{\psi}_2 \circ \hat{\psi}_1 \models \varphi$*.*

Proof. The proof is an induction on φ.
Case $c \subseteq x$**.** Assume that $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} c \subseteq x$, we have that $c \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$ by rule 1 in Table 3.3 and thus $c \subseteq \hat{\psi}_2 \circ \hat{\psi}_1(\alpha)$. From rule 1 in Table 3.2 we have $\hat{\psi}_2 \circ \hat{\psi}_1 \models c \subseteq \alpha$.

Case $\alpha \subseteq \beta$**.** Assume that $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \subseteq \beta$. From rule 2 in Table 3.3 we have $\hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$. Then rule 2 in Table 3.2 ensures that $\hat{\psi}_2 \circ \hat{\psi}_1 \models \alpha \subseteq \beta$.

Case $\alpha \setminus c \subseteq \beta, \alpha \setminus (D) \subseteq \beta$, and $\alpha \cap \beta \subseteq \gamma$ are similar.

Case $\alpha = \beta$**.** Assume that $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha = \beta$. From rule 3 in Table 3.3 we have $\hat{\psi}_1(\alpha) = \hat{\psi}_1(\beta)$ and thus $\hat{\psi}_2(\hat{\psi}_1(\alpha)) = \hat{\psi}_2(\hat{\psi}_1(\beta))$. Then rule 3 in Table 3.2 allows that $\hat{\psi}_2 \circ \hat{\psi}_1 \models \alpha = \beta$.

Case $\varphi_1 \wedge \varphi_2$**.** Assume that $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1 \ \wedge \ \varphi_2$. From rule 6 in Table 3.3, we immediately get that $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1$ and $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_2$. The induction hypothesis then gives that $\hat{\psi}_2 \circ \hat{\psi}_1 \models \varphi_1$ and $\hat{\psi}_2 \circ \hat{\psi}_1 \models \varphi_2$. By rule 6 in Table 3.2, we conclude that $\hat{\psi}_2 \circ \hat{\psi}_1 \models \varphi_1 \ \wedge \ \varphi_2$. □

Intuitively the following proposition says that if a least model is concerned the map from analysis variable to data fields for two semantics is exactly the same: the use of type variables have no side effect on it.

Proposition 3.27 *Let* $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) = \hat{\sqcap}\{(\hat{\psi}_1, \hat{\psi}_2) | \, (\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi\}$ *for some* $\varphi \in$ **Clause***, and* $\hat{\psi}^{\sqcap} = \sqcap\{\hat{\psi} | \, \hat{\psi} \models \varphi\}$*, then* $\hat{\psi}_2^{\hat{\sqcap}} \circ \hat{\psi}_1^{\hat{\sqcap}} = \hat{\psi}^{\sqcap}$*.*

Proof. First, by Proposition 3.26 and Theorem 3.24 we have that $\hat{\psi}_2^{\hat{\sqcap}} \circ \hat{\psi}_1^{\hat{\sqcap}} \sqsupseteq \hat{\psi}^{\sqcap}$.

Second, to prove $\hat{\psi}_2^{\hat{\sqcap}} \circ \hat{\psi}_1^{\hat{\sqcap}} \sqsubseteq \hat{\psi}^{\sqcap}$, we first let **AVar** $= \{\alpha_i \, | \, i \geq 0\}$ and **TV** $=$ $\{i \, | \, i \geq 0\}$. Then we define the pair $(\hat{\psi}'_1, \hat{\psi}'_2)$ from $\hat{\psi}^{\sqcap}$ by

(1) $\forall i : \hat{\psi}'_1(\alpha_i) = \mathsf{min}(j \, | \, \hat{\psi}^{\sqcap}(\alpha_i) = \hat{\psi}^{\sqcap}(\alpha_j))$, where the function min returns a least number given a set of integers.

(2) $\forall i : \hat{\psi}'_2(i) = \hat{\psi}^{\sqcap}(\alpha_i)$.

The function $\hat{\psi}'_2$ is well-defined following from the condition (1). It is straightforward to verify that $\hat{\psi}'_2 \circ \hat{\psi}'_1 = \hat{\psi}^{\sqcap}$. We then have the following lemma.

Lemma 3.28 *For* $\hat{\psi}^{\sqcap} = \sqcap\{\hat{\psi} | \, \hat{\psi} \models \varphi\}$*, there is* $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \varphi$ *where* $(\hat{\psi}'_1, \hat{\psi}'_2)$ *is given as the above definition.*

Proof. The proof is a straightforward induction on φ. □

By Lemma 3.28 and the definition of $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}})$ we have that $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) \preceq (\hat{\psi}'_1, \hat{\psi}'_2)$. From Lemma 3.10, we conclude that $\hat{\psi}_2^{\hat{\sqcap}} \circ \hat{\psi}_1^{\hat{\sqcap}} \sqsubseteq \hat{\psi}'_2 \circ \hat{\psi}'_1 \sqsubseteq \hat{\psi}^{\sqcap}$ as desired. □

3.4 Concluding Remarks

This chapter has presented a basic inclusion constraint language. Similar to many other Datalog Solvers [NSN02, SSW94, WACL05], the universe of our constraint language consists of atomic values. In terms of set constraints, an atomic value is a term of arity 0, i.e. ground term. The choice of the constructs of the language guarantees the Moore family property that is normally required by static program analysis.

Based on our constraint language, we have also described a parameterized framework with which analysis designers can tune a system depending on his specific needs on performance and precision. As a result, this framework allows analysis designers to actively participate in the process of optimizing their analyses.

Finally our development based on the double-layer interpretation shows that the Moore family approach carries over to the description of more advanced internal representation using type variables. This forms a firm foundation for the solution calculated by the algorithm (specified in the next chapter).

Chapter 4

Constraint Solving

This chapter specifies an algorithm for the constraint solving of the inclusion constraint language presented in Chapter 3. The double-layer semantics in the previous chapter functions as a specification for the algorithm design in the current chapter. The complexity and correctness of the algorithm is subsequently studied.

As a running example we apply the parameterized framework on an intraprocedural reaching definitions analysis for a simple imperative language. We demonstrate the effect of applying unification on the analysis by tuning the constraint program and using our lifting strategy. A thorough study on how imprecision may arise is conducted and therefore in all of our benchmarks we show a good control on the level of precision. A comparative study between the Succinct Solver [NSN02] and our solver is also conducted. Part of the work was previously presented in [ZN08].

4.1 Design of Algorithm

The aim of our constraint solver is to calculate a least solution for a set of constraints. A worklist algorithm is described for computing the least solution of

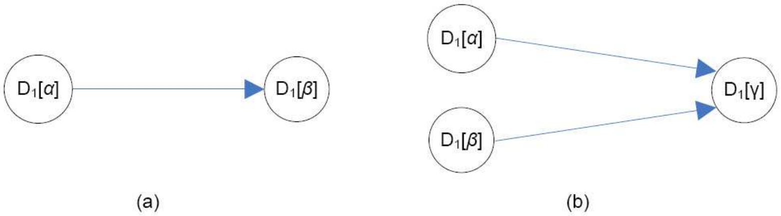


Figure 4.1: Graph representation of data flow: (a) for constraints of $\alpha \subseteq \beta, \alpha \setminus c \subseteq \beta$, and $\alpha \setminus (D) \subseteq \beta$; (b) for constraint $\alpha \cap \beta \subseteq \gamma$.

a constraint program. Referring to the example of constraint solving in [NNH99], we translate a constraint program into a collection of labeled nodes, which is called labeled cluster. Each node represents a type variable, and each label is decorated with the constraint that gives rise to it. Corresponding to the double-layer semantics presented in Chapter 3, the two data structures D_1 and D_2 is designed to associate nodes with analysis variables and constants with nodes respectively. For the reason of simplicity, the algorithm uses a set of natural numbers as type variables. Initially D_1 maps each analysis variable to a unique number and D_2 is initialized by the constraints of the form $c \subseteq \alpha$.

In order to build up a labeled cluster, we use a data structure E to record the list of labels (constructs) attached on each type variable. To be concrete, the constraints $\beta \subseteq \alpha, \alpha \setminus c \subseteq \beta$ and $\alpha \setminus (D) \subseteq \beta$ give rise to labels attached on the node $\mathsf{D}_1[\alpha]$; similarly, the constraint $\alpha \cap \beta \subseteq \gamma$ gives rise to two labels attached on the nodes $\mathsf{D}_1[\alpha]$ and $\mathsf{D}_1[\beta]$ respectively. These labels provide information to track the data flow from one node to another: whenever the data field of a node is enlarged, the algorithm retrieves the attached labels of the node to check if this change should be updated onto any relevant nodes. For instance, the label decorated by constraint $\alpha \subseteq \beta$ reflects the data flow from $\mathsf{D}_1[\alpha]$ to $\mathsf{D}_1[\beta]$. This is visualized as the graph (a) in Figure 4.1. The data flow of constraints $\alpha \setminus c \subseteq \beta, \alpha \setminus (D) \subseteq \beta$ and $\alpha \cap \beta \subseteq \gamma$ is also described in the figure following the same idea.

An equality constraint, however, never yields any label. This is because whenever two analysis variables are coalesced onto one type variable, they will always share the same data field and thus a label for equality constraint is not needed. As a result, when lifting is applied fewer labels are generated and the resulting labeled cluster becomes smaller. Furthermore, equality constraints helps to simplify a labeled cluster. For instance, suppose $\alpha = \gamma$, then it is possible to remove the labels corresponding to the constraints $\alpha \subseteq \gamma, \gamma \subseteq \alpha, \alpha \setminus c \subseteq \gamma$ and $\alpha \cap \beta \subseteq \gamma$.

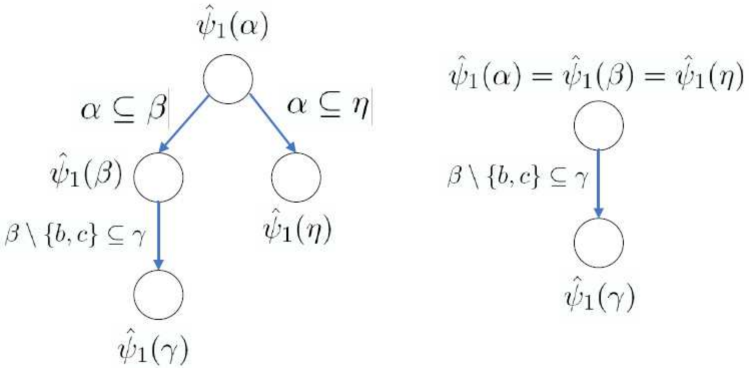


Figure 4.2: Graph representation of data flow: Example 3.7.

Example 4.1 Consider the constraint program of Example 3.7 again.

$$\{a,b,c\} \subseteq \alpha \ \wedge \ \{a,b\} \subseteq \beta \ \wedge \alpha \subseteq^{\dagger} \beta \ \wedge \ \beta \setminus \{b,c\} \subseteq \gamma \ \wedge \{a,c\} \subseteq \eta \ \wedge \ \alpha \subseteq^{\dagger} \eta$$

The data flow between nodes are visualized as the left graph of Figure 4.2. Now suppose we change the set-inclusion (marked with †) to equality, the resulting data flow graph is presented as the right graph of Figure 4.2 As the figure shows, the labels decorated by constraints $\alpha \subseteq \beta$, and $\alpha \subseteq \eta$ can be dispensed with in the lifted version of the constraint program and thereby simplifies the labeled cluster. □

Alternatively, a graph can be used to formulate a constraint program, i.e. a label could be considered as an edge which represents the data flow between nodes directly. The formulation of a labeled cluster is preferred here because it can be naturally implemented by the algorithm. More important, this formulation is more general when the pointer analysis is considered in Chapter 5 in which the basic constraint language is extended.

To be more specific consider the algorithm of Table 4.1 and 4.2. It operates on the following data structures.

- Two sets $\mathsf{U}, \mathsf{N} \subseteq \widehat{\mathbf{Constr}}$ in which U contains all equality constraints and N contains all the others;
- A data array $\mathsf{D}_1 : \mathbf{AVar} \to \mathbf{TV}$ that for each analysis variable returns a node;

- A data array $\mathsf{D}_2 : \mathbf{TV} \rightarrow \widehat{\mathbf{Const}}$ that for each node returns a set of tuples;
- An edge array $\mathsf{E} : \mathbf{TV} \rightarrow \mathbf{Constr}\ \mathit{list}$ that for each analysis variable returns a list of constraints from which the algorithm can detect a set of nodes to be updated and thereafter update the worklist W.
- A *worklist* $\mathsf{W} \subseteq \mathbf{AVar}\ \mathit{list}$, i.e. a list of analysis variables. The attached labels of the corresponding nodes of these analysis variables will be retrieved. By storing analysis variables instead of nodes, we keep the design of the algorithm as general as possible: we can safely dispense with the effect of unification on the elements of the worklist W by always looking up D_1 for the up-to-date information. Maintaining this generality becomes quite necessary whenever unification may happen during the iteration of the worklist algorithm, and that will be the case in Chapter 5.

The algorithm takes as input a pair of constraint lists (U, N), in which U contains all equality constraints and N all the others. Given a conjunction of constraints it is straightforward to generate the pair. The output of the algorithm is the pair $(\mathsf{D}_1, \mathsf{D}_2)$. We restrict ourselves to entities occurring in the constraints of interest: Let $\mathbf{AVar}_\star \subseteq \mathbf{AVar}$ and $\mathbf{TV}_\star \subseteq \mathbf{TV}$ be the finite sets of interest respectively.

Step 1 initializes the data structures used through the algorithm. Each analysis variable is assigned a unique type variable; the data field and the list of labels of each type variable are empty at the beginning. Step 2 implements a fast union-find data structure [Tar83] to coalesce equivalent analysis variables onto the designated type variables according to the given equality constraints. It first initializes three arrays needed only by unification. A set of type variables are unified and represented by a tree, which is implemented by the array A: for each node i, $\mathsf{A}[i]$ points to its parent if there is one or otherwise itself. The data structure H, which records the rank of each tree, is to balance a tree when two trees are merged into one.

The two important heuristics of the fast union/find data structure, which result in the almost linear-time boundary of the algorithm, are union-by-rank and path compression. The first one keeps the trees shallow as demonstrated in (a) of Figure 4.3: the tree with low rank is always merged into the high one. The second heuristic is to change the structure of a tree during a find operation by moving nodes closer to the root as demonstrated in Figure 4.4. This is implemented by three procedures: (1) the procedure unify combines trees and keeps the resulting tree's rank as small as possible; (2) the procedure find returns the root of a tree with the help of the procedure getRoot which conducts path-compression at the same time. The last loop of Step 2 updates D_1 according to the unification result.

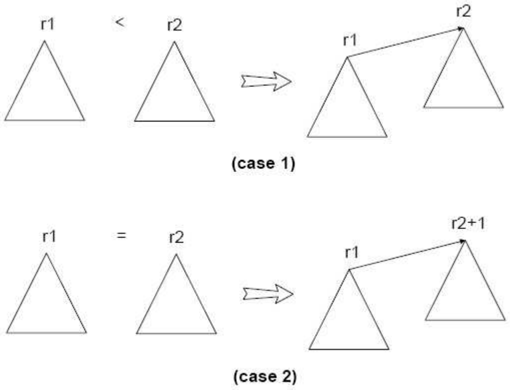


Figure 4.3: Tree with lower rank is always merged into that with higher rank. If two trees have same rank, the rank of new tree increases by one.

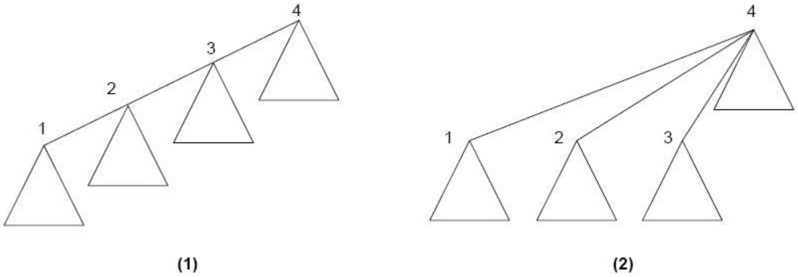


Figure 4.4: Path Compression: performing path-compression from node 1 to the root of the tree denoted as (1) results in the tree denoted as (2). Triangles represents subtrees.

The third step constructs a labeled cluster according to the given constraint or executes the initial assignments to D_2. For the constraint of the form $c \subseteq \alpha$, we use the procedure $\mathsf{add}(\alpha, c)$ to incorporates c into $\mathsf{D}_2[\mathsf{D}_1[\alpha]]$ and add α to the worklist W if constant c was not contained in $\mathsf{D}_2[\mathsf{D}_1[\alpha]]$. For other non-equality constraints, we make use of the result of unification to simplify the cluster by testing if two analysis variables are unified already. If that is the case, no label is added. Otherwise the label(s) are added as described before.

The fourth step keeps propagating data changes with respect to the labels attached to each analysis variable from the worklist W until it becomes empty. Note that for the current development, there is no unification during the iteration. Therefore, we can use D_1 directly instead of invoking find in Step 4.

```
INPUT :$(\mathsf{U}, \mathsf{N})$
OUTPUT :$(\mathsf{D}_1, \mathsf{D}_2)$

Step 1 : Initializing Data Structures
         $\mathsf{W} :=$ nil;
         for $\alpha_i$ in $\mathsf{AVar}_\star$ do
            $\mathsf{D}_1[\alpha_i] := i$;
            $\mathsf{D}_2[i] = \emptyset$;
            $\mathsf{E}[i] =$ nil;

Step 2 : Implementing fast union/find data structure
         for $i$ in $\mathbf{TV}_\star$ do
            $\mathsf{A}[i] := i$;
            $\mathsf{H}[i] := 0$;
         for $\alpha \subseteq \beta$ in $\mathsf{U}$ do $\mathsf{unify}(\mathsf{find}(\alpha), \mathsf{find}(\beta))$;
         for $\alpha \in \mathsf{AVar}_\star$ do $\mathsf{find}(\alpha)$;

Step 3 : Constructing the labeled cluster
         for $cc$ in $\mathsf{N}$ do
            case $cc$ of
               $c \subseteq \alpha$ :          $\mathsf{add}(\alpha, c)$;
               $\alpha \subseteq \beta$ :          if $\mathsf{D}_1(\alpha) \neq \mathsf{D}_1(\beta)$ then $\mathsf{E}[\mathsf{D}_1(\alpha)] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1(\alpha)]$);
               $\alpha \backslash c \subseteq \beta$ :     if $\mathsf{D}_1(\alpha) \neq \mathsf{D}_1(\beta)$ then $\mathsf{E}[\mathsf{D}_1(\alpha)] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1(\alpha)]$);
               $\alpha \backslash (D) \subseteq \beta$ : if $\mathsf{D}_1(\alpha) \neq \mathsf{D}_1(\beta)$ then $\mathsf{E}[\mathsf{D}_1(\alpha)] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1(\alpha)]$);
               $\alpha \cap \beta \subseteq \gamma$ :    if $\mathsf{D}_1(\alpha) \neq \mathsf{D}_1(\gamma)$ then $\mathsf{E}[\mathsf{D}_1(\alpha)] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1(\alpha)]$);
                                 if $\mathsf{D}_1(\beta) \neq \mathsf{D}_1(\gamma)$ then $\mathsf{E}[\mathsf{D}_1(\beta)] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1(\beta)]$);

Step 4 : Iteration
         While $\mathsf{W} \neq$ nil do
            $\gamma :=$ SELECT-FROM$(\mathsf{W})$;
            $t_e := \mathsf{E}[\mathsf{D}_1(\gamma)]$;
            for $cc$ in $t_e$ do
               case $cc$ of
                  $\alpha \subseteq \beta$ :          $\mathsf{add}(\beta, \mathsf{D}_2[\mathsf{D}_1(\alpha)])$;
                  (* standard set minus    *)
                  $\alpha \backslash c \subseteq \beta$ :     $\mathsf{add}(\beta, \mathsf{D}_2[\mathsf{D}_1(\alpha)] \backslash c)$;
                  (* overloaded set minus *)
                  $\alpha \backslash (D) \subseteq \beta$ : $\mathsf{add}(\beta, \mathsf{D}_2[\mathsf{D}_1(\alpha)] \backslash (D))$;
                  $\alpha \cap \beta \subseteq \gamma$ :    $\mathsf{add}(\gamma, \mathsf{D}_2[\mathsf{D}_1(\alpha)] \cap \mathsf{D}_2[\mathsf{D}_1(\beta)])$;
```

Table 4.1: Worklist Algorithm

```
procedure $\mathsf{add}(\alpha, c)$ is
  $i := \mathsf{find}(\alpha)$;
  if $\neg(c \subseteq \mathsf{D}_2[i])$ then $\mathsf{D}_2[i] := \mathsf{D}_2[i] \cup c$;
                           $\mathsf{W} := \{\alpha\} \cup \mathsf{W}$;

procedure $\mathsf{find}(\alpha)$ is
  $r = \mathsf{getRoot}(\alpha)$;
  $\mathsf{D}_1[\alpha] = r$;
  return $r$

procedure $\mathsf{unify}(m, n)$ is
  if $H[m] >= H[n]$ then $A[n] := m$
                           if $H[m] = H[n]$
                           then $H[m] := H[m] + 1$
                     else $A[m] := n$

procedure $\mathsf{getRoot}(m)$ is
  $p := A[m]$;
  if $p = m$ then return $m$
         else $A[m] := \mathsf{getRoot}(p)$
              $A[m]$
```

Table 4.2: Worklist Algorithm: Auxiliary Functions

We study the properties of the algorithm and have the following two theorems.

Theorem 4.2 *Given a clause φ the algorithm of Table 4.1 terminates and the result $(\mathsf{D}_1, \mathsf{D}_2)$ produced by the algorithm satisfies*

$$(\mathsf{D}_1, \mathsf{D}_2) = \hat{\sqcap}\{(\psi_1', \psi_2')|(\psi_1', \psi_2') \models_{\mathcal{T}} \varphi\}$$

Proof. We first prove that the algorithm always terminates. It is immediate that the steps 1, 2, and 3 terminate considering that the sets $\mathbf{AVar}_\star, \mathbf{TV}_\star, \mathsf{U}$ and N are finite. For Step 4, observe that for each type variable i the data $\mathsf{D}_2[i]$ never decreases and it can increase a finite number of times at most. For each analysis variable placed on the worklist a finite amount of calculation needs to be executed in order to remove the node from the worklist. This completes the first part of the proof.

To show the result calculated by the algorithm is a least solution, let (ψ_1', ψ_2')

be an estimate, such that $(\psi'_1, \psi'_2) \models_{\mathcal{T}} \varphi$. We then have the following invariant

$$\begin{array}{rl} \forall \alpha, \beta \in \mathbf{AVar}: & \mathsf{D}_1[\alpha] = \mathsf{D}_1[\beta] \Rightarrow \psi'_1(\alpha) = \psi'_1(\beta) \wedge \\ \forall \gamma \in \mathbf{AVar}: & \mathsf{D}_2[\mathsf{D}_1[\gamma]] \subseteq \psi'_2(\psi'_1(\gamma)) \end{array}$$

maintained everywhere in Step 4. It follows that $(\mathsf{D}_1, \mathsf{D}_2) \preceq (\psi'_1, \psi'_2)$ upon the completion of the algorithm by Lemma 3.10.

Next we show by contradiction that $(\mathsf{D}_1, \mathsf{D}_2)$ is indeed a solution for constraint program φ. Suppose there exists $cc \in \mathsf{U} \cup \mathsf{N}$ such that $(\psi_1, \psi_2) \models_{\mathcal{T}} cc$ does not hold.

If cc is the form $c \subseteq \alpha$ then the first case in the loop of Step 3 ensures that $c \subseteq \mathsf{D}_2[\mathsf{D}_1[\alpha]]$ and this is maintained throughout the algorithm; hence cc can not have this form.

If cc is the form $\alpha \subseteq \beta$, it must be the case that the final value of $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \neq \emptyset$ since otherwise $(\mathsf{D}_1, \mathsf{D}_2) \models_{\mathcal{T}} \alpha \subseteq \beta$ would hold. Now consider the last time $\mathsf{D}_2[\mathsf{D}_1(\alpha)]$ was modified and note that α was placed on the worklist at that time (by procedure add); since the final worklist is empty we must have considered the constraint $\alpha \subseteq \beta$ (which is in $E[\mathsf{D}_1(\alpha)]$) and updated $\mathsf{D}_2[\mathsf{D}_1[\beta]]$ accordingly; hence cc can not have this form either.

If cc is the form $\alpha = \beta$ then after the execution of Step 2, we can be sure that $\mathsf{D}_1[\alpha] = \mathsf{D}_1[\beta]$ and this is maintained throughout the algorithm; hence cc can not have this form.

If cc is the form $\alpha \setminus c \subseteq \beta$ then similar to the case of $\alpha \subseteq \beta$, $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \neq \emptyset$ since otherwise $(\mathsf{D}_1, \mathsf{D}_2) \models_{\mathcal{T}} \alpha \setminus c \subseteq \beta$ would hold. Now consider the last time $\mathsf{D}_2[\mathsf{D}_1[\alpha]]$ was modified and note that α was placed on the worklist at that time (by procedure add); since the final worklist is empty we must have considered the constraint $\alpha \setminus c \subseteq \beta$ (which is in $E[\mathsf{D}_1[\alpha]]$) and updated $\mathsf{D}_2[\mathsf{D}_1[\beta]]$ eventually; hence cc can not have this form.

Similarly, we can show that cc can not have the form of $\alpha \setminus (D) \subseteq \beta$ or $\alpha \cap \beta \subseteq \gamma$. This completes the proof. □

Theorem 4.3 *The time complexity of the algorithm in Table 4.1 and 4.2 is $O(n^3)$ where n is the size of a constraint program.*

Proof. First note that the number of analysis variables, type variables, and constraints are bound to $O(n)$. It is straightforward to show that the time complexity of Step 1 and 3 are both linearity. The complexity of Step 2 is directly from the result of Tarjan's study in [Tar83]. It is almost linear because the inverse Ackermann's function α grows very slow and for all practical purposes $\alpha(2n, n)$ is a constant not larger than four.

To analyze the complexity of Step 4, it is important to make it clear that what the complexity of the operations upon set of tuples (constant) actually is. In the actual implementation, each tuple is represented as a bit in a bit-vector. Since the number of tuples is bound to $O(n)$, the length of the bit-vector is also bound to $O(n)$; thus the set operations, e.g. set union and set intersection, are over bit-vectors of the length $O(n)$ and have linear time complexity.

Next observe that there are $O(n)$ labels generated from a constraint program of size $O(n)$ and each label can be retrieved at most $O(n)$ times as there are $O(n)$ nodes. Therefore let n_i be the number of labels bound to the node i, we have that the time complexity of iteration is $O(\Sigma_{i \in \mathbf{TV}_\star}(n \cdot n_i \cdot n)) = O(n^3)$ where the first n is the upper bound of traversals on each edge and the second is the time of set operations. □

Although the worst case complexity of the algorithm is cubic, the use of unification on equality constraints over analysis variables reduces the problem space and thus is expected to speed up the constraint solving. The rest of this chapter will demonstrate the use of the parameterized framework and study the effects of applying unification on performance and precision with a working example, reaching definitions analysis.

Note that the algorithm does not specify any worklist strategy to be used. In the implementation of the solver algorithm, different strategies could be tried on the analyses of interest. We shall discuss the effect of worklist strategy on the performance of our solver during our study on working examples.

4.2 Case Study: Reaching Definitions Analysis

In this section we present a simple C-like imperative language and specify a reaching definitions analysis for the language.

Definition 4.4 (Reaching Definitions Analysis) For each program point, which assignments may have been made on the target program variable without an intervening assignment during the execution of the program.

With the framework we have described, we apply the lifting strategy and study how the imprecision can be controlled and how much improvement on performance can be achieved from using unification. The analysis is specified both in our constraint language and in the alternation-free fragment of *Least Fixpoint Logic* (ALFP) and thereby is implemented by our constraint solver and the Succinct Solver respectively. The results are then evaluated from the aspects of time, space and precision. The fact that both of the two solvers are implemented in New Jersey SML makes the comparison more reliable.

We assume some countable set of program variables, $x, y \in \mathbf{ProgVar}$, rational numbers, $r \in \mathcal{R}$, and labels, $\ell \in \mathsf{Lab}$. The statement $S \in \mathbf{Stmt}$ of the C-like language is specified as the abstract syntax in Table 4.3.

a	$::=$	$x \mid r \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid a_1/a_2$
b	$::=$	$true \mid false \mid neg\ b \mid b_1\ and\ b_2 \mid b_1\ or\ b_2 \mid$
		$a_1 \ < \ a_2 \mid a_1 \ > \ a_2 \mid a_1 \ = \ a_2 \mid a_1 \ \neq \ a_2$
exp	$::=$	$a \mid b$
S	$::=$	$[x := a]^\ell \mid [\mathsf{skip}]^\ell \mid [\mathsf{exp}]^\ell \mid S_1; S_2 \mid \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \mid$
		$\mathsf{while}\ [b]^\ell\ \mathsf{do}\ S$

Table 4.3: A Simple Imperative Language.

There are three kinds of elementary statements, i.e. assignments, skips and expressions, and three kinds of complex statements, i.e. compositional statements, if-branches and while-loop. We assume each elementary statement is assigned a *unique* label in order to avoid unnecessary imprecision of an analysis. If it is clear in context, we may call a statement by its label directly for the reason of simplicity.

4.2.1 Initial Label and Final Labels

Data flow analyses usually use some operations on programs. These operations are designed to extract basic information of a program to be analyzed. Before specifying our analysis, we introduce two such operations: the first is $\mathsf{init} : \mathbf{Stmt} \rightarrow \mathbf{Lab}$, which given a statement returns the *initial label* of it; the second operation is $\mathsf{final} : \mathbf{Stmt} \rightarrow \mathcal{P}(\mathbf{Lab})$, which given a statement returns the set of *final labels* in a statement. The definition of the two operations are listed in Table 4.4 and 4.5 as below.

$$
\begin{aligned}
\mathsf{init}([x := a]^{\ell}) &= \ell \\
\mathsf{init}([skip]^{\ell}) &= \ell \\
\mathsf{init}([exp]^{\ell}) &= \ell \\
\mathsf{init}(S_1; S_2) &= \mathsf{init}(S_1) \\
\mathsf{init}(\text{if } [b]^{\ell} \text{ then } S_1 \text{ else } S_2) &= \ell \\
\mathsf{init}(\text{while } [b]^{\ell} \text{ do } S) &= \ell
\end{aligned}
$$

Table 4.4: Initial Function.

$$
\begin{aligned}
\mathsf{final}([x := a]^{\ell}) &= \{\ell\} \\
\mathsf{final}([skip]^{\ell}) &= \{\ell\} \\
\mathsf{final}([exp]^{\ell}) &= \{\ell\} \\
\mathsf{final}(S_1; S_2) &= \mathsf{final}(S_2) \\
\mathsf{final}(\text{if } [b]^{\ell} \text{ then } S_1 \text{ else } S_2) &= \mathsf{final}(S_1) \cup \mathsf{final}(S_2) \\
\mathsf{final}(\text{while } [b]^{\ell} \text{ do } S) &= \{\ell\}
\end{aligned}
$$

Table 4.5: Final Function.

Note that for the while-loop the initial label is ℓ and the set of final labels is $\{\ell\}$, i.e. the entry point and exit point of while loop are same. This is because the while-loop terminates immediately after the test has evaluated to false.

4.2.2 Analysis Using Inclusion Constraints

We specify a reaching definitions analysis in our inclusion constraint language. Two caches are used for recording the analysis results of each program point:

$$RD_{\circ}, RD_{\bullet} : \mathbf{Lab} \rightarrow \mathcal{P}(\mathbf{ProgVar} \times \mathbf{Lab})$$

where $\circ$ and $\bullet$ denote the entry and exit point of a elementary statement respectively. Following the tradition of the Flow Logic specification, the judgement of

$$
\begin{array}{lllll}
[\mathrm{ass}] & (RD_\circ, RD_\bullet) \models [x := e]^\ell & \text{iff} & \{(x,\ell)\} \subseteq RD_\bullet(\ell) \wedge & \\
 & & & RD_\circ(\ell) \setminus (x,?) \subseteq RD_\bullet(\ell) & \\
[\mathrm{skip}] & (RD_\circ, RD_\bullet) \models [skip]^\ell & \text{iff} & RD_\circ(\ell) \subseteq RD_\bullet(\ell) & \text{(i)} \\
[\mathrm{exp}] & (RD_\circ, RD_\bullet) \models [exp]^\ell & \text{iff} & RD_\circ(\ell) \subseteq RD_\bullet(\ell) & \text{(ii)} \\
[\mathrm{comp}] & (RD_\circ, RD_\bullet) \models S_1; S_2 & \text{iff} & (RD_\circ, RD_\bullet) \models S_1 \wedge & \\
 & & & (RD_\circ, RD_\bullet) \models S_2 \wedge & \\
 & & & \wedge_{\forall \ell \in \mathsf{final}(S_1)} RD_\bullet(\ell) \subseteq RD_\circ(\mathsf{init}(S_2)) & \text{(iii)} \\
[\mathrm{if}] & \multicolumn{4}{l}{(RD_\circ, RD_\bullet) \models \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2} \\
 & & \text{iff} & (RD_\circ, RD_\bullet) \models S_1 \wedge & \\
 & & & (RD_\circ, RD_\bullet) \models S_2 \wedge & \\
 & & & (RD_\circ, RD_\bullet) \models b \wedge & \\
 & & & RD_\bullet(\ell) \subseteq RD_\circ(\mathsf{init}(S_1)) \wedge & \text{(iv)} \\
 & & & RD_\bullet(\ell) \subseteq RD_\circ(\mathsf{init}(S_2)) & \text{(v)} \\
[\mathrm{wh}] & \multicolumn{4}{l}{(RD_\circ, RD_\bullet) \models \text{while } [b]^\ell \text{ do } S} \\
 & & \text{iff} & (RD_\circ, RD_\bullet) \models S \wedge & \\
 & & & (RD_\circ, RD_\bullet) \models b \wedge & \\
 & & & RD_\bullet(\ell) \subseteq RD_\circ(\mathsf{init}(S)) \wedge & \text{(vi)} \\
 & & & \wedge_{\forall \ell' \in \mathsf{final}(S)} RD_\bullet(\ell') \subseteq RD_\circ(\ell) & \text{(vii)}
\end{array}
$$

Table 4.6: Reaching Definitions Analysis: Inclusion Constraint Language.

the analysis has the form

$$(RD_\circ, RD_\bullet) \models S \text{ iff } \varphi$$

The judgement is true if and only if an analysis result $(RD_\circ, RD_\bullet)$ correctly describes S. It associates a program S with the constraint program φ. The implementation is, therefore, to calculate a least solution for the constraint program φ generated for a given program S with respect to the analysis specification.

The analysis specification using our constraint language is presented in Table 4.6 and the liftable constraints are numbered.

In the constraints for assignment $[x := e]^l$, all the assignments on the program variable x are removed using the non-standard set minus operation and the new assignment then adds to the cache of the exit of the statement. For the statements of skip and expression, we simply copy the reaching definition information from the entry of the statement to the exit of the statement since there is no side effect occurring between the entry and exit of each of the two statements.

Remark 4.5 *This example shows that a careful choice of the constraints used in an analysis could have significant effect on performance. Note that the use of the non-standard set minus operation generates constraints of constant size. It not only gives a succinct specification, but decreases the asymptotic complexity of the analysis: if a standard set minus operation were used, the constraints generated could be linear, i.e.* $\wedge_{\forall \ell' \in \mathbf{Lab}} RD_{\circ}(\ell) \setminus (x, \ell') \subseteq RD_{\bullet}(\ell)$*; and once these constraints are generated, it is difficult for a solver to do any optimization. Therefore, it could be a good practice for analysis designers to be clear what is the complexity of their analysis and how this happens. Thereby they may have better chance to make a good use of features provided by a solver and improve the usability of their analysis.*

Remark 4.6 ***Correctness of the analysis.*** *Once an analysis is specified, it is necessary to show that the analysis result is correct for the programs to be analyzed. For example, the correctness result for the reaching definitions analysis should express that the sets of reaching definition information computed by the analysis are correct throughout the computation. To give formal proof, one would need some formal semantics for the imperative language and reason the correctness of the analysis holds for each statement. A small step semantics, e.g. Structural Operational Semantics, is preferred because this kind of semantics allows to reason about intermediate stages in a program execution and to handle non-terminating programs, i.e. some endless loop in a program. In this dissertation, we focus on the implementation of program analyses instead of the discussion of the correctness of program analyses. Considering the language studied is standard, we have alternatively conducted an inspection on each specification in order to ensure the correctness of the analysis with respect to the meaning of each statement. For an extensive introduction on the formal proof of the correctness of program analyses, see [NNH99].*

4.2.2.1 Heuristics about Tuning Constraints

In the rest of this section, we conduct a heuristic study on when and how imprecision may be incurred by the use of unification. Knowing these heuristics is useful for analysis designers who want to achieve a good balance between performance and precision.

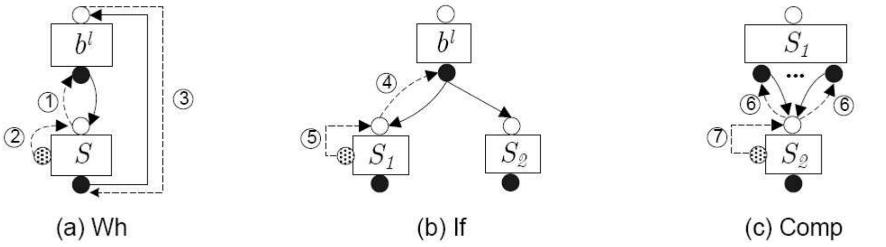


Figure 4.5: Graph Representation of Data Flow for [wh], [if] and [comp].

First observe that the constraints of (i) and (ii) can be changed to equality constraints without causing any imprecision because each label is unique and there is no data updated between the entry and exit of [skip] and [exp] statements. The cases of [comp], [if], and [wh] are more complex and we analyze the case [wh] first because it helps to understand some tricky situation of [if] and [comp]. To be illustrative, we simulate the flow of data by the graphs in Figure 4.5 where $\circ$ and $\bullet$ denote the entry and exit point(s) respectively, the square represents statement(s), and the arrowed lines denote the direction of data flow.

At the beginning suppose only set-inclusion constraints are used and ignore the dashed lines in (a). The remaining part basically denotes that the information goes through the test $[b]^l$, then flows through S where the information may be updated, and finally goes back to the entry of the test. Consider lifting the constraint (vi) which is represented by adding the dashed line labeled 1. Checking if the change preserves precision then amounts to verifying if the entry of S has no more data than the exit of $[b]^l$. This is the case if the first elementary statement, say S_t, of S is not a while-loop because no more reaching definition could be updated under such a situation. Otherwise as shown by the dashed line labeled 2, some updated reaching definition information, which may happen inside S_t, would flow back to the entry of S from the exit of S_t, denoted by a dotted circle on the side of a square. Last the new reaching definition information reaches the exit and entry of $[b]^l$ (assuming lifting is applied on (vi)).

Now there are two possibilities: (1) This information is not further updated by any assignment in the rest of S. Using equality at (vi), as a result, would give no more data to the exit of $[b]^l$ than before considering the circle of the data formed by while-loop, i.e. no imprecision occurs. (2) Otherwise some analysis variable assigned in S_t must have been re-assigned later in S and instead of removing the former assignment information both of them are kept at the entry and exit of $[b]^l$ and hence imprecision happens.

	Solution (before lifting)	Solution (after lifting)
$RD_\circ(\ell_1)$	$\{(x,*)\}$	$\{(x,*)\}$
$RD_\bullet(\ell_1)$	$\{(x,\ell_1)\}$	$\{(x,\ell_1)\}$
$RD_\circ(\ell_2)$	$\{(x,\ell_1),(x,\ell_3)\}$	$\{(x,\ell_1),(x,\ell_3)\}$
$RD_\bullet(\ell_2)$	$\{(x,\ell_1),(x,\ell_3)\}$	$\{(x,\ell_1),(x,\ell_3)\}$
$RD_\circ(\ell_3)$	$\{(x,\ell_1),(x,\ell_3)\}$	$\{(x,\ell_1),(x,\ell_3)\}$
$RD_\bullet(\ell_3)$	$\{(x,\ell_3)\}$	$\{(x,\ell_1)^\dagger,(x,\ell_3)\}$

Table 4.7: Solutions for the constraints.

Finally changing constraint (vii) to equality is represented as adding a dashed line 3. It means that any reaching definition information available at the entry of test should also be available at the exit of S. Intuitively, the reaching definition information at the entry of test may be updated inside S whereas using equality at (vii) may introduce extra false positives.

Similarly lifting the constraint (iv) and (v) of [if] maintains precision if S_1 does not begin with a while-loop and otherwise may decrease precision at the exit $RD_\bullet(l)$. For the case [comp] first observe that S_1 may have multiple exits and thus lifting all constraints of (iii) results in unifying the data of these exits and hence decreases precision. On the other hand, if S_2 starts with a loop, we may also have more false positives as argued in the cases [if] and [wh].

Notation 4.7 *We distinguish between the exit point of the first atomic statement S_t and that of the whole statement S in order to convenient the explanation but still notice that they could be the same.*

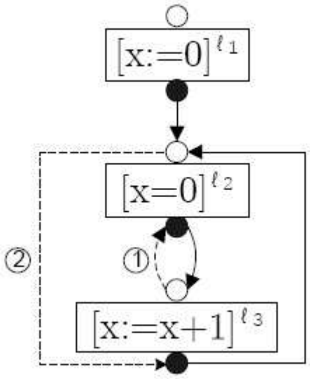
As discussed above, the tricky case is the while-loop statement which is the main cause of extra false positives. To better understand how imprecision occurs in the case [wh], we therefore consider the following example programs written in our imperative language.

Example 4.8 The first example program is

$$[x := 0]^{\ell_1};\ \text{while}\ [x = 0]^{\ell_2}\ \text{do}\ [x := x + 1]^{\ell_3}$$

The constraints and the data flow are described as Figure 4.6.

We here only consider to change the set-inclusion generated for the while-loop and mark them with †. The solutions before and after lifting the constraints are summarized in Table 4.7, in which "*" is used for uninitialized variables.



$$\begin{aligned}
&\mathsf{RD}_\circ(\ell_1)\backslash(x,?) \subseteq \mathsf{RD}_\bullet(\ell_1) \wedge\\
&\{(x,\ell_1)\} \subseteq \mathsf{RD}_\bullet(\ell_1) \wedge\\
&\mathsf{RD}_\bullet(\ell_1) \subseteq \mathsf{RD}_\circ(\ell_2) \wedge\\
&\mathsf{RD}_\circ(\ell_2) \subseteq \mathsf{RD}_\bullet(\ell_2) \wedge\\
&\mathsf{RD}_\bullet(\ell_2) \subseteq^\dagger \mathsf{RD}_\circ(\ell_3) \wedge\\
&\mathsf{RD}_\circ(\ell_3)\backslash(x,?) \subseteq \mathsf{RD}_\bullet(\ell_3) \wedge\\
&\{(x,\ell_3)\} \subseteq \mathsf{RD}_\bullet(\ell_3) \wedge\\
&\mathsf{RD}_\bullet(\ell_3) \subseteq^\dagger \mathsf{RD}_\circ(\ell_2)
\end{aligned}$$

Figure 4.6: Constraints and Graph Representation: Example 4.8.

One extra false positive occurs as marked with †. This is introduced by the lifted constraint $RD_\bullet(\ell_3) = RD_\bullet(\ell_2)$: the unification adds the pair (x, ℓ_1) to the exit of the statement ℓ_3, which would be removed for set-inclusion, as denoted by the dashed line 3 in Figure 4.6. This shows that lifting constraint (vii) of Table 4.6 causes extra false positive as long as there is any assignment in the loop. However, lifting the constraint $RD_\bullet(\ell_2) \subseteq RD_\circ(\ell_3)$ causes no loss in precision for the program considered. □

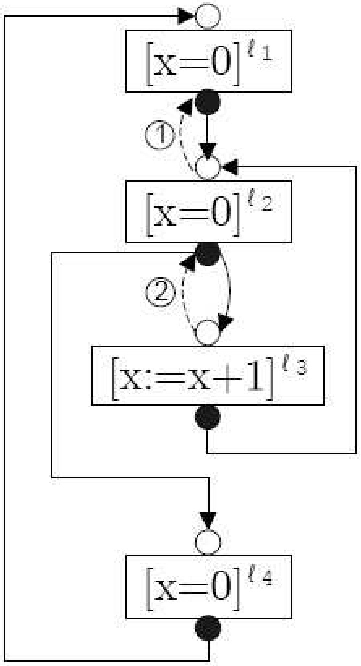
Example 4.9 Consider the program:

$$\text{while } [x \neq 0]_1^\ell \text{ do (while } [x = 1]_2^\ell \text{ do } x := x + 1;\ \text{x:=0)}$$

The constraints and the data flow are shown as Figure 4.7. In this example, we consider to lift constraints (vi) of Table 4.6, which are marked with † in Figure 4.7. As argued in Example 4.8, lifting constraints (vii) of Table 4.6 will cause extra false positives because of the assignments in the while-loops.

As demonstrated in Table 4.8, changing constraint $RD_\bullet(\ell_1) \subseteq RD_\circ(\ell_2)$ to an equality constraint would add the assignment of the inner loop to the exit of ℓ_1 as denoted by the dashed line 1 and thereby cause an extra false positive. However lifting constraint $RD_\bullet(\ell_2) \subseteq RD_\circ(\ell_3)$ keeps the level of precision because there is no new assignment between the two program points. □

To summarize the discussion of current subsection, we would initially try lifting all the constraints from (i) to (vi), and keep the flexibility of changing back to



$$\begin{aligned}
&\mathsf{RD}_\circ(\ell_1) \subseteq \mathsf{RD}_\bullet(\ell_1) \wedge\\
&\mathsf{RD}_\bullet(\ell_1) \subseteq^\dagger \mathsf{RD}_\circ(\ell_2) \wedge\\
&\mathsf{RD}_\circ(\ell_2) \subseteq \mathsf{RD}_\bullet(\ell_2) \wedge\\
&\mathsf{RD}_\bullet(\ell_2) \subseteq^\dagger \mathsf{RD}_\circ(\ell_3) \wedge\\
&\mathsf{RD}_\circ(\ell_3)\backslash(\mathsf{x},?) \subseteq \mathsf{RD}_\bullet(\ell_3) \wedge\\
&\{(\mathsf{x},\ell_3)\} \subseteq \mathsf{RD}_\bullet(\ell_3) \wedge\\
&\mathsf{RD}_\bullet(\ell_3) \subseteq \mathsf{RD}_\circ(\ell_2) \wedge\\
&\mathsf{RD}_\bullet(\ell_2) \subseteq \mathsf{RD}_\circ(\ell_4) \wedge\\
&\mathsf{RD}_\circ(\ell_4)\backslash(\mathsf{x},?) \subseteq \mathsf{RD}_\bullet(\ell_4) \wedge\\
&\{(\mathsf{x},\ell_4)\} \subseteq \mathsf{RD}_\bullet(\ell_4) \wedge\\
&\mathsf{RD}_\bullet(\ell_4) \subseteq \mathsf{RD}_\circ(\ell_1)
\end{aligned}$$

Figure 4.7: Constraints and Graph Representation: Example 4.9.

	Solution (before lifting)	Solution (after lifting)
$RD_\circ(\ell_1)$	$\{(x,*),(x,\ell_4)\}$	$\{(x,*),(x,\ell_4)\}$
$RD_\bullet(\ell_1)$	$\{(x,*),(x,\ell_4)\}$	$\{(x,*),(x,\ell_3)^\dagger,(x,\ell_4)\}$
$RD_\circ(\ell_2)$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$
$RD_\bullet(\ell_2)$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$
$RD_\circ(\ell_3)$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$
$RD_\bullet(\ell_3)$	$\{(x,\ell_3)\}$	$\{(x,\ell_3)\}$
$RD_\circ(\ell_4)$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$	$\{(x,*),(x,\ell_3),(x,\ell_4)\}$
$RD_\bullet(\ell_4)$	$\{(x,\ell_4)\}$	$\{(x,\ell_4)\}$

Table 4.8: Solutions for the constraints.

set-inclusion for the constraints of last five cases when necessary. According to our study, lifting the constraint (vii), however, is very likely to decrease precision: whether or not to change it depends on the specific consideration on the tradeoff between performance and precision; if the performance of some benchmarks is improved significantly and the precision is just acceptable, one

$$
\begin{array}{lcl}
t & ::= & a \mid u \mid f(t_1, \cdots, t_k) \\
p & ::= & R(t_1, \cdots, t_k) \mid \neg R(t_1, \cdots, t_k) \mid \\
 & & t_1 = t_2 \mid t_1 \neq t_2 \mid \\
 & & p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \exists u : p \mid \forall u : p \\
cl & ::= & R(t_1, \cdots, t_k) \mid 1 \mid cl_1 \wedge cl_2 \mid p \Rightarrow cl \mid \forall u : cl
\end{array}
$$

Table 4.9: Syntax of the Alternation-free Least Fixed Point logic

may still choose to lift the constraint (vii).

4.2.3 Analysis Using ALFP Logic

In this subsection, we give a brief introduction to the Alternation-free Least Fixed Point logic (ALFP) and the Succinct Solver. An reaching definitions analysis is then specified and its asymptotic complexity is studied accordingly.

4.2.3.1 ALFP Logic and the Succinct Solver

The ALFP logic is a fragment of first order predicate logic. The grammar of an ALFP formula cl is defined in Table 4.9, in which a term t consists of a finite set of constant symbols a, a fixed countable set of variables u, a finite set of function name f. We also write R for a finite ranked alphabet of predicate symbols, and p for pre-conditions of ALFP formulae.

Given a non-empty and countable universe $\mathcal{U}$ over the ground terms, the semantics is defined in terms of two satisfaction relations:

$$(\rho, \sigma) \models p \text{ and } (\rho, \sigma) \models cl$$

where ρ is an interpretation of predicate symbols and σ is an interpretation of terms. These satisfaction relations are defined in the standard way and the rules of the semantics are summarized in Table 4.10.

Whenever a clause cl is closed, i.e. each free variable in cl is fixed by a given interpretation σ_0, the set of interpretations $\{\rho|(\rho, \sigma_0) \models cl\}$ satisfies the Moore

Rules for pre-conditions:		
$(\rho, \sigma) \models R(u_1, \cdots, u_k)$	iff	$(\sigma(u_1), \cdots, \sigma(u_k)) \in \rho(R)$
$(\rho, \sigma) \models \neg R(u_1, \cdots, u_k)$	iff	$(\sigma(u_1), \cdots, \sigma(u_k)) \notin \rho(R)$
$(\rho, \sigma) \models p_1 \wedge p_2$	iff	$(\rho, \sigma) \models p_1$ and $(\rho, \sigma) \models p_2$
$(\rho, \sigma) \models p_1 \vee p_2$	iff	$(\rho, \sigma) \models p_1$ or $(\rho, \sigma) \models p_2$
$(\rho, \sigma) \models \exists u : p$	iff	$(\rho, \sigma[u \mapsto a]) \models p$ for some $a \in \mathcal{U}$
$(\rho, \sigma) \models \forall u : p$	iff	$(\rho, \sigma[u \mapsto a]) \models p$ for all $a \in \mathcal{U}$
Rules for clauses:		
$(\rho, \sigma) \models R(u_1, \cdots, u_k)$	iff	$(\sigma(u_1), \cdots, \sigma(u_k)) \in \rho(R)$
$(\rho, \sigma) \models 1$	iff	*always*
$(\rho, \sigma) \models cl_1 \wedge cl_2$	iff	$(\rho, \sigma) \models cl_1$ and $(\rho, \sigma) \models cl_2$
$(\rho, \sigma) \models p \Rightarrow cl$	iff	$(\rho, \sigma) \models cl$ whenever $(\rho, \sigma) \models p$
$(\rho, \sigma) \models \forall u : cl$	iff	$(\rho, \sigma[u \mapsto a]) \models cl$ for all $a \in \mathcal{U}$

Table 4.10: Semantics of the Alternation-free Least Fixed Point logic

family property [NSN02].

The *Succinct Solver*, which uses the (ALFP) logic as the specification logic, adopts former insights of solver technologies [CH92, FS99, FS98a], including the use of recursion, continuations, prefix tree and memorization. The solver achieves the best known theoretical bounds for Datalog solvers [NSN02]. Because of the expressiveness of ALFP logic, the solver has been used for the implementation of a variety of analyses [NNB02, ZN06, BBD$^+$05].

Upon to the asymptotic complexity, the solver computes a least solution of a ALFP formula *cl* (with respect to an interpretation σ_0 of the constant symbols) in the time of

$$O(\#\rho + N^\tau \cdot n)$$

where $\#\rho$ is the sum of cardinalities of predicates $\rho(R)$, N is the size of the universe, n is the size of *cl*, and τ is the maximal nesting depth of quantifiers in *cl* [NSN02].

The notion of stratification is used by the Succinct Solver when negations present in pre-conditions in order to ensure the solvability of clauses. This is, however, not relevant in the context of this dissertation. For further information about the Succinct Solver, see [NSN02].

4.2.3.2 Reaching Definitions Analysis in ALFP Logic

To compare the performance of our solver with that of the Succinct Solver, we also specify a reaching definitions analysis for our imperative language in ALFP logic. The two components of the estimate are redefined as:

$$RD_\circ, RD_\bullet \subseteq \mathbf{Lab} \to \mathbf{Lab}$$

A predicate $RD_\circ(\ell_1, \ell_2)$ or $RD_\bullet(\ell_1, \ell_2)$ estimates that some variable defined at ℓ_1 reaches ℓ_2 through some path. In order to get the complete reaching definition information, i.e. which variable is defined at ℓ_1, one may need a table that maps each label to either a program variable x defined at ℓ_1 or NULL. Looking up the table takes constant time and building up the table takes linear time (by scanning a program to be analyzed).

The judgement of the analysis has the form

$$(RD_\circ, RD_\bullet) \models S \text{ iff } cl$$

that is quite similar as before except that the judgement now associates each program with a ALFP formulae cl. The implementation of the analysis is then to input the clause cl (generated according to the analysis specification) into the Succinct Solver. The analysis specification in ALFP logic is specified in Table 4.11.

For each program variable x, we introduce the auxiliary predicate DEF_x to record where x is (re)defined. When the reaching definition information is copied from entry to exit in the analysis for assignment statement, i.e. $\forall u : RD_\circ(u, l) \land \neg DEF_x(u) \Rightarrow RD_\bullet(u, l)$, the predicate is used as a precondition to remove any pairs that can not reach the exit of the assignment. Intuitively, this has the exact same meaning as the corresponding inclusion constraint in Table 4.6, i.e. $RD_\circ(\ell) \setminus (x, ?) \subseteq RD_\bullet(\ell)$.

The analysis implemented by the Succinct Solver has the complexity $O(n^2)$ because the maximum depth of the universal quantification is only 1. This achieves the best theoretical worst case complexity using the Succinct Solver as far as we know.

$$
\begin{array}{lll}
(RD_\circ, RD_\bullet) \models [x := e]^l & \text{iff} & RD_\bullet(l,l) \wedge DEF_x(l) \wedge \\
 & & \forall u : RD_\circ(u,l) \wedge \neg DEF_x(u) \Rightarrow RD_\bullet(u,l) \\
(RD_\circ, RD_\bullet) \models [skip]^l & \text{iff} & \forall u : RD_\circ(u,l) \Rightarrow RD_\bullet(u,l) \\
(RD_\circ, RD_\bullet) \models [e]^l & \text{iff} & \forall u : RD_\circ(u,l) \Rightarrow RD_\bullet(u,l) \\
(RD_\circ, RD_\bullet) \models S_1; S_2^l & \text{iff} & (RD_\circ, RD_\bullet) \models S_1 \wedge \\
 & & (RD_\circ, RD_\bullet) \models S_2 \wedge \\
 & & \wedge_{\forall l \in \mathrm{final}(S_1)} \forall u : RD_\bullet(u,l) \Rightarrow RD_\circ(u, \mathrm{init}(\mathrm{S}_2)) \\
\multicolumn{3}{l}{\quad (RD_\circ, RD_\bullet) \models \text{if } [b]^l \text{ then } S_1 \text{ else } S_2} \\
 & \text{iff} & (RD_\circ, RD_\bullet) \models S_1 \wedge \\
 & & (RD_\circ, RD_\bullet) \models S_2 \wedge \\
 & & (RD_\circ, RD_\bullet) \models b \wedge \\
 & & \forall u : RD_\bullet(u,l) \Rightarrow RD_\circ(u, \mathrm{init}(S_1)) \wedge \\
 & & \forall u : RD_\bullet(u,l) \Rightarrow RD_\circ(u, \mathrm{init}(S_2)) \\
\multicolumn{3}{l}{\quad (RD_\circ, RD_\bullet) \models \text{while } [b]^l \text{ do } S} \\
 & \text{iff} & (RD_\circ, RD_\bullet) \models S \wedge \\
 & & (RD_\circ, RD_\bullet) \models b \wedge \\
 & & \forall u : RD_\bullet(u,l) \Rightarrow RD_\circ(u, \mathrm{init}(S)) \wedge \\
 & & \wedge_{\forall l' \in \mathrm{final}(S)} \forall u : RD_\bullet(u,l') \Rightarrow RD_\circ(u,l)
\end{array}
$$

Table 4.11: Reaching Definitions Analysis in ALFP.

4.3 Experimental Study

4.3.1 Methodology

We design two groups of benchmarks: one group is called representative programs, which implement 8 mathematical algorithms and one application; another group is called scalable program, which can be as large as we want so that the scalability of our solver and the Succinct Solver can be evaluated. The reaching definitions analysis is accordingly conducted on these benchmarks. We apply the parameterized framework on the inclusion constraints generated. The comparison is made between the result before and after lifting inclusion constraints for our solver, and between the Succinct Solver and our inclusion solver. The fact that both of the two solvers are implemented in New Jersey SML makes the comparison more reliable.

All the benchmarks are run on a 2.0 GHz processor with 1.5 GB of memory under Windows XP SP2. Each experiment is repeated three times and the average time and minimum memory consumption are reported. The reason of choosing the minimum memory consumption is based on the observation that the mechanism of the garbage collection of New Jersey SML compiler consumes larger memory than what is actually needed, and thus the minimum memory consumption we report are the ones closest to the actual values.

Regarding to the worklist strategy, our solver adopts the last-in-fist-out (LIFO) to prioritize the member of the worklist. New strategy will be introduced when more applications are considered in the following chapters.

4.3.2 Benchmarks: Representative Programs

Eight representative programs are used to evaluate the effect of using unification on our constraint solver: fibonacci is an algorithm calculating fibonacci numbers; isPrime verifies if a given number is a prime number or not; lcm returns the least common multiplier given two natural numbers; ext_gcd calculates two natural numbers' greatest common dividend; newtonIter is an algorithm for computing the square root of a number via the recurrence equation; wlfIter is simply another algorithm for computing the square root of a number; sum computes the sum of two natural numbers; log conducts the logarithm operation given a base and a number; calculator is an application which given proper arguments conducts all kinds of mathematical calculations.

All the time performance are measured in milliseconds (ms.). The improvement of time, represented as $\triangle T$ is in percent %. Besides evaluating the constraint programs using only set-inclusion constraints (called set-inclusion version), we tune the programs by lifting the constraints generated at (i) through (vi) (called equality version) as suggested in Subsection 4.3.2. Because we have done a thorough study in Subsection on where and how imprecision may occur, we furthermore make a fine-tuned constraint program for each benchmark (called enhanced equality version), in which we change back to set-inclusion constraints for the equality constraints causing any loss of precision. The information of the representative programs is summarized in Table 4.12.

As Table 4.12 shows, both the equality version and enhanced equality version give many equality constraints which helps to reduce the problem space indicated by the number of analysis variables (in the last column of the table). The experimental data in Table 4.13 report the execution time of the three versions of the constraint programs and calculate the performance improvement using unification. The last line reports the overall improvement on time performance, in which the length of each benchmark is proportionated.

Name	**LOC**	**Constraint**	**Equality**	**Equality$^{\mathbf{EN}}$**	**Variable**
fibonacci	15	41	19	15	28
isPrime	18	39	22	18	28
lcm	23	51	17	16	34
ext_gcd	22	48	16	15	32
nwtIter	14	30	12	9	20
wlfIter	20	56	24	21	38
sum	16	33	13	11	22
log	71	145	62	50	98
calculator	258	604	268	237	418

Table 4.12: Benchmarks: Representative Programs.

Name	**TS**	**TE**	**TE′**	$\triangle\mathbf{T_1}$	$\triangle\mathbf{T_2}$
fibonacci	0.24	0.11	0.12	53%	52%
isPrime	0.24	0.13	0.15	48%	39%
lcm	0.25	0.20	0.21	21%	18%
ext_gcd	0.23	0.16	0.16	32%	32%
nwtIter	0.16	0.07	0.09	56%	43%
wlfIter	0.38	0.20	0.21	48%	46%
sum	0.17	0.13	0.13	25%	25%
log	0.70	0.48	0.53	31%	25%
calculator	6.04	4.30	4.45	29%	26%
Improvement on Average				32%	29%

where:
$\triangle T_1 = (TS - TE)/TS$

$\triangle T_2 = (TS - TE')/TS$

Table 4.13: Time Performance of the Inclusion Constraint Solver.

For each benchmark, the columns TS, TE and TE' give the time of performing the analysis on set-inclusion version, equality version, and enhanced equality version, respectively. Using unification results in a significant reduction in execution time - on average 32% ($\triangle T_1$) for equality version and 29% ($\triangle T_2$) for enhance equality version. The enhanced equality version is a little slower than equality version. In order to measure the level of precision, Table 4.14 reports the size of the solutions of the three versions where each pair (x, ℓ) is counted as one. As the table shows, the enhanced version is as precise as the pure inclusion one.

For the program fibonacci and isPrime, using unification in the equality version results in many extra false positives while as it seems quite fine with the rest

Name	**SS = SE′**	**SE**	$\triangle\mathbf{S_1}$
fibonacci	207	346	67%
isPrime	153	234	53%
lcm	427	435	1.9%
ext_gcd	396	404	2.0%
nwtIter	134	150	12%
wlfIter	396	444	12%
sum	127	133	4.7%
log	1460	1579	8.1%
calculator	17532	17986	2.6%
Loss in Prec. on Average			8.3%

where:
$\triangle S_1 = (SE - SS)/SS$

Table 4.14: Precision of the Inclusion Constraint Solver using unification (evaluated by the size of the solutions).

of benchmarks. The overall loss in precision is 8.3% and seems OK. However, the size of solutions itself may be not enough to judge the effect of extra false positives although it is a direct indication of the precision of an analysis. For some analyses, e.g. reaching definitions analysis and pointer analysis, the effect of reduced precision of these analyses may affect their client analysis. Therefore, the precision of client analyses may also need to be considered. How to choose a proper metric to measure the precision of an analysis is out of the range of this dissertation. For further information, see [Hin01] (in which several metrics of pointer analysis are described and their strengths and weaknesses are discussed.)

In fact, the enhanced version achieves a quite good performance. The only weakness is that one has to manually tune the constraints and for large programs this approach is not practical. However from the enhanced version, we demonstrate that many equivalent classes of analysis variables do exist in the analysis and can be used to speed up the calculation significantly.

Table 4.15 compares the results of our solver and the Succinct Solver in terms of time performance. The column TA reports the performance of the analysis implemented in ALFP logic. We observe that our solver is considerably faster - on average 81% faster for set-inclusion version, 87% faster for equality version, and 86% faster for enhanced equality version[1]. This may be explained by the fact that our solver employs much simpler data structure than the Succinct Solver and thus has lower space usage.

The number of Table 4.15 calculates the percentage of the improvement of our

[1] Note that the solution of set-inclusion version or enhanced equality version is as precise as that computed by the Succinct Solver.

Name	**TA**	$\triangle \mathbf{T_3}$	$\triangle \mathbf{T_4}$	$\triangle \mathbf{T_5}$
fibonacci	1.47	84%	92%	92%
isPrime	1.20	80%	90%	88%
lcm	3.80	93%	95%	95%
ext_gcd	2.54	91%	94%	94%
nwtIter	1.09	85%	93%	91%
wlfIter	1.87	80%	89%	89%
sum	1.15	85%	89%	89%
log	7.12	90%	93%	93%
calculator	25.36	76%	83%	82%
Improvement on Average		81%	87%	86%

where:

$\triangle T_3 = (TA - TS)/TA$

$\triangle T_4 = (TA - TE)/TA$

$\triangle T_5 = (TA - TE')/TA$

Table 4.15: Inclusion Constraint Solver v.s. the Succinct Solver: TS, TE and TE' represent the time performance of set-inclusion version, equality version, and enhanced equality version respectively.

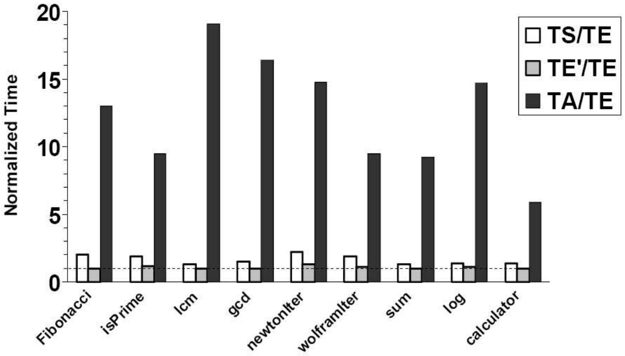


Figure 4.8: Performance comparison of individual benchmarks, where the performance of set-inclusion version TS, enhanced equality version TE', and the Succinct Solver TA is normalized against the equality version TE.

solver compared to the Succinct Solver. To make it clear how fast our solver using unification is, Figure 4.8 visualizes the comparison of the performance of the three versions between our solver and the Succinct Solver by normalizing the performance of the set-inclusion version, the enhanced equality version, and the ALFP logic version against that of the equality version. Our solver is clearly faster than the Succinct Solver, being 6.2 times faster for set-inclusion version, 8.9 times faster for enhanced equality version, and 9.3 times faster for equality version. If only our solver is concerned, the enhanced equality version achieves a good performance, which is only 1.1 times slower than that of the equality version. Both of the equality version and the enhanced equality version are around 1.5 times faster than the set-inclusion version.

As demonstrated by the double-layer semantics in Chapter 3, using unification would reduce the memory consumption of our solver. But this reduction is not quite observable for the representative programs of interest considering the size of the constraints generated for these programs is relatively small. When scalable programs are considered in the next sub-section, an apparent decrease in memory consumption shall be observed as expected.

4.3.3 Benchmarks: Scalable Programs

The representative programs allow us to measure the effect of using equality constraints on time performance and precision. These programs, however, cannot easily be extended to any size desired. In order to evaluate the scalability of the solvers we designed eight series of scalable programs with the desired size potential: each series of scalable programs consists of many individual programs and some part of them could be enlarged as many times as desired. Especially with well-designed scalable programs we are able to measure asymptotic complexity of the constraint solving for each series of benchmarks, and further analyze the impact of using unification on complexity. Finally, scalable programs allow us to measure the differences of the memory consumption between set-inclusion version and equality version, and between our constraint solver and the Succinct Solver. Upon precision, because of the simple structure of the scalable programs, one can simply use the heuristics presented in Subsection 4.3.2 to point out any (potential) loss in precision for the equality version and therefore the level of precision can be well-controlled. We, in this subsection, focus on analyzing the effect of using unification on performance.

Two families of scalable programs are selected for detailed presentation as in Table 4.16. Appendix B specifies a complete version of the eight series of scalable programs. For two series of scalable programs, the first number of the subscript denotes the nesting depth of loops or conditions, and the second yields the number of all assignments (at the deepest level). According to the analysis specification, the number constraints generated for $\mathrm{Wh}_{(1,n)}$ and $\mathrm{If}_{(n,1)}$ are both of $O(n)$. Given the method used for constructing the graph in the algorithm, both of the clusters constructed for the constraints have $O(n)$ labels.

As explained in Section 4.1, adopting unification will simplify the labeled cluster and further reduce the number of iterations. We show that the results of this simplification differ for the two examples: for the first one, it decreases the number of labels by a constant factor; in contrast, for the second there is only a

```
$\mathbf{Wh_{(1,n)}}$ :    while $x_0 < 2$ do ($x_1 := x_2$;
                                    ⋮
                                    $x_{n-1} := x_n$;
                                    $x_n := 1$)

$\mathbf{If_{(n,1)}}$ :    if $x_1 < 0$ then skip
               else  ⋮
                      if $x_n < 0$ then skip
                        else $x_0 := 1$
```

Table 4.16: Scalable Programs: Wh$(1, n)$ and If$(n, 1)$

constant number of labels left. This is because the constraints generated for n assignments of $\text{Wh}_{(1,n)}$ remain $O(n)$ by the rule [ass]. But applying unification to $\text{If}_{(n,1)}$ means we only keep set-inclusion in the constraints for one assignment(s) and thus the resulting graph has only a constant number of labels. The experimental results of the family $\text{Wh}_{(1,n)}$ are presented in Figure 4.9 and 4.10.

The first diagram shows that the execution time is improved 25% by using unification and the computation using set-inclusion is at least 70 times and sometimes even 200 times faster than the Succinct Solver. Compared to the Succinct Solver we postulate this is because of the two reasons: (1) a simpler data structure adopted by our solver that is more efficient to operate, and (2) a lower level of memory consumption of our solver (as discussed later). Notice that both of the solvers suffer a sharp performance-decline for large values of n: $n \geq 750$ in the case of the Succinct Solver, and $n \geq 9000$ and $n \geq 11000$ in the case of our solver. As a result, the use of unification enables our solver to scale to larger programs. For our solver especially, the computation time is so small when n is less than 250 that the initialization time becomes a major constant factor impacting the asymptotic complexity. To get the asymptotic growth rate of the solvers, we therefore select the data before performance deterioration happens and after the constant factor is no longer dominating. By a least square fit technique on the model $t = c_1 \cdot m^c + c_0$, we estimate that the time complexity of the Succinct Solver, and our solver without and with unification are $O(n^{2.21}), O(n^{2.02})$ and $O(n^{2.01})$ respectively. As expected, unification only helps to reduce the exponent value a bit.

The second diagram shows that unification saves up to 40% space compared to

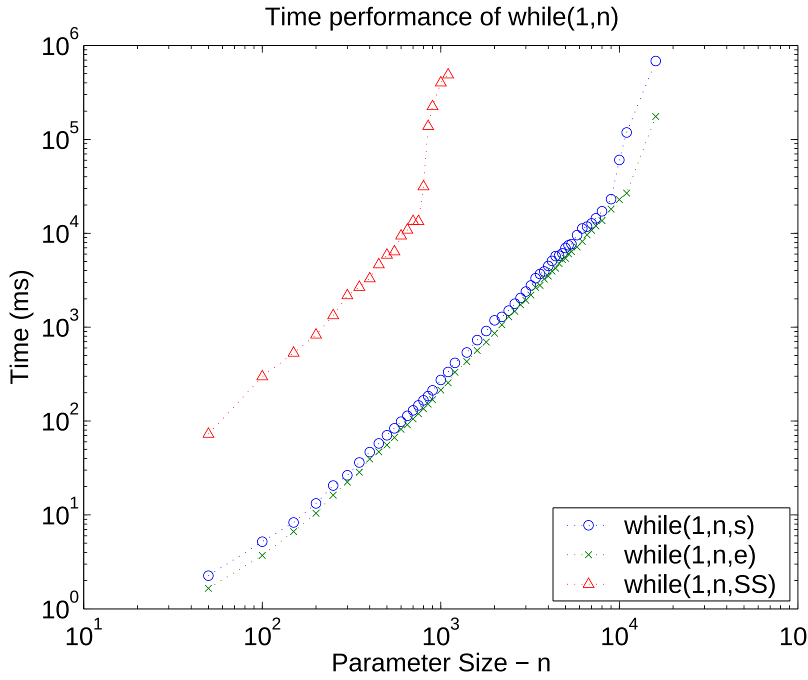


Figure 4.9: Time Performance of $\mathrm{Wh}_{(1,n)}$.

set-inclusion. Within the scalability of the Succinct Solver, we observe that our solver consumes much less memory than the Succinct Solver. Relating to the time performance presented in Figure 4.9, we postulate that the level of memory consumption of the two solvers is a key factor affecting their time performance when the program size is quite large: large memory consumption requires much extra effort in memory management and thus slow down the speed of a solver.

For the program family $\mathrm{If}_{(1,n)}$, a significant improvement is observed and its experimental results are presented as Fig. 4.11 and 4.12. This time our solver remains 30 times faster than the Succinct Solver and consumes much less memory when using unification. Again, we consider this is achieved by the high-efficient data structure and low memory consumption. Since no performance-deterioration is observed, the estimated complexities are printed out directly. As Figure 4.11 shows, unification results in almost linear time complexity while set-inclusion takes more than quadratic time and the Succinct Solver takes time $O(n^{1.3})$. The use of unification speeds up the constraint solving significantly, i.e. 28 times faster at least.

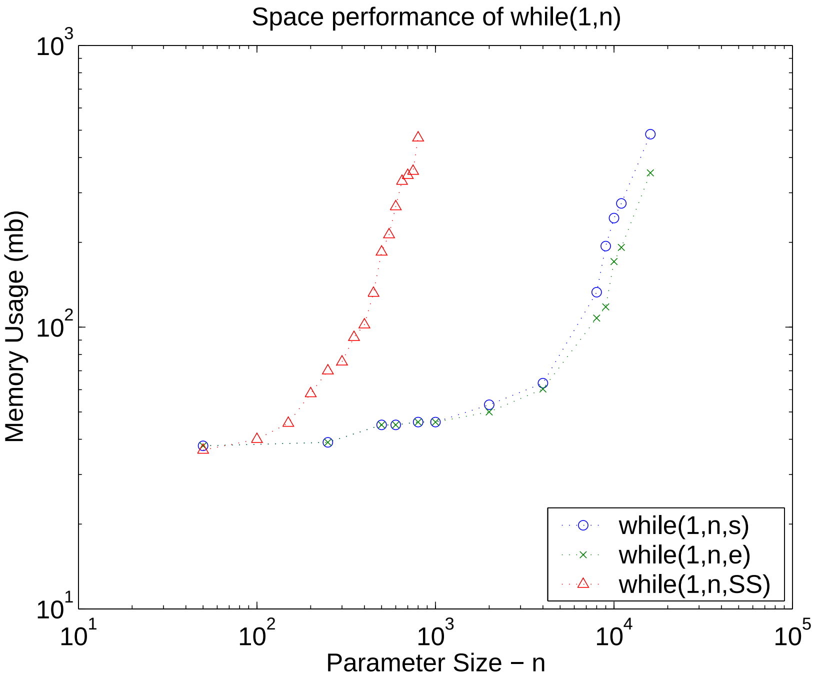


Figure 4.10: Memory Consumption of $\mathrm{Wh}_{(1,n)}$.

For space consumption we have a very similar result as presented in Figure 4.12. Relating the two figures, we can see a clear relation between time performance and space consumption: the less space consumed, the faster the constraint solving is.

Other series of scalable programs have been tested and in general the results are similar to either $\mathrm{Wh}_{1,n}$ or $\mathrm{If}_{n,1}$. We conclude that the performance improvement is proportional to the percentage of equality constraints in a constraint program.

4.4 Concluding Remarks

We have presented a worklist algorithm for the inclusion constraint solving. The termination and the correctness of the algorithm are proved. The complexity of the algorithm is $O(n^3)$ in a worst case study. With the use of unification, the asymptotic complexity could be reduced to almost linearity.

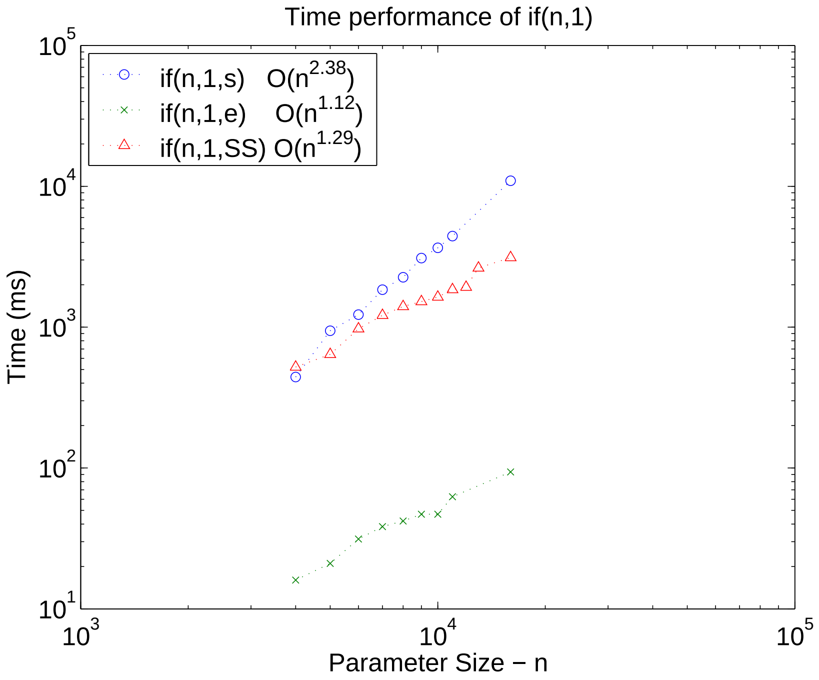


Figure 4.11: Time Performance of $\text{If}_{(n,1)}$.

To demonstrate the use of the framework and unification, two groups of benchmarks are designed: representative programs and scalable programs. A state-of-the-art solver, the Succinct Solver, is selected to make a comparison study. The experimental results show that our constraint solver is a large constant factor faster than the Succinct Solver for all the benchmarks. For some scalable programs, using unification may lower the asymptotic complexity even down to almost linear time. At the same time, unification helps to reduce the memory consumption which, in return, saves effort in memory-related operations and thus speeds up the constraint solving.

Unification need not give rise to imprecision: the enhanced equality version of the analysis for the representative programs shows the existence of many equivalent analysis variables in the reaching definitions analysis. Our experimental results demonstrate that these equivalences can be taken advantage of by our unification technique to improve the solver performance considerably.

When tuning a constraint program, analysis designers also need a good control on the performance and precision tradeoff. Our heuristic study on the example

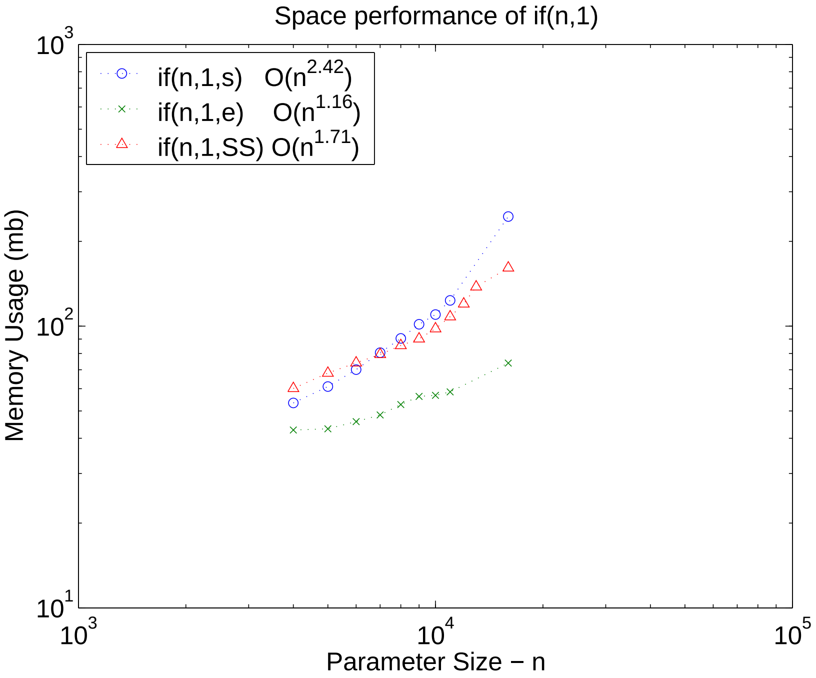


Figure 4.12: Memory Consumption of $\mathrm{If}_{(n,1)}$.

imperative language shows that a careful study on the conditions where imprecision may or may not occur pays off in gaining the expected level of precision.

CHAPTER 5

Extended Inclusion Constraint Language for Pointer Analysis

In this chapter, we extend our inclusion constraint language in order to implement pointer analysis. The analysis is especially interesting to us because analyzing very large programs written in languages with pointers, such as C or Java, requires the information of pointer behavior. When a pointer dereference is encountered, an analysis without good pointer information is either quite imprecise or unsound. Thus pointer analysis is a prerequisite of many other program analyses. Another motivation of studying pointer analysis is that with all kinds of unification techniques the previous research on pointer analysis demonstrates many interesting tradeoffs between the efficiency of the analysis and the precision of the computed solution [Ste96, Das00, SH97b, RC00, PKH04, HL07a, HL07b]. However, designing a pointer analysis, which is both scalable and precise, remains a challenge to researchers.

In the following sections, we first give an introduction of pointer analysis to motivate the development of this chapter. Then we specify the syntax and semantics of the extended language. The theories presented before are adjusted in order to rebuild the theoretical foundation. Next the algorithm of the constraint solver is redefined and its properties of interest are analyzed. Finally we present the implementation of the C pointer analysis, i.e. how we model the pointer behavior of the C programming language, e.g. aggregations, multiple dereferences, functional pointers, etc. In the next chapter, we further describe the technique of how to detect equivalent analysis variables and improve the efficiency of the analysis with the help of the extended language constructs. Part of the work was previously presented in [ZAN08].

5.1 Introduction to Pointer Analysis

A pointer analysis approximates that for each variable of pointer type what storage locations it can point to. The analysis is a prerequisite for a variety of analyses relating to compiler optimization, e.g. reaching definitions analysis, live variable analysis, constant propogation, etc. It is also a key technique for error detection, e.g., NULL pointer dereferencing, and program understanding. Unfortunately, exact pointer analysis is undecidable [Cha03, Ram94] and even flow-insensitive pointer analysis is NP-hard [LR91, Hor97]. Therefore there has to be some trade-off between precision and performance for pointer analyses.

Several dimensions could affect the performance and precision trade-offs of interprocedural pointer analysis[1]. The way that a pointer analysis deals with each of these dimensions is then used to classify the analysis. Two major dimensions are

Flow-sensitivity: If an analysis uses control-flow information of a procedure during the analysis, the analysis is flow-sensitive, otherwise it is flow-insensitive. A flow-sensitive analysis computes a solution for each program point, whereas a flow-insensitive analysis computes one solution for either the whole program, such as [And94, Ste96, ZRL96, HL07a], or for each function or method, such as [BCCH94, HBCC99, LH99]. Therefore a flow-sensitive analysis is more precise but much more expensive to compute than a flow-insensitive one. Despite a great deal of work on flow-sensitive analyses [CBC93, HBCC99, TGL06], scalability remains a challenge to them.

Context-sensitivity: If an analysis shows respect to the semantics of function calls, i.e. the calling context is considered when analyzing a function, the analysis is context-sensitive. In contrast, a context-insensitive analysis discards the calling context of a function. A context-sensitive analysis is equivalent to fully inline each function before performing the analysis, which turns to be exponential in program size. Thus, this approach is impractical for large programs

[1] Since arguments of functions are often pointer type in the C programming language, only interprocedural pointer analysis is considered in this dissertation.

Program Variable	**Analysis Variable**		
	Field-insensitive	**Field-based**	**Field-sensitive**
$aggr1.f$	$aggr1$	f	$aggr1\_f$
$aggr2.f$	$aggr2$		$aggr2\_f$
$aggr1.f$	aggr1	f	$aggr1\_f$
$aggr1.g$		g	$aggr1\_g$

Table 5.1: Aggregate Modeling.

[EGH94, NKmWH04, WL04, WL95, ZC04]. A context-insensitive analysis generalizes a function's calling context into one and thereby is faster but more imprecise than a context-sensitive analysis.

There are also several minor dimensions, including

Aggregate modeling: The manner in which aggregates (arrays and structs) are modeled. There are three approaches: *field-insensitive* in which the whole array or struct is modeled as one memory block; *field-based* in which one analysis variable models all instances of a field of an aggregate; *field-sensitive* in which a unique analysis variable models each field of an aggregate. Table 5.1 clarifies the differences between these approaches.

As the table illustrates, the field-insensitive and field-based approaches are imprecise in different ways. The first combines the solution of different fields in an aggregation and thereby sacrifices some precision. The second combines all the solutions for each instance of a given field into one and thus loses some precision as well. Field-sensitive analysis is more precise but also more difficult to implement. And for arrays its precision is compromised because of pointer operations conducted on arrays. Note that field-based analysis is unsound for C programs. Consider the below example.

Example 5.1 For the program code

```
typedef struct { int f; } aggr;
aggr h, *p;
int a = 0;
p = &h;
*p = &a;
```

The last assignment $*p = \&a$ assigns the address of a to field $aggr.f$, i.e. it is semantically equivalent to $(*p).f = \&a$ in C program according to the **ISO/ANSI** standard [ISO00]. But field-based analysis will report that f may not point to a. □

Heap modeling: The manner in which the heap is modeled. A straightforward approach models memory locations by *named objects*. The names can be either variable names or synthetic names introduced by an analysis to represent a set of memory locations, such as parts of the heap. A sophisticated approach performs shape analysis to get more precise analysis about heap behavior.

Whole program: Many pointer analyses require the whole program to be analyzed, while as some remain sound by analyzing only part of a program.

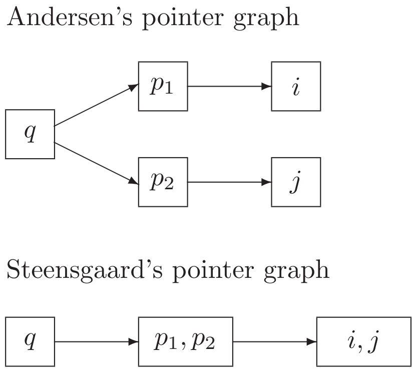
Because of the scalability problem of flow- and context-sensitivity, we consider, in this dissertation, flow- and context-insensitive analysis. Flow- and context-insensitive analyses are further classified into two major classes: *unification-based* [Ste96, ZRL96], which treat assignments as a bidirectional data-flow, and *inclusion-based* [And94, BCCH94, HBCC99, HL07a], which treat an assignment as unidirectional data-flow. Because of the bidirectional nature of the unification-based analysis, it gains high efficiency by sacrificing a lot of precision. On the other hand, inclusion-based analyses are much more precise but its worst-case complexity is $O(n^3)$ and has difficulty to scale to large programs. The following example is composed with the attempt to give an intuitive explanation to the two approaches. For further information about the two analysis, see [Ste96, And94].

Example 5.2 Consider the following program:

$$\begin{aligned} p_1 &= \&i; \\ p_2 &= \&j; \\ q &= \&p_1; \\ q &= \&p_2; \end{aligned}$$

The pointer graph for this program using the Andersen's analysis and the Steensgaard analysis are represented as Figure 5.1.

Compared to Andersen's graph, the points-to sets $p_1 \rightarrow j$ and $p_2 \rightarrow i$ are false positives in Steensgaard's graph. However, since Steensgaard's analysis unifies the two pairs of analysis variables, (p_1, p_2) and (i, j), its problem space is much smaller than that of Andersen's analysis, thereby is quite efficient. □


Andersen's pointer graph

Steensgaard's pointer graph

Figure 5.1: Andersen's vs Steensgaard's analysis

$$
\begin{array}{lll}
pre & ::= & c \subseteq \alpha \mid [\alpha_1, \cdots, \alpha_n] \\
\varphi & ::= & c \subseteq \alpha \mid \alpha \subseteq \beta \mid \alpha = \beta \mid \alpha \cap \beta \subseteq \gamma \mid \\
 & & \alpha \setminus c \subseteq \beta \mid \alpha \setminus (D) \subseteq \beta \mid \varphi_1 \wedge \varphi_2 \mid \\
 & & [\![\alpha]\!] \subseteq \beta \mid \alpha \subseteq [\![\beta]\!] \mid [\![\alpha]\!] = \beta \mid pre \Rightarrow \varphi \\
D & ::= & ? \mid ?, D \mid m \mid m, D
\end{array}
$$

Table 5.2: Syntax of the Extended Constraint Language

In the rest of this chapter, we shall extend our inclusion constraint language for implementing Andersen's analysis. We are especially interested in exploring equivalent analysis variables in order to reduce the problem space and thus improve the performance of Andersen's analysis.

5.2 Extended Inclusion Constraint

The syntax of the extended constraint language is specified in Table 5.2.

The universe considered is the same as before. A new syntax category *pre* is the antecedent (precondition) of a conditional constraint $pre \Rightarrow \varphi$ in which φ is a postfix, which must hold only when the antecedent *pre* holds. Two constructs are in the category: the first tests the membership of an analysis variable; the second checks if all of the analysis variables $\alpha_1, \cdots, \alpha_n$ are nonempty. Besides the normal equality constraint, we introduce a dynamic equality constraint

Rules for *pre*:			
	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} c \subseteq \alpha$	iff	$c \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$
	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} [\alpha_1, \cdots, \alpha_n]$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \neq \{\}$ for all $\alpha \in \{\alpha_1, \cdots, \alpha_n\}$
Rules for φ:			
1.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} c \subseteq \alpha$	iff	$c \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$
2.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \beta \subseteq \alpha$	iff	$\hat{\psi}_2(\hat{\psi}_1(\beta)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$
3.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \beta = \alpha$	iff	$\hat{\psi}_1(\beta) = \hat{\psi}_1(\alpha)$
4.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \cap \beta \subseteq \gamma$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \cap \hat{\psi}_2(\hat{\psi}_1(\beta)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\gamma))$
5.1	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \setminus c \subseteq \beta$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \setminus c \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
5.2	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \setminus (D) \subseteq \beta$	iff	$\hat{\psi}_2(\hat{\psi}_1(\alpha)) \setminus (D) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
6.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1 \wedge \varphi_2$	iff	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1$ and $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_2$
7.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \llbracket\alpha\rrbracket \subseteq \beta$	iff	$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\alpha)) : \hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
8.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \subseteq \llbracket\beta\rrbracket$	iff	$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\beta)) : \hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor))$
9.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \llbracket\alpha\rrbracket = \beta$	iff	$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\alpha)) : \hat{\psi}_1(\lfloor v \rfloor) = \hat{\psi}_1(\beta)$
10.	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} pre \Rightarrow \varphi$	iff	$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi$ whenever $\hat{\psi} \models_{\mathcal{T}} pre$

Table 5.3: Semantics of Extended Constraint Language Using Type Variables

$\llbracket\alpha\rrbracket = \beta$, which together with another two dynamic constraints $\llbracket\alpha\rrbracket \subseteq \beta$ and $\alpha \subseteq \llbracket\beta\rrbracket$, is used for formulating dynamic transitive closure problems, e.g., Andersen's analysis. These constraints are dynamic in the sense that they may yield new equality constraints or set-inclusion constraints during constraint solving. The conditional constraints are also dynamic since the equality or set-inclusion relation is enforced in the consequence of a conditional constraint only when its antecedent becomes true.

The semantics using the type variables in Table 3.3 is extended and presented in Table 5.3. We adopt the double-layer semantics here since it is the one implemented by the algorithm. It should also be straightforward to specify a standard version of the semantics. The two components $\hat{\psi}_1 \in \mathbf{Env_T}$ and $\hat{\psi}_2 \in \widehat{\mathbf{Env_{TB}}}$ are defined in the same way as before. An anonymous function $\lfloor \cdot \rfloor$ maps each tuple v to an analysis variable $\lfloor v \rfloor$. For instance, let $\lfloor v \rfloor = \kappa_v$ where κ is a reserved analysis variable name.

Program code	Meaning w.r.t. Pointer Analysis	Constraint
$x = \&y$	$loc(y) \in pts(x)$	$\{y\} \subseteq \kappa_x$
$x = y$	$pts(y) \subseteq pts(x)$	$\kappa_y \subseteq \kappa_x$
$x = *y$	$\forall p \in pts(y) : pts(p) \subseteq pts(x)$	$[\![\kappa_y]\!] \subseteq \kappa_x$
$*x = y$	$\forall p \in pts(x) : pts(y) \subseteq pts(p)$	$\kappa_y \subseteq [\![\kappa_x]\!]$

Table 5.4: Constraints for Andersen's Pointer Analysis.

The rule for constraint $[\![\alpha]\!] \subseteq \beta$ declares that for each tuple $v \in \hat{\psi}_2(\hat{\psi}_1(\alpha))$ a new subset inclusion is generated, i.e. $\hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$ in which $\lfloor v \rfloor$ maps the tuple v to its corresponding analysis variable. The rule for constraint $\alpha \subseteq [\![\beta]\!]$ is similar except that the tuples are from β this time. For rule 9, the constraint $[\![\alpha]\!] = \beta$, called dynamic equality constraint, yields new equivalences, i.e. $\lfloor v \rfloor = \beta$ for each tuple $v \in \hat{\psi}_2(\hat{\psi}_1(\alpha))$.

The following examples motivate the introduce of these new constraints.

Example 5.3 Andersen's pointer analysis [And94] is an analysis calculating dynamic transitive closure. A well-designed C-frontend can simplify the assignments of interest into four forms during a linear scan through the C source code [HT01] and four kinds of constraints are generated accordingly. Table 5.4 summarizes the four kinds of program code, the meaning of them, and the constraints used to model them respectively. Here $pts(p)$ represents the points-to set of p and $loc(p)$ represents the memory location named by p. □

Example 5.4 Given a program code

$$\begin{aligned} x &= *y; \\ *y &= x; \end{aligned}$$

we have the constraints

$$[\![\kappa_y]\!] \subseteq \kappa_x \wedge \kappa_x \subseteq [\![\kappa_y]\!]$$

The semantics of the constraint language then allows us to conclude that $[\![\kappa_y]\!] = \kappa_x$, which declares that any analysis variable described by κ_y is equivalent to κ_x. □

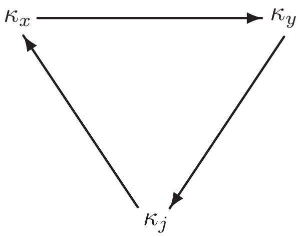


Figure 5.2: Cycles formed by the constraints of Example 5.5. The arrowed line → represents set-inclusion $\subseteq$.

Example 5.5 Given a program code

$$\begin{aligned} x &= \&i; \\ z &= \&j; \\ y &= x; \\ *z &= y; \\ x &= *z; \end{aligned}$$

we have the constraints

$$\{i\} \subseteq \kappa_x \wedge \{j\} \subseteq \kappa_z \wedge \kappa_x \subseteq \kappa_y \wedge \kappa_y \subseteq [\![\kappa_z]\!] \wedge [\![\kappa_z]\!] \subseteq \kappa_x$$

Intuitively, for the least solution of interest we have that $\hat{\psi}_2(\hat{\psi}_1(\kappa_x)) = \hat{\psi}_2(\hat{\psi}_1(\kappa_y)) = \hat{\psi}_2(\hat{\psi}_1(\kappa_j)) = \{i\}$ and $\hat{\psi}_2(\hat{\psi}_1(\kappa_z)) = \{j\}$. If we instantiate $[\![\kappa_z]\!]$ to be κ_j, we can see a loop formed by inclusion relation as visualized in the graph of Figure 5.2. Notice that $\kappa_x = \kappa_y$ only when the data of γ is non-empty. □

Example 5.6 Given a program code

$$\begin{aligned} x &= \&p; \\ y &= \&i; \\ y &= *x; \\ *z &= y; \\ m &= *z; \\ *x &= m; \end{aligned}$$

we have the constraints

$$\{p\} \subseteq \kappa_x \wedge \{i\} \subseteq \kappa_y \wedge [\![\kappa_x]\!] \subseteq \kappa_y \wedge \kappa_y \subseteq [\![\kappa_z]\!] \wedge [\![\kappa_z]\!] \subseteq \kappa_m \wedge \kappa_m \subseteq [\![\kappa_x]\!]$$

$$\begin{tikzcd}
\llbracket \kappa_x \rrbracket \arrow[r] & \kappa_y \arrow[d] \\
\kappa_m \arrow[u] & \llbracket \kappa_z \rrbracket \arrow[l]
\end{tikzcd}$$

Figure 5.3: Cycles formed by the constraints of Example 5.6. The arrowed line $\rightarrow$ represents set-inclusion $\subseteq$.

This time a least solution gives that $\hat{\psi}_2(\hat{\psi}_1(\kappa_x)) = \{p\}$, $\hat{\psi}_2(\hat{\psi}_1(\kappa_y)) = \{i\}$ and $\hat{\psi}_2(\hat{\psi}_1(\kappa_z)) = \hat{\psi}_2(\hat{\psi}_1(\kappa_m)) = \hat{\psi}_2(\hat{\psi}_1(\kappa_p)) = \{\}$. Intuitively, a circle is formed as shown in Figure 5.3 by the constraints $\llbracket \kappa_x \rrbracket \subseteq \kappa_y, \kappa_y \subseteq \llbracket \kappa_z \rrbracket, \llbracket \kappa_z \rrbracket \subseteq \kappa_m$, and $\kappa_m \subseteq \llbracket \kappa_x \rrbracket$. However, the circle is not completed by $\llbracket \kappa_x \rrbracket$ and $\llbracket \kappa_z \rrbracket$ but rather by any analysis variables described by $\llbracket \alpha \rrbracket$ and $\llbracket \gamma \rrbracket$. Thus the analysis variables κ_y and κ_m are not equivalent because the points-to set of κ_z is empty.

However, the loop still provides some insight about where equivalences may happen. For the above example program, for instance, we may use a conditional constraint

$$[\kappa_x, \kappa_z] \Rightarrow (\kappa_y = \llbracket \kappa_x \rrbracket \wedge \kappa_y = \llbracket \kappa_z \rrbracket \wedge \kappa_y = \kappa_m)$$

i.e. whenever neither of α and β are empty, (dynamic) equivalences are proposed by the consequent. □

As demonstrated by the above examples, from the inclusion constraints, new equivalences may be discovered and modeled by our (dynamic) equality constraints. In the next chapter, we shall present technique to explore how to identify them automatically.

5.3 Theoretical Properties of the Language

In this section, we show some properties discussed in Chapter 3 is well maintained by the extended constraint language. The definitions of the relation $\preceq$ and $\equiv$ remain the same as before. We first prove Proposition 3.12 for the extended language which state that for a constraint program φ if an estimate is acceptable then so are all its equivalences.

Proposition 5.7 *If* $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$*, then* $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \varphi$*.*

Proof. The proof is an induction on φ. For cases proved in Proposition 3.12, the argument remains the same. We continue with the new extended constructs.

Case $[\![\alpha]\!] \subseteq \beta$**.** Assume that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} [\![\alpha]\!] \subseteq \beta \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$$

From rule 7 in Table 5.3 we have

$$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\alpha)) : \hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$$

By Lemma 3.11 we have

$$\forall \alpha \in \mathbf{AVar} : \hat{\psi}_2(\hat{\psi}_1(\alpha)) = \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$$

and thus

$$\forall v \in \hat{\psi}'_2(\hat{\psi}'_1(\alpha)) : \hat{\psi}'_2(\hat{\psi}'_1(\lfloor v \rfloor)) \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\beta))$$

by the transitivity of inclusion relation. From rule 7 in Table 5.3 again we conclude that $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} [\![\alpha]\!] \subseteq \beta$.

Case $\alpha \subseteq [\![\beta]\!]$**.** Assume that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \subseteq [\![\beta]\!] \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$$

From rule 8 in Table 5.3 we have

$$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\beta)) : \hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor))$$

By Lemma 3.11 we have

$$\forall \alpha \in \mathbf{AVar} : \hat{\psi}_2(\hat{\psi}_1(\alpha)) = \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$$

and thus

$$\forall v \in \hat{\psi}'_2(\hat{\psi}'_1(\beta)) : \hat{\psi}'_2(\hat{\psi}'_1(\alpha)) \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\lfloor v \rfloor))$$

by the transitivity of inclusion relation. Finally by rule 8 in Table 5.3 we conclude that $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \alpha \subseteq [\![\beta]\!]$.

Case $[\![\alpha]\!] = \beta$**.** Assume that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} [\![\alpha]\!] = \beta \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$$

From rule 9 in Table 5.3 we have

$$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\alpha)) : \hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor)) = \hat{\psi}_2(\hat{\psi}_1(\beta))$$

By Lemma 3.11 we have

$$\forall \alpha \in \mathbf{AVar} : \hat{\psi}_1(\alpha) = \hat{\psi}'_1(\alpha)$$

and thus

$$\forall v \in \hat{\psi}'_2(\hat{\psi}'_1(\alpha)) : \hat{\psi}'_2(\hat{\psi}'_1(\lfloor v \rfloor)) = \hat{\psi}'_2(\hat{\psi}'_1(\beta))$$

by the transitivity of inclusion relation. From rule 9 in Table 5.3 again we conclude that $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} [\![\alpha]\!] = \beta$.

Case $pre \Rightarrow \varphi$**.** There are two sub-cases for the antecedent *pre*.

Sub-case 1: *pre* has the form $c \subseteq \alpha$. Assume that

$$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} c \subseteq \alpha \Rightarrow \varphi \wedge (\hat{\psi}_1, \hat{\psi}_2) \equiv (\hat{\psi}'_1, \hat{\psi}'_2)$$

By Lemma 3.11 we have

$$\forall \alpha \in \mathbf{AVar} : \hat{\psi}_2(\hat{\psi}_1(\alpha)) = \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$$

and thus if $c \nsubseteq \hat{\psi}_2(\hat{\psi}_1(\alpha))$, i.e. $c \nsubseteq \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$, $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} c \subseteq \alpha \Rightarrow \varphi$ holds trivially. Otherwise we have

$$c \subseteq \hat{\psi}_2(\hat{\psi}_1(\alpha)) \Rightarrow c \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\alpha))$$

From the induction hypothesis we then have that

$$c \subseteq \hat{\psi}'_2(\hat{\psi}'_1(\alpha)) \Rightarrow (\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} \varphi$$

holds. Finally by rule 10 in Table 5.3 we conclude that $(\hat{\psi}'_1, \hat{\psi}'_2) \models_{\mathcal{T}} c \subseteq \alpha \Rightarrow \varphi$.

Sub-case 2: *pre* has the form $[\alpha_1, \cdots, \alpha_n]$. Similarly. □

Next, we show that for the extended language the Moore family property for the complete prelattice $(\mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}, \preceq)$ is maintained for a set of solutions.

Theorem 5.8 *A set of solutions given by* $\{(\hat{\psi}_1, \hat{\psi}_2) | (\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi\}$ *is a Moore family for a complete prelattice.*

Proof. From Fact 3.21, the pre-ordered set $(\mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}, \preceq)$ is a complete prelattice. The proof is an induction on φ and the argument of the cases is the same as those in Theorem 3.24. We show the theorem remains valid for the new introduced constructs.

Case $[\![\alpha]\!] \subseteq \beta$**.** Assume

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} [\![\alpha]\!] \subseteq \beta$$

for some set I and let $(\hat{\psi}_1^{\sqcap}, \hat{\psi}_2^{\sqcap}) = \hat{\sqcap}_i(\hat{\psi}_1^i, \hat{\psi}_2^i)$. From rule 7 in Table 5.3 we have

$$\forall i \in I : \forall v \in \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) : \hat{\psi}_2^i(\hat{\psi}_1^i(\lfloor v \rfloor)) \subseteq \hat{\psi}_2^i(\hat{\psi}_1^i(\beta))$$

We then have that

$$\forall v \in \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) : \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\lfloor v \rfloor)) \subseteq \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\beta))$$

By Lemma 3.15 we get $\hat{\psi}_2^{\sqcap}(\hat{\psi}_1^{\sqcap}(\gamma)) = \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\gamma))$ for any analysis variable γ. This allows us to conclude that $(\hat{\psi}_1^{\sqcap}, \hat{\psi}_2^{\sqcap}) \models_{\mathcal{T}} [\![\alpha]\!] \subseteq \beta$.

Case $\alpha \subseteq [\![\beta]\!]$**.** Assume

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} \alpha \subseteq [\![\beta]\!]$$

for some set I and let $(\hat{\psi}_1^{\sqcap}, \hat{\psi}_2^{\sqcap}) = \hat{\sqcap}_i(\hat{\psi}_1^i, \hat{\psi}_2^i)$. From rule 8 in Table 5.3 we have

$$\forall i \in I : \forall v \in \hat{\psi}_2^i(\hat{\psi}_1^i(\beta)) : \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) \subseteq \hat{\psi}_2^i(\hat{\psi}_1^i(\lfloor v \rfloor))$$

We then have that

$$\forall v \in \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\beta)) : \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) \subseteq \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\lfloor v \rfloor))$$

By Lemma 3.15 we get $\hat{\psi}_2^{\sqcap}(\hat{\psi}_1^{\sqcap}(\gamma)) = \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\gamma))$ for any analysis variable γ. This allows us to conclude that $(\hat{\psi}_1^{\sqcap}, \hat{\psi}_2^{\sqcap}) \models_{\mathcal{T}} \alpha \subseteq [\![\beta]\!]$.

Case $[\![\alpha]\!] = \beta$**.** Assume

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} [\![\alpha]\!] = \beta$$

for some set I and let $(\hat{\psi}_1^{\sqcap}, \hat{\psi}_2^{\sqcap}) = \hat{\sqcap}_i(\hat{\psi}_1^i, \hat{\psi}_2^i)$. From rule 9 in Table 5.3 we have

$$\forall i \in I : \forall v \in \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha)) : \hat{\psi}_1^i(\lfloor v \rfloor) = \hat{\psi}_1^i(\beta)$$

By Lemma 3.15 we get

$$\forall v \in \hat{\psi}_2^{\sqcap}(\hat{\psi}_1^{\sqcap}(\alpha)) : \hat{\psi}_1^{\sqcap}(\lfloor v \rfloor) = \hat{\psi}_1^{\sqcap}(\beta)$$

This allows us to conclude that $(\hat{\psi}_1^{\sqcap}, \hat{\psi}_2^{\sqcap}) \models_{\mathcal{T}} [\![\alpha]\!] = \beta$.

Case $\mathit{pre} \Rightarrow \varphi$**.** Assume

$$\forall i \in I : (\hat{\psi}_1^i, \hat{\psi}_2^i) \models_{\mathcal{T}} \mathit{pre} \Rightarrow \varphi$$

for some set I. We prove by the two cases of *pre*.

Sub-Case 1: *pre* has the form $c \subseteq \alpha$. If $\exists i \in I : c \nsubseteq \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha))$, then we know that $c \nsubseteq \hat{\psi}_2^{\Cap}(\hat{\psi}_1^{\Cap}(\alpha))$ since $\hat{\psi}_2^{\Cap}(\hat{\psi}_1^{\Cap}(\alpha)) = \cap_i \hat{\psi}_2^i(\hat{\psi}_1^i(\alpha))$ by Lemma 3.15. Thus the judgement $(\hat{\psi}_1^{\Cap}, \hat{\psi}_2^{\Cap}) \models_{\mathcal{T}} c \subseteq \alpha = \varphi$ is trivially true. Otherwise we have $c \subseteq \hat{\psi}_2^{\Cap}(\hat{\psi}_1^{\Cap}(\alpha))$ holds. Together with the induction hypothesis on the postfix φ, we have that

$$c \subseteq \hat{\psi}_2^{\Cap}(\hat{\psi}_1^{\Cap}(\alpha)) \Rightarrow (\hat{\psi}_1^{\Cap}, \hat{\psi}_2^{\Cap}) \models_{\mathcal{T}} \varphi$$

holds. From rule 10 in Table 5.3, we conclude that $(\hat{\psi}_1^{\Cap}, \hat{\psi}_2^{\Cap}) \models_{\mathcal{T}} c \subseteq \alpha \Rightarrow \varphi$.

Sub-Case 2: *pre* has the form $[\alpha_1, \cdots, \alpha_n]$. Similarly. □

Finally, Proposition 5.9 shows that the extended constraint language and semantics are compliant with the lifting strategy.

Proposition 5.9 *If* $\varphi_1 \leq \varphi_2$ *and* $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_2$ *then* $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \varphi_1$.

Proof. The proof is a straightforward induction on the clause φ_1. □

5.4 Constraint Solving

We continue using labeled cluster for translating the new introduced constraints. Different from the constraints presented before, however, the constraints $[\![\alpha]\!] \subseteq [\![\beta]\!]$, $\alpha \subseteq [\![\beta]\!]$, $[\![\alpha]\!] = \beta$, and *pre* $\Rightarrow \varphi$ are special in that labels corresponding to these constraints do not reflect the information transfer from one node to another directly but may cause the update of a labeled cluster by generating new labels. Thus, for the labels given by these four kinds of constraints we check whether the labeled cluster needs to be updated instead of propagating data changes.

For instance, the constraint $[\![\alpha]\!] \subseteq \beta$ yields a label associated with a node (type variable) corresponding to α; the constraint $\alpha \subseteq [\![\beta]\!]$ yields a label associated with a node corresponding to β; $[\![\alpha]\!] = \beta$ yields a label associated with node corresponding to α. For a conditional constraint, we identify all the nodes whose change may cause an update to a labeled cluster and attach a label decorated by the conditional constraint to each of these nodes.

Besides equality constraints, the dynamic equality can also be used to simplify a labeled cluster: The label for constraint $[\![\alpha]\!] \subseteq \beta$ or $\beta \subseteq [\![\alpha]\!]$ is dispensed with whenever $[\![\alpha]\!] = \beta$.

```
INPUT :    $(\mathsf{U}, \mathsf{N})$
OUTPUT :   $(\mathsf{D}_1, \mathsf{D}_2)$

Step 1 :   Initializing data structures
           $\mathsf{W} :=$ nil;
           for $\alpha_i$ in $\mathsf{AVar}_\star$ do
              $\mathsf{D}_1[\alpha_i] := i$;
              $\mathsf{D}_2[i] = \emptyset$;
              $\mathsf{E}[i] =$ nil;

Step 2 :   Implementing fast union/find data structure
           for $i$ in $\mathbf{TV}_\star$ do
              $\mathsf{A}[i] := i$;
              $\mathsf{H}[i] := 0$;
           for $\alpha \subseteq \beta$ in $\mathsf{U}$ do $\mathsf{unify}(\mathsf{find}(\alpha), \mathsf{find}(\beta))$;
           for $\alpha \in \mathsf{AVar}_\star$ do $\mathsf{find}(\alpha)$;

Step 3 :   Constructing the labeled cluster
           for $cc$ in $\mathsf{N}$ do $\mathsf{addEdge}(cc, cc)$

Step 4 :   Iteration
           While $\mathsf{W} \neq$ nil do
              $\gamma :=$ SELECT-FROM$(\mathsf{W})$;
              $t_e := \mathsf{E}[\mathsf{find}(\gamma)]$;
              for $cc$ in $t_e$ do $\mathsf{solve}(cc)$

Step 5 :   Recording the result
           for $\alpha_i$ in $\mathsf{AVar}_\star$ do $\mathsf{D}_1[\alpha_i] := \mathsf{find}(i)$;
```

Table 5.5: Worklist Algorithm (Modified)

5.4.1 Algorithm

Table 5.5 presents a modified worklist algorithm that calculates the least solution for a constraint program. Besides the functions defined in Table 4.2, three new auxiliary functions are specified in Table 5.6. The data structures $\mathsf{D}_1, \mathsf{D}_2, \mathsf{E}$ and W are defined in the exact same way as before. The algorithm takes as input a pair of constraint lists (U, N) and takes as output a solution $(\mathsf{D}_1, \mathsf{D}_2)$, where U contains all equality constraints and N contains all the rest.

The first two steps of the worklist algorithm are same as before and are pre-

```
procedure $\mathsf{addEdge}(cc', cc)$ is
    case $cc'$ of
        $c \subseteq \alpha$ :          if $cc' = cc$ then $\mathsf{add}(\alpha, c)$;
        $\alpha \subseteq \beta$ or $\alpha \setminus c \subseteq \beta$ or $\alpha \setminus D \subseteq \beta$ or $[\![\alpha]\!] \subseteq \beta$ or $\beta \subseteq [\![\alpha]\!]$ or $[\![\alpha]\!] = \beta$ :
                                        $\mathsf{E}[\mathsf{D}_1[\alpha]] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1[\alpha]]$;
        $\alpha \cap \beta \subseteq \gamma$ :    $\mathsf{E}[\mathsf{D}_1[\alpha]] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1[\alpha]]$;
                                        $\mathsf{E}[\mathsf{D}_1[\beta]] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1[\beta]]$;
        $c \subseteq \alpha \Rightarrow cc''$ :  $\mathsf{E}[\mathsf{D}_1[\alpha]] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1[\alpha]]$;
                                        $\mathsf{addEdge}(cc'', cc)$;
        $\neg[\ell] \Rightarrow cc''$ :    for $\alpha$ in $\ell$ do $\mathsf{E}[\mathsf{D}_1[\alpha]] := \{cc\} \cup \mathsf{E}[\mathsf{D}_1[\alpha]]$;

procedure $\mathsf{solve}(cc)$ is
    case $cc$ of
        $c \subseteq \alpha$ :             $\mathsf{add}(\alpha, c)$;
        $\alpha \subseteq \beta$ :             $\mathsf{add}(\beta, \mathsf{D}_2[\mathsf{find}(\alpha)])$;
        $\alpha = \beta$ :             $\mathsf{unify2}(\alpha, \beta)$;
        $\alpha \setminus c \subseteq \beta$ :       $\mathsf{add}(\beta, \mathsf{D}_2[\mathsf{find}(\alpha)] \setminus c)$;
        $\alpha \setminus D \subseteq \beta$ :       $\mathsf{add}(\beta, \mathsf{D}_2[\mathsf{find}(\alpha)] \setminus D)$;
        $\alpha \cap \beta \subseteq \gamma$ :       $\mathsf{add}(\gamma, \mathsf{D}_2[\mathsf{find}(\alpha)] \cap \mathsf{D}_2[\mathsf{find}(\beta)])$;
        $[\![\alpha]\!] \subseteq \beta$ :          for $\mu$ in $\mathsf{D}_2[\mathsf{D}_1[\alpha]]$ do
                                    $\mathsf{E}[\mathsf{D}_1[\lfloor\mu\rfloor]] := \{\lfloor\mu\rfloor \subseteq \beta\} \cup \mathsf{E}[\mathsf{D}_1[\lfloor\mu\rfloor]]$;
                                    $\mathsf{W} := \{\lfloor\mu\rfloor\} \cup \mathsf{W}$
        $\alpha \subseteq [\![\beta]\!]$ :          for $\mu$ in $\mathsf{D}_2[\mathsf{D}_1[\beta]]$ do
                                    $\mathsf{E}[\mathsf{D}_1[\alpha]] := \{\alpha \subseteq \lfloor\mu\rfloor\} \cup \mathsf{E}[\mathsf{D}_1[\alpha]]$;
                                    $\mathsf{W} := \{\alpha\} \cup \mathsf{W}$
        $[\![\alpha]\!] = \beta$ :          for $\mu$ in $\mathsf{D}_2[\mathsf{D}_1[\alpha]]$ do
                                    $\mathsf{unify2}(\lfloor\mu\rfloor, \beta)$
        $c \subseteq \alpha \Rightarrow cc'$ :   if $c \subseteq \mathsf{D}_2[\mathsf{find}(\alpha)]$ then $\mathsf{solve}(cc')$;
        $\neg[\ell] \Rightarrow cc'$ :      if IS-NON-EMPTY($\ell$) then $\mathsf{solve}(cc')$;

procedure $\mathsf{unify2}(\alpha, \beta)$ is
    $t_1 := \mathsf{find}(\alpha)$;
    $t_2 := \mathsf{find}(\beta)$;
    $r := \mathsf{unify}(t_1, t_2)$;
    $td := \mathsf{D}_2[t_1] \cup \mathsf{D}_2[t_2]$;
    $\mathsf{E}[r] := \mathsf{E}[t_1] \cup \mathsf{E}[t_2]$;
    if $\mathsf{D}_2[t_1] \neq \mathsf{D}_2[t_2]$ then $\mathsf{D}_2[r] := td$;
                                   $\mathsf{W} := \{\alpha\} \cup \mathsf{W}$;
```

Table 5.6: Worklist Algorithm (Modified): Auxiliary Functions

sented here for the reason of completeness. Step 3 builds up the labeled cluster and initializes the data of D_2 by invoking the recursive function addEdge. The recursion is needed for the conditional constraint which contains postfixes. For the same reason, the recursive function solve is invoked at Step 4 to propagate data changes or update the labeled cluster with respect to the labels attached to each analysis variable from the worklist W. Note that instead of the map D_1 the function find is used in the iteration to acquire the type variable for each analysis variable. This is because new equivalences may appear during the calculation by the use of dynamic equality constraints and thus the map from analysis variables to type variables is not static any more but may be dynamically updated. Finally, the last step records the final value of $\mathsf{D}_1(\alpha)$ for each analysis variable α.

To simplify the presentation, the pseudo-code ignores some obvious optimizations. For the same reason, we suppose all the conjunction operators in the consequent have been moved to the top level for the input (U, N), e.g. $c_0 \subseteq \alpha \Rightarrow (\varphi_1 \wedge \varphi_2)$ is transferred to $(c_0 \subseteq \alpha \Rightarrow \varphi_1) \wedge (c_0 \subseteq \alpha \Rightarrow \varphi_2)$, by the following fact.

Fact 5.10 *$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} pre \Rightarrow (\phi_1 \wedge \phi_2)$ if and only if $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} pre \Rightarrow \phi_1$ and $(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} pre \Rightarrow \phi_2$.*

5.4.2 Properties of the Algorithm

In order to show that the modified algorithm still guarantees termination property and computes a least solution for a given clause, we state that

Theorem 5.11 *Given a clause φ the algorithm of Table 5.5 terminates and the result $(\mathsf{D}_1, \mathsf{D}_2)$ produced by the algorithm satisfies*

$$(\mathsf{D}_1, \mathsf{D}_2) = \hat{\sqcap}\{(\psi_1', \psi_2') | (\psi_1', \psi_2') \models_{\mathcal{T}} \varphi\}$$

Proof. We first prove that the algorithm always terminates. It is immediate that the steps 1, 2, and 3 terminate considering that the sets $\mathbf{AVar}_\star, \mathbf{TV}_\star, \mathsf{U}$ and N are finite. For Step 4, observe that for each type variable i the data $\mathsf{D}_2[i]$ never decreases and it can increase a finite number of times at most. Observe that a node α may be added to the worklist although $\mathsf{D}_2[\mathsf{D}_1[\alpha]]$ may not have increased. However this only happens a finite number of times (specifically the number of nodes at most) because it may occur only when (1) two nodes need to unify into one or (2) the labeled cluster is updated by dynamic constraints. For each analysis variable placed on the worklist a finite amount of calculation needs to be executed in order to remove the node from the worklist and thus guarantees termination. This completes the first part of the proof.

To show the result calculated by the algorithm is a least solution, assume (ψ'_1, ψ'_2) be an estimate, such that $(\psi'_1, \psi'_2) \models_{\mathcal{T}} \varphi$. It is straightforward to verify that the following invariant

$$\begin{array}{rl} \forall \alpha, \beta \in \mathbf{AVar}: & \mathsf{D}_1[\alpha] = \mathsf{D}_1[\beta] \Rightarrow \psi'_1(\alpha) = \psi'_1(\beta) \land \\ \forall \gamma \in \mathbf{AVar}: & \mathsf{D}_2[\mathsf{D}_1[\gamma]] \subseteq \psi'_2(\psi'_1(\gamma)) \end{array}$$

is maintained everywhere in Step 4 and Step 5. It follows that $(\mathsf{D}_1, \mathsf{D}_2) \preceq (\psi'_1, \psi'_2)$ upon the completion of the algorithm by Lemma 3.10.

Next we show by contradiction that $(\mathsf{D}_1, \mathsf{D}_2)$ is indeed a solution for constraint program φ. Suppose there exists $cc \in \mathsf{U} \cup \mathsf{N}$ such that $(\psi_1, \psi_2) \models_{\mathcal{T}} cc$ does not hold.

If cc is the form $c \subseteq \alpha$ then the first case of the function $\mathsf{addEdge}$ invoked at Step 2 ensures that $c \subseteq \mathsf{D}_2[\mathsf{D}_1[\alpha]]$ and this is maintained throughout the algorithm; hence cc can not have this form. Note that $\mathsf{D}_1[\alpha]$ may have different value at Step 4 and Step 5. However, this does not change the fact that the constant c is inside the set $\mathsf{D}_2[\mathsf{D}_1[\alpha]]$.

If cc is the form $\alpha \subseteq \beta$, it must be the case that the final value of $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \neq \emptyset$ since otherwise $(\mathsf{D}_1, \mathsf{D}_2) \models_{\mathcal{T}} \alpha \subseteq \beta$ would hold. Now consider the last time $\mathsf{D}_2[\mathsf{find}(\alpha)]$ was modified and note that α was placed on the worklist at that time (by procedure add); since the final worklist is empty we must have considered the constraint $\alpha \subseteq \beta$ (which is in $\mathsf{E}[\mathsf{find}(\alpha)]$) and updated $\mathsf{D}_2[\mathsf{find}[\beta]]$ accordingly. By Step 5, we finally have that $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \subseteq \mathsf{D}_2[\mathsf{D}_1[\beta]]$; hence cc can not have this form either.

If cc is the form $\alpha = \beta$ then after the execution of Step 2, we can be sure that $\mathsf{find}[\alpha] = \mathsf{find}[\beta]$ and this is maintained throughout the algorithm. At the last step, we have that $\mathsf{D}_1[\alpha] = \mathsf{D}_1[\beta]$; hence cc can not have this form.

If cc is the form $\alpha \setminus c \subseteq \beta$ then similar to the case of $\alpha \subseteq \beta$, $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \neq \emptyset$ since otherwise $(\mathsf{D}_1, \mathsf{D}_2) \models_{\mathcal{T}} \alpha \setminus c \subseteq \beta$ would hold. Now consider the last time $\mathsf{D}_2[\mathsf{find}(\alpha)]$ was modified and note that α was placed on the worklist at that time (by procedure add); since the final worklist is empty we must have considered the constraint $\alpha \setminus c \subseteq \beta$ (which is in $E[\mathsf{find}(\alpha)]$) and updated $\mathsf{D}_2[\mathsf{find}[\beta]]$ eventually. This ensures that $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \setminus c \subseteq \mathsf{D}_2[\mathsf{D}_1[\beta]]$; hence cc can not have this form.

Similarly, we can show that cc can not have the form of $\alpha \setminus (D) \subseteq \beta$ or $\alpha \cap \beta \subseteq \gamma$.

If cc is the form $[\![\alpha]\!] \subseteq \beta$, then $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \neq \emptyset$ since otherwise $(\mathsf{D}_1, \mathsf{D}_2) \models_{\mathcal{T}} [\![\alpha]\!] \subseteq \beta$ would hold. Consider the last time $\mathsf{D}_2[\mathsf{find}(\alpha)]$ was modified and note that α was placed on the worklist at that time (by procedure add); since the final worklist is empty we must have considered the constraint $[\![\alpha]\!] \subseteq \beta$ (which is in $E[\mathsf{find}(\alpha)]$), i.e. for each $\mu \in \mathsf{D}_2[\mathsf{find}[\alpha]]$ we have $\mathsf{D}_2[\mathsf{find}[\lfloor\mu\rfloor]] \subseteq \mathsf{D}_2[\mathsf{find}[\beta]]$ eventually. This ensures that $\mathsf{D}_2[\mathsf{D}_1[\alpha]] \setminus c \subseteq \mathsf{D}_2[\mathsf{D}_1[\beta]]$ at Step 5 after we record the final result; hence cc can not have this form.

Similarly, we can show that cc can not have the form of $\alpha \subseteq [\![\beta]\!]$ or $[\![\alpha]\!] = \beta$.

If cc is the form $c \subseteq \alpha \Rightarrow cc'$ or $\neg[\ell] \Rightarrow cc'$, then all the antecedents (including those inside cc') hold otherwise the statement is true trivially. For the consequence, it must be one of the form which has been considered above. Consider the last time the conditional constraint is retrieved (because one of its associated analysis variable is on the worklist) and all the antecedents are true, then Step 4 ensures that the consequence hold as shown above. This completes the whole proof. □

Upon the complexity of the new algorithm, we have the following theorem.

Theorem 5.12 *The asymptotic complexity of the algorithm presented in Table 5.5 and 5.6 is*

$$O((n+m)\cdot n^3)$$

where n is the size of constraint programs and m is the number of equality constraints that can specify new equivalences between analysis variables.

Proof. Note that the number of analysis variables, type variables, and constraints are bound to $O(n)$.

The highest complexity is dominated by the fourth step. First note that there could be up to $O(n^2)$ labels which is given by a program of the size $O(n)$ because of the dynamic constraints. And each label (no matter exists originally or generated during computation) could be retrieved $O(n+m)$ times. The extra m times are because an analysis variable may be put on the list even when they are not enlarged by doing unification over equality constraints. The number $O(m)$ is bound to $O(n)$ because for $O(n)$ analysis variables at most $O(n)$ equality constraints are needed to specify any fresh equivalences. Considering each set-union operation takes $O(n)$ (as we explained in previous chapter), we get $(n+m)\cdot n^2 \cdot n$.

Next note there could be $O(n^2)$ equality constraints for $O(n)$ nodes whereas only $O(n)$ of them could specify new equivalences. Thus m is bound to $O(n)$.

The redundant equalities can be handled in time $O(n\alpha(n^2 - m, n)$.

Finally consider there could be some conditional constraints that have a length $O(n)$ and handling each of such constraints may take $O(n)$ set-union operations. But then the number of such constraints must be constant and each of them can be retrieved up to $O(n + m)$ times. □

5.4.3 Highlights of the Implementation

According to the above theorem, the complexity of the new algorithm is $O(n^4)$. However, in implementation many techniques could be introduced in order to improve the efficiency of the solver. Some of highlights of the implementation are summarized as below:

- The result of unification from (dynamic) equality constraints is taken advantage of to simplify a labeled cluster in constraint solving. For instance, the labels for constraints $\alpha \subseteq \beta, \alpha \setminus c \subseteq \beta$ and $\alpha \cap \gamma \subseteq \beta$ (this label is attached to both α and γ) are removed, if α and β are detected to be equivalent. Similarly the labels for constraints $[\![\alpha]\!] \subseteq \beta$ and $\beta \subseteq [\![\alpha]\!]$ are dispensed with whenever $[\![\alpha]\!] = \beta$.
- In the iteration, we prioritize the constraint solving of dynamic equality constraints in order to maximize the benefit acquired from the new found equivalences.
- For the constraints of the forms $[\![\alpha]\!] \subseteq \beta, \beta \subseteq [\![\alpha]\!]$, and $[\![\alpha]\!] = \beta$, we adopt a difference propagation technique to reduce repetitive operations: in each iteration, if α is enlarged with a set of tuples, say δ, we consider the elements of δ only instead of the whole data of α. This technique, which is considered as an instance of the general framework proposed by Fecht and Seidl [FS98b], was used by Pearce et al. [PKH04] for pointer analysis, and also by Nielson et al. [NSN02] for the Succinct Solver.
- As a worklist algorithm, our constraint solver provides support for two strategies to prioritize the worklist, last-in-first-out (LIFO) and Least Recent Fired (LRF). LIFO has been introduced before and LRF was initially proposed by Kanamori et al. [KW94], and applied by Pearce et al. to pointer analysis [PKH04]. Intuitively when there are a lot of cycles of data transfer among analysis variables, the stack strategy may reduce the times of iteration by attempting to reach fixed point within the elements of a cycle first before going on the calculation of any other elements. On the other hand, the LRF strategy is to favorite the analysis variable which has largest change by picking up the least recently fired one. This is based

on the fact that the several changes on data field are propagated in one set operation.

Another mechanism adopted for worklist management is that the worklist can be divided into two lists, called current and next, as described by Nielson et al. [NNH99]. We shall further discuss the effect of using these mechanisms on different analyses in our experimental study in next chapter.

5.5 C Pointer Analysis

As illustrated by Example 5.3, our aim is to yield four kinds of constraints in a linear scan of the C program. But the types of program code considered is rather restrictive in the example and the real C program has more features, which need to be carefully handled in order to maintain the correctness of the analysis. In this section, we describe how the features of a real C program are treated in our analysis.

Nested Pointer Dereference. Multiple pointer dereferences are a normal phenomena in C programming. We summarize them into four classes and discuss how to reduce the level of dereferences in assignments case by case. For the reason of simplicity, we assume that none of address-taking operations happens together with any dereferences, i.e. $**\&*a$ is represented as $**a$ directly.

(a) For the assignment of the form $*^{(n)}p = q$[2], we transform it into standard forms by introducing temporary variables. Specifically

$$\begin{array}{rcl} t_1 & = & *p \\ t_2 & = & *t_1 \\ & \vdots & \\ t_{n-1} & = & *t_{n-2} \\ *t_{n-1} & = & q \end{array}$$

For example, the assignment $***p = q$ is transformed into three assignments, $t_1 = *p$; $t_2 = *t_1$; $*t_2 = q$. One can verify the correctness of this transformation following the meaning of the program.

[2] We use $*^{(n)}$ to denote the n depth of dereferences where $n \geq 2$.

(b) For the assignment of the form $p = *^{(n)}q$, we can similarly transform it into

$$\begin{aligned} t_1 &= *q \\ t_2 &= *t_1 \\ &\vdots \\ t_{n-1} &= *t_{n-2} \\ p &= *t_{n-1} \end{aligned}$$

Here the idea is still to reduce the depth of nested dereferences by introducing extra program variables and assignments.

(c) For the assignment of the form $*^{(n)}p = *^{(m)}q$, we simply use the technique described as the cases (a) and (b) to reduce the level of nested dereferences on the both sides of the assignment, specifically,

$$\begin{aligned} l_1 &= *p \\ l_2 &= *l_1 \\ &\vdots \\ l_{n-1} &= *l_{n-2} \\ r_1 &= *q \\ r_2 &= *r_1 \\ &\vdots \\ r_n &= *r_{n-1} \\ *l_{n-1} &= r_n \end{aligned}$$

(d) For the assignment of the form $*^{(n)}p = \&q$, we need to be a little more careful. A straightforward transformation could be

$$\begin{aligned} t_1 &= *p \\ t_2 &= *t_1 \\ &\vdots \\ t_n &= *t_{n-1} \\ *t_n &= \&q \end{aligned}$$

But this does not show respect to the original assignment. To be illustrative, consider the assignment, $**p = \&q$, for example. Following the above transformation, we have that $t_1 = *p$; $t_2 = *t_1$; $t_2 = \&q$. Suppose the existing points-to relation is $p \rightarrow p_1 \rightarrow p_2 \rightarrow p_3$, we visualize the points-to graph of the transformed program as the left part of Figure 5.4.

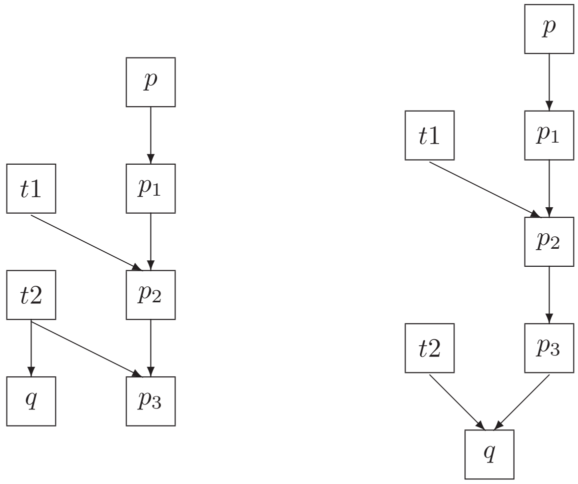


Figure 5.4: Pointer Graphs

As it shows, q is not in the points-to set of p_3 as desired and as a result the transformation does not guarantee correctness of the analysis. The reason is that the points-to set of t_2, which contains q, is not pointed by p_3. To solve the problem, we simply exchange the left hand side and the right hand side of the second last assignment. The resulting points-to graph of our example is shown at the right part of Figure 5.4 that shows the point-to relation of the resulting assignments are correct with respect to the original one. Finally we generalize the transformation by

$$\begin{aligned} t_1 &= *p \\ t_2 &= *t_1 \\ &\vdots \\ *t_{n-1} &= t_n \\ *t_n &= \&q \end{aligned}$$

Heap and Aggregations. We use a distinct analysis variable to model every heap object allocated at a particular program point. Aggregations, such as arrays and structs, are collapsed into one object and considered as a whole. This approach also simplifies the treatment of binary mathematical expressions in which the address of some individual member of the aggregation is accessed after mathematical calculations.

Indirect Function Calls. Functional pointers provide extra convenience for C programming whileas they also raise challenge for modeling and the pointer

$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} [\![\alpha]\!] + d \subseteq \beta$	iff	$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\alpha)) : \hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor + d) \subseteq \hat{\psi}_2(\hat{\psi}_1(\beta))$
$(\hat{\psi}_1, \hat{\psi}_2) \models_{\mathcal{T}} \alpha \subseteq [\![\beta]\!] + d$	iff	$\forall v \in \hat{\psi}_2(\hat{\psi}_1(\beta)) : \hat{\psi}_2(\hat{\psi}_1(\alpha)) \subseteq \hat{\psi}_2(\hat{\psi}_1(\lfloor v \rfloor + d))$

Table 5.7: Adjusted Constraints for Indirect Function Calls

analysis. In our implementation, indirect function calls are treated in the way described by Pearce et al. [PKH07]: program variables are encoded as numbers, and function arguments are numbered continuously after their corresponding function variable. Therefore they can be accessed as the offset to that function variable in the case of indirect function calls. For instance, consider the program and the meaning with respect to the pointer analysis:

(1)	int f(int $*p$){return p;}	$pts(p) \subseteq pts(f+1)$
(2)	$int(*y)$(int $*$) $= \&f$;	$\{f\} \subseteq pts(y)$

(3)	x = * y(z);	for each $v \in pts(y) : pts(z) \subseteq pts(v+2)$
		for each $v \in pts(y) : pts(v+1) \subseteq pts(x)$

where suppose all the variables have been encoded as integers. In the first line, we use the number right after f to hold the return value, i.e. $pts(p) \subseteq pts(f+1)$, and thus the number(s) of the arguments of the function start(s) with the number $f+2$. For the last statement, we need to calculate the actual argument according to the possible functions invoked. Last, $pts(x)$ acquires the possible return values.

The new requirement can be met by extending the two constraints $[\![\alpha]\!] \subseteq \beta, \alpha \subseteq [\![\beta]\!]$ with the plus operation, such as $[\![\alpha]\!] + d \subseteq \beta, \alpha \subseteq [\![\beta]\!] + d$. Their semantics are summarized in Table 5.7 in which we suppose the anonymous function $\lfloor \cdot \rfloor$ returns integers.

For deferences on pointers other than functional pointers, we simply set d to be 0. Although the inclusion constraint language is a little modified, it is straightforward to verify that the Moore family result and other theoretical properties of the language are preserved. Upon the constraint solving, the extra mathematical operations introduced by the adjusted constraints clearly do not affect the termination property and the complexity of the algorithm.

Library Function Calls. The source code of external library functions are not available to analyze. These library function calls are handled by hand-crafted function stubs: we read into the source code of each external library function to check if there is any side effect that may affect the pointer behavior of the programs of interest; if that is the case, we manually add the constraints for the library function.

5.6 Concluding Remarks

In this chapter, we have extended our constraint language in order to be able to deal with Andersen's pointer analysis [And94]. To motivate the development, we have given a brief introduction to pointer analysis and especially to the unification-based pointer analysis [Ste96] and inclusion-based pointer analysis [And94]. The redefined algorithm for solving the extended constraint language has also been specified, and its termination property and complexity have been studied accordingly. As the real C program is rich in the kinds of language features, we have made a detailed description of the way of treating the features interesting to our pointer analysis.

Chapter 6

Off-line Optimization Technique for the Inclusion Constraint Solver

In the previous chapters, we presented an inclusion constraint language. As demonstrated in Chapter 4, analysis designers can deliberate on which kinds of constraints to use in order to achieve a good balance between performance and precision. On the other hand, from the solver designers' point of view, automatic techniques could be introduced to analyze the inclusion constraints and identify potential equivalences (expressed in (dynamic) equality constraints). In particular, this approach can be conveniently applied to analyze very large systems, for example, hundred thousand lines of C programs.

In this chapter, we describe a strategy that given a set of constraints first performs off-line optimizations, which is conducted before the execution of the constraint solver. These off-line optimizations enable a constraint solver to find (potential) equivalences between analysis variables so as to reduce the problem space and thus improve performance both in time and space. As a matter of fact, different analyses use different subsets of constraints. Therefore, a specific property may hold for the subsets and a specific optimization can be performed on the constraints given by an analysis. Finally, since the off-line optimizations are conducted on constraints instead of analysis specifications, we can generalize the technique by detecting all the applicable optimizations for the given constraints in order to reuse the existing optimizations.

As an automatic technique, all the optimizations preserve the level of precision because analysis designers have no control on the optimization process and otherwise it is hard to track where the extra false positives actually occur.

We apply the strategy to our case studies, such as Andersen's pointer analysis, reaching definitions analysis, and live variable analysis for C programming language. The experimental result demonstrates that the off-line algorithms dramatically reduce the effort of solving the constraints, by identifying equivalent analysis variables. Part of the contents of this chapter was previously presented in [ZAN08].

6.1 Off-line Optimization Algorithms

In this section, we describe three off-line optimization algorithms. The aim of off-line optimization is to identify more equivalence relations between analysis variables in order to reduce the problem space. Specifically, the optimization generates more (dynamic) equality constraints, as well as more conditional equality constraints, i.e. conditional constraints having some (dynamic) equality constraints in their consequent. As the effort of the off-line optimization would be counted in the overall performance of constraint solving, we would like to keep the complexity of the off-line optimization algorithms as low as possible.

6.1.1 Optimization 1

Our first off-line optimization is motivated by pointer analysis. It builds an off-line version of the constraint graph for the constraints of interest, and accordingly detects the potential equivalent analysis variables by certain properties hold by the constructed graph, e.g. variables being on the same strongly connected components (SCCs). Five kinds of constraints are of interest, i.e. $\alpha \subseteq \beta, \alpha = \beta, [\![\alpha]\!] \subseteq \beta, \alpha \subseteq [\![\beta]\!]$, and $[\![\alpha]\!] = \beta$, because they are useful to generate the (dynamic) equality constraints desired. The (dynamic) equality constraints are helpful to form larger equivalent classes.

Similar to the technique of Hybrid Cycle Detection (HCD) described by Hard-

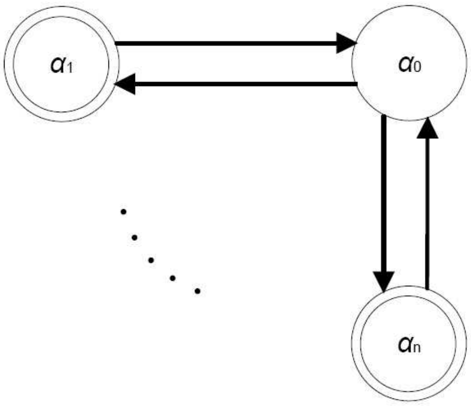
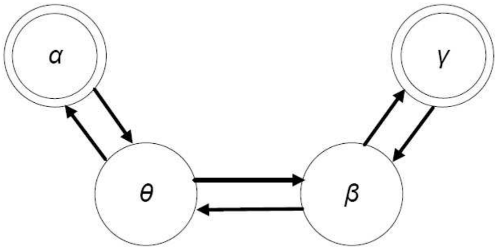


Figure 6.1: Building Off-line Constraint Graph: subcase 1.

ekopf and Lin [HL07a], an off-line version of the constraint graph is built from the five types of constraints: (1) the constraint of the form $\alpha \subseteq \beta$ gives two *normal nodes* α and β, and a directed edge from α to β; (2) The constraint of the form $\alpha = \beta$ gives two *normal nodes* α and β, and two directed edges between α and β; (3) the constraint of the form $[\![\alpha]\!] \subseteq \beta$ gives one *special node* $[\![\alpha]\!]$, one normal node β, and a directed edge from $[\![\alpha]\!]$ to β; (4) the constraint of the form $\alpha \subseteq [\![\beta]\!]$ gives one normal node α, one special node $[\![\beta]\!]$, and a directed edge from α to $[\![\beta]\!]$; (5) the constraint of the form $[\![\alpha]\!] = \beta$ gives one *special node* $[\![\alpha]\!]$, one normal node β, and two directed edges between $[\![\alpha]\!]$ and β. Note that we consider α and $[\![\alpha]\!]$ as two different nodes: the node $[\![\alpha]\!]$ is considered as a node in its own right.

The algorithm then identifies any SCCs, which consists of more than two nodes, in the constraint graph using Tarjan's linear-time algorithm [Tar72]. For the SCCs containing only normal nodes, the algorithm selects one node from these nodes and let all others be equivalent to it using equality constraints. For any SCCs containing not only normal nodes, it must at least have one special node and one normal node because there is no constraint of the form $[\![\alpha]\!] \subseteq [\![\beta]\!]$. We classify the situation into two subcases: only one normal node in the SCC, and otherwise. For the first subcase, each special node, say $[\![\alpha]\!]$, must form a sub-SCC together with the unique normal node, say β, and has no more edges from or to any other special nodes of the SCC. The constraint graph of such a SCC is illustrated as Figure 6.1, where we use a double circle to represent a special node. We therefore generate the dynamic equality constraint $[\![\alpha]\!] = \beta$ for each of such pairs.

For the second subcase, the algorithm yields a conditional constraint: its precondition tests the emptiness of each special nodes of the SCC, and its consequent chooses one of the normal nodes from the SCC and lets all other nodes be (dynamically) equivalent to it. If there is any sub-SCCs in the SCC considered, the precondition may be too strict. For example, consider a off-line constraint graph as



where two sub-SCCs exist in a parent SCC. As a result, the (dynamic) constraints, $[\![\alpha]\!] = \theta, [\![\gamma]\!] = \beta$, and $\theta = \beta$ can be yielded directly without testing the emptiness of the analysis variables α and γ. Under such a situation, a more complex analysis than the one we present might remove some unnecessary check. The reason why we choose not to detect these sub-SCCs is because we want to keep the algorithm as efficient as possible and a little too strict precondition might miss some equivalences but do not cause any loss in precision.

Consider the constraint program of Example 5.6, Figure 6.2 illustrates its off-line version of the constraint graph. The algorithm described as above will yield a conditional constraint $[\kappa_x, \kappa_z] \Rightarrow ([\![\kappa_x]\!] = \kappa_y \wedge [\![\kappa_z]\!] = \kappa_y \wedge \kappa_m = \kappa_y)$. Here the dynamic equality constraints, i.e. the conditional constraint and its postcondition, provide hints on where a SCC may complete and be collapsed during the computation of the solver. Therefore the test of emptiness on the special nodes $[\![\kappa_x]\!]$ and $[\![\kappa_z]\!]$ guarantees the existence of the SCC.

Comparison to HCD. The major difference between HCD and our off-line optimization is the way of treating SCCs which have two or more normal nodes and at least one special node. HCD selects one normal node α and for each special node $[\![\beta]\!]$, stores the pair $(\alpha, [\![\beta]\!])$ in a list that serves the same purpose as dynamic equality constraints. Our off-line optimization refines the technique by (1) making more complete use of the SCC result by also doing unification between all normal nodes and (2) using conditional constraints to ensure that the level of precision will not be impacted by the use of unification. For the first refinement, notice that the equivalences omitted by HCD could be detected in isolation by other on-line cycle detection algorithms, such as Heintze and Tardieu's algorithm [HT01], which could run together with HCD. However, since these equivalences can be easily detected by our off-line analysis with very

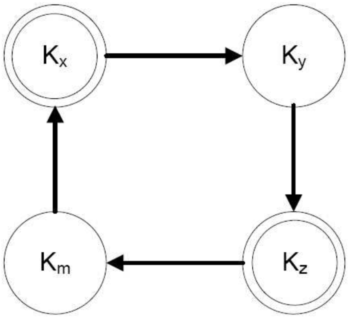


Figure 6.2: Off-line Constraint Graph of Example 5.6

low cost, we save the effort of finding them again by an on-line cycle detection algorithm. Take for instance the constraint of Example 5.6, where HCD does not consider the possibility of unifying κ_y and κ_m, but rather generates the pairs $(\kappa_y, [\![\kappa_x]\!])$ and $(\kappa_y, [\![\kappa_z]\!])$ so that κ_x's points-to set will be unified with κ_y. However, when κ_z's points-to set is empty, the SCC expected does not complete and unifying any variables on the component may yield a loss of precision, e.g. $\kappa_p \rightarrow \{i\}$ is a false positive introduced by unifying κ_p and κ_y. In the C language, a run-time error is normally reported for a dereference over a non-initialized pointer variable. However no error pops up if the dereference occurs at any unreachable code. These unreachable codes are often caused by either deprecated functions, that is functions that are suppressed by new ones but still remain in the current version of a software, or by conditionals or loops where the test is always true or always false. The use of conditional constraints can therefore preserve the level of precision in analyzing real programs.

Complexity. For the optimization 1, detecting SCCs takes linear time and so does generating new constraints. Therefore the overall complexity of the optimization 1 is linear. Note that the size of new generated constraints is linear to the constraint program size.

6.1.2 Optimization 2

The second off-line optimization analyzes the direct dependency of data flow between analysis variables in order to determine which set-inclusion constraint $\alpha \subseteq \beta$ can be changed to equality constraint $\alpha = \beta$ without sacrificing precision. This is illustrated by the following example.

Example 6.1 Consider the constraint program

$$\{a, b\} \subseteq \kappa_p \wedge \{c\} \subseteq \kappa_r \wedge \kappa_p \subseteq \kappa_q \wedge \kappa_q \setminus \{b\} \subseteq \kappa_r$$

The least solution of the example program is that $\hat{\psi}_2(\hat{\psi}_1(\kappa_p)) = \hat{\psi}_2(\hat{\psi}_1(\kappa_q)) = \{a, b\}$ and $\hat{\psi}_2(\hat{\psi}_1(\kappa_r)) = \{a, c\}$. The analysis variable κ_q shares the same data with the analysis variable κ_p because it gets all its values from κ_p only. □

Example 6.2 Consider the constraint program

$$\{a, b\} \subseteq \kappa_p \wedge \{c\} \subseteq \kappa_r \wedge \kappa_p \subseteq \kappa_q \wedge \kappa_r \subseteq [\![\kappa_s]\!] \wedge \{q\} \subseteq \kappa_s$$

The least solution of the example program is that $\hat{\psi}_2(\hat{\psi}_1(\kappa_p)) = \{a, b\}$, $\hat{\psi}_2(\hat{\psi}_1(\kappa_q)) = \{a, b, c\}$, $\hat{\psi}_2(\hat{\psi}_1(\kappa_r)) = \{c\}$, and $\hat{\psi}_2(\hat{\psi}_1(\kappa_s)) = \{q\}$. This time the analysis variables κ_p and κ_q do not have the same data because $[\![\kappa_s]\!]$ of the constraint $\kappa_r \subseteq [\![\kappa_s]\!]$ can be instantiated to κ_q that yields the set-inclusion constraint $\kappa_r \subseteq \kappa_q$. □

As demonstrated by the second example, it is in general not sound to replace $\kappa_p \subseteq \kappa_q$ with $\kappa_p = \kappa_q$ in case that there is any dynamic constraint.

Our second optimization therefore aims at the constraint programs *without* dynamic constraints: It first scans the program and changes any constraint of the form $\alpha \subseteq \beta$ to $\alpha = \beta$ if β only appears once at the right hand side of this particular inclusion constraint and never shows up at either side of any equality constraints. Note that we also need to look into the postcondition of any conditional constraint to make sure none of postfixes violates the condition.

Complexity. The second algorithm scans a constraint program twice. In the first round, the algorithm counts the number of appearances of each program variable in inclusion and equality constraints. In the second round, it changes some set-inclusion constraints to equality according to the result from the first round. The operations of each round is linear in the size of the constraint program and thus the overall complexity of the whole algorithm is linear.

6.1.3 Optimization 3

Optimization 3 could be considered as a variant of optimization 2 and the idea is motivated from pointer analysis. We change the constraint $\kappa_p \subseteq \kappa_q$ to $\kappa_p = \kappa_q$ when each of the following three conditions is satisfied:

(1) κ_q only appears once at the right hand side of this particular inclusion constraint and never shows up at either side of any (dynamic) equality constraints. As the Optimization 2 we also need to look into the postcondition of any conditional constraint to make sure none of postfixes violates the condition.

(2) q is not pointed to by any other pointers at all, i.e. the operation $[\![\cdot]\!]$ can not give the analysis variable κ_q with respect to a least solution of a given program.

(3) when functional pointers are concerned, the analysis may generate the constraint of the form $\alpha \subseteq [\![\beta]\!]+k$. The off-line analysis needs furthermore to ensure that κ_q may not be acquired from the mathematic calculation $[\![\beta]\!] + k$.

To check the condition (2), we simply collect all the tuples a given by the constraint $\{a\} \subseteq \kappa_s$ which may appear as an individual constraint or as a postfix of a conditional constraint. The set of analysis variables S, which correspond to these tuples respectively, are those that may be pointed to (with respect to a least approximation).

Based on the above result, we first chose the maximum k_{max} from all the constraints of the form $\alpha \subseteq [\![\beta]\!] + k$, then for each analysis variable $\gamma \in S$ (encoded as an integer) we calculate the possible value of $[\![\beta]\!] + k$ by $\gamma + i$ where $i \in [1, .., k_{max}]$. This allows us to check the third condition.

Complexity. The third algorithm scans a constraint program two times. In the first round, the algorithm counts the number of appearances of each program variable in inclusion and equality constraints; at the same time it collects the information about the analysis variables that may be pointed to and the maximum k_{max} form all constraints of the form $\alpha \subseteq [\![\beta]\!] + k$. Accordingly we calculate the set of analysis variables that may be yielded by $[\![\beta]\!] + k$. In the second round, the algorithm changes some set-inclusion constraints to equality by checking the above three conditions according to the results from the first round. The operations of each round is linear in the size of the constraint program and thus the overall complexity of the whole algorithm is still linear.

6.1.4 General Strategy of Using Off-line Optimization

The optimizations described above is independent of any specific analysis. Thus we can generalize the strategy as the following steps:

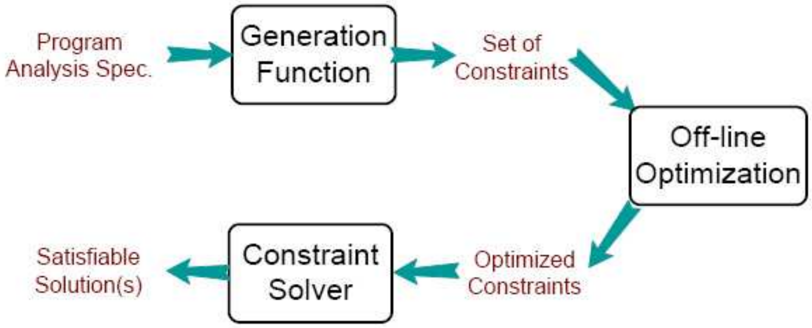


Figure 6.3: Extended framework of implementing program analyses using constraint solver and off-line optimization.

(1) Collect the subset of constraints in a given program;

(2) Determine the proper optimization(s) to apply;

(3) Perform off-line optimization(s);

(4) Solve the resulting constraint program with the inclusion constraint solver.

With this generalization, we would like to encourage the reuse of an existing optimization algorithm (that is in general motivated by a specific analysis) on more program analyses. The process is linear in the size of constraint programs: the first step makes a linear scan on a constraint program, and the second step needs constant operations to make the choice. We therefore extend the framework of implementing program analyses by introducing a step of off-line optimization as demonstrated in Figure 6.3.

As presented in later sections, some off-line analyses may not improve the performance of the solver. However, if the off-line optimizations maintain the complexity of linearity, running all applicable optimizations should be non-noticable for large benchmarks. For small programs, off-line analyses can be dispensed with since the original constraints can be easily solved by the solver itself.

When there are two or more optimizations applicable, the order of applying them may be interesting to consider. In the experimental study presented in the next section, we shall try the different orders and give further discussion based on the experimental results.

6.2 Experimental Study

To evaluate the effects of the off-line technique, we implemented the algorithm of the three off-line optimizations in SML New Jersey. We evaluate the effect of using these optimizations on three analysis for the C programming language: an interprocedural flow-insensitive and context-insensitive pointer analysis, originally due to Lars Ole Andersen [And94], and two intraprocedural data flow analyses, a reaching definitions analysis and a live variable analysis.

To be concrete, we yield inclusion constraints from C programs using the C Intermediate Language (CIL) tool [NMRW02]. Our implementation supports all features of the C language except for variable arguments.

As in Chapter 4, all the benchmarks are run on a 2.0 GHz processor with 1.5 GB of memory under Windows XP SP2. Each experiment is repeated three times and the average time and minimum memory consumption are reported.

6.2.1 Case Study: Andersen's Pointer Analysis

Andersen's analysis [And94] is an inclusion-based pointer analysis that is a prerequisite for many program analyses. In Chapter 5 we have discussed the major implementation considerations for the analysis.

Upon to the worklist strategy, we shall first use the least recent fire (LRF) to prioritize the worklist elements (refer back to Subsection 5.4.3 for the introduction of LRF). To be comparative, we shall also conduct experiment using the last-in-first-out (LIFO) with a clear mark.

6.2.1.1 Evaluation

By inspecting the types of constraints used in Andersen's pointer analysis, we can apply the first and third off-line optimizations, but not the second as explained in Section 6.1.2. The benchmarks for Andersen's pointer analysis are presented in Table 6.1. ‘grep’ is a text editor; ‘make’ is a building tool; ‘gawk’ is a string manipulation tool; ‘cvs’ is a version control software; ‘bash’ is a sh-compatible shell. As the table shows, the amount of constraints does not imply the length of solving time, e.g. ‘cvs’ yields 15,644 constraints that is more than that of ‘gawk’ but can be solved much faster than gawk. This is because new constraints could generate during the calculation over dynamic constraints.

Program	**LOC**	**Constraint**	$\mathbf{T_{LRF}}$(ms.)	$\mathbf{M_{LRF}}$(MB)
grep2.5.3	21, 827	4, 880	171	6.5
make3.81	26, 872	7, 224	7610	89.2
gawk3.1.5	35, 231	14, 742	20797	214.7
cvs1.11.21	82, 317	15, 644	3625	54.7
bash3.2	98, 603	20, 988	32500	166.2

Table 6.1: Benchmarks for Andersen's pointer analysis: for each benchmark the table shows the number of lines of code (with all comment lines removed), the number of constraints generated using CIL, the time and memory performance of solving the original constraint program.

Program	**Eq**(off1)	**Dyn. Eq.**	**Cond. Eq.**	$\mathbf{T_{off1}}$(ms.)	**Eq**(off3)	$\mathbf{T_{off3}}$(ms.)
grep2.5.3	38	28	27	18	489	5
make3.81	73	25	25	22	542	21
gawk3.1.5	89	60	75	52	1260	32
cvs1.11.21	98	38	27	52	1086	71
bash3.2	457	113	85	151	3314	58

Table 6.2: The results of applying the optimization 1 or 3: for each benchmark we present the number of three kinds of constraints generated by the optimization 1 individually, the equality constraints given by the optimization 3 individually, and the time of performing the two optimizations individually.

We then perform the optimization 1 and the optimization 3 individually and summarize the results in Table 6.2. As the table shows, for the optimization 1 more equality constraints are yielded than dynamic equality constraints. The amount of conditional constraints generated is quite close to that of dynamic equality constraints. For the case of 'gawk', there are even more conditional constraints than dynamic equality constraints. The optimization 3 generates quite a number of equality constraints for each benchmark. For each benchmark the time of executing both of the two off-line optimizations is very short compared to that of the constraint solving.

Table 6.3 and 6.4 report the effect of using the two off-line optimizations respectively in terms of time and space consumption. For the optimization 1, we observe that the performance significantly improves for each benchmark both in terms of time and memory consumption. The data for the optimization 3 is quite surprising: even though the problem space is reduced and the labeled

Program	$\mathbf{T_{opt1}}$(ms.)	$\mathbf{\Delta T_{opt1}}$	$\mathbf{M_{opt1}}$(MB)	$\mathbf{\Delta M_{opt1}}$
grep2.5.3	125	27%	4.9	25%
make3.81	4,625	39%	65	27%
gawk3.1.5	13,766	34%	143.2	33%
cvs1.11.21	3,047	16%	49.3	9.9%
bash3.2	17,610	46%	141.7	15%
Average	–	33%	–	18%

where:
$\Delta T_{opt1} = 1 - T_{opt1}/T_{LRF}$
$\Delta M_{opt1} = 1 - M_{opt1}/M_{LRF}$

Table 6.3: Performance evaluation using the optimization 1: for each benchmark we present the time and memory consumption after applying the first off-line optimization. The average of the improvement is summarized at the last line.

Program	$\mathbf{T_{opt3}}$(ms.)	$\mathbf{\Delta T_{opt3}}$	$\mathbf{M_{opt3}}$(MB)	$\mathbf{\Delta M_{opt3}}$
grep2.5.3	114	33%	3.5	46%
make3.81	7,766	−2%	58.4	35%
gawk3.1.5	18,812	10%	187.8	13%
cvs1.11.21	4,172	−15%	53.8	2%
bash3.2	24,021	26%	156.8	6%
Average	–	12%	–	18%

where:
$\Delta T_{opt3} = 1 - T_{opt3}/T_{LRF}$
$\Delta M_{opt3} = 1 - M_{opt3}/M_{LRF}$

Table 6.4: Performance evaluation using the optimization 3: for each benchmark we present the time and memory consumption after applying the third off-line optimization. The average of the improvement is summarized at the last line.

cluster is simplified by the use of unification detected by the optimization 3, the performance of two benchmarks, 'make' and 'cvs', is slowed down. Why is that? Observing that a lot of extra calls to the add procedure happen on the two benchmarks, we postulate that the changes to the constraint program also affect the order of solving the constraints and cause performance deterioration. This phenomenon reminds us that the size of problem space is just one major parameter which can affect the performance of a solver. Different worklist strategies can also have considerable impact on a solver's performance. To verify our postulation, we evaluate the time performance on the solver using LIFO

Program	$\mathbf{T_{LIFO}}$(ms.)	$\mathbf{T'_{opt1}}$(ms.)	$\mathbf{\Delta T'_{opt1}}$	$\mathbf{T'_{opt3}}$(ms.)	$\mathbf{\Delta T'_{opt3}}$
grep2.5.3	141	99	30%	110	22%
make3.81	9,875	9,063	8%	9,156	7%
gawk3.1.5	19,567	19,333	1%	19,375	1%
cvs1.11.21	3,656	3,860	−6%	3,938	−8%
bash3.2	25,312	25,750	−2%	26,297	−4%
Average	–	–	1%	–	−1%

where:
$\Delta T'_{opt1} = 1 - T'_{opt1}/T_{LIFO}$
$\Delta T'_{opt3} = 1 - T'_{opt3}/T_{LIFO}$

Table 6.5: Performance evaluation for the solver using LIFO worklist strategy: for each benchmark we present the time performance before and after applying the first off-line optimization or the third off-line optimization. The average of the improvement is summarized at the last line.

worklist strategy (presented in Table 6.5). The space performance is omitted for the reason of simplicity. The solver's source code is reused maximally to provide a fair comparison.

The second column of Table 6.5 presents the running time before using any off-line optimization. When the optimization 1 is applied, the performance of 'cvs' and 'bash' has performance deterioration this time while as their performance is significantly improved using the same optimization in the case of LRF. When the optimization 3 is concerned, the performance of 'make' is improved in contrast to what is observed in the case of LRF. The performance of 'cvs' is still slowed down after using the optimization but it is not that significant as before. However, 'bash' experiences a performance deterioration this time. As a result, we conclude that a worklist strategy can significantly affect the effect of the off-line optimization for Andersen's pointer analysis.

Next we consider to execute both of the optimization 1 and the optimization 3, and see its effect on the performance. Table 6.6 and 6.7 present the performance for the two possible orders of executing the optimizations when LRF is used.

As Table 6.6 shows, a significant improvement is achieved for each of the benchmarks. Performing the optimization 1 and then 3 is faster than performing any one of them individually. This means the equivalent classes detected by off-line optimization can be made good use of by the solver with LRF. When the order of running the optimizations is reversed, the performance improvement, as presented in Table 6.7, is not as large as before although it still improves a lot.

Program	$\mathbf{T_{opt1,3}}$(ms.)	$\mathbf{\Delta T_{opt1,3}}$	$\mathbf{M_{opt1,3}}$(MB)	$\mathbf{\Delta M_{opt1,3}}$
grep2.5.3	104	39%	2.9	55%
make3.81	4,906	36%	58.6	34%
gawk3.1.5	13,375	36%	141	34%
cvs1.11.21	2,969	18%	51.9	5%
bash3.2	15,625	52%	147.8	11%
Average	–	37%	–	18%

where:
$\Delta T_{opt1,3} = 1 - T_{opt1,3}/T_{LRF}$
$\Delta M_{opt1,3} = 1 - M_{opt1,3}/M_{LRF}$

Table 6.6: Performance evaluation using both the optimization 1 and the optimization 3: for each benchmark we present the time after applying the two off-line optimizations with two different orders. The subscript **opt1, 3** represents that the optimization 1 is first performed. The average of the improvement is summarized at the last line.

Program	$\mathbf{T_{opt3,1}}$(ms.)	$\mathbf{\Delta T_{opt3,1}}$	$\mathbf{M_{opt3,1}}$(MB)	$\mathbf{\Delta M_{opt3,1}}$
grep2.5.3	99	42%	3.1	52%
make3.81	4,910	35%	54.4	39%
gawk3.1.5	13,734	34%	144.8	33%
cvs1.11.21	2,985	18%	56.8	4%
bash3.2	22,391	31%	157.9	5%
Average	–	29%	–	13%

where:
$\Delta T_{opt3,1} = 1 - T_{opt3,1}/T_{LRF}$
$\Delta M_{opt3,1} = 1 - M_{opt3,1}/M_{LRF}$

Table 6.7: Performance evaluation using both the optimization 1 and the optimization 3: for each benchmark we present the time after applying the two off-line optimizations with two different orders. The subscript **opt3, 1** represents that the optimization 3 is first performed. The average of the improvement is summarized at the last line.

Overall speaking, the improvement of running both of the two optimizations is smaller than running only the optimization 1. Comparing the data in these two tables, we conclude that the order of performing the two off-line optimizations has noticeable effect on the performance of the solver. For the solver using LRF, performing the optimization 1 first and then the optimization 3 is better than

Program	$\mathbf{T'_{opt1,3}}$(ms.)	$\mathbf{\Delta T'_{opt1,3}}$	$\mathbf{T'_{opt3,1}}$(ms.)	$\mathbf{\Delta T'_{opt3,1}}$
grep2.5.3	109	23%	100	29%
make3.81	4,469	55%	4,453	55%
gawk3.1.5	14,839	24%	14,203	27%
cvs1.11.21	2,927	20%	2,969	19%
bash3.2	27,016	−7%	18,735	26%
Average	–	14%	–	27%

where:

$\Delta T'_{opt1,3} = 1 - T'_{opt1,3}/T_{LIFO}$

$\Delta M_{opt1,3} = 1 - T'_{opt3,1}/T_{LIFO}$

Table 6.8: Performance evaluation for the solver using LIFO worklist strategy: for each benchmark we present the time performance before and after applying the first off-line optimization or the third off-line optimization. The average of the improvement is summarized at the last line.

the other way round.

For the solver using LIFO worklist strategy, we observe an opposite result. As shown in Table 6.8, performing the optimization 3 and then the optimization 1 is much faster than the opposite order. And this time no matter which order is chosen, executing both of the optimizations is faster than only executing one of the two optimizations. Therefore, we conclude that the best order of performing optimizations may vary according to the specific worklist strategy, i.e. the order itself can not guarantee a best result.

Finally we compare the fastest solver performance of using LRF and LIFO individually, i.e. the values of $T_{opt1,3}$ and $T'_{opt3,1}$, by normalizing the value of $T'_{opt3,1}$ against that of $T_{opt1,3}$ for each benchmark. The result is presented as in Figure 6.4. On average, using LRF is 1.27 times faster than using LIFO.

To conclude, we have studied the effect of using off-line optimizations on Andersen's pointer analysis using our constraint solver. The experimental results demonstrated that beside the use of unification, a worklist strategy can also significantly affect the efficiency of a solver. Thus in order to take fully advantage of the result of off-line analyses, it is worthwhile to spend time on selecting a proper worklist strategy.

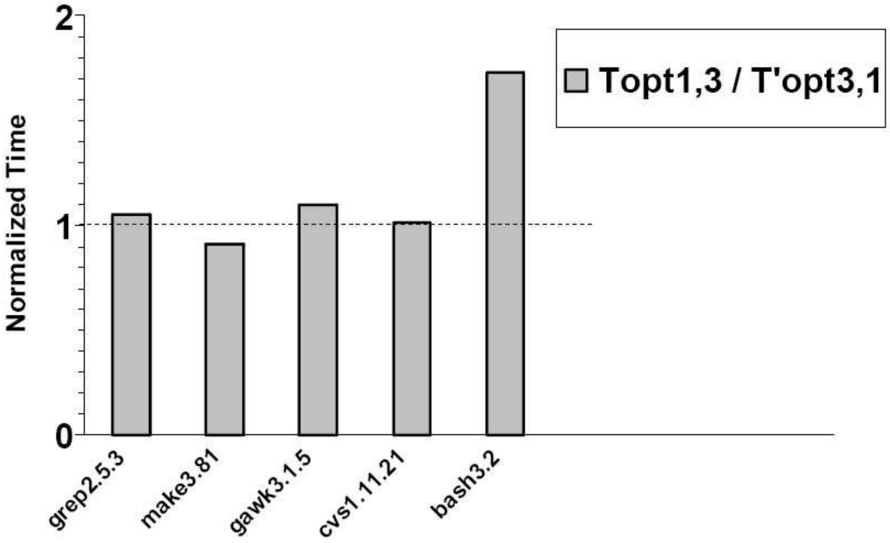


Figure 6.4: Performance comparison of the solver using LRF and LIFO individually: for each benchmarks the time of $T_{opt1,3}$ is normalized against that of $T'_{opt3,1}$, where $T_{opt1,3}$ is acquired by adopting the worklist strategy LRF and $T'_{opt3,1}$ is acquired by adopting the worklist strategy LIFO.

6.2.2 Case Study: Reaching Definitions Analysis

In this section, we present an intraprocedural reaching definitions analysis for the C language. As stated in Chapter 4, for each program point the analysis estimates which assignments may have been made and not overwritten by other assignments along some execution path. Each elementary block is assigned a unique label $l \in \mathbf{Lab}$, and $\circ$ and $\bullet$ denote entry and exit of a elementary statement respectively. The two functions initial and final are standard and can be defined by a straightforward extension to the definitions in Table 4.4 and 4.5. As before, we use the pair $(x, l) \in \mathbf{ProgVar} \times \mathbf{Lab}$ to denote that the program variable x may be defined at the assignment labeled as l. Instead of giving an analysis specification (which is quite straightforward), we present the implementation of the analysis directly by specifying a generation function $\mathcal{G}$ as presented in Table 6.9, which takes program code as input and returns constraint programs as output.

For reasons of simplicity, we assume that the level of the dereference operation has been reduced to one on the left-hand side of each assignment. The function

$$
\begin{array}{lcl}
\mathcal{G}([x = e]^l) & = & l_\circ \setminus \{(x, ?) \subseteq l_\bullet \wedge \\
 & & \{(x, l)\} \subseteq l_\bullet \\
\mathcal{G}([*x = e]^l) & = & \wedge_{\forall v \in pts(x)} \{(v, l)\} \subseteq l_\bullet \\
 & & l_\circ \subseteq l_\bullet \\
\mathcal{G}([e]^l) & = & l_\circ \subseteq l_\bullet \\
\mathcal{G}(S_1; S_2) & = & \textbf{Let } l' = \mathsf{init}(S_2) \\
 & & \textbf{in } \ \mathcal{G}(S_1) \ \wedge \mathcal{G}(S_2) \wedge \\
 & & \quad \wedge_{\forall l \in \mathsf{final}(S_1)} l_\bullet \subseteq l'_\circ \\
\mathcal{G}(\text{while } [e]^l \text{ do } S) & = & \textbf{Let } l' = \mathsf{init}(S) \\
 & & \textbf{in } \ \mathcal{G}([e]^l) \ \wedge \ \mathcal{G}(S) \wedge \\
 & & \quad l_\bullet \subseteq l'_\circ \wedge \\
 & & \quad \wedge_{\forall l'' \in \mathsf{final}(S)} l''_\bullet \subseteq l_\circ \\
\multicolumn{3}{l}{\mathcal{G}(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2) =} \\
 & & \textbf{Let } l' = \mathsf{init}(S_1) \\
 & & \quad l'' = \mathsf{init}(S_2) \\
 & & \textbf{in } \ \mathcal{G}([e]^l) \ \wedge \ \mathcal{G}(S_1) \ \wedge \ \mathcal{G}(S_2) \wedge \\
 & & \quad l_\bullet \subseteq l'_\circ \ \wedge \ l_\bullet \subseteq l''_\circ
\end{array}
$$

Table 6.9: Generation Function for Reaching Definitions Analysis

pts uses the result of Andersen's pointer analysis to estimate the points-to set of each program variable. Several representative statements are selected to illustrate the process and the rest can be handled in the same manner or very similarly. With the use of CIL, the cases we need to consider are much simplified: side-effects of expressions are turned into explicit assignments, e.g. the statement $x = i++$ is transformed into the three statements $tmp = i; i++; x = tmp$; the conditional operator 'exp1?exp2:exp3' is compiled into explicit conditionals, etc.

Observing that there is no dynamic constraint generated for the analysis, we apply the optimization 1 and 2. Especially, for the first optimization, all SCCs contain only normal nodes this time.

Program	**LOC**	**Constraints**	$\mathbf{T^{RD}}$(ms.)	$\mathbf{M^{RD}}$(MB)
phonebook1.0	968	$1,505$	9	3.7
make3.81: main	$2,370$	$2,731$	62	5.4
wdiff0.5	$2,627$	$2,132$	26	4.3
gnuchess	$9,668$	$19,641$	$1,875$	23.1
grep2.5.3	$21,827$	$25,755$	$2,890$	53.0

Table 6.10: Benchmarks: For each benchmark the table shows the number of lines of code (with all comment lines removed), the number of constraints generated using CIL, the time and memory performance of solving the original constraint program.

Program	$\mathbf{Eq^{RD}}$(off2)	$\mathbf{T^{RD}_{off2}}$(ms.)
phonebook1.0	831	1
make3.81:main	769	2
wdiff0.5	$1,123$	2
gnuchess	$10,122$	22
grep2.5.3	$12,782$	34

Table 6.11: The results of applying the optimization 2: for each benchmark we present the number of the equality constraints yielded by the optimization 2, and the time of performing the optimization.

6.2.2.1 Evaluation

For evaluating our approach on reaching definitions analysis we have to choose a set of medium-sized programs: phonebook is an application; main is the largest function of make; wdiff compares two files; gnuchess is a game software; grep is as before. The reason why we can not handle large benchmarks is that they usually generate too many constraints to be handled by our solver; this is due to the flow-sensitive nature of the analysis and its imprecision inherited from the pointer analysis. For example, the number of constraints generated from make is 1,978,605. But since reaching definitions analysis is an intraprocedural analysis, it suffices to analyze each function individually and then simply combine the results to get the whole analysis information [1]. The benchmarks used here are up to 21,827 LOC which is large enough for scaling to most functions.

[1] No set union is needed for each analysis variable if each label is unique.

Program	$\mathbf{T^{RD}_{opt2}}$(ms.)	$\mathbf{\Delta T^{RD}_{opt2}}$	$\mathbf{M^{RD}_{opt2}}$(MB)	$\mathbf{\Delta M^{RD}_{opt2}}$
phonebook1.0	7	29%	3.6	3%
make3.81: main	47	24%	5.3	2%
wdiff	16	38%	3.8	12%
gnuchess	$1,312$	30%	18.6	19%
grep2.5.3	$2,890$	44%	45.8	14%
Average	–	38%	–	14%

where:
$\Delta T^{RD}_{opt2} = 1 - T^{RD}_{opt2}/T^{RD}$
$\Delta M' = 1 - M^{RD}_{opt2}/M^{RD}$

Table 6.12: Performance evaluation: for each benchmark we present the the time and memory consumption before and after applying the *second* off-line optimization only. The average of improvement is summarized in the last line.

Table 6.10 describes the benchmarks and presents the performance of our solver on original constraint programs. Table 6.11 reports the number of equality constraints generated by the second off-line optimization and the execution time of running the optimization. The optimization 1 finds very few equivalences, up to 15 equalities – the lack of cycles could be explained by the fact that it is very rare that in a real C program no assignment happens in a while-loop. We therefore report the performance improvement of using the second optimization only, and the results are summarized in Table 6.12.

Note that we adopt the LIFO worklist strategy this time: it is simple, but up to 10% more efficient than a priority queue. The time of performing the optimization on each benchmark is very short compared to the overall execution time. We observe that the more unification the optimization identifies, the better performance the solver achieves. The overall performance is improved by 38% on time and by 14% on memory consumption without any loss of precision. The number of equality constraints generated by the off-line optimization demonstrates that many equivalent analysis variables do exist in the analysis. Since identifying these equivalences is linear time, we conclude that the off-line optimization can be used to speed up the constraint solving for the reaching definitions analysis significantly.

$$
\begin{aligned}
\mathsf{U}(n) &= \{\} \\
\mathsf{U}(x) &= \{x\} \\
\mathsf{U}(*x) &= \{x\} \cup pts(x) \\
\mathsf{U}(op_{bin}(e_1, e_2)) &= \mathsf{U}(e_1) \cup \mathsf{U}(e_2) \\
\mathsf{U}(op_{un}(e)) &= \mathsf{U}(e)
\end{aligned}
$$

Table 6.13: Definition of function U.

6.2.3 Case Study: Live Variable Analysis

In Subsection 6.1.4, we discussed the possibility that the insights of the optimizations can be shared by different analyses. In fact, the strategy of off-line optimization encourages to apply an off-line optimization (which is motivated by a specific analysis) on other analyses. To be illustrative, in this subsection we present an implementation of a live variable analysis for the C programming language. The possibility of reusing off-line optimization(s) on the analysis is discussed accordingly.

A *live variable analysis* estimates for each program point, which variable could still be used later. The implementation is similar to that for reaching definitions analysis. A new function **U** (in Table 6.13) is introduced to acquire a set of program variables from a given expression. As shown in the table, the function needs to use the result of a pointer analysis to calculate the possible live variables used in an expression.

The generation function for the live variable analysis is defined in Table 6.14. For the case $*x = e$, the constraint $l_\bullet \subseteq l_\circ$ copies the whole information from the exit to the entry without removing any element from $l_\circ$. This is because pointer analysis is a may-analysis and $*x$ could even be NULL in the case of un-reachable code. Therefore we do not know which variable is redefined at the assignment and have to assume all are active to ensure the correctness of the analysis.

6.2.3.1 Evaluation

We choose the same set of benchmarks used in the reaching definitions analysis for evaluation. The LIFO worklist strategy is adopted considering the live

$$
\begin{array}{lcl}
\mathcal{G}([x = e]^l) & = & \wedge_{y \in \mathbf{U}(e)} \{ y \} \subseteq l_\circ \ \wedge \\
 & & l_\bullet \setminus \{x\} \subseteq l_\circ \\[1ex]
\mathcal{G}([*x = e]^l) & = & \wedge_{y \in \mathbf{U}(e)} \{ y \} \subseteq l_\circ \ \wedge \\
 & & \{x\} \subseteq l_\circ \ \wedge \\
 & & l_\bullet \subseteq l_\circ \\[1ex]
\mathcal{G}([e]^l) & = & \wedge_{x \in \mathbf{U}(e)} \{ x \} \subseteq l_\circ \ \wedge \\
 & & l_\bullet \subseteq l_\circ \\[1ex]
\mathcal{G}(S_1; S_2) & = & \textbf{Let } l' = \mathsf{init}(S_2) \\
 & & \textbf{in} \quad \mathcal{G}(S_1) \ \wedge \mathcal{G}(S_2) \ \wedge \\
 & & \qquad \wedge_{\forall l'' \in \mathsf{final}(S_1)} l'_\circ \subseteq l''_\bullet \\[1ex]
\mathcal{G}(\text{while } [e]^l \text{ do } S) & = & \textbf{Let } l' = \mathsf{init}(S) \\
 & & \textbf{in} \quad \mathcal{G}([e]^l) \ \wedge \ \mathcal{G}(S) \ \wedge \\
 & & \qquad l'_\circ \subseteq l_\bullet \ \wedge \\
 & & \qquad \wedge_{\forall l'' \in \mathsf{final}(S)} l_\circ \subseteq l''_\bullet \\[1ex]
\multicolumn{3}{l}{\mathcal{G}(\text{if } [e]^l \text{ then } S_1 \text{ else } S_2) =} \\
 & & \textbf{Let } l' = \mathsf{init}(S_1) \\
 & & \qquad l'' = \mathsf{init}(S_2) \\
 & & \textbf{in} \quad \mathcal{G}([e]^l) \ \wedge \ \mathcal{G}(S_1) \ \wedge \ \mathcal{G}(S_2) \ \wedge \\
 & & \qquad l'_\circ \subseteq l_\bullet \ \wedge \ l''_\circ \subseteq l_\bullet
\end{array}
$$

Table 6.14: Generation Function for Live Variable Analysis

variable analysis uses the same subset of constraints as the reaching definitions analysis and they belong to the same class of program analyses. Table 6.15 presents the number of constraints generated for each benchmark, and the performance of the solver on the original constraint programs in terms of time and space consumption.

We then execute the off-line optimizations. For the first off-line optimization, it detects quite few equivalences, e.g. it yields 82 equality constraints out of 32,607 constraints for grep. This is because live variable information is updated

Program	**Constraints**	$\mathbf{T^{LV}}$(ms.)	$\mathbf{M^{LV}}$(MB)
phonebook1.0	2,420	79	4.7
make3.81: main	3,202	688	4.1
wdiff0.5	2,487	94	4.3
gnuchess	26,271	3,157	22.7
grep2.5.3	32,607	4,704	28.6

Table 6.15: Benchmarks: For each benchmark the table shows the number of constraints generated using CIL, the time and memory performance of solving the original constraint program.

Program	$\mathbf{Eq^{LV}}$(off2)	$\mathbf{T^{LV}_{off2}}$(ms.)
phonebook1.0	589	2
make3.81: main	534	2
wdiff0.5	712	3
gnuchess	6,810	31
grep2.5.3	8,579	39

Table 6.16: The results of applying the optimization 2: for each benchmark we present the number of the equality constraints given by the optimization 2, and the time of performing the optimization.

very frequently, almost everywhere, and a circle of the same data has a very low chance to form. We, therefore, focus on the second optimization. Table 6.16 summarizes the number of equalities yielded and the execution time of the optimization for each benchmark. As the table shows, the second optimization efficiently detects a lot of equivalences.

Last, the solver's performance is measured on the optimized version of the constraints and compared with that of the original version for each benchmark in Table 6.17. On average, the time performance is improved by 36% and the memory consumption is saved by 4%. Similar to what we observed in reaching definitions analysis, the performance improvement is positively proportional to the amount of equivalences identified.

Program	$\mathbf{T^{LV}_{opt2}}$(ms.)	$\mathbf{\Delta T^{LV}_{opt2}}$	$\mathbf{M^{LV}_{opt2}}$(MB)	$\mathbf{\Delta M^{LV}_{opt2}}$
phonebook1.0	55	30%	4	15%
make3.81: main	437	36%	4.1	0%
wdiff	62	34%	3.8	12%
gnuchess	2,063	34%	22.8	0%
grep2.5.3	2,953	37%	27.1	5%
Average	–	36%	–	4%

where:
$\Delta T' = 1 - T^{LV}_{opt2}/T^{LV}$
$\Delta M' = 1 - M^{LV}_{opt2}/M^{LV}$

Table 6.17: Performance evaluation: for each benchmark we present the the time and memory consumption after applying the *second* off-line optimization only. The average of improvement is summarized in the last line.

6.3 Concluding Remarks

In this section, we described three off-line optimizations and a strategy in order to generalize the use of off-line optimizations on our constraint solver. We demonstrate the use of the technique by implementing three program analyses: Andersen's pointer analysis, a reaching definitions analysis, and a live variable analysis. We evaluate the performance of the solver with two parameters: the number of equivalences detected and the kind of worklist strategy.

For Andersen's analysis, we observe that the equivalences yielded by the optimization improves the performance of the solver in general. The order of the optimization together with the kind of worklist strategy has considerably impact on the efficiency of constraint solving. When a high-efficiency combination is chosen, performance improves significantly.

For the two data flow analyses, we observe that the more equivalences are identified, the higher performance is achieved. And the experimental results show that a large amount of equivalences exist and can be detected by our off-line optimization algorithm. As the experiment demonstrates, these equivalences can be taken advantage of by our constraint solver to achieve much better performance than that not using the off-line technique.

As an automatic technique, the off-line optimization only conservatively specifies equivalences, i.e. the level of precision is always preserved. Principally the off-line technique is transparent to the analysis designer and is used by a solver internally. This is different from the lifting strategy (introduced in Chapter 3), which allows some loss in precision and the analysis designer can have a complete control on the use of equality constraints. The different considerations reflect our understanding of solver technology for program analyses: in order to gain a better control over the precision of an analysis (which is often quite important for the utility of program analysis), an analysis designer should be noticed whenever a mechanism may introduce any extra loss in precision compared to the original analysis; otherwise, an optimization technique is better to maintain the level of precision during the constraint solving. In Chapter 4 we have conducted a heuristic study on our case study and achieved a good control on precision. The insights provided by the heuristics enable analysis designers to be aware of any possible loss in precision by the use of unification.

Chapter 7

Conclusion

To conclude our work, in this chapter we shall first compare and contrast the contents presented in this dissertation with related work in Section 7.1. Our research work does not happen in a vacuum. Related work challenges and inspires our work. Many of the ideas in this dissertation were inspired by related work by other researchers.

Next we recapitulate the main thesis of this dissertation and make a review of research contributions of this dissertation in Section 7.2. This leads to a discussion of a further development of the thesis in Section 7.3.

7.1 Related Work

There are several aspects from which our work could be compared and contrasted with other researcher's. We, therefore, organize the comparison into the following three subsections.

7.1.1 Flat Term v.s. Structured Term

In the field of solver technology for solving program analysis problems, all kinds of solvers can be classified into two classes with respect to the types of terms used: one class of solvers considers flat terms (unstructured terms) and thereby can compute a complete solution, which is finite; another class of solvers uses a more general form of terms, complex terms (structured terms), and thus a least model could become infinite no matter a universe is finite or infinite. As a result, the second class of solvers solves constraints by calculating a finite representative (also known as normalization) for an infinite number of solutions. In this dissertation the solver we designed and implemented belongs to the first class, and so is the Succinct Solver, and other Datalog solvers. Although our solver uses inclusion constraints whileas the Succinct Solver and Datalog Solvers use some declarative logics as their specification language, all of them output a finite and complete solution. For the second class, a specification language could be classical set constraints [HJ90, CP97, PP97, Aik99], or be some restricted form of Horn clauses [NNS02]. We shall focus on the comparison between our solver and the solvers of the first class and shall also discuss some common interest by both of the two classes, e.g. Moore family property.

Datalog is a logic programming language originally introduced for deductive database [Ull90]. It is also found to be convenient for specifying program analyses [DRW96, Rep93, LS03]. The solvers of Datalog and their variations have been used for the implementation of many program analyses [NSN02, SSW94, WACL05, ZN08]. Concerning the expressiveness, our constraint language is a subset of Datalog and the ALFP logic further extends the classical Datalog by including explicit quantification, disjunctions in pre-conditions and conditional clauses. All of them guarantee the existence of a least model.

Upon set constraints, Heintze and Jaffar [HJ90] investigate definite set constraints and show that all satisfiable constraints in the class have a least model. Charatonik and Podelski [CP02] further showed that solving definite set constraints has DEXPTIME complexity. Although the set minus operation, which contains negative set expression, i.e. $\alpha \setminus c \equiv \alpha \cap \neg c$, makes our constraints fall out of the scope of definite set constraints, we show that the Moore family result still holds for the constraints of interest. Melski and Reps [MR97] proved a subclass of definite set constraints can be solved in cubic time by studying a simple data-flow reachability problem. While their constraints use only projection and terms, our basic inclusion constraint language includes the operations set minus and intersection on a flat universe and it is shown that the constraint solving has the same complexity. With unification, however, the complexity can be reduced to almost linear time. Due to unification having the property of dual-direction information flow, unifying inequivalent elements may cause a loss in precision.

The question is what is the tradeoff from doing this.

7.1.2 Parameterized Framework

The use of a general solver simplifies the implementation of program analyses. How to enhance the usability of program analyses, though, remains a challenge. To achieve high usability, analysis designers would hope to have an both efficient and precise analysis. However, as presented in previous chapters, there is often a trade-off between efficiency and precision. While too many false alarms make an analysis impractical, some level of imprecision should be allowed in order to ensure termination and even scalability. Thus a parameterized framework described in Chapter 3 and 4 gives an analysis designer flexibility in tuning a system to achieve a good balance between performance and precision. With set constraints, Fähndrich et al. [FA97] present a parameterized framework for a type system that allows expression of set-constraint-based analyses in varying levels of efficiency and precision using mixed-terms. A performance evaluation of using the framework to tune a system is, however, missing. We demonstrate in this dissertation the effect of tuning a reaching definitions analysis in terms of time, space performance, and the level of precision. By conducting a heuristic study on where and how imprecision may occur by the use of unification in the analysis specification, we show the analysis designer can gain a good control on the level of precision.

7.1.3 Optimization Techniques

7.1.3.1 Reordering Clauses

Besides tuning program analyses from a user's point of view, a large amount of effort has been devoted to improving solver performance. One approach is to optimize the order of constraints or clauses to be solved. The experimental results of [BNN02] demonstrate that reordering clauses can improve performance of the Succinct Solver considerably. A comparsion study in [Pil03] shows that the performance of the Succinct Solver is at most a small constant factor worse than XSB Prolog [SSW94, SSW$^+$02] but in optimum cases the Succinct Solver outperforms XSB Prolog significantly. This has the similar flavor of to choose proper worklist strategy in our iteration algorithm. From our experimental result in Subsection 6.2.1, we observed that the order of analysis variables on the worklist can significantly affect the efficiency of our solver. In order to generate efficiently solvable constraints for the Succinct Solver, however, one needs to understand how clauses are solved by the solver, and that restricts the use of the optimization heuristics. We use the technique of off-line optimizations in Chapter 5 that is fully automatic, cheap to execute, and more important minimizes the amount of knowledge about the solver required for its users.

7.1.3.2 Unification Techniques for Reducing the Problem Space

Another approach is to reduce the problem space by the use of unification. Since Robert Tarjan shows that unification can be performed in almost linear time, using unification to improve performance has been extensively studied in the literature of pointer analysis [Ste96, Das00, FFSA98, HL07a, PKH04], control flow analysis [HM97a], data flow analyses [LH02, ZN08], type inference system [Mil78], and program analysis in general [Hen92]. In the field of flow- and context-insensitive pointer analysis, the use of unification has been demonstrated to be crucial in cracking the scalability bottleneck. Steensgaard proposes a unification-based pointer analysis in [Ste96] to improve performance. Compared to Andersen's inclusion-based pointer analysis, it is much more efficient but also has much greater imprecision. Another version of unification-based pointer analysis, known as One-level Flow analysis [Das00] improves the precision of Steensgaard's analysis by restricting the use of unification. As a result, both the precision and performance of his analysis are between Steensgaard's analysis and Andersen's analysis. Sharpiro et al. [SH97b] further present a "tunable" algorithm so that its performance and precision range from those of Steensgaard's analysis and Andersen's analysis. All of these three versions of pointer analyses may reduce the level of precision when unification is applied because they do not guarantee that unification happens only within the elements of each equivalent class.

As demonstrated in [SH97a], the imprecision of a pointer analysis is inherited by a subsequent analysis and may have significant impact on the efficiency of the subsequent analysis. Therefore, a lot of effort has be spent on improving the scalability of Andersen's pointer analysis. By the nature of the problem of Andersen's pointer analysis, it is a dynamic transitive closure problem. Such a problem has been showed to be in the class of two way nondeterministic push down automata (2NPDA) and is 2NPDA-hard [HM97b]. Thus it is considered inherently cubic as no sub-cubic algorithm for any 2NPDA problem is known so far.

However, unification can be used to reduce a problem space, as well as simplify the calculation significantly. The key issue is how to identify equivalence classes in an efficient manner. Depending on the time of performing the detection, the optimization algorithms are classified as online technique and offline technique, i.e. during the constraint solving and before the constraint solving. Using the phrase online and offline to classify equivalence detection algorithms reflects the fact that online algorithms need to be performed many times whileas offline algorithms run only once. Therefore a linear time online equivalence-detection algorithm itself does not guarantee high efficiency of a program analysis. In order to avoid a too high overhead (compared to the benefit acquired), some mechanisms are needed to artificially restrict the times of executing the algorithm.

Fähndrich et al. [FFSA98] use set constraints to specify the analysis and represent the constraints using a graph. Cycles in the graph are detected by a depth-first search of the graph upon each edge insertion, and all analysis variables of a detected cycle are collapsed. They demonstrate that online cycle detection is very important for scalability. But the search of cycles is artificially restricted because the overhead of the search may be too high to pay off the efficiency acquired.

Later Pearce et al. improves the technique by introducing two algorithms of online cycle detection for Andersen's pointer analysis [PKH04, PKH07]. The first is a more efficient algorithm than that used by Fähndrich et al. and the second detects cycles periodically instead of at every edge insertion in order to minimize the effort of the detection. Heintze and Tardieu [HT01] present a field-based pointer analysis that embeds cycle detection technique. Their analysis can analyze a C program with 1.3M LOC in less than a second. But a field-based analysis is not sound for C.

Recently Hardekopf and Lin [HL07a] introduce an online cycle detection algorithm that they call Lazy Cycle Detection (LCD). Observing that equivalent analysis variables must have the same data, they invoke cycle detections only when the source and destination share the same points-to sets. Together with HCD, they analyze C programs up to 2.17 M LOC. The internal data structure of our constraint solver allows us to smoothly integrate an on-line technique, such as LCD, and thereby apply the optimization to more analyses.

Off-line optimization were used by Rountev et. al. [RC00] and then by Hardekopf and Lin [HL07a, HL07b] to reduce the cost of pointer analysis. These techniques have been reported to improve the performance of the analysis considerably. The first optimization algorithm described in Chapter 6 is directly motivated by the technique HCD presented in [HL07a]. By taking an inclusion constraint approach we record the off-line optimization results with explicit unification constraints: all off-line analyses are conducted on the constraints and can potentially be reused by more analyses. The reason that off-line optimizations are so attractive is because they are very efficient and can still identify a large amount of equivalent classes.

From a constraint solver's point of view, techniques using unification can naturally be classified by another dimension: analysis variable domain and the tuple-space domain. Recall that the essence of using unification is to reduce a problem space. Beside collapsing analysis variables, collapsing elements in the tuple-space can also reduce a problem space and thus improve efficiency of a solver. Steensgaard's analysis is a typical analysis that does unification over both of the two domains. So are its variants, such as Das' One-level Flow analysis, and the algorithms described in [SH97b].

The techniques using the online cycle detection [FFSA98, PKH04, PKH07, HT01, HL07a], and using the offline processing [RC00, HL07a] fall into the class that only do unification within the analysis variable domain. The constraint language studied in this dissertation also considers only doing unification over analysis variables.

Liang and Harroid make use of location equivalence to optimize dataflow analyses [LH02]. Hardekopf and Lin introduce an off-line algorithm to identify location equivalence for Andersen's pointer analysis. The so-called location equivalence has the same flavor as doing unification over the tuple-space domain.

Classifying techniques with the two domains makes it clear that which domain is actually reduced by the use of unification. It also demonstrates the two basic places where analysis designers and solver designers may consider to apply unification technique. Finally the phrase, unification over analysis variables and over tuple-space, seems general enough to be used in that it can cover all kinds of names for equivalences, such as pointer-equivalent variables, location equivalence, equivalent program variables, etc.

7.1.3.3 Efficient Data Representations

The third approach considers to improve solver performance by using data structures that can be efficiently operated. In the work presented in this dissertation, for example, we encode each tuple as a bit and each constant is represented as a bit vector. We have observed that a solver using bit vectors is much more efficient than that using a data structure, such as lists, binary trees.

Recently, researchers try to use Binary Decision Diagrams (BDDs) [Bry86] to implement program analyses. BDDs were traditionally applied for hardware verification and model checking. Berndl et al. [BLQ$^+$03] use it to represent both the constraint graph and points-to solution for a field-sensitive inclusion-based pointer analysis. Later Zhu et al. [Zhu02, ZC04], and Whaley et al. [WL04] showed that BDDs could be used to solve context-sensitive pointer analysis more efficiently than other known efficient algorithms in literature. Whaley et al. also introduced a Datalog solver using BDDs for implementing program analyses [WACL05]. The time of a BDD operation is not proportional to the number of tuples in a relation, but to the size of the data structure. This leads to a fast execution time for their Datalog Solver. The purpose of the techniques described in this dissertation is to reduce a problem space with the use of unification. As long as the algorithm and the data structure of a solver can take advantage of this result, we believe these techniques can be used on these solvers and improve their performance.

7.2 Review of Research Contributions

In this dissertation, we have investigated the constraint solver technology with unification for program analysis problems. Our main thesis was that

> A constraint solver with well-designed techniques using unification can *significantly* improve the usability of a program analysis.

In order to support this thesis, we have conducted a series of theoretical and experimental studies in the previous chapters of this dissertation. In the remainder of this section, we summarize the main research contributions:

- We have specified an inclusion constraint language in Chapter 3 and then extended the language in Chapter 5. The well-designed constraint language, especially the use of explicit (dynamic) equality constraints, and the conditional constraints, enables a constraint solver to take good advantage of unification for improving its efficiency.
- We have presented a parameterized framework for tuning the constraint program. With the framework, analysis designers can try testing various levels of performance and precision trade-off, and accordingly achieve a satisfiable balance between these two factors.
- We have demonstrated the use of the parameterized framework by a heuristic study with an intraprocedural reaching definitions analysis for a C-like imperative language. The study shows that a careful study of the conditions where imprecision may or may not be incurred pays off in gaining the expected level of precision.

- We have designed constraint-solving algorithms for constraint programs specified in the basic and extended inclusion constraint language respectively. The important properties of the algorithms, such as termination and worst-cast asymptotic complexity, have been studied. Based on the presentation of the algorithm design, we have explained further how unification enables a solver more efficient.

- We have used the off-line optimization as a general strategy for improving the performance of a constraint solver. Three off-line optimizations have been described: all of them are linear time complexity and preserve precision. By detecting the subset of constraints, the optimizations can be chosen automatically and the insights of existing off-line optimizations could be easily shared among different analyses.

- We have demonstrated the effect of using unification-based technique on performance by many experimental studies on several program analyses, including intraprocedural reaching definitions analyses for a simple imperative language and for C language individually, Andersen's pointer analysis for C language, and a live variable analysis for C language. A comparison study on the reaching definitions analysis conducted between our constraint solver and the Succinct Solver has shown that using unification may lower the asymptotic complexity of constraint-solving even down to almost linear time for some benchmarks. In general, we have observed that with the techniques described in this dissertation the performance of program analyses could be improved significantly even without sacrificing any precision.

7.3 Future Work

We firmly believe that automatically generating high-efficiency implementation for program analysis is a very important direction of the research on static program analysis. Solver technology is the core engine of the approach. The use of general constraint solvers significantly simplifies the implementation of program analysis, and provides a platform to test and compare all kinds of ideas for automatically generating high-efficiency implementation, and thereby encourages the generalization of the insights of various of optimizations motivated by different analyses. However, there are still many improvements that could further enhance our constraint solver to get better results.

7.3.1 Unification over the Tuple Space

As we explained in the section of related work, our constraint solver considers only the unification over analysis variables. To provide a constraint solver that can make fully use of unification techniques, our constraint language should introduce new constructs to formulate the equivalence relation between tuples. This extension should allow us to implement more unification-based analyses and introduce more off-line optimizations using the new equality constructs.

7.3.2 Flow- and Context-sensitive Pointer Analysis

We have observed that a large amount of equivalences exist in flow sensitive analyses, such as reaching definitions analysis and live variable analysis. We postulate, therefore, unification technique can be applied to flow- or context-sensitive pointer analysis or even both to improve the scalability of the analyses.

7.3.3 Other Data Representations

We have used bit-vector data structure to represent constants in the algorithm of our constraint solver. It works well for the analyses we have studied. But to improve the expressiveness of our constraint language, we may need to consider more possible data representations, such as BDDs, balanced binary trees, etc. For instance, when cartesian product or projection is concerned, new tuples may yield during the constraint solving and therefore using bit-vector may become not flexible and convenient, and thus a binary tree may be a better choice under such a situation. As we discussed in the section of related work, BDDs have been used for implementing pointer analysis. But they are not successful as well in data flow analyses. As a general constraint solver, our solver should support more data representations and thereby support a further evaluation on what and how each of these data representations is good at in constraint solving.

Appendix A

Constructive Definition of Designated Greatest Lower Bound

Definition A.1 (Operator $\hat{\sqcap}_T$) For some set $I = \{1, 2, \cdots, n\}$ and a family $(\psi_1^i, \psi_2^i)_{(i \in I)} \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$, let $(\psi_1^{\hat{\sqcap}}, \psi_2^{\hat{\sqcap}}) = \hat{\sqcap}_{T_{i \in I}}(\psi_1^i, \psi_2^i)$ which is given by:

$$\forall x \in \mathbf{AVar} : \psi_1^{\hat{\sqcap}}(x) = \alpha_{1_{(\alpha_2, \cdots, \alpha_n)}}$$

where $\forall i \in I : \alpha_i = \psi_1^i(x)$ and

$$\psi_2^{\hat{\sqcap}}(\beta_{1_{(\beta_2, \cdots, \beta_n)}}) = \cap_i \psi_2^i(\beta_i)$$

where $\beta_{1_{(\beta_2, \cdots, \beta_n)}} \in dom(\psi_1^{\hat{\sqcap}})$.

We then show it is indeed a greatest lower bound by Lemma A.2 and A.3 below.

Lemma A.2 *For some set $I = \{1, 2, \cdots, n\}$ and a family $(\psi_1^i, \psi_2^i)_{(i \in I)} \in \mathbf{Env_T} \times \widehat{\mathbf{Env_{TB}}}$, let $(\psi_1^{\hat{\sqcap}}, \psi_2^{\hat{\sqcap}}) = \hat{\sqcap}_{T_{i \in I}}(\psi_1^i, \psi_2^i)$, then for all $x, y, z \in \mathbf{AVar}$ and $i \in I$:*

$$\hat{\psi}_1^{\hat{\sqcap}}(x) = \hat{\psi}_1^{\hat{\sqcap}}(y) \Leftrightarrow \psi_1^i(x) = \psi_1^i(y) \wedge \tag{A.1}$$

$$\psi_2^{\hat{\sqcap}}(\psi_1^{\hat{\sqcap}}(z)) = \cap_i \hat{\psi}_2^i(\psi_1^i(z)) \tag{A.2}$$

Proof. It is straightforward to prove (4) and (5) according to Definition A.1. The proof relies on the fact that for any analysis variable x: $\hat{\psi}_1^{\hat{\sqcap}}(x) = \hat{\psi}_1^1(x)_{\hat{\psi}_1^2(x),\cdots,\hat{\psi}_1^n(x)}$. □

Lemma A.3 *Let* $E = \{(\hat{\psi}_1^i, \hat{\psi}_2^i) | \, i \in I \wedge (\hat{\psi}_1^i, \hat{\psi}_2^i) \in \mathbf{Env_T} \times \mathbf{Env_{TB}}\}$ *for some set* I*, and* $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}}) = \hat{\sqcap}_T E$*. Then* $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}})$ *is a greatest lower bound of the set* E*.*

Proof. It is straightforward to show $(\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}})$ is a lower bound by Lemmata 3.10 and A.2. We prove it is a greatest one. For any lower bound of E, e.g. $(\hat{\psi}_1^\ell, \hat{\psi}_2^\ell)$, we have for all $i \in I$ and $x, y, z \in \mathbf{AVar}$

$$\begin{array}{l} \hat{\psi}_1^\ell(x) = \hat{\psi}_1^\ell(y) \Rightarrow \hat{\psi}_1^i(x) = \hat{\psi}_1^i(y) \wedge \\ \hat{\psi}_1^\ell(\hat{\psi}_1^\ell(z)) \subseteq \hat{\psi}_1^i(\hat{\psi}_1^i(z)) \end{array}$$

from Lemma 3.10 and therefore

$$\begin{array}{l} \hat{\psi}_1^\ell(x) = \hat{\psi}_1^\ell(y) \Rightarrow \hat{\psi}_1^{\hat{\sqcap}}(x) = \hat{\psi}_1^{\hat{\sqcap}}(y) \\ \hat{\psi}_1^\ell(\hat{\psi}_1^\ell(z)) \subseteq \hat{\psi}_1^{\hat{\sqcap}}(\hat{\psi}_1^{\hat{\sqcap}}(z)) \end{array}$$

by Lemma A.2. Now the preorder definition of $\preceq$ allows us to conclude that $(\hat{\psi}_1^\ell, \hat{\psi}_2^\ell) \preceq (\hat{\psi}_1^{\hat{\sqcap}}, \hat{\psi}_2^{\hat{\sqcap}})$ for any lower bound $(\hat{\psi}_1^\ell, \hat{\psi}_2^\ell)$ of E. □

Appendix B

Scalable Programs

We organize the scalable programs into three groups.

1. Three series of scalable programs using while loop.

```
Wh$_{(n,1)}$ :   while $x_0 < 2$ do
                     $(x_1 := x_2; \cdots x_{n-1} := x_n; x_n := 1)$

Wh$_{(1,n)}$ :   while $x_1 < 2$ do
                     while $x_2 < 2$ do
                             $\vdots$
                         while $x_n < 2$ do $x_0 := 1$

Wh$_{(n,n)}$ :   while $e_1$ do
                     while $e_2$ do
                             $\vdots$
                         while $e_n$ do
                             $(x_1 := x_2; \cdots x_{n-1} := x_n; x_n := 1)$
```

2. Three series of scalable programs using if-branch.

```
If_{(n,1)} :   if $x_1 < 0$ then skip
                 else if $x_2 < 0$ then skip
                        else
                           ⋮
                           if $x_n < 0$ then skip
                             else $x_0 := 1$
```

```
If_{(n,n)} :   if $x_1 < 0$ then skip
                 else if $x_2 < 0$ then skip
                        else
                           ⋮
                           if $x_n < 0$ then skip
                             else $(x_1 := x_2; \cdots x_{n-1} := x_n; x_n := 1)$
```

```
If_{(1,n)} :   if $x_0 < 0$ then skip
                 else $(x_1 := x_2; \cdots x_{n-1} := x_n; x_n := 1)$
```

3. Two series of scalable programs use both if-branch and while loop.

```
If-wh_{(n,1,1)} :   if $x_1 < 0$ then skip
                      else if $x_2 < 0$ then skip
                             else
                                ⋮
                                if $x_n < 0$ then skip
                                  else (while $x_n > 2$ do $x_n := 1$)
```

```
Wh-if_{(n,1,1)} :   while $x_1 < 2$ do
                      while $x_2 < 2$ do
                                ⋮
                        while $x_n < 2$ do
                          if $x = 0$ then skip
                            else $x := 1$
```

Appendix C

Experimental Results of Scalable Programs: Asymptotic Complexity of Time Performance

We summarize as the following diagrams the experimental results for the six series of the scalable programs that was not included in the main text, i.e. $\mathrm{Wh}_{(1,n)}$, $\mathrm{Wh}_{(n,n)}$, $\mathrm{If}_{(n,n)}$, $\mathrm{If}_{(1,n)}$, $\text{If-wh}_{(n,1,1)}$, and $\text{Wh-if}_{(n,1,1)}$.

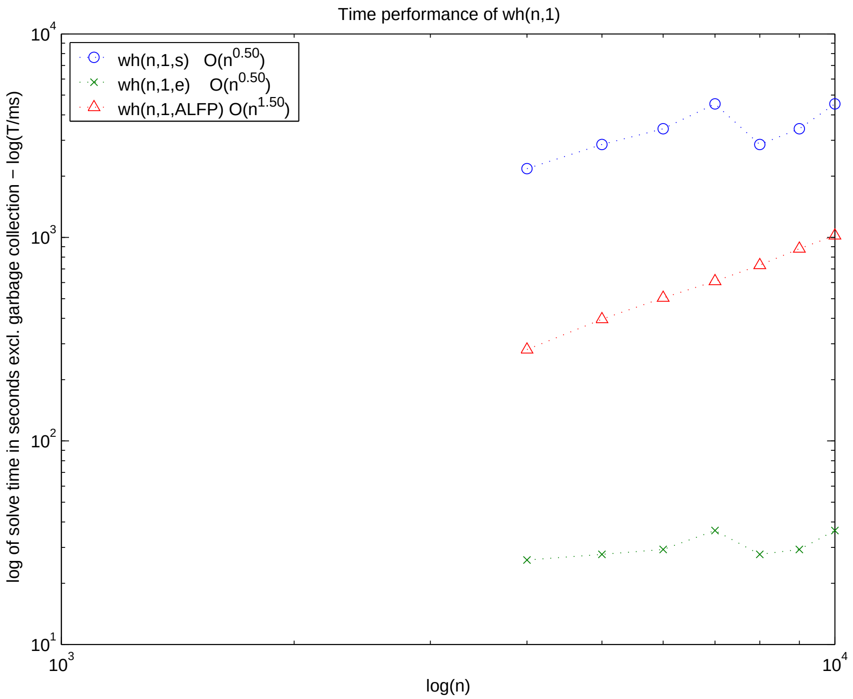


Figure C.1: Asymptotic Complexity of Time Performance: $\mathrm{Wh}_{(n,1)}$.

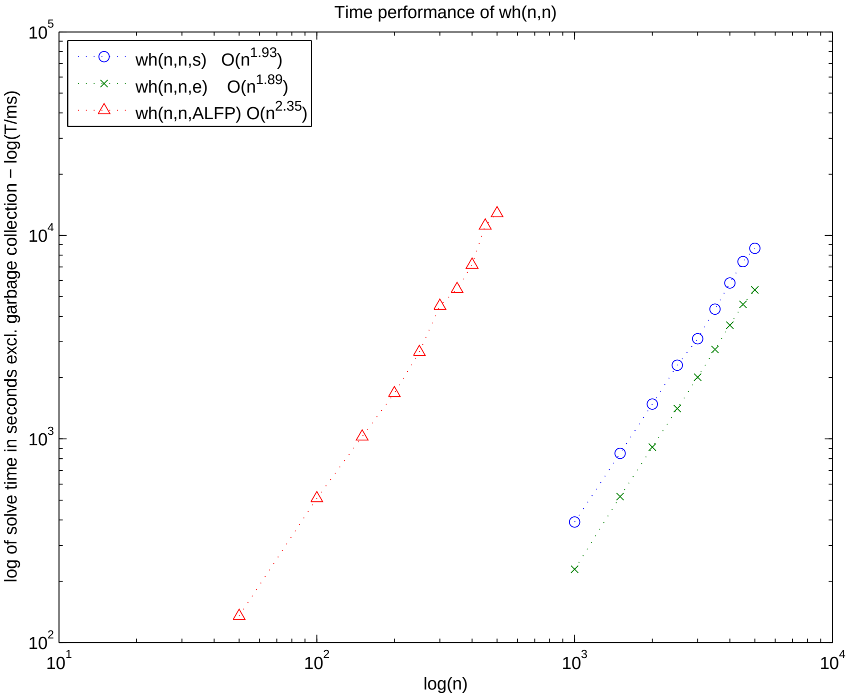


Figure C.2: Asymptotic Complexity of Time Performance: $\mathrm{Wh}_{(n,n)}$.

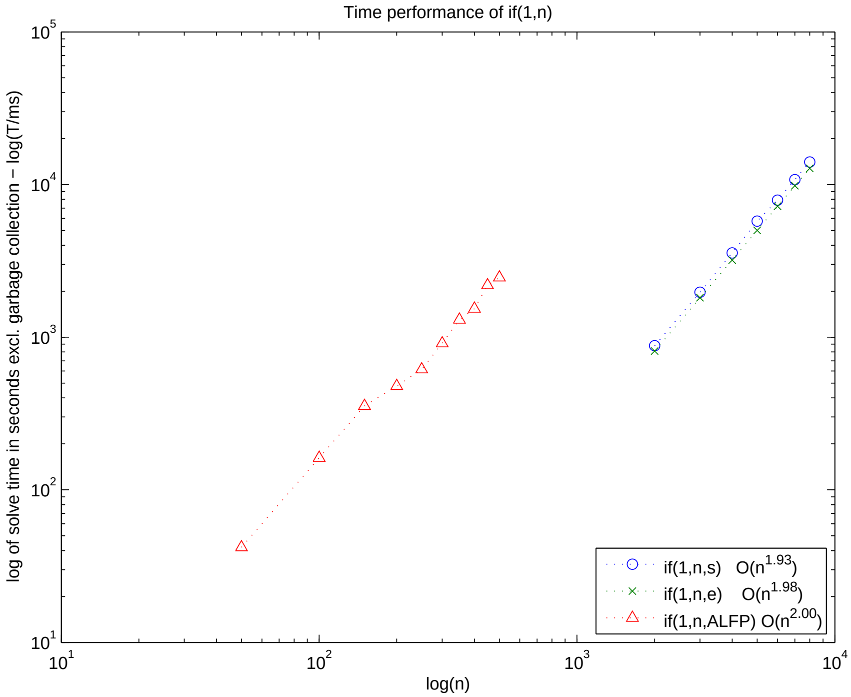


Figure C.3: Asymptotic Complexity of Time Performance: $\text{If}_{(1,n)}$.

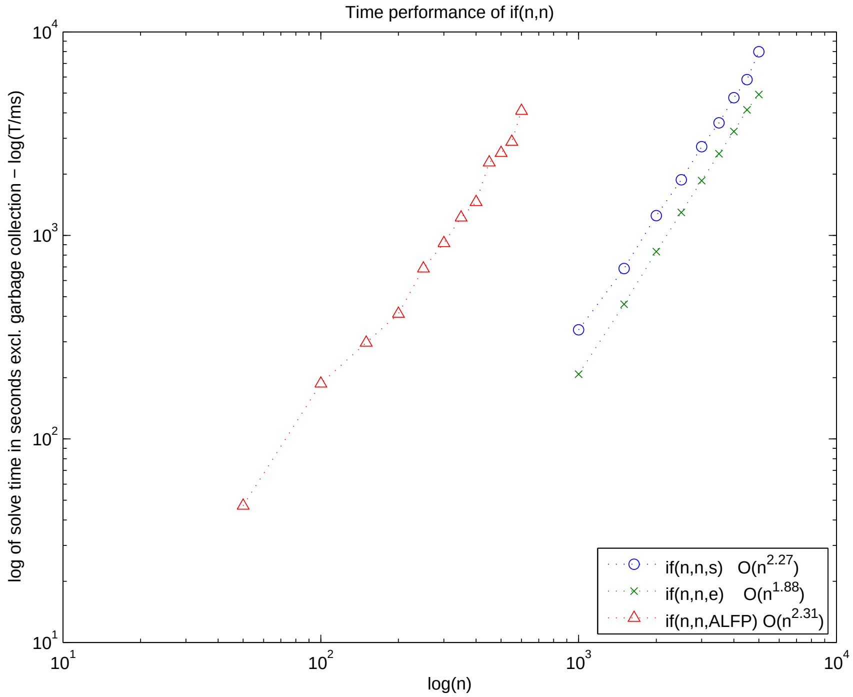


Figure C.4: Asymptotic Complexity of Time Performance: $\text{If}_{(n,n)}$.

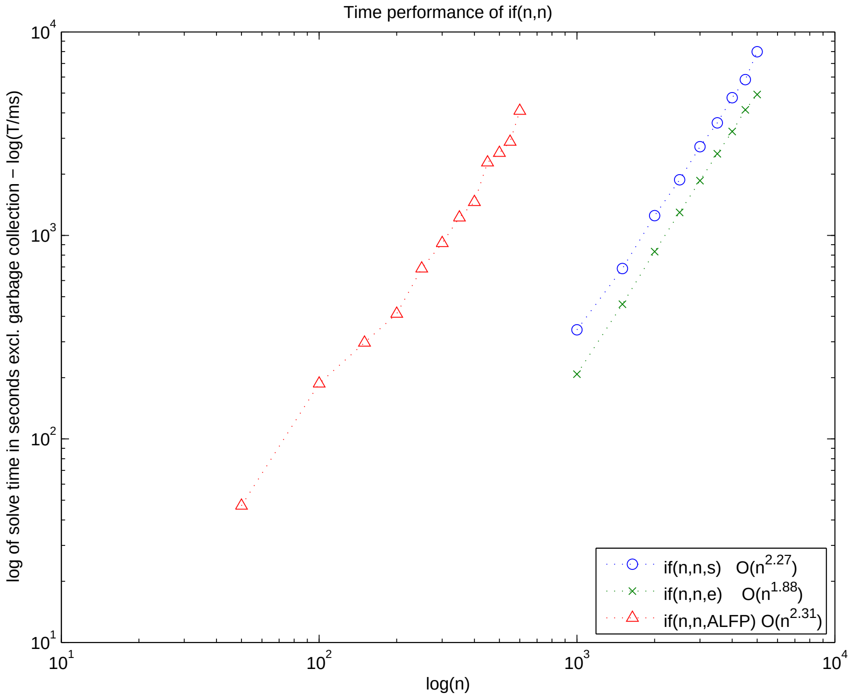


Figure C.5: Asymptotic Complexity of Time Performance: $\text{If}_{(n,n)}$.

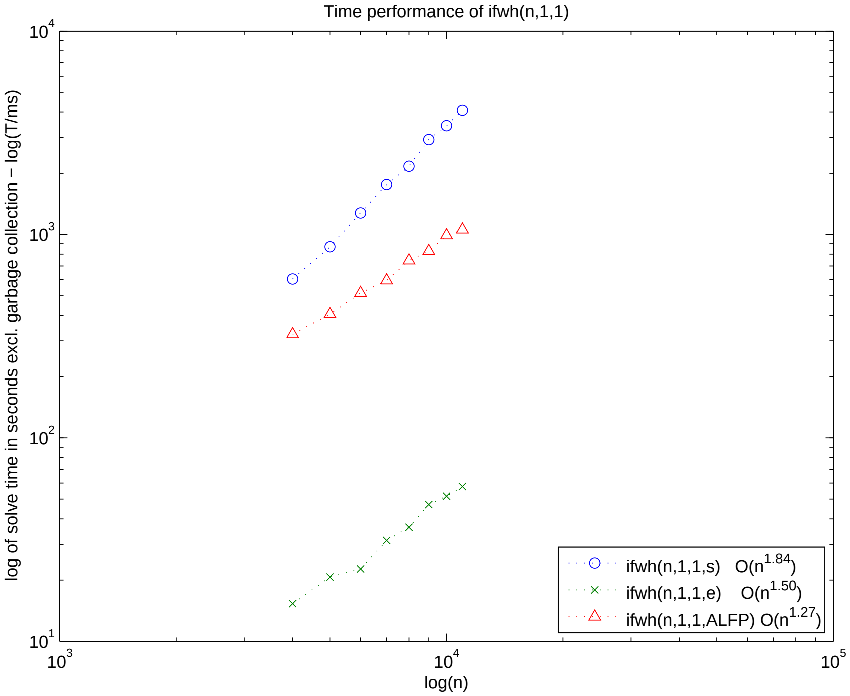


Figure C.6: Asymptotic Complexity of Time Performance: If-wh$_{(n,1,1)}$.

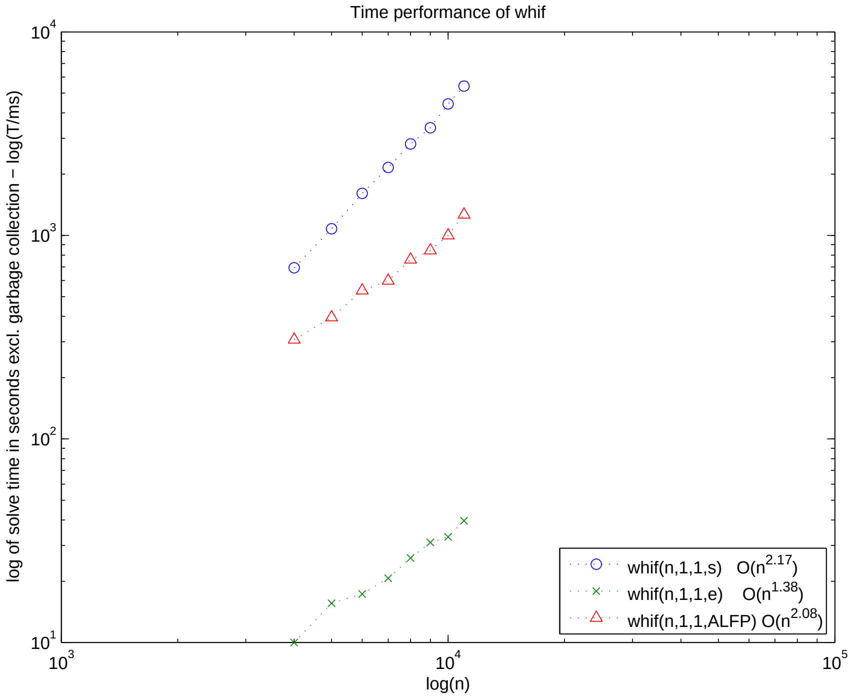


Figure C.7: Asymptotic Complexity of Time Performance: Wh-if$_{(n,1,1)}$.

Bibliography

[Aik99] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2):79–111, 1999.

[And94] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[BBD$^+$05] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[BBN$^+$03] Lorenzo Bettini, Viviana Bono, Rocco De Nicola, Gian Luigi Ferrari, Daniele Gorla, Michele Loreti, Eugenio Moggi, Rosario Pugliese, Emilio Tuosto, and Betti Venneri. The klaim project: Theory and practice. In Corrado Priami, editor, *Global Computing*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003.

[BCC01] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In Naoki Kobayashi and Benjamin C. Pierce, editors, *TACS*, volume 2215 of *Lecture Notes in Computer Science*, pages 38–63. Springer, 2001.

[BCCH94] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In Keshav Pingali, Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *LCPC*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer, 1994.

[BLQ$^+$03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114. ACM, 2003.

[BNN02] M. Buchholtz, H. Riis Nielson, and F. Nielson. Experiments with succinct solvers. Technical report, Informatics and Mathematical Modelling, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, Denmark, 2002.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[CBC93] Jong-Deok Choi, Michael G. Burke, and Paul R. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245. ACM, 1993.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CG00] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[CH92] Baudouin Le Charlier and Pascal Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, 1992.

[Cha03] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points - to analysis. In *POPL*, pages 115–125. ACM, 2003.

[CP97] Witold Charatonik and Andreas Podelski. Set constraints with intersection. In *LICS*, pages 362–372, 1997.

[CP02] Witold Charatonik and Andreas Podelski. Set constraints with intersection. *Inf. Comput.*, 179(2):213–229, 2002.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI*, pages 35–46. ACM, 2000.

[DP02] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, April 2002.

[DRW96] Steven Dawson, C. R. Ramakrishnan, and David Scott Warren. Practical program analysis using general purpose logic programming systems - a case study. In *PLDI*, pages 117–126. ACM, 1996.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256. ACM, 1994.

[FA97] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In Hentenryck [Hen97], pages 114–126.

[FFSA98] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI*, pages 85–96. ACM, 1998.

[FG95] Riccardo Focardi and Roberto Gorrieri. A taxonomy of security properties for process algebras. *Journal of Computer Security*, 3(1):5–34, 1995.

[FS98a] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. *Nord. J. Comput.*, 5(4):304–329, 1998.

[FS98b] Christian Fecht and Helmut Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In Chris Hankin, editor, *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 1998.

[FS99] Christian Fecht and Helmut Seidl. A faster solver for general systems of equations. *Sci. Comput. Program.*, 35(2):137–161, 1999.

[HBCC99] Michael Hind, Michael G. Burke, Paul R. Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[Hen92] Fritz Henglein. Global tagging optimization by type inference. In *LISP and Functional Programming*, pages 205–215, 1992.

[Hen97] Pascal Van Hentenryck, editor. *Static Analysis, 4th International Symposium, SAS '97, Paris, France, September 8-10, 1997, Proceedings*, volume 1302 of *Lecture Notes in Computer Science*. Springer, 1997.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM, 2001.

[HJ90] Nevin Heintze and Joxan Jaffar. A decision procedure for a class of set constraints (extended abstract). In *LICS*, pages 42–51. IEEE Computer Society, 1990.

[HL07a] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 290–299. ACM, 2007.

[HL07b] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 265–280. Springer, 2007.

[HM97a] Nevin Heintze and David A. McAllester. Linear-time subtransitive control flow analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–272, 1997.

[HM97b] Nevin Heintze and David A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS*, pages 342–351, 1997.

[Hor97] Susan Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.

[HT01] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI*, pages 254–263. ACM, 2001.

[ISO00] ISO/IEC. *International Standard ISO/IEC 9899:1999, Programming Languages . C.* 2000.

[KW94] Atsushi Kanamori and Daniel Weise. Worklist management strategies. Technical Report MSR-TR-94-12, Microsoft Research, 1994.

[LH99] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 1999.

[LH02] Donglin Liang and Mary Jean Harrold. Equivalence analysis and its application in improving the efficiency of program slicing. *ACM Trans. Softw. Eng. Methodol.*, 11(3):347–383, 2002.

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*, pages 93–103. ACM, 1991.

[LS03] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *PPDP*, pages 172–183. ACM, 2003.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[MR97] David Melski and Thomas W. Reps. Interconveritibility of set constraints and context-free language reachability. In *PEPM*, pages 74–89, 1997.

[NFP98] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.

[NFP00] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Programming access control: The klaim experience. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2000.

[NKmWH04] Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In Roberto Giacobazzi, editor, *SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2004.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2002.

[NN97] Hanne Riis Nielson and Flemming Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *POPL*, pages 332–345. ACM, 1997.

[NN98] Hanne Riis Nielson and Flemming Nielson. Flow logics for constraint based analysis. In Kai Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1998.

[NN02] Hanne Riis Nielson and Flemming Nielson. Flow logic: A multiparadigmatic approach to static analysis. In Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 223–244. Springer, 2002.

[NNB02] Hanne Riis Nielson, Flemming Nielson, and Mikael Buchholtz. Security for mobility. In Riccardo Focardi and Roberto Gorrieri, editors, *FOSAD*, volume 2946 of *Lecture Notes in Computer Science*, pages 207–265. Springer, 2002.

[NNH99] F. Nielson, H. Riis Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NNS02] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Normalizable horn clauses, strongly recognizable relations, and spi. In Manuel V. Hermenegildo and Germán Puebla, editors, *SAS*, volume 2477 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2002.

[NSN02] Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A succinct solver for ALFP. *Nord. J. Comput.*, 9(4):335–372, 2002.

[PC04] William Pugh and Craig Chambers, editors. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. ACM, 2004.

[Pil03] Henrik Pilegaard. A feasibility study: The Succinct Solver v2.0, XSB prolog v2.6, and flow-logic based program analysis for carmel. Technical Report SECSAFE-IMM-008-1.0, Technical University of Denmark, 2003.

[PKH04] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):311–337, 2004.

[PKH07] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.

[PP97] Leszek Pacholski and Andreas Podelski. Set constraints: A pearl in research on constraints. In Gert Smolka, editor, *CP*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–562. Springer, 1997.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[RC00] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI*, pages 47–56. ACM, 2000.

[Rep93] Thomas W. Reps. Demand interprocedural program analysis using logic databases. In *Workshop on Programming with Logic Databases, ILPS*, pages 163–196, 1993.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematics Society*, 74(2):358–366, 1953.

[SH97a] Marc Shapiro and Susan Horwitz. The effects of the presision of pointer analysis. In Hentenryck [Hen97], pages 16–34.

[SH97b] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *POPL*, pages 1–14. ACM, 1997.

[SSW94] Konstantinos F. Sagonas, Terrance Swift, and David Scott Warren. XSB as an efficient deductive database engine. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD Conference*, pages 442–453. ACM Press, 1994.

[SSW$^+$02] Konstantinos F. Sagonas, Terrance Swift, David Scott Warren, Juliana Freire, Prasad Rao, Boaqiu Cui, and Ernie Johnson. The XSB system - version 2.5 - programers manual. Technical report, 2002.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL*, pages 32–41. ACM, 1996.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms*, volume CMBS44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[TGL06] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In Alan Mycroft and Andreas Zeller, editors, *CC*, volume 3923 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2006.

[Ull90] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies.* W. H. Freeman & Co., New York, NY, USA, 1990.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI*, pages 1–12. ACM, 1995.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In Pugh and Chambers [PC04], pages 131–144.

[ZAN08] Ye Zhang, Torben Amtoft, and Flemming Nielson. From generic to specific: Off-line optimization for general constraint solver. In *Proceedings of the ACM SIGPLAN 7th International Conference on Generative Programming and Component Engineering*. ACM Press, 2008.

[ZC04] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In Pugh and Chambers [PC04], pages 145–157.

[Zhu02] Jianwen Zhu. Symbolic pointer analysis. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *ICCAD*, pages 150–157. ACM, 2002.

[ZN06] Ye Zhang and Hanne Riis Nielson. Analyzing security protocols in hierarchical networks. In Susanne Graf and Wenhui Zhang, editors, *ATVA*, volume 4218 of *Lecture Notes in Computer Science*, pages 430–445. Springer, 2006.

[ZN08] Ye Zhang and Flemming Nielson. A scalable inclusion constraint solver using unification. In Andy King, editor, *LOPSTR'07*, volume 4915 of *LNCS*, pages 121–136. Springer-verlag, 2008.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *SIGSOFT FSE*, pages 81–92, 1996.