Program Analysis based on Weighted Pushdown Model Checking

by

Li Xin

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Mizuhito Ogawa

School of Information Science
Japan Advanced Institute of Science and Technology

March, 2007

Abstract

Program analysis can be regarded as model checking of abstract interpretation. This view enables a systematic way of program analysis with soundness guarantees from abstract interpretation and model checking. Following to this methodology, this thesis [1] is dedicated to exploring interprocedural program analysis based on weighted pushdown model checking. Weighted pushdown model checking provides a general framework for program analysis with combining both CFL-reachability and an algebraic description of dataflow.

Our focus is an interprocedural extension of Bandera-like approach, i.e. program analysis design and prototype implementation with a model checker as the analysis engine. Our first target is a points-to analysis of an object-oriented programming language, specifically Java.

A points-to analysis is the basis of interprocedural program analyses and is not a simple matter. For instance, a dynamically dispatched method depends on the type of a receiver object. Thus, the call graph construction of method invocations and points-to analysis are mutually dependent. Another difficulty comes from possibly unbounded nesting of field access and array structures, and the aliasing need to be cast and proper abstraction must be chosen for various sources of infinity.

This thesis investigates *context/field/flow-sensitive* points-to analyses for Java, based on weighted pushdown model checking. They are characterized by the following orthogonal dimensions:

- An *exploded supergraph* vs an *interprocedural control flow graph*
  An exploded supergraph and interprocedural control flow graph are typical backborn models for pushdown model checking. The former is an efficient point-wise graph representation of dataflow, and the analysis based on it is more likely to scale. The latter is more conventional for program analysis, and surpasses the former in the automatic removal of invalid call pathes.

- An *on-the-fly* vs an *ahead-of-time* call graph construction
  A call graph can be constructed in either on-the-fly or ahead-of-time manner. The former is regarded as a least fixed point computation, i.e., starting with the empty call graph, a call edge is added if points-to analysis detects its possibility. The latter is regarded as a greatest fixed point computation, i.e., starting with a redundant call graph (by collecting syntactically possible all call edges), a call edge is eliminated if points-to analysis refutes it.

- Parameterized *flow-sensitivity* and *field-sensitivity*

[1]This research is conducted as a program for the "Fostering Talent in Emergent Research Fields" in Special Coordination Funds for promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology.

Furthermore, the relatively unexplored problem of *parametrization* is explored. Parameterized *flow-sensitivity* is naturally obtained in our algorithms by either model reduction in the analysis based on the exploded supergraph; or by simplifying the weight space design based on the interprocedural CFG.

A call graph construction, which is essentially a points-to analysis, is the basis of interprocedural program analyses. An example shown in this thesis is an interprocedural irrelevant code under PER (partial equivalence relation) based abstraction. Interaction among procedures are captured by variable dependency based on the exploded supergraph model design. An irrelevant code is more *semantical* than a dead code, which is syntactically defined by use/def relation and is essentially intraprocedural.

Most of analyses design above are impletemented within a prototype framework. Similar to Bandera-like approach, our analysis works on Jimple, a three-address intermediate representation for Java with fewer language constructs. Soot is used as a frontend preprocessing from Java to Jimple, and the *weighted PDS* library as a back-end model checking engine. The strategy of using existing tools enable us a relatively fast prototype development.

Key Words: Program Analysis, Formal Verification, Pushdown Model Checking, Weighted Pushdown Model Checking, Abstract Interpretation, Points-to analysis, Java

Acknowledgments

Contents

Chapter 1

Introduction

The increasing complexity of software and hardware systems nowadays makes their validation more challenging. Current practiced methods for system validation are mostly based on simulation and testing. The fundamental problem for these methods is that they cannot cover all possible scenarios of system runs. Thus, they cannot find subtle errors of the system. A promising alternative to system validation is *formal verification*, of which popular approaches are *theorem proving*, *model checking*, etc. Theorem proving is a deductive approach capable of verifying infinite state space. But the use of it usually need expertise and heuristics. In contrast, model checking [1], the so-called "*push-button technique*", is a fully automatic and algorithmic technique for verification on temporal safety of reactive and concurrent systems. In particular, if model checking once fails, counterexamples are provided as evidences for the failure and clues for fixing the problem.

There are successful stories of applying finite model checking to verify hardware design (and some telecommunication protocols) that essentially has a finite state space. Automatic software validation is not easy because of software's infinity, i.e. infinite program structures, infinite data domains, concurrency, etc. Automatic software validation demands efforts from program analysis/abstraction to model checking techniques on infinite state space. Popular model checkers, such as Spin [6], NuSMV/SMV [1], are model checkers on finite state space. Recently, some practical model checking algorithms on the (weighted) pushdown system, the finite-state system with an unbounded stack memory, have been developed [7, 8] and implemented as tools, such as MOPED [2], Weighted PDS library [3], WPDS++ library [4], etc.

B. Steffen [4] and D. A. Schmidt [5] observed that data flow analysis can be regraded as model checking of abstract interpretation. This view enables separation of the design (abstraction) and implementation (back-end model checking) of program analysis. For prototype implementations, an universal analysis engine is advantageous to clarify whether an analysis design is correct wrt the language semantics based on abstract interpretation. Following to this methodology, this thesis investigates program analysis based on weighted pushdown model checking.

[1]http://www.cs.cmu.edu/ modelcheck/code.htm

[2]http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/

[3]http://www.fmi.uni-stuttgart.de/szs/tools/wpds/

[4]http://www.cs.wisc.edu/wpis/wpds++/

1.1 Motivations

1.1.1 Model Checking

It is well known that the correctness of programs (total correctness [9]) is ascribed to termination and partial correctness [10, 11]. theory tells that the termination problem is undecidable. A. Pnueli first introduces *temporal logic* to computer science for verifying concurrent and reactive systems [12], which usually do not terminate. In the 1980s, *model checking*, an automatic verification technique on finite-state concurrent systems, is proposed by E.M. Clarke and E.A. Emerson [13] (independently by J.P. Quielle and J. Sifakis [14]). By model checking, we usually mean whether a model (historically, Kripke structure) satisfies some property represented in a temporal logic formula.

Model checking is an algorithmic verification technique by exhaustively searching the state space of the underlined model. It is attractive for enjoying two advantages: (1) it is fully automatic; (2) counterexamples are provided once a model checking fails. The original *explicit model checking* suffers from the notorious *state space explosion* problem. Model checking is enabled to scale by applying Ordered Binary Decision Diagrams (OBDD) to represent boolean formulas and Kripke structures, which results in the so-called *symbolic model checking* algorithms. Other important techniques such as *partial order reduction* further help to reduce the size of the state space. Furthermore, bounded model checking based on SAT solver, that bounds the length of paths to search counterexamples, is also effective for finding bugs [15].

The automata theory provides an alternative LTL model checking algorithm. That is, a system $\mathcal{A}$ satisfies a specification $\mathcal{S}$ if $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{S})} = \emptyset$, where $\mathcal{A}$ and $\mathcal{S}$ are automaton corresponding to the model and specification respectively. Thus, the decidability of a model checking problem is ascribed to the emptiness problem of automaton. The automata-theoretic approach also enables on-the-fly model checking [18]. Popular model checkers, such as Spin [6], NuSMV/SMV [5], are model checkers based on finite automata (i.e. finite state space). Finite model checking is successful in hardware verification (and some telecommunication protocols) that has a finite state space in nature. Automatic software validation is not easy, because software is inherently infinite. The infinities come from infinite program structures (e.g. nested procedure calls, recursions), infinite data domains (e.g. integers), concurrency, etc. Automatic software validation demands efforts from program analysis/abstraction to model checking techniques on infinite state space.

The pushdown system (PDS) is one kind of infinite state system which has an unbounded stack memory. The automata-theoretic approach works for pushdown model checking on regular properties, since the intersection of context-free language and regular language is closed, i.e. still context-free. The practical algorithms for pushdown model checking have been developed based on the key that regular pushdown stores are closed under forward and backward reachability [19]. Recently, they are implemented as tools, such as Moped, Weighted PDS library, WPDS++ library, etc. A relatively thorough treatment of verification on infinite structures is [20].

Obviously, the correctness of system validation based on model checking depends on whether a sound and enough precise (complete) abstraction can be made: an under-approximation may be unsound; whereas an over-approximation may cause false alarms.

[5]http://www.cs.cmu.edu/ modelcheck/code.htm

One novel solution towards this is the counterexample-guided abstraction refinement (CEGAR) approach [17], originated with lazy abstraction [16].

1.1.2 Program Analysis

Program analysis [6] [3] provides static approximations on the dynamic behaviors of programs when they are running on the computer. The key common to all program analysis is that only *safe* answers could be provided in order to remain decidable and sound. Originally, program analysis is one phase of the compilation. The result of program analysis shows the opportunities of compiler optimization, such as removing the redundant computations. The optimized object codes excels at either size or run-time efficiency. Nowadays, apart from compiler optimization, program analysis also plays a crucial role in a lot of fields such as program understanding, program transformations, software validation, etc.

There are four classic data flow analysis [2], so-called bit-vector analysis: *available expressions* analysis, *reaching definitions* analysis, *very busy expressions* analysis, and *live variables* analysis, etc. For instance, very busy expressions analysis says that:

> An expression is *very busy* at the program point p if along every path from p the expression is used before any variable in it is redefined.

A basic optimization based on it is *code hoisting.* That is, the expression is evaluated and the value is stored for later use. A live variable analysis says that:

> A variable x is *live* at some program point p if there exists a path from p such that x is used before redefined.

The result of live variable analysis helps *dead code elimination.*

Traditional data flow analysis is to find fix-point solutions on the flow equations or constraints, which are data flows facts abstracted from the program statements, in a forward or backward manner wrt control flows. For instance, the flow equation that defines the set of very busy expressions at some program point p is

$$\texttt{VBE}(p) = \texttt{Used}(p) \cup (\texttt{notModified}(p) \cap \bigcap_{p' \in succ\ p} \texttt{VBE}(p'))$$

and a greatest solution is interested in this analysis. For the live variable analysis, the set of live variables at some program point p is captured as the following flow equation:

$$\texttt{LiveVar}(p) = \texttt{Used}(p) \cup (\texttt{notDefined}(p) \cap \bigcup_{p' \in succ\ p} \texttt{LiveVar}(p'))$$

and a least solution is interested. This equation reads that:

> a variable is live after p if it is used at p or it is not defined at p and live after one of the next states p', where p' is one of the successor of p.

A well understood general framework for program analysis is based on so-called gen-kill functions. Typical instantiations are the classic bit-vector analysis. For instance, a variable is *generated* if it is used at some point; and *killed* if it is redefined.

[6] Throughout this thesis, we only mean static analysis.

Another influential analysis framework IFDS (interprocedural, finite, distributive, subset) is proposed in [38] by applying CFL-reachability to program analysis. CFL-reachability nicely capture the context-sensitivity, and can date back to [31]. Although previously there are work on program analysis based on CFL-reachability, they basically exploit restricted forms of CFL-reachability, such as Dyck language for balanced parentheses [35]. Reps provides solutions with general CFL-reachability to program analysis.

Later, it is found that only CFL-rechability is not enough for program analysis, such as constant propagation. Another framework called IDE (interprocedural distributive environment) [37] is proposed to solve problems whose data flow facts can be captured as transformer functions on program states. The analysis such as constant propagation is shown to be solved. The key of IDE is a graph representation, called *exploded supergraph*, of transfer functions. The idea is: control flows are exploded for individual variables; and edges capture the data flow dependence among variables, labeled with abstract transfer functions. Our first points-to analysis algorithms explore the exploded supergraph as the underlined model for model checking.

Program analysis are in general characterized by the following primary dimensions. There could be more dimensions for different problems. We will discuss more later when examining points-to analysis in detail.

- *intraprocedural* (*context-insensitive*) versus *interprocedural* (*context-sensitive*). Interprocedural analysis takes into account procedure calls, otherwise it is intraprocedural. In an interprocedural case, the analysis results wrt different procedure calls and returns cannot mix up, i.e context-sensitive; and some difficulties are caused by aliasing and call-by-reference mechanism.

- *forward* versus *backward*. By a forward analysis, we mean the data flow facts at some point depends on all data flows reaching at it; whereas, for a backward analysis, the data flow facts at some point depends on all outgoing data flows from it.

- *sequential* versus *concurrent*. Concurrency is a crucial feature of advanced programming languages in practice.It is shown that a context-sensitive interprocedural analysis of multi-tasking concurrent programs is undecidable [22]. Approximations on a context-sensitive program analysis concerning concurrency cannot be avoided.

With the advent of advanced programming languages, new program analysis is demanded, such as points-to analysis for C++ and Java, and the elimination of polymorphism for ML [21], etc. Since the treatments on languages of imperative style and functional style are a bit different, we consider object-oriented programming languages throughout this thesis.

1.1.3 Program Analysis = Abstract Interpretation + Model Checking

Static analysis on the concrete semantic domain of programs is undoubtedly undecidable, e.g. the well-known termination problem. Thus, proper approximations are needed to make an analysis trackable and even practical to reduce the problem size. Abstract interpretation [24] is a theory on sound approximation of the program semantics. There are further studies that examine the design of compositional abstract semantic domains.

$$\begin{tikzcd}
L \arrow[r, "f_c"] & L \arrow[d, "\alpha"'] \\
M \arrow[u, "\gamma"] \arrow[r, "f_a \sqsupseteq \alpha \circ f_c \circ \gamma"'] & M
\end{tikzcd}$$

Figure 1.1: Abstract Interpretation

A sound abstraction is often characterized by the *Galois connection.* That is, given two complete lattice $(L, \sqsubseteq)$ and $(M, \sqsubseteq)$ that are related by an abstraction function $\alpha : L \to M$ and a concretisation function $\gamma : M \to L$, such that (L, α, γ, M) is a Galois connection between L and M if and only if

- α and γ are monotone functions; and
- It is satisfied that

$$\gamma \circ \alpha \sqsupseteq \lambda x.x$$
$$\alpha \circ \gamma \sqsubseteq \lambda x.x$$

Galois connection ensures that an abstraction is safe although not precise enough (i.e. not necessarily complete). More specifically, if an error is detected in an abstract semantic domain, then it is really an error in the concrete semantic domain.

Given a Galois connection (L, α, γ, M), the static analysis can be performed on the abstract semantic domains of programs. Correspondingly, the original functions on the concrete domains $f_c : L \to L$ are "abstracted" as functions on the abstract domains $f_a : L \to L$. In particular, the analysis is sound if $f_a \sqsupseteq \alpha \circ f_c \circ \gamma$, as shown in Figure 1.1.

Originated with B. Steffen, D. A. Schmidt demonstrates that an iterative data flow analysis is a model checking of a modal logic formula on a program's abstract interpretation. For instance, the above analysis of very busy expressions can be reformulated as:

$$\texttt{isVBE}(e) = \nu Z.\ \texttt{isUsed}(e) \vee [\neg\texttt{isModeified}(e)]Z$$

Where $isVBE(e)$ states a very busy expression e in the propositional μ-calculus of Kozen [23]. Intuitively, "$[a]f$" means that a formula f holds in all states reachable by making a one-step transition labeled with a. Similarly, the live variable analysis can be reformulated as

$$\texttt{isLive}(x) = \mu Z.\texttt{isUsed}(x) \vee \langle\neg\texttt{isDefined}(x)\rangle Z)$$

where $\texttt{isLive}(x)$ declares a live variable x in the propositional μ-calculus. Informally, "$\langle a\rangle f$" means that a formula f possibly holds in a state reachable by making a one-step transition labeled with a.

The above observation shows a crucial connection among abstract interpretation, program analysis, and model checking, which provides an attractive systematic way to program analysis. The design (the front-end abstraction) and the solution (the back-end model checking) of a static analysis can be separated, with model checking as a general framework. The correctness of this methodology is correspondingly ensured by abstract interpretation and the model checking machinery.

As mentioned before, popular model checkers, such as Spin, NuSMV/SMV, are model checkers on finite state space. Program analysis based on them, such as BANDERA, is

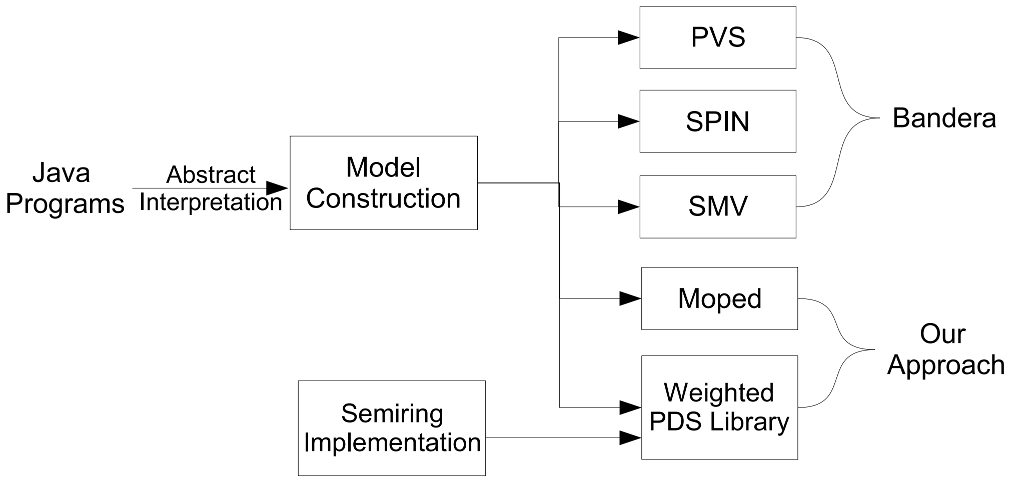


Figure 1.2: Our Methodology

intraprocedural in nature. Recently, the practical (weighted) pushdown model checking algorithms have been developed [7, 8] and implemented as tools, such as MOPED [7], Weighted PDS library [8], WPDS++ library [9], etc., which enables design of real *interprocedural* context-sensitive program analysis.

1.2 Contributions

Following to the methodology that *program analysis = abstract interpretation + model checking*, this thesis is dedicated to interprocedural program analysis based on weighted pushdown model checking. Java is selected as our analysis target, due to its popularity in practice and rich semantics that challenges the analysis effort. Our ideas can be applied to other object-oriented programming languages as well.

Our work is an *interprocedural* extension of Bandera [10]-like approach, i.e. program analysis design and prototype implementation with a model checker as the analysis engine, as shown in Figure 1.2. Bandera is a a tool set for automatic analysis generation for Java programs. It provides model constructions from Java programs to the inputs of several popular finite model checkers. Our work is to extend this methodology to interprocedural case based infinite model checking supported with prototype implementation.

More specifically, this thesis aims at exploring the following questions:

- Whether crucial program analysis, as well as design choices traditionally concerned in the analysis, can be solved as weighted pushdown model checking problems ?
- How the infinite model checking based approach benefits program analysis compared with traditional ones ?

We take *points-to analysis*, the most essential program analysis in Java as our first research target. Points-to analysis for Java is not a simple matter, primary difficulties

[7]http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/
[8]http://www.fmi.uni-stuttgart.de/szs/tools/wpds/
[9]http://www.cs.wisc.edu/wpis/wpds++/
[10]http://bandera.projects.cis.ksu.edu/

are: (1) points-to analysis and call graph construction are mutually dependent and quite equivalent; (2) the aliasing among references need to be cast for a precise analysis when field access is involved; (3) various infinities exist, such as the unbounded nesting of field access and array structures.

We propose *context/field/flow-sensitive* points-to analysis algorithms for Java based on weighted pushdown model checking. We also show how the above problems are settled as a model checking problem with proper abstraction. Our analysis is characterized by the following dimensions, as summarized in Table 1.1.

- An *exploded supergraph* vs an *interprocedural control flow graph* based model design
  The exploded supergraph and control flow graph are typical models for program analysis. In our analysis, both of them are explored as the underlined model for model checking. We show how points-to analysis is formulated within the same framework by these two different model designs.

  The analysis based on exploded supergraph is more likely to scale, due to the efficient graph representation of transfer functions. However, an on-the-fly analysis dispatches overwhelming model checking requests. It deserves a try on automatically removing invalid paths from the analysis result. The automatic path removal is restricted by the exploded supergraph based model design, since each control flow is exploded for individual variables. In contrast, an interprocedural CFG based approach provides facility of algorithm constructions. We propose an ahead-of-time analysis algorithm based on the interprocedural CFG model design, which can be done in one-run model checking. The choice is a tradeoff among efficiency and design choices.

- An *on-the-fly* vs an *ahead-of-time* algorithm construction
  Since call graph construction and points-to analysis are mutually dependent, iterative procedures are inevitable to ensure soundness. There are two ways to solving this mutual dependency: an on-the-fly manner is to find the least solution, i.e. starting with the empty call graph, a call edge is added if points-to analysis detects its possibility; and an ahead-of-time manner is to find the greatest solution, i.e. starting with an imprecise call graph, a call edge is eliminated if points-to analysis refutes it. In our analysis, both of these flavors are investigated.

  Although an on-the-fly manner is likely to be more precise by empirical studies, to handle various iterations in the algorithm construction needs delicate cares. In contrast, an ahead-of-time analysis with automatic path removal and bounded field tracing shows that, more design efforts on iterative procedures can be handed over to the back-end model checker.

  Our study also shows that an ahead-of-time construction needs to explicitly give a bound on tracing nesting of field access, whereas an on-the-fly construction automatically bounds it (up to the number of abstract heap location).

- Parametrization
  Furthermore, the relatively unexplored problem of *parametrization* is explored. Parameterized *flow-sensitivity* is naturally obtained in our algorithms by model reduction in the analysis based on the explode supergraph; and by simplifying the weight space design based on the interprocedural CFG.

	Exploded Supergraph	Interprocedural CFG
On-the-fly	Chapter 4.1.2	Chapter 5.1.2
Ahead-of-time	Chapter 4.1.3	Chapter 5.1.3
Parameterized flow-sensitivity	Model Reduction (Chapter 4.2)	Weight Design Simplification (Chapter 5.2)

Table 1.1: Design Choices Concerning Three Primary Dimensions

- Prototype implementation
  Most of combinations, expect for the one with ahead-of-time call graph construction based on the model design of exploded supergraph , are also supported with implementations. A prototype framework is presented, which combines SOOT [11] compiler as preprocessing to convert Java to Jimple and the Weighted PDS (pushdown system) library as the back-end model checking engine.

Given the result of points-to analysis, other program analyses, so-called *client applications*, are possible. We further explore an interprocedural irrelevant code elimination analysis. Compared with dead code elimination, our irrelevant code elimination is more on a semantical sense. The interaction among procedures is essentially captured by the variable (both local and global) dependency based on the exploded supergraph, under PER based abstraction. Our motivation on this work is that dead code elimination is essentially intraprocedural. Even interprocedural ones are only talking about global variables. We present encodings of dead code eliminations based on both pushdown and weighted pushdown model checking.

1.3 Thesis Organization

The rest of the thesis is organized as follows, by making each chapter self-contained to the best of my abilities.

- Chapter 2 introduces the basic ideas of model checking algorithms on pushdown systems, including pushdown model checking on regular properties based on simple valuations, and the key idea of weighted pushdown model checking algorithms. The motives and application of weighted pushdown model checking to interprocedural program analysis is presented, as well as possible encodings of programs as pushdown systems.

- Chapter 3 first introduces points-to analysis for Java and shows the difficulties. Then, a detailed discussion on the design choices that we concerned in our analysis is presented. Related work is finally given.

- Chapter 4 presents our first points-to analysis algorithms based on an exploded supergraph model desgin. Both on-the-fly and ahead-of-time algorithm construction of the analysis are explored. In particular, parameterized flow-sensitivity is realized by model reduction. A prototype framework is presented with an explanation on the implementation aspects. A preliminary evaluation is presented with discussion on efficiency.

[11]http://www.sable.mcgill.ca/soot/

- Chapter 5 introduces a new points-to analysis algorithms based on an interprocedural CFG model design. Both on-the-fly and ahead-of-time call graph construction are also investigated. Especially, the ahead-of-time analysis under such a choice enables an automatic path removal and field tracing by the machinery of model checking. The analysis can be performed in one-run model checking. Parameterized flow-sensitivity is also obtained by simplifying the weight design. An explanation on the the implementation aspects is presented.

- Chapter 6 explores an interprocedural irrelevant code elimination algorithm based on weighted pushdown model checking. First, how a dead code elimination analysis can be solved by both pushdown and weighted pushdown model checking is shown. The interprocedural flavor is possibly obtained by global variable renaming with preprocessing. An interprocedural irrelevant code elimination is further presented, by capturing the interaction among procedures based on the exploded supergraph construction.

- Chapter 7 concludes the thesis with a presentation of our future plans.

Chapter 2

Pushdown Model Checking

Model checking is an algorithmic verification of finite or infinite transition systems, by exhaustively exploring the state space of systems. Finite model checking is successful in hardware verification, because hardware has a finite state space in essence. Software is essentially infinite. To apply model checking to software validation demands infinite model checking techniques plus expertise on abstraction.

The pushdown system (PDS) is one kind of infinite-state systems. It is a finite transition system carrying an unbounded stack memory. Other infinite-state systems are like parameterized systems, Petri Nets, etc. In particular, the verification on parameterized systems has been extensively studied, such as regular model checking [25]. These aforementioned infinite systems are discrete-time. Infinite systems working on dense time are such as timed automata, hybrid systems, etc.

PDSs are pushdown automata with different concerns. PDSs are not interested in the languages accepted but the transition graph generated from running the transition system. The automata-theoretic approach works for pushdown model checking on regular properties, since the intersection of context-free languages and regular languages are closed, i.e. context-free. The study of pushdown model checking algorithms follows to several approaches. For instance, one approach regards the problem of pushdown model checking as games on the transition graph of PDSs [27]. Another stream of the study on pushdown model checking follows to the automata-theoretic approach [26]. The key of this stream is based on the following observation: *regular pushdown stores are closed under forward and backward reachability.* In this thesis, we are interested in the latter. The pushdown model checking problem on CTL (computation tree logic) and μ-calculus is known to be DEXPTIME-complete. Whereas pushdown model checking on LTL (linear time logic) is known to be polynomial.

Pushdown systems are extended to weighted pushdown systems by associating each pushdown transition with a weight. The weight comes from a bounded idempotent semiring. The model checking problem on weighted pushdown system is generalized reachability analysis, i.e. reachability analysis between regular pushdown configurations on the transition graph of PDSs with calculating and satisfying weight constraints. Rather following to automata-theoretic approach, weighted pushdown model checking is first settled as a grammar valuation problem by grammar characterization of pushdown transitions [8]. A more efficient algorithm is also developed by creating an annotated automata [46]. The introduction of weighted pushdown model checking targets on interprocedural program analysis.

An algebraic view on the analysis of graph problems is not new. For instance, some general path-finding problems over closed semiring are presented in [29, 30]. However, the general graph reachability analysis is not applicable for context-sensitive program analysis. CFL-reachability nicely captures the demands that procedure calls and returns are correctly paired in the program analysis. Whereas, some program analysis problems are beyond CFL-reachability, such as constant propagation. Weighted pushdown systems provide a generalized framework by combining both of these respects.

This chapter is organized as follows:

In Section 2.1, some definitions and terminologies on model checking on pushdown systems are prepared.

In Section 2.2, the key of an efficient pushdown model checking algorithm on regular properties is presented, based on *simple valuations*.

In Section 2.3, model checking algorithms on weighted pushdown systems, an extension of pushdown systems, are presented. A grammar valuation based algorithm by reformulation of pushdown transitions as grammar productions is presented first. A more efficient algorithm is later introduced by annotated automaton construction.

In Section 2.4, we show the application of weighted pushdown model checking to interprocedural program analysis. The typical encoding of programs as pushdown systems is also presented.

2.1 Preliminaries

Given a model $\mathcal{M}$ and some property in temporal logic formulas $\mathcal{L}$, model checking answers the question that whether the model satisfies with the property, represented as $\mathcal{M} \models \mathcal{L}$. Here, we are interested in the model checking algorithm when $\mathcal{M}$ is a pushdown system and $\mathcal{L}$ is regular properties specified in LTL.

2.1.1 Linear Time Logic

First of all, LTL (linear time logic) is briefly prepared to make our contents are self-contained (we mean propositional LTL without specific declaration). As a temporal logic, the underlying time model of LTL is linear, that is, each moment has only one possible future time evolution.

The LTL formulas are formed from atomic proposition with connectives such as $\neg$, $\vee$, $\wedge$, etc. The basic temporal operators of LTL are shown as follows, for some LTL formula f, p, and q

- $\mathbf{F}f$ reads that f holds *eventually* on some state.
- $\mathbf{G}f$ reads that f *always* holds.
- $\mathbf{X}f$ reads that f holds on the next state.
- $p\mathbf{U}f$ reads that p always holds until a state on which q holds.

To note that, the until operator U is called the *strong* until operator. It is *weak* by loosening that q is allowed not to happen and p essentially always holds on an interminating run. Among these temporal operators, only $\mathbf{X}$ and $\mathbf{U}$ are essential. For instance,

$$\mathbf{F}f \equiv \text{true}\mathbf{U}f; \quad \text{and} \quad \mathbf{G}f \equiv \neg\mathbf{F}\neg f$$

Assume AP is the finite set of atomic propositions of LTL. It is well understood that properties expressed in LTL are $\omega-$languages over the alphabet 2^{AP}. The historically-chosen Kripke Structure also directly corresponds to the ω-language. This provides an alternative, i.e. the automata-theoretic approach, model checking algorithm. The model checking problem is reduced to checking the emptiness problem of automata.

Example 1 *Figure 2.1 illustrates a Büchi automaton that represents the simple liveness properties in LTL* ***F****q. By liveness property, we mean that "some thing good will eventually happen".*

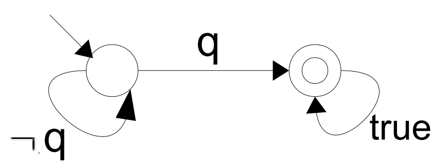


Figure 2.1: A Liveness Property in Büchi Automaton

2.1.2 Pushdown System

A pushdown system is a finite transition system with an unbounded stack. Compared with pushdown automata, pushdown systems concerns the transition graph of the system rather than languages accepted.

Definition 1 *A* ***pushdown system*** $P = (Q, \Gamma, \Delta, q_0, w_0)$ *is a pushdown automata regardless of input, where* Q *is a finite set of states,* Γ *is a finite set of stack alphabet,* $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$ *is a set of transition rules,* $q_0 \in Q$ *is the initial state,* $w_0 \in \Gamma^*$ *is the initial stack contents, satisfying that* $\forall \gamma \in \Gamma, \omega \in \Gamma^*, p,\ q \in Q,$

$$\textit{if } \langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle, \ \textit{then } \langle p, \gamma\omega' \rangle \Rightarrow \langle q, \omega\omega' \rangle, \forall\ \omega' \in \Gamma^*$$

where if $((q_1, w_1), (q_2, w_2)) \in \Delta$*, it is represented by* $\langle q_1, w_1 \rangle \hookrightarrow \langle q_2, w_2 \rangle$*.*

A ***configuration*** *of* P *is a pair* $\langle q, w \rangle$*, where* $q \in Q$ *and* $w \in \Gamma^*$*. The* ***head*** *of a configuration* $\langle p, \gamma\omega \rangle$ *is* $\langle p, \gamma \rangle$*, where* $\gamma \in \Gamma$*. The* ***head*** *of a pushdown transition* $\langle q_1, w_1 \rangle \hookrightarrow \langle q_2, w_2 \rangle$ *is* $\langle q_1, w_1 \rangle$*.*

A ***valuation*** *of* P *is the function* $\nu : AP \to 2^{Q \times \Gamma^*}$*.*

In Definition 1, Q is usually called *control locations*. The *configuration* of pushdown systems is a pair of control locations and stack contents, and a configuration is a state in the transition graph of pushdown systems. The definition also says that, a pushdown system make a transition based on the current control location and the topmost stack symbol; and a modification on the stack can only be done on the top of the stack.

Furthermore, if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then for every $\omega' \in \Gamma^*$, $\langle p, \gamma\omega' \rangle$ is called the *immediate predecessor* of $\langle q, \omega\omega' \rangle$, and similarly $\langle q, \omega\omega' \rangle$ is called the *immediate successor* of $\langle q, \omega\omega' \rangle$.

As a matter of easy representation, it is often restricted that, Let P be a pushdown system,

$$\text{for each } \langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle \text{ in } P, \text{ s.t. } |\omega| \leq 2$$

A pushdown system can be simulated by another pushdown system satisfying the above restriction [28].

The automata-theoretic approach amounts to checking whether the intersection of the automata wrt LTL properties and transition systems is empty. Therefore, by *valuation*, we mean assigning atomic propositions to the states of the transition system. For pushdown systems, the states are pushdown configurations of the pushdown transition graph. There are various choices on the way of valuation. Pushdown model checking on arbitrary valuations is undecidable. However, it remains decidable for some restricted class of valuations.

A basic choice of valuation is so-called *simple valuation*, for which only topmost stack symbol, as well as the current control location, matters for the atomic proposition.

Definition 2 *Let $P = (Q, \Gamma, \Delta, q_0, w_0)$ be a pushdown system, and and let AP be a finite set of atomic proposition. Let label : $Q \times \Gamma \to AP$ be a labeling function. A simple valuation $\nu : AP \to 2^{Q \times \Gamma^*}$ is defined as, for each $\psi \in AP$, $\nu(\psi) = \{\langle p, \gamma\omega\rangle \mid \psi \in$ label $(p, \gamma),\ \omega \in \Gamma^*\}$.*

A more general extension to simple valuation is so-called *regular valuation*, for which a regular words over the pushdown stack alphabet have been associated to an atomic proposition wrt some control location. That is, intuitively, an atomic proposition ψ is valid for a configuration $\langle p, \omega\rangle$, if and only if the reverse of the stack contents ω is accepted by some finite automata wrt ψ and p. Pushdown model checking on regular properties with regular valuations can be applied to problems such as Java stack inspection []. The model checking algorithm on regular valuation is solved by encoded as a simple valuation problem. Thus, only model checking algorithm with simple valuation is presented.

Definition 3 *Let $P = (P, \Gamma, \Delta, q_0, w_0)$ be a pushdown system, and let AP be a finite set of atomic proposition. Each pair of (ψ, p) $(\psi \in AP, p \in Q)$ is associated with a deterministic finite-state automaton $\mathcal{M}^p_\psi$. A regular valuation $\nu : AP \to 2^{Q \times \Gamma^*}$ is defined as, for each $\psi \in AP$, $\nu(\psi) = \{\langle p, \omega\rangle \mid \omega^R \in L(\mathcal{M}^p_\psi)\}$.*

Following to the automata-theoretic approach, the key of efficient pushdown model checking algorithm is based on the observation that

> The set of regular pushdown configurations is closed under forward and backward reachability.

In order to represent the regular set of pushdown configurations, a so-called P-automaton is introduced.

Definition 4 *Let $P = (Q, \Gamma, \Delta, q_0, w_0)$ be a pushdown system. A P-automaton is an automaton $A = (D, \Gamma, \delta, Q, F)$, where D is the finite set of states, $\delta \subseteq D \times \Gamma \times D$ is a finite set of transitions, Q is the finite set of initial states, and F is the set of final states.*

If $(q_1, \gamma, q_2) \in \delta$, then it is denoted by $q_1 \xrightarrow{\gamma} q_2$. A pushdown configuration $\langle p, \omega\rangle$ is accepted by the P-automaton if there exists some $q \in F$ such that $p \xrightarrow{\omega} q$. In particular, $\forall q \in D$, $q \xrightarrow{\varepsilon} q$.

The last requirement says that a set of configurations is the immediate predecessor or successor of itself.

Let $\mathcal{C}$ be the set of pushdown configurations. Let $Pre : 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ be the function that computes the predecessors of configurations, such that

$$pre^*(C) = \{c \in \mathcal{C} \mid \exists c' \in C, \text{ s.t. } c \Rightarrow^* c'\}$$

Given a P-automaton A that accepts the set of configurations C, it is straightforward to construct the automaton $A_{pre^*(C)}$ that accepts $pre^*(C)$. Assume there is a pushdown transition $\langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle$. Thus, if the configuration $\langle q, \omega\omega'\rangle$ is accepted by A for some $\omega' \in \Gamma^*$, then $\langle p, \gamma\omega'\rangle$ is also accepted by A. The principle is summarized as follows

$$\boxed{\begin{array}{l}\text{If } \langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle, \text{ and } q \xrightarrow{\omega} s \text{ in the } P\text{-automaton,}\\ \text{then a transition } (p, \gamma, s) \text{ is added to the } P\text{-automaton}\end{array}}$$

The above procedure proceeds until no more rules can be added. The idea of constructing the successors of regular configurations is similar. Please refer to [7] for more details.

Example 2 *A pushdown system* $\mathcal{P} = (\{q_0, q_1, q_2\}, \{w_0, w_1, w_2, w_3\}, \Delta, q_0, w_0)$*, the pushdown transitions in* Δ *are as follows:*

$$\begin{array}{ll}
(1) & \langle q_0, w_0\rangle \hookrightarrow \langle q_1, w_1 w_0\rangle\\
(2) & \langle q_1, w_1\rangle \hookrightarrow \langle q_2, w_2\rangle\\
(3) & \langle q_1, w_1\rangle \hookrightarrow \langle q_2, w_3\rangle\\
(4) & \langle q_2, w_3\rangle \hookrightarrow \langle q_0, \epsilon\rangle\\
(5) & \langle q_2, w_2\rangle \hookrightarrow \langle q_0, \epsilon\rangle
\end{array}$$

In Figure 6.3, the left-hand-side illustrates the changes of stack contents wrt pushdown transitions. These three kinds of transitions (i.e. modifications on the stack) are typical; and the right-hand-side illustrates the corresponding transition graph of $\mathcal{P}$*.*

The P*-automaton* A *that accepts the a singleton set of configurations* $C = \{\langle q_2, w_3 w_0\rangle\}$ *is shown on the left-hand-side of Figure 2.3 and the automaton that accepts* $A_{pre^*(C)}$ *is shown on the right-hand-side.*

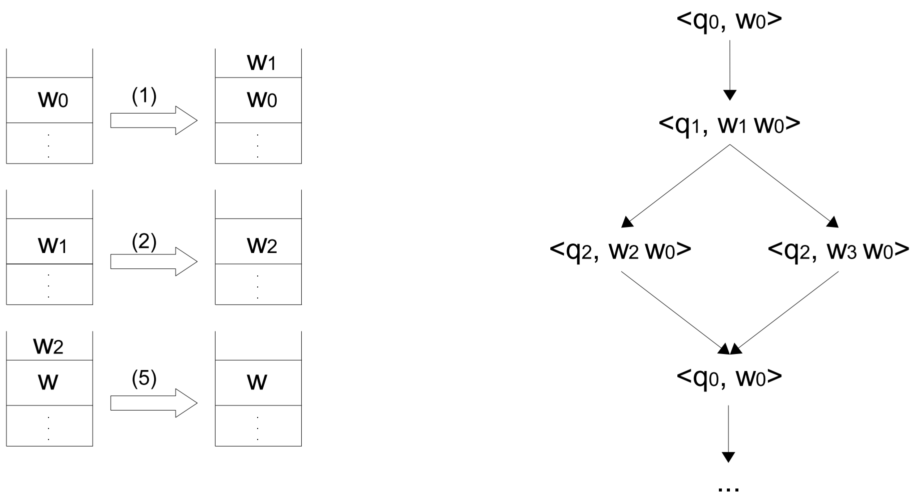
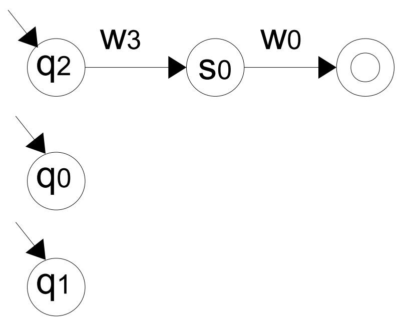


Figure 2.2: The Transition Graph of $\mathcal{P}$



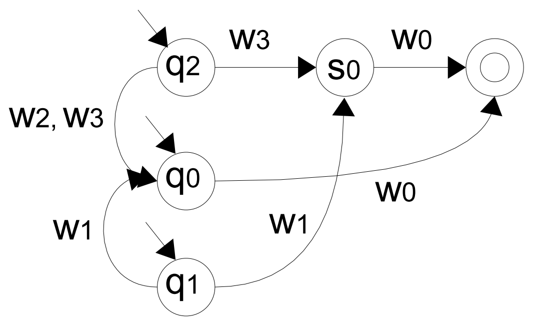


Figure 2.3: The automata accepting regular pushdown configurations C and $pre^*(C)$

2.2 Pushdown Model Checking on Regular Properties

We will present pushdown model checking on LTL properties with simple valuations. With regard to simple valuations, we can construct the product of pushdown systems and Büchi automata.

Definition 5 *Let $P = (Q, \Gamma, \Delta, q_0, w_0)$ be a pushdown system. Let AP be the finite set of atomic propositions for LTL and ψ is a LTL formula. Let $B = (S, 2^{AP}, \delta, q_0, F)$ be a Büchi automata that accepts $\neg\psi$. A Büchi pushdown system $BP = (D, \Gamma, \Delta', q_0', w_0, F')$ is a product of P and B, satisfying that*

- $D = Q \times S$
- $(((p_1, q_1), \gamma), ((p_2, q_2), \omega)) \in \Delta'$, *if*
  - $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, \omega\rangle \in \Delta$
  - $q_1 \xrightarrow{\psi} q_2$ *and* $\psi \subseteq \nu(q_1, \gamma)$

  and $(((p_1, q_1), \gamma), ((p_2, q_2), \omega))$ *is represented as* $\langle (p_1, q_1), \gamma\rangle \rightharpoonup \langle (p_2, q_2), \omega\rangle$.
- $(p, q) \in F'$ *if* $q \in F$

A head of $\langle q, \gamma\rangle$ of BP is ***repeated*** *if there exists some $\omega \in \Gamma^*$, such that*

- $\langle q, \gamma\rangle \rightharpoonup^* \langle q, \gamma\omega\rangle$; *and*
- *there exists some configuration $\langle q', \upsilon\rangle$ of BP, where $q' \in F', \upsilon \in \Gamma^*$, such that $\langle q, \gamma\rangle \rightharpoonup^* \langle q', \upsilon\rangle \rightharpoonup^+ \langle q, \gamma\omega\rangle$, represented as $\langle q, \gamma\rangle \rightharpoonup^* \langle q, \gamma\omega\rangle$.*

Given a pushdown system $P = (Q, \Gamma, \Delta, q_0, w_0)$ and a LTL formula φ, let B be the Büchi automata that accepts $\neg\varphi$. Therefore, the the pushdown model checking problem $P \models \varphi$ amounts to answering

> Whether there is an accepting run in the product BP of P and B that starts from $\langle q_0, w_0\rangle$ and visits some finite states infinitely often.

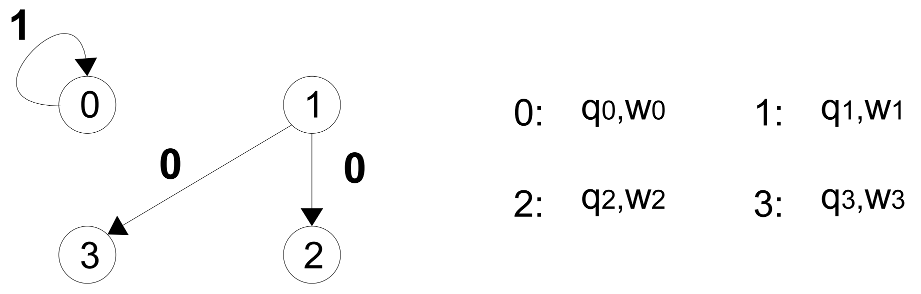


Figure 2.4: The Head Reachability Graph

Namely, the emptiness problem of Büchi pushdown system. Let R be the set of accepted heads of BP, define

$$R\Gamma^* = \{\langle q, \gamma\omega\rangle \mid \langle q, \gamma\rangle \in R,\ \omega \in \Gamma^*\}$$

Then, BP has an accepting run starting from some configuration c if $c \in pre^*(R\Gamma^*)$.

Therefore, the problem is reduced to computing the set of repeated heads. [7] proposes a more efficient method of computing the repeated heads of the Büchi pushdown system than [26], by constructing a so-called head reachability graph $G = ((D \times \Gamma), \{0,1\}, E)$, given a Büchi pushdown system $BP = (D, \Gamma, \Delta', q_0', w_0, F')$. The set of edges $E \subseteq (D \times \Gamma) \times \{0,1\} \times (D \times \Gamma)$ is constructed as follows:

If there exists $\langle q_1, \gamma_1\rangle \hookrightarrow \langle q_2, \omega_1\gamma_2\omega_2\rangle$,
(1) if $\langle q_2, \omega_1\rangle \hookrightarrow^* \langle q_3, \varepsilon\rangle$, then $((q_1, \gamma_1), G(q_1), (q_2, \gamma_2)) \in E$; and further
(2) if $\langle q_2, \omega_1\rangle \hookrightarrow_r^* \langle q_3, \varepsilon\rangle$, then $((q_1, \gamma_1), 1, (q_2, \gamma_2)) \in E$

Where $G(q) = 1$ if $p \in F'$, and $G(q) = 0$ otherwise. Then the set of repeated heads can be computed:

A head is repeated iff it belongs to a strongly connected component (SCC) of G, and this SCC has an edge labeled with 1.

Example 3 *For simplicity, just abuse the previous pushdown system in Example 2 as a Büchi pushdown system with $\{q_2\}$ as the final states. Figure 2.4 shows its head reachability graph.*

2.3 Weighted Pushdown Model Checking

A weighted pushdown system extends a pushdown system by associating a weight to each transition rule. The weights come from a bounded idempotent semiring.

Definition 6 *A **bounded idempotent semiring** is a quintuple $S = (D, \oplus, \otimes, 0, 1)$, where a set D, and two binary operations $\oplus$ and $\otimes$ on D such that*

1. *$(D, \oplus)$ is a commutative monoid with 0 as its neutral element, and $\oplus$ is idempotent.*
2. *$(D, \otimes)$ is a monoid with 1 as the neutral element.*

3. *$\otimes$ distributes over $\oplus$, that is, $\forall a,b,c \in D$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.*

4. *$\forall a \in D, a \otimes 0 = 0 \otimes a = 0$.*

5. *The partial order $\sqsubseteq$ is defined on D: $\forall a,b \in D, a \sqsubseteq b$ iff $a \oplus b = a$, and there are no infinite descending chains on D wrt $\sqsubseteq$.*

Definition 7 *A* ***weighted pushdown system*** *is a triple $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, and $f\colon \Delta \to D$ is a function that assigns a value from D to each rule of P.*

Example 4 *A weighted pushdown system $W = (\mathcal{P}, S, f)$ extends the pushdown system $\mathcal{P}$ in Example 2, where $S = (D, \oplus, \otimes, 0, 1)$ is a bounded semiring with $1 \sqsubseteq 0$, where $D = \{$TRUE,FALSE$\}$, and TRUE is 1 and FALSE is 0.*

The $\oplus$ and $\otimes$is defined as:

$$\forall a \in D, a \oplus 0 = 0 \oplus a = a$$

$$\forall a \in D, a \otimes 0 = 0 \otimes a = 0$$

$$\forall a \in D, a \otimes 1 = 1 \otimes a = a$$

And $f\colon \Delta \to D$ is defined as:

$(1)\langle q_0, w_0\rangle \hookrightarrow \langle q_1, w_1 w_0\rangle$	TRUE
$(2)\langle q_1, w_1\rangle \hookrightarrow \langle q_2, w_2\rangle$	TRUE
$(3)\langle q_1, w_1\rangle \hookrightarrow \langle q_2, w_3\rangle$	FALSE
$(4)\langle q_2, w_3\rangle \hookrightarrow \langle q_0, \epsilon\rangle$	TRUE
$(5)\langle q_2, w_2\rangle \hookrightarrow \langle q_0, \epsilon\rangle$	TRUE

The transition graph with weight computation is shown in Figure 2.5.

Definition 8 *Given a weighted pushdown system $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, and $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring. Assume $\sigma = [r_0, ..., r_k]$ be a sequence of pushdown transition rules, where $r_i \in \Delta(0 \leq i \leq k)$, and $val(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$. Let path(c,c′) be the set of all rule sequences that transform configurations from c into c'. Let $C \subseteq Q \times \Gamma^*$ be a set of regular configurations, the* ***generalized pushdown predecessor problem****(GPP) is to find for each $c \in Q \times \Gamma^*$:*

$$\delta(c) = \bigoplus\{val(\sigma) | \sigma \in path(c, c'), c' \in C\}$$

A witness set of paths $w(c) \subseteq \bigcup_{c' \in C}$ such that $\oplus_{\sigma \in w(c)} val(\sigma) = \delta(c)$.

The generalized pushdown successor (GPS) problem can be similarly defined.

The above definition says that, the $\otimes$ operation propagates data along each path, and the $\oplus$ operator combines pathes by satisfying the ordering $\sqsubseteq$.

In the following of this section, the key idea of an efficient algorithm of solving GPP problem is presented. Weighed pushdown model checking algorithms are implemented and provided as libraries. They are Weighted PDS and WPDS++. We will apply the former to our prototype implementation.

Remark 1 *As stated in Section 4.4 in [46], the distributivity of $\oplus$ can be loosened to $a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$.*

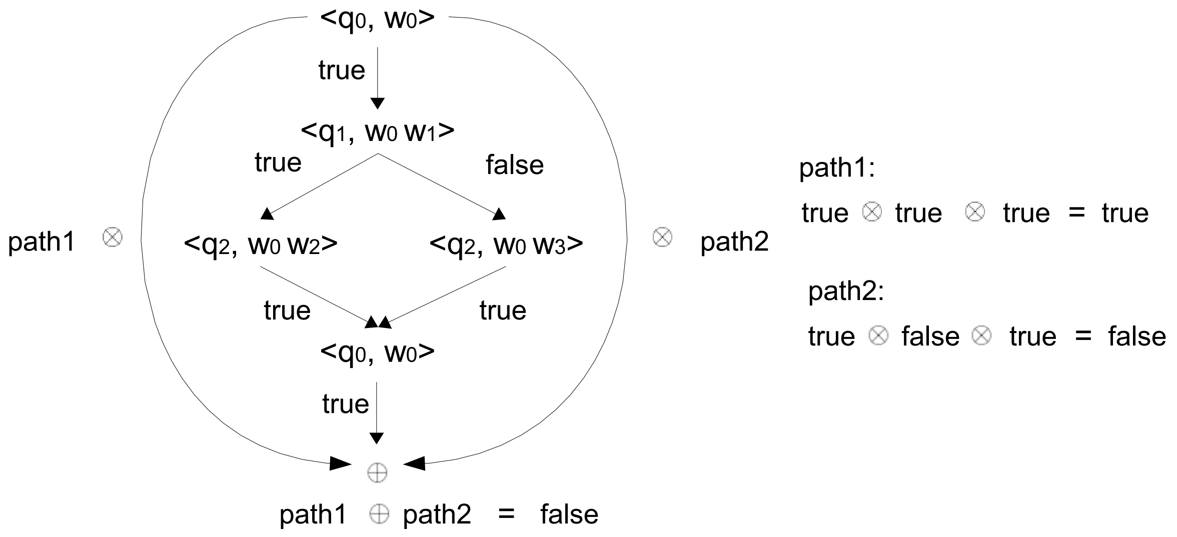


Figure 2.5: The Transition Graph of W

The Grammar-based Characterization and Solution

The first algorithm of weighted pushdown model checking is based on a traditional grammar valuation algorithm on the context-free grammar, by characterizing pushdown transitions as productions of a CFL grammar.

Definition 9 *A combined pushdown system $PA = (D, \Gamma, \Delta', q_0, w_0)$ is a combination of a pushdown system $P = (Q, \Gamma, \Delta, q_0, w_0)$ and the P-automaton $A = (D, \Gamma, \delta, Q, F)$ wrt the set of configurations C, where $\Delta' = \Delta \cup \texttt{Trans}$, where*

$$\texttt{Trans} = \{\langle q_1, \gamma\rangle \hookrightarrow \langle q_2, \varepsilon\rangle \mid q_1 \xrightarrow{\gamma} q_2 \in \delta\}$$

A configuration is c accepted by PA if there is a path from c to a configuration $\langle q, \varepsilon\rangle$, such that $q \in F$.

Since there is no transitions leading to the initial states in A, whenever a non-initial state is encountered during the pushdown transitions, only a sequence of pop transitions are possibly applied and leads to a final state in F. Therefore, given C, and starting from a pushdown configuration c, the key idea of the solution is:

> The transitions of PA can be regarded as two phases: First, PA mimics P; once PA reaches a non-initial state in A (i.e. $D \setminus Q$), PA starts mimicking A and only pop transitions are possible. If a final state in F is reached finally, and the stack is empty, then $c \in pre^*(C)$.

In view of this, a context-free grammar that characterizes pushdown transitions is defined. Before presenting it, a concept called *pop sequence* need to be prepared.

Definition 10 *Let $PA = (Q, \Gamma, \Delta, q_0, w_0)$ be a combined pushdown system. A pop sequence for $q \in Q$, $\gamma \in \Gamma, q' \in Q$, denoted by $PopSeq(q, \gamma, q')$, is a sequence of transitions of PA satisfying that*

- *starts with a configuration $\langle q, \gamma\omega\rangle$, where $\omega \in \Gamma^*$; and*
- *ends with a configuration $\langle q', \omega\rangle$; and*
- *the stack is always of the form $\omega'\omega$ for some $\omega' \in \Gamma^*$ except for the last step.*

It is not hard to see that: if there exists $PopSeq(q, \gamma, q')$ wrt an untouched stack ω, iff there exists $PopSeq(q, \gamma, q')$ wrt the stack ε.

Figure 2.6 depicts the rules for grammar characterization of pushdown transitions, and the generated grammar is shown in Table 2.1. The left-hand-side gives the pushdown transition rule of the combined pushdown system, the left-hand-side gives the productions of the generated grammar.

In particular, the last two extra productions are added. We can see that the set of pre^* witnesses for $\langle p, \gamma_1\gamma_2...\gamma_n\rangle$ is characterized by the complete derivation trees derivable from the nonterminal $Accepted[\gamma_1...\gamma_n](p)$.

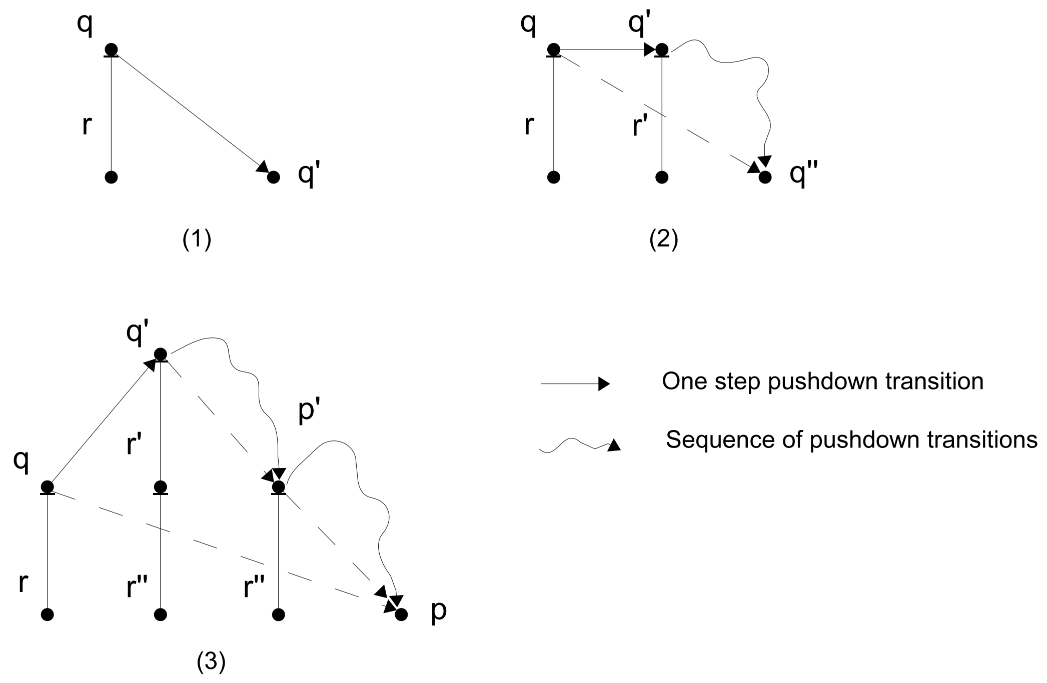

q
r
q'
(1)
q
q'
r
r'
q''
(2)
q'
p'
q
r'
r
r''
r''
p
(3)
One step pushdown transition
Sequence of pushdown transitions

Figure 2.6: The Grammar Characterization of Pushdown Transitions

(1)	$PopSeq(q, \gamma, q') \rightarrow \varepsilon$	$\langle q, \gamma\rangle \hookrightarrow \langle q', \varepsilon\rangle$
(2)	$PopSeq(q, \gamma, q') \rightarrow PopSeq(q'', \gamma', q')$	$\langle q, \gamma\rangle \hookrightarrow \langle q'', \gamma'\rangle$
(3)	$PopSeq(q, \gamma, p) \rightarrow PopSeq(q', \gamma', p')PopSeq(p', \gamma'', p)$	$\langle q, \gamma\rangle \hookrightarrow \langle q', \gamma'\gamma''\rangle$
(4)	$Accepting[\gamma_1...\gamma_n](p, q) \rightarrow PopSeq(p, \gamma_1, q_1), ..., PopSeq(q_{n-1}, \gamma_n, q)$ for $p \in Q, q_i \in D$ $(1 \leq i \leq n-1)$ and $q \in F$	
(5)	$Accepted[\gamma_1...\gamma_n](p) \rightarrow Accetping[\gamma_1...\gamma_n](p, q)$, for $p \in Q, q \in F$	

Table 2.1: The Grammar Characterization of Pushdown Transitions

Therefore, Given a domain of interest, by associating each grammar production with a function g on this domain that propagates the "value" of grammar derivations in the backward manner wrt the derivation tree (also as shown in table 2.2), the GPP of weighted pushdown systems can be solved be a traditional grammar evaluation problem.

(1)	$PopSeq(q, \gamma, q') \rightarrow g_1(\varepsilon)$	$r = \langle q, \gamma\rangle \hookrightarrow \langle q', \varepsilon\rangle$
	$g_1 = 1$ if $r \in \delta$ else $g_1 = f(r)$	
(2)	$PopSeq(q, \gamma, q') \rightarrow g_2(PopSeq(q'', \gamma', q'))$	$\langle q, \gamma\rangle \hookrightarrow \langle q'', \gamma'\rangle$
	$g_2 = \lambda x.f(r) \otimes x$	
(3)	$PopSeq(q, \gamma, p) \rightarrow g_3(PopSeq(q', \gamma', p'), PopSeq(p', \gamma'', p))$	$\langle q, \gamma\rangle \hookrightarrow \langle q', \gamma'\gamma''\rangle$
	$g_3 = \lambda x.\lambda y.f(r) \otimes x \otimes y$	
Furthermore,		
(4)	$Accepting[\gamma_1 ... \gamma_n](p, q) \rightarrow g_4(PopSeq(p, \gamma_1, q_1), ..., PopSeq(q_{n-1}, \gamma_n, q))$ for $p \in Q, q_i \in D$ $(1 \leq i \leq n-1)$ and $q \in F$	
	$g_4 = \lambda x_1 ... \lambda x_n.x_1 \otimes ... \otimes x_n$	
(5)	$Accepted[\gamma_1 ... \gamma_n](p) \rightarrow g_5(Accetping[\gamma_1 ... \gamma_n](p, q))$, for $p \in Q, q \in F$	
	$g_5 = \lambda x.x$	

Table 2.2: The Grammar Characterization of Pushdown Transitions

This solution is not efficient because starting with terminals, all possible derivations (including those not contribute the final result of interest) are explored.

The Annotated Automata-based Soluation

More efficient algorithms are developed by borrowing the idea of computing $pre^*(C)$ and $post^*(C)$ for pushdown model checking. Fore weighted pushdown model checking, a weighted version is developed for $pre^*(C)$, given a set C of regular configurations. Initially, a P-automaton A is constructed wrt C, and each edge in A is labeled with the weight 1. Edges are added and the labels are updated according to the some rules. Here we provide a basic sketch of the algorithm construction, helped with figures.

Given a weighted pushdown system $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$, and $S = (D, \oplus, \otimes, 0, 1)$. Given a P-automaton $A = (D, \Gamma, \delta, Q, F)$ that accepts some set C of configurations, and there are no transitions into states in P initially. Let `getLabel` be the function that keeps the up-to-date mappings between edges and their labeled weights in A. The algorithm of computing $pre^*(C)$ is an iterative procedure as follows, as well as explanations.

1. Initialization
   - `getLabel` $= \lambda t.0$, for $t \in \delta$
   - workset $= \emptyset$
   - for all $t \in \delta$, `getLabel`$(t) = 1$
   - for all $r = \langle q_1, v_1\rangle \hookrightarrow \langle q_2, \varepsilon\rangle$, $\delta = \delta \cup \{e\}$ and `getLabel`$(e) = f(r)$, where $e = (q_1, v_1, q_2)$
2. Iterative Procedure
   For $v_1,\ v_2,\ v_3 \in \Gamma$, $q_1,\ q_2 \in Q$, $s,\ s' \in D$, if workset $\neq \emptyset$, take and remove a transition $t = (q_2, v_2, s)$ from workset, if

- $r = \langle q_1, v_1 \rangle \hookrightarrow \langle q_2, v_2 \rangle$
  if for some $w \in \Gamma^*$, there exists $\langle q_2, v_2 w \rangle$ is accepted by A, then there exists

$$\langle q_1, v_1 w \rangle \overset{r}{\hookrightarrow} \langle q_2, v_2 w \rangle \hookrightarrow^* c, \quad \text{for some } c \in C$$

  Let $e = \{(q_1, v_1, s)\}$, and value $= f(r) \otimes \texttt{getLable}(t)$ as shown in Figure 2.7.
  - if $e \notin \delta$, $\delta = \delta \cup \{e\}$; $\texttt{getLable}(e) =$ value, and workset $=$ workset $\cup \{e\}$.
  - otherwise, let tmpval $=$ value $\oplus$ $\texttt{getLable}(e)$. If $\texttt{getLable}(e) \neq$ tmpval, $\texttt{getLable}(e) =$ value $\oplus$ $\texttt{getLable}(e)$ and workset $=$ workset $\cup \{e\}$.

  To note that, no infinite ascending chain ensures termination.

- If $r = \langle q_1, v_1 \rangle \hookrightarrow \langle q_2, v_2 v_3 \rangle$
  if for some $w \in \Gamma^*$, there exists $\langle q_2, v_2 v_3 w \rangle$ is accepted by A, then there exists

$$\langle q_1, v_1 w \rangle \overset{r}{\hookrightarrow} \langle q_2, v_2 v_3 w \rangle \hookrightarrow^* c, \quad \text{for some } c \in C$$

  In this case, $e = \{(q_1, v_1, s')\}$, and value $= f(r) \otimes \texttt{getLable}(t) \otimes \texttt{getLable}(t')$, as shown in Figure 2.8. The updates on edges and labels are the same with the first case.

- If $r = \langle q_1, v_1 \rangle \hookrightarrow \langle q_2, v_2 v_2 \rangle$
  if for some $w \in \Gamma^*$, there exists $\langle s', v_3 v_2 w \rangle$ is accepted by A, then there exists

$$\langle q_1, v_1 w \rangle \overset{r}{\hookrightarrow} \langle s', v_3 v_2 w \rangle \hookrightarrow^* c, \quad \text{for some } c \in C$$

  In this case, $e = \{(q_1, v_1, s)\}$, and value $= f(r) \otimes \texttt{getLable}(t') \otimes \texttt{getLable}(t)$, as shown in Figure 2.9. The updates on edges and labels are the same with the first case.

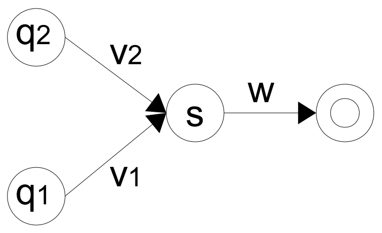


Figure 2.7: $\langle q_1, v_1 \rangle \hookrightarrow \langle q_2, v_2 \rangle$

Given the automaton A^*_{pre} constructed above, for each $c \in P \times \Gamma^*$, the value $v(c)$ is read off from the automaton by accumulating values along each accepted run by $\otimes$ operator; and all accepted runs are combined by $\oplus$ operator.

Remark 2 *It can be seen from the algorithm that, the associativity of $\otimes$ operation can also be loosened, as long as both directions are sound wrt the abstract program semantics.*

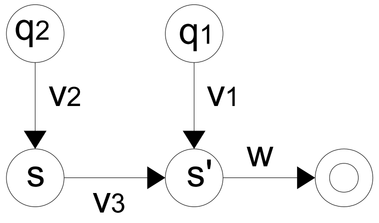


Figure 2.8: $\langle q_1, v_1 \rangle \hookrightarrow \langle q_2, v_2 v_3 \rangle$

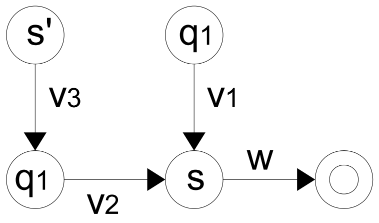


Figure 2.9: $\langle q_1, v_1 \rangle \hookrightarrow \langle s', v_3 v_2 \rangle$

2.4 Application to Interprocedural Program Analysis

2.4.1 Why Weighted Pushdown Model Checking

The model checking based approach provides an attractive systematic way to program analysis. Pushdown systems, i.e. equivalently context-free languages, naturally model procedural calls and recursions by remembering the return call sites. Thus, program analysis based on pushdown model checking is essentially *context-sensitive* and *interprocedural*. In contrast, some bound is traditionally placed on the recursion steps such as k-CFA [42].

Weighted pushdown systems extend pushdown systems with a bounded idempotent semiring. It enjoys the aforementioned advantages of pushdown systems. Whereas, the proposal of weighed pushdown model checking aims at solving the traditional *meet-over-all-valid-pathes* (MOVP) solution. By *valid*, it means calls and returns can correctly match each other along a program path. The MOVP solution propagates data flow analysis along each path and combine the results of pathes that meets some criterion.

Apart from natural context-sensitivity, weighted pushdown model checking has more preferable advantages for interprocedural program analysis.

- Pushdown model checking problem is reduced to the emptiness problem of automata. Whereas, weighted pushdown model checking provides reachability analysis from the single source to regular target and vice versa. This flexibility allows pushdown systems to be constructed on-the-fly and consequently the analysis of partial balanced parentheses. That is, the analysis can go into called procedures before the procedure call returns.

- Due to follow the known-as automata-theoretical approach, Pushdown model checking demands finite domain abstraction. Weighted pushdown model checking loosens the requirement as no infinite descending chains on the weighted domain. For instance, an interprocedural constant propagation analysis can be done by weighted pushdown model checking with a possibly infinite domain abstraction[46].

- Model checking based on weighted pushdown systems also provides program analysis with extra bonus: a witness set is provided by model checking for the analysis result. Such an "explanation" helps both program analysis and debugging.

When applied to program analysis, the intuition behind $\otimes$ and $\oplus$ of weights is:

- A weight intends a function to represent how a property is carried at each step of program execution.

- $f \otimes g$ intends the function composition $g \circ f$.

- $f \oplus g$ intends a conservative approximation at the meet of two control flows, such as a conditional sentence.

- Non-commutativity of $\otimes$ facilitates modeling a flow-sensitive analysis.

Therefore, a weighted pushdown system computes how properties are carried along each execution path and how they are combined. Naturally,

- 1 could be interpreted as properties keep unchanged by this transition step;

- 0 could be interpreted as the program execution is interrupted by some error.

Assume a Galois connection (L, α, γ, M) between the concrete domain L and the abstract domain M, and a monotone function space $\mathcal{F}_l : L \rightarrow L$. The corresponding functions $\mathcal{F}_m : M \rightarrow M$ on the abstract domain is defined as

$$\mathcal{F}_m = \{f_m \mid f_m \supseteq \alpha \circ f_l \circ \gamma,\ f_l \in \mathcal{F}_\updownarrow\}$$

$\mathcal{F}_m$ is a monotone function space, and less precise but conservative approximation can always be obtained by $\otimes$ and $\oplus$ operations. Therefore, although program analysis based on weighted pushdown model checking works on flow function spaces instead of property domains, soundness can be naturally preserved, with a systematic abstraction design.

2.4.2 An Encoding of Programs as Pushdown Systems

Abstraction is a fundamental step for program analysis based on model checking. A sound and enough compact abstraction is expected. The choice of the underlined model also matters on the problem solving. Control flow graphs (CFG) are probably the simplest transition systems of the program in terms of procedures. We follow a conventional graph representation of interprocedural CFGs (also called *supergraph* in [38]).

Definition 11 *An interprocedural control flow graph is defined as $G_f = (N_f, E_f)$, where N_f is the set of nodes associated with program statements* `Stmt`*, and $E_f \subseteq N_f \times N_f$ is the set of edges.*

Let `StmtOf`: $N_f \rightarrow S \cup \{\emptyset\}$ be a function that gets the program statement associated with a node in an interprocedural CFG. Furthermore, an unique entry node and exit node are introduced to CFGs of each procedures. A node associated with a procedural call is represented by three nodes, one for call sites, one for return points, and another for getting return values if any.

Example 5 *As shown in Figure 2.10, a node n with* `StmtOf`*(n)= "$z = x.f(y)$" will be represented by: call site n_1 with* `StmtOf`*(n_1)= "$x.f(y)$"; return point n_2 with* `StmtOf`*(n_2)= $\emptyset$; n_3 with* `StmtOf`*(n_3)= "$z =$ **`ret`**", where* **`ret`** *is a newly-introduced global variable for return values.*

$$\begin{tikzcd}
 & n_0 \arrow[d, "e_i \in E_i"'] & \\
\{x.f(y)\} & n_1 \arrow[r, dashed, "e_c \in E_c"] \arrow[d, "e_t \in E_t"'] & f\_e \arrow[dd, dashed] \\
\emptyset & n_2 \arrow[d, "e_i \in E_i"'] & \\
\{z = \texttt{ret}\} & n_3 & f\_x \arrow[ul, dashed, "e_r \in E_r"]
\end{tikzcd}$$

Figure 2.10: Edges in An Interprocedural CFG

In addition to intraprocedural edges E_i, three kinds of edges are added:

- A *call edge* E_c from the call site in the caller to the entry of the callee.
- A *return edge* E_r back from the callee to the return point in the caller.
- A *call to return edge* E_t from call site to its return point in the caller.

Therefore, $E_f = E_i \cup E_c \cup E_r \cup E_t$. To note that, there is no control flow corresponding to *call to return* edge in the real program execution. However, values of local variables keep unchanged during a procedure call, and can take a short-cut by E_t for efficiency when there are no bothers due to aliasing. There are some "*dummy edges*" in the interprocedural CFG after the above modifications, such as the edge of (n_0, n_1) that leads to the call site. All variables keep unchanged along the dummy edges.

Although weighted pushdown systems are further extended to *extended weighted pushdown systems*(EWPDS) [34], we think the the abstraction based on supergraph is enough. EWPDS excludes the *call to return edge* in the supergraph abstraction. Instead, a merge function is proposed to restore the local variables of caller procedures when callee procedures return.

Recall the usual abstractions on programs for finite model checking, the product of *global variables* and *local variables* and *program control points*(nodes of the CFG) is encoded as a state of the finite transition system. For pushdown model checking, a stack enables simulation of the runtime stack of programs. Based on the interprocedural CFG, there are various choices of encoding programs as pushdown systems. Just recall usual program executions based on stack machine, a stack can be exploited for remembering the contexts for each procedural call, i.e. the returned control point and local program states. Here we follow the convention that all program points are pushed into and popped up from the stack.

The encoding from $G_f = (N_f, E_f)$ to $P = (Q, \Gamma, \Delta, q_0, w_0)$ is: Q is a singleton set $\{\cdot\}$; $\Gamma = N_f$; w_0 is the entry node of G_f; and Δ is constructed as follows, $\forall e = (n_i, n_j) \in E_f$,

- $\langle \cdot, n_i \rangle \hookrightarrow \langle \cdot, n_j\ n_k \rangle$ if $e \in E_c$ and n_k is the return point of n_i
- $\langle \cdot, n_i \rangle \hookrightarrow \langle \cdot, n_j \rangle$ if $e \in E_i \cup E_t$
- $\langle \cdot, n_i \rangle \hookrightarrow \langle \cdot, \varepsilon \rangle$ if $e \in E_r$

Beside control flows, a usual encoding on data flows such as in Moped is: *global variables* are encoded as control states, and the product of *local variables* and *program control points* is encoded as stack symbols. After all, the problem to be analyzed decides the encoding choice.

Chapter 3

Points-to Analysis

Points-to analysis analyzes the set of dynamically allocated heap objects pointed to by variables of pointer type, such as pointers in C and reference values (often just *references* [36]) in Java. Points-to analysis is not a simple matter, especially for object-oriented languages such as Java. For instance, on some occasions, the method to be called for a method invocation depends on the actual type of the implicit parameter (i.e. the "receiver" object to which a method call is applied). That is, some method invocations are dynamically decided by the type information of object variables. It tells that the call graph generation and points-to analysis are mutually dependent and quite equivalent.

The control flow graph is basically the starting point for the static analysis of imperative and object-oriented programming languages. Furthermore, lots of analyses for Java, such as aliasing analysis, escape analysis, side-effect analysis, and safety downcast analysis, etc., require precise points-to information. Since points-to analysis is the basis of most other program analysis, it comes as our first target. This chapter is primarily dedicated to a brief introduction to points-to analysis for Java, and a discussion on the design choices that we concerned on the points-to analysis algorithms proposed in the following chapters.

This chapter is organized as follows: In Section 3.1, a brief introduction to points-to analysis for Java is given. In Section 3.2, primary difficulties in points-to analysis are presented. In Section 3.3, a detailed discussion on our design choices of designing the points-to analysis algorithms are presented. In Section 3.4, related work is given.

3.1 Points-to Analysis

In Java, an object that is allocated on the heap memory is either a class instance or an array. References are pointers to these objects, and a special *null* reference that refers to no object. The purpose of points-to analysis for Java is to analyze the set of allocated heap objects possibly pointed to by references at runtime. Since objects are pervasive for an object-oriented programming language, it is not surprising that points-to analysis is the basis of most program analysis for Java. For easy representation, we use the flowing example throughout this thesis.

Example 6 *Figure 3.1 and 3.2 presents a simple personal management with two Java classes,* `Employee` *and* `Professor`*. Class* `Professor` *inherits Class* `Employee`*, and overwrites two methods:* `getSalary()` *and* `method(Employee)`*. This example is a variant from [40], the "dummy"* `method` *is introduced to show the taste of points-to analysis.*

```
class Employee
{
   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public Employee method(Employee a)
   {
      return a;
   }
   private String name;
   private double Salary;
   private Employee f;
}
```

Figure 3.1: An Example of Personnel Management (1)

The research of points-to analysis (or *reference analysis*) for objected-oriented languages has been going on for more than ten years. Before that, points-to analysis on C has been extensively investigated. Compared with C/C++, Java has a simpler pointer syntax, e.g. no explicit dereferencing operator such as "*" and the "&" operator for fetching the address of variables. For instance, after the following code fragment, the reference `y` points to the same heap object with `x`.

```
Employee x, y;
x = new Employee("Ben", 1000);
y = x;
```

A graph representation of points-to relations, such as pointer assignment graph (PAG)[57], is the basis of most flow-insensitive points-to analysis. For instance, subset-based algorithms [47], or CFL-reachability algorithms [52] are performed on PAG-like graphs. A PAG represents the *flows-to* relation, i.e. the inverse of the *points-to* relation.

Example 7 *Figure 3.3 (a) shows an example Java code fragment, and its PAG is presented in Figure 3.3 (b). In (b), dotted edges show interprocedural points-to relations;* o_1*,* o_2*,* o_3*, and* o_4 *are abstract heap objects allocated at the allocation sites* 1, 2, 3, *and* 6 *respectively.*

```
27. class Professor extends Employee
28. {
29.    public Employee(String n, double s)
30.    {
31.       super(n, s);
32.       bonus = 0;
33.    }
34.
35.    public void setBonus(double b)
36.    {
37.       bonus = b;
38.    }
39.
40.    public double getSalary()
41.    {
42.       double base = super.getSalary();
43.       return (base + bonus);
44.    }
45.
46.    public Employee method(Employee a)
47.    {
48.        return a.f;
49.    }
50.    private double bonus;
51. }
```

Figure 3.2: An Example of Personnel Management (2)

In this example, a virtual method call occurs at line 5. Due to polymorphism, the receiver object of this invocation can be instances of either Class `Employee` or Class `Professor`. For instance, the following assignment is valid. Thus, the exact method to be called will be postponed at runtime.

```
Employee x;
x = new Professor("Ben", 2000);
```

In (b), a conservative result is given. That is, both `Employee.method` and `Professor.method` are considered to be called. A flow-sensitive analysis can figure out that the actual type of `x` is o_2. Thus, an invalid path to `Employee.method` from `x` can be eliminated. Based on the calling structures resolved, it can be inferred that `z` points to o_3. This example clearly shows, points-to analysis needs a *call graph* which represents calling relation among methods in a program; on the other hand, the *call graph* construction depends on the results of points-to analysis.

3.2 Difficulties

As mentioned above, call graph construction and points-to analysis are mutually dependent and quite equivalent. Program analysis for Java becomes difficult due to this inter-

```
  public static void main(String[] args)
  {
     Employee x;
     Professor y;
1.   x = new Employee("Ben", 2000);
2.   y = new Professor("Jean", 5000);
3.   y.f = new Employee("Mike", 1000);
4.   x = y;
     if(true)
     {
5.      Employee z = x.method(y);
     }
     else
     {
6.      x.f = new Employee("Tom", 1000);
7.      Employee v = y.method(x);
     }
  }
```

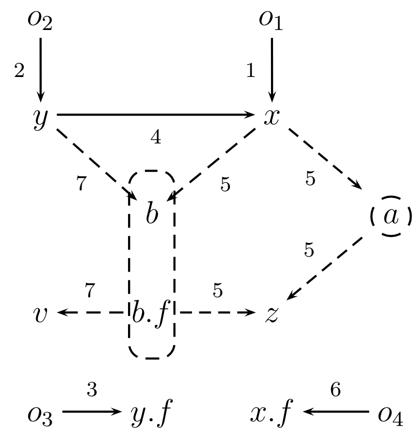


Figure 3.3: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

dependency. To construct a precise control flow for Java, the primary issue is to handle *dynamic binding*. That is, due to parametric *polymorphism*, an object variable can refer to objects of multiple actual types. That is, a reference variable can point to heap objects of its subtype in addition to those of its declared type. Since Java enforces declared type strategy, which method is to be called in some method invocations is postponed to the run-time.

To understand the later formalizations and analysis algorithms, a brief explanation on Java dynamic binding is presented here [40]. Given a method call `x.f(arg)`

1. First, the compiler checks the declared type `DeclaredType` of the implicit parameter `x`, and collects all methods called `f` in the class `DeclaredType` and all public methods called `f` in the super-classes of `DeclaredType`. These methods are all possible candidates to be called.
2. Then, the compiler checks the types of all parameters, i.e. `arg` here. If the compiler cannot find any method with matching parameters or there are multiple matches. A compiler time error will be reported.
3. After that, if the method is declared as private, static, final, or a constructor, it can be determined statically. Otherwise, the method to be called is dispatched at run-time.
4. A dynamic binding happens at run-time when the actual type `ActualType` of the object that `x` points to is clear. The Java virtual machine will look for the method `f(arg)` in the class `ActualType`. If `ActualType` does not redefine this method (namely not directly inherit), the super class of `ActualType` will be searched, and so on.

Besides, other difficulties of points-to analysis come from:

- The number of nestings of array structures, of method invocations, and of field access can be unbounded.
  For instance, the components of an array can again be arrays, the field access can be in the form of "$x.f_1.f_2...f_n$" syntactically.

- The number of heap objects allocated can be unbound.
  For instance, a heap allocation is within a while loop, or a method containing a heap allocation is recursively called from different contexts.

- Due to aliasing, fields could be changed dynamically and implicitly.
  For instance, consider a program fragment

  ```
  y = o1;
  y.f = o2;
  x = y;
  x.f = o3;
  ```

  After this fragment, `y.f` points to `o3` (not `o2`) by the side effect of `x.f = o3`. Such side effect is caused by the aliasing of `x = y`.

3.3 Design Choices

Our aim is to propose context-sensitive points-to analysis algorithms in the framework of weighted pushdown systems. First of all, pushdown systems provide us with handy context-sensitivity in two respects:

- Method calls and returns are guaranteed to be correctly paired;

- Calling contexts are naturally represented as regular pushdown stores. That is, no bound is placed on the depths of recursions and nested calls.

Under such a choice, three are still primary dimensions to be examined on both model construction and algorithm design.

- An *exploded supergraph* [37, 38] based model construction vs An *interprocedural CFG* based model design
  These are two typical choices of underlying models for program analysis. An exploded supergraph represents dataflow among variables as a graph [37, 38]. This enables us a compact model encoding as a weighted pushdown system with a moderate loss of precision. The control flow graph based model construction is conventional and straightforward, but the analysis on data flow facts need to be captured by the weight space design, which may result in relatively heavy weight space in the weighted pushdown system.

- *On-the-fly* vs *Ahead-of-time* call graph construction
  To solve the mutual dependence between points-to analysis and call graph generation, there are basically two approaches.

  – By on-the-fly call graph construction, we mean that call edges are resolved once possible, when points-to analysis proceeds. Thus, on-the-fly call graph construction computes the *least* solutions, i.e., starting with the empty call graph, a call edge is added if points-to analysis detects its possibility.

    This essentially requires repeated applications of model checking on partial models. Note that this procedure must be continued until stabilized; otherwise, soundness will not be guaranteed. (Section 4.1.2, Section 5.1.2.)

  – By ahead-of-time call graph construction, we mean that an approximated call graph is computed in advance; and the result of points-to analysis on it further refines the set of call edges. This procedure proceeds until the results stabilize. Thus, ahead-of-time call graph construction computes the *greatest* solutions, i.e., starting with an imprecise call graph (e.g. generated by a syntactical analysis only), a call edge is eliminated if points-to analysis refutes it.

    This would also require repeated applications of model checking on models, but how many times one repeats is up to the choice of trade-off between precision and efficiency, because soundness is guaranteed by nature. (Section 4.1.3, Section 5.1.3)

  Based on empirical studies, the on-the-fly manner is regarded to be more precise than the ahead-of-time manner, except more iterations are usually needed.

Apart from the above primary dimensions, we also have choices of *flow sensitivity* and *field sensitivity.*

- flow-sensitive vs flow-insensitive
  A flow-sensitive analysis takes into account the execution order of program codes; otherwise, the analysis is flow-insensitive. Flow sensitivity is a rather design choice under the tradeoff between precision and efficiency. One smart idea is combining SSA (Static Single Assignment) and complete flow insensitivity [43]; this reduces each method to a single control point with the cost of larger number of variables.

  We do not take this approach; instead the scope management of variables lies in the weight design. We also design how to introduce parameterized field insensitivity: the model reduction (Section 4.2) and the weight simplification (Section 5.2).

- field-sensitive vs field-insensitive
  Field-sensitivity mainly talks about how instance fields are abstracted in the points-to analysis. A field-sensitive analysis distinguishes the instance fields of objects and traces field access to get precise points-to information. The issue of field sensitivity is rather on an abstraction level. Since field access in Java can unboundedly nest, we need to give a bound to it, similarly to an abstraction on arrays.

A taxonomy of dimensions in points-to analyses for objected-oriented programming languages are presented in [39]. There are still some dimensions we have not mentioned, such as *object representation*, *reference representation*, and *directionality.*

Since theses concerned design choices are independent of the use of model checking framework, their effect on the analysis design will not be explored. The choices on them is rather a tradeoff among efficiency and precision. Our abstraction strategy on them is as precise as possible.

3.4 Related Work

Over the last decade, points-to analysis is an active research field due to its crucial role in program analysis and compiler optimizations.

One of the pioneer work in this field is Andersen's points-to analysis [47] for C. It is a subset-based, flow-insensitive analysis implemented via constraint solving, such that object allocations and pointer assignments are described by subset constraints, such as $x = y \Rightarrow pta(y) \subseteq pta(x)$. The scalability of Andersen's analysis has been greatly improved by more efficient constraint solvers [48, 49]. Andersen's analysis was introduced into Java by using annotated constraints[50].

Reps presents a points-to analysis algorithm for C [38], by formulating pointer assignments as productions of context-free grammars. Borrowing this view, Andersen's analysis is further formulated as a CFL-reachability problem for Java [51]. The key insight is that the field access in Java is formulated as a balanced-parentheses problem. A refined context-sensitive points-to analysis is proposed later by excluding invalid paths with CFL-reachability[52] also. The demand-driven strategy makes this analysis scale.

A scalable context-sensitive points-to analysis for Java is presented in [54]. Programs, as well as analyses, are abstracted and encoded as the set of rules in logic programming. Their context-sensitivity is obtained by cloning a method for each calling context, by regarding loops as equivalent classes. The analysis under such a construction should be out of control. However, the implementation based on BDD makes the analysis scale.

Spark is an influencing framework for experimenting with points-to analysis for Java. It supports both equality and subset-based analysis; provides various algorithms for call graph construction, e.g., CHA[55], RTA[56], and on-the-fly algorithm; enables variations on field-sensitivity, etc. The BDD-based implementation of the subset-based algorithm further improves the efficiency of operations on points-to sets[58]. It is the testbed for many points-to analysis for Java. Our analysis also borrow its facility for generating call graph in an ahead-of-time analysis.

Apart from CFL-reachability based context-sensitivity, Milanova et. al [44] explore another approach to context-sensitivity based on so-called *object-sensitivity*. The key is a method call is distinguished by a sequence of distinct receiver objects, rather than call strings. However, similar to call strings based approach, the sequence of receiver objects can be unbounded. Some bound is placed similar to the occasion of k-CFA.

A recent empirical study on the effect of context-sensitivity is presented [45], which evaluate the precision of subset-based points-to analysis with an eye on various abstractions wrt context-sensitivity. Their study shows that object-sensitivity based analysis is likely to perform better than the call strings based and the BDD-based context-sensitive [58] analysis. In particular, it shows that a context-sensitive analysis is crucial due to the rich cycles in Java call graphs.

Barbara [39] also gives a taxonomy of dimensions that concerned in the points-to analysis over nearly a decade.

Chapter 4

Context-sensitive Points-to Analysis based on Exploded Supergraph

Since points-to analysis is the basis of interprocedural program analysis for Java, it comes as our first analysis target. This chapter, as well as the next chapter 5, is dedicated to designing context-sensitive points-to analysis algorithms for Java based on weighted pushdown model checking, following to the methodology that *Program analysis is abstraction plus model checking.* We aim at exploring the following questions:

- Whether a crucial and tough program analysis like points-to analysis can be solved based on model checking ?
- Points-to analysis is an extensively-explored topic. Whether the primary design choices concerned in the traditional approaches can be solved as model checking problems ?
- How the model checking based approach benefits points-to analysis compared with traditional ones ?

First of all, pushdown systems provides us with handy context-sensitivity. To solve the mutual dependency among call graph construction and points-to analysis, both on-the-fly and ahead-of-time analysis are investigated. Both *an exploded supergraph* [37, 38] based (Chapter 4) and *an interprocedural control flow graph* based (Chapter 5) approaches are explored and compared. Our study shows the CFG based model design provides facilities for the algorithm construction.

For other difficulties from various infinity (Section 3.2), our tentative solutions are:

- An array is abstracted to a single heap allocation, (i.e., difference among indices in an array are ignored, but difference among the arrays are distinguished);
- A bound is set on field nesting in the ahead-of-time call graph constructions. That is, access to a nested field beyond the bound will be treated as access to any heap allocation;
- The context-insensitive heap abstraction is adopted, i.e. an abstract heap object is associated with its allocation site.
- To handle aliasing and delicate Java semantics, an abstraction obeying to Java operational semantics is carefully designed (Section 4.1.2 and 5.1.1).

Flow-sensitivity is a choice on the tradeoff among precision and efficiency. We explore a relatively unexplored topic of *parametrization*. In our algorithms, parameterized flow-sensitivity can be naturally obtained by either model reduction (Section 4.2) or simplifying the weight space design (Section 5.2).

This chapter is organized as follows:

In Section 4.1.1, an abstraction on Java heap memory by exactly following to Java semantics is given. In Section 4.1.2, a context/field/flow-sensitive points-to analysis algorithm is proposed, based on the exploded supergraph model design and on-the-fly algorithm construction. In section 4.1.3, another context/field/flow-sensitive points-to analysis algorithm is proposed, based on the exploded supergraph model design and ahead-of-time algorithm construction.

In Section 4.2, we further investigate the relatively unexplored parametrization problem. By applying model reduction, parameterized flow-sensitivity is naturally obtained on the algorithms proposed in Section 4.1.

In Section 4.3, a prototype implementation is proposed, and some preliminary evaluation is given. The algorithms and implementation aspects of our analysis are also presented and discussed finally.

4.1 Exploded Supergraph Based Model Design

In this section, we propose context-sensitive points-to analysis algorithms based on an exploded supergraph model design. Call graph generation (i.e. model construction for model checking) and points-to analysis (i.e. model checking) are mutually dependent. Both the on-the-fly (Section 4.1.2) and ahead-of-time (Section 4.1.3) model construction are investigated.

4.1.1 An Abstraction of Java Heap Memory

A heap object in Java is either a class instance or an array. These objects are allocated in the heap memory. Each heap object has a corresponding type and allocation site, which can be decided syntactically.

Definition 12 *Let $\mathcal{V}$ be a set of abstract reference variables and $\mathcal{O}$ be a set of abstract heap objects respectively. A transitive and reflexive points-to relation is defined as* $\mapsto$: $\mathcal{V} \times (\mathcal{V} \cup \mathcal{O})$.

Let $\mathcal{F}$ be a set of field names and $\mathcal{L}$ be a set of local variables. An instance field is referred in the form of $l.f(l \in \mathcal{L}, \{ \in \mathcal{F})$ syntactically.

Definition 13 *Let $\mathbb{C}$ be a set of program calling contexts and $\mathcal{P}$ be the powerset operator, a context-sensitive points-to analysis is defined as* $\text{pta} : \mathcal{V} \times \mathbb{C} \to \mathcal{P}(\mathcal{O})$.

Definition 14 *A field sensitive analysis abstracts an instance field $l.f$ as pairs of* $\{(o, f) \mid o \in \text{pta}(l, \text{context}),\ \text{context} \in \mathbb{C}\}$. *A field-based (resp. field-insensitive) analysis ignores the first component (resp. the second component).*

Definition 15 *A pointer assignment graph $G_a = (N_a, E_a)$ is a graph where $N_a = \mathcal{V} \cup \mathcal{O}$ and $E_a = \rightsquigarrow = \mapsto^{-1}$.*

Let $\mathscr{O}$ be a set of run-time objects allocated in the heap memory. Let $\mathcal{T}$ be a finite set of types (class names) of heap objects. Let Loc be a finite set of memory allocation sites in the program. Let $\eta_\tau : \mathscr{O} \to \mathcal{T}$ and $\eta_\iota : \mathscr{O} \to \text{Loc}$ be functions that get the corresponding type and allocation site of a heap object respectively.

Definition 16 *Let $\mathcal{V}_l$ be the set of local variables, let $\mathcal{V}_s$ be the set of static fields. The set of abstract reference variables is* $\text{RefVar} = \mathcal{V}_l \cup \mathcal{V}_s \cup \{\mathit{arg}_k, \mathit{this}, \mathit{ret} \mid k \in \mathbb{N}\}$*, and the set of reference fields* $\text{RefField} = \mathcal{O} \times \mathcal{F}^+$*. Let* $\mathcal{V}_{ref} = \text{RefVar} \cup \text{RefField}$.

$\mathcal{V}_{ref}$ defines the concrete kinds of references that we are interested in during the following analysis. By *reference fields*, we mean fields of reference type. $\mathcal{O} \times \mathcal{F}^+$ represents the set of reference fields (that could be nested) with a precise field-sensitive abstraction. The new variables `ret` and $\texttt{arg}_k (k \in \mathbb{N})$ are introduced to model return values and parameters passed in a method invocation respectively. Another new variable `this` is introduced to model the keyword **this** in Java. For instance, it may denote a reference to the object which an instance method call was applied to.

Definition 17 *Let $\mathcal{O} \subseteq \mathcal{T} \times \mathcal{L} \cup \{\diamond, \top\}$ be a set of abstract heap objects, where $\diamond$ represents null reference that can also serve to represent uninitialized, $\top$ represents a "generic" heap object whose type and allocation site can be any. An abstraction on $\mathscr{O}$ is defined as $\alpha : \mathscr{O} \to \mathcal{O}$, such that*

$$\forall o \in \mathscr{O},\ \alpha(o) = (\eta_\tau(o), \eta_\iota(o))$$

In particular, it is required that

$$\forall (\tau_i, \iota_i), (\tau_j, \iota_j) \in \mathcal{O},\ \iota_i = \iota_j \Rightarrow \tau_i = \tau_j$$

The above abstraction says, the same heap memory allocation site in the program results in an unique abstract heap object. At the current stage, we approximate an array with a single array element to be a representative.

Let $\texttt{type} : \mathcal{O} \to \mathcal{T}$ be the function that gets the type of an abstract object. Let `any` be the corresponding type of $\top$, and `none` be the corresponding type of null reference, i.e.

$$\texttt{any} = \texttt{type}(\top)$$

$$\texttt{none} = \texttt{type}(\diamond)$$

Program analyses based on weighted pushdown model checking examine transfer functions on the data domain of programs. For points-to analysis, our focus is transfer functions on heap environments. As will be seen in Chapter 5, a weight space is delicately designed based on transfer functions on abstract heap environments.

In this section, a model construction based on the exploded supergraph is proposed. By exploding the heap environment separately for each reference variable, a weight space is designed based on the changes of data flows. Thus the data domain of interest is a powerset construction on heap objects as follows.

Definition 18 *A Galois connection $(\mathcal{P}(\mathscr{O}), \alpha, \mathcal{P}(\mathcal{O}), \gamma)$ is defined between the two complete lattices $(\mathcal{P}(\mathscr{O}), \subseteq)$ and $(\mathcal{P}(\mathcal{O}), \subseteq)$ with $\alpha : \mathcal{P}(\mathscr{O}) \to \mathcal{P}(\mathcal{O})$ and $\gamma : \mathcal{P}(\mathcal{O}) \to \mathcal{P}(\mathscr{O})$ such that*

$$\alpha(\emptyset) = \gamma(\emptyset) = \emptyset$$

$$\forall \text{SO} \neq \emptyset \subseteq \mathscr{O},\ \alpha(\text{SO}) = \{\tilde{\alpha}(o) \mid o \in \text{SO}\}$$

$$\forall \text{SC} \neq \emptyset \subseteq \mathcal{O},\ \gamma(\text{SC}) = \{o \in \mathscr{O} \mid \tilde{\alpha}(o) \in \text{SC}\}$$

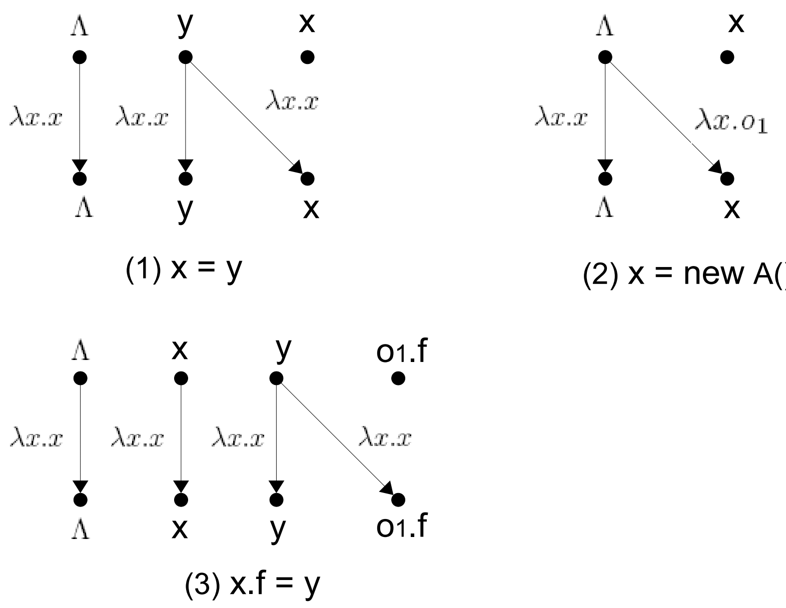


Table 4.1: A Point-wise Representation of Pointer Assignments

4.1.2 On-the-fly call graph construction

A straightforward approach to points-to analysis is a reachability analysis on data (i.e. references) flows. For efficiency, a variation of "exploded supergraph"[38], named a *flow-sensitive flows-to graph*, is used as the underlined model for model checking. The exploded supergraph represents data flows by the explicit product of control flows.

The key of the exploded supergraph construction is a point-wise representation of transfer functions on the data flow "facts" of interest [38]. Table 4.1 shows such a basic point-wise representation of pointer assignments in our case, with the assumption that an abstract heap object o_1 is associated with the allocation in the case (2), and the x points to o_1 for the case (3). Λ could be interpreted as a heap environment that allocates new heap objects.

Let $\mathcal{A}[\![\cdot]\!] : \text{Stmt} \rightarrow \mathcal{P}(\rightsquigarrow)$ be the abstraction function on program statements Stmt, as defined in Table 4.2, where `context` $\in \mathbb{C}$ is the program calling context wrt the analysis point. As is shown in Table 4.2, a points-to analysis is performed for a field-sensitive abstraction whenever a field access is encountered. Labeled functions show the changes of data flow wrt references on the ends of the edge.

Definition 19 *Let $G_f = (N_f, E_f)$ be an interprocedural CFG. Let $N_p = (\{\Lambda\} \cup \mathcal{V}_{\mathcal{O}}) \times \mathcal{N}_{\{}$, let $L_p = \{\lambda x.x\} \cup \{\lambda x.o \mid o \in \mathcal{O}\}$, and $E_p = N_p \times L_p \times N_p$. A flow-sensitive flows-to graph $G_p = (N_p, E_p, L_p)$ is constructed as follows.*

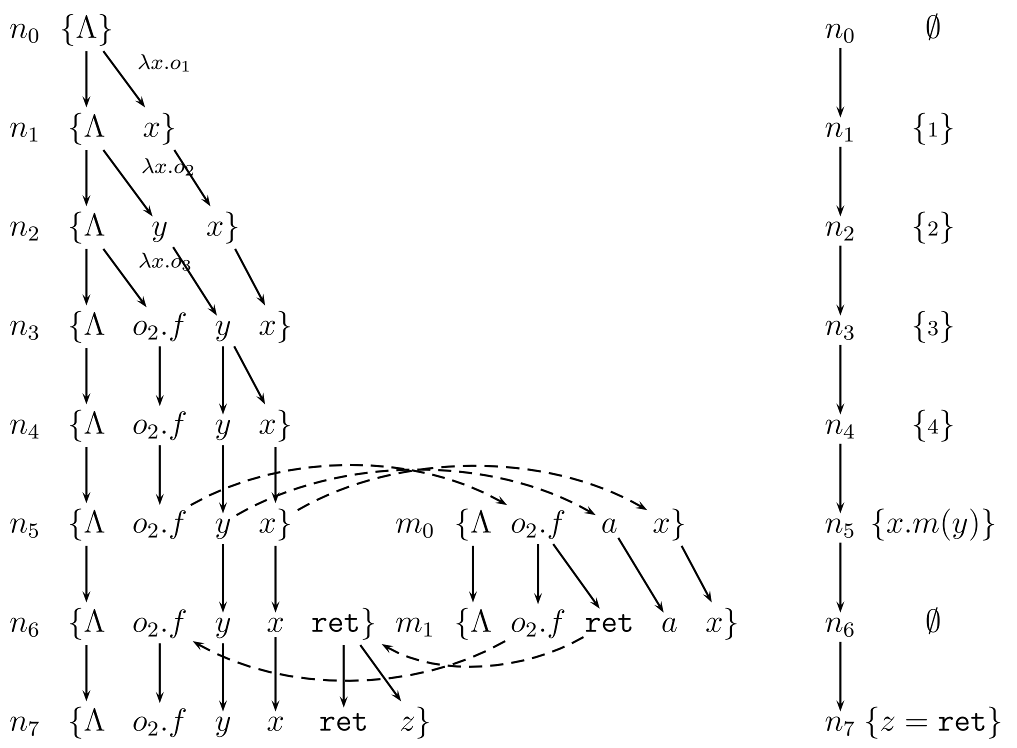
Table 4.2: $\mathcal{A}[\![\cdot]\!] : \text{Stmt} \rightarrow \mathcal{P}(\rightsquigarrow)$

Jimple Syntax	Flows-to Relation
$x = \text{new T()}$	$\{x \mapsto o\}^{-1}$, where $o \in \mathcal{O}$ is a fresh abstract heap object and $\texttt{type}(o) = \text{T}$
$x = \text{newarray T()}$	$\{x \mapsto o\}^{-1}$, where $o \in \mathcal{O}$ is a fresh abstract heap object and $\texttt{type}(o) = \text{T}$
$x = \mathit{null}$	$\{x \mapsto \diamond\}^{-1}$
$x = y$	$\{x \mapsto y\}^{-1}$
$x = y.f$	$\{x \mapsto o.f \mid o \in \texttt{pta}(y, \texttt{context})\}^{-1}$
$y.f = x$	$\{o.f \mapsto x \mid o \in \texttt{pta}(y, \texttt{context})\}^{-1}$
return x	$\{\texttt{ret} \mapsto v\}^{-1}$, where x is a reference variable
$x.f(m_1, ..., m_l, m_{l+1}, ..., m_n)$	$\{\texttt{arg}_1 \mapsto m_1, ..., \texttt{arg}_l \mapsto m_l, \texttt{this} \mapsto x\}^{-1}$, where $m_i (1 \leq i \leq l)$ are reference variables, $m_j (l+1 \leq j \leq n)$ are variables of primitive type
$f(m_1, ..., m_l, m_{l+1}, ..., m_n)$	$\{\texttt{arg}_1 \mapsto m_1, ..., \texttt{arg}_l \mapsto m_l, \}^{-1}$, where $m_i (1 \leq i \leq l)$ are reference variables, $m_j (l+1 \leq j \leq n)$ are variables of primitive type
$z = \texttt{ret}$	$\{z \mapsto \texttt{ret}\}^{-1}$
x := @this: T	$\{x \mapsto \texttt{this}\}^{-1}$
x := @parameter$_k$: T	$\{x \mapsto \texttt{arg}_k\}^{-1}$

$E_p = \bigcup_{e_f \in E_f} E_{p,e_f}$, *where for each* $e_f = (n_1, n_2) \in E_f$

$$E_{p,e_f} = \begin{cases} \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in \text{StableSet} \cup \{\Lambda\}\} & \textit{if } e_f \in E_i \\ \cup\ \{((v_1, n_1), \lambda x.x, (v_2, n_2)) \mid (v_1, v_2) \in \text{FlowsTo},\ v_1 \in \mathcal{V}_{\mathcal{O}}\} & \\ \cup\ \{((\Lambda, n_1), \lambda x.o, (v, n_2)) \mid (o, v) \in \text{FlowsTo}, o \in \mathcal{O}\} & \\ \textit{where } \text{FlowsTo} = \mathcal{A}[\![\texttt{StmtOf}(n_2)]\!] & \\ \qquad \text{StableSet} = \mathcal{V}_{\mathcal{O}} - \{v \mid (v', v) \in \text{FlowsTo}\} & \\ & \\ \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in \mathcal{V}_l\} & \textit{if } e_f \in E_t \\ & \\ \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in \mathcal{V}_{ref} \setminus \mathcal{V}_l\} & \textit{if } e_f \in E_c \\ \cup\ \{((v, n_1), \lambda x.x, (v', n_2)) \mid (v, v') \in \text{FlowsTo}\} & \\ \textit{where } \text{FlowsTo} = \mathcal{A}[\![\texttt{StmtOf}(n_1)]\!] & \\ & \\ \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in \mathcal{V}_{ref} \setminus \mathcal{V}_l\} & \textit{if } e_f \in E_r \\ & \\ \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in \mathcal{V}_{ref}\} & \textit{if } e_f \textit{ is dummy} \end{cases}$$

Based on the on-the-fly call graph construction, G_p is also constructed on-the-fly whenever one edge in G_f is known. For efficiency, only the set of reference variables, which appear in the program up to this point, contributes to the construction of G_p. The edge of $(v_1, \lambda x.x, v_2)$ intends that, points-to sets of v_2 are the same with that of v_1; and the edge of $(\Lambda, \lambda x.c, v)$ intends that v points to c.



n_0 $\{\Lambda\}$
$\lambda x.o_1$
n_1 $\{\Lambda \quad x\}$
$\lambda x.o_2$
n_2 $\{\Lambda \quad y \quad x\}$
$\lambda x.o_3$
n_3 $\{\Lambda \quad o_2.f \quad y \quad x\}$
n_4 $\{\Lambda \quad o_2.f \quad y \quad x\}$
n_5 $\{\Lambda \quad o_2.f \quad y \quad x\}$
m_0 $\{\Lambda \quad o_2.f \quad a \quad x\}$
n_6 $\{\Lambda \quad o_2.f \quad y \quad x \quad \texttt{ret}\}$
m_1 $\{\Lambda \quad o_2.f \quad \texttt{ret} \quad a \quad x\}$
n_7 $\{\Lambda \quad o_2.f \quad y \quad x \quad \texttt{ret} \quad z\}$
n_0 $\emptyset$
n_1 $\{1\}$
n_2 $\{2\}$
n_3 $\{3\}$
n_4 $\{4\}$
n_5 $\{x.m(y)\}$
n_6 $\emptyset$
n_7 $\{z = \texttt{ret}\}$

Figure 4.1: (a) Part of the G_p *w.r.t* (b) (b) One Run in G_f of Figure 1 (a)

Example 8 *Figure 4.1 shows part of the G_p of the example in Figure 3.3 (a). It corresponds to one possible run of the program along line numbers of 1-2-3-4-5. Dotted edges are used to stress the interprocedural flows-to relations. Abstract reference variables enclosed by {} belong to one node of G_f, within which there are no transitions inside. For simplicity, edges without labels in the figure are labeled with $\lambda x.x$. By a pushdown reachability analysis on G_p, we know z points to o_3.*

Let $\mathcal{S} = \mathcal{P}(\mathcal{O})$, define

$$D_1 = \{\lambda x.s \ \mid s \in \mathcal{S}\} \text{ and } D_2 = \{\lambda x.x \cup s \ \mid s \in \mathcal{S}\}$$

Therefore $\lambda x.\emptyset \in D_1$ and $\lambda x.x = \lambda x.x \cup \emptyset \in D_2$. D_1 intends that some reference variable points to some set of abstract heap objects. D_2 intends that an abstract reference variable may keep unchanged along one path and be changed along another.

Definition 20 *The composition of $d_1, d_2 \in (D_1 \cup D_2)$ is defined by the standard η-expansion as follows:*

$$d_2 \circ d_1 =_\eta \lambda x.\ d_2 \circ d_1 x = \lambda x.\ d_2(d_1(x))$$

Definition 21 *A bounded idempotent semiring $S = (D, \oplus, \otimes, 0, 1)$ is defined as*

- *The weight space $D = D_1 \cup D_2$*
- ***1** is defined as $\lambda x.x$*
- ***0** is defined as $\lambda x.\emptyset$*

Weighted PDS	G_p
control states	$\{\Lambda\} \cup \mathcal{V}_{ref}$
stack alphabet	N_f
pushdown transitions	E_p
weight space	D

Table 4.3: The Encoding of Weighted PDS from G_p

- *The $\otimes$ operator is defined as*

$$d_1 \otimes d_2 = d_2 \circ d_1 \textit{ for } d_1, d_2 \in D \setminus \{\boldsymbol{0}\}$$

- *The $\oplus$ operator equals set union $\cup$, defined as*

$$\begin{array}{ll} d_1 \oplus d_2 = \lambda x.s_1 \cup s_2 & \textit{for } d_1 = \lambda x.s_1,\ d_2 = \lambda x.s_2 \in D_1 \\ d_1 \oplus d_2 = \lambda x.x \cup s_1 \cup s_2 & \textit{for } d_1 = \lambda x.s_1 \in D_1,\ d_2 = \lambda x.x \cup s_2 \in D_2 \\ d_1 \oplus d_2 = \lambda x.x \cup s_1 \cup s_2 & \textit{for } d_1 = \lambda x.x \cup s_1,\ d_2 = \lambda x.x \cup s_2 \in D_2 \end{array}$$

Hitherto, points-to analysis can be solved as the pushdown reachability analysis on G_p. A weighted PDS is encoded from G_p as shown in Table 4.3.

4.1.3 Ahead-of-time call graph construction

For the points-to analysis based on an on-the-fly call graph construction, there are two occasions on which model checking need to be dispatched:

- Virtual method call resolutions; and
- A precise abstraction on fields (i.e. field-sensitivity with handling on aliasing).

As will be seen in Section 4.3.2, the number of model checking requests during an on-the-fly analysis is overwhelming, which affects the efficiency a lot. To avoid such frequent model checking requests, an alternative is a points-to analysis algorithm based on an ahead-of-time call graph construction.

Provided with the machinery of weighted pushdown model checking, we expect

- Some invalid paths (i.e. pushdown transitions or control flows) could be eliminated within the points-to analysis (i.e. model checking) procedure;
- Fields could also be analyzed during points-to analysis with proper treatment on aliasing,

rather than judge the path removal and clarify the aliasing with extra efforts, by frequently interrupting the points-to analysis. Unfortunately, the features of exploded supergraph prohibits such preferable analysis, in that

- The effect of an edge removal in an exploded supergraph is limited to the data flow of specific variables. i.e., only the approximated call edge wrt the method receivers (i.e. implicit parameters of an instance invocation) can be removed. The data flow wrt other variables rather than method receivers can not be prevented from reaching the destination state.

- When the field sensitivity is concerned, an exploded supergraph wrt a field access $x.f$ can be constructed only after aliasing on x is clarified.

In view of this, our choice is a two-phase analysis, which is a compromise between design choices (i.e. efforts on possibly reducing cost) and precision. The first step is a points-to analysis with the approximation that each field points to any abstract heap allocations, i.e., $x.f = \top$. That is, the field access is ignored for a while. Our effort is dedicated to exploring the possibility of automatic path removal. In Chapter 5, we will present a new analysis algorithm with treatments on both invalid path removal and aliasing, enabled by an interprocedural CFG based model design. The second step is an exclusive analysis on fields with 1-depth nesting.

The key of automatic path removal is:

> *A call edge will be removed if no edges reaching this invocation site possibly satisfy the type constraints on the receiver object that it declares.*

This idea is formalized by extending the data flow facts with type constraints on the receiver objects required by a virtual call edge. Initially, the type constraint on a virtual call edge is a singleton set; and no type constraints are placed on other kinds of edges. We will extend weights in Definition 21 by pairing a set of type constraints. Type constraints intend the expected type of the receiver object for a virtual method invocation. An invalid path is excluded from the analysis result when the current data flow facts conflicting with the type constraints.

Definition 22 *For $t, t' \in \mathcal{T} \setminus \{\texttt{any}, \texttt{none}\}$, t'* conflicts with *t if and only if*

- *$t' \neq t$, and*
- *either t' does not inherit from t, or t' inherits from t but t' redefines the method to be invoked.*

Otherwise, we say t' satisfies with *t. Furthermore,*

- *t satisfies with* `any`, *for each t in $\mathcal{T}$;*
- `none` *conflicts with t, for each t in $\mathcal{T}$.*

Definition 22 defines the relation among types. Definition 23 defines the function that returns the set of objects currently known to satisfy with type constraints. Correspondingly, Definition 24 defines the function that returns the new weight function after filtering unsatisfied objects.

Definition 23 *Define a function* `Filter` $: (D_1 \cup D_2) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{O})$, *such that for each $d \in (D_1 \cup D_2)$, $c \in \mathcal{P}(\mathcal{T})$,*

$$\texttt{Filter}(d, c) = \{o \in \texttt{body}(d) \mid \exists t \in c,\ t' = \texttt{type}(o),\ s.t.\ \ t' \textit{satisfies with } t\}$$

where $\texttt{body}(d) = s$ *for* $d = \lambda x.s \in D_1$ *and* $d = \lambda x.x \cup s \in D_2$.

Definition 24 *Define a function* $\texttt{FilteredReceiver} : (D_1 \cup D_2) \times \times \mathcal{P}(\mathcal{T}) \to (\mathcal{D}_\infty \cup \mathcal{D}_\in)$, *for each* $d \in (D_1 \cup D_2)$, $c \in \mathcal{P}(\mathcal{T})$,

$$\texttt{FilteredReceiver}(d, c) = \begin{cases} \lambda x.s & \textit{if } d \in D_1 \\ \lambda x.x \cup s & \textit{if } d \in D_2 \end{cases}$$

where $s = \texttt{Filter}(d, c)$.

Definition 25 defines the function that returns the set of type constraints that have already known to be satisfied by the current data flow facts. In contrast, Definition 26 defines the function that returns the set of type constraints that have already known not to satisfied by the current data flow facts. In particular, if $d \in D_2$, both $\textit{\texttt{KnownNotSatisfy}}(d, c)$ and $\textit{\texttt{KnownSatisfy}}(d, c)$ are pending, because there exist other data flows reaching d.

Definition 25 *Define a function* $\texttt{KnownSatisfy} : (D_1 \cup D_2) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T})$, *such that for each* $d \in (D_1 \cup D_2)$, $c \in \mathcal{P}(\mathcal{T})$,

$$\texttt{KnownSatisfy}(d, c) = \begin{cases} \emptyset & \textit{if } d \in D_2 \\ \mathbb{S} & \textit{if } d \in D_1 \end{cases}$$

where

$$\mathbb{S} = \{t \in c \mid \exists o \in \texttt{body}(d),\ t' = \texttt{type}(o),\ \textit{s.t. } t' \textit{satisfies with } t\}$$

Definition 26 *Define a function* $\texttt{KnownNotSatisfy} : (D_1 \cup D_2) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T})$, *such that for each* $d \in (D_1 \cup D_2)$, $c \in \mathcal{P}(\mathcal{T})$,

$$\texttt{KnownNotSatisfy}(d, c) = \begin{cases} \emptyset & \textit{if } d \in D_2 \\ c \setminus \texttt{KnownSatisfy}(d, c) & \textit{if } d \in D_1 \end{cases}$$

Definition 27 *Define* $\cup_t : \mathcal{P}(\mathcal{T}) \times \mathcal{P}(\mathcal{T}) \to \mathcal{P}(\mathcal{T})$ *such that for each* $T_1, T_2 \subseteq \mathcal{T}$,

$$T_1 \cup_t T_2 = \bigcup_{t_1 \in T_1, t_2 \in T_2} \{t_1 \sqcup_t t_2\}$$

where

$$t_1 \sqcup_t t_2 = \begin{cases} t_1 & \textit{if } t_1 \textit{ satisfies with } t_2 \\ \{t_1\} \cup \{t_2\} & \textit{otherwise} \end{cases}$$

Definition 28 presents a new weight space with automatic path removal. The weight domain in Definition 21 is extended by pairing a set of expected types of receiver objects. The initial exploded supergraph on $\mathcal{V}_\mathcal{O} \setminus (\mathcal{O} \times \mathcal{F}^+)$ is constructed by Definition 19, provided with a pre-computed call graph based on CHA (Class Hierarchy Analysis) [55]. Compared with Table 4.2, the only difference of abstraction on the program (i.e. CFG) is, provided with an approximated call graph,

- The weight function of an approximated virtual call edge is paired with a singleton set that consists of the assumed type of the receiver object.
- The weight function of other kinds of edges is paired with a singleton set $\{\texttt{any}\}$.

Definition 28 *We borrow the notation from Definition 21. A bounded idempotent semiring $S_a = (D_a, \oplus_a, \otimes_a, 0_a, 1_a)$ is defined as*

- *The weight space $D_a = D \times C$, and $C = P(\mathcal{T})$.*
- ***1** is defined as $(\lambda x.x, \{\texttt{any}\})$*
- ***0** is defined as $(\lambda x.\emptyset, \emptyset\})$.*
- *The $\otimes_a$ operator is defined for $(d_1, c_1),\ (d_2, c_2) \in D_a \setminus \{\boldsymbol{0}_a, \boldsymbol{1}_a\}$,*

$$(d_1, c_1) \otimes_a (d_2, c_2) \ = \ \begin{cases} 0_a & \textit{if}\ \texttt{Filter}(d_1, c_2) = \emptyset \\ & \textit{and}\ d_1 \in D_1 \\ (\texttt{FilterReceiver}(d_1, c_2) \otimes d_2, c) & \textit{otherwise} \end{cases}$$

  where $c = c_1 \cup_t c_2 \setminus (\texttt{KnownSatisfy}(d_1, c_2) \cup \texttt{KnownNotSatisfy}(d_1, c_2))$.

- *The $\oplus_a$ operator is defined for $(d_1, c_1),\ (d_2, c_2) \in D_a$,*

$$(d_1, c_1) \oplus_a (d_2, c_2) \ = \ (d_1 \oplus d_2, c_1 \cup c_2)$$

The definition of $\oplus_a$ operator says:

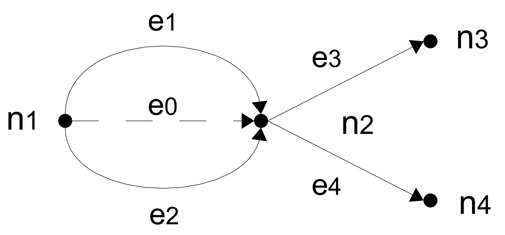
- If $d_1 \in D_1$, and there is no receiver object that satisfies the type constraints in c_2, the result will be 0_a, i.e. an invalid control flow.
- Otherwise, some valid control flows may happen by removing
  - The set of objects reaching d_1 that have been known not to satisfy any type constraint in c_2;
  - The set of type constraints from c_2 that have been known to be satisfied with some object reaching d_1;
  - The set of type constraints from c_2 that have been know not to be satisfied with any object reaching d_1;

Example 9 *Figure 4.2 provides an example to illustrate how the algorithm in Definition 28 works. It is assumed that there is a virtual method call at the node n_2 as:*

```
Employee x;
......
n2: x.getName();
```

Two possible types of method receivers are approximated in the present call graph: the class `Professor` shown as e_3 and the class `CarMaker` shown as e_4. Reminds that, the class `Professor` inherits `Employee`, but there are no inheritance relations among `Student`, `CarMaker`, `Pofessor` and `Employee`.

Therefore, when excluding the dotted edge e_0 for a while, only the control flow "$n_1 \rightarrow_{e_1} n_2 \rightarrow_{e_3} n_3$" is valid. The proposed algorithm in Definition 28 works as follows:



e_0 : $\{\lambda x.x, \{\texttt{any}\}\}$
e_1 : $\{\lambda x.o_1, \{\texttt{any}\}\}$, and $\texttt{type}(o_1) = \texttt{Professor}$
e_2 : $\{\lambda x.o_2, \{\texttt{any}\}\}$, and $\texttt{type}(o_2) = \texttt{Student}$
e_3 : $\{\lambda x.x, \{\texttt{Employee}\}\}$
e_4 : $\{\lambda x.x, \{\texttt{CarMaker}\}\}$

Figure 4.2: An Example for the Ahead-of-time Points-to Analysis

$$
\begin{aligned}
&e_1 \otimes_a e_3 = \{\lambda x.o_1, \{\mathit{any}\}\}\\
&e_1 \otimes_a e_4 = 0_a\\
&(e_1 \oplus_a e_2) \otimes_a e_3 = \{\lambda x.o_1, \{\mathit{any}\}\}\\
&(e_1 \oplus_a e_2) \otimes_a e_4 = 0_a
\end{aligned}
$$

If e_0 is included, the flows "$n_1 \rightarrow_{e_0} n_2 \rightarrow_{e_3} n_3$" and "$n_1 \rightarrow_{e_0} n_2 \rightarrow_{e_4} n_3$" may be valid, it then works as:

$$
\begin{aligned}
&(e_0 \oplus_a e_2) \otimes_a e_4 = \{\lambda x.x, \{\mathit{CarMarker}\}\}\\
&(e_0 \oplus_a e_1) \otimes_a e_3 = \{\lambda x.x \cup o_1, \{\mathit{Employee}\}\}
\end{aligned}
$$

After one time points-to analysis(model checking), the initial call graph is refined according to points-to information. Then another turn of points-to analysis is performed based on the newly-refined call graph. The analysis is an iterative procedure to find a greatest fixed point. Therefore, it is only a choice on the trade-off between precision and efficiency to stop this iterative procedure. Whereas, for an analysis based on an on-the-fly call graph construction, the iterative procedure can not be stopped until stabilized for soundness.

The second step is an exclusive points-to analysis concerning fields (we focus on the field sensitivity at most depth 1), based on the points-to information obtained in phase 1. The underlined mode is an exploded supergraph on $\mathcal{O} \cup \mathcal{O} \times \mathcal{F}$ and working on the algorithm in Definition 21. The problem of aliasing is cast in the first step, and we naturally obtain SSA on $\mathcal{O} \cup \mathcal{O} \times \mathcal{F}$.

4.2 Parameterized Flow-sensitivity by Model Reduction

Hitherto, we have presented two context/field-sensitive points-to analysis algorithms based on an exploded supergraph model design. A flow-sensitive points-to analysis is precise but heavy. In this section, a *parameterized* flow-sensitivity, under the tradeoff between precision and efficiency, is shown to be naturally obtained by shrinking an underlying interprocedural CFG G_f before an exploded supergraph is constructed (in Algorithm 19).

After shrinking, a node in G_f is possibly associated with a set of program statements; and the size of the resulted exploded supergraph is reduced. Example 10 shows how a model reduction is applied to the running example.

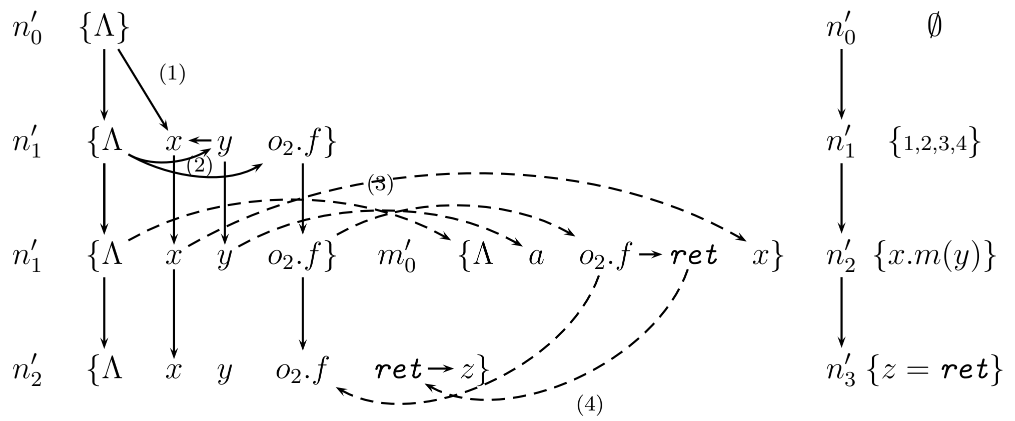

n'_0 $\{\Lambda\}$
(1)
n'_1 $\{\Lambda$ x y $o_2.f\}$
(2)
(3)
n'_1 $\{\Lambda$ x y $o_2.f\}$ m'_0 $\{\Lambda$ a $o_2.f$ $\mathtt{ret}$ $x\}$
n'_2 $\{\Lambda$ x y $o_2.f$ $\mathtt{ret}$ $z\}$
(4)
n'_0 $\emptyset$
n'_1 $\{1,2,3,4\}$
n'_2 $\{x.m(y)\}$
n'_3 $\{z = \mathtt{ret}\}$

Figure 4.3: (a) Part of the G_p *wrt* (b) (b) Figure 4.1 (b) after Shrinking

Example 10 *Figure 4.3 (b) shows the run in Figure 4.1 (b) after shrinking; and Figure 4.3 (a) shows the part of exploded supergraph G_p wrt the run in (b), where labels are omitted for simplicity. some representative pushdown transition rules wrt Figure 4.3 (a) are as follows:*

$$\begin{array}{ll}\langle \Lambda, n'_0\rangle \hookrightarrow \langle x, n'_1\rangle & (1)\\ \langle y, n'_1\rangle \hookrightarrow \langle x, n'_1\rangle & (2)\\ \langle y, n'_1\rangle \hookrightarrow \langle a, m'_0 n'_2\rangle & (3)\\ \langle \mathtt{ret}, n'_1\rangle \hookrightarrow \langle \mathtt{ret}, \varepsilon\rangle & (4)\end{array}$$

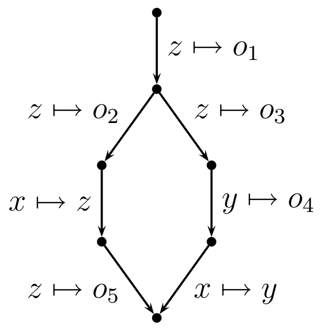
To note that, there are transitions inside one node of the interprocudural CFG after shrinking. These transitions inside one node are analyzed in a flow-insensitive manner. That is, except the first entry statement, others are analyzed regardless of their execution orders. Correspondingly, the pushdown stack keeps unchanged for such a transition rule inside nodes enclosed by $\{\}$.

For soundness, the shrinking is not arbitrary. Usually, it is required that each node has an unique entry statement after shrinking. Flow-sensitivity is parameterized by variations on shrinking strategies. For instance,

- A moderate shrinking strategy in our analysis is
  - The entry points of programs start new nodes.
  - A procedure call starts a new node.
  - Any target of branches starts a new node.
- An extreme shrinking strategy is that all nodes in a method are grouped into a single node, which results in a complete flow-insensitive analysis. The interprocedural CFG G_f is equivalent to the call graph in this case.

Figure 4.4 compares the precision of points-to analysis with various shrinking strategies.

Due to the mutual dependency among call graph construction and points-to analysis, iterative procedures cannot be avoided in the analysis design to guarantee soundness. In the on-the-fly analysis, a points-to analysis (model checking) request is dispatched



Shrinking	x	z	y
no	$\{o_2, o_4\}$	$\{o_3, o_5\}$	$\{o_4\}$
moderate	$\{o_2, o_4, o_5\}$	$\{o_2, o_3, o_5\}$	$\{o_4\}$
extreme	$\{o_1, o_2, o_3, o_4, o_5\}$	$\{o_1, o_2, o_3, o_5\}$	$\{o_4\}$

Figure 4.4: A Comparison on the Precision of Variations on Flow-Sensitivity

whenever a virtual method call need to be resolved or a field access is encountered. Iterations are needed when there are model checking dispatched from some looping structures. Detailed discussions and algorithms will be given in Section 4.3.

For a flow-insensitive analysis, soundness becomes simpler: each strongly connected component (SCC) in a PAG (that does not across procedures) is collapsed as a single node. That is, variables in one SCC have the same points-to sets. To note that, this threatens the precision of the analysis when many method invocations are just inside the SCCs [45]. The situation is a bit more complicated for a flow-sensitive analysis. There are two kinds of iterations to be considered,

- *Global iterations*: SCCs in the call graph, i.e. nested procedural calls and recursions.
- *Local iterations*: SCCs inside each procedures, i.e. looping structures of programs.

The way of handling local iterations and global iterations are rather tradeoff between designs and efficiency.

Parameterized flow-sensitivity provides an analysis design with flexibility, but some cares need to be paid on the iteration procedures, especially for the on-the-fly analysis. There are two primary issues to be noticed:

- *irreducible* SCCs vs *reducible* SCCs
  Some SCC might be irreducible [59], that is intuitively, there are jumps into the middles of loops. So there are more than one node that is the target of branches. By the aforementioned moderate shrinking, it is easily to be seen that irreducible SCCs are broken into multiple shrunk nodes after shirking. However, reducible SCCs could be kept inside one shrunk node, as shown in Figure 4.5 (a) and (b). Design of iterative procedures on these SCCs need cares.

- choices on field-sensitivity
  Based on an exploded supergraph model design, the aliasing among references need to be settled before each field-sensitive abstraction is applied. Parameterized flow-sensitivity demands some cares on field processing, as illustrated in Example **??**. An easy solution is to further require that each field access starts a new node or each new node only consists of a single field access statement. For the latter, the statement involved field access is analyzed finally when processing a new node.

Example 11 *Figure 4.6 illustrates an occasion on which the processing order matters when field access is involved. It becomes more problematic when aliasing is involved. In*

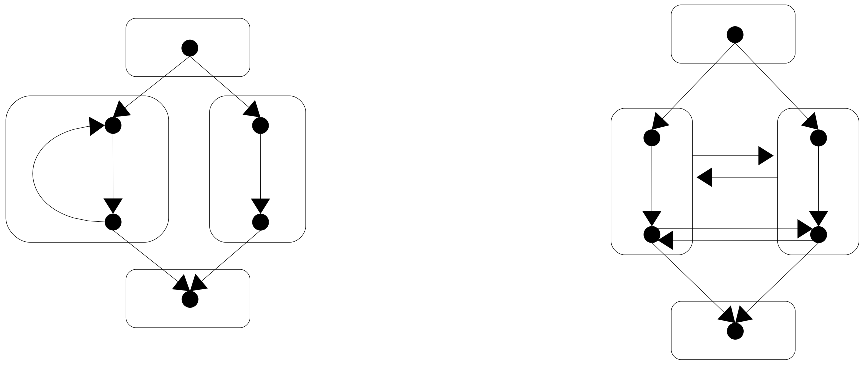


Figure 4.5: (a) Shrinking on Reducible SCCs (b) Shrinking on Irreducible SCCs

this example, x and y are aliased at line 2; and after line 5, z may point to the $o_1.f$ or $o_2.f$ or o. In particular, more iterative processing is needed when the loop shown in dotted edges exists.

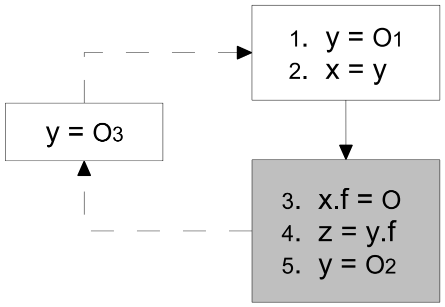


Figure 4.6: The Effect of Shrinking on Handling Field Access

4.3 Prototype Implementation

We tentatively test the on-the-fly points-to analysis algorithm based on the exploded supergraph model design. In this section, the prototype framework used is presented and some preliminary evaluation is given.

4.3.1 Jimple Semantics Related to Points-to Analysis

Our implementation use SOOT, which is a Java optimization/compilation framework [41], as the frontend preprocessor. More specifically, our analyzer is implemented as a new phase in the SOOT compiler.

SOOT provides four intermediate representations that are suitable for analyzing and transforming Java bytecode. Our analysis works on Jimple that is a typed three-address intermediate representation of Java. Jimple has a small set of language constructs keeping Java semantics, which makes program analyses and optimization easier. For instance, there is no such a nesting of fields as "$x.f_1...f_n$" syntactically; and an instance invocation can be only dispatched on object variables like $x.f(...)$.

We are interested in the part of Java semantics that relates to points-to analysis. For simplicity, some advanced language features, such as reflection, exception, native methods, etc., are excluded at the first stage. These topics will be discussed in Chapter **??**. Table 4.4 shows the language model that we work on, which is part of the Jimple syntax that relates to points-to analysis.

Table 4.4: Jimple Syntax Related to Points-to Analysis

(*Variables*)	Variable	::=	Local | Reference
	Reference	::=	Field
		|	Local.Field
		|	Local[Imm]
	Imm	::=	Local | Constant
	Constant	::=	NullConstant | StringConstant
(*Statements*)	Stmt	::=	AssignStmt | IdentityStmt
			| InvokeStmt | ReturnStmt
	AssignStmt	::=	Variable = Expr;
	IdentityStmt	::=	Local := @this: RefType;
		|	Local := @parameter$_n$: RefType;
	InvokeStmt	::=	InvokeExpr;
	ReturnStmt	::=	return | return Imm;
(*Expressions*)	Expr	::=	new RefType
		|	newarray (Type)[Imm]
		|	newmultiarray (Type)[Imm$_0$]...[Imm$_n$]
		|	InvokeExpr
		|	CastExpr
		|	Reference
		|	Imm
	CastExpr	::=	(RefType) Variable
	InvokeExpr	::=	NonStaticInvokeExpr | StaticInvokeExpr
	NonStaticInvokeExpr	::=	VirtualInvoke Local.m(Imm$_0$, ..., Imm$_n$)
		|	InterfaceInvoke Local.m(Imm$_0$, ..., Imm$_n$)
		|	SpecialInvoke Local.m(Imm$_0$, ..., Imm$_n$)
	StaticInvokeExpr	::=	StaticInvoke m(Imm$_0$, ..., Imm$_n$)

4.3.2 Prototype Framework and Preliminary Evaluation

We use the prototype framework in Figure 6.6 to implement points-to analysis algorithms presented in this chapter. There are generally three phases:

- In *phase 1*, the SOOT [41] compiler is explored as the frontend for preprocessing from Java programs to Jimple codes. Jimple is a three-address intermediate representation for Java with smaller language constructs. SOOT also provides facilities of call

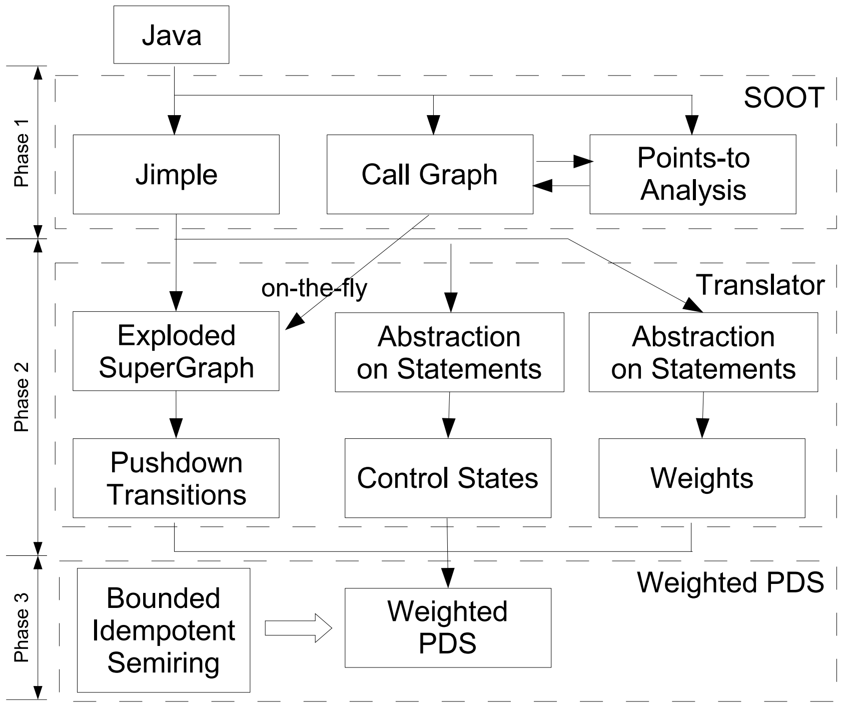


Figure 4.7: A Prototype Framework for Points-to Analysis

graph generation and points-to analysis based on various well-known algorithms, and we will apply the most imprecise analysis (Class Hierarchy Analysis) [55] to produce a preliminary call graph for the ahead-of-time call graph construction in Chapter 5.

- In *phase 2*, the model abstraction translates Jimple codes to the underlying model (i.e. weighted pushdown systems) for model checking. For abstraction, it takes around 3000 lines of Java codes for the on-the-fly exploded supergraph based model design (Section 4.1.2).
- In *phase 3*, Weighted PDS library is explored as the back-end model checking engine, which calls an implementation of semiring designs. The semiring package is 1000 lines in C for the on-the-fly exploded supergraph based model design.

Preliminary Evaluation

Our tentative experiments, except a part (about 3000 lines of Jimple codes) of an open source Java program *jetty*, are restricted to small examples. Only the flow insensitive points-to analysis on an on-the-fly exploded supergraph based model design (Section 4.1.2) can analyze it in reasonable time.

The total time for call graph generation takes 632.23secs (the average of three tests), which resolves nearly 40% of virtual calls. 97.99% (619.57secs) of the execution time is devoted to incremental model generation with 1173 times model checking requests involved. All above experiments are performed on a 1.4GHz Pentium 4 processor with 1 GB memory, running RedHat Linux 2.4.20.

The first obstacle could be the time complexity of solving the GPR problem [46]. Our choice was context/flow/field sensitive points-to analysis, and this would be quite inevitable. The time complexity specific to our cases is $\Theta(|C|^2 \cdot |\Delta| \cdot s)$, where s is the cardinality of the weight space. C is the cardinality of N_p for the exploded supergraph based model design, and can be ignored for the CFG based model design.

The current dominant factor seems I/O between modules. In our current implementation, the weighted PDS library and the bounded idempotent semiring module is in C, while SOOT and the translator module is in Java. For the above example, 75.5% (468 secs) of incremental model generation (619.57secs) is devoted to just file-based I/O between these modules. This situation also prevents data structures and internal program states from sharing, so that a weighted pushdown system is constructed from scratch for each model checking request. It is one of the primary bottlenecks of current implementations. By diagnosis, the weighted PDS library runs quite fast. An incremental model construction grows to with 100,000 to 200,000 transition rules, and each cycle runs in seconds to less than a minute.

Currently, native methods, libraries, static initializers, are all ignored. Concurrent behavior, such as threads, is also out of scope (treated as sequential executions currently). Further approximation is needed to cover concurrent behavior, since pushdown model checking with more than two stacks is undecidable.

4.3.3 Implementation Aspects

Algorithms wrt on-the-fly points-to analysis is usually complicated, because of delicate iterative procedures. So usually they will not be given. In this section, our strategy for presenting the algorithms is: a basic skeleton will be presented and key points of details will be discussed.

Algorithm 1 shows one basic skeleton of processing a method without shrinking, which is the first step for both algorithms presented (i.e. either on-the-fly or ahead-of-time). Our choice is to handle local iterations on-the-fly induced by SCCs.

From line 1 to line 4 is the basic preprocessing part. For efficiency, a list L of CFG nodes of the method in a topological order is constructed, by regarding each SCC as one node; and an extra entry node is introduced to the CFG G. Then, Starting with the first entry node, all nodes in L are processed sequentially as follows, for any node v in L

- If v has no successors in G, edges are established among v and the exist node, as shown by line 10 to line 13.

- If v has successors in G, edges are established among v and each successor (line 15 to line 16). If the successor is a SCC, then edges induced by this SCC will be established (line 18 to line 23). If there are model checking requests dispatched during this processing, these dispatching nodes will be analyzed iteratively (line 24 to line 28).

On-the-fly and ahead-of-time manner primarily differ in the method of **processStmt**. If there is a viral call invoked on this point (as well as field access), the former will dispatch a model checking request, and the latter will refer to the approximated call graph.

Algorithm 1 **processMethod**(m)

```
1: build the CFG G(V, E) of $m$ with ndoes V and edges E
2: find all maximal SCCs in G
3: build a list $L$ of V in a topological order on G, by regarding each SCC as one node
4: add an extra entry node *entry* into G
5: get the first node *head* of $L$
6: processStmt(*entry*, *head*)
7: while $L$ is not empty do
8:    get the first node $v$ of $L$
9:    get the set of successors *succSet* of $v$ in G
10:   if *succSet* is empty then
11:      add an unique *exit* node into G
12:      processStmt(v, *exit*)
13:   end if
14:   while *succSet* is not empty do
15:      get one successor *succ* from *succSet*
16:      processStmt($v$, *succ*)
17:      if *succ* is a shrunk SCC $G_s = (V_s, E_s)$ then
18:         for all $e = (v_1, v_2) \in E_s$ do
19:            processStmt($v_1, v_2$)
20:            if points-to analysis is dispatched then
21:               add $e$ into the set *mcSites*
22:            end if
23:         end for
24:         if *mcSites* is not empty then
25:            while all analysis results from *mcSites* do not stabilize do
26:               process all edges in *mcSites*
27:            end while
28:         end if
29:      end if
30:      remove *succ* from *succSet*
31:   end while
32:   remove $v$ from $L$
33: end while
34: processStmt(*exit*, *null*)
```

More attention need to be paid on the on-the-fly algorithm. Reminds that, the generalized pushdown predecessor problem computes data flows facts along all pathes leading from some point to another set of points (i.e. regular pushdown stores) in the pushdown computation tree. To make use of this facility, the starting and ending point need to be specified each time a model checking request is dispatched. That is, the returned points for invocations need to be remembered.

As mentioned above, there are local iterations within procedures and global iterations among procedures to be concerned. Fortunately, the regular property of pushdown configurations reduces the number of iterations, with no sacrifice on soundness. Assume a model checking is dispatched when processing some node v in L, all possible calling contexts to v are simply represented as $(\Sigma)^* v$, where Σ is the current set of return points of the program.

Chapter 5

Context-sensitive Points-to Analysis based on Interprocedural CFG

Chapter 4 presents points-to analysis algorithms based on an exploded supergraph model design. By algorithm design and preliminary evaluation, we found that

- Model checking requests dispatched during the on-the-fly analysis are overwhelming, which influences a lot on the efficiency;
- Whereas the ahead-of-time analysis expected with automatic path removal and field tracing are restricted by the exploded supergraph based model design.
- An on-the-fly analysis design is annoyed by the delicate treatment on iterative procedures.

Therefore, this Chapter is dedicated to exploring points-to analysis based on an interprocedural control flow graph (CFG) model design. Under such a choice, the flavor of both on-the-fly and ahead-of-time call graph construction will also be investigated. In particular, we expect the following features that could be realized in the ahead-of-time analysis:

- Invalid pathes in the approximated call graph are removed automatically by model checking. More specifically, invalid control flows are excluded from the analysis result by the model checking engine.
- A precise analysis on fields (i.e. field-sensitive abstraction), as well as the problem of aliasing, are properly solved as part of the model checking procedure, rather than clarified with extra efforts by frequent interrupts.
- Hopefully, points-to analysis, as well as all the above, can be done with one-time model checking.

By examining various context-sensitive points-to analysis algorithms, our study shows that:

- Only ahead-of-time call graph construction based on a control flow graph can avoid repeated applications of model checking.

- An ahead-of-time construction needs to explicitly give a bound on tracing nesting of field access, whereas an on-the-fly construction automatically bounds it (up to the number of abstract heap location).

This Chapter is organized as follows:

In Section 5.1.1, a delicate abstraction design of transfer functions on heap environments is presented, by exactly following to Java semantics. In Section 5.1.2, a context/field/flow-sensitive points-to analysis algorithm is proposed, based on the interprocedural CFG model design and on-the-fly call graph construction. In Section 5.1.3, another context/field/flow-sensitive points-to analysis algorithm is proposed, based on the interprocedural CFG model design and ahead-of-time call graph construction.

In Section 5.2, we also further investigate the relatively unexplored parametrization problem. Compared with parameterized flow-sensitivity by applying model reduction, it is shown that parameterized flow-sensitivity can be naturally obtained by simplifying the weight design.

In Section 5.3, the implementation aspects are presented, by characterizing weight functions as ordered sets.

5.1 Interprocedural CFG Based Model Construction

In this section, we propose context-sensitive points-to analysis algorithms based on the interprocedural CFG model design. Both the on-the-fly 5.1.2 and ahead-of-time 5.1.3 model construction are investigated.

5.1.1 Abstract Heap Environment Transformers

This chapter will share the same abstraction on the heap objects and references, which are the basic components of the points-to relation. For reminding, it is again presented here in Definition 29

Definition 29 *let $\mathcal{O}$ be the set of abstract heap objects, and let $\mathcal{F}$ be the set of field names. Let $\mathcal{V}_l$ be the set of local variables, let $\mathcal{V}_s$ be the set of static fields. The set of abstract reference variables is* $\text{RefVar} = \mathcal{V}_l \cup \mathcal{V}_s \cup \{\texttt{arg}_k, \texttt{this}, \texttt{ret} \mid k \in \mathbb{N}\}$*, and the set of reference fields* $\text{RefField} = \mathcal{O} \times \mathcal{F}^+$*. Let* $\mathcal{V}_{ref} = \text{RefVar} \cup \text{RefField}$.

Points-to analysis algorithms in this chapter work on the weight space that consists of transfer functions on heap environments.

Definition 30 *The set of abstract heap environments is defined as*

$$\text{Henv} = \{\text{henv} \mid \text{henv} : \mathcal{V}_{ref} \rightarrow \mathcal{O}\}$$

Thus, the points-to information is regarded as an abstract heap environment $\mathtt{h}_{\mathbb{V}} \in \text{Henv}$ such that $[v_1 \mapsto \mathtt{h}_{\mathbb{V}}(v_1), ..., v_n \mapsto \mathtt{h}_{\mathbb{V}}(v_n)]$, where $\mathbb{V} = \{v_i \mid 1 \leq i \leq n\}$. A field reference $x.f$ is "evaluated" as $\mathtt{h}(\mathtt{h}(x).f)$, i.e. a field-sensitive abstraction, where $\mathtt{h} \in \text{Henv}$.

Definition 31 *Let $v \in \texttt{RefVar}$, $f \in \mathcal{F}$, $o \in \mathcal{O}$ and $\texttt{henv} \in \text{Henv}$. $\mathbb{F}$ is the set of abstract heap environment transformers, given by* ExpFun *in Table 5.1.*

Table 5.1: Syntax of Expressions of Abstract Heap Environment Transformers

$$
\begin{array}{lcl}
\text{ExpFun} & ::= & \lambda\texttt{henv}.\ \text{ExpHenv} \\
\text{ExpHenv} & ::= & \texttt{henv} \\
 & | & \text{ExpHenv} \bullet \text{ExpMap} \\
\text{ExpMap} & ::= & [v \mapsto o] \\
 & | & [v_1 \mapsto \text{Expt}, ..., v_n \mapsto \text{Expt}] \\
 & | & [\text{Expf} \mapsto \text{Expt}] \\
\text{Expf} & ::= & \text{Expt}.f \\
\text{Expt} & ::= & o \\
 & | & \texttt{henv}(v) \\
 & | & \texttt{henv}(\text{Expt}.f)
\end{array}
$$

The meaning of $\bullet$ is shown by rewriting rules in Definition 32. The intuition of $\texttt{henv} \bullet \texttt{henv}'$ is the union of $\texttt{henv}$ and $\texttt{henv}'$ except that mapping in $\texttt{henv}$ are overridden by those that have the same arguments in $\texttt{henv}'$. For instance,

$$\texttt{henv} \bullet [x \mapsto o_1, y \mapsto o_2, x \mapsto o_3](x) = o_3$$

where $\texttt{henv} \in \text{Henv}$.

Definition 32 *Let* $\texttt{henv} \in \text{ExpHenv}$, $o \in \mathcal{O}$, $v, v_i \in \mathcal{V}_{ref}$, $\texttt{expf} \in \text{Expf}$, $\texttt{expt}, \texttt{expt}_i \in \text{Expt}$ *for* $1 \le i \le n$.

$$\texttt{henv} \bullet [v_1 \mapsto o](v) = \begin{cases} o & \textit{if } v = v_1 \\ \texttt{henv}(v) & \textit{otherwise} \end{cases}$$

$$\texttt{henv} \bullet [v_1 \mapsto \texttt{expt}_1, ..., v_n \mapsto \texttt{expt}_n](v) = \begin{cases} \texttt{expt}_i & \textit{if } v = v_i (1 \le i \le n) \\ & \textit{and } \forall 1 \le j \le n,\ v_j = v,\ \textit{s.t. } j \le i \\ \texttt{henv}(v) & \textit{otherwise} \end{cases}$$

$$\text{henv} \bullet [\texttt{expf} \mapsto \texttt{expt}](v) = \begin{cases} \texttt{expt} & \textit{if } v = \texttt{expf} \\ \texttt{henv}(v) & \textit{otherwise} \end{cases}$$

Definition 33 *The composition of abstract heap environment transformers is defined as follows, for* $\text{exph}_1, \text{exph}_2 \in \text{ExpHenv}$,

$$(\lambda\texttt{henv}.\ \text{exph}_2) \circ (\lambda\texttt{henv}.\ \text{exph}_1) = \lambda\texttt{h}.\ \text{exph}_2[\texttt{henv} := \text{exph}_1[\text{henv} := \texttt{h}]]$$

The intuition of this definition is the standard η-expansion, shown as follows,

$$\begin{array}{lcl}(\lambda\texttt{henv}.\ \text{exph}_2) \circ (\lambda\texttt{henv}.\ \text{exph}_1) & =_\eta & \lambda\text{h}.\ (\lambda\texttt{henv}.\ \text{exph}_2(\lambda\texttt{henv}.\ \text{exph}_1\texttt{h})) \\ & =_\beta & \lambda\text{h}.\ \text{exph}_2[\texttt{henv} := \text{exph}_1[\texttt{henv} := \text{h}]]\end{array}$$

Definition 34 *A bounded idempotent semiring* $S_c = (D_c, \oplus_c, \otimes_c, 0_c, 1_c)$ *is defined as*

- *The weight space is* $D_c = \mathcal{P}(\mathbb{F})$

- $0_c = \emptyset$
- $1_c = \{\lambda \texttt{henv}.\texttt{henv}\}$
- *The* $\otimes_c$ *operator is defined as* $\forall w_1, w_2 \in D_c$,

$$w_1 \otimes_c w_2 = \{\texttt{func}_2 \circ \texttt{func}_1 \mid \texttt{func}_1 \in w_1, \texttt{func}_2 \in w_2\}$$

- *The* $\oplus_c$ *operator is defined as* $\forall w_1, w_2 \in D_c$,

$$w_1 \oplus_c w_2 = w_1 \cup w_2$$

The $\otimes$ operation in Definition 34 is the reverse of function composition. Thus, the associativity of $\otimes$ can be easily checked. The analysis result is a set of abstract heap environment transformers. Each transformer corresponds to changes on heap environment along a possible program run. The final points-to information can be obtained by applying these transformers to the initial abstract heap environment of the program. A generalized definition of *evaluation* is provided in Definition 35.

Definition 35 *Let* $v \in \text{RefVar}$*,* $o \in \mathcal{O}$*,* $w \in \mathcal{F}^*$*, and* $\texttt{henv} \in \text{Henv}$*. The evaluation of* $v.w$*,* $o.w$ *by* $\texttt{henv}$ *is defined as*

$$eval(\texttt{henv}, v.w) = \begin{cases} \texttt{henv}(v) & \text{if } w = \epsilon \\ (\texttt{henv} \ \dots \ \texttt{henv}(v).f_1) \ \dots \ f_n) & \text{if } w = f_1 \cdots f_n \in \mathcal{F}^* \end{cases}$$

$$eval(\texttt{henv}, o.w) = \begin{cases} o & \text{if } w = \epsilon \\ (\texttt{henv} \ \dots \ (\texttt{henv}(o.f_1)) \ \dots \ f_n) & \text{if } w = f_1 \cdots f_n \in \mathcal{F}^* \end{cases}$$

The algorithm in Definition 34 provides the basic points-to analysis algorithm based on weighted pushdown model checking. By abuse of terminology, the machinery we have created till now actually defines a kind of language that is about to work on the pushdown system. However, what we are interested is the static analysis with it. Obviously, it is demanded by Definition 34 that $\mathbb{F}$ need to be finite. It can be seen from Definition 5.1 that $\mathbb{F}$ can be infinite due to the field nesting, such as $o.f_0...f_n$, where $o \in \mathcal{O}$ and f_i $(0 \leq in) \in \mathcal{F}$. The set of abstract heap objects is finite, for that each allocation site is associated with an unique abstract heap object (Definition 17 in Chapter 4).

In the following section, we will present points-to analysis algorithms with call graph constructed in both on-the-fly and ahead-of-time manner. The decidability of the analysis is either ensured immediately by the algorithm itself (in Section 5.1.2) or by placing a bound on the field nesting (in Section 5.1.3).

5.1.2 On-the-fly Call Graph Construction

By the on-the-fly analysis, we mean that the call graph is constructed on-the-fly when the points-to analysis proceeds. Based on the interprocedural CFG model design, an on-the-fly points-to analysis can be performed similarly to that in section 4.1.2. Reminds that, for the on-the-fly points-to analysis based on the exploded supergraph model design, to perform field-sensitive abstraction and clarify aliasing also dispatch model checking requests during the points-to analysis.

Based on the interprocudural CFG model design, we have more choices on analyzing fields, e.g. a field-sensitive abstraction could be solved by encoded as part of the weight design. Since this choice will be explored in the next section anyway, our choice here is to clarify aliasing and the field-sensitive abstraction on-the-fly as well.

Table 5.2 shows how a weight function, i.e. a transformer on abstract heap environments, is assigned to each Jimple statement with abstraction. Except the last two statements that are Jimple-specific, Java shares the same syntax and semantics. In Table 5.2, the abstraction of x = new T() is $\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto o]\}$. Note that this o is unique for this statement at some allocation site (i.e. line number) in the program, even if it is executed in a looping structure. This abstraction keeps an abstract heap objects finite, i.e., at most the number of program lines.

As is shown in Table 5.2, whenever a virtual method invocation (resp. a field access) is encountered during the analysis, model checking is dispatched to resolve possible call edges (resp. perform a field-sensitive abstraction). For instance, one of the weight functions associated with the field read access "$x = y.f$" is

$$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(o.f)]$$

where $o \in \texttt{pta}(y, \texttt{context})$, and $\texttt{context} \in \mathbb{C}$ is the program calling context wrt this analysis point. That is, a points-to analysis (i.e. weighted pushdown model checking) is performed on y, when this statement is encountered. The possible aliasing on x is also casted correspondingly.

Example 12 shows how the same example in Example 7 in Chapter 4 is solved by the on-the-fly points-to analysis based on the interprocedural CFG model design.

Example 12 *Figure 5.1 shows part of the underlined model for model checking that corresponds to one possible run with program line numbers of 1-2-3-4-5. Each edge is assigned with a weight according to Table 5.2.*

First, when the field access $y.f = o_3$ is encountered at line 3. A model checking is dispatched on the node n_2, such that

$$e_1 \otimes e_2 = \lambda \textit{henv}.\textit{henv} \bullet [x \mapsto o_1] \bullet [y \mapsto o_2]$$

By Definition 35 and 32,

$$eval(e_1 \otimes e_1, y) = o_2$$

Thus,$\texttt{y.f}$ is abstracted as $o_2.f$.

With an on-the-fly analysis, virtual method calls are resolved on-the-fly when points-to analysis proceeds. Reminds that there is a virtual method call at node n_5. Thus a model checking is dispatched on it. The resulted abstract heap transformer reaching n_5 is computed as follows:

$$\begin{aligned} \texttt{weight} &= e_1 \otimes_c e_2 \otimes_c e_3 \otimes_c e_4 \otimes_c e_5 \\ &= \lambda \textit{henv}.\textit{henv} \bullet [x \mapsto o_1] \bullet [y \mapsto o_2] \bullet [o_2.f \mapsto o_3] \bullet \\ &\qquad [x \mapsto \textit{henv} \bullet [x \mapsto o_1] \bullet [y \mapsto o_2] \bullet [o_2.f \mapsto o_3](y)] \\ &= \lambda \textit{henv}.\textit{henv} \bullet [x \mapsto o_1] \bullet [y \mapsto o_2] \bullet [o_2.f \mapsto o_3] \bullet [x \mapsto o_2] \end{aligned}$$

$$eval(\texttt{weight}, x) = o_2$$

It is known that x points to o_2 after evaluation. Thus, only `method` *from Class* `Professor` *is invoked. And similarly, we know that z points to o_3 at the node n_7.*

Table 5.2: Jimple Syntax and Abstraction Related to Points-to Analyses

Jimple Syntax	Abstraction
$x = \text{new T()}$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto o]\}$
$x = null$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \diamond]\}$
$x = y$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(y)]\}$
$x = y.f$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(o.f)] \mid o \in \texttt{pta}(y, \texttt{context})\}$, where $\texttt{context} \in \mathbb{C}$ is the calling context.
$y.f = x$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [o.f \mapsto \texttt{henv}(x)] \mid o \in \texttt{pta}(y, \texttt{context})\}$ where $\texttt{context} \in \mathbb{C}$ is the calling context.
return x	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{ret} \mapsto \texttt{henv}(x)]\}$, when x is a reference variable.
$x.f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{arg}_1 \mapsto \texttt{henv}(m_1), ..., \texttt{arg}_l \mapsto \texttt{henv}(m_l),$ $\texttt{this} \mapsto \texttt{henv}(x)]\}$, where $m_i (1 \leq i \leq l) \in \text{RefVar}$ and $m_j (l \leq j \leq n)$ are variables of primitive type a call edge is generated for each receive class in $\{t \mid t = \texttt{type}(o), o \in \texttt{pta}(y, \texttt{context})\}$, for the calling context $\texttt{context} \in \mathbb{C}$.
$f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{arg}_1 \mapsto \texttt{henv}(m_1), ..., \texttt{arg}_l \mapsto \texttt{henv}(m_l)]\}$, where $m_i (1 \leq i \leq l) \in \text{RefVar}$ and $m_j (l \leq j \leq n)$ are variables of primitive type a call edge is generated by syntactical analysis.
$z = \texttt{ret}$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [z \mapsto \texttt{henv}(\texttt{ret})]\}$
x := @this: T	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(\texttt{this})]\}$
x := @parameter$_k$: T	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(\texttt{arg}_k)]\}$

$$\begin{tikzcd}
n_0 \arrow[d, "e_1\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[x\mapsto o_1]}"'] & \\
n_1 \arrow[d, "e_2\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[y\mapsto o_2]}"'] & \texttt{Employee}.m_0 \\
n_2 \arrow[d, "e_3\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[o_2.f\mapsto o_3]}"'] & \\
n_3 \arrow[d, "e_4\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[x\mapsto \texttt{henv}(y)]}"'] & \\
n_4 \arrow[d, "e_5\text{: } \lambda\texttt{henv}.\texttt{henv}"'] & \\
n_5 \arrow[r, "e_6\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[\texttt{this}\mapsto \texttt{henv}(x)]}", "e_7\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[\texttt{arg}\mapsto \texttt{henv}(y)]}"'] \arrow[ruuuu, dashed] & \texttt{Professor}.m_0 \arrow[d, "e_8\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[b\mapsto \texttt{henv}(\texttt{arg})]}"] \\
n_6 \arrow[d, "e_{11}\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[z\mapsto \texttt{henv}(\texttt{ret})]}"'] & \texttt{Professor}.m_1 \arrow[d, "e_9\text{: } \lambda\texttt{henv}.\texttt{henv}\bullet{[\texttt{ret}\mapsto o_2.f]}"] \\
n_7 & \texttt{Professor}.m_2 \arrow[lu, "e_{10}\text{: } \lambda\texttt{henv}.\texttt{henv}"']
\end{tikzcd}$$

Figure 5.1: Part of the Underlined Model for Model Checking of Example 7

5.1.3 Ahead-of-time Call Graph Construction

This section presents a points-to analysis algorithm based on an ahead-of-time call graph construction. The algorithm starts with an imprecise call graph (which could be generated by syntactical analysis only) and infeasible call paths are eliminated during model checking. Compared with the analysis based on an on-the-fly call graph construction, this algorithm is characterized by:

- Field sensitivity is explicitly modeled as part of the weight design, whereas it is settled by model checking in the on-the-fly algorithm.
- The ahead-of-time algorithm calls model checking once on a larger call graph, whereas the on-the-fly algorithm calls it frequently on smaller call graphs.
- The ahead-of-time algorithm computes the greatest fixed point (wrt set inclusion), whereas the on-the-fly algorithm computes the least fixed point.

These features are supported by:

- Transfer functions (i.e. the weight space) are restricted to k-bounded field access. In contrast, it is settled by the on-the-fly analysis in Section 5.1.2 of itself.
- Weights are extended with type constraints, i.e., if the abstract heap environment transformer reaching the current analysis point conflicts with type constraints, such a path will be eliminated.

Bound the field nesting

Example 13 shows that the field nesting during the analysis can be unbound.

Example 13 *The analysis on a sequence of Java statements works as follows.*

$$l_1: \; x = y.f;$$

$$l_2: \; z = x.g;$$

Let w_1 and w_2 be the corresponding weight functions for l_1 and l_2 respectively, then

$$\begin{aligned} w_1 \otimes w_2 &= w_2 \circ w_1 \\ &= \lambda \boldsymbol{henv}.\boldsymbol{henv} \bullet [x \mapsto \boldsymbol{henv}(\boldsymbol{henv}(y).f)] \bullet [z \mapsto \boldsymbol{henv}(\boldsymbol{henv}(\boldsymbol{henv}(y).f).g)] \end{aligned}$$

where

$$w_1 = \lambda \boldsymbol{henv}.\boldsymbol{henv} \bullet [x \mapsto \boldsymbol{henv}(\boldsymbol{henv}(y).f)]$$
$$w_1 = \lambda \boldsymbol{henv}.\boldsymbol{henv} \bullet [z \mapsto \boldsymbol{henv}(\boldsymbol{henv}(x).g)]$$

This example tells that field nestings can be unbound such as

$$\mathtt{henv}(\mathtt{henv}(x).g) = \mathtt{henv}(\mathtt{henv}(\mathtt{henv}(y).f).g)$$

which makes the set of variables $\mathcal{V}_{ref}$ (namely the set of abstract heap transfers $\mathbb{F}$) infinite. Therefore, a bound k is placed on tracking fields, such that for each $\mathtt{henv} \in \text{Henv}$,

$$\mathtt{henv}(eval(\mathtt{henv}, v.w).f) = \begin{cases} eval(\mathtt{henv}, v.w') & \text{if } |w| + 1 \leq k \text{ and } w' = w \cdot f \\ \top & \text{otherwise} \end{cases}$$

Table 5.3: Modified Abstraction for Ahead-of-time Construction

$x = y.f$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(\texttt{henv}(y).f)]\}$
$y.f = x$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{henv}(y).f \mapsto \texttt{henv}(x)]\}$
$x.f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$\{\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{arg}_1 \mapsto \texttt{henv}(m_1), ..., \texttt{arg}_l \mapsto \texttt{henv}(m_l),$ $\texttt{this} \mapsto \texttt{henv}(x)]\},$ where $m_i(1 \leq i \leq l) \in$ RefVar, $m_j(l \leq j \leq n)$ are variables of primitive type.

That is, the evaluation on a field nesting more than k depth returns any possible heap object $\top$(Definition 16).

The corresponding changes on the weight design is in Definition 33. The way that the field nesting is bounded as shown above is applied during the composition of abstract heap environment transformers. For later use, we name the (weight) function composition with a bound on field nesting as " $\bar{\circ}$ ".

Compared with the on-the-fly analysis, the difference of the abstraction on Jimple statements is shown in Table 5.3.

Automatic invalid path removal

For points-to analysis in the head-of-time manner, we start with analysis on an imprecise call graph, and gradually eliminate infeasible paths based on the current analysis result. More specifically, after one-time run of points-to analysis (i.e. model checking), an approximated call edge could be removed when the type of the approximated receiver object for this call edge conflicts with the expected type of this call edge. This procedure proceeds until a greatest solution is found.

Since model checking provides a ready-made facility for the fixed-point calculation, we would like to explore whether we can make full use of it. As shown from the points-to analysis in Chapter 4, it is annoying to handle various kinds of iteration demands. In particular, the situation becomes more complicated for a field-sensitive points-to analysis based on an on-the-fly call graph construction. For example, both local iterations within procedures and global iterations across procedures need to be concerned.

To keep a clear mind on the design choices of iterative procedures is arduous. Therefore, it is desired that such a job could be handed over to the back-end model checking engine. Hopefully, we expect points-to analysis with an eye on soundness can be performed in the *one-time* run weighted pushdown model checking.

In Section 4.1.3, automatic path removal is restricted by the exploded supergraph based model design, because data flow facts associated with one edge are exploded. Thus, the data flow wrt other references rather than the receiver object cannot be prohibited. The problem has been investigated more or less anyway by putting field access aside, which provides enough insights for the further analysis design. The points-to analysis based on an interprocedural CFG model design makes a real automatic path removal be in place.

Based on the interprocedural CFG, the solution for automatic path removal shares the same key idea:

A call edge will be removed if no edges reaching this invocation site possibly satisfy the type constraints on the receiver object that it declares.

Table 5.4: Modified Abstraction with Type Constraints

$x.f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$(\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{arg}_1 \mapsto \texttt{henv}(m_1), ..., \texttt{arg}_l \mapsto \texttt{henv}(m_l), \texttt{this} \mapsto \texttt{henv}(x)],\ \{(x, \epsilon, t)\})$, where $m_i(1 \leq i \leq l)$ are reference variables, $m_j(l \leq j \leq n)$ are variables of primitive type, and this method invocation occurs when the type of the object pointed to by x satisfies with t.

This idea comes true by introducing path constraints into the weight space design. The weight space S_c in Definition 34 is extended by pairing a set of *path constraints* `PathCons`, such that

$$\texttt{PathCons} \subseteq \mathbb{V} \times \mathcal{T}$$

where $\mathbb{V} = \text{RefField} \cup \text{RefVar} \times \mathcal{F}^*$.

Sometime $(v, w, t) \in c$ is denoted by $(v.w, t)$ as a matter of convenience. By a path constraint $(v.w, t) \in c$, we mean that a call edge demands the actual type of $eval(\texttt{henv}, v.w)$ to satisfy with t.

Table 5.4 shows the changes of the abstraction on Jimple statements, by pairing path constraints with previous weight function. That is, the virtual call edge is labeled with a singleton set that consists of the expected type declares by this call edge. An empty set $\emptyset$ is assigned to other translations in the interprecedural CFG as the initial type constraints.

We show the primary concerns in the design of automatic path removal by the following example.

Example 14 *Figure 5.1.3 presents an Java code example, and Figure 5.1.3 the corresponding weights for each statement on the right-hand-side. We assume that the approximated call graph is constructed by CHA. The abstract heap objects o_1, o_2, o_3 are associated with line 1, 3, and 5 respectively.*

As shown in Figure 5.1.3, there are three virtual invocations at line 2, 4 and 7 respectively. By CHA, there are invalid paths wrt these invocations in the call graph. For simplicity, it is only shown for line 2. Two possible call edges are provided by the call graph, i.e. ${e_2}^1$ and ${e_2}^2$, and we know the latter is an invalid path.

- *Judgement on invalid transitions is by path constraints*
  Considering $e_1 \otimes_e {e_2}^2$ (from line 1 to line 2), assume the abstract heap environment reaching line 1 is `henv` $\in$ *Henv, by Definition 35 and 32,*

  $$eval(e_1(\texttt{henv}), x) = o_1$$

  and `type`(o_1) = `Employee` *conflicts to the expected type* `Professor` *on x in the path constraints wrt the method* `getBonus`*.*

- *Known satisfied constraints are not handed over backwards*
  Considering $e_3 \otimes_e e_4$, the path constraint of e_4 is satisfied with the heap environment after passing line 3. To note that, before line 3, x still points to o_1.

  Since $e_1 \otimes_e (e_3 \otimes_e e_4)$ results in a valid transition, the result of $e_3 \otimes_e e_4$ should not contain the already satisfied constraints in e_4.

```
  public static void main(String[] args)
  {

    Employee x;
    Professor y;
    Employee z;

1.  x = new Employee(‘‘Tom’’, 1000);
2.  x.getBonus();

3.  x = new Professor(‘‘Jean’’, 5000);
4.  x.getBonus();

5.  y = new Professor(‘‘Ben’’, 5000);
6.  z = y;
7.  z.getBonus();

  }
```

Figure 5.2: An Example to Shown the Key Designs of Automatic Path Removal

- *Aliasing among variables are captured by tracing references in the path constraints*

  Considering $e_6 \otimes_e e_7$*, for* $\texttt{henv} \in \mathit{Henv}$*,*

  $$\mathit{eval}(e_6(\texttt{henv}), z) = \texttt{henv}(y)$$

  So the judgement on this transition is pending; and we know the type judgement on z *depends on the type of* y*. It is of course because* z *and* y *are aliased at line 6. To note that, before line 6,* z *probably does not point to an object whose type satisfies with this transition. To capture the effect of aliasing, the result of* $e_6 \otimes_e e_7$ *should declares the new constraint as* $\{y, \epsilon, \textit{\texttt{Professor}}\}$*.*

The the points-to analysis algorithm (i.e. weight space design primarily) is to capture the insights revealed by Example 14.

Lemma 1 *Let* $v \in \mathrm{RefVar} \ \cup \ \mathcal{O}$ *and* $w \in \mathcal{F}^*$*. Let* $\tau = \lambda\texttt{henv}.\ \texttt{henv} \bullet \mathrm{Map}_1 \bullet \ \ldots \ \bullet \mathrm{Map}_n \in \mathbb{F}$ *be an abstract heap environment transformer. There uniquely exists a pair of* $v' \in \mathrm{RefVar} \ \cup \ \mathcal{O}$ *and* $w' \in \mathcal{F}^*$ *such that* $\mathit{eval}(\tau(\texttt{henv}), v.w) = \mathit{eval}(\texttt{henv}, v'.w')$ *for each* $\texttt{henv} \in \texttt{Henv}$*. We denote such* $v'.w'$ *by* $\tau^{-1}(v.w)$*.*

Lemma 1 is proved by induction on n, where n is the length of a sequence of $\mathrm{Map}_1 \bullet \ldots \bullet \mathrm{Map}_n$. For instance, let

$$\tau = \lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto o_1] \bullet [y \mapsto \texttt{henv}(z)] \bullet [z \mapsto \texttt{henv}(\texttt{henv}(t).f)] \bullet [\texttt{henv}(z).f \mapsto o_2]$$

Then

$$
\boxed{\begin{array}{l}
e_1 : \ \{\lambda \texttt{henv}.\texttt{henv} \bullet [x \mapsto o_1], \emptyset\} \\[1em]
{e_2}^1 : \ \{\lambda \texttt{henv}.\texttt{henv}, \{(x, \epsilon, \texttt{Employee})\} \\[1em]
{e_2}^2 : \ \{\lambda \texttt{henv}.\texttt{henv}, \{(x, \epsilon, \texttt{Professor})\} \\[1em]
e_3 : \ \{\lambda \texttt{henv}.\texttt{henv} \bullet [x \mapsto o_2], \emptyset\} \\[1em]
e_4 : \ \{\lambda \texttt{henv}.\texttt{henv}, \{(x, \epsilon, \texttt{Professor})\} \\[1em]
e_5 : \ \{\lambda \texttt{henv}.\texttt{henv} \bullet [y \mapsto o_3], \emptyset\} \\[1em]
e_6 : \ \{\lambda \texttt{henv}.\texttt{henv} \bullet [z \mapsto \texttt{henv}(y)], \emptyset\} \\[1em]
e_7 : \ \{\lambda \texttt{henv}.\texttt{henv}, \{(z, \epsilon, \texttt{Professor})\}
\end{array}}
$$

Figure 5.3: Weights wrt the Statements in Figure 5.1.3

$$
\begin{array}{ll}
\tau^{-1}(x.\epsilon) = o_1 \qquad & \tau^{-1}(y.\epsilon) = z \\
\tau^{-1}(z.\epsilon) = t.f \qquad & \tau^{-1}(z.f) = o_2
\end{array}
$$

Definition 36 *Let $c \subseteq (\mathcal{O} \cup \mathcal{V}_l) \times \mathcal{F}^* \times P(\mathcal{T}) \cup \{\textit{error}\}$ be a path constraint and $\textit{func} \in \mathbb{F}$. $\tau = \textit{func}$, For $(v, w, t) \in c$,*

$$
\textit{func}^{-1}(\{(v, w, t)\}) = \begin{cases}
\textit{error} & \text{if } o = \tau^{-1}(v.w) \in \mathcal{O} \text{ and} \\
 & \quad \texttt{type}(o) \text{ conflicts with } t \\
 & \\
\phi & \text{if } o = \tau^{-1}(v.w) \in \mathcal{O} \text{ and} \\
 & \quad \text{type}(o) \text{ satisfies with } t \\
 & \\
\{(v', w', t)\} & \text{if } v'.w' = \tau^{-1}(v.w) \notin \mathcal{O} \\
\{(o', w', t)\} & \text{if } o'.w' = \tau^{-1}(v.w) \notin \mathcal{O}
\end{cases}
$$

and

$$
\textit{func}^{-1}(c) = \begin{cases}
\textit{error} & \text{if there exits } (v, w, t) \in c \\
 & \text{s.t. } \textit{func}^{-1}(\{(v, w, t)\}) = \textit{error} \\
\cup_{(v,w,t) \in c} \ \textit{func}^{-1}(\{(v, w, t)\}) & \text{otherwise}
\end{cases}
$$

Definition 36 captures the aforementioned primary concerns.

- The first case means that the current abstract heap environment does not satisfy with the path constraints on $v.w$.
- The second case means that a known satisfied constraint will not contribute to the result of combining these transitions.
- The last two cases say that, the judgement on path constraints is pending, and the effect of aliasing is traced by the new path constraints.

Definition 37 *We borrow notations from Definition 34.* $S_e = (D_e, \oplus_e, \otimes_e, 0_e, 1_e)$*, where*

- $D_e = \mathcal{P}(\mathbb{D})$ *where* $\mathbb{D} = \{(f, c) \mid f \in \mathbb{F}, \ \subseteq \mathcal{V} \times \mathcal{T}\}$
- $0_e = \emptyset$
- $1_e = \{\lambda \texttt{henv}.\texttt{henv}, \emptyset)\}$
- *The* $\otimes_e$ *operator is defined as* $\forall w_1, w_2 \in D_e$

$$w_1 \otimes_e w_2 = \{(\texttt{func}_1, c_1) \odot (\texttt{func}_2, c_2) \mid (\texttt{func}_1, c_1) \in w_1, (\texttt{func}_2, c_2) \in w_2\}$$

where

$$(\texttt{func}_1, c_1) \odot (\texttt{func}_2, c_2) = \begin{cases} 0_e & \textit{if } \texttt{func}_1^{-1}(c_2) = \texttt{\textit{error}} \\ (\texttt{func}_2 \ \bar{\circ} \ \texttt{func}_1, c_1 \cup \texttt{func}_1^{-1}(c_2)) & \textit{otherwise} \end{cases}$$

- *The* $\oplus_e$ *operator is defined as* $\forall w_1, w_2 \in D_e$

$$w_1 \oplus_e w_2 = w_1 \cup w_2$$

5.2 Parameterized Flow-Sensitivity by Weight Simplification

In this section, we show how to obtain parameterized flow-sensitivity by simplifying the weight design, rather than by model reduction (Section 4.2). The call-by-value Java semantics (Table 5.2) tells that, for a pointer assignment, the state of the reference variable on the left-hand side is changed by the right-hand side. Since flow-sensitivity concerns the execution order of program codes, such an "override" operation is precisely captured by a flow-sensitive analysis and compromised more or less in a flow-insensitive context. Parameterized flow-sensitivity is enabled based on the following dimensions.

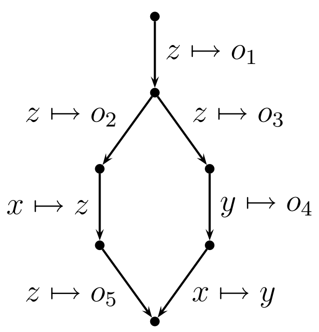
1. whether "override" operations as mentioned above are ignored.
2. whether the ordering of function application is kept among a sequence of program codes.
3. whether analyses along various control flows are distinguished.

Those dimensions are reflected on choices of the weight space design as follows.

Definition 38 *The set of abstract heap set environment is*

$$Henv^* = \{\textbf{\textit{henv}}^* \mid \textbf{\textit{henv}}^* : \mathcal{V} \to \mathcal{P}(\mathcal{O})\}$$

By reinterpreting $\bullet$ as $\cup$ for the abstract heap environment transformers from $\mathbb{F}$ (Definition 5.1) and based on the domain Henv*, "override" operations are approximated. A moderate flow-sensitive analysis (named choice 1) can be obtained.



Choices	x	z	y
choice 0	$\{o_2\}$	$\{o_5\}$	$\{o_4\}$ or
	$\{o_4\}$	$\{o_3\}$	$\{o_4\}$
choice 1	$\{o_1, o_2\}$	$\{o_1, o_2, o_5\}$	$\{o_4\}$ or
	$\{o_4\}$	$\{o_1, o_3\}$	$\{o_4\}$
choice 2	$\{o_1, o_2, o_5\}$	$\{o_1, o_2, o_5\}$	$\{o_4\}$ or
	$\{o_4\}$	$\{o_1, o_3\}$	$\{o_4\}$
choice 3	$\{o_1, o_2, o_4, o_5\}$	$\{o_1, o_2, o_3, o_5\}$	$\{o_4\}$
choice 4	$\{o_1, o_2, o_3, o_4, o_5\}$	$\{o_1, o_2, o_3, o_5\}$	$\{o_4\}$

Figure 5.4: A Comparison on the Precision of Variations on Flow-Sensitivity

Definition 39 *Let $\mathbb{F}^*$ be the new set of abstract heap environment transformers. Define a function $\bowtie$: $\mathbb{F}^* \times \mathbb{F}^* \to \mathbb{F}^*$, such that $\forall \boldsymbol{henv}^* \in Henv^*$*

$$f_1 \bowtie f_2 = \lambda \boldsymbol{henv}^*.(f_1 \circ f_2(\boldsymbol{henv}^*) + f_2 \circ f_1(\boldsymbol{henv}^*))$$

By $\bowtie$, the ordering of function application among two functions from $\mathbb{F}^*$ is loosened. A further flow-insensitive analysis (named choice 2) is obtained, by redefining the $\otimes$ operation in choice 1 as

$$w_1 \otimes w_2 = \{f_1 \bowtie f_2 \mid f_1 \in w_1, f_2 \in w_2\}$$

The analysis in Definition 34 is based on the powerset construction. That is, $\oplus$ keeps all possibilities along each control flow. A thorough flow-sensitive analysis do not distinguish various control flows (paths in CFGs). A straight forward way towards this is to combine the analysis results of different paths, rather than a powerset construction for $\oplus$, defined as follows(named choice 3).

Definition 40 *A bounded idempotent semiring $S_c^* = (D_c^*, \oplus_c^*, \otimes_c^*, 0_c^*, 1_c^*)$ is*

- *The weight space is $D_c^* = \mathbb{F}^*$*
- $0_c^* = \text{ZERO}$
- $1_c^* = \lambda \boldsymbol{henv}^*.\boldsymbol{henv}^*$
- $\forall w_1, w_2 \in D_c^* \setminus \{\text{ZERO}\},\ w_1 \otimes_c^* w_2 = w_1 \bowtie w_2$
  $\forall w \in D_c^*,\ \text{ZERO} \otimes_c^* w = w \otimes_c^* \text{ZERO} = \text{ZERO}$
- $\forall w_1, w_2 \in D_c^* \setminus \{\text{ZERO}\},\ w_1 \oplus_c^* w_2 = \lambda \boldsymbol{henv}^*.w_1(\boldsymbol{henv}^*) \cup w_2(\boldsymbol{henv}^*)$
  $\forall w \in D_c^*,\ \text{ZERO} \oplus_c^* w = w \oplus_c^* \text{ZERO} = w$

An almost flow-insensitive analysis can be obtained (named choice 4), if $\oplus$ is further considered as

$$\forall w_1, w_2 \in D_c^* \setminus \{\text{ZERO}\},\ w_1 \oplus w_2 = w_1 \bowtie w_2$$

A comparison on the precision of variations on flow-sensitivity based on different choices is shown in Table 5.4, where choice 0 is the original flow-sensitive analysis presented in section 5.1.3.

5.3 Prototype Implementation

5.3.1 Prototype Framework

We use the basically same prototype framework to implement points-to analysis algorithms presented in this chapter, as shown in 5.5. There are generally three phases:

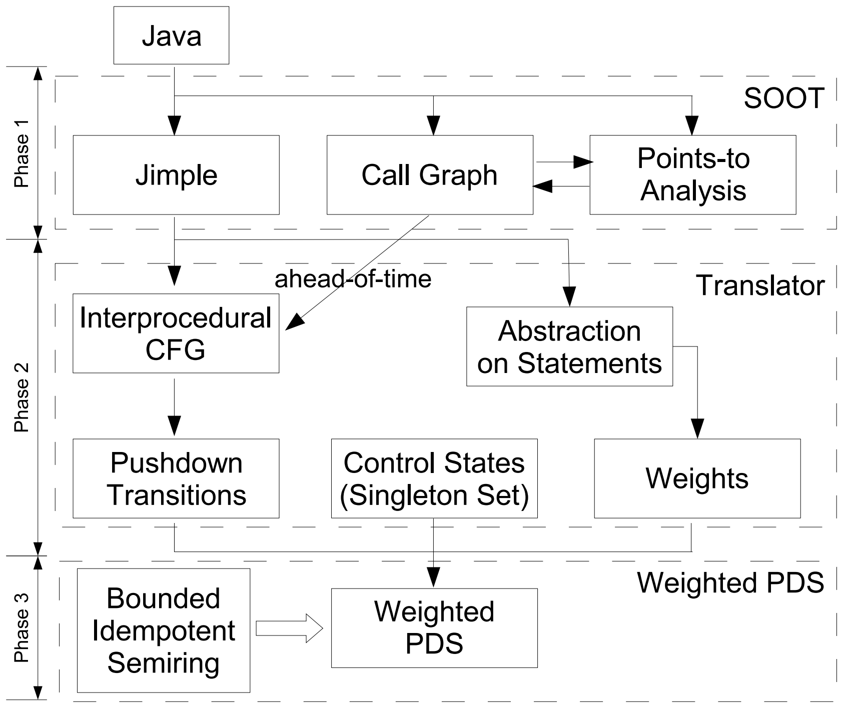


Figure 5.5: A Prototype Framework for Points-to Analysis

- In *phase 1*, the SOOT [41] compiler is explored as the frontend for preprocessing from Java programs to Jimple codes.
- In *phase 2*, the model abstraction part translates Jimple codes to weighted pushdown systems. it takes around 1000 lines of Java codes for the CFG based model designs for either on-the-fly (Section 5.1.2) or ahead-of-time construction (Section 5.1.3).
- In *phase 3*, Weighted PDS library is explored as the back-end model checking engine, which calls an implementation of semiring designs. for the CFG based model design, it takes around 1400 lines of C codes for on-the-fly construction and 2400 lines for the ahead-of-time construction.

Our implementation strategy is to apply the existing ready-made tools. Unfortunately, as shown in Chapter 4.3.2, our evaluation is restricted by the interaction among SOOT compiler and the backend model checker. Concurrently, our current implementation only works on small examples. When the existing tools are ready for a proper evaluation, we think, there are much rooms to improve it in the future.

Furthermore, the weighted domain easily grows exponentially. For instance, the weight space in the ahead-of-time analysis (Section 5.1.3) is quite heavy. Whereas, weight operations in the algorithms presented here are naturally characterized as set operations (Section 5.3.2). We suppose there will be promising improvement by applying efficient data structures to the implementation, such as BDD.

5.3.2 Implementation Aspects

Previously, we present points-to analysis algorithms based on CFG-based model design, in which transformers on the abstract heap environment are formulated as the weighted domain. These static analysis are implemented with weight elements characterized as ordered sets, i.e. the ordered set of points-to relation $\mathbb{L} = \texttt{PointsTo}^*$, where `PointsTo` is defined as

$$\texttt{PointsTo} : \mathbb{V} \times (\mathbb{V} \cup \mathcal{O})$$

and $\mathbb{V} = \texttt{RefField} \cup (\texttt{RefVar} \times \mathcal{F}^*)$. We will still use $\mapsto$ to denote the points-to relation `PointsTo`. Let $\texttt{len} : \mathbb{L} \to \mathbb{N}$ be the function that returns the length of a ordered set. For $l \in \mathbb{L}$, $l(i) \in l$ $(1 \leq i \leq len(i))$ denotes the i^{th} element of the ordered set l. For simplicity, we sometime use $[(vf_1, vt_1)...(vf_n, vt_n)]$ to denote the ordered set when no confusion exists.

For efficiency on time and space, it is further required that $\forall\ l \in \mathbb{L}$,

$$\forall\ (vf_1, vt_1), (vf_2, vt_2) \in l,\ vt_1 = vt_2 \text{ if } vf_1 = vf_2$$

It reads that: each reference variable can possibly have an unique target heap object anytime along one run of the program.

In Section 5.1.1, we give the abstraction on Java language model wrt references by exactly following the Java semantic. The key of the implementation is:

The evaluation of some reference on the abstract heap environment is captured as computing *backward* transitive closure on the set relation.

Example 15 *By backward, we mean the ordering of the set (i.e. flow-sensitivity) matters. For instance, the sequence of statements*

```
x = o1;
y = x;
x = o2;
```

results in the points-to sets of $\{x \mapsto o_2, y \mapsto o_1\}$*, which can be interpreted as*

$$\{x \mapsto o_1\} \otimes \{y \mapsto x\} \xlongequal{\{x' \mapsto o_1\} \otimes \{y \mapsto x', x \mapsto x'\}} \{y \mapsto o_1, x \mapsto o_1\}$$
$$\{y \mapsto x\} \otimes \{x \mapsto o_2\} \xlongequal{\{y' \mapsto x''\} \otimes \{x \mapsto o_2, y \mapsto y'\}} \{x \mapsto o_2, y \mapsto x\}$$

Variables with primes (') attached to denotes those from the previous states.

Definition 41 *Define a function* $\textit{\textbf{BoundedCat}} : \mathbb{V} \times \mathcal{F}^* \to \mathbb{V} \cup \{\textit{\textbf{void}}\}$*, such that for* $\nu.\omega \in \mathbb{V}$*,* $\gamma \in \mathcal{F}^*$*,*

$$\textit{\textbf{BoundedCat}}(v_1, v_2) = \begin{cases} \textit{\textbf{void}} & \textit{if } v_1 = \textit{\textbf{void}} \\ \textit{\textbf{void}} & \textit{if } |\omega \cdot \gamma| > k \\ \nu.\omega \cdot \gamma & \textit{else} \end{cases}$$

In Definition 41, a new variable `void` is first introduced. `void` is a special reference that points to objects of all possible types. Thus, it has the most general type and exactly corresponds to $\top$. The introduction of `void` is not abrupt. This idea is also important in real programming languages, such as the `Object` class type in Java and the `void *` pointer type in C.

Figure 5.6 gives the iterative algorithm for computing the backward transitive closure. It can be regarded as the corresponding implementation for Definition 35. The situation is a bit complicated when field access is involved.

Example 16 *This example intends to show the concerns behind the the algorithm design in Figure 5.6.*

```
0. x = o1;        (x, o1)
1. x.f = o2;      (x.f, o2)
2. x = o3;        (x, o3)
3. y = x.f;       (y, x.f)
```

By examining the above Java code sequence, `(x.f, o2)` $\otimes$ `(x, o3)` *results in* $w = \{$`(x.f, o2)`, `(x, o3)`$\}$ *by implementation, and the analysis of* $w \otimes$ `(y, x.f)` *brings questions that whether the result of analysis is* $\{$`(y, o2)`$\}$.

The answer is no. By `x.f` *after line 2, we mean the current field* `f` *of* `o3` *rather than* `o1` *that* `x` *previously pointed to. Therefore, the analysis results in* $\{$`(y, o3.f)`$\}$ *by implementation.*

This simple example tells that the computation of transitive closure by such a "substitution" should be performed incrementally, with the growth of the field nestings. It also exactly corresponds to the previous formulation of the analysis based on the function application.

Definition 42 *A bounded idempotent semiring* $S_c = (D_c, \oplus_c, \otimes_c, 0_c, 1_c)$ *is defined as*

- $D_c = \mathcal{P}(\mathbb{D})$, *where* $\mathbb{D} = \mathbb{L} \cup \{\text{ID}\} \setminus \emptyset$
- $0_c = \emptyset$
- $1_c = \{\text{ID}\}$, *where* $\text{ID} = \lambda x.x$
- $\forall w_1, w_2 \in D_c$, $w_1 \otimes_c w_2 = \{l_1 \odot_c l_2 \mid l_1 \in w_1, l_2 \in w_2\}$, *such that*

$$l_1 \odot_c l_2 = \begin{cases} l_1 \ (\textit{resp. } l_2) & \textit{if } l_2 = \text{ID} \ (\textit{resp. } l_1 = \text{ID}) \\ \boldsymbol{cat}(f_5(l_1, \mathbb{l}_2), \mathbb{l}_2) & o.w. \end{cases}$$

where $\mathbb{l}_2 = f_4(f_3(f_2(l_1, l_2)))$

```
BackwardTrans(v, l)
Input: $v \in \mathbb{V}$, $l \in \mathbb{L}$
Output: $tmpv$, the current evaluation on $v$ wrt $l$
1.  $k = 1$
2.  while(k $\leq$ len($v$))
3.  begin
4.      $subv_1 = \texttt{SubString}(v, 1, k)$
5.      $subv_2 = \texttt{SubString}(v, k+1, |v|)$
6.      if $(\exists(subv_1, v') \in l)$
7.      begin
8.          tmpv $= \texttt{BoundedCat}(v', subv_2)$
9.          if $(tmpv == \texttt{void})$
10.         begin
11.             return $tmpv$
12.         end
13.     end
14.     else
15.     begin
16.         return $tmpv$
17.     end
18.     $k = k + 1$
19. end
```

Figure 5.6: The Algorithm of Computing Backward Transitive Closure

$f_2(l_1, l_2) = [\ (\mathtt{eval}(v), \mathtt{eval}(h))\ ]$
where for $(v, h) \in l_2$,

$$\mathtt{eval}(v) = \begin{cases} v & \text{if } v \in RefVar \cup \{\mathtt{void}\} \\ \mathtt{BackwardTrans}(v, l_1) & \text{otherwise} \end{cases}$$
$$\mathtt{eval}(h) = \begin{cases} h & \text{if } h \in \mathcal{O} \cup \{\mathtt{void}\} \\ \mathtt{BackwardTrans}(h, l_1) & \text{otherwise} \end{cases}$$

$$f_3(l) = [\ l(i) = (v, h), 1 \le i \le \mathtt{len}(l) \mid \forall i < j \le \mathtt{len}(l), not\ \exists h' \in \mathbb{V} \cup \mathcal{O},\ s.t.\ l(j) = (v, h')\ ]$$
$$f_4 = [\ (v, \hat{h}) \mid \forall 1 \le i \le \mathtt{len}(l), l(i) = (v, h),\ \hat{h} = \begin{cases} h' & \text{if } \exists j < i, l(j) = (v', h'),\ s.t.\ v' = h \\ h & o.w. \end{cases}\ ]$$

$$f_5(l_1, l_2) = [\ (v_1, h_1) \in l_1 \mid \forall (v_2, h_2) \in l_2,\ s.t.\ v_1 \neq v_2\ ]$$

$\mathtt{cat} : \mathbb{L}^* \times \mathbb{L}^* \to \mathbb{L}^*$ *is a function that concatenates two ordered lists into one.*

- $\forall w_1, w_2 \in \mathscr{S}$, $w_1 \oplus_c w_2 = w_1 \cup w_2$

NOTE 1. f_5 overwrites the previous state of references by the next step points-to facts, which intend to ensure that a reference only possibly points to some unique heap object for among each run. But for this purpose, only f_5 is not enough, because this will be broken when the aliasing among references is known later (see Example 17). That is why f_3 and f_4 introduced.

NOTE 2. However, f_3 and f_4 are optimal. The choice on them is rather tradeoff among the efficiency and precision (instead of soundness). If f_3 and f_4 are omitted from the algorithm, the only difference is that a further step is needed to get the points-to facts after weighted pushdown model checking ends. The job is exactly what • says in Definition 32, also shown in Example 17.

Example 17 *The following two Java code sequences respectively show the typical occasions, on which the aliasing matters the way of analysis.*

```
1. x = y;        (x, y)
2. x.f = o1;     (x.f, o1)
3. y.f = o2;     (y.f, o2)
```

```
1. x = y;        (x, y)
2. y.f = o1;     (y.f, o2)
3. z = x.f;      (z, x.f)
```

The problem occurs when analyzing the program backwards, thus the aliasing among x *and* y *is known later after the analysis passing 2 and 3. For the first case,* $[(y.f, o_1), (y.f, o_2)]$ *happens, and for the second one,* $[(x.f, o_1), (z, x.f)]$ *happens. By handling these occasions earlier by* f_3 *and* f_4*, some space is saved and some analysis time is cost.*

For automatic path removal, path constraints is also implemented by characterized as set operations, defined as

$$\mathscr{C} \subseteq \mathbb{V} \times \mathcal{T}$$

The algorithm design for implementing the semiring is given in Definition 43.

Definition 43 *S in Definition 34 is extended to be $S_e = (D_e, \oplus_e, \otimes_e, 0_e, 1_e)$, where*

- $D_e = \mathcal{P}(\mathscr{D})$, *where* $\mathscr{D} = \{(d, c) \mid d \in \mathbb{D},\ c \in \mathscr{C}\}$
- $1_e = \{(\textsc{id}, \emptyset)\}$
- $0_e = \emptyset$
- $\forall w_1, w_2 \in D_e,\ w_1 \otimes_e w_2 = \{\mathbb{d}_1 \odot_e \mathbb{d}_2 \mid \mathbb{d}_1 \in w_1, \mathbb{d}_2 \in w_2\}$, *such that* $\forall \mathbb{d}_1 = (d_1, c_1), \mathbb{d}_2 = (d_2, c_2) \in \mathscr{D}$,

$$\mathbb{d}_1 \odot_e \mathbb{d}_2 = \begin{cases} 0_e & \textit{if } \mathbb{c}_2 \propto d_1 \\ (d_1 \odot_c d_2, c_1 \uplus \mathbb{c}_2) & \textit{o.w.} \end{cases}$$

where $c_1 \uplus \mathbb{c}_2 = c_1 \cup f_8(\mathbb{c}_2 \setminus c, d_1)$, *and* $c = f_7(\mathbb{c}_2, d_1)$.

$$\mathbb{c}_2 = \{(\hat{v}, t) \mid (v, t) \in c_2,\ \textit{s.t. } \hat{v} = \begin{cases} v & \textit{if } v \in \texttt{RefVar} \cup \{\texttt{void}\} \\ \texttt{BackwardTrans}(v, d_1) & \textit{otherwise} \end{cases}$$

$$\textit{and } \hat{v} \neq \texttt{void}\}$$

$\forall c \in \mathscr{C}, d \in \mathscr{D}$,

$$f_7(c, d) = \{(v, t) \in c \mid \exists (v, o) \in d,\ t' = \text{type}(o),\ \textit{s.t. } t' \rtimes t \textit{ or } \exists (v, \top) \in d\}$$

$$f_8(c, d) = \{(\tilde{v}, t) \mid \forall (v, t) \in c, \tilde{v} = \begin{cases} v' & \textit{if } \exists (v, v') \in d, v' \in \mathcal{V} \\ v & \textit{o.w.} \end{cases}\}$$

- $\propto: \mathscr{C} \times \mathscr{D} \to \{\textbf{true}, \textbf{false}\}$ *is introduced as an judgement relation. That is,* $\forall d \in \mathscr{D}, c \in \mathscr{C},\ c \propto d$ *iff* $\exists (v, t) \in c$, *and* $(v, o) \in d$, *such that* $t' \ltimes t$, *where* $t' = \text{type}(o)$.
- $\ltimes : \mathcal{T} \times \mathcal{T} \to \{\text{true}, \text{false}\}$ *defines a relation among classes.*
  $\forall t, t' \in \mathcal{T},\ t' \ltimes t$ *iff*

  *r*1. $t' \neq t$

  *r*2. *a)* t' *does not inherit from* t*; or*

  b) t' *inherits from* t*, but* t' *redefines the method to be invoked.*
- $\rtimes$ *is defined as the reverse of* $\ltimes$*. That is,*

$$\forall t, t' \in \mathcal{T},\ t' \rtimes t \textit{ iff } t' \ltimes t = \textbf{false}$$

- $\forall w_1, w_2 \in D_e,\ w_1 \oplus_e w_2 = w_1 \cup w_2$.

Chapter 6

Interprocedural Irrelevant Code Elimination

Classic data flow analysis play a crucial role in program analysis and are always stating point for new methodologies. The crucial connection among program analysis, model checking, and abstract interpretation is revealed based on the study of bit-vector analysis. This chapter is dedicated to exploring an interprocedural irrelevant code elimination analysis, under PER (partial equivalence relation) based abstraction. The irrelevant code analysis originates from dead code analysis, but more on a semantical sense. Our primary motives are:

- The bit-vector analysis is traditionally intraprocedural, as well as those based on finite model checking engines. Whereas, even based on infinite model checking such as pushdown model checking, the analysis only talks about global variables.
- As previously explored, weighted pushdown model checking provides a general framework for interprocedural data flow analysis. We would like to explore the difference between pushdown and weighted pushdown model checking from the study of a classic data flow analysis.

The conventional approach to dead code elimination is based on live variable analysis (Section 1.1.2). Thus, a line of program code is dead if

- this code is an assignment statement; and
- the variable that is assigned by this code is not live after this line.

Based on model checking, a dead code detection usually takes the *used-and-defined* approach [32]. That is, some variable x of interest is evaluated with predicates "Used_x" and "$\mathit{Defined}_x$". Whether the transition system violates the property of " $!\mathit{Used}_x$ $\mathbf{W}$ $\mathit{Defined}_x$ " ($\mathbf{W}$ is weak until) is model checked for all possible transition sequences.

Based on the conventional gen/kill functions, an example for live variable analysis is given in WPDS++ library. The weighted domain is defined as $D = \{\lambda S.S \setminus \mathit{KillSet}(i) \cup \mathit{GenSet}(i) \mid i \in \mathbf{N}\}$, where S is the finite set of variable alphabet, and $\mathbf{N}$ is labels of all the program statements. A dead code elimination can be done based on the result.

However, these analysis or case studies are either intraprocedural, or partially interprocedural, since interactions among procedures are not captured by their constructions. Thus, these approaches are suitable for analyzing global variables in essence.

In this chapter, we will explore how pushdown and weighted pushdown model checking works on dead code elimination, following to the used-and-defined approach. Then, an interprocedural irrelevant code elimination is proposed based weighted pushdown model checking, under PER based abstraction.

This Chapter is organized as follows:

In Section 6.1, solutions for an interprecedural dead code elimination are presented by both pushdown and weighted pushdown model checking. Under such a choice, the "interprecedural" flavor is obtained by explicit scope management on variables, such as global renaming, in the abstraction phase. These analysis are partial interprocedural, because the interaction among procedures are not captured.

In Section 6.2, we propose an interprocedural irrelevant code elimination based on weighted pushdown model checking, under PER based abstraction. The underlying model for model checking is the exploded supergraph. The interaction among procedures are captured with taking into account parameter passing and return values.

We also implement the algorithm in Section 6.2 within a prototype framework, presented in Section 6.3. Our implementation exploits SOOT as Java preprocessing and the Weighted PDS library as the back-end model checking engine. The *call graph generation* and *pointer-to analysis* facilities are borrowed from SOOT to handle virtual method calls and the aliasing among references.

6.1 Interprocedural Dead Code Elimination

Throughout this chapter, we will use Example 18 as a running example. Compared with points-to analysis, we limit our focus on data type of Boolean and Integers only.

Example 18 *Figure 6.1 and 6.2 present Java programs with three classes:* `Example`*,* `Call`*, and* `CallSuper`*. The class* `Call` *inherits the class* `CallSuper` *and redefines the* `call` *method for calculating the factorial of some integer* `a`*, with a dead parameter* `b`*.*

In class `Example`*, there is a virtual call at line 13. The method call from class* `Call` *should be invoked at runtime.*

There is also a method `example` *in the class* `Example`*, in which both line 21 and line 22 are irrelevant codes.*

In this section, following the *used-and-defined* approach, we explore how to perform an interprocedural dead code elimination by both pushdown model checking and weighted pushdown model checking. Let the set of atomic propositions be

$$AP = \{\mathtt{Def}_x, \mathtt{Use}_x\}$$

where x is any variable.

6.1.1 LTL Pushdown Model Checking with Simple Valuations

Based on pushdown model checking, the problem is straightforward by following the automata-theoretic approach. Recall that, by pushdown model checking on properties in LTL with simple valuations, the assignment of atomic propositions only depends on the control location and the topmost stack symbol.

A straightforward way of encoding programs into pushdown systems is:

```java
class CallSuper
{
  public int call(int a, int b)
  {
    int c;
    if(a <= 0)
      c = 1;
    else
      c = a * call(b-1, a-1)
    return c;
  }
}
```

```java
class Call extends CallSuper
{
  public int call(int a, int b)
  {
    int c;
    if(a == 0)
      c = 1;
    else
      c = a * call(a-1, b-1)
    return c;
  }
}
```

Figure 6.1: Factorial Calculation in Java

- the set of control states is a singleton set $\{\cdot\}$;
- the set of stack alphabet is the product of (either global or local) variables and program points, i.e. line numbers.
- the transition set is constructed from the control flow graph by rules presented in Chapter 2.
- the initial stack content is the program's entry point.

Obviously, it is demanded that the domain of variables is abstracted to be finite; and also global variables can be encoded as part of the control locations.

Example 19 *By examining the method* ***example*** *in Figure 6.2, its pushdown transitions are as follows, and the corresponding transition graph is in Figure 6.3.*

$$\begin{array}{l}
\langle\cdot,(x=1,y=0,l_{21})\rangle \hookrightarrow \langle\cdot,(x=1,y=2,l_{22})\rangle \\
\langle\cdot,(x=1,y=2,l_{22})\rangle \hookrightarrow \langle\cdot,(x=3,y=2,l_{23})\rangle \\
\langle\cdot,(x=3,y=2,l_{23})\rangle \hookrightarrow \langle\cdot,(x=3,y=6,l_{24})\rangle \\
\langle\cdot,(x=3,y=6,l_{24})\rangle \hookrightarrow \langle\cdot,\varepsilon\rangle
\end{array}$$

By examining labeling of atomic propositions wrt x in Figure 6.3, The used-and-defined approach will conclude that x is always live, i.e. no dead code wrt x. By examining y the same way, y is not used between two definition at line 22 and line 24. The used-and-defined approach will conclude that l_{22} is a dead code.

However, we can find x is also dead after removing line 22. But The used-and-defined approach cannot detects this propagation directly.

6.1.2 Weighted Pushdown Model Checking

Since weighted pushdown model checking does not follow the automata-theoretic approach, the judgement on dead codes need to be captured by the weight space design. As will be seen in Chapter 7, we will discuss the relation between pushdown model checking on regular languages and weighted pushdown model checking.

```
class Example
{
   public static int f;

   public static void main(String[] args)
   {
      int y = 0;
      int x = 4;
      f = 5;
      Call c = new Call();
      CallSuper cs = new CallSuper();
      cs = c;
      y = cs.call(f, x);
      System.out.println(y);
   }

   public static void example()
   {
      int x;
      int y;
      x = 1;
      y = x + 1;
      x = 3;
      y = x + 3;
   }
}
```

Figure 6.2: An Example on Factorial Calculation

A variable is dead if it is not used among some two definitions on it. By exactly capturing this idea, the algorithm for dead code detection based on weighted pushdown model checking is designed.

Definition 44 *A bounded idempotent semiring $S = (D, \otimes, \oplus, 0, 1)$ is defined as:*

1. *Weighted domain D is defined as*

   $$D = \{\lambda x.x, \lambda x.\mathtt{Use}_v, \lambda x.\mathtt{Def}_v, \textsc{dead}, \textsc{zero} \ \mid x \in L\}$$

   with the ordering that

   $$\textsc{dead} \sqsubseteq \lambda x.\mathtt{Def}_v \sqsubseteq \lambda x.x \sqsubseteq \lambda x.\mathtt{Use}_v \sqsubseteq \textsc{zero}$$

   where ZERO *is naturally interpreted as that the program execution is interrupted by an error, and the control flow is prevented. The newly-introduced element* DEAD *represents dead code is found along some control flow.*

2. *1 is defined as $id = \lambda x.x$*

3. *0 is defined as* ZERO

$$\begin{tikzcd}
\langle \cdot, (x=1, y=0, l_{21}) \rangle \arrow[d] & \texttt{Def}_x & \texttt{true} \\
\langle \cdot, (x=1, y=2, l_{22}) \rangle \arrow[d] & \texttt{Use}_x & \texttt{Def}_y \\
\langle \cdot, (x=3, y=2, l_{23}) \rangle \arrow[d] & \texttt{Def}_x & \texttt{true} \\
\langle \cdot, (x=3, y=6, l_{24}) \rangle \arrow[d] & \texttt{Use}_x & \texttt{Def}_y \\
\langle \cdot, \varepsilon \rangle & \texttt{true} & \texttt{true}
\end{tikzcd}$$

Figure 6.3: The Transition Graph of `method` as Pushdown Systems

4. *The $\otimes$ operator composes the effects by transfer functions along one path. The operation $\otimes$ on D is defined as*

$$\forall d \in D,\ \text{ZERO} \otimes d = d \otimes \text{ZERO} = \text{ZERO}$$

$$\forall d \in D,\ \lambda x.x \otimes d = d \otimes \lambda x.x = d$$

$$\forall d \in D \setminus \{\text{ZERO}\}, d \otimes \text{DEAD} = \text{DEAD} \otimes d = \text{DEAD}$$

$$\lambda x.\texttt{Def}_v \otimes \lambda x.\texttt{Use}_v = \lambda x.\texttt{Use}_v$$

$$\lambda x.\texttt{Use}_v \otimes \lambda x.\texttt{Def}_v = \lambda x.\texttt{Def}_v$$

$$\lambda x.\texttt{Use}_v \otimes \lambda x.\texttt{Use}_v = \lambda x.\texttt{Use}_v$$

$$\lambda x.\texttt{Def}_v \otimes \lambda x.\texttt{Def}_v = \text{DEAD}$$

5. *The $\oplus$ operator combines effects on the property domain by transfer functions from different branches. The operation $\oplus$ on D is defined as*

$$\forall d \in D,\ \text{ZERO} \oplus d = d \oplus \text{ZERO} = d$$

$$\forall d \in D,\ \text{DEAD} \oplus d = d \oplus \text{DEAD} = \text{DEAD}$$

$$\forall d \in D,\ \lambda x.\texttt{Def}_v \oplus d = d \oplus \lambda x.\texttt{Def}_v = \lambda x.\texttt{Def}_v$$

$$\lambda x.\texttt{Use}_v \oplus \lambda x.\texttt{Use}_v = \lambda x.\texttt{Use}_v$$

$$\lambda x.x \oplus \lambda x.x = \lambda x.x$$

$$\lambda x.\texttt{Use}_v \oplus \lambda x.x = \lambda x.x \oplus \lambda x.\texttt{Use}_v = \lambda x.x$$

By the weight design in Definition 44, the dead code detection works as follows:

These exists dead codes if the result of weighted pushdown model checking is DEAD, and the paths that dead codes are involved in are provided as the witness set.

Due to follow the used-and-defined approach, the above solution also cannot detect x is dead at line 21 until line 22 is removed.

By looking at the solutions in Section 6.1.1 and 6.1.2, it is easy to see that whether the analysis is interprocedural or intraprocedural is independent of the algorithms. The interprocedural flavor can be thus obtained by renaming variables with preprocessing in the abstraction phase.

However, such an analysis is also not fully interprocedural, because the interaction among procedures are still not captured. For example, when variables are passed as parameters to some procedures, these variables are basically considered as "*Used*", regardless of its effect on the result of computation. Thus, these solutions are more fitted to an interprocedual dead code elimination analysis concerning only global variables.

6.2 Interprocedural Irrelevant Code Elimination

In this section, we explore an interprocedural irrelevant code elimination analysis. Compared with dead code detection, it concerns more on a semantical sense. The basic intent behind is straightforward: *a line of code is dead as long as its removal does not affect the final result of interest.* Instead of *used-and-defined* approach, We apply PER-based abstraction [33] as a forward abstract intepretation [24]. This approach naturally detects transitivity of dead codes. For instance, this solution directly detects line 21 in the Figure 6.2 is a dead code. While the *used-and-defined* based approach cannot detect it unless line 22 is removed.

6.2.1 Abstraction from Java programs to Pushdown Systems

Before abstraction, control flow graphs (CFG) are first prepared for each procedure. However, it is not easy to get a precise interprcedural control flow graph (or supergraph) for Java, due to polymorphism and dynamic binding of virtual method calls, as thoroughly explored in the previous chapters. At the first stage, we make use of the results of *call graph* generated by SOOT under the help of points-to analysis. Call graph is a set of resolved call edges among procedures.

Example 20 *In Figure 6.2 of Example 18, reference variable* `cs` *may point to instances of either class* `CallSuper` *or* `Call`*. At the virtual call site of line 13,* `cs` *will invoke the method* `call` *from the class* `Call` *instead of its declared type class* CallSuper*.*

Figure 6.4 shows part of the supergraph of the Java program in Example 18. Compared with CFGs from intraprocedural cases, three more kinds of edges are added (Section 2.4.2). Local variables in the calling procedure keep unchanged and can take a short-cut over the call-to-return edge.

In an interprocedural case, a problem is how to abstract interactions among procedures through parameter passings and return values. We use an *abstract parameter passing* mechanism to handle these interactions, by introducing two extra kinds of global variables for procedure parameters and return values respectively. In particular, variables for procedure parameters are characterized by their positions declared in the procedure apart from names.

This approach is shown in Figure 6.4 with regard to Example 18, in which only the static class member f is "global". To depict the interactions among procedures, two extra kinds of global variables are introduced: the integer parameter variables **call_arg0**, **call_arg1** for the method *call* and the integer return variable **r_int** . The introduction of parameter and return variables correctly depict the localness of local variables. Whenever a procedure is invoked, the corresponding procedure variables are assigned if they exist. Whenever a procedure invocation returns with some non-void value, the global return variable with coincident type is assigned. Local variables within one procedure are always unseen to others.

Each edge in the supergraph is labelled with a transfer function on program states. i.e. usually a set of mappings from program variables to some abstract data domain. In Figure 6.4 (b), *top* could be understood as non-exist values and used for later generation of the *exploded supergraph*. Please refer to [37] for formal definitions. Some treatments on transfer functions need to be mentioned here:

- When a procedure is invoked (*call edge*), all local variables in the caller procedure are assigned to *top*.
- When an invoked procedure returns(*return edge*), all local variables in the callee procedure are assigned to *top*.
- All global variables are assigned to *top* along *call to return edge*.

Although Java takes call-by-value mechanism, the state of an object can still be implicitly changed due to aliasing. To further identify the aliasing among object reference variables, pointer-to analysis in SOOT is borrowed for simplicity.

To capture the dependency among variables, we exploit the exploded supergraph as the underlined model. An edge in the supergraph is exploded into a set of edges with every variables as ends. Table 6.1 shows the basic point-wise representation of the language syntax that we work on, i.e. the way of how each edge is exploded. In this analysis, our target is primitive types of Java, i.e. Numbers and Boolean values.

In Table 6.1, `a`, `b` are constant, `x`, `y` are variables of Numbers or Boolean, and `op` denotes an binary arithmetic operator. To note that, whether `x` is dead or not depends on both `y` and `z` for the third case. In particular, a variable is assigned to *top* will not contribute to the exploded supergraph.

Provided with an exploded supergraph G of the program to be analyzed, the encoding of the pushdown system for our running example is:

- All program variables (global variables and local variables) are encoded as the set of control states.
- Program control points are encoded as stack symbols.
- Each edge in G is encoded as a pushdown rule according to the following cases:
  - $\langle q, w_i \rangle \hookrightarrow \langle q', w_k\ w_j \rangle$
    A call edge from node w_i to w_k with w_j as return point.
  - $\langle q, w_i \rangle \hookrightarrow \langle q', w_j \rangle$
    An intraprocedural edge from node w_i to w_j.
  - $\langle q, w_i \rangle \hookrightarrow \langle q', \varepsilon \rangle$
    A return edge from exit node w_i to corresponding return points.

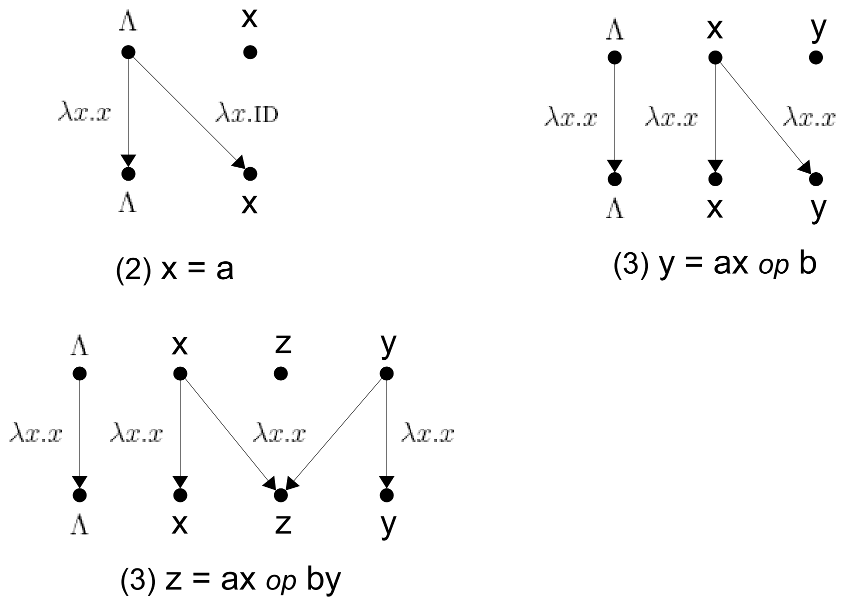


Table 6.1: A Point-wise Representation of Pointer Assignments

6.2.2 PER based Data Abstraction

A *partial equivalence relation* R on a set S is a transitive and symmetric relation $S \times S$. If R is reflexive, it is an equivalence relation. Our abstract data domain L is a 2 point domain based on PER [33][1], defined as

$$L = \{\textsc{any},\ \textsc{id}\}, \quad \text{with the ordering } \ \textsc{any} \supset \textsc{id}$$

With the original domain as integer $\mathbf{Z}$, the concretisation γ of L is defined as

$$\gamma\ \textsc{any} = \{(x, y) \mid x, y \in \mathbf{Z}\}$$

$$\gamma\ \textsc{id} = \{(x, x) \mid x \in \mathbf{Z}\}$$

Where ANY is interpreted as anything, and ID is interpreted as values being fixed. It is easy to see that

$$\forall l \in L,\ \gamma\ l \text{ is a PER}$$

A finite set of *transfer functions* $\mathcal{F} :\ \mathcal{L} \rightarrow \mathcal{L}$ is defined as:

$$\mathcal{F} = \{\lambda x.x, \lambda x.\textsc{any}, \lambda x.\textsc{id} \ \mid x \in L\}$$

Let $f_0 = \lambda x.\textsc{any}$, $f_1 = \lambda x.x$, and $f_2 = \lambda x.\textsc{id}$, it is obvious that

$$\forall x \in L,\ f_0\ x \supset f_1\ x \text{ and } f_1\ x \supset f_2\ x$$

We define an ordering $\sqsubset$ on $\mathcal{F}$ as the reverse of $\supset$ as

$$\lambda x.\textsc{any} \sqsubset \lambda x.x \sqsubset \lambda x.\textsc{id}$$

[1] [11] uses a 3 point abstract domain {ANY,ID,BOT}. Since our focus is on "irrelevance" not on "strictness", BOT is left out.

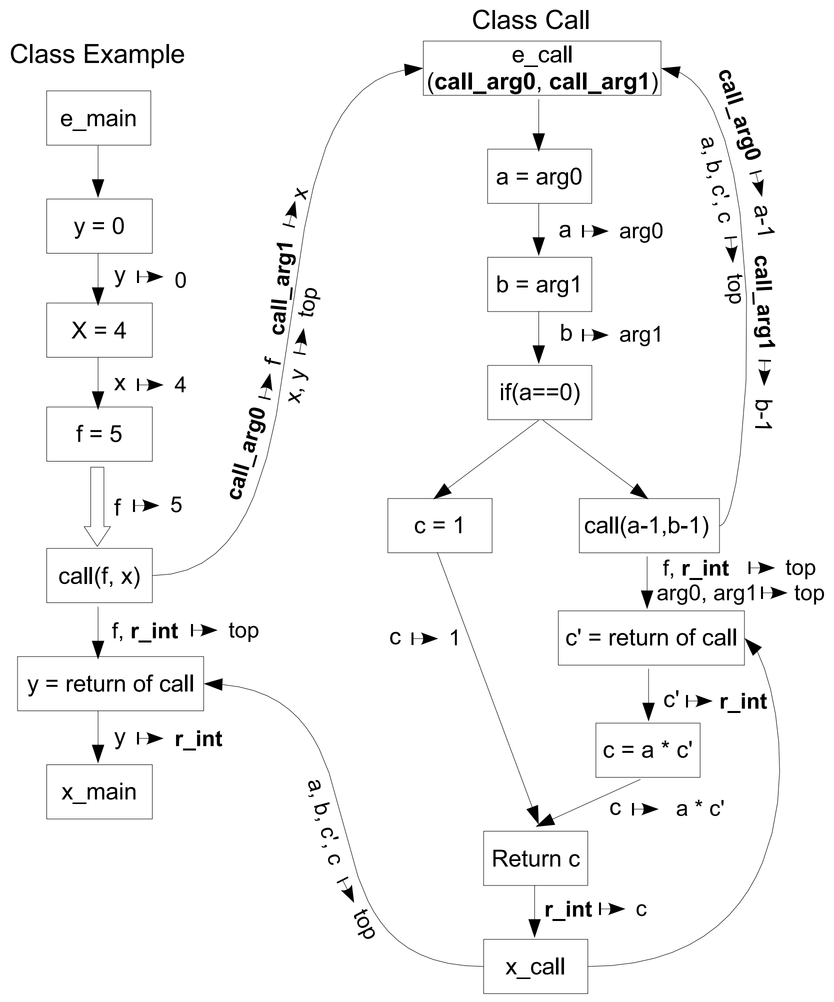


Figure 6.4: A supergraph with abstract parameter-passing

The intention for the interprocedural dead code analysis under this data abstraction is: if a variable is assigned to be ANY at some line of code, and the result is still ID, then this line of code is considered as dead.

The abstract interpretation p' of a primitive operation p is derived as $p'(l_1) = l_2$ where $l_2 \in L$ is the least PER that inculdes $\{p(x) \mid x \in l_1\}$ for each $l_1 \in L$. Then, the result is computed by the least fixed point computation under the oredering $\sqsubseteq$, which will be performed by weighted pushdown model checking.

Definition 45 *With L as the abstract data domain, a bounded idempotent semiring $S = (D, \otimes, \oplus, 0, 1)$ is defined as:*

1. *Weighted domain D is defined as*

$$D = \{\lambda x.x, \lambda x.\text{ANY}, \lambda x.\text{ID}, \text{ZERO} \mid x \in L\}$$

with the ordering that

$$\lambda x.\text{ANY} \sqsubseteq \lambda x.x \sqsubseteq \lambda x.\text{ID} \sqsubseteq \text{ZERO}$$

where ZERO *is naturally interpreted as that the program execution is interrupted by an error.*

2. *1 is defined as id* $= \lambda x.x$
3. *0 is defined as* ZERO
4. *The* $\otimes$ *operator composes the effects by transfer functions along one path. The operation* $\otimes$ *on* D *is defined as*

$$\forall d \in D,\ \text{ZERO} \otimes d = d \otimes \text{ZERO} = \text{ZERO}$$

$$\forall d \in D,\ \lambda x.x \otimes d = d \otimes \lambda x.x = d$$

$$\lambda x.\text{ANY} \otimes \lambda x.\text{ID} = \lambda x.\text{ID}$$

$$\lambda x.\text{ANY} \otimes \lambda x.\text{ANY} = \lambda x.\text{ANY}$$

$$\lambda x.\text{ID} \otimes \lambda x.\text{ANY} = \lambda x.\text{ANY}$$

$$\lambda x.\text{ID} \otimes \lambda x.\text{ID} = \lambda x.\text{ID}$$

5. *The* $\oplus$ *operator combines effects on the property domain by transfer functions from different branches. The operation* $\oplus$ *on* D *is defined as*

$$\forall d \in D,\ \text{ZERO} \oplus d = d \oplus \text{ZERO} = d$$

$$\forall d \in D,\ \lambda x.\text{ANY} \oplus d = d \oplus \lambda x.\text{ANY} = \lambda x.\text{ANY}$$

$$\lambda x.\text{ID} \oplus \lambda x.\text{ID} = \lambda x.\text{ID}$$

$$\lambda x.x \oplus \lambda x.x = \lambda x.x$$

$$\lambda x.\text{ID} \oplus \lambda x.x = \lambda x.x \oplus \lambda x.\text{ID} = \lambda x.x$$

Distributivity of $\otimes$ *over* $\oplus$ *is easily checked.*

With the bounded idempotent semiring in Definition 45, a typical way of interprocedural dead code detection works as follows:

> select some line of code and assign the weight of its transition associated to be $\lambda x.$ANY, this line of code is dead if the weight of the result is either $\lambda x.x$, or $\lambda x.$ID.

The soundness of this analysis is guaranteed by the facts that:

- the construction of PER-based forward abstract interpretation, and
- the conservative approximation of the definition of $\oplus$.

Example 21 *By examining the method* `example` *in Figure 6.2, the corresponding underlined model, i.e. the exploded supergraph, is in Figure 6.5, where labels of* $\lambda x.x$ *are omitted for simplicity. Some typical pushdown transitions are as follows,*

$$\begin{array}{ll} \langle \Lambda, n_0 \rangle \hookrightarrow \langle x, n_1 \rangle & \lambda x.\text{ID} \\ \langle x, n_0 \rangle \hookrightarrow \langle y, n_1 \rangle & \lambda x.x \end{array}$$

$$\begin{tikzcd}
\{\Lambda\} \arrow[d] \arrow[dr, "\lambda x.\textsc{id}"] & & & n_0 \\
\{\Lambda \arrow[d] & x\} \arrow[d] \arrow[dr] & & n_1 \\
\{\Lambda \arrow[d] \arrow[dr, "\lambda x.\textsc{id}"] & x & y\} \arrow[d] & n_2 \\
\{\Lambda \arrow[d] & x \arrow[d] \arrow[dr] & y\} & n_3 \\
\{\Lambda & x & y\} & n_4
\end{tikzcd}$$

Figure 6.5: The Exploded Supergraph for `Example`

When line 21 is assigned to $\lambda x.\textsc{any}$*, the data flow facts reaching* $\langle x, n_4\rangle$ *is still* $\lambda x.\textsc{id}$*. We know line 21 is dead.*

This solution only shows the key step of an interprecedural dead code elimination. The analysis is of course inefficient by trying lines of codes one after another. How to collects hints for selecting codes with preprocessing or analysis is needed. We intend to show how a real interprocedural dead code elimination works.

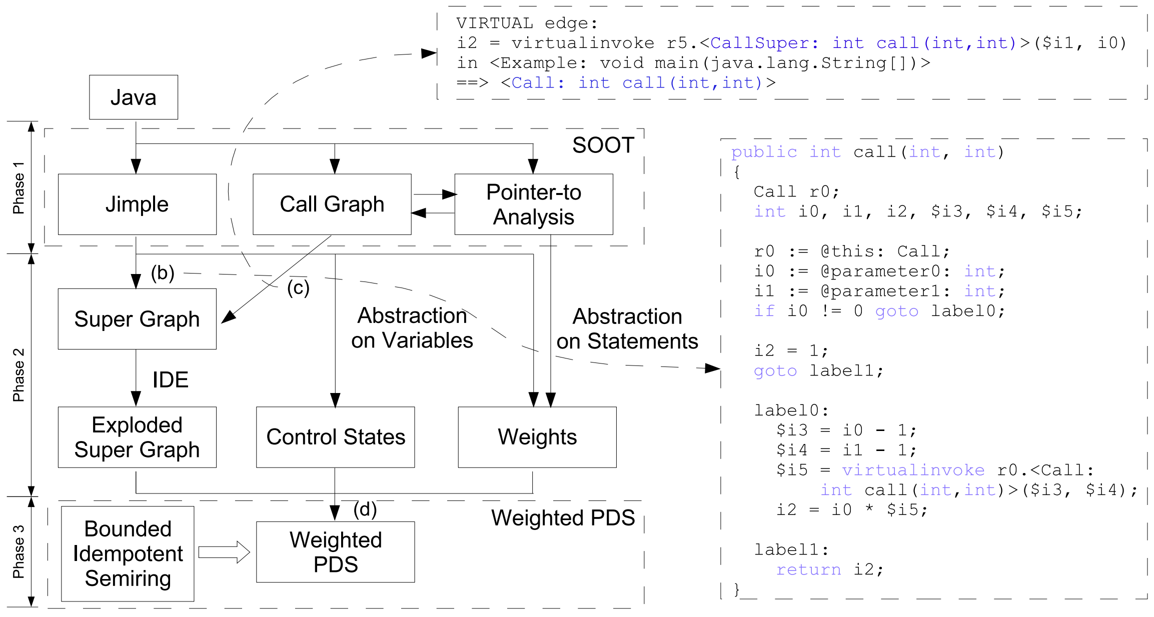
6.3 The Prototype Implementation

Our prototype is implemented as shown in Figure 6.6. It is developed with SOOT for Java preprocessing and the WPDS library as the back-end model checking engine. Jimple, a typed stackless 3-address intermediate representation, is the analysis target. To make use of the existing tools enables us a rapid prototyping for our analysis design. The analysis procedure is illustrated with Example 18. There are three phases:

- In *phase 1*, Java programs are translated into Jimple with SOOT. Some sample result is shown in (b).
- In *phase 2*, abstraction is performed. This phase is implemented as 1500 lines of Java code. To construct the interprocedural control flow graph of the program, the *call graph*, i.e. a set of possible call edges among procedures generated by SOOT, is borrowed. As shown in (c), the virtual method call in Example 18 is resolved and the corresponding call edge is given. The pointer-to analysis module in SOOT also helps to handle the aliasing among object references when performing abstractions on variables of interest.

  The output of abstraction, i.e. the exploded supergraph as the underlined model for the later model checking, as shown graphically [2] in Figure 6.7.
- In *phase 3*, model checking is performed on the generated model by phase 2. A bounded idempotent semiring, specific to the analysis of interest, is designed (Section 4.2) and implemented based on the weighted PDS library beforehand.

[2]This graphical drawing is part of our implementation. It is automatically generated for debugging purpose.



```
VIRTUAL edge:
i2 = virtualinvoke r5.<CallSuper: int call(int,int)>($i1, i0)
in <Example: void main(java.lang.String[])>
==> <Call: int call(int,int)>
```

```
public int call(int, int)
{
  Call r0;
  int i0, i1, i2, $i3, $i4, $i5;

  r0 := @this: Call;
  i0 := @parameter0: int;
  i1 := @parameter1: int;
  if i0 != 0 goto label0;

  i2 = 1;
  goto label1;

  label0:
    $i3 = i0 - 1;
    $i4 = i1 - 1;
    $i5 = virtualinvoke r0.<Call:
          int call(int,int)>($i3, $i4);
    i2 = i0 * $i5;

  label1:
    return i2;
}
```

Figure 6.6: A Prototype Framework

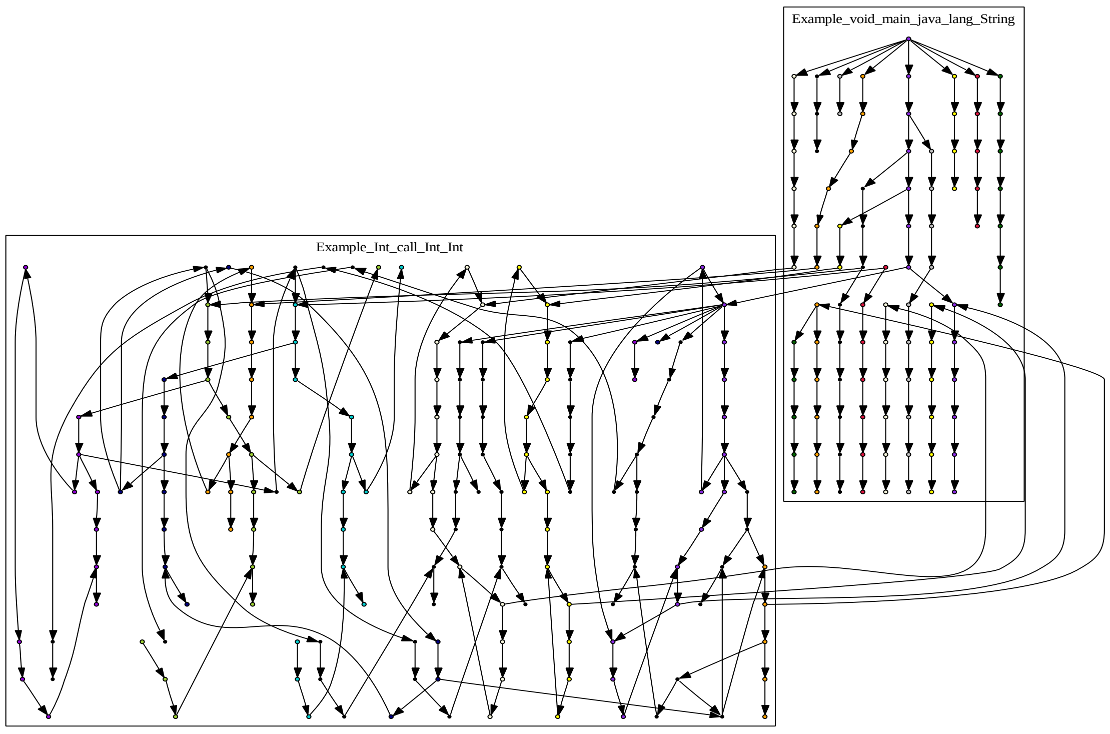


Figure 6.7: An exploded Interprocedural Control Flow Graph

Chapter 7

Conclusions

This thesis is dedicated to interprocedural program analysis based on weighted pushdown model checking, following to the crucial view that program analysis can be regarded as model checking of abstract interpretation. This approach provides program analysis with soundness guarantees by the machinery of abstract interpretation and model checking.

Our work is an interprocedural extension to Bandera-like approach. Bandera is a tool set for automatic generation of program analysis based on several popular finite model checkers. Thus, program analysis generated by it is essentially intraprocedural. We extend this methodology to interprocedural program analysis based on infinite model checking on pushdown systems, supported with prototype implementation.

We take points-to analysis, the basis of interprocedural program analysis for Java as our first target. Points-to analysis for Java is not easy. First of all, it is quite equivalent to call graph generation. The dynamically dispatched method depends on the actual type of receiver object. On the contrary, interprocedural points-to analysis expects a precise call graph information. Besides, the analysis on fields need to handle the aliasing among references. It is not easy in an interprocedural case. Although Java takes call-by-value mechanism, the state of objects still can be changed implicitly by passing references to other procedures. Other problems come from various infinities, such as the unbounded nesting of array, field access, method invocations, etc. Our work shows how such a crucial analysis can be naturally settled as a weighted pushdown model checking problem.

We present context/field/flow-sensitive points-to analysis algorithms for Java, with exploring primary choices on model design and algorithm construction. They are orthogonally two dimensional: an *on-the-fly* vs an *ahead-of-time* algorithm construction, an *exploded supergraph* vs an interprocedural *control flow graph* based model design. These primary design choices traditionally concerned in program analysis are also shown to be nicely formulated.

Our study shows that an on-the-fly analysis dispatches overwhelming model checking requests. It deserves a try on automatically removing invalid paths from the analysis result. Whereas, automatic path removal is restricted by the exploded supergraph construction, since each control flow is exploded. In contrast, based on the interprocedural CFG model design, we propose an ahead-of-time points-to analysis algorithm which can be done in one-run model checking.

From the attempts on automatic path removal and bounded field tracing (i.e. cast aliasing automatically), we show that more design efforts can be handed over to the back-end model checking engine. Although iterative analysis is basically inevitable for soundness, even for a model checking based approach, model checker can be exploited as a ready-made fix-point calculator. To note that, the ahead-of-time construction needs to explicitly give a bound on tracing the nesting of field access; and it is automatically bounded by an on-the-fly algorithm construction.

Furthermore, the relatively unexplored problem of parametrization is explored. Parameterized *flow-sensitivity* is naturally obtained in our algorithms based on either model reduction or simplifying the weight design.

From the preliminary evaluations, we feel that a complete context/field/flow-sensitive choices of the analysis will not scale regardless of implementations. Probably, flow-insensitive points-to analysis on SSA [43] would be a realistic solution.

Dead code elimination is intraprocedural in essence. For instance, variables passed as arguments to procedures are directly considered to be used. Even based on infinite model checking, a dead code elimination analysis is rather talking about global variables. We present dead code elimination encoded as both pushdown and weighted pushdown model checking problems. Our interprocedural irrelevant code elimination analysis goes a step further and is more on a semantical sense. The interaction among procedures is essentially captured by the variable dependency based on the exploded supergraph, under PER based abstraction.

Future Directions

After a thorough study on program analysis based on weighted pushdown model checking, a natural question is about the expressive power of weighted pushdown systems. We think weighted pushdown model checking is more powerful than regular pushdown model checking. One possible encoding could be that equivalence classes of ω-languages is taken as the weight space and concatenation is taken as $\otimes$. The difficulty is, the semiring based formulation is inherently restricted from representing properties that depends on pathes rather on states. More theoretical study on systematic derivation of a bounded idempotent semiring from an abstraction is demanding. That is, what kind of abstractions can be encoded into weighted pushdown systems by keeping sound property need to be further examined.

Based on the same weight design, an on-the-fly points-to analysis would be more precise than an ahead-of-time points-to analysis. This appears in the field-sensitive analysis, especially when a model design is based on an exploded supergraph. For field insensitive points-to analyses based on a control flow graph, it is an interesting theoretical question whether an ahead-of-time points-to analysis is equally precise as an on-the-fly analysis.

Currently, we work on the model checking based analysis and verification of sequential programs. Whereas modern programming languages are characterized by concurrent behaviors. Since the model checking problem on pushdown systems with more than 1 stack becomes undecidable, abstractions cannot be avoided. For instance, a context-bounded concurrent pushdown model checking algorithm is presented in [62] by restricting the number of contexts switches among processes. [60] presents an assume-guarantee approach plus a counterexample guided abstraction for model checking on concurrency. The key is only one thread is examined once, with assumption on the environment about how threads may interfere. Another approach is back to the emptiness problem for the intersection of languages by applying upper approximations on the context-free languages [61]. How to cover concurrency will be our next topic.

Bibliography

[1] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking, MIT,1999.

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

[3] F. Nielson, H.R. Nielson, C. Hankin, Principles of Program Analysis, Springer, 1999.

[4] B. Steffen. Data Flow Analysis as Model Checking. In TACS '91, LNCS 526, pages 346–365. Springer, 1991.

[5] D.A. Schmidt. Data ow analysis is model checking of abstract interpretation. *In Proceedings of the Twenty Fifth Annual Symposium on Principles of Programming Languages*, pages 38-48. ACM Press, 1998.

[6] The Model Checker Spin, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279-295.

[7] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 232 C 247. Springer, July 2000.

[8] S. Schwoon, S. Jha, T. Reps, and S. Stubblebine. On generalized authorization problems. In Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW), pages 202 C 218. IEEE Computer Society, June 2003.

[9] Manna, Z. and Pnueli, A. Axiomatic approach to total correctness of programs. Acta lnformatica 3 (1974), 243-263.

[10] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz (ed.): Mathematical aspects of computer science. Proc. Symposia in Applied Mathematics 19. Providence (R.I.): Amer. Math. Soc. 1967, p. 19 C 32.

[11] C.A.R. Hoare. An axiomatic basis of computer programming. Comm. ACM 12, 576 C 580, 583 (1969).

[12] A. Pnueli. The Temporal Logic of Programs. In Proceedings of the 18th IEEE Symposium Foundations of Computer Science (FOCS 1977), pages 46-57, 1977.

[13] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981 Lecture Notes in Computer Science*, volume 131. Springer-Verlag, 1981.

[14] J.P. Quielle, and J. Sifkis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th International Symposium on Programming. Lecture Notes in Computer Science* 137, Springer Verlag, New York, 1981, pp. 337-350.

[15] A. Biere, A. Cimatti, E. M. Clarke and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, *Lecture Notes in Computer Science* Vol. 1579.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *Proceedings of the 29th Annual Symposium on Principles of Programming Languages* (POPL), ACM Press, 2002, pp. 58-70.

[17] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. JACM 50(5): 752-794 (2003).

[18] R. Gerth, D. Peled, M. Y. Vardi and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of 15th International Symposium on Protocol Specification, Testing, and Verification* (PSTV'95), pages 3-18. Chapman & Hall, 1995.

[19] A. Bouajjani, J. Esparza, A. Finkel, et al. An efficient automata approach to some problems on context-free grammars. Information Processing Letters archive Volume 74, Issue 5-6 (June 2000).

[20] J. A. Bergstra, A. Ponse, and S. A. Smolka, Editors. Handbook of Process Algebra. Elsevier, 2001.

[21] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *the 22th ACM Symposium on Principles of Programming Languages* (POPL'95), pages 130 C 141, San Francisco, 1995.

[22] G. Ramalingam. Context sensitive synchronization sensitive analysis is undecidable. ACM Trans. on Programming Languages and Systems, 22:416 C 430, 2000.

[23] D. Kozen. Results on the propositional μ-calculus. Theoretical Computer Science Volume 27, Issue 3, 1983, Pages 333-354.

[24] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *the 4th ACM Symposium on Principles of Programming Languages* (POPL'77), pages 238-252, Los Angeles, 1977.

[25] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar. Symbolic model checking with rich assertional languages. In *the 9th International Conference on Computer Aided Verification* (CAV'97), Lecture Notes in Computer Science, pages 424-435, 1997.

[26] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In the 8th International Conference on Concurrency Theory (CONCUR'97), volume 1243 of Lecture Notes in Computer Science, pages 135-150. Springer-Verlag, 1997.

[27] games and pushdown processes

[28] S. Schwoon. Model-checking pushdown systems. PhD thesis, Technische Universität München, 2002.

[29] A. V. Aho, J. E. Hopcroft, J. D. Ullman. The design and analysis of computer algorithms. Addison-Wesley, 1997.

[30] K. Mehlhorn . Graph algorithms and NP-completeness. EATCS Monographs On Theoretical Computer Science; Vol. 2, Springer-Verlag, 1984.

[31] M. Yannakakis. Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS'90).

[32] Lacey, D., Jones, N. D., Wyk, E. V. and Frederiksen, C. C., Proving correctness of compiler optimizations by temporal logic, *Proc. 29th ACM Symposium on Principles of Programming Languages*, *Symposium on Principles of Programming Languages*, pp. 283–294, 2002.

[33] Hunt, S., PERs Generalize Projections for Strictness Analysis (Extended Abstract), *Functional Programming: Proc. 1990 Glasgow Workshop*, pp. 114–125 (1991), Springer-Verlag.

[34] Lal, A., Balakrishnan, G., and Reps, T., Extended weighted pushdown systems. In Proc. Computer-Aided Verification, 2005.

[35] D. Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In the Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, Pages:47-56, 1988.

[36] J. Gosling, B. Joy, G. Steele, G. Bracha. The Java™ Language Specification (Third Edition), 2005.

[37] Sagiv, M., Reps, T., and Horwitz, S., Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167 (1996), 131–170.

[38] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701 C 726, November/December 1998.

[39] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *the International Conference on Compiler Construction* (CC'03), pages 126-137, 2003.

[40] C. S. Horstmann, G. Cornell. Core Java™ 2 Volume I and II - Fundamentals, Seventh Edition. Prentice Hall PTR, August, 2004.

[41] R. Vallee, Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay: Soot - a Java bytecode optimization framework, Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research 1999 (CASCON '99), Ontario, Canada, November 1999.

[42] Olin Shivers. Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[43] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *SIGPLAN 98 Conference on Programming Language Design and Implementation*, pages 97-105, June 1998.

[44] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *the International Symposium on Software Testing and Analysis*, pages 1 C 11, 2002.

[45] O. Lhoták and L. Hendren. In the *15th International Conference on Compiler Construction* (CC 2006). LNCS volume 3923, Pages 47-64, 2006.

[46] Reps, T., Schwoon, S., Jha, S., and Melski, D., Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming, 58(1 C 2):206–263, October 2005.

[47] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.

[48] M. Fändrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI),Montreal, Canada, June 1998.

[49] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), Snowbird, Utah, June 2001.

[50] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), Tampa Bay, Florida, October 2001.

[51] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 2005.

[52] Manu Sridharan and Ras Bodik. Refinement-Based Context-Sensitive Points-To Analysis for Java. UCB/EECS-2006-31, EECS Department, University of California, Berkeley.

[53] M. F ahndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*(PLDI), pages 253 C 263, June 2000.

[54] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. Proceedings of the *ACM SIGPLAN Conference on Programming Language Design and Implementation*(PLDI), 2004.

[55] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95) (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77-101.

[56] Bacon, D. F. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[57] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the 12th International Conference on Compiler Construction* (CC), pages 153-169, April 2003.

[58] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), San Diego, CA, June 2003.

[59] M.S. Hecht, J.D.Ullman. Flow graph reduibility. SIAM J. Computing 1-2, 1972.

[60] T.A. Henzinger, R. Jhala, R. Majumdar, S. Qadeer. Thread-modular Abstraction Refinement. In the 15th Internatial Conference on Computer-Aided Verification (CAV03).

[61] A. Bouajjani, J. Esparza, T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In the *30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL03), New Orleans, Louisisana, January, 2003.

[62] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In the Proceedings of the *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS05), 2005.

Publications

[1] Li Xin, Mizuhito Ogawa. A Lightweight Mutual Authentication based on Proxy Certificate Trust List. Computer Software, Vol.22, No.2, pp.85-89, 2005.

[2] Li Xin, Mizuhito Ogawa. A Lightweight Mutual Authentication based on Proxy Certificate Trust List. The *5th International Conference on Parallel and Distributed Computing*, Applications and Technologies (PDCAT'04), Springer LNCS3320, pp.424-440, 2004.

[3] Li Xin, Mizuhito Ogawa. Interprocedural Program Analysis for Java based on Weighted Pushdown Model Checking. The *5th International Workshop on Automated Verification of Infinite-State Systems* (AVIS'06).