Program Analysis with Regularly Annotated Constraints

by

John Kodumal

B.S. (Harvey Mudd College) 2000

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Alexander Aiken, Chair
Professor George Necula
Professor Steven N. Evans

Spring 2006

The dissertation of John Kodumal is approved:

Professor Alexander Aiken, Chair	Date
Professor George Necula	Date
Professor Steven N. Evans	Date

University of California, Berkeley

Spring 2006

Program Analysis with Regularly Annotated Constraints

Copyright © 2006

by

John Kodumal

Abstract

Program Analysis with Regularly Annotated Constraints

by

John Kodumal

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Alexander Aiken, Chair

Static program analysis can help improve the quality of large software systems. Unlike many other techniques for finding software defects, a sound static analysis can verify the absence of bugs. Effectively, sound analyses are able to check the behavior of code along all possible execution paths. Unfortunately, static analyses are often extremely difficult to implement. In order to guarantee that a program is correct, the analysis itself must not contain any defects. Analyses must also scale to handle real software systems, which can consist of millions of lines of source code. Meeting both of these requirements is a daunting task, as scalable algorithms are often non-obvious and tricky to get right.

This dissertation proposes the use of a single constraint-based toolkit for constructing static analyses. By using such a toolkit, analysis designers can take advantage of a large body of work on scalable algorithms for constraint satisfiability. While the idea of a generic program analysis toolkit is not new, this thesis work presents several innovations that differentiate our work from prior approaches. We present a compilation technique that allows the analysis designer to specify their analysis in a succinct, declarative language. Specialization reduces the effort required to write an analysis, and results in cleaner, more maintainable code. We present new approaches

to object serialization and incremental analysis that are implemented system-wide, ensuring that these features are automatically made available to clients of our toolkit. Most importantly, we provide new results on the expressiveness of set constraints through a new reduction from context-free language reachability, as well as a new extension to the constraint language based on annotations. With these new features, even more analyses can be expressed using set constraints, making a constraint toolkit even more useful.

Professor Alexander Aiken, Chair Date

Acknowledgements

First, I would like to thank my advisor Alex Aiken, for his eternal optimism, inspired guidance, and thoughtful mentorship. I'd also like to thank the other "Alex Students", past and present, who have been a great source of ideas and anecdotes: Kirsten Chevalier, Jeff Foster, David Gay, Simon Goldsmith, Ben Liblit, Zhendong Su, and Tachio Terauchi.

I'm indebted to the other members of the Open Source Quality Project, especially George Necula, who was kind enough to become my surrogate advisor after Alex's move to Stanford.

Special thanks go to the users of BANSHEE, especially Jeff Foster, Iulian Neamtiu, Polyvios Pratikakis, and Jonathan Ragan-Kelley. I should also point out that the construction in Figure 6.2 is due to Dan Wilkerson.

Finally, I would like to thank my good friends Edith Harbaugh, Matt Muto and Miki Yamamoto. Words can't express how much I appreciate your friendship and support.

To my parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A 2002 study published by the National Institute of Standards and Technology found that software defects cost the U.S. economy almost $60 billion annually [NIS02], and this number will undoubtedly grow as software systems pervade all aspects of our daily lives. The same study shows that software testing could potentially eliminate $22 billion of the losses. In many critical applications, however, eliminating one third of the defects does not ensure a high enough standard of quality— a single failure might mean the loss of lives, not just money. Moreover, the notorious cases of software failure have taught us an important lesson about the nature of bugs: they occur in exceptional situations, under an improbable set of conditions, which correspond precisely to the code paths missed by conventional testing.

Much of the current work in programming languages is focused on developing static program analyses that discover software defects. Static analysis does not suffer from the same drawbacks as testing: sound analyses check the behavior of code along all possible execution paths. Unlike testing, a sound static analysis can verify the absence of bugs. The unfortunate reality, however, is that such analyses are extremely difficult to implement. In order to guarantee that a program is correct, the analysis

itself must not contain any defects. Analyses must also scale to handle real software systems, which can consist of millions of lines of source code. Meeting both of these requirements is a daunting task, as scalable algorithms are often non-obvious and tricky to get right.

Constraint-Based Analysis

One approach to lowering implementation cost is to express the analysis using *constraints*. Constraints separate analysis *specification* (constraint generation) from analysis *implementation* (constraint resolution). By exploiting this separation, designers can benefit from existing algorithms for constraint resolution. Concretely, this separation can be realized by implementing a generic constraint resolution algorithm. This separation helps, but leaves several issues unaddressed. First, a generic constraint resolution algorithm with no knowledge of the client may pay a large performance penalty for generality. For example, the fastest hand-written version of Andersen's analysis [HT01b] is much faster than the fastest version built using a generic toolkit [AFFS98]. Furthermore, existing toolkits lack support for separate compilation, which is needed to integrate analyses into real build systems. A final issue is that although there are a number of results demonstrating the expressiveness of set constraints, most of these results are of theoretical interest only, as the reductions used do not lead to scalable implementations. A more practical understanding of the expressiveness of the constraint formalism is lacking.

We have built Banshee, a constraint-based analysis toolkit that addresses these problems. Banshee succeeds Bane, a first-generation toolkit for constraint-based program analysis [AFFS98]. Banshee inherits several features from Bane, particularly support for *mixed constraints*, which allow several constraint formalisms to be combined in one application. Banshee also provides a number of innovations that

make it more useful and easier to use:

- We use a code generator to specialize the constraint back-end for each program analysis. The analysis designer describes a set of constructor signatures in a specification file, which Banshee compiles into a specialized constraint resolution engine. Specialization allows checking a host of correctness conditions statically. Software maintenance also improves: specialization allows the designer to modify analysis with few manual changes simply by tweaking the specification file.

- We have added support for a limited form of incremental analysis via *backtracking*, which allows constraint systems to be rolled back to any previous state. Backtracking can be used to analyze large projects incrementally: when a source file is modified, we roll back the constraint system to the state just before the analysis of that file. By choosing the order in which files are analyzed to exploit locality among file edits, we show experimentally that backtracking is very effective in avoiding reanalysis of files.

- We show an efficient mechanism for serialization and deserialization of constraint systems. The ability to save and load constraint systems is important for integrating Banshee-derived analysis into real build processes as well as for supporting incremental analysis. This feature is nontrivial, especially in conjunction with backtracking; our solution exploits Banshee's use of explicit regions for memory management.

- We provide two new results showing the practical expressiveness of our constraint framework: first, a new reduction from Dyck context-free language reachability to set constraints, and second, an extension to the constraint language based on *regular annotations*. Using these new results, a wide variety of ap-

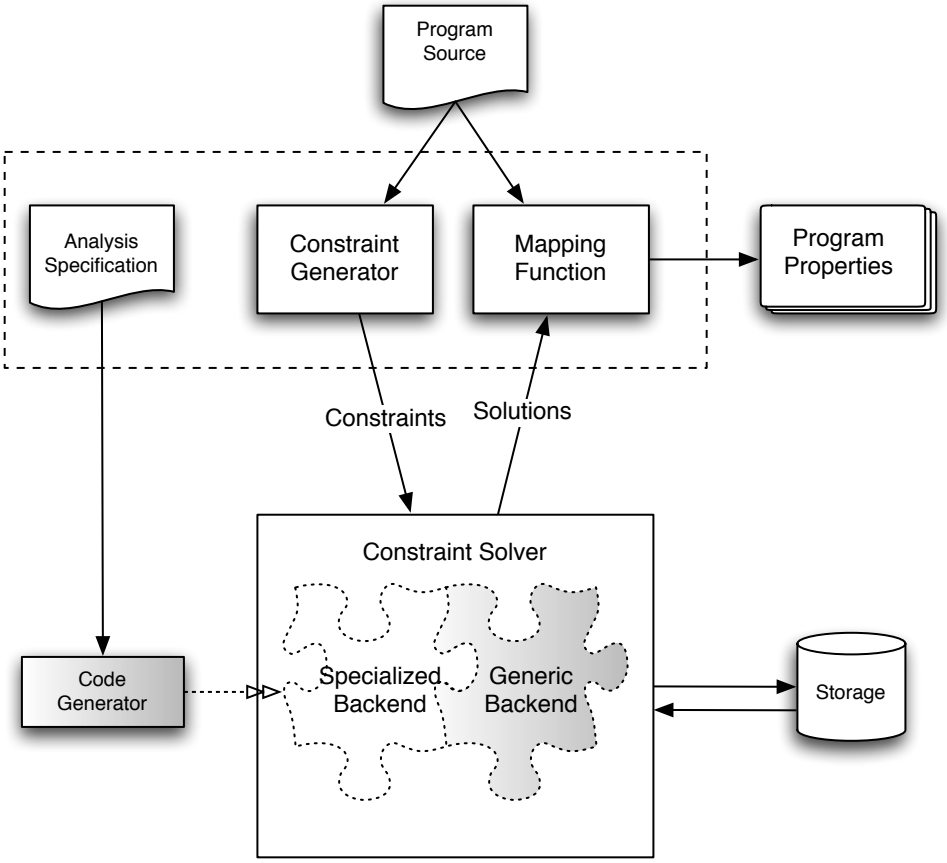


Figure 1.1: The Banshee approach to constructing program analyses

plications not previously known to be expressible using set constraints can be realized using Banshee.

- We have written Banshee from the ground up, implementing all of the important optimizations in Bane, while the code generation framework has enabled us to add a host of engineering and algorithmic improvements. We show how Banshee's specification mechanism allows a diverse class of analyses to be easily expressed, while the performance is nearly 100 times faster than Bane on some standard benchmarks.

Figure 1.1 illustrates the process of creating a program analysis using Banshee. To use Banshee, the analysis designer writes an analysis specification that defines the constructor signatures and annotations for the analysis. A code generator reads the specification and generates C code containing the specialized components of the constraint solver. This specialized backend is linked together with a generic backend to produce a specialized constraint solver. The analysis designer must also provide a constraint generator, which reduces the program source code to a collection of constraints, and a mapping function which relates the solutions of the constraints back to the program source. The constraint solver can also save/load constraint systems to/from disk for further processing.

Outline of the Dissertation

This dissertation is organized around a collection of program analyses that use a constraint framework introduced in the next chapter. New features and extensions to the basic framework are introduced as needed by the applications. For reference, the complete framework is summarized in the penultimate chapter.

Chapter 2

Mixed Constraints

In this chapter we introduce the *mixed constraint* formalism upon which Banshee is based. Our goal is not to provide a complete reference on mixed constraints, but rather to cover just the concepts needed to understand the remaining chapters.

2.1 Motivation

Many different constraint formalisms have proved useful in program analysis. For example, the use of unification constraints to infer simple types for the λ-calculus is well understood. A possible type rule for function application shows the use of such constraints:

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 \\ \tau_1 = \tau_2 \to \alpha & \alpha \text{ fresh}\end{array}}{\Gamma \vdash e_1\ e_2 : \alpha}\ \text{(App)}$$

Here, the unification constraint $\tau_1 = \tau_2 \to \alpha$ forces τ_1 to be a function with domain τ_2 and range α. Such constraints can be solved very efficiently, for instance by using a fast union-find algorithm. There is a tradeoff, however— the constraints force all uses of a given function to have the same type. Many "safe" programs will be rejected

by the type checker.

A more sophisticated analysis might add a subsumption rule to the type system, which requires the addition of inclusion constraints. Fewer "safe" programs will be rejected by the type checker, but the resulting system of constraints will (in general) be more expensive to solve.

In practice, many program analyses require "targeted" precision: some aspects of the program must be modeled very precisely in order to allow more of the "safe" programs to pass, but other aspects must be modeled coarsely in order for the analysis to scale to large programs. The mixed constraint framework is a formalism that allows multiple constraint languages (or *sorts*) to be combined in a single application. The purpose of mixed constraints is to provide the analysis designer with the ability to trade off precision for efficiency where necessary.

2.2 Constraint Sorts

A sort s is a tuple $(V_\mathsf{s}, C_\mathsf{s}, O_\mathsf{s}, R_\mathsf{s})$ where V_s is a set of variables, C_s is a set of constructors, O_s is a set of operations, and R_s is a set of constraint relations. Each n-ary constructor $c_\mathsf{s} \in C_\mathsf{s}$ and operation $op_\mathsf{s} \in O_\mathsf{s}$ has a signature $\iota_1 \ldots \iota_n \to \mathsf{s}$ where ι_i is either s_i, $\bar{\bar{\mathsf{s}}}_i$, or $\bar{\mathsf{s}}_i$ for sorts s_i. Overlined arguments in a signature are contravariant, double overlined arguments are nonvariant, and all other arguments are covariant. It is assumed that all constructors are non-strict. A n-ary constructor c_s is *pure* if the sort of each of its arguments is s. Otherwise, c_s is *mixed*. For a sort s, a set of

variables, constructors, and operations defines a language of s-expressions e_s:

$$
\begin{array}{lll}
e_\mathsf{s} ::= & & \\
\quad | & v & v \in V_\mathsf{s} \\
\quad | & c_\mathsf{s}(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) & c_\mathsf{s} \text{ with signature } \iota_1 \ldots \iota_n \to \mathsf{s} \\
 & & \text{and } \iota_i \text{ is } \mathsf{s}_i, \overline{\overline{\mathsf{s}_i}}, \text{ or } \bar{\mathsf{s}}_i \\
\quad | & op_\mathsf{s}(e_{\mathsf{s}_1}, \ldots, e_{\mathsf{s}_n}) & op_\mathsf{s} \text{ with signature } \iota_1 \ldots \iota_n \to \mathsf{s} \\
 & & \text{and } \iota_i \text{ is } \mathsf{s}_i, \overline{\overline{\mathsf{s}_i}}, \text{ or } \bar{\mathsf{s}}_i
\end{array}
$$

Constraints between expressions are written $e_{1_\mathsf{s}}\ r_\mathsf{s}\ e_{2_\mathsf{s}}$ where r_s is a constraint relation ($r_\mathsf{s} \in R_\mathsf{s}$). Each sort s has two distinguished constraint relations: an inclusion relation (denoted $\subseteq_\mathsf{s}$) and a unification relation (denoted $=_\mathsf{s}$). For contravariant arguments, the inclusion relation is reversed: $e_1 \subseteq_{\bar{\mathsf{s}}} e_2 \Leftrightarrow e_2 \subseteq_\mathsf{s} e_1$. For non-variant arguments, inclusion is replaced by unification: $e_1 \subseteq_{\bar{\bar{\mathsf{s}}}} e_2 \Leftrightarrow e_1 =_\mathsf{s} e_2$. A constraint system $\mathcal{C}$ is a finite conjunction of constraints.

Expressions occurring on the left (right) of a constraint relation $\subseteq_\mathsf{s}$ are said to occur in an *L-context* (*R-context*). Subexpressions occur in the same context as their immediate enclosing parent, unless the subexpression is the ith argument to a constructor contravariant in i. In that case, the context of the subexpression flips; e.g. if $f(e)$ appears in an L-context and f is contravariant in its argument, then e occurs in an R-context. Some expressions may only appear in certain contexts; expressions that can only occur in L-contexts (R-contexts) are called *L-compatible* (*R-compatible*).

To fix ideas, we introduce the sorts supported in BANSHEE and informally explain their semantics. A formal presentation of the semantics of mixed constraints is given in [FA97]. We leave the set of constructors C_s unspecified in each example, as this set parameterizes the constraint language and is application-specific. There are three

basic sorts provided by BANSHEE: Set, Term, and FlowTerm, which we abbreviate s, t, and ft. We discuss each basic sort in turn.

Definition 2.2.1. *The Set sort is the tuple:* $(V_{\mathsf{s}}, C_{\mathsf{s}}, \{\cup_{\mathsf{s}}, \cap_{\mathsf{s}}, \bot_{\mathsf{s}}, \top_{\mathsf{s}}\} \cup Proj_C, \{\subseteq_{\mathsf{s}}, =_{\mathsf{s}}\})$.

Here V_{s} is a set of set-valued variables, $\cup_{\mathsf{s}}, \cap_{\mathsf{s}}$, $\subseteq_{\mathsf{s}}$, and $=_{\mathsf{s}}$ are the standard set operations, $\bot_{\mathsf{s}}$ is the empty set, and $\top_{\mathsf{s}}$ is the universal set. The set $Proj_C$ is a set of projection pattern operations, which we explain shortly. Each pure Set expression denotes a set of *ground terms*: a constant or a constructor $c_{\mathsf{s}}(t_1, \ldots, t_n)$ where each t_i is a ground term.

For each constructor $c : \iota_1 \cdots \iota_n \to \mathsf{s} \in C_{\mathsf{s}}$, $Proj_C$ includes n projection pattern operations. Given a constraint of the form $c(e_1, \ldots, e_n) \subseteq c^{\sim i}(e)$, the projection pattern $c^{\sim i}(e)$ has the effect of selecting e_i, the i-th component of the term on the left-hand side, and adding the constraint $e_i \subseteq_{\iota_i} e$. Projection patterns are closely related to the more standard projection notation $c^{-i}(e)$. We can express $c^{-i}(e)$ using the following equivalence:

$$\begin{aligned} c^{-i}(e) \quad \equiv \quad & \mathcal{V} \\ & \text{where } \mathcal{V} \text{ is a fresh variable and} \\ & e \subseteq c^{\sim i}(\mathcal{V}) \end{aligned}$$

The equivalence preserves least solutions under a compatibility restriction: the variable $\mathcal{V}$ used to represent $c^{-i}(se)$ is only L-compatible. We use the $c^{\sim i}(e)$ notation because it simplifies the presentation of the constraint re-write rules and occasionally makes certain concepts clearer. However, we also use $c^{-i}(e)$ wherever that notation is more natural. Our uses of $c^{-i}(e)$ always adhere to the compatibility restriction; therefore in this work it is always safe to replace that notation with the equivalent using $c^{\sim i}(e)$.

Definition 2.2.2. *The Term sort is the tuple:* $(V_t, C_t, \{\bot_t, \top_t\}, \{\leq_t, =_t\})$.

Here V_t is a set of term-valued variables, and $=_\mathsf{t}$ and $\leq_\mathsf{t}$ are unification and conditional unification [Ste96], respectively. The meaning of a pure Term expression is, as expected, a constant or a constructor $c_\mathsf{t}(t_1, \ldots, t_n)$ where the t_i are terms.

Definition 2.2.3. *The FlowTerm sort is the tuple:* $(V_{ft}, C_{ft}, \{\bot_{ft}, \top_{ft}\}, \{\subseteq_{ft}, =_{ft}\})$

The FlowTerm sort combines some of the features of the Set and Term sorts: like the Term sort, the FlowTerm sort requires sets to be built from a single head constructor. However, the FlowTerm sort permits directional constraints, much like the Set sort.

For each of the three base sorts s, BANSHEE provides a Row sort (written Row(s)). A Row of base sort s denotes a partial function from an infinite set of names to terms of sort s. Row expressions are used to model record types with width and depth subtyping. We omit further details of the Row sort here: for our purposes, it suffices to view rows as mappings between names and terms.

2.3 Constraint Resolution

In BANSHEE, algorithms for solving constraints operate by applying a set of resolution rules to the constraint system until no rules apply; this produces a constraint system in *solved form*. The solved form makes the process of reading off a particular solution (or all solutions) simple. For our language of mixed constraints, we apply the resolution rules as left-to-right re-write rules to reduce the constraint system to solved form. Figures 2.1, 2.2, and 2.3 show the resolution rules for the three base sorts.

$$
\begin{array}{rcll}
\mathcal{C} \wedge \{\mathcal{X} \subseteq_{\mathsf{s}} \mathcal{X}\} & \Rightarrow & \mathcal{C} & \\
\mathcal{C} \wedge \{e_{\mathsf{s}} \subseteq_{\mathsf{s}} \top\} & \Rightarrow & \mathcal{C} & \\
\mathcal{C} \wedge \{\bot \subseteq_{\mathsf{s}} e_{\mathsf{s}}\} & \Rightarrow & \mathcal{C} & \\
\mathcal{C} \wedge \{e_1 \cup e_2 \subseteq_{\mathsf{s}} e\} & \Rightarrow & \mathcal{C} \wedge \{e_1 \subseteq_{\mathsf{s}} e, e_2 \subseteq_{\mathsf{s}} e\} & \\
\mathcal{C} \wedge \{e \subseteq_{\mathsf{s}} e_1 \cap e_2\} & \Rightarrow & \mathcal{C} \wedge \{e \subseteq_{\mathsf{s}} e_1, e \subseteq_{\mathsf{s}} e_2\} & \\
\mathcal{C} \wedge \{c(e_1, \ldots, e_n) \subseteq_{\mathsf{s}} c(e'_1, \ldots, e'_n)\} & \Rightarrow & \mathcal{C} \wedge \bigwedge_{i=1}^{n} \{e_i \subseteq_{\iota_i} e'_i\} & \text{if } c : \iota_1 \cdots \iota_n \to \mathsf{s} \\
\mathcal{C} \wedge \{c(e_1, \ldots, e_n) \subseteq_{\mathsf{s}} c^{\sim i}(e)\} & \Rightarrow & \mathcal{C} \wedge \{e_i \subseteq_{\iota_i} e\} & \text{if } c : \iota_1 \cdots \iota_n \to \mathsf{s} \\
 & & & \text{and } 1 \leq i \leq n \\
\mathcal{C} \wedge \{c(\ldots) \subseteq_{\mathsf{s}} \bot\} & \Rightarrow & \text{inconsistent} & \\
\mathcal{C} \wedge \{\top \subseteq_{\mathsf{s}} c(\ldots)\} & \Rightarrow & \text{inconsistent} & \\
\mathcal{C} \wedge \{\top \subseteq_{\mathsf{s}} \bot\} & \Rightarrow & \text{inconsistent} & \\
\mathcal{C} \wedge \{c(\ldots) \subseteq_{\mathsf{s}} d(\ldots)\} & \Rightarrow & \text{inconsistent} &
\end{array}
$$

Figure 2.1: Constraint resolution for the Set sort

2.4 Constraint Graphs

Algorithmically, a system of set constraints $\mathcal{C}$ can be represented as a directed graph $G(\mathcal{C})$ where the nodes of the graph are set expressions and the edges denote *atomic constraints*. A constraint is atomic if either the left-hand side or the right-hand side is a set variable. Computing the solved form involves closing $G(\mathcal{C})$ under the resolution rules, which are descriptions of how to add new edges to the graph.

Inductive form is a particular graph representation for the Set sort that exploits the fact that variable-variable constraints $\mathcal{X} \subseteq \mathcal{Y}$ can be represented as either a successor edge ($\mathcal{Y} \in \mathit{succ}(\mathcal{X})$) or a predecessor edge ($\mathcal{X} \in \mathit{pred}(\mathcal{Y})$). The choice is made based on a fixed total order *ord* (generated randomly) on the variables. For a constraint $\mathcal{X} \subseteq \mathcal{Y}$, the edge is stored as a successor edge on $\mathcal{X}$ if $\mathit{ord}(\mathcal{X}) > \mathit{ord}(\mathcal{Y})$, otherwise, it is stored as a predecessor edge on $\mathcal{Y}$. Constraints of the form $c(\ldots) \subseteq \mathcal{X}$ are always stored as predecessor edges on $\mathcal{X}$, and constraints of the form $\mathcal{X} \subseteq c^{\sim i}(se)$ are always stored as successor edges on $\mathcal{X}$.

$$
\begin{array}{rcll}
\mathcal{C} \wedge \{\mathcal{X} =_{\mathsf{t}} \mathcal{X}\} & \Rightarrow & \mathcal{C} & \\
\mathcal{C} \wedge \{c(e_1, \ldots, e_n) =_{\mathsf{t}} c(e'_1, \ldots, e'_n)\} & \Rightarrow & \mathcal{C} \wedge \bigwedge_{i=1}^{n} \{e_{\iota_i} =_{\iota_i} e'_{\iota_i}\} & \text{if } c : \iota_1 \cdots \iota_n \to \mathsf{t} \\
\mathcal{C} \wedge \{e_{\mathsf{t}} \leq e'_{\mathsf{t}}\} & \Rightarrow & \mathcal{C} \wedge \{e_{\mathsf{t}} =_{\mathsf{t}} e'_{\mathsf{t}}\} & \text{if } e_{\mathsf{t}} \text{ is not } \bot \\
 & & \mathcal{C} & \text{if } e_{\mathsf{t}} \text{ is } \bot \\
\mathcal{C} \wedge \{c(\ldots) =_{\mathsf{t}} d(\ldots)\} & \Rightarrow & \text{inconsistent} &
\end{array}
$$

Figure 2.2: Constraint resolution for the Term sort

$$
\begin{array}{rcll}
\mathcal{C} \wedge \{\mathcal{X} =_{\mathsf{ft}} \mathcal{X}\} & \Rightarrow & \mathcal{C} & \\
\mathcal{C} \wedge \{c(e_1, \ldots, e_n) \subseteq_{\mathsf{ft}} c(e'_1, \ldots, e'_n)\} & \Rightarrow & \mathcal{C} \wedge \bigwedge_{i=1}^{n} \{e_{\iota_i} \subseteq_{\iota_i} e'_{\iota_i}\} & \text{if } c : \iota_1 \cdots \iota_n \to \mathsf{ft} \\
\mathcal{C} \wedge \{\mathcal{X} \subseteq_{\mathsf{ft}} c(e_1, \ldots, e_n)\} & \Rightarrow & \mathcal{C} \wedge \{\mathcal{X} =_{\mathsf{ft}} c(\mathcal{Y}_1, \ldots, \mathcal{Y}_n)\} \wedge \bigwedge_{i=1}^{n} \{\mathcal{Y}_i \subseteq_{\iota_i} e_i\} & \\
 & & & \text{if } c : \iota_1 \cdots \iota_n \to \mathsf{ft} \\
 & & & \text{and } \mathcal{Y}_i \text{ fresh} \\
\mathcal{C} \wedge \{c(e_1, \ldots, e_n) \subseteq_{\mathsf{ft}} \mathcal{X}\} & \Rightarrow & \mathcal{C} \wedge \{\mathcal{X} =_{\mathsf{ft}} c(\mathcal{Y}_1, \ldots, \mathcal{Y}_n)\} \wedge \bigwedge_{i=1}^{n} \{e_i \subseteq_{\iota_i} \mathcal{Y}_i\} & \\
 & & & \text{if } c : \iota_1 \cdots \iota_n \to \mathsf{ft} \\
 & & & \text{and } \mathcal{Y}_i \text{ fresh} \\
\mathcal{C} \wedge \{c(\ldots) \subseteq_{\mathsf{ft}} d(\ldots)\} & \Rightarrow & \text{inconsistent} &
\end{array}
$$

Figure 2.3: Constraint resolution for the FlowTerm sort

Given these representations, the transitive closure rule for inductive form is as follows:

$$L \in pred(\mathcal{X}) \wedge R \in succ(\mathcal{X}) \Rightarrow L \subseteq R$$

This rule, in conjunction with the resolution rules, produce a solved graph in inductive form. The advantage of inductive form is that many fewer transitive edges are added in comparison to other graph representations [FFSA98]. Inductive form does not explicitly compute the least solution, but the least solution is easily calculated by

computing transitive lower bounds on the constraint graph:

$$LS(\mathcal{Y}) = \{c(\ldots)|c(\ldots) \in pred(\mathcal{Y})\} \cup \bigcup_{\mathcal{X} \in pred(Y)} LS(\mathcal{X})$$

2.5 Constraint Solutions

It turns out that each constraint variable describes a *regular tree language* and the solved form of the constraint graph can be viewed as a collection of regular tree grammars. Each atomic constraint $c(\ldots) \subseteq \mathcal{X}$ in the solved form of the constraint system can be interpreted as a production in a tree grammar by treating it as a production $X \Rightarrow c(\ldots)$. For example, consider the solved constraint system

$$cons(zero, \mathcal{X}) \subseteq \mathcal{X}$$
$$nil \subseteq \mathcal{X}$$

where *zero* and *nil* are nullary constructors, and *cons* is a binary constructor. We see that $\mathcal{X}$ describes the set of all lists where every element is *zero*. The least solution for $\mathcal{X}$ can be viewed as a tree language $L(\mathcal{X})$ whose grammar is

$$\begin{aligned} X &\Rightarrow cons(zero, X) \\ X &\Rightarrow nil \end{aligned}$$

We will appeal to this characterization of solutions in Chapter 5.

Chapter 3

An Overview of Banshee

This chapter describes three features of BANSHEE (specialization, "poor man's" incremental analysis, and persistence) that address some of the practical limitations of other program analysis toolkits. Each feature is introduced in the context of a basic program analysis application, a type and effect inference system.

3.1 A Simple Application

Our source language is the untyped λ-calculus with references. The syntax is standard:

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \texttt{ref}\ e \mid \texttt{!}e \mid e_1\ \texttt{:=}\ e_2$$

The goal of the analysis is to infer the type of each expression, and additionally to infer each function's latent effect, which is a set of abstract locations read or written by the function when applied. The analysis is not meant to be a soft typing system: type errors may occur if, e.g., an attempt is made to dereference a function. We choose to represent types using the FlowTerm sort, and effects using the Set sort.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau; \bot} \text{ (Var)}$$

$$\frac{\Gamma, x : \mathcal{X} \vdash e : \tau; \rho}{\Gamma \vdash \lambda x.e : fun(\rho, \mathcal{X}, \tau); \bot} \text{ (Lam)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1; \rho_1 \qquad \Gamma \vdash e_2 : \tau_2; \rho_2 \\ \tau_1 \subseteq fun(\mathcal{E}, \tau_2, \mathcal{X}) \qquad \mathcal{E}, \mathcal{X} \text{ fresh}\end{array}}{\Gamma \vdash e_1\ e_2 : \mathcal{X}; \rho_1 \cup \rho_2 \cup \mathcal{E}} \text{ (App)}$$

$$\frac{\Gamma \vdash e : \tau; \rho \qquad c \text{ fresh}}{\Gamma \vdash \texttt{ref}\ e : ref(c, \tau); \rho} \text{ (Ref)}$$

$$\frac{\Gamma \vdash e : \tau; \rho \qquad \tau \subseteq ref(\mathcal{E}, \mathcal{X}) \qquad \mathcal{E}, \mathcal{X} \text{ fresh}}{\Gamma \vdash\ !e : \tau; \rho \cup \mathcal{E}} \text{ (Deref)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1; \rho_1 \qquad \Gamma \vdash e_2 : \tau_2; \rho_2 \\ \tau_1 \subseteq ref(\mathcal{E}, \tau_2) \qquad \mathcal{E} \text{ fresh}\end{array}}{\Gamma \vdash e_1\ \texttt{:=}\ e_2 : \tau_2; \rho_1 \cup \rho_2 \cup \mathcal{E}} \text{ (Assign)}$$

Figure 3.1: Type and effect inference rules for λ-calculus

Thus, our analysis uses the following constructors:

$$\begin{aligned} fun &: \mathsf{s}\ \overline{\mathsf{ft}}\ \mathsf{ft} \to \mathsf{ft} \\ ref &: \mathsf{s}\ \overline{\overline{\mathsf{ft}}} \to \mathsf{ft} \end{aligned}$$

which we use to represent the following language of types and effects:

$$\begin{aligned} \tau &::= fun(\rho, \tau_1, \tau_2) \mid ref(\rho, \tau) \mid \mathcal{X} \\ \rho &::= c \mid \mathcal{E} \mid \rho_1 \cup \rho_2 \mid \bot \mid \top \end{aligned}$$

Figure 3.1 shows the type rules for our analysis. Type judgments have the form $\Gamma \vdash e : \tau; \rho$, which states that in environment Γ, expression e has type τ and effect ρ. [1]

[1]For clarity, we use $\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \ldots$ to range over type variables and $\mathcal{E}, \mathcal{F}, \mathcal{G}, \ldots$ to range over effect variables.

We briefly explain each of the rules in turn. Rules (Var) and (Lam) are largely standard; note though that both forms have an empty effect. In rule (App), expressions e_1 and e_2 are type checked, yielding types τ_1 and τ_2, respectively, and τ_1 is constrained to be a function type with domain τ_2. Fresh variables $\mathcal{X}$ and $\mathcal{E}$ are used to represent the function's range and latent effect. The resulting expression has type $\mathcal{X}$, and effect $\rho_1 \cup \rho_2 \cup \mathcal{E}$. Rule (Ref) creates a fresh constant c to represent the set of memory locations created at this allocation site. The resulting type is a reference to expression e's type. Rule (Deref) adds the set of locations being dereferenced to the effect set. In (Assign), e_1's type is constrained to be a reference to τ_2, the type of e_2. The resulting effect is the union of the effect of evaluating e_1 and e_2 and the set of locations $\mathcal{E}$ that τ_1 represents.

With this analysis in mind, we continue with a discussion of the new features available in Banshee.

3.2 Specialization

Banshee employs a partial evaluator to specialize the constraint resolution engine for each program analysis application. The key observation behind our partial evaluation technique is that most analyses rely on a small, fixed number of constructor signatures, and because few analyses change the set of constructors at run time, constructor signatures can be specified statically. For example, our type and effect system uses only three signatures (one each for the *ref* and *fun* constructors, and one more for all the constants used to represent abstract locations). Banshee uses these signatures to implement customized versions of the constraint resolution rules.

To use Banshee, the analysis designer writes a specification file defining the constructor signatures for the analysis. We outline Banshee's specification syntax, which is inspired by ML recursive data type declarations. Each **data** dec-

laration defines a disjoint alphabet of constructors. For example, the declaration `data rho : set` defines `rho` to be a collection of constructors of sort Set. The `rho` alphabet serves only as a source of fresh constants, modeling the statically unknown set of abstract locations.

Each **data** declaration may be followed by an optional list of `|`-separated constructor declarations defining the (statically fixed) set of n-ary constructor signatures. Recall that our analysis requires a ternary constructor *fun*, which has two covariant fields and a contravariant field. To define this constructor, we need notation to specify variance. In BANSHEE's specification language, a signature element prefixed with `+` (resp. `-`) denotes a covariant (resp. contravariant) field. A signature elements prefixed with `=` denotes a nonvariant field. By default, fields are nonvariant.

Here is the complete specification for our type and effect example:

```
data tau : flowterm = fun of +rho * −tau * +tau
                    | ref of +rho * =tau
and rho : set
```

Knowing the signatures of the constructors enables a host of static checks that are not possible if new constructors are introduced at run time. To see this, consider the constructor signature as a restricted programming language. This language specifies how to build expressions with the constructor (encoded by the arity of the constructor and the types of its component expressions), and how constraints between expressions matching the constructor should be resolved (encoded by the sort and variance of each of its component expressions). From this perspective, allowing dynamic constructors necessitates an interpreter for the language of constructor signatures (this is essentially what BANE is). Specialization compiles the functionality encoded in a signature into C code, eliminating the overhead and runtime checking of an interpreter.

One of the most important advantages of Banshee specifications is that they make program analyses easier to debug and maintain. After writing a Banshee specification, the analysis designer's main task is to write C code to traverse abstract syntax, calling functions from the generated interface to build expressions, generate constraints, and extract solutions. This task is typically straightforward, as there should be a tight correspondence between type judgments and the Banshee code to implement them. Recall the rule (App) for our example analysis (Figure 3.1). Assuming a typical set of AST definitions, the corresponding Banshee code to implement this rule is:[2]

```
struct type_and_effect analyze(env gamma, ast_node n) {
  if (is_app_node(n)) {
    tau t1, t2, x;
    rho r1, r2, e;
    (t1, r1) = analyze(gamma,n->e1);
    (t2, r2) = analyze(gamma,n->e2);
    x = tau_fresh();
    e = rho_fresh();
    tau_inclusion(t1, fun(e,t2,x));
    return (x, rho_union([r1;r2;e]));
  }
  ...
}
```

This code is representative of the "handwritten" part of a Banshee analysis. Banshee's code generator makes the handwritten part clean: there is a close correspondence between clauses in the type rules and the Banshee code to implement

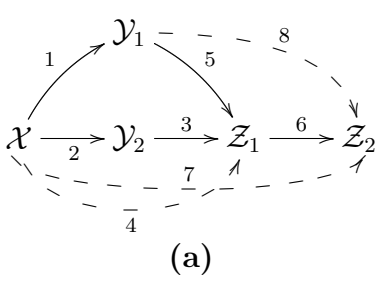
[2]We use a little syntactic sugar for pairs and lists in C to avoid showing the extra type declarations.

them.

3.3 "Poor Man's" Incremental Analysis

Incremental analysis is important in large projects where it is necessary to maintain a global analysis in the face of small edits. In such a setting, resource constraints make it infeasible to recompute analysis results from scratch in response to each edit. In this section we describe Banshee's support for a limited form of incremental analysis via *backtracking*. We believe that our use of backtracking as a mechanism for incrementalizing static analyses is novel, as is the idea of adding backtracking to a mixed constraint solver.

We assume a setting where constraints can be added to or deleted from the constraint system. In addition, queries to the constraint system can be interleaved arbitrarily between addition and deletion operations. Since our constraint solver is online, additions are handled without any additional machinery (the trick is handling constraint deletions). Informally, we define an incremental algorithm as an algorithm that updates its output in response to program edits without recomputing the analysis from scratch. In this setting, we call an incremental algorithm optimal if it never deletes (in other words, is never forced to rediscover) an atomic constraint that is part of the updated solution in response to a deletion operation. We also make the distinction between a *top level* atomic constraint, which is a constraint that has been introduced to the system by an addition operation, and an *induced* atomic constraint, which is a constraint that is added as a consequence of the resolution rules.



Edge	Dependencies
4	(2,3), (1,5)
7	(4,6), (1,8)
8	(5,6)

(b)

Figure 3.2: (a) Constraint graph and (b) edge dependency list

3.3.1 An Optimal Incremental Algorithm

We begin by motivating our decision not to implement an optimal incremental algorithm in Banshee. We outline an optimal algorithm, then illustrate the problems with it.

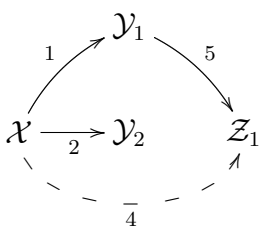
The key to implementing an optimal incremental algorithm is to maintain explicit dependency information. Besides adding edges to the constraint graph, the algorithm must keep track of the reasons *why* each edge was added. A straightforward representation of dependencies is a list of the edges in the graph that triggered each new edge addition. When a constraint is deleted, all occurrences of the corresponding edge in the constraint graph are removed from the dependency list. An empty dependency list means that all of the reasons an edge was added have been removed, and hence, that constraint should be removed. This process is repeated until no more constraints are removed.

We demonstrate these ideas with a simple example. Suppose the constraints

$$\mathcal{X} \subseteq \mathcal{Y}_1 \quad (1)$$

$$\mathcal{X} \subseteq \mathcal{Y}_2 \quad (2)$$

$$\mathcal{Y}_2 \subseteq \mathcal{Z}_1 \quad (3)$$



(a)

$\mathcal{Z}_2$

Edge	Dependencies
4	(1,5)
7	
8	

(b)

Figure 3.3: (a) Constraint graph and (b) edge dependency list after deleting constraints 3 and 6.

are added to an initially empty constraint system. By transitive closure, the latter two constraints would trigger the constraint

$$\mathcal{X} \subseteq \mathcal{Z}_1 \tag{4}$$

The incremental solver adds the entry (2,3) to the dependency list for constraint 4. This records the fact that the presence of constraints 2 and 3 together triggered the addition of constraint 4. If either constraint were deleted, constraint 4 should be deleted as well (unless there is some other entry in the dependency list for constraint 4). Next, suppose we add the constraints:

$$\mathcal{Y}_1 \subseteq \mathcal{Z}_1 \tag{5}$$

$$\mathcal{Z}_1 \subseteq \mathcal{Z}_2 \tag{6}$$

The introduction of constraint 5 triggers a redundant edge addition—so the entry (1,5) is added to the dependency list for constraint 4. Finally, the addition of

constraint (6) causes the constraints

$$\mathcal{X} \subseteq \mathcal{Z}_2 \tag{7}$$

$$\mathcal{Y}_1 \subseteq \mathcal{Z}_2 \tag{8}$$

to be added, with (4,6) and (1,8) added to the dependency list for constraint 7. Figure 3.2 shows the constraint graph and edge dependency list at this point in the example.

Now suppose we delete constraints 3 and 6 from the system, in that order. Deleting constraint 3 removes the entry (2,3) from the edge dependency list for constraint 4. However, the dependency list for constraint 4 is non-empty, so we do not remove it from the system. Deleting constraint 6 removes the entries (4,6) and (5,6) from the edge dependency table. Since constraints 7 and 8 have an empty dependency list, they are deleted from the constraint system. Notice that for correctness this algorithm must store *every* set of edges that may cause an edge to be added. Thus, the algorithm must maintain extra information even for redundant edge additions. While much work has been done to reduce redundant edge additions, constraint solvers still add many redundant edges to the graph [SFA00]. Having to store extra information on each redundant edge addition would hinder the scalability of the constraint solver.

Another practical concern is the engineering effort required to support an optimal incremental analysis. While efforts have been made to factor out incrementality at the data structure level, there does not seem to be a practical, general solution for arbitrary data structures [DSST86]. Adding ad-hoc support for incremental updates to each sort in Banshee is a daunting task, especially since the solver algorithms use highly optimized representations. As an example, our set constraint solver implements partial online cycle elimination, which uses a union-find data structure internally [FFSA98]. Adding incremental support to just the fast union find algorithm

is not easy—in fact, some of the published solutions are incorrect (see [GI91] for a discussion).

3.3.2 Backtracking

Instead of implementing a fully incremental analysis, we have added backtracking to Banshee. Backtracking allows a constraint system to be rolled back to any previous state. In this subsection, we explain the general approach to adding backtracking, and discuss the practical advantages of backtracking over optimal incremental analysis.

Backtracking is based on a simple approximation of edge dependencies: each induced constraint is assumed to be dependent on the *all* of the constraints introduced into the constraint system earlier. This seemingly gross over-approximation yields a simple strategy for handling constraint deletions: we simply delete all of the constraints added to the system after the constraint we wish to delete. Then we add back any top level constraints we didn't intend to delete, and allow the constraint solver to solve for any induced constraints we shouldn't have removed.

Operationally, this strategy can be implemented simply by time stamping each atomic constraint in the system. To delete constraint i, we perform a scan of all the edges in the constraint graph, deleting any edge whose timestamp is greater than i. Then we simply add back all the top level constraints timestamped j with $j > i$. Referring to our example in Figures 3.2 and 3.3, we see that deleting constraint 6 triggers the deletion of constraints 7 and 8 (in this case, backtracking does no more work than the optimal analysis). Deleting constraint 3, on the other hand, forces us to delete constraints 5 and 4. We will have to re-introduce the top level constraint 5 and solve to re-discover the induced constraint 4.

While backtracking is clearly not an optimal incremental technique, it does have a number of practical advantages over the optimal algorithm. First, backtracking is

fast: it can be accomplished in a linear scan over the edges in the constraint graph. Second, the storage overhead to support backtracking is minimal, requiring only a timestamp per edge.

In addition, we have devised a new data type called a *tracked reference* that allows us to add efficient backtracking support to general data structures. This has greatly simplified the task of incorporating backtracking into Banshee's algorithms, especially in the presence of optimizations like cycle elimination and projection merging [FFSA98; SFA00]. A tracked reference is essentially a mutable reference that maintains a repository of its old versions. Each tracked reference is tied to a clock; each tick of the clock checkpoints the reference's state. When the clock is rolled back, the previous state is restored. Rolling back is a destructive operation. Figure 3.4 contains a compilable Objective Caml implementation of the tracked reference data type. The implementation keeps a stack of closures containing the old reference contents. Invoking a closure restores the old contents. A backtracking operation simply pops entries off the stack until the previous recorded clock tick.

In a functional language, adding support for backtracking to a data structure is simple: it suffices to replace all references with tracked references and add calls to the clock's tick function when a checkpoint is desired. Obvious inefficiencies arise, however, if tracked references are applied carelessly. In general, they should be applied so as to avoid storing copies of large objects in the closures. Although Banshee is implemented in C rather than a functional language, we implemented backtracking by applying the tracked reference concept systematically to each data structure. Interestingly, we do not pay in any way for this factoring: there are no "tricks" that are obscured by applying the tracked reference idea generally. For example, applying tracked references to a standard union-find algorithm yields an algorithm that is equivalent to a well-known algorithm for union-find with backtracking [WT89].

3.4 Persistence

In this section, we briefly explain Banshee's approach to making Banshee's constraint systems persistent. Persistence is a useful feature to have when incorporating incremental analyses into standard build processes. We require persistence (rather than a feature to save and load in some simpler format) because we need to reconstruct the complete representations of our data structures to support backtracking and online constraint solving.

To support persistent constraint systems, we added serialization and deserialization capabilities to the region-based memory management library that Banshee uses for memory allocation. This approach allows us to save constraint systems by serializing a collection of regions, and load a constraint system by deserializing regions and updating any pointer values stored in the regions. Initially, we implemented serialization using a standard pointer tracing approach, but found this strategy to be too slow. Region-based serialization allows us to simply write sequential pages of memory to disk, which we found to be an order of magnitude faster than pointer tracing. With region-based serialization, we are able to serialize a 170 MB constraint graph in 2.4 seconds, vs. 30 seconds to serialize the same graph by tracing pointers.

Region-based memory management is also a good fit for our backtracking proposal. One effective strategy is to create a new region for each file being analyzed and allocate new objects in that region. When backtracking, any obsolete regions can be freed. In addition, if we know before loading a saved constraint system that a backtrack operation will take place immediately after the load, we can selectively deserialize only the live regions of memory (assuming that the metadata for backtracking is stored in a different set of regions that is deserialized).

```
module S = Stack
exception Tick

type clock = {
  mutable time : int;
  repository : (unit -> unit) S.t
}
type 'a tref = clock * 'a ref

let tref (clk: clock) (v :'a) : 'a tref =
  (clk, ref v)
let read (clk,r: 'a tref) : 'a =
  !r
let write (clk,r: 'a tref) (v : 'a) : unit =
  let old_v = !r in
  let closure = fun () -> r := old_v in
    begin
      S.push closure clk.repository;
      r := v
    end
let clock () : clock = {
    time = 0;
    repository = S.create () }
let time (clk : clock) : int =
  clk.time
let tick (clk : clock) : unit =
  let closure =
    fun () -> raise Tick in
    begin
      S.push closure clk.repository;
      clk.time <- clk.time + 1;
    end
let rollback (clk : clock) : unit =
  try
    while (not (S.is_empty clk.repository)) do
      let closure = (S.pop clk.repository) in
        closure()
    done
  with Tick -> (clk.time <- clk.time - 1)
```

Figure 3.4: Tracked reference data type

Chapter 4

Points-to Analysis: A Case Study

We continue with several realistic examples derived from points-to analyses we have formulated in Banshee. Our intent is to demonstrate that Banshee can be used to explore different design points and prototype several variations of a given program analysis. In the first examples, we refine the degree of subtyping used in the points-to analysis. Much of the recent research on points-to analysis has focused on this issue [Ste96; SH97; Das00]. In the last example, we show how to augment the points-to analysis to perform receiver class analysis in an object-oriented language with explicit pointer manipulations (e.g. C++). This last analysis is interesting as it computes the call graph on the fly during the points-to computation, instead of using a pre-computed call graph obtained from a coarser analysis (e.g. class hierarchy analysis).

We conclude the chapter with experimental results from our implementation of Andersen's analysis.

4.1 Formulating Andersen's Analysis

We begin by examining a set-based formulation of Andersen's points-to analysis built with Banshee. The analysis constructs a *points-to graph* from a set of abstract

$$\frac{\Gamma(x) = \mathit{ref}(\ell_x, \mathcal{X}_{\ell_x}, \overline{\mathcal{X}}_{\ell_x})}{\Gamma \vdash x : \mathit{ref}(\ell_x, \mathcal{X}_{\ell_x}, \overline{\mathcal{X}}_{\ell_x})} \text{ (Var)}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \&e : \mathit{ref}(0, \tau, \overline{1})} \text{ (Addr)}$$

$$\frac{\Gamma \vdash e : \tau \qquad \tau \subseteq \mathit{ref}(1, \mathcal{T}, \overline{0})}{\Gamma \vdash *e : \mathcal{T}} \text{ (Deref)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \\ \tau_1 \subseteq \mathit{ref}(1, 1, \overline{\mathcal{T}}_1) \qquad \tau_2 \subseteq \mathit{ref}(1, \mathcal{T}_2, \overline{0}) \\ \mathcal{T}_2 \subseteq \mathcal{T}_1\end{array}}{\Gamma \vdash e_1 \ = \ e_2 : \tau_2} \text{ (Assign)}$$

Figure 4.1: Constraint generation for Andersen's analysis

memory locations $\{\ell_1, \ldots, \ell_n\}$ and set variables $\mathcal{X}_{\ell_1}, \ldots, \mathcal{X}_{\ell_n}$.

Intuitively, we model a reference as an object with an abstract location and methods $\mathit{get} : \mathit{void} \to \mathcal{X}_{\ell_x}$ and $\mathit{set} : \mathcal{X}_{\ell_x} \to \mathit{void}$, where $\mathcal{X}_{\ell_x}$ represents the points-to set of the location. Updating the location corresponds to applying the *set* function to the new value. Dereferencing a location corresponds to applying the *get* function. In our vocabulary of Set expressions, references are modeled with a constructor *ref* containing three fields: a constant ℓ_x representing the abstract location, a covariant field $\mathcal{X}_{\ell_x}$ representing the *get* function, and a contravariant field $\overline{\mathcal{X}}_{\ell_x}$ representing the *set* function.

Figure 4.1 shows a subset of the inference rules for Andersen's analysis. The type judgments assign a set expression to each program expression, possibly generating some side constraints. To avoid having separate rules for l-values and r-values, each type judgment infers a set denoting an l-value. Hence, the set expression in the conclusion of (Var) denotes the location of program variable x, rather than its contents.

In Banshee, Andersen's analysis is defined with the following specification:

```
specification andersen : ANDERSEN =
  spec
    data location : set
    data T : set = ref of +location * −T * +T
  end
```

4.1.1 Steensgaard's Analysis

Andersen's analysis has cubic time complexity. A coarser, near-linear time alternative is Steensgaard's analysis. In Banshee, Steensgaard's analysis can be implemented using the Term sort. Modify the specification for Andersen's analysis by changing `T`'s sort, eliminating the duplicate `T` field in the *ref* constructor, and removing variance annotations:

```
specification steensgaard : STEENSGAARD =
  spec
    data location : set
    data T : term = ref of label * T
  end
```

This example raises a question: How many modifications to the constraint generator and mapping function (the manually written parts of the analysis) must we make to support the changed specification? This is an important practical consideration, as it effects the ability to quickly investigate the impact of a particular design choice. There are two answers to this question. For a prototype implementation over a toy language, say one with just the operations shown in Figure 4.1, no changes are required. Although the sort of the `T` data declaration has changed, many of the common operations across sorts share the same names. In such a prototype, one

could simply re-run BANSHEE with the specification changes and test the resulting analysis. On the other hand, to completely handle a complex language such as C, a small number changes must be made. In our full implementation, the `T` declaration includes a second constructor to enable sound handling of function pointers. Since term solutions are restricted to a single head constructor, a complete implementation of Steensgaard's analysis must introduce product types representing a combination of function and data pointers (this is the approach taken in [Ste96]). Despite these limitations, we believe that BANSHEE greatly simplifies the task of refining analyses, especially when compared to the prospect of performing this transformation manually.

4.1.2 One Level Flow

Experience shows that the lack of subtyping in Steensgaard's analysis leads to many spurious points-to relations. Another proposal is to use one level of subtyping. Restricting subtyping to one level has been shown experimentally to be nearly as accurate as full subtyping [Das00].

Once again, altering the specification to support the new analysis is simple:

```
specification olf : OLF
  spec
    data location : set
    data T : flowterm = ref of +label * T
  end
```

Recall that the `location` field models a set of abstract locations in our analysis. Making this field covariant allows subtyping at the top level. However, notice that the `T` field is nonvariant. This change restricts subtyping to the top level: at lower levels, the engine performs unification. An alternative explanation of this signature is that it implements the following sound rule for subtyping in the presence of updateable

references [AC96]:

$$\frac{\ell_x \subseteq \ell_y \qquad \mathcal{X}_{\ell_x} = \mathcal{X}_{\ell_y}}{ref(\ell_x, \mathcal{X}_{\ell_x}) \leq ref(\ell_y, \mathcal{X}_{\ell_y})} \text{ (sub-ref)}$$

4.1.3 Receiver Class Analysis

Now that we have thoroughly explored the effect of subtyping on the precision of the points-to relation, we focus on adding new capabilities to the analysis. In this case, we use the points-to information as the basis of a receiver class analysis (RCA) for an object-oriented language with explicit pointer operations.

RCA computes a static approximation of the set of classes each expression in the program can evaluate to. In a language like C++, the analysis must also use points-to information to track the effects of pointer operations. For example, consider the following program:

```
a = new C;
x = &a;
*x = new D;
```

In this case, we need to know that variable `x` points to `a` in order to conclude that `a` may contain a `D` object after the third assignment.

In addition to modeling object initialization and pointer operations, our analysis must accurately simulate the effects of method dispatch. To accomplish these tasks, new constructors representing class definitions and dispatch tables are added to our points-to analysis specification.

For simplicity of presentation, we assume that methods in this language have a single formal argument in addition to the implicit `this` parameter. We also assume that the inheritance tree for the program has been flattened as a preprocessing step, such that if class `B` extends `A`, `B` contains definitions for methods in `A` not overridden in

B. Here is the BANSHEE specification for this example, assuming we chose Andersen's analysis as our base points-to analysis:

```
specification rca : RCA =
  spec
    data location : set
    data T : set = ref of +location * −T * +T
                 | class of +location * +dispatch
    and dispatch : row(method)
    and method : set = fun of −T * −T * +T
  end
```

The `class` constructor contains a location field containing the name of the class and a `dispatch` field representing the dispatch table for objects of that class. Notice that `dispatch` uses the Row sort. We model an object's dispatch table as a collection of methods (each in turn modeled by the `method` constructor) indexed by name. Given a dispatch expression like `e.foo()`, our analysis should compute the set of classes that `e` may evaluate to, search each class's dispatch table for a method named `foo`, and execute it (abstractly) if it exists. Methods are modeled by the `fun` constructor. Methods model the implicit `this` parameter with the first `T` field, the single formal parameter by the second `T` field, and a single return value by the third `T` field. Recall that the function constructor must be contravariant in its domain and covariant in its range, as reflected in the specification.

For this approach to work, our dispatch table abstraction must maintain a mapping between method names and `method` terms. This mapping is accomplished using the Row sort.

Figure 4.2 shows the new rules for handling object initialization and method dispatch. These rules in conjunction with the rules in Figure 4.1 comprise our receiver class analysis. We discuss the additional rules in turn. For a class `C`, rule (New)

$$\frac{}{\Gamma \vdash \texttt{new C} : ref(0, class(\ell_C, < \ell_{m_i} : fun(\mathcal{X}_{this}, \mathcal{X}_{arg}, \mathcal{X}_{ret}) \ldots >), \overline{1})} \text{ (New)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 \\ \tau_1 \subseteq ref(1, \mathcal{T}_1, \overline{0}) & \tau_2 \subseteq ref(1, \mathcal{T}_2, \overline{0}) \\ \multicolumn{2}{c}{\mathcal{T}_1 \subseteq class(1, < \ell_m : fun(\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{ret}) \ldots >)}\end{array}}{\Gamma \vdash e_1 \texttt{.m}(e_2) : ref(0, \mathcal{T}_{ret}, \overline{1})} \text{ (Dispatch)}$$

Figure 4.2: Rules for receiver class analysis (add to the rules from Figure 4.1).

returns a `class` expression with label ℓ_C. The `dispatch` component of this expression is a row which maps labels ℓ_{m_i} to `method`s for each method $\texttt{m}_i$ defined in `C`. To remain consistent with the type judgments for Andersen's analysis (where the result of each type judgment is an l-value) we wrap the resulting class in a `ref` constructor. Note that since our analysis is context-insensitive, each instance of `new C` occurring in the program yields the same `class` expression, which is created when the definition of `C` is analyzed. In (Dispatch), e_1 is analyzed and assumed to contain a set of `class`es.[1] For each class which defines a method named `m` (i.e. the associated `dispatch` row contains a mapping for label ℓ_m), the corresponding method body is selected, and constrained such that the actual parameters flow into the formal parameters, and the return value of the function (lifted to an l-value) is the result of the entire expression.

This specification takes advantage of BANSHEE's strong type system. With BANSHEE's compilation approach, the type of `method` expressions is incompatible with the type of `T` expressions, despite the fact that both are kinds of `Set` expressions. Thus, BANSHEE eliminates the possibility of (e.g.) a `class` abstraction appearing in a `dispatch` table.

[1] In practice we use projections to handle the case where e_1 contains both pointers and objects.

4.2 Experimental Results

To demonstrate the scalability and performance of Banshee, we implemented field- and context-insensitive Andersen's analysis for C and tested it on several large benchmarks.[2] We also ran the same analysis implemented with the Bane toolkit. While Bane is written in SML and Banshee in C, among other low-level differences, the comparison does demonstrate Banshee's engineering improvements. Table 4.1 shows wall clock execution times in seconds for the benchmarks. Benchmark size is measured in preprocessed lines of code (the two largest benchmarks, gimp and Linux, are approximately 430,000 and 2.2 million source lines of code, respectively). We compiled the Linux benchmark using a "default" configuration that answers "yes" to all interactive configuration options. All reported times for this experiment include the time to calculate the transitive lower bounds of the constraint graph, which simulates points-to queries on all program variables [FFSA98]. Parse times are not included. Interestingly, a significant fraction of the analysis time for the Bane implementation is spent in garbage collection, which may be because almost all of the objects allocated during constraint resolution (nodes and edges in the constraint graph) are live for the entire analysis. We also report (in the column labeled Bane) the wall clock execution time for Andersen's analysis exclusive of garbage collection. The C front-end used in the Bane implementation cannot parse the Linux source, so no number is reported for that benchmark. Although it is difficult to compare to other implementations of Andersen's analysis using wall-clock execution time, we note that our performance appears to be competitive with the fastest hand-optimized Andersen's implementation for answering all points-to queries [HT01b].

[2]These experiments were run on a 2.80 GHz Intel Xeon machine with 4 GB of memory running Red Hat Linux.

Benchmark	Description	LOC	Andersen(s)		
			BANE(+gc)	BANE	BANSHEE
gs	Ghostscript	437211	35.5	27.0	6.9
spice	Circuit simulator	849258	14.0	11.3	3.0
pgsql	PostgreSQL	1322420	44.8	34.9	6.0
gimp	GIMP v1.1.14	7472553	1688.8	962.9	20.2
linux	Linux v2.4	3784959	—	—	54.5

Table 4.1: Benchmark data for Andersen's analysis

4.2.1 Backtracking

We also evaluated the strategy described for backtracking-based incremental analysis (Section 3.3) by running Andersen's analysis on CQual, a type qualifier inference tool. CQual contains approximately 45,000 source lines of C code (250,000 lines preprocessed). We chose CQual because of our familiarity with its build process: without manual guidance, it is difficult to compile and analyze multiple versions of a code base spanning several years. We looked at each of the 13 commits made to CQual's CVS repository from November 2003 to May 2004 that modified at least one source file and compiled successfully. For each commit we report three different numbers (Figure 4.3(a)):[3]

- Column 3: Andersen's analysis run from scratch; this is the analysis time assuming no backtracking is available.
- Column 4: Incremental Andersen's analysis, assuming that analysis of the previous commit is available. To compute this number, we take the previous analysis, backtrack (pop the analysis stack) to the earliest modified file, and re-analyze all files popped off of the stack, placing the modified files on top (as described in Section 3.3). The initial stack (for the full analysis of the first commit)

[3]These times do not include the time to serialize or deserialize the constraints.

contained the files in alphabetical order.

- Column 5: Incremental Andersen's analysis, assuming that the files modified during this commit are already on the top of the analysis stack. To compute this number, we pop and reanalyze just the modified files.

Column 4 gives the expected benefit of backtracking if the analysis is run only once per commit. However, if the developer runs the analysis multiple times before committing (each time the code is compiled, or each time the code is edited in an interactive environment) then Column 5 gives a lower bound on the eventual expected benefit. To see this, assume that a single source file is modified during an editing task. The first time the analysis is run, that source file may be placed on the bottom of the stack, so after a program edit, a complete reanalysis might be required. Subsequently, however, that file is on top of the stack and we only pay the cost of reanalyzing a single file. In general, if n files are modified in an editing task, at worst we analyze the entire code base n times (to move each file one by one from the bottom to the top of the stack) and subsequently only pay (at most) the cost to reanalyze the n modified files.

Backtracking, then, can be an effective incremental analysis technique as long as only a small fraction of the files in a code base is modified per editing task. Table 4.2 shows this property holds for CQual. To test this hypothesis on a larger, more active code base with more than a few developers, we looked at the CVS history for OpenSSL, which contains over 4000 commits that modify source code. For each commit, we recorded the percentage of source files modified. Figure 4.3 shows a plot of the sorted data. The percentage of files modified obeys a power law: very infrequently, between 5 and 25 percent of the files are modified, but in the common case, less than .1 percent of the files are modified. We have confirmed similar distributions hold for other code bases as well.

Commit Date	Files Modified	All Files(s)	Cross Commit(s)	Modified Only(s)
11-16	0/58	2.0	-	-
11-17	1/58	2.0	0.9	0.05
12-03	1/58	2.0	0.9	0.15
12-10	1/58	2.0	1.0	0.04
12-11	1/58	2.3	2.0	0.09
12-12	5/58	2.3	1.8	0.51
2-29	1/58	2.0	0.5	0.08
3-05	1/58	2.0	0.9	0.06
3-05	25/59	2.0	2.0	1.0
3-05	5/60	2.0	2.4	0.27
3-11	4/60	2.4	1.0	0.5
3-22	3/61	2.0	0.14	0.14
5-03	2/61	2.0	1.2	0.13

Table 4.2: Data for backtracking experiment

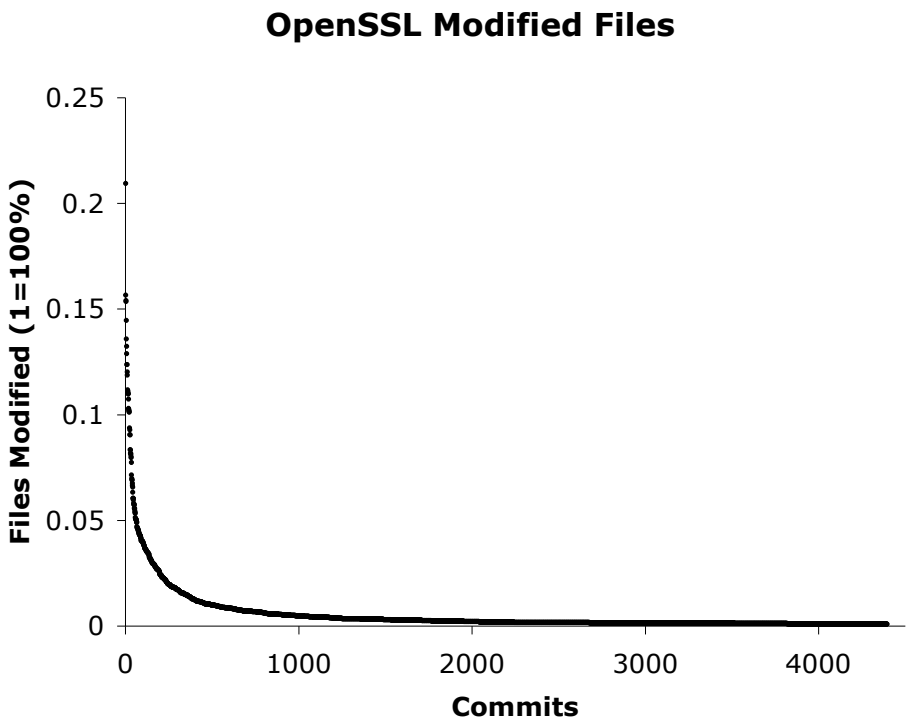


Figure 4.3: OpenSSL modified files per commit

Chapter 5

Set Constraints and CFL Reachability

In this chapter, we present a result that allows context-free language (CFL) reachability problems to be expressed using set constraints. A number of program analyses have been formulated as CFL reachability problems, including applications such as type-based polymorphic flow analysis, field-sensitive points-to analysis, and interprocedural dataflow analysis [RF01; Rep98; RHS95]. Getting CFL-based implementations to scale has proved as tricky as implementing set constraints, leading to new algorithms and optimization techniques. Using our result, algorithms for solving set constraints can be used to implement these analyses.

Set constraints and CFL reachability were shown to be closely related in work by Melski and Reps [MR00]. While their result is useful for understanding the conceptual similarity between the two problems, it does not serve as an implementation strategy; as with many reductions, the cost of encoding one problem in the other formalism proves to be prohibitive in practice.

Furthermore, the Melski-Reps reduction does not show how to relate the state-of-the-art algorithms for set constraints to the state-of-the-art algorithms for CFL reachability. Optimizations in one formalism are not preserved across the reduction

to the other formalism. Demand-driven algorithms for CFL reachability do not automatically lead to demand algorithms for set constraints (except by applying the reduction first).

Our insight is based on the observation that almost all of the applications of CFL reachability in program analysis are based on *Dyck languages*, which contain strings of matched parentheses. For Dyck languages, we show an alternative reduction from CFL reachability to set constraints that addresses the issues mentioned above. The principal contributions presented in this chapter are as follows:

- We give a novel construction for converting a Dyck-CFL reachability problem into a set constraint problem. The construction is simpler than the more general reduction described in [MR00]. In fact, the constraint graphs produced by our construction are nearly isomorphic to the original CFL graph.

- We show that on real polymorphic flow analysis problems, our implementation of Dyck-CFL reachability based on set constraints remains competitive with a highly tuned CFL reachability engine. This is somewhat surprising since that implementation contains optimizations that exploit the specific structure of the CFL graphs that arise in flow analysis.

These results have several consequences:

- Our results show that it is possible to use an off-the-shelf set constraint solver to solve Dyck-CFL reachability problems without suffering the penalties we normally expect when using a reduction. Furthermore, reasonable performance can be expected without having to tune the solver for each specific application.

- We believe that our reduction bridges the gap between the various algorithms for Dyck-CFL reachability [HRS95] and the algorithms for solving set constraints. In light of our reduction, it seems that the distinction between these problems is

illusory: the manner in which a problem is specified (as Dyck-CFL reachability vs. set constraints) is orthogonal to other algorithmic issues (e.g. whether the solver is online vs. offline, demand-driven vs. exhaustive). This has some important consequences– for example, we can immediately apply our backtracking scheme to to solve Dyck-CFL reachability problems incrementally.

- Furthermore, our reduction shows how techniques used to solve set constraints (inductive form and cycle elimination [FFSA98]) can be applied to Dyck-CFL reachability problems.

The remainder of this chapter is structured as follows. In Section 5.1 we briefly introduce CFL reachability. In Section 5.2 we review the reduction from CFL reachability to set constraints. Section 5.3 presents our specialized reduction from Dyck-CFL reachability to set constraints. In Section 5.4 we use polymorphic flow analysis as a case study for our reduction.

5.1 CFL and Dyck-CFL Reachability

In this section we review basic material on CFL reachability and Dyck-CFL reachability and describe an approach to solving these problems. Readers familiar with CFL reachability may wish to skip this section, which is largely standard.

Let $CFG = (T, N, P, S)$ be a context free grammar with terminals T, nonterminals N, productions P and start symbol S. Let G be a directed graph with edges labeled by elements of T. The notation $A\,\langle u, v\rangle$ denotes an edge in G from node u to node v labeled with symbol A. Each path in G defines a word over T by concatenating, in order, the labels of the edges in the path. A path in G is an S-path if its word is in the language of CFG. The *all-pairs CFL reachability problem* determines the pairs of vertices (u, v) where there exists an S-path from u to v in G.

$$
\begin{array}{rcl}
G \cup \{B\,\langle u, u'\rangle, C\,\langle u', v\rangle\} & \Leftrightarrow & \text{add } A\,\langle u, v\rangle \text{ for each production of the form } A \to B\ C \\
G \cup \{B\,\langle u, v\rangle\} & \Leftrightarrow & \text{add } A\,\langle u, v\rangle \text{ for each production of the form } A \to B \\
G & \Leftrightarrow & \text{add } A\,\langle u, u\rangle \text{ for each production of the form } A \to \epsilon
\end{array}
$$

Figure 5.1: Closure rules for CFL reachability

Now let $O = \{(_1, \ldots, (_n\}$ and $C = \{)_1, \ldots,)_n\}$ be two disjoint sets of terminals (interpreted as opening and closing symbols, respectively). The subscripts are the terminals' *indices*. The Dyck language over $O \cup C \cup \{s\}$ is generated by a context free grammar with productions of the following form:

$$S \to S\ S \mid (_1\ S\ )_1 \mid \cdots \mid (_n\ S\ )_n \mid \epsilon \mid s$$

(where s is a distinguished terminal not in O or C). The *all-pairs Dyck-CFL reachability problem* is the restriction of the all-pairs CFL reachability problem to Dyck languages.

Figure 5.1 shows the closure rules for the CFL and Dyck-CFL reachability algorithms. The rules assume that the CFL grammar has been normalized such that no right-hand side of any production contains more than two symbols[1]. A naive CFL reachability algorithm might apply the rules as follows: whenever the CFL graph matches the form on the left-hand side of the rule, add the edges indicated by the right-hand side of the rule; iterate this process until no rule induces a new edge addition. The application of these rules produces a graph $closure(G)$ that contains all possible edges labeled by nonterminals in the grammar. To check whether there is an S-path from nodes u to v in G, we simply check for the existence of an edge $S\,\langle u, v\rangle$ in $closure(G)$.

In general, the all-pairs CFL-reachability problem can be solved in time $O(|T \cup$

[1]For instance, by converting the grammar to Chomsky normal form.

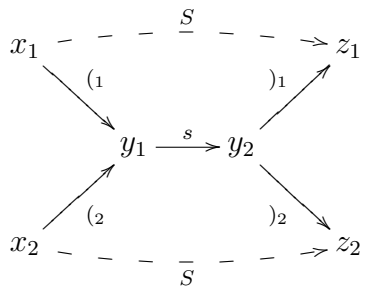


Figure 5.2: Example Dyck-CFL reachability problem

$N|^3 n^3$) for a graph with n nodes.

Figure 5.2 shows an example of a Dyck-CFL reachability problem. The dashed lines show the edges added by the computation of $closure(G)$. The edges show that there are S-paths between nodes x_1 and z_1, and x_2 and z_2. Self-loop S-paths derived from the production $S \rightarrow \epsilon$ are not shown.

5.2 The Melski-Reps Reduction

In this section we review the reduction from CFL reachability to set constraints [MR97; MR00]. We assume that the CFL grammar is normalized so that the right-hand side of every production contains at most two symbols.

The construction encodes each node u in the initial CFL graph G with one set variable $\mathcal{U}$, one nullary constructor n_u and the constraint

$$n_u \subseteq \mathcal{U}$$

In encoding the graph, the goal is to represent an edge $A\,\langle u, v\rangle$ in the closed graph by the presence of a term $c_A(\mathcal{V})$, where c_A is a unary constructor corresponding to symbol A, in the least solution of $\mathcal{U}$. Accordingly, an initial edge of the form $a\,\langle u, v\rangle$

in G is captured by a constraint

$$c_a(\mathcal{V}) \subseteq \mathcal{U}$$

This completes the representation of the initial graph. The next step of the reduction is to encode the productions of the CFL grammar. Since edges are encoded as constructed terms, the notion of "following an edge" is captured by projection. The left-hand side of each production tells us which edge we should add if we follow edges labeled by the right-hand side. For instance, a binary production of the form $A \to B\ C$ says that we should add an A edge from node u to any nodes reached by following a B edge from u and then following a C edge. The set constraint that encodes this rule is

$$c_A(c_C^{-1}(c_B^{-1}(\mathcal{U}))) \subseteq \mathcal{U}$$

For example, the CFL graph

$$u \xrightarrow{B} u' \xrightarrow{C} v$$

and the production $A \to B\ C$ result in a system of set constraints containing the following inclusion relations:

$$\begin{gathered} n_u \subseteq \mathcal{U} \\ c_C(\mathcal{V}) \subseteq \mathcal{U}' \\ c_B(\mathcal{U}') \subseteq \mathcal{U} \\ c_A(c_C^{-1}(c_B^{-1}(\mathcal{U}))) \subseteq \mathcal{U} \end{gathered}$$

These constraints imply the desired relationship $c_A(n_v) \subseteq \mathcal{U}$, which represents the edge $A\,\langle u, v\rangle$. Note these constraints only encode each production locally: similar constraints must be generated for every node in the graph.

Besides binary productions, productions of the form $A \to B$ and $A \to \epsilon$ may occur in the grammar. Productions of the first form are encoded locally for each node u by

the constraint

$$c_A(c_B^{-1}(\mathcal{U})) \subseteq \mathcal{U}$$

representing the fact that any B edge from node u is also an A edge. Finally, productions of the form $A \to \epsilon$ are encoded locally for each node u by the constraint

$$c_A(\mathcal{U}) \subseteq \mathcal{U}$$

Correctness of the reduction is proved by showing there is an S-path from node u to node v in $closure(G)$ if and only if $c_S(n_v) \subseteq \mathcal{U}$ is in the least solution of the constructed system of constraints.

5.3 A Specialized Dyck Reduction

This section presents our reduction, which is specialized to Dyck-CFL reachability problems. We first explain the intuition behind our approach. We then describe the specifics of the encoding, sketch a proof of correctness, and provide a complexity argument. Finally, we provide experimental results suggesting that the specialized reduction is more efficient in practice than the Melski-Reps reduction applied to Dyck-CFL grammars.

The idea behind our reduction is that the set constraint closure rules can encode Dyck language productions. The intuition is straightforward: the rule $S \to S\ S$ corresponds to the transitive closure rule, and the rules $S \to (_i\ S\ )_i$ correspond to matching constructed terms with projection patterns. By encoding the original Dyck-CFL graph as a system of set constraints in a way that exploits this correspondence, we can avoid encoding the productions of the grammar at each node. This leads to a more natural and compact encoding of the input graph, and better performance in applications where the number of parenthesis kinds (hence, the number of productions)

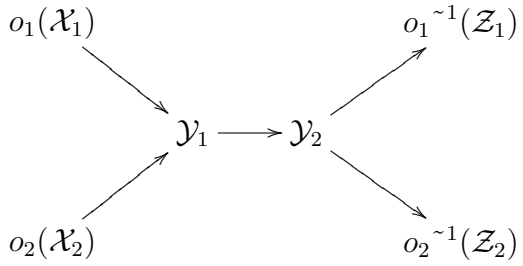


Figure 5.3: The constraint graph corresponding to the Dyck-CFL graph from Figure 5.2 using the reduction from Section 5.3

is not a constant.

The first step of the encoding is unchanged. We represent each node u with a variable $\mathcal{U}$ and a nullary constructor n_u. We connect these with constraints of the form

$$n_u \subseteq \mathcal{U}$$

Edges are encoded so that subset relationships $\mathcal{U} \subseteq \mathcal{V}$ between variables in the constraint system represent discovered $S\,\langle u, v\rangle$ edges in the input graph. Since we are only considering Dyck languages, there are only three kinds of labeled edges to consider:

1. For each $s\,\langle u, v\rangle$ edge in the input graph, we add the constraint $\mathcal{U} \subseteq \mathcal{V}$ (where $\mathcal{U}$ and $\mathcal{V}$ are the variables corresponding to nodes u and v).

2. For each $(_i\langle u, v\rangle$ edge in the input graph, we create a unary constructor o_i and add the constraint $o_i(\mathcal{U}) \subseteq \mathcal{V}$.

3. For each $)_i\,\langle u, v\rangle$ edge in the input graph, we add the constraint $\mathcal{U} \subseteq o_i{}^{\sim 1}(\mathcal{V})$.

Notice that there appears to be a slight asymmetry in the reduction: edges labeled with $(_i$ symbols are encoded using constructed terms, while edges labeled with $)_i$

symbols are encoded using projections. Naively, one might attempt to devise a "symmetric" reduction by encoding an edge $)_i \langle u, v \rangle$ with a constraint $\mathcal{U} \subseteq o_i(\mathcal{V})$, making a correspondence between the production $S \to (_i\ S\ )_i$ and constructor matching rule in Figure 2.1. However, this approach would yield an inconsistent system of set constraints for any interesting graph. For example, the graph in Figure 5.2 would result in a constraint system containing the constraints $o_1(\mathcal{X}_1) \subseteq \mathcal{Y}_1 \subseteq \mathcal{Y}_2 \subseteq o_2(\mathcal{Z}_2)$, which are clearly inconsistent. We avoid this problem by using projection patterns to represent $)_i$ edges. Thus, the graph in Figure 5.2 results in a constraint system containing $o_1(\mathcal{X}_1) \subseteq \mathcal{Y}_1 \subseteq \mathcal{Y}_2 \subseteq o_2{}^{\sim 1}(\mathcal{Z}_2)$, which is consistent.

This construction only introduces one constraint per node and one constraint per edge of the original CFL graph. In contrast, the reduction described in Section 5.2 introduces $O(k)$ constraints per node (where k is the number of parenthesis kinds in the Dyck grammar) to encode the grammar productions locally, in addition to the cost of encoding the graph edges.

To solve the all-pairs Dyck-CFL reachability problem, we apply the rewrite rules in Figure 2.1 to the resulting constraint system. As we will show, there is an S-path from nodes u to v in the initial graph if and only if $n_u \subseteq \mathcal{V}$ in the least solution of the constructed constraint system.

Figure 5.3 shows the constraint graph corresponding to the Dyck-CFL reachability problem shown in Figure 5.2 using this reduction. The nullary constructors are omitted for clarity.

5.3.1 Correctness

We sketch a proof of correctness for the specialized reduction. The reduction is correct if the solution to the constructed set constraint problem gives a solution to the original Dyck-CFL reachability problem.

Theorem 5.3.1. *Let G be an instance of the Dyck-CFL reachability problem, and $\mathcal{C}$ be the collection of set constraints constructed as above to represent G. Then there is an edge $S\langle u, v\rangle$ in closure(G) if and only if $n_u \subseteq \mathcal{V}$ is present in $LS(\mathcal{C})$.*

The theorem follows immediately from the following lemmas:

Lemma 5.3.2. *Given C and G as in Theorem 5.3.1, let $\mathcal{C}'$ denote the conjunction of $\mathcal{C}$ with the system of constraints introduced by applying the resolution rules shown in Figure 2.1 along with the transitive closure rule $e \subseteq \mathcal{X} \wedge \mathcal{X} \subseteq e' \Rightarrow e \subseteq e'$. Then there is an edge $S\langle u, v\rangle$ in closure(G) if and only if $\mathcal{U} \subseteq \mathcal{V}$ is present in $\mathcal{C}'$.*

Proof. We apply the set constraint resolution rules in lock-step with the CFL closure rules, and show that for every S-edge added to *closure*(G), there is a corresponding variable-variable constraint that can be added to $\mathcal{C}'$. This approach requires us to normalize the Dyck-CFL grammar in a very specific way:

$$
\begin{aligned}
S &\rightarrow S\ S \\
S &\rightarrow L_i\ R_i \\
L_i &\rightarrow (_i \\
L_i &\rightarrow L_i\ S \\
R_i &\rightarrow)_i \\
R_i &\rightarrow S\ R_i \\
S &\rightarrow s \\
S &\rightarrow \epsilon
\end{aligned}
$$

It is easy to see that the above grammar derives the same set of strings as the original Dyck-CFL grammar. The intuition here is that normalizing the Dyck-CFL grammar this way causes the CFL closure rules to add edges in exactly the same way as the set constraint resolution rules.

Constraints	New Constraints	Production	Edges	New Edge
$\mathcal{X} \subseteq \mathcal{Y}$ $\wedge \mathcal{Y} \subseteq \mathcal{Z}$	$\mathcal{X} \subseteq \mathcal{Z}$	$S \rightarrow S\ S$	$S\langle x, y\rangle$ $\wedge S\langle y, z\rangle$	$S\langle x, z\rangle$
$o_i(\mathcal{X}) \subseteq \mathcal{Y}$ $\wedge \mathcal{Y} \subseteq \mathcal{Z}$	$o_i(\mathcal{X}) \subseteq \mathcal{Z}$	$L_i \rightarrow L_i\ S$	$L_i\langle x, y\rangle$ $\wedge S\langle y, z\rangle$	$L_i\langle x, z\rangle$
$\mathcal{X} \subseteq \mathcal{Y}$ $\wedge \mathcal{Y} \subseteq o_i^{\sim 1}(\mathcal{Z})$	$\mathcal{X} \subseteq o_i^{\sim 1}(\mathcal{Z})$	$R_i \rightarrow S\ R_i$	$S\langle x, y\rangle$ $\wedge R_i\langle y, z\rangle$	$R_i\langle x, z\rangle$
$o_i(\mathcal{X}) \subseteq \mathcal{Y}$ $\wedge \mathcal{Y} \subseteq o_i^{\sim 1}(\mathcal{Z})$	$o_i(\mathcal{X}) \subseteq o_i^{\sim 1}(\mathcal{Z})$ $\wedge \mathcal{X} \subseteq \mathcal{Z}$	$S \rightarrow L_i\ R_i$	$L_i\langle x, y\rangle$ $\wedge R_i\langle y, z\rangle$	$S\langle x, z\rangle$

Table 5.1: Constraints added to $\mathcal{C}'$ and the corresponding edges in $closure(G)$

We now formalize this intuition. Consider the following order for adding edges to $closure(G)$ and constraints to $\mathcal{C}'$:

1. Add all edges implied by the productions $S \rightarrow s$, $S \rightarrow \epsilon$, $R_i \rightarrow\ )_i$, and $L_i \rightarrow\ (_i$. Note that there are only finitely many edges that can be added, and that they are all added in this step.

2. Add all edges implied by the productions $S \rightarrow S\ S$, $S \rightarrow L_i\ R_i$, $L_i \rightarrow L_i\ S$, and $R_i \rightarrow S\ R_i$, and add the corresponding constraints to $\mathcal{C}'$ according to Table 5.1.

In the first step, notice that the added S edges correspond to variable-variable edges that already exist in the initial system of constraints $\mathcal{C}$ (refer to the first rule of the reduction in Section 5.3). In the second step, notice that for every S edge added to $closure(G)$, there are corresponding constraints that must exist in $\mathcal{C}'$ that lead to the appropriate variable-variable constraint.

□

Lemma 5.3.3. *Given $\mathcal{C}$ and G as in Theorem 5.3.1, let $\mathcal{C}'$ denote the conjunction of $\mathcal{C}$ with the system of constraints introduced by applying the resolution rules shown in Figure 2.1 along with the transitive closure rule $e \subseteq \mathcal{X} \wedge \mathcal{X} \subseteq e' \Rightarrow e \subseteq e'$. Then the constraint $n_u \subseteq \mathcal{V}$ implies the existence of a constraint $\mathcal{U} \subseteq \mathcal{V}$ in $\mathcal{C}'$.*

Proof. Clearly, the rules in Figure 2.1 cannot add a constraint of the form $n_u \subseteq \mathcal{V}$, so such a constraint is either present initially in $\mathcal{C}$ or it was added to $\mathcal{C}'$ by transitive closure. In the former case, $\mathcal{U} = \mathcal{V}$ and the lemma holds trivially. In the latter case, we can show (by induction on the number of applications of the transitive closure rule) that there are variables $\mathcal{X}_1, \ldots, \mathcal{X}_n$ with $n_u \subseteq \mathcal{U} \subseteq \mathcal{X}_1 \subseteq \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{V}$ present in $\mathcal{C}'$. Then by the transitive closure rule, $\mathcal{U} \subseteq \mathcal{V}$ must be present in $\mathcal{C}'$. □

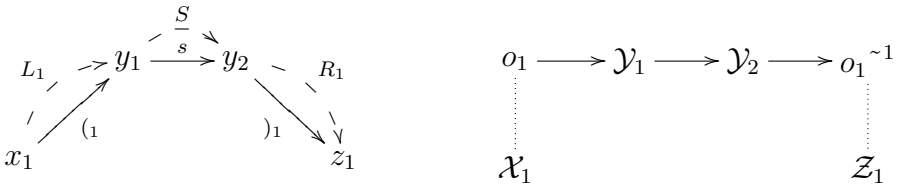
We now prove Theorem 5.3.1 using the two lemmas. To show that $S\langle u, v\rangle \in \mathit{closure}(G)$ implies $n_u \subseteq \mathcal{V}$ is present in $\mathcal{C}'$, we note that Lemma 5.3.2 guarantees the existence of a constraint $\mathcal{U} \subseteq \mathcal{V}$ in $\mathcal{C}'$. Our construction guarantees an initial constraint $n_u \subseteq \mathcal{U}$, so by the transitive closure rule, $n_u \subseteq \mathcal{V}$ exists in $\mathcal{C}'$. To show that a constraint $n_u \subseteq \mathcal{V}$ in $\mathcal{C}'$ implies $S\langle u, v\rangle \in \mathit{closure}(G)$, we appeal to Lemma 5.3.3, which guarantees the existence of a constraint $\mathcal{U} \subseteq \mathcal{V}$ in $\mathcal{C}'$. Appealing once again to Lemma 5.3.2, we have that $S\langle u, v\rangle \in \mathit{closure}(G)$.

Figure 5.4 shows the proof technique in action for a fragment of the graph from Figure 5.2 and the corresponding constraint graph from our reduction. For clarity, terms in the graph are represented as trees.

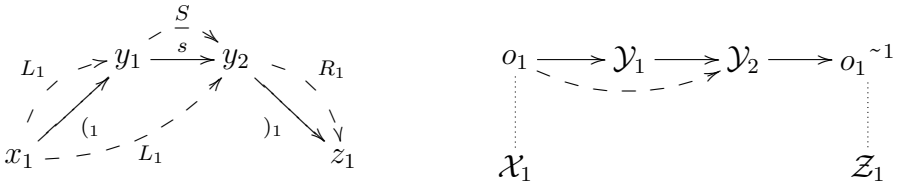
5.3.2 Complexity

We first discuss the running time to solve the all-pairs Dyck-CFL reachability problem using the generic CFL reachability algorithm outlined in Section 5.1. In general, the generic algorithm has complexity $O(|T \cup N|^3 n^3)$. Applying this result directly to an instance of the Dyck-CFL problem with k parenthesis kinds and n graph nodes yields an $O(k^3 n^3)$ running time.

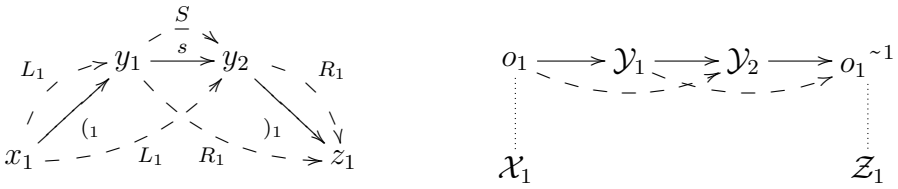
However, we can sharpen this result by specializing the complexity argument to the Dyck-CFL grammar. The running time of the generic CFL reachability algorithm is dominated by the first rule in Figure 5.1. In this step, each edge in the graph might



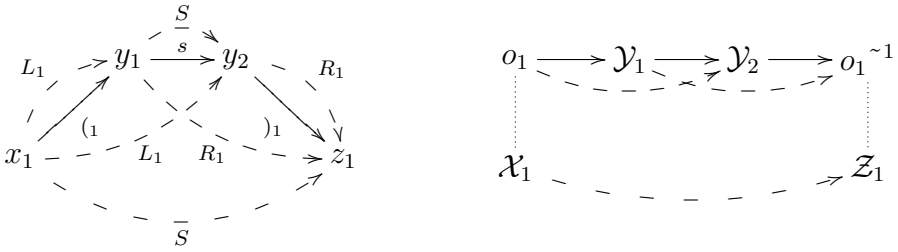
(a) The graph after the edges from step 1 have been added (ϵ edges and nullary constructors ommitted)



(b) The addition of edge $L_1 \langle x_1, y_2 \rangle$ and the corresponding constraint $o_1(\mathcal{X}_1) \subseteq \mathcal{Y}_2$



(c) The addition of edge $R_1 \langle y_1, z_1 \rangle$ and the corresponding constraint $\mathcal{Y}_1 \subseteq o_1{}^{\sim 1}(\mathcal{Z}_1)$



(d) The final edge $S \langle x_1, z_1 \rangle$ and the corresponding constraint $\mathcal{X}_1 \subseteq \mathcal{Z}_1$

Figure 5.4: The correctness proof in action for a fragment of the graph from Figure 5.2

pair with each of its neighboring edges. In other words, for a given node j, each of j's incoming edges might pair with each of j's outgoing edges. Each node in the graph might have $O(kn)$ edges labeled with $(_i$ or $)_i$ (there are k different kinds, and n potential target nodes), as well as $O(n)$ edges labeled with S. For node j, each of the $O(kn)$ incoming edges may pair with $O(n)$ outgoing edges, by a single production. Similarly, each of the $O(n)$ incoming S edges can match with $O(n)$ outgoing edges by the single $S \rightarrow S\ S$ production. Thus, the work for each node j is bounded by $O(kn^2)$. Since there are n nodes in the graph, the total work is $O(kn^3)$.

An analogous argument holds for our reduction. For the fragment of set constraints used in this report, a constraint system $\mathcal{C}$ can be solved in time $O(v^2|\mathcal{C}|)$ where v is the number of variables in $\mathcal{C}$ and $|\mathcal{C}|$ is the number of constraints in $\mathcal{C}$. Given an instance of the Dyck-CFL problem with k parenthesis kinds and n nodes, our reduction yields a constraint system with $O(n)$ variables and $O(kn^2)$ constraints, yielding an $O(kn^4)$ running time. Again, a more precise running time can be achieved by noting that for a given variable node i in the constraint graph, the rules in Figure 2.1 can apply only $O(kn^2)$ times to pairs of i's upper and lower bounds. In an n node graph, then, the total work is $O(kn^3)$.

5.3.3 Empirical Comparison

We have implemented both the Melski-Reps reduction and our reduction in order to compare the two approaches. Both implementations use Banshee the underlying set constraint solver. Our implementation of the Melski-Reps reduction uses the following

normalized form of the Dyck-CFL grammar:

$$\begin{aligned} S &\rightarrow S\ S \\ S &\rightarrow L_i\ )_i \\ L_i &\rightarrow (_i\ S \\ S &\rightarrow s \\ S &\rightarrow \epsilon \end{aligned}$$

We ran both implementations on randomly generated graphs. To test correctness, we checked that both implementations gave consistent answers to random reachability queries. We found that our implementation of the Melski-Reps reduction did not scale to large graphs, even when the number of parenthesis kinds was kept small. For example, on a 500 node graph with 7500 edges and 20 parenthesis kinds, the Melski-Reps implementation computed the graph closure in 45 seconds, while our reduction completed in less than 1 second. This prevented us from testing the Melski-Reps approach on real applications: we tried the implementation on the benchmarks in our case study (see Section 5.4) but found that none finished within 10 minutes.

5.4 Application: Polymorphic Flow Analysis

Flow analysis statically estimates creation, use, and flow of values in a program [Mos96]. Problems such as computing may-alias relationships, determining receiver class information, and resolving control flow in the presence of indirect function calls can be solved using flow analysis. As with most static analyses, flow analysis can be made more precise by treating functions polymorphically. Polymorphic flow analysis eliminates spurious flow paths that arise from conflating all call sites of a function.

Recent work has established a connection between type-based polymorphic flow analysis and CFL reachability [RF01]. As an alternative to constraint copying, sys-

tems of instantiation constraints are reduced to a CFL graph. The flow relation is described by a Dyck-CFL grammar, hence, the problem of finding the flow of values in the input program reduces to an all-pairs Dyck-CFL reachability problem.

In this section, we apply our technique to the problem of polymorphic flow analysis. For concreteness, we consider the specific problem of polymorphic tainting analysis [STFW01]. In tainting analysis, we are interested in checking whether any values possibly under adversarial control can flow to functions that expect trusted data. As a trivial example, consider the following code:

```
int main(void)
{
    char *buf;
    buf = read_from_network();
    exec(buf);
}
```

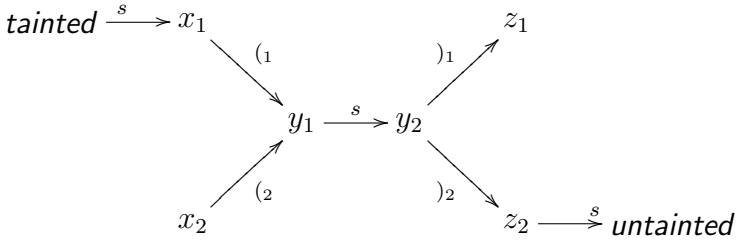
The call `exec(buf)` is probably not safe, since the buffer may come from a malicious or untrusted source. The particular tainting analysis we use employs *type qualifiers* to solve this problem [FFA99]. Briefly, we can detect errors like the example shown above by introducing two type qualifiers *tainted* and *untainted*. Data that may come from an untrusted source is given the type qualifier *tainted*, and data that must be trusted is given the type qualifier *untainted*. A type error is produced whenever a *tainted* value flows to a place that must be *untainted*.

Figure 5.5 illustrates the connection between polymorphic tainting analysis and CFL reachability with another small program. There are two calls to the identity function `id`. At the first call site, `id` is passed *tainted* data. At the second call site, `id` is passed *untainted* data. The variable `z2` is required to be *untainted*. Monomorphic flow analysis would conflate the two call sites, leading to a spurious error (namely, that

```
int id(int y1) { int y2 = y1; return y2; }

int main(void)
{
    tainted int x1;
    int z1,x2;
    untainted int z2;
    z1 = id(x1); // call site 1
    z2 = id(x2); // call site 2
}
```

(a)



(b)

Figure 5.5: (a) Example C program and (b) the corresponding Dyck CFL reachability graph

tainted data from variable `x1` reaches the *untainted* variable `z2`). Polymorphic flow analysis can distinguish these two call sites, eliminating the spurious error. The flow graph in Figure 5.5 shows the connection to Dyck-CFL reachability: each function call site is labeled with a distinct index. The flow from an actual parameter to a formal at call site i is denoted by an edge labeled $(_i$. The flow corresponding to the return value of the function is denoted by an edge labeled $)_i$. Intraprocedural assignment is denoted by s edges. In this example, the spurious flow path between x_1 and z_2 corresponds to a path whose word is $(_1\ s\ )_2$, which is not in the Dyck language.

In the remainder of this section, we discuss how to extend our technique to solve polymorphic flow analysis problems. We also discuss some optimizations that improve the performance of our reduction on polymorphic flow graphs. We conclude with an experimental assessment of our approach on a set of C benchmarks.

5.4.1 Extensions

It turns out that simple Dyck-CFL reachability isn't quite sufficient for most polymorphic flow analysis applications. In this section, we discuss two additional aspects of the problem that required us to extend our technique.

5.4.1.1 PN Reachability

The first problem is that Dyck-CFL reachability fails to capture many of the valid flow paths. Besides the matched reachability paths represented by Dyck languages, certain kinds of partially matched reachability also represent valid flow. For example, in the graph shown in Figure 5.5, there should be a valid flow path from the *tainted* qualifier to the node y_2, since it is possible for `y2` to contain a *tainted* integer at runtime. However, there is no S path between *tainted* and y_2. This issue also arises in other applications such as interprocedural dataflow analysis.

The conceptual solution to this problem is to add productions to the Dyck grammar that admit these additional partially matched paths. For our application, it turns out that we should admit all paths generated by the following grammar (see [RF01]):

$$
\begin{array}{rcl}
\mathit{Start} & \rightarrow & P\ N \\
P & \rightarrow & S\ P \\
 & | &)_i\ P \\
 & | & \epsilon \\
N & \rightarrow & S\ N \\
 & | & (_i\ N \\
 & | & \epsilon \\
S & \rightarrow & (_i\ S\ )_i \\
 & | & S\ S \\
 & | & s \\
 & | & \epsilon
\end{array}
$$

Intuitively, this grammar accepts any substring of a string in a Dyck language: P paths correspond to prefixes of Dyck strings, and N paths correspond to suffixes of Dyck strings.

Since our reduction (unlike the Melski-Reps reduction) is specialized to Dyck languages, it is not immediately obvious that we can modify our approach to accept exactly the strings in the above language. Fortunately, it turns out that we can handle these additional flow paths without fundamentally changing our reduction. We separate the problem into three subproblems. First, we tackle the problem of admitting N paths (open-paren suffixes) within our reduction. Second, we tackle the problem of admitting P paths (close-paren prefixes) within our reduction. Finally, we combine the two solutions and handle the above language in its full generality.

We first consider the problem of finding N paths in the system of constraints

produced by our reduction. Suppose we ask whether there is an N path between nodes u and v. To answer this query, we first recall that the least solution of a set variable is a regular tree language. Let $L(\mathcal{V})$ denote the language corresponding to the least solution of $\mathcal{V}$. Now consider the following tree language LN_u:

$$\begin{array}{rcl} N & \Rightarrow & n_u \\ & | & o_i(N) \end{array}$$

We claim (without proof) that there is an N path in the closed graph between nodes u and v if and only if the intersection of LN_u and $L(\mathcal{V})$ is non-empty. To see why, recall that edges of the form $({}_i\langle u, v\rangle$ are represented by constraints $o_i(\mathcal{U}) \subseteq \mathcal{V}$ in our reduction while edges of the form $S\,\langle u, v\rangle$ are represented by constraints $\mathcal{U} \subseteq \mathcal{V}$. Then an N path between nodes u and v is indicated by the presence of a term of the form $o_i(\ldots o_j(n_u))$ in the least solution of $\mathcal{V}$. The language LN_u generates exactly the terms of this form, so if $L(\mathcal{V}) \cap LN_u$ is non-empty, there is at least one such term in the least solution of $\mathcal{V}$, hence, there is an N path from u to v. As a simple example, consider the following CFL graph:

$$u \xrightarrow{(_i} x \xrightarrow{s} y \xrightarrow{(_j} v$$

and suppose we ask whether there is an N path from node u to node v. Our reduction yields a constraint system which includes the following constraints:

$$\begin{array}{c} n_u \subseteq \mathcal{U} \\ o_i(\mathcal{U}) \subseteq \mathcal{X} \\ \mathcal{X} \subseteq \mathcal{Y} \\ o_j(\mathcal{Y}) \subseteq \mathcal{V} \end{array}$$

Note that $L(\mathcal{V})$ contains the term $o_j(o_i(n_u))$, which is also an element of LN_u, indicating that there is an N path from u to v.

We now consider the problem of finding P paths in the system of constraints produced by our reduction. The technique is essentially the same as with finding N paths. There is one wrinkle, however: the $)_i$ edges that form P paths are represented as projections in the constraint system. There is no term representation of a $)_i$ edge as there is with a $(_i$ edge. The solution is straightforward: we simply add a term representation for these edges. We modify the reduction so that for each edge of the form $)_i \langle u, v \rangle$, we also add the constraint $p(\mathcal{U}) \subseteq \mathcal{V}$ (where p is a new unary constructor) to the system. Note that there is only one p constructor used for all indexed $)_i$ symbols. This is because the indices of the $)_i$ symbols are irrelevant to the P paths. Now, to check for P paths between nodes u and v, we simply check the non-emptiness of the intersection of $L(\mathcal{V})$ with the following language LP_u:

$$\begin{array}{rcl} P & \Rightarrow & n_u \\ & | & p(P) \end{array}$$

Finally, to check for PN paths from u to v, we combine the above two solutions, and check the non-emptiness of $L(\mathcal{V}) \cap LPN_u$, where LPN_u is defined as follows:

$$\begin{array}{rcl} N & \Rightarrow & P \\ & | & o_i(N) \\ P & \Rightarrow & n_u \\ & | & p(P) \end{array}$$

Note that this language consists of terms of the form

$$o_i(\ldots o_j(p(\ldots p(n_u))))$$

which precisely characterize the PN paths.

In practice, we do not actually build a tree automata recognizing the language LPN_u to compute the intersection. Instead, to check for a PN path from node u to node v, we traverse the terms in the least solution of $\mathcal{V}$, searching for n_u. We prune the search when the expanded term is no longer in the language LPN_u or when a cycle is found. Since our implementation applies the rules in Figure 2.1 online, this divides our algorithm into two phases: an exhaustive phase where all S paths are discovered, and a demand-driven phase where PN paths are discovered as reachability queries are asked[2].

5.4.1.2 Global Nodes

Another problem we face is that global variables in languages such as C must be treated specially to discover all the valid flow paths. The problem is that assignments to global variables can flow to any context, since global storage can be accessed at any program point. Conceptually, nodes corresponding to global variables should have self-loops labeled $(_i$ and $)_i$ for every index i that may appear elsewhere in the graph. This solution essentially treats globals monomorphically (see [DLFR01] for a more complete discussion). While this solution is simple and easy to implement, in practice it is too expensive to represent these edges explicitly. With an explicit representation, each global variable's upper bounds may be proportional to the size of the input program. Explicitly constructing this list is prohibitively expensive: with an explicit representation of self-edges, our implementation did not finish after 10 minutes on even the smallest benchmark.

We solved this problem by adding a new feature to the constraint solver. The added feature has a simple interpretation, and may prove useful in other contexts. We introduce *constructor groups*, which are simply user-specified sets of constructors. The

[2]A completely demand-driven algorithm could be implemented using the approach of [HT01a].

elements (constructors) of a constructor group must all have the same signature. In addition, constructor groups must be disjoint. Once a constructor group g is defined, two new types of expressions can be created. The first, called a *group projection pattern* uses the syntax $g^{\sim i}(se)$ and has essentially the same semantics as a projection pattern, except that instead of specifying a single constructor, an entire group is specified. The new rules for handling group projections are as follows:

$$\begin{aligned} \mathcal{C} \cup \{c(e_1, \ldots, e_{a(c)}) \subseteq g^{\sim i}(e)\} &\Leftrightarrow \mathcal{C} \cup \{e_i \subseteq e\} \\ &\quad \text{if } c \in g \\ \mathcal{C} \cup \{c(e_1, \ldots, e_{a(c)}) \subseteq g^{\sim i}(e)\} &\Leftrightarrow \mathcal{C} \\ &\quad \text{if } c \notin g \end{aligned}$$

Like all projection patterns, group projection patterns are R-compatible.

The second new expression is called a *group constructor expression* and uses the syntax $g(e_1, \ldots, e_n)$. Semantically, a group constructor expression $g(e_1, \ldots, e_n)$ is equivalent to the expression $c_1(e_1, \ldots, e_n) \cup c_2(e_1, \ldots, e_n) \cup \ldots \cup c_k(e_1, \ldots, e_n)$, where $c_j \in g$ for $1 \leq j \leq k$. As a simplifying assumption, we choose to make group constructor expressions L-compatible. The new rules for handling group constructor expressions are as follows:

$$\begin{aligned} \mathcal{C} \cup \{g(e_1, \ldots, e_{a(g)}) \subseteq c(e_1, \ldots, e_{a(c)})\} &\Leftrightarrow \text{no solution} \\ \mathcal{C} \cup \{g(e_1, \ldots, e_{a(g)}) \subseteq c^{\sim i}(e)\} &\Leftrightarrow \mathcal{C} \cup \{e_i \subseteq e\} \\ &\quad \text{if } c \in g \\ \mathcal{C} \cup \{g(e_1, \ldots, e_{a(g)}) \subseteq c^{\sim i}(e)\} &\Leftrightarrow \mathcal{C} \\ &\quad \text{if } c \notin g \\ \mathcal{C} \cup \{g(e_1, \ldots, e_{a(g)}) \subseteq g^{\sim i}(e)\} &\Leftrightarrow \mathcal{C} \cup \{e_i \subseteq e\} \\ \mathcal{C} \cup \{g(e_1, \ldots, e_{a(g)}) \subseteq g'^{\sim i}(e)\} &\Leftrightarrow \mathcal{C} \end{aligned}$$

To handle global nodes, we add each o_i constructor to a new group g_o. For each global node u, we add the constraint

$$\mathcal{U} \subseteq {g_o}^{\sim 1}(\mathcal{U})$$

which simulates the addition of self loops $)_i$ for each index i in the graph. Additionally, we add the constraint

$$g_o(\mathcal{U}) \subseteq \mathcal{U}$$

which simulates the addition of self loops $(_i$ for each index i in the graph.

5.4.2 Specialization and Dyck-CFL Reachability

The specialized Dyck reduction presented here can't immediately be expressed using BANSHEE's specification language, because the number of constructors needed is not known statically. However, the number of constructor *signatures* is known (each of the o constructors is a pure arity 1 Set constructor), and it turns out that we can still specialize as long as this condition holds. We use the notation $\forall i.c_i$ to denote a *parameterized constructor*, which specifies a (statically unknown) set of constructors sharing the same signature. As the notation suggests, a parameterized constructor can be *instantiated* to yield a constructor by providing a specific integer value j:

$$\forall i.c_i[j/i] \Rightarrow c_j$$

Each parameterized constructor can also be naturally associated with a constructor group containing its elements (in a sense, a parameterized constructor *is* just a constructor group). Now all that is needed is to add new concrete syntax to the BANSHEE specification language for parameterized constructors. We chose the syntax `g<c>`, which is intended to suggest a connection to universally quantified types. The syntax `g<c>` makes available a constructor group g containing the elements of

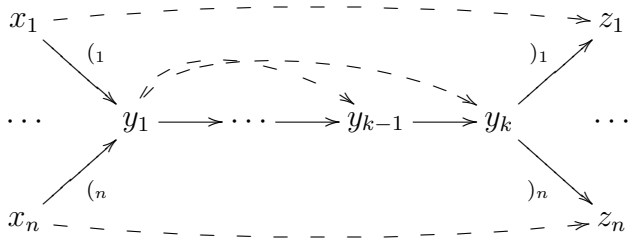


Figure 5.6: The summary edge optimization

a parameterized constructor $\forall i.c_i$. When using BANSHEE, this is the only way to create a constructor group or add constructors to a group, which trivially enforces the requirement that groups be disjoint.

With the new syntax in hand, we are able to write a specification for the Dyck CFL reduction, which is as follows:

```
specification dyckcfl_terms : DYCKCFL_TERMS =
  spec
    data node_T : set = NGroup<OPos> of +node_T
                      | Pos of +node_T
  end
```

5.4.3 Optimizations

In this subsection, we discuss some optimizations for polymorphic flow analysis applications.

5.4.3.1 Summary Edges

The CFL graphs that arise in this application have a particular structure that can be exploited to eliminate redundant work during the computation of the graph closure. An exemplar of this structure is shown in Figure 5.6. There are two important features of this graph that we exploit. First, there is a long chain of nodes $y_2, \ldots, y_{k-1}$, none

of which contain edges other than s-edges. Second, the nodes y_1 and y_k have a large number of predecessor $(_i$ edges and successor $)_i$ edges, respectively. This situation arises frequently in polymorphic flow analysis applications: the x_i nodes represent the inflow of actual arguments to the formal argument y_1 at call sites, and the z_i nodes represent the outflow of return values back to call sites from the returned value y_k.

The trick is to discover all the $S\langle x_i, z_i\rangle$ paths with as little redundant work as possible. To find these edges, the generic CFL algorithm discussed in Section 5.1 would add a total of nk new edges from each of the x_i's to each of the y_j's. Conceptually, this corresponds to analyzing the function body once per static call site.

One way to avoid this redundant work is to construct a so-called *summary edge* after analyzing the chain of y nodes once [HRS95]. This optimization works as follows: the first path explored from an x_i node triggers a new search forward from node y_1. Analyzing the chain of y nodes causes $k-1$ edge additions from y_1 to each of the remaining y_i's. When the edge $S\langle y_1, y_k\rangle$ is finally discovered, it is marked as a summary edge, and y_1 is marked as having a summary edge. When searching forward for new paths from the other x_i nodes, the summary edge $S\langle y_1, y_k\rangle$ is used, avoiding repeated analysis of the y chain. In total, the summary edge approach adds a total of $n+k-1$ new edges to discover the $S\langle x_i, z_i\rangle$ edges. Conceptually, the summary edge approach corresponds to analyzing a function body once, and copying the summarized flow to each call site. The dashed edges in Figure 5.6 show the edges added by a closure algorithm with the summary edge optimization.

Our reduction as described may perform the same work as the naive CFL closure algorithm on this example. Figure 5.7(a) shows the worst-case edge additions performed on this example in inductive form. In inductive form, the number of edge additions depends on the ordering of the $\mathcal{Y}_i$ variables. In the worst case, the variable ordering might cause inductive form to represent all variable-variable edges as as successor edges (i.e. if $o(\mathcal{Y}_i) > o(\mathcal{Y}_{i+1})$ for every i in the chain). On the other

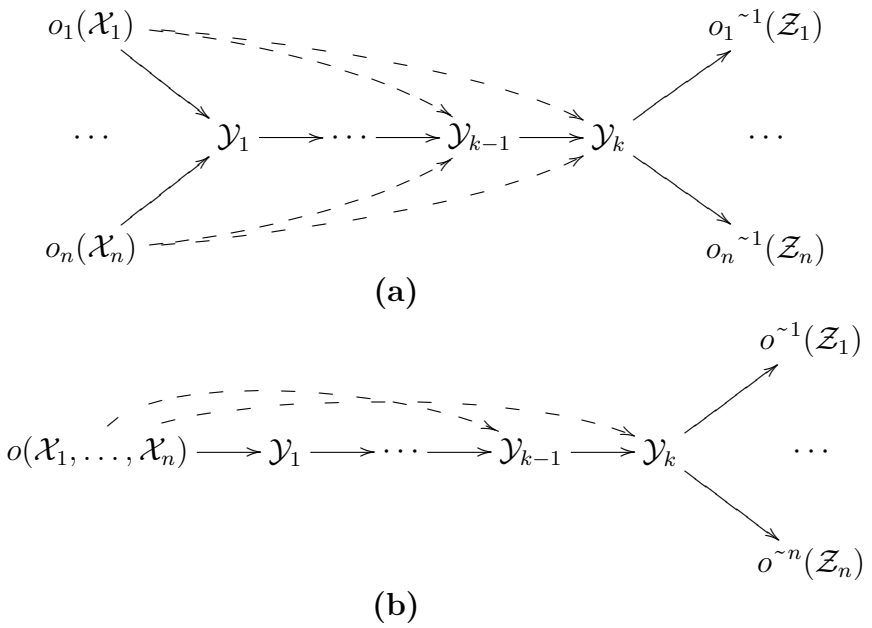


Figure 5.7: Edges added by the set constraint solver for the graph in Figure 5.6 using (a) the standard reduction and (b) the reduction with the clustering optimization

hand, if the ordering is such that some of the variable-variable edges are represented as predecessor edges, inductive form can reduce the number of edge additions. Under the best variable ordering, inductive form adds the same number of edges as the summary edge optimization.

It is also possible to modify the reduction so that the number of edge additions is minimized regardless of the variable ordering. The key is to "cluster" the variables corresponding to the x_i nodes into a single n-ary constructed term instead of n unary constructed terms. The indices can be encoded by the position within the constructed term: for instance, the ith subterm represents the source end of an edge labeled $(_i$. Our representation of $)_i$ edges is modified as well, so that, e.g., the constraint $\mathcal{U} \subseteq o^{\sim i}(\mathcal{V})$ represents an edge $)_i \langle u, v \rangle$. Figure 5.7(b) shows the effect of the clustering optimization on the same example graph. The net effect of this optimization is the same as with summary edges: the edges $S \langle x_i, z_i \rangle$ are discovered with the addition of

at most $n + k - 1$ edges instead of nk edges. With some work, this reduction can be extended to handle PN-reachability and global nodes, though we omit the details here.

5.4.3.2 Cycle Elimination

One advantage of our reduction is that it preserves the structure of the input graph in a way that the Melski-Reps reduction does not. The constraint graphs produced by our reduction are isomorphic (modulo the representation of edges) to the original CFL graph. In fact, the connection is so close that cycle elimination, one of the key optimizations used in set constraint algorithms, is revealed to have an interpretation in the Dyck-CFL reachability problem.

Cycle elimination exploits the fact that variables involved in cyclic constraints (constraints of the form $\mathcal{X}_1 \subseteq \mathcal{X}_2 \subseteq \mathcal{X}_3 \ldots \subseteq \mathcal{X}_n \subseteq \mathcal{X}_1$) are equal in all solutions, and thus can be collapsed into a single variable. By reversing our reduction, we see that cyclic constraints correspond to cycles of the form:

$$x_1 \xrightarrow{s} x_2 \xrightarrow{s} \cdots \xrightarrow{s} x_n \xrightarrow{s} x_1$$

in the Dyck-CFL graph. The corresponding set variables are equivalent in all solutions; the CFL notion is that any node reaching one of the x_i nodes reaches every x_i node because we can always concatenate additional s terminals onto any word in the language and still derive a valid S-path. Hence, all the x_i's in the CFL graph can be collapsed and treated as a single node.

Note that the Melski-Reps reduction does not preserve such cycles: their reduction applied to CFL graph shown above produces constraints of the form $s(\mathcal{X}_i) \subseteq \mathcal{X}_{i-1}$ which do not expose a cyclic constraint for cycle elimination to simplify.

We validate these observations experimentally in Section 5.5 by showing the effect

of partial online cycle elimination [FFSA98] on our implementation.

5.5 Experimental Results

In this section we compare an implementation of the reduction in Section 5.3 to a hand-written Dyck-CFL reachability implementation by Robert Johnson, which is based on an algorithm described in [HRS95]. Johnson's implementation is customized to the polymorphic qualifier inference problem, and contains optimizations (including summary edges) that exploit the particular structure of the graphs that arise in this application.

We implemented the extensions for PN-reachability and global nodes as described in Section 5.4.1. In addition, all of the optimizations described in Section 5.4.3 are part of our implementation: we support clustering, and the Banshee toolkit uses inductive form and enables cycle elimination by default.

We used the C benchmark programs shown in Table 5.3 for our experiments. For each benchmark, the table lists the number of lines of code in the original source, the number of pre-processed lines of code, the number of distinct nodes in the CFL graph, the number of edges in the CFL graph, and the number of distinct indices (recall that this number corresponds to the number of function call sites)[3].

We ran CQual to generate a Dyck-CFL graph, and computed the closure of that graph using Johnson's Dyck-CFL reachability implementation and our own. We also ran our implementation with the clustering optimization enabled. Table 5.3 shows the analysis times (in seconds) for each experiment. The analysis times also include the time for CQual to parse the code and build the initial graph. Column 7 shows the time required for our implementation without cycle elimination. Column 8 shows

[3]These experiments were performed on a dual-processor 550 MHz Pentium III with 2GB of memory running RedHat 9.0, though only one processor was actually used.

Benchmark	LOC	Preproc	Nodes	Edges	Indices
identd-1.0.0	385	1224	3281	1440	74
mingetty-0.9.4	441	1599	5421	1469	111
bftpd-1.0.11	964	6032	9088	37558	380
woman-3.0a	2282	8611	10638	69171	450
patch-2.5	7561	11862	20587	76121	899
m4-1.4	13881	18830	30460	268313	1187
muh-2.05d	4963	19083	19483	141550	684
diffutils-2.7	15135	23237	33462	281736	1191
uucp-1.04	32673	69238	91575	2007054	4725
mars_nwe-0.99	25789	72954	115876	1432531	4312
imapd-4.7c	31655	78049	388794	2402204	7714
ipopd-4.7c	29762	78056	378085	2480718	7037
sendmail-8.8.7	44004	93383	126842	4173247	6076
proftpd-1.20pre10	23733	99604	184195	4048202	7206
backup-cffixed	39572	125350	139189	3657071	6339
apache-1.3.12	51057	135702	152937	6509818	5627
cfengine-1.5.4	39909	141863	146274	7008837	6339

Table 5.2: Benchmark sizes for CFL reachability experiments

times for the same implementation with partial online cycle elimination enabled. Column 9 shows times with cycle elimination and the clustering optimization. Finally, column 10 shows the times for Johnson's implementation, which acts as the gold standard. Figure 5.8 plots the analysis times for our implementation, normalized to Johnson's implementation. The results show that our implementation exhibits the same scaling behavior as Johnson's. In all the benchmarks except one, the analysis time for our implementation remains within a factor of two of Johnson's implementation. Because parse time is a constant for all the implementations, factoring it out does have an affect on these ratios; however, we found that parse time accounts for a small fraction of the total analysis time. The clustering optimization improved a few benchmarks significantly, but did not seem to improve scalability overall.

Figure 5.9 compares the performance of our implementation (no clustering) with

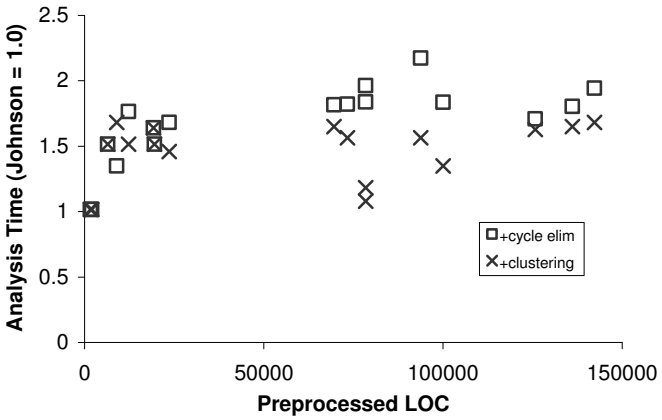


Figure 5.8: Results for the *tainted*/*untainted* experiment

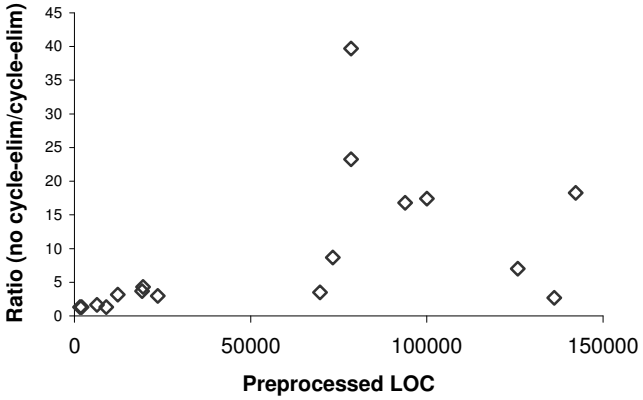


Figure 5.9: Speedup due to cycle elimination

Benchmark	BANSHEE(s)	+ cycle elim(s)	+ clusters(s)	johnson(s)
identd-1.0.0	.25	.25	.25	.25
mingetty-0.9.4	.25	.25	.25	.25
bftpd-1.0.11	1	.75	.75	.5
woman-3.0a	1	1	1.25	.75
patch-2.5	5	1.75	1.5	1
m4-1.4	11	3	3	2
muh-2.05d	9	2	2	1.5
diffutils-2.7	10	4	3	2
uucp-1.04	43	14	12	8
mars_nwe-0.99	117	14	12	8
imapd-4.7c	1782	45	25	23
ipopd-4.7c	877	38	25	21
sendmail-8.8.7	454	28	20	13
proftpd-1.20pre10	561	33	24	18
backup-cffixed	139	21	20	12
apache-1.3.12	75	32	29	18
cfengine-1.5.4	597	33	29	17

Table 5.3: Benchmark data for CFL reachability experiments

and without cycle elimination. We found that our benchmarks ran up to 40 times faster with cycle elimination enabled. These results suggest that online cycle elimination could be incorporated into the "standard" algorithms for Dyck-CFL reachability, an idea that has been implicitly suggested before [HT01b].

Chapter 6

Regular Annotations

As suggested in the previous chapter, many program analyses are expressible as reachability problems on labeled graphs with requirements that certain labels match: a constructor must be matched with a corresponding destructor, a function call must be matched with a function return, and so on. Dynamic transitive closure of a graph [Pal95], context-free reachability [RHS95], and the cubic-time fragment of set constraints [Hei92; AFFS98] are all formalisms that describe such analyses. These three approaches are closely related.

There are, however, more complex analysis problems in which multiple reachability properties must be satisfied simultaneously. For example, one can easily define problems that require matching of both function calls/returns and data type constructors/destructors. Unfortunately, any analysis problem requiring satisfying two or more context-free properties simultaneously is undecidable [Rep00]. A general class of reachability properties that remains decidable is the intersection of a context-free language with any number of regular languages. A number of natural analysis problems fall into this class [AM04; CW02; JMT99; HRS95].

In this chapter we show how to extend set constraints to express program analyses

involving the intersection of one context-free and any number of regular reachability properties. Existing implementations of analyses that combine context-free and regular reachability are hand optimized and tuned to a particular analysis problem. Our constraint resolution algorithm allows these analyses to be written at a higher level while also providing an implementation that is more efficient than those written by hand. In short, we enlarge the class of program analyses that can be solved efficiently with a single constraint resolution algorithm. In addition, our method enables us to resolve an open problem: we give a practical method to combine (predicative) parametric polymorphic recursion with non-structural subtyping in a label flow analysis.

Our approach builds on an idea first introduced in [RMR01], in which terms and constraints can be annotated with a word from some language. We introduce *regularly annotated set constraints*, in which each constructor of a term and each constraint can be annotated with a word from a regular language. This chapter makes the following contributions:

- We introduce regularly annotated set constraints and give a formal semantics. Previous work on annotated constraints has not addressed semantics, probably because the applications use only finite languages [MR05]. Because our annotations can be drawn from infinite regular languages, annotations are not bounded in size and understanding even the termination of a constraint solver requires formalization. We also find that a regular language is a more natural specification mechanism for annotations than the *concat* and *match* operators used in [MR05], and the regular language formulation is amenable to automatic generation of the constraint resolution rules.

- We discuss several algorithmic strategies for solving regularly annotated set constraints. We show that, unlike in the unannotated case, solving the constraints either forward or backward has an asymptotic advantage over a bidirectional

strategy[1]. On the other hand the bidirectional strategy has the advantage of allowing separate analysis and separate compilation. If the finite state machine for the annotations is small, the bidirectional strategy remains feasible.

- We show how to apply annotated inclusion constraints to solve pushdown model checking problems and interprocedural bit-vector dataflow problems that operate on the program's control flow graph. We also show how to combine the result from the previous chapter with annotations to express a type-based flow analysis that supports polymorphic recursion and non-structural subtyping in a label flow analysis.

To ease exposition, we focus on adding annotations to pure set constraints instead of mixed constraints. However, extending our results to the full mixed constraint framework is straightforward.

6.1 Semantics of Annotated Constraints

We extend set constraints as follows. Let $c, d, \ldots \in C$ be a collection of pure Set constructors and $M = (\Sigma, S, s_0, \delta, S_{accept})$ be a minimized finite state automaton. While regular languages and finite automata are equivalent, it is technically more convenient to work with automata. We occasionally refer to $L(M)$, the language accepted by M, whenever we need to appeal to a language characterization. Also, because regular languages are closed under intersection, it is sufficient to deal only with a single machine representing the intersection of all the regular reachability properties for a given application.

[1] We explain the idea of solver directionality in Section 6.4

6.1.1 Annotated Terms

The first step in our extension is to define the universe of terms. The intuition is that each term should be annotated with a word from $L(M)$; such a term encodes information for both the set constraint property (the term) and the regular reachability property (the word). This idea does not work, however, without two modifications:

- Word annotations must be included at every level of the term, not just at the root; every constructor must be annotated, and different constructors in the same term may have different annotations.

- Because individual constraints express only part of a global solution of all the constraints, it is too strong to require annotations be full words in $L(M)$. Instead, annotations may be partial words that may eventually be extended to full words in $L(M)$. The language of partial words that are admissible depends on the particular solution strategy: for example, a forwards (backwards) solver should admit prefixes (suffixes) of words in $L(M)$; a bidirectional solver should accept substrings of words in $L(M)$.

We will focus on bidirectional solving, not only because Banshee's algorithms are bidirectional, but also because bidirectional solving allows us to formulate a local set of resolution rules (in the style of the rules shown in Figure 2.1). In contrast to the unannotated case, the directionality of the solver for annotated constraint systems is significant; we discuss the tradeoffs shortly. For now, we continue the exposition assuming bidirectional solving.

The *annotated ground terms* over constructors $c, d, \ldots \in C$ and finite automaton M are

$$T^M = \{c^w(t_1, \ldots, t_{a(c)}) | t_i \in T^M \wedge c \in C \wedge w \in L(M)\}$$

Let M^{sub} be the minimal deterministic finite state automaton accepting substrings of $L(M)$ (the set of all substrings of a regular language is also regular). The domain we are interested in for bidirectional solving is $T^{M^{sub}}$.

To define the semantics of annotated constraints we will need an operation that appends a word to all levels of an annotated term:

$$c^w(t_1, \ldots, t_{a(c)}) \cdot w' = c^{ww'}(t_1 \cdot w', \ldots, t_{a(c)} \cdot w')$$

If Q is a set of terms then $Q \cdot w = \{t \cdot w | t \in Q\}$.

6.1.2 Annotated Set Constraints

A *regularly annotated set constraint* is an inclusion constraint $e_1 \subseteq^w e_2$, where e_1, e_2 are set expressions and $w \in L(M^{sub})$. We normally abbreviate $e_1 \subseteq^\epsilon e_2$ by dropping the annotation $e_1 \subseteq e_2$.

The next step is to define assignments ρ that map set expressions to sets of annotated ground terms. A wrinkle arises, however, because the set expressions are not themselves annotated; it turns out that we do not need to burden the analysis designer with annotating the set expressions. The insight is that it is possible to infer the needed annotations on set expressions during constraint resolution. We extend set expressions with *word set variables* attached to each constructor:

$$se ::= \mathcal{X} \mid c^\alpha(e_1, \ldots, e_{a(c)})$$

The word set variables $\alpha, \beta, \ldots$ range over subsets of $L(M^{sub})$. An assignment ρ now maps set variables to sets of annotated terms and word variables to sets of words.

$$\rho(c^\alpha(e_1, \ldots, e_{a(c)})) \quad = \quad \{c^w(t_1, \ldots, t_{a(c)}) | w \in \rho(\alpha) \wedge t_i \in \rho(e_i)\}$$

An assignment ρ is a solution of a system of annotated constraints $\{e_1 \subseteq^w e_2\}$ if

$$\rho(e_1) \cdot w \subseteq \rho(e_2)$$

for every constraint in the system.

Solutions may assign arbitrary sets to the word and term variables, provided they satisfy the constraints. We now show that a restricted family of solutions, the *regular solutions*, are sufficient to characterize all solutions.

We define the following congruence on words in $L(M)$:

$$w \equiv w' \Leftrightarrow \forall x, y \in \Sigma^*.\ xwy \in L(M) \text{ iff } xw'y \in L(M)$$

Theorem 6.1.1. $w \equiv w' \implies \forall s \in S.\ \delta(w, s) = \delta(w', s)$

Proof. For the sake of obtaining a contradiction, assume $w \equiv w'$ yet there is some $s \in S$ such that $\delta(w, s) = s_i$ but $\delta(w', s) = s_k$. Since the machine is minimal, s must be reachable from s_0, which implies the existence of a word x with $\delta(x, s_0) = s$. Note also that minimality implies the existence of a word y so that $\delta(y, s_i) \in S_{accept}$, but $\delta(y, s_k) \notin S_{accept}$, or vice versa. Without loss of generality, assume the first case. Then $xwy \in L(M)$ but $xw'y \notin L(M)$, which contradicts the assumption that $w \equiv w'$. □

This theorem suggests that every word equivalence class W defines a unique function from states to states: $f(s) = \delta(w, s)$ for $w \in W$. We call such a function a *representative function*. Every automaton M can be associated with a finite set of representative functions $F_M^{\equiv}$, computed inductively as follows:

$$F_M^0 = \{f_\sigma \mid \sigma \in \Sigma \text{ where } f_\sigma(s) = \delta(\sigma, s)\}$$
$$F_M^i = \{f \circ g \mid f, g \in F_M^{i-1}\} \cup F_M^{i-1}$$

Note that the representative function for the word ϵ in any machine is the identity function f_ϵ, which maps each state to itself. We will appeal to this function characterization of equivalence classes shortly.

We also extend $\equiv$ to an equivalence relation on annotated terms:

$$c^w(t_1, \ldots, t_{a(c)}) \equiv c^{w'}(t'_1, \ldots, t'_{a(c)}) \Leftrightarrow w \equiv w' \wedge \bigwedge_i t_i \equiv t'_i$$

Again, the constants (zero-ary constructors) form the base case of this definition. An assignment ρ is *regular* if

$$\begin{aligned} w \in \rho(\alpha) \wedge w \equiv w' &\implies w' \in \rho(\alpha) \\ t \in \rho(X) \wedge t \equiv t' &\implies t' \in \rho(X) \end{aligned}$$

Lemma 6.1.2. If $t \equiv t'$ and $t \cdot w \in T^{M^{sub}}$ then $t \cdot w \equiv t' \cdot w$.

Proof. The proof is by induction on the structure of t. Without loss of generality, assume $t = c^x(t_1, \ldots, t_{a(c)})$. Then because $t \equiv t'$, we know $t' = c^y(t'_1, \ldots, t'_{a(c)})$ where $x \equiv y$ and $t_i \equiv t'_i$.

$$\begin{aligned} c^x(t_1, \ldots, t_{a(c)}) \cdot w &= \\ c^{xw}(t_1 \cdot w, \ldots, t_{a(c)} \cdot w) &\equiv \\ c^{xw}(t'_1 \cdot w, \ldots, t'_{a(c)} \cdot w) &\equiv \\ c^{yw}(t'_1 \cdot w, \ldots, t'_{a(c)} \cdot w) &= \\ c^y(t'_1, \ldots, t'_{a(c)}) \cdot w &= \end{aligned}$$

The first step is just the definition of $\cdot$. For the second step, note that $t \cdot w \in T^{M^{sub}}$ implies that, for each i, $t_i \cdot w \in T^{M^{sub}}$ and therefore we can apply the induction hypothesis to conclude that $t_i \cdot w \equiv t'_i \cdot w$. For the third step, we observe that $x \equiv y$

implies that $xw \equiv yw$. The last step is another application of the definition of $\cdot$.

□

We say $\rho \leq \rho'$ if $\rho(a) \subseteq \rho'(a)$ for all word and set variables a. We say ρ' is the *regular completion* of ρ if ρ' is the smallest assignment such that $\rho' \geq \rho$ and ρ' is regular.

Theorem 6.1.3. If ρ is a solution of a system of annotated constraints, then its regular completion ρ' is also a solution.

Proof. Consider any constraint $e_1 \subseteq^w e_2$ and term $t \in \rho'(e_1)$; the goal is to show $t \cdot w \in \rho'(e_2)$.

$$\begin{aligned} t \in \rho'(e_1) &\Rightarrow \\ t' \in \rho(e_1) &\Rightarrow \quad \text{for some } t' \equiv t \\ t' \cdot w \in \rho(e_2) &\Rightarrow \\ t' \cdot w \in \rho'(e_2) &\Rightarrow \\ t \cdot w \in \rho'(e_2) & \end{aligned}$$

We briefly explain each step. The first implication follows from the fact that ρ' is the regular completion of ρ: there must be at least one term t' in $\rho(e_1)$ such that $t' \equiv t$. The second implication follows from the fact that ρ is a solution of the constraints, and the third implication follows because $\rho' \geq \rho$. The last step follows from Lemma 6.1.2, using the fact that $t' \equiv t$ from the first step and the fact that $t' \cdot w \in \rho(e_2)$ implies $t' \cdot w \in T^{M^{sub}}$ (because ρ is a solution, which by definition ranges over subsets of $T^{M^{sub}}$). □

Because for any solution of a system of annotated constraints the regular comple-

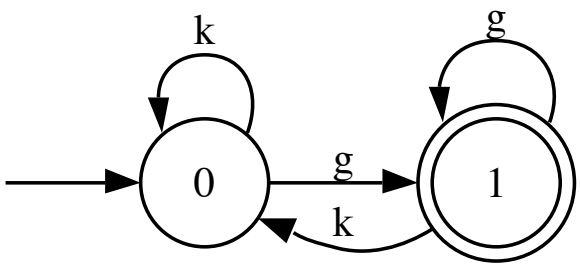


Figure 6.1: Finite state automaton M_{1bit} for the single fact bit-vector language

tion is also a solution, it suffices to compute only the regular solutions. Furthermore, in the regular solutions, each word set variable represents a set of full equivalence classes, and by Theorem 6.1.1 each equivalence class corresponds to a unique representative function. Thus, we can replace word set variables with representative function variables. In addition we can replace each annotation with the representative function for the annotation's equivalence class. This shift of perspective has the important advantage that we can now deal with finite sets of representative functions instead of potentially infinite sets of words. A mapping $\rho(\alpha) = \{f_1, f_2, \ldots\}$ corresponds to the regular solution $\rho(\alpha) = \{w | f(s) = \delta(s, w) \in \{f_1, f_2, \ldots\}\}$.

From here on we will refer to sets of representative functions, not sets of words, in constructor annotations; we use the term *representative function variables* instead of word variables for clarity. Our algorithm infers the representative functions needed as annotations automatically, which is why we do not represent them in the surface syntax.

We illustrate annotated constraints with a simple example. Assume that the input finite state machine M_{1bit} is the automaton shown in Figure 6.1.

Example 6.1.4. *Consider the following constraint system:*

$$\begin{aligned} c^{\alpha} &\subseteq^{g} \mathcal{W} \\ o^{\beta}(\mathcal{W}) &\subseteq^{g} \mathcal{X} \\ \mathcal{X} &\subseteq^{\epsilon} o^{\gamma}(\mathcal{Y}) \\ o^{\gamma}(\mathcal{Y}) &\subseteq^{\epsilon} \mathcal{Z} \end{aligned}$$

We have annotated each constructor with a function variable. Map the function variables as follows, where f_g is a function mapping state 0 to 1 and state 1 to itself (we call this function f_g because it is the representative function for the equivalence class containing the word g): $\alpha, \beta = \{f_\epsilon\}$, $\gamma = \{f_g\}$. Set variables are mapped as follows: $\mathcal{Y}, \mathcal{W} = \{c^{f_g}\}, \mathcal{X} = \{o^{f_g}(c^{f_g})\}$, and $\mathcal{Z} = \{o^{f_g}(c^{f_g})\}$. It is easy to check that this assignment is a solution of the constraints.

6.2 Solving Annotated Constraints

Our constraint solving algorithm takes a standard, two-phase approach:

- The first phase nondeterministically applies a set of resolution rules to the constraints until no more rules apply. The rules preserve all regular solutions of the constraints. If no manifest contradiction is discovered, the final constraint system is in *solved form*, which is guaranteed to have at least one solution.

- The second phase tests entailment queries on the solved form system: Do the constraints imply, for example, that $t \in X$ for some annotated term t and set variable X?

6.2.1 Resolution Rules

The first two resolution rules deal with constraints between constructor expressions:

$$c^{\alpha}(e_1, \ldots, e_{a(c)}) \subseteq^{f} c^{\beta}(e'_1, \ldots, e'_{a(c)}) \;\Rightarrow\; \bigwedge_i e_i \subseteq^{f} e'_i \wedge f \circ \alpha \subseteq \beta$$

$$c^{\alpha}(\ldots) \subseteq^{f} d^{\beta}(\ldots) \;\Rightarrow\; \text{no solution}$$

The first rule propagates inclusions between constructed terms to inclusions on the components. Recall that we have replaced all annotations with representative functions at this point. The other part of the first rule produces *representative function constraints* between the function variables annotating constructor expressions: the possible function annotations β on the right-hand side constructor expression are constrained to contain at least $f \circ \alpha$, the composition of functions in α with f (we define $f \circ G$, where G is a set of functions, to be $\{f \circ g \mid g \in G\}$). Because functions are just constants (zero-ary constructors) and the set $F_M^{\equiv}$ of representative functions is known and fixed for a given application, these function constraints are themselves simple examples of set constraints.

The second resolution rule simply recognizes manifestly inconsistent constraints.

The only other resolution rule is transitive closure. Transitive closure propagates annotations by concatenating them together. To ensure termination, we must bound the number of possible annotations between two variables. Here we make essential use of the function representation of annotations. Because we are only concerned with computing the regular solutions, constraint solving is not concerned with the exact word in the annotated constraint but only with the state of the automaton reached with the annotation word as the input. The transitive closure rule that reflects this observation is:

$$e_1 \subseteq^{f} \mathcal{X} \subseteq^{g} e_2 \Rightarrow e_1 \subseteq^{g \circ f} e_2$$

Notice that because M is minimized, no work is done propagating annotations that are necessarily non-accepting. This obviates the need for a *match* operation as in [MR05].

Returning to Example 6.1.4, the solved form of this system is:

$$\begin{array}{rcl} c^{\alpha} & \subseteq^{f_g} & \mathcal{W} \\ o^{\beta}(\mathcal{W}) & \subseteq^{f_g} & \mathcal{X} \\ \mathcal{X} & \subseteq & o^{\gamma}(\mathcal{Y}) \\ o^{\gamma}(\mathcal{Y}) & \subseteq & \mathcal{Z} \\ o^{\beta}(\mathcal{W}) & \subseteq^{f_g} & o^{\gamma}(\mathcal{Y}) \\ \mathcal{W} & \subseteq^{f_g} & \mathcal{Y} \\ c^{\alpha} & \subseteq^{f_g} & \mathcal{Y} \\ \\ f_g \circ \beta & \subseteq & \gamma \end{array}$$

Notice that the transitive constraints $c^{\alpha} \subseteq^{f_g} \mathcal{W} \subseteq^{f_g} \mathcal{Y}$ result in the constraint $c^{\alpha} \subseteq^{f_g} \mathcal{Y}$ because $f_g \circ f_g = f_g$ for the machine in Figure 6.1.

Lemma 6.2.1. Constraint resolution applying the transitive closure and constructor rules terminates.

Proof. (Sketch) The interesting case is the transitive closure rule. The number of possible constraints depends on the maximum number of distinct functions and the number of set expressions. As the resolution rules do not create any new set expressions and the number of distinct functions is bounded, the total number of possible constraints is also bounded. □

6.2.2 Queries

In this section we outline queries on solved systems. The simplest form of query we are interested in is, roughly speaking, whether a particular term t with an annotation in $L(M)$ is always in a particular set variable $\mathcal{X}$ in every solution. Intuitively, this question models whether a particular abstract value t can flow to a program point corresponding to the set variable $\mathcal{X}$ along a path annotated with a word in $L(M)$.

More precisely, we say that a system of constraints $\mathcal{C}_1$ entails a system of constraints $\mathcal{C}_2$, written $\mathcal{C}_1 \models \mathcal{C}_2$, if every solution of $\mathcal{C}_1$ is a solution of $\mathcal{C}_2$. Let $\mathcal{C}$ be the solved system of constraints. The formalization of the simple query is:

$$\mathcal{C} \wedge f_\epsilon \subseteq \alpha \wedge f_\epsilon \subseteq \beta \ldots \models \bigvee_{f \in F_{accept}} t \subseteq^f \mathcal{X}$$

where $\alpha, \beta, \ldots$ are the function variables appearing t, and $F_{accept} = \{f \mid f \in F_M^{\equiv} \wedge f(s_0) \in S_{accept}\}$. Intuitively, F_{accept} is the subset of representative functions $F_M^{\equiv}$ that lead to an accept state from the initial state; these are the functions that represent full words in $L(M)$. We can now explain a number of important aspects of querying annotated set constraints:

- If all of our constructors are monotonic, constraint systems have least solutions and to check the entailment it suffices to check that $\bigvee_{f \in F_{accept}} t \subseteq^f \mathcal{X}$ holds in the least solution of $\mathcal{C} \wedge f_\epsilon \subseteq \alpha \wedge f_\epsilon \subseteq \beta \ldots$. If anti-monotonic (contravariant) constructors are included, the constraints can be solved and the same entailments can be checked, but there may not be least solutions of the global system of constraints. Without the function constraints $f_\epsilon \subseteq \alpha \wedge \ldots$, the least solution of the constraints would assign the empty set to every function variable, as the constraints generated by the constructor rule (recall Section 6.2.1) do not require any function variables to be non-empty. Thus, it is important in our

approach that the constraint resolution phase preserve all solutions of the constraints; it is only when we ask a query and constrain some function variables to include the ϵ equivalence class that there are non-trivial least solutions.

- Set constraint solvers differ in how much work they assign to the solving phase and the query phase. We have described an eager solver that does essentially all the work in the resolution rules, as in [Hei92]; queries in this case are particularly easy to solve. For example, for a constant c^α, the entailment

$$\mathcal{C} \wedge f_\epsilon \subseteq \alpha \models c^\alpha \subseteq^f \mathcal{X}$$

holds if and only if the constraint $c^\alpha \subseteq^f \mathcal{X}$ is present in the solved form system $\mathcal{C}$. Because our actual implementation uses inductive form (recall Section 2.4); it does not compute some transitive constraints. More specifically, the transitive lower bound computation adds the final transitive constraints and therefore must be modified to compute the final annotations. If all constructors are monotonic, for example, the least solution is given by:

$$LS(\mathcal{Y}) = \{c^f(\ldots) | c^f(\ldots) \in pred(\mathcal{Y})\} \cup \bigcup_{\mathcal{X}^g \in pred(Y)} g \circ LS(\mathcal{X})$$

- Demand driven solvers essentially move all of the work of resolution to queries [HT01b]. As another optimization, our implementation solves function constraints on demand. Our solver actually does not generate function constraints or annotate constructors at all during resolution and this decision has important performance advantages (see Section 6.7). For our queries, the representative function constraints needed to answer a query can be reconstructed as part of the entailment computation itself.

- Our applications do need queries beyond asking whether a single term is in a set variable. The general form of a query is to ask whether a set of terms (given by a set expression) intersected with a variable is non-empty, given that the that constructors must be annotated in certain states. We present only the simpler case formally because it requires no additional notation, and the general case introduces no new ideas.

Returning again to Example 6.1.4, let $\mathcal{C}_1$ be the solved form of the constraints. The query

$$\mathcal{C}_1 \wedge f_\epsilon \subseteq \alpha \wedge f_\epsilon \subseteq \beta \models o^\beta(c^\alpha) \subseteq^{f_g} \mathcal{Z}$$

is true. The least solution of $\mathcal{C}_1 \wedge f_\epsilon \subseteq \alpha \wedge f_\epsilon \subseteq \beta$ is the assignment given in Example 6.1.4.

We can now explain in more detail why we solve constraints over $T^{M^{sub}}$ instead of T^M. The transitive closure rule, in particular, cannot simply reject concatenations of words that are not in $L(M)$, as such annotations may later combine with other annotated constraints through other uses of the transitive closure rule to form a word in $L(M)$. By solving in the larger domain $T^{M^{sub}}$ we preserve termination and also preserve all entailment queries.

6.2.3 An Example: Bit-Vector Annotations

As an example we show how to express bit-vector problems as a regular annotation language. This annotation language could be used to implement bit-vector based interprocedural dataflow analysis [HRS95]. For an analysis that tracks n facts, we pick an alphabet Σ partitioned into two sets $G = \{g_1, \ldots, g_n\}$ and $K = \{k_1, \ldots, k_n\}$ (`gens` and `kills`, respectively). The idea is that a `gen` cancels an adjacent matching kill `kill`, as in the word $g_i k_i$, and that `gen`s and `kill`s are idempotent. Figure 6.1 shows the finite state automaton M_{1bit} for a single dataflow fact. For this language,

$F_M^{\equiv} = \{f_\epsilon, f_g, f_k\}$, since:

$$f_g \circ f_g = f_g$$
$$f_k \circ f_g = f_\epsilon$$

and so on. Thus, we do not need to keep track of arbitrary sequences of `gens` and `kills` in constraint resolution; it suffices to track just three different possible annotations. An n-bit language can be derived from a product construction.

6.3 Complexity

We sketch a generic complexity argument for the constraint resolution algorithm described Section 6.2. A system of constraints containing no annotations can be solved in $O(n^3)$ time, where n is the number of variables in the constraint system. Intuitively, this is because each of the n variables can have up to n lower bounds and n upper bounds; thus every variable in the transitive closure causes at most $O(n^2)$ work and there are $O(n)$ variables.

For an annotated system of constraints, we must derive a new bound on the number of lower and upper bounds. Consider a particular lower bound in a set constraint system $e \subseteq \mathcal{X}$ (the argument for upper bounds is the same). In an annotated constraint system there may be many lower bounds between e and $\mathcal{X}$, one for each distinct equivalence class that can annotate the constraints, so the problem is to bound the number of distinct representative functions $e \subseteq^f \mathcal{X}$. Clearly, then, $|F_M^{\equiv}|$ gives the number of possible lower bounds on any variable. In the n-bit language, for instance, this approach automatically exploits order independence of distinct bits: If (shifting back to word annotations) a constraint $X \subseteq^{g_1 g_2} Y$ is already present in the system, the constraint $X \subseteq^{g_2 g_1} Y$ is redundant (i.e., $g_1 g_2 \equiv g_2 g_1$) and need not be added. This redundancy check takes constant time.

Thus, each variable in an annotated system can have up to $n \cdot |F_M^{\equiv}|$ lower bounds

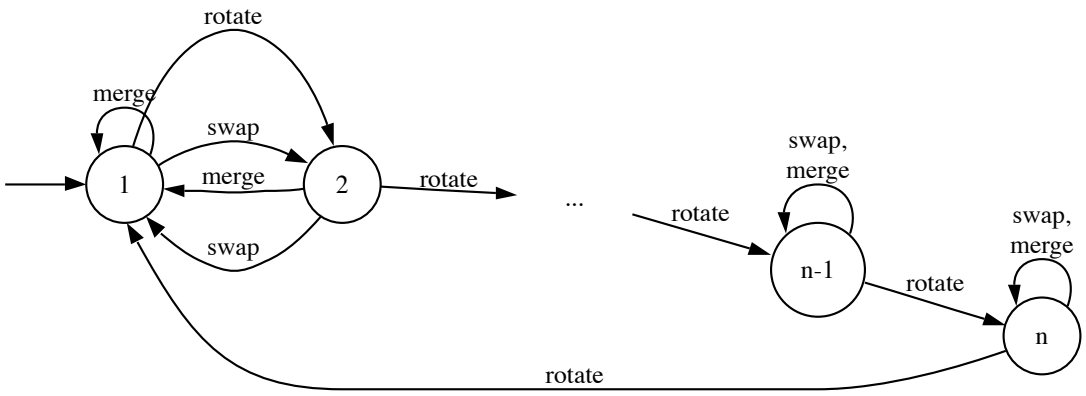


Figure 6.2: A machine with a small alphabet where $F_M^{\equiv}$ is prohibitively large

and $n \cdot |F_M^{\equiv}|$ upper bounds. With representative functions as the annotations on constraints, new annotations (the $\circ$ operation) can be computed in constant time using a table lookup, so for each of n variables in the constraint system the solver does at most $O(n^2 \, |F_M^{\equiv}|^2)$ work. The total complexity is therefore $O(n^3 \, |F_M^{\equiv}|^2)$. Note that this is a generic argument that can usually be sharpened for the constraints generated by a particular application.

We might wonder how to relate the complexity in terms of $|F_M^{\equiv}|$ to the complexity in terms of $|S|$, the number of states in M. It turns out that one can adversarially build a machine such that $|F_M^{\equiv}|$ is superexponential in $|S|$, the number of states in M. The idea is to construct a machine such that $F_M^{\equiv}$ contains every possible function from domain S to range S. In fact, it is possible to exhibit such a machine using an alphabet of only three symbols, which we suggestively name *merge*, *swap*, and *rotate*. Figure 6.2 shows the machine.

The key to this machine is that every function from S to S can be constructed from a combination of the operations *merge*, *swap*, and *rotate*: these three functions are generators for the entire space of functions. Therefore, $F_M^{\equiv}$ must have n^n elements.

This is clearly a pathologically bad case, yet the possibility of having an implementation that is superexponential in the size of the automata specification is unsatisfying. In the next subsection, we discuss alternative solver strategies that trade off the ability to do separate analysis and compilation for improved asymptotic complexity.

6.4 Alternative Algorithmic Strategies

Up to now, we have assumed that constraint resolution operates by adding new constraints to the constraint system in any order until the system is in solved form. We have been referring to this as bidirectional solving, as constraints can be "extended" either forwards or backwards via the transitive closure rule. Two natural alternatives are forward solving and backward solving. As a concrete illustration of the differences between these approaches, consider the constraints $f(c) \subseteq \mathcal{X} \subseteq \mathcal{Y} \subseteq f(\mathcal{Z})$. A forwards solver only propagates transitive closure by pushing lower bound sources towards upper bound sinks: in this case pushing $f(c)$ forwards by adding the constraint $f(c) \subseteq \mathcal{Y}$ and discovering $f(c) \subseteq f(\mathcal{Z})$. A backwards solver only applies transitive closure by pushing upper bound sinks towards lower bound sources: in this case pushing $f(\mathcal{Z})$ backwards by adding the constraint $\mathcal{X} \subseteq f(\mathcal{Z})$ to discover the same fact $f(c) \subseteq f(\mathcal{Z})$. Bidirectional solvers can add both new constraints, in any order.

6.4.1 Tradeoffs

The bidirectional approach has two principal advantages over unidirectional solving. First, bidirectional solving enables separate compilation. Intuitively, this is because the closure rules do not make any assumptions about the surrounding environment. A related advantage is that it is possible to solve the constraints online. Unidirectional

solvers usually defer most processing until the entire constraint graph is built.

On the other hand, unidirectional solvers can naturally be made demand driven. For instance, solving forward from a constant can selectively answer the query "For what set of variables must this constant appear in every solution?" Conversely, solving backward from a variable can selectively answer the query "What is the least solution of this variable?" Bidirectional solvers typically operate by exhaustively computing the entire global solution of the constraints prior to the query phase.

6.4.2 Complexity Arguments

The choice of solver directionality also affects the complexity of solving annotated constraints. We proceed by re-deriving some of our results that assumed bidirectional solving. Assume for simplicity that M has a single accept state– this assumption is not strictly necessary but simplifies the presentation somewhat.

Instead of the domain $T^{M^{sub}}$, in the forward case we need only consider the domain $T^{M^{pre}}$, where words are prefixes of full words in M. In the backward case, we consider $T^{M^{suf}}$, where words are suffixes of full words in M.

Next, we can relax the equivalence relation $\equiv$. In the forwards case, we relax the congruence $\equiv$ to a right congruence $\equiv_r$:

$$w \equiv_r w' \Leftrightarrow \forall x \in \Sigma^*.\ wx \in L(M) \text{ iff } w'x \in L(M)$$

In the backwards case, we relax $\equiv$ to a left congruence $\equiv_l$:

$$w \equiv_l w' \Leftrightarrow \forall x \in \Sigma^*.\ xw \in L(M) \text{ iff } xw' \in L(M)$$

Here, $|F_M^{\equiv_r}| = |S|$ and similarly $|F_M^{\equiv_l}| = |S|$ (this is essentially the Myhill-Nerode theorem). In the forwards case, $F_M^{\equiv_r}$ only distinguishes functions that map s_0 to one

of the different $|S|$ states in M, because of the following analog of Theorem 6.1.1:

$$w \equiv w' \implies \delta(w, s_0) = \delta(w', s_0)$$

In the backwards case, $F_M^{\equiv_l}$ only distinguishes functions that map different states in M to a state in S_{accept} (remember, though, that we are assuming there is only one such state for simplicity):

$$w \equiv w' \implies \forall s \in S.\ \delta(w, s) = s_{accept} \text{ iff } \delta(w', s) = s_{accept}$$

One significant difference from the bidirectional case is that our initial annotations are representative functions drawn from $F_M^{\equiv}$, while the annotations we derive are drawn from the coarser $F_M^{\equiv_r}$ in the forwards case, or $F_M^{\equiv_l}$ in the backwards case. Notice that the forwards solver will always compute new annotations as $g \circ f$, where $f \in F_M^{\equiv_r}$ and $g \in F_M^{\equiv}$. Furthermore the resulting function $g \circ f$ will always be in $F_M^{\equiv_r}$. Similarly, in the backwards case, when deriving $g \circ f$, $g \in F_M^{\equiv_l}$ and $f \in F_M^{\equiv}$, with the result $g \circ f \in F_M^{\equiv_l}$. In the bidirectional case, it happened that both the initial and derived annotations were drawn from $F_M^{\equiv}$.

In fact, this is precisely the source of the asymptotic complexity difference between unidirectional and bidirectional solving. In the unidirectional case, we are able to use a coarser equivalence relation, resulting in a much tighter bound on the number of possible derived annotations between two set expressions.

In general, the complexity of annotated constraint solving is $O(n^3 i^2)$, where i is the number of possible derived annotations. Here, we have shown how to express that number in terms of the number of states in the machine M. In the unidirectional case, i is exactly $|S|$. In the bidirectional case, i may be as much as $|S|^{|S|}$. In practice, it seems doubtful that this worst case will be realized, as it relies on an extremely contrived transition function. Also, in many applications the size of the finite state

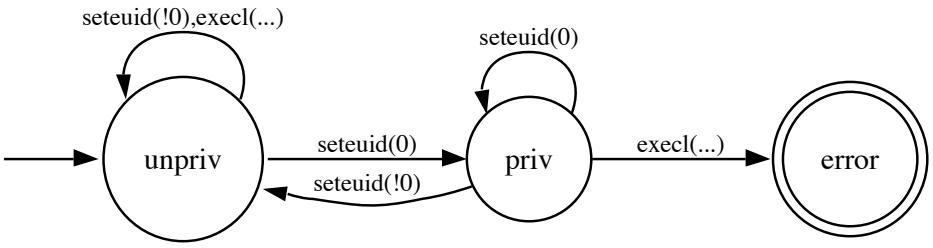


Figure 6.3: Automaton for process privilege

machine is a small constant. In exchange for the added complexity, the bidirectional approach retains support for separate compilation.

6.5 Application: Pushdown Model Checking

In this section, we use regularly annotated set constraints to solve pushdown model checking problems. We show how to verify the same class of temporal safety properties as MOPS, a model checking tool geared towards finding security bugs in C code [CW02].

Following the approach of [CW02], we model the program as a pushdown automata P. Transitions in the PDA are determined by the control flow graph, and the stack is used to record the return addresses of unreturned function calls. Temporal safety properties are modeled by a finite state machine M. Intuitively we want to compute the parallel composition of $L(M)$ and $L(P)$; the program is treated as a generator for this composed language.

We use the following property concerning Unix process privilege as a running example: a process should never execute an untrusted program in a privileged state—it should drop all permissions beforehand. Concretely, if a program calls `seteuid(0)`, granting root privilege, it should call `seteuid(!0)` before calling the `execl()` func-

tion. A program that violates this property may give an untrusted program full access to the system. Figure 6.3 shows a finite state machine that characterizes this property, and the following is a C program that violates the property:

```
seteuid(0);
...
execl(``/bin/sh'', ``sh'', NULL);
```

This program gives the user a shell with root privileges, which probably represents a security vulnerability.

6.5.1 Modeling Programs with Constraints

We now show how to find violations of temporal safety properties using annotated constraints. Our approach essentially uses annotations to build on the reduction presented in the previous chapter. For each statement s in the control flow graph, we associate a set constraint variable $\mathcal{S}$. For each successor statement s_i of s (with constraint variable $\mathcal{S}_i$), we add a constraint to the graph. The annotations are those program statements that are relevant to the security property (i.e., the statements labeling transitions in Figure 6.3). The specific form of the constraint depends on s; there are three cases to consider:

1. If s is not relevant to the security property, and is not a function call, add the constraint $\mathcal{S} \subseteq \mathcal{S}_i$.

2. If s is relevant to the security property (labels a state transition in the FSM for the security property) add the constraint $\mathcal{S} \subseteq^s \mathcal{S}_i$.

3. If s is a call to function f at call site i, add the constraints $o_i(\mathcal{S}) \subseteq \mathcal{F}_{entry}$ and $\mathcal{F}_{exit} \subseteq o^{\sim i}(\mathcal{S}_i)$, where $\mathcal{F}_{entry}$ ($\mathcal{F}_{exit}$) is the node representing the entry (exit) point of function f.

To model the program counter, we create a single 0-ary constructor pc and add the constraint $pc \subseteq \mathcal{S}_{main}$, where $\mathcal{S}_{main}$ is the constraint variable corresponding to the first statement (entry point) of the program's `main` function.

6.5.2 Checking for Security Violations

In order to check for violations of the property, we record each statement that could cause a transition to the error state. For each such statement, we query the least solution of the constraints by intersecting with an automaton for partially-matched reachability (PN reachability, see Section 5.4.1.1). The presence of an annotated ground term pc^{error} denotes a violation of the security property. The ground terms themselves serve as witness paths (in this setting, a possible runtime stack) that leads to the error.

6.5.3 An Example

Consider the following C program:

```
s1: seteuid(0); // acquire privilege
s2: if (...) {
s3:     seteuid(getuid()); // drop privilege
    }
        else {
s4:     ...
    }
s5: execl(``/bin/sh'', ``sh'', NULL);
s6: ...
```

This program violates the security property: the programmer has made the common error of forgetting to drop privileges on all paths to the `execl` call.

In the surface syntax, the constraints for this example are as follows:

$$pc \subseteq \mathcal{S}_1$$
$$\mathcal{S}_1 \subseteq^{seteuid(0)} \mathcal{S}_2$$
$$\mathcal{S}_2 \subseteq \mathcal{S}_3$$
$$\mathcal{S}_2 \subseteq \mathcal{S}_4$$
$$\mathcal{S}_3 \subseteq^{seteuid(!0)} \mathcal{S}_5$$
$$\mathcal{S}_4 \subseteq \mathcal{S}_5$$
$$\mathcal{S}_5 \subseteq^{execl(...)} \mathcal{S}_6$$

In order to translate the example to our internal syntax, we need to compute the set $F_M^{\equiv}$ and replace word annotations with representative functions. The functions relevant to our example are shown in Figure 6.4. We translate our constraints accordingly to yield:

$$pc^{f_\epsilon} \subseteq \mathcal{S}_1$$
$$\mathcal{S}_1 \subseteq^{f_0} \mathcal{S}_2$$
$$\mathcal{S}_2 \subseteq \mathcal{S}_3$$
$$\mathcal{S}_2 \subseteq \mathcal{S}_4$$
$$\mathcal{S}_3 \subseteq^{f_1} \mathcal{S}_5$$
$$\mathcal{S}_4 \subseteq \mathcal{S}_5$$
$$\mathcal{S}_5 \subseteq^{f_2} \mathcal{S}_6$$

Constraint resolution eventually discovers the following constraint path:

$$pc^{f_\epsilon} \subseteq \mathcal{S}_1 \subseteq^{f_0} \mathcal{S}_4 \subseteq^{f_2} \mathcal{S}_6$$

The constraints imply that $pc^{f_{error}}$ is in the least solution of $\mathcal{S}_6$, the constraint variable corresponding to the program point after the `execl` call, indicating the presence

$$
\begin{array}{rcll}
f_0 & : & \text{unpriv} & \rightarrow \text{priv} \\
 & & \text{priv} & \rightarrow \text{priv} \\
 & & \text{error} & \rightarrow \text{error} \\
 & & & \\
f_1 & : & \text{unpriv} & \rightarrow \text{unpriv} \\
 & & \text{priv} & \rightarrow \text{unpriv} \\
 & & \text{error} & \rightarrow \text{error} \\
 & & & \\
f_2 & : & \text{unpriv} & \rightarrow \text{unpriv} \\
 & & \text{priv} & \rightarrow \text{error} \\
 & & \text{error} & \rightarrow \text{error} \\
 & & & \\
f_{error} & : & \text{unpriv} & \rightarrow \text{error} \\
 & & \text{priv} & \rightarrow \text{error} \\
 & & \text{error} & \rightarrow \text{error}
\end{array}
$$

Figure 6.4: A few of the representative functions in $F_M^{\equiv}$ for the process privilege model

of a possible security vulnerability.

6.5.4 Parametric Annotations

Occasionally, pushdown model checking applications require a limited ability to correlate certain pieces of data. An excellent example of this is an analysis that tracks the opening and closing of files. The automaton for this analysis is shown in Figure 6.5. The annotations `x = open(...)` and `close(x)` are *parametric*: the `x` should be treated as a parameter that should be matched in the open and close calls. In Figure 6.6 we show a simple program that manipulates two file descriptors `fd1` and `fd2`. We would like an analysis that determines that `fd2` remains open at the end of the program, but not `fd1`.

To separate the states of each of the file descriptors in the program, the automaton in Figure 6.5 must be instantiated for each possible file descriptor that occurs in the input program. However, our approach precludes explicit instantiation because we

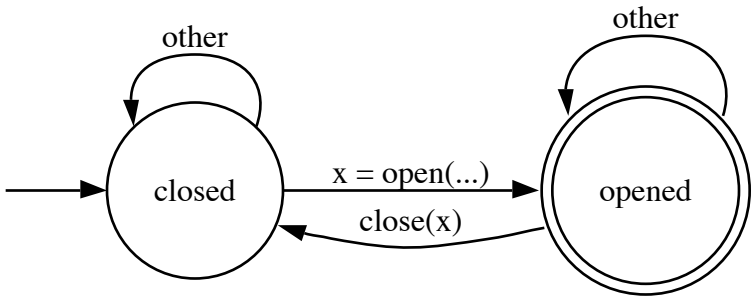


Figure 6.5: Automaton for tracking file state

```
s1: int fd1 = open(``file1'',O_RDONLY);
s2: int fd2 = open(``file2'',O_RDONLY);
s3: close(fd1);
s4: ...
```

Figure 6.6: Example C program that manipulates file descriptors

compile away the automaton statically, before the input program is available.

Instead of explicit instantiation, we perform instantiations on-the-fly by maintaining a special data structure called a *substitution environment*. Essentially, this data structure allows us to lazily construct the product automaton when there are multiple instantiations of a given parameter. We first explain a simpler version of the data structure that supports only one parameter per automaton. We then extend the data structure to handle multiple parameters.

A substitution environment ϕ maps instantiated parametric annotations (stored as a parameter name/label pair) to a representative function. The parameter name/label pairs form the *domain* of the substitution environment, while the representative functions are the *range*. Substitution environments have an additional component called a *residual*, which is itself just a representative function. The residual stores any non-parametric transitions that have occurred. Here is an example substitution environment ϕ:

$$[(x : \text{“}fd1\text{”}) \mapsto f;\ (x : \text{“}fd2\text{”}) \mapsto g \mid r_\phi]$$

This particular substitution environment contains two entries: one where parameter x has been instantiated to “fd1”, and another where x has been instantiated to “fd2”. The residual function is r_ϕ. The intuition is that this substitution tracks two copies of the automation in Figure 6.5. If any new instantiations are added via composition (e.g. $(x : \text{“}fd3\text{”})$), the residual r_ϕ should be incorporated into that instantiation's representative function. In a given substitution environment, the residual has already been incorporated into the existing instantiations The preceding explanation can be formalized by showing the composition operation $\circ$ on substitution environments:

$$(\phi_1 \circ \phi_2)(i) = \phi_1(i) \circ \phi_2(i)$$

Notice that these definitions gracefully degrade to the nonparametric case if we treat nonparametric annotations as empty substitution environments with a residual function given by the representative function for that annotation. We'll allude to this by dropping the brackets when writing an empty substitution environment: e.g. $[\ |\ r]$ will simply be written r.

Before discussing how to handle multiple parameters, we walk through the example from Figure 6.6.

6.5.4.1 Example

In the surface syntax, the program in Figure 6.6 results in the following system of constraints:

$$pc \subseteq \mathcal{S}_1$$
$$\mathcal{S}_1 \subseteq^{open(fd1)} \mathcal{S}_2$$
$$\mathcal{S}_2 \subseteq^{open(fd2)} \mathcal{S}_3$$
$$\mathcal{S}_3 \subseteq^{close(fd1)} \mathcal{S}_4$$

Internally, the constraint system is translated into the following system of constraints (Figure 6.7 gives the definitions of the relevant representative functions and substitution environments):

$$pc^{f_\epsilon} \subseteq \mathcal{S}_1$$
$$\mathcal{S}_1 \subseteq^{\phi_1} \mathcal{S}_2$$
$$\mathcal{S}_2 \subseteq^{\phi_2} \mathcal{S}_3$$
$$\mathcal{S}_3 \subseteq^{\phi_3} \mathcal{S}_4$$

$$
\begin{array}{rcll}
f_1 & : & \text{closed} & \rightarrow \text{opened} \\
 & & \text{opened} & \rightarrow \text{opened} \\
 & & & \\
f_2 & : & \text{closed} & \rightarrow \text{closed} \\
 & & \text{opened} & \rightarrow \text{closed} \\
 & & & \\
\phi_1 & = & [(x : \text{“}fd1\text{”}) \mapsto f_1 \mid f_\epsilon] & \\
\phi_2 & = & [(x : \text{“}fd2\text{”}) \mapsto f_1 \mid f_\epsilon] & \\
\phi_3 & = & [(x : \text{“}fd1\text{”}) \mapsto f_2 \mid f_\epsilon] &
\end{array}
$$

Figure 6.7: A few of the representative functions and substitution environments for the file state example

Eventually, thes constraints lead to the discovery of the constraint $pc \subseteq^{\phi_3 \circ \phi_2 \circ \phi_1} \mathcal{S}_4$, where:

$$\phi_3 \circ \phi_2 \circ \phi_1 = [(x : \text{“}fd1\text{”}) \mapsto f_2;\ (x : \text{“}fd2\text{”}) \mapsto f_1 | f_\epsilon]$$

6.5.4.2 Multiple Parametric Annotations

We previously assumed a single parametric annotation. In the case of multiple parameters, we need a slightly more complex substitution environment. Specifically, each entry in the environment could be a list of instantiations instead of a single instantiation. For example,

$$[(x : \text{“}i\text{”}, y : \text{“}j\text{”}) \mapsto f;\ (x : \text{“}k\text{”}) \mapsto g \mid r_\phi]$$

An entry i in a substitution environment is *compatible* with another entry j, written $i \preceq j$, if all the common parameter/label pairs agree and i has at least as many entries as j. By convention, every entry is compatible with the residual. When computing $\circ$ on substitution environments, compatible entries are *merged* by expanding the entries to contain the union of all the parameter label pairs. If we

define $\phi(i)$ to return the largest entry[2] in the domain of ϕ that i is compatible with, we can reuse our definition of $\circ$ on substitution environments and the desired effect is achieved.

6.6 Application: Flow Analysis

In this section we describe a novel flow analysis application that uses regular annotations to increase precision. Our motivation is to investigate practical algorithms for context-sensitive, field-sensitive flow analysis. A proof by Reps shows the general problem to be undecidable [Rep00]. As mentioned previously, the core issue is the arbitrary interleaving of two matching properties: function calls and returns, and type constructors and destructors. Viewing Reps' result in a type-based setting, we see that the problem involves precisely handling flow through polymorphic recursive functions and recursive types. Practical solutions to this problem require approximating one or the other matching property. In practice the approach taken almost universally is to approximate function matchings, which is typically done by analyzing sets of mutually recursive functions monomorphically.

Our view is that the essence of this approximation is reducing one matching to a regular language, while precisely modeling the other matching as a context-free language. For example, treating recursive functions monomorphically reduces the language of calls and returns to a regular language, while leaving the type-constructor matching language context-free. While this approach can be modeled using annotated set constraints (see Section 6.6.6), we first present a natural alternative that models function matchings as a context-free language, while reducing the type-constructor matching problem to a regular language. For this analysis, we apply the reduction

[2]This is unambiguous– if two entries have equal length, a larger entry in the domain of the substitution environment must also be compatible.

strategy described in Chapter 5 to model context-free language reachability of function matchings as a set constraint problem. We use regular annotations to model regular language reachability of type constructor/destructor matchings.

This analysis permits non-structural subtyping constraints. To our knowledge, ours is the first practical attempt to combine polymorphic recursion with non-structural subtyping constraints (we discuss a previous effort in Chapter 7).

6.6.1 Source Language

The analysis operates on the following source language:

$$\begin{array}{rcl} e & ::= & n \\ & | & x \\ & | & (e_1, e_2) \\ & | & e.i \ i = 1, 2 \\ & | & f^i \ e \\ fd & ::= & f(x : \tau) : \tau' = e \\ & | & fd; fd \end{array}$$

In the function definition $f(x : \tau) : \tau' = e$, f is bound within e. For simplicity, the source language does not include useful features such as conditionals, mutual recursion or higher-order functions. The analyses presented here can be extended to these features; we omit them only to simplify the presentation. We use τ to range over unlabeled types (pairs, integers, type variables, and first-order functions). Types are labeled with set variables $\mathcal{L}$. We use σ to range over labeled types, which are introduced by a *spread* operator:

$$\frac{\Gamma(x) = \sigma}{\Gamma \vdash x : \sigma} \text{ (Var)}$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash (e_1, e_2)^{\mathcal{L}} \vdash \sigma_1 \times^{\mathcal{L}} \sigma_2} \text{ (Pair)}$$

$$\frac{\Gamma \vdash e : \sigma_1 \times^{\mathcal{L}} \sigma_2}{\Gamma \vdash e.i : \sigma_i} \text{ (Proj } i = 1, 2)$$

$$\frac{\Gamma, x : \sigma, f : \sigma \to \sigma' \vdash e : \sigma'}{\Gamma \vdash f(x : \tau) : \tau' = e : \sigma \to \sigma'} \text{ (Def)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \vdash \sigma \leq \sigma'}{\vdash e : \sigma'} \text{ (Sub)}$$

$$\frac{\Gamma(f) = \sigma \quad \sigma \preceq^{i}_{+} \sigma'}{\Gamma \vdash f^{i} : \sigma'} \text{ (Inst)}$$

Figure 6.8: Type rules for polymorphic recursive system

$$\begin{array}{lcll} spread(\tau_1 \times \tau_2) & = & spread(\tau_1) \times^{\mathcal{L}} spread(\tau_2) & \mathcal{L} \text{ fresh} \\ spread(int) & = & int^{\mathcal{L}} & \mathcal{L} \text{ fresh} \\ spread(\alpha) & = & \alpha^{\mathcal{L}} & \mathcal{L} \text{ fresh} \end{array}$$

The function tl returns the label on the top-level constructor of a labeled type.

6.6.2 Type Rules and Constraint Generation

Figure 6.8 shows the type system for the polymorphic recursive analysis. The rules for variables, pairs, and pair projection are straightforward. The rule (Def) adds the types of the argument variable x and the function f to the environment, allowing recursive uses of f. Functions must be instantiated before use via the rule (Inst). The rule (Sub) permits non-structural subtyping steps, i.e. σ and σ' do not need to share the same type structure.

$$\frac{tl(\sigma) \subseteq tl(\sigma')}{\sigma \leq \sigma'} \text{ (Sub)}$$

$$\frac{tl(\sigma) \subseteq {o_1}^{\sim 1}(tl(\sigma'))}{\sigma \preceq^i_+ \sigma'} \text{ (Pos Inst)}$$

$$\frac{o_i(tl(\sigma')) \subseteq tl(\sigma)}{\sigma \preceq^i_- \sigma'} \text{ (Neg Inst)}$$

$$\frac{\sigma_1 \preceq^i_- \sigma_3 \quad \sigma_2 \preceq^i_+ \sigma_4}{\sigma_1 \to \sigma_2 \preceq^i_+ \sigma_3 \to \sigma_4} \text{ (Fun Inst)}$$

$$\frac{}{int^{\mathcal{L}}} \text{ (Int WL)}$$

$$\frac{}{\alpha^{\mathcal{L}}} \text{ (Var WL)}$$

$$\frac{\begin{array}{cc} tl(\sigma_1) \subseteq^{[^1_{\tau_1}} \mathcal{L} & tl(\sigma_1) \subseteq^{[^2_{\tau_2}} \mathcal{L} \\ \mathcal{L} \subseteq^{]^1_{\tau_1}} tl(\sigma_1) & \mathcal{L} \subseteq^{]^2_{\tau_2}} tl(\sigma_2) \end{array}}{\sigma_1 \times^{\mathcal{L}} \sigma_2} \text{ (Pair WL)}$$

Figure 6.9: Constraint generation for polymorphic recursive system

The constraint generation rules are shown in Figure 6.9. One key aspect of constraint generation is that we do not apply constraints downward through types—constraints extend only to the top-level constructors.[3] Constraints between substructures of types are discovered automatically as needed during constraint resolution.

We require annotated constraints representing type constructors and destructors be only between the labels representing the constructed term and its components. Rules (Pair WL), (Var WL) and (Int WL) define a *well-labeling* relation on labeled types. All labeled types in the program must be well-labeled.

6.6.2.1 Function Call Matching with Terms

We apply the reduction from Chapter 5 to model the matching of function calls and returns. The result in [RF01] shows that this is equivalent to polymorphic recursive treatment of functions.

6.6.2.2 Type Constructor Matching with Annotations

Annotations are used to model the matching language of type constructors and destructors. For example, in the expression $(x^{\mathcal{X}}, y^{\mathcal{Y}})^{\mathcal{P}}.1^{\mathcal{Z}}$, the constraint $\mathcal{X} \subseteq^{[^1_{int}} \mathcal{P}$ models the flow from the first component of the pair to the pair constructor, and the constraint $\mathcal{P} \subseteq^{]^1_{int}} \mathcal{Z}$ models the flow from the pair to the projected result. The two annotations $[^1_{int}$ and $]^1_{int}$ should "cancel" each other, reflecting the flow from $\mathcal{X}$ to $\mathcal{Z}$ via the un-annotated constraint $\mathcal{X} \subseteq \mathcal{Z}$. While this language of matchings appears context-free, in the absence of recursive types it is not possible for a symbol of the form $[^i_\tau$ to be followed by another of the same symbol without first encountering a corresponding $]^i_\tau$ symbol to cancel the first symbol. For this reason we need the extra τ component on annotations: to distinguish pair projection on different levels of the

[3] As noted earlier we could also treat function types as type constructors and extend our construction to handle higher-order function types; we treat function types specially here only for brevity.

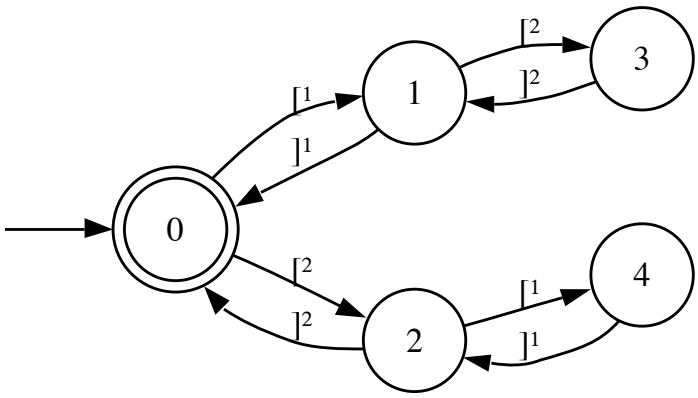


Figure 6.10: Finite state automaton for single level pairs

type. Thus, for a given input program, we can place a bound the longest string of annotations we need consider by the size of the largest type. In Figure 6.10 we show the finite state automaton for this annotation language when the program's largest type is $pair(int)$. In the presence of recursive types, flow must be approximated, for example by replacing annotated constraints on recursive types with empty annotations.

6.6.3 Answering Flow Queries

To ask whether a particular label (say $\mathcal{X}$) flows to another label (say $\mathcal{Y}$), the constraint $x \subseteq \mathcal{X}$, where x is a fresh constant, is added to the system. Then $\mathcal{X}$ flows to $\mathcal{Y}$ if x is in the least solution of $\mathcal{Y}$. This query yields answers for *matched* flow, and this approach can be extended to partially-matched reachability through functions (called PN reachability in Chapter 5).

```
pair (y:int) :  β = (1^A, y^Y)^P;
main () :  int = (pair^i 2^B).2^V
```

Figure 6.11: Non-structural subtyping example

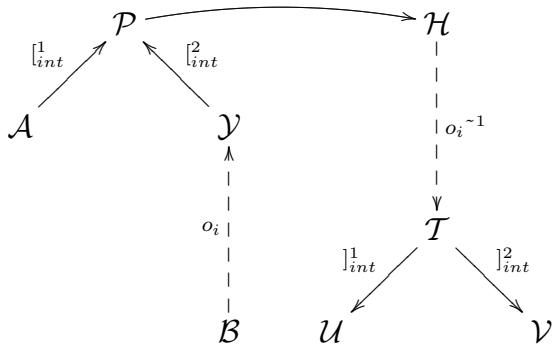


Figure 6.12: Constraint graph for the program in Figure 6.11 (only relevant edges are shown)

6.6.4 An Example

Consider the program in Figure 6.11 (taken from [FRD99]). Non-structural subtyping can assign `pair` the type $int^{\mathcal{Y}} \rightarrow \beta^{\mathcal{H}}$ along with the constraint $\beta = int^{\mathcal{A}} \times^{\mathcal{P}} int^{\mathcal{Z}}$. Figure 6.12 shows a slightly simplified constraint graph for this program. Flow from $\mathcal{B}$ to $\mathcal{V}$ is captured by the constraints:

$$\begin{array}{rcl} o_i(\mathcal{B}) & \subseteq & \mathcal{Y} \\ \mathcal{Y} & \subseteq^{\lfloor^2_{int}} & \mathcal{P} \\ \mathcal{P} & \subseteq & \mathcal{H} \\ \mathcal{H} & \subseteq & o_i{}^{\sim 1}(\mathcal{T}) \\ \mathcal{T} & \subseteq^{\rfloor^2_{int}} & \mathcal{V} \end{array}$$

which imply the relationship $\mathcal{B} \subseteq \mathcal{V}$.

6.6.5 Stack-Aware Aliasing

The analysis presented in this section can be used to implement context-sensitive, field-sensitive alias analysis. An interesting consequence of this formulation is that an additional dimension of sensitivity can be recovered during the alias query phase. A standard approach to computing aliasing information from a points-to analysis is to intersect sets of abstract locations– an empty intersection indicates that two expressions do not alias. In our setting, we can instead intersect the solutions of two variables and test for emptiness, giving *stack-aware* alias queries.

Consider the following C program:

```
void main() {
    int a,b;
    foo¹(&a,&b); // constructor o₁
    foo²(&b,&a); // constructor o₂
}
void foo(int *x, int *y) {
    // May x and y be aliased?
}
```

If the above represents the whole program, `x` and `y` clearly cannot be aliased within `foo`. If the points-to sets themselves are not considered context-sensitively, however, the points-to results contain $pt(\mathtt{x}) = pt(\mathtt{y}) = \{a, b\}$, and the analysis would report that `x` and `y` may alias.

In our setting, points-to sets are terms where unary constructors encode information about function calls. Our analysis would yield the following solution (annotations elided) representing points-to sets for the above program:

$$\begin{aligned} X &= \{o_1(a), o_2(b)\} \\ Y &= \{o_2(a), o_1(b)\} \end{aligned}$$

Intersecting the solutions for X and Y reveals that there are no common ground terms; hence the two variables are not aliased. The view put forth here is that the constraint solutions themselves are an appropriate data structure for representing context-sensitive points-to sets.

While the above example is somewhat contrived, stack-aware alias queries allay a real problem: in most alias analyses, the memory abstraction is based on syntactic occurrences of calls to allocation routines (e.g. `malloc` and `new`). Simple refactorings such as wrapping an allocation function (or allocating an object's fields within a constructor in an object-oriented language) can destroy the precision of the analysis. Stack-aware alias queries use the call stack to disambiguate object allocation sites, giving a form of object sensitivity.

6.6.6 A Dual Analysis

As mentioned earlier, a more widely used approach to combining context-sensitivity and field sensitivity is to approximate the language of function calls and returns by treating mutually recursive functions monomorphically. We note that this analysis is also expressible in our framework. The key change is to swap the roles of annotations and terms: now annotations $[^i$ and $]^i$ model call/return paths to a function call site i, and constructors $o_i(\ldots)$ and projection patterns $o_i{}^{\sim 1}(\ldots)$ model constructing/destructing the ith field from a tuple. We can also take advantage of n-ary constructors to cluster types, so that instead of using two constructors o_1 and o_2 to represent the first and second components of a pair, we use a binary constructor $pair$ to construct a pair, and projections $pair^{\sim 1}(\ldots)$ and $pair^{\sim 2}(\ldots)$ to deconstruct

a pair. This more natural representation can actually improve performance, as edge additions that would need to be discovered twice using unary constructors can be discovered a single time instead using a binary constructor (see Section 5.4.3.1). With this approach, the constraint system for the example program in Figure 6.11 is as follows:

$$\begin{aligned} \mathcal{B} &\subseteq^{\lfloor_i} \mathcal{Y} \\ pair(\mathcal{A}, \mathcal{Y}) &\subseteq \mathcal{H} \\ \mathcal{H} &\subseteq^{\rfloor_i} \mathcal{T} \\ \mathcal{T} &\subseteq pair^{\sim 2}(\mathcal{V}) \end{aligned}$$

which implies the desired constraint $\mathcal{B} \subseteq \mathcal{V}$.

6.7 Experimental Results

We have implementated regularly annotated set constraints in the Banshee toolkit by adding support for annotations to Banshee's existing implementation of the Set sort. Some of the technical details (such as handling projection merging [SFA00] and cycle elimination [FFSA98] in the presence of annotations) are similar to those addressed in [RMR01]; we omit them here.

Since Banshee already does specialization based on a statically-specified description of the term constructors used in an analysis, it is very natural to extend specialization to the input finite state automaton. We have created an annotation specification language whose syntax is loosely based on ML pattern matching syntax. For example, the process privilege automaton shown in Figure 6.3 is specified as follows in our language:

start state Unpriv :

```
  | seteuid_zero -> Priv;

state Priv :
  | seteuid_nonzero -> Unpriv
  | execl -> Error;

accept state Error;
```

This specification is compiled by computing the set $F_M^{\equiv}$ and creating a lookup table that implements the $\circ$ operation. This allows us to compute a new annotation for the transitive closure operation in constant time. At runtime, annotations are represented as substitution environments in order to support parametric annotations. Multiple parametric annotations are supported.

Our implementation omits representative function variables on set expressions completely during constraint solving and instead does all of the calculations involving these functions during queries (recall Section 6.2.2). By omitting these variables from the solver we can do aggressive hash-consing of terms, and the memory savings from hash consing is substantial. The time and space overhead needed to implement the operations in the transitive closure rule is minimal, since function composition can be reduced to table lookups by the specializer (recall Section 6.3).

To illustrate the viability of our approach, we have reproduced some experiments first done using MOPS. We chose to examine the applications of MOPS because pushdown model checking is superficially very different from the usual applications of Banshee. We chose a security property (Property 1 from [CDW04]) and checked several sensitive software packages for security violations using the approach outlined in Section 6.5. The property we checked is a complete model of the simple process privilege property described in Section 6.5. The complete model (shown in Figure 6.13) contains 11 states and 9 different alphabet symbols. It is the most complex property

```
state r0e0s0 :
        | setuid –> error
        | seteuid –> error
        | badcall –> violation;

start state r1e0s0 :
        | setuid0 –> r0e0s0
        | setuidgeteuid –> r0e0s0
        | setuidgetuid –> r1e1s1
        | setuid –> error
        | seteuidgetuid –> r1e1s0
        | seteuid –> error
        | badcall –> violation;

state r0e1s0 :
        | setuid0 –> r0e0s0
        | setuidgetuid –> r0e0s0
        | setuid –> error
        | seteuid0 –> r0e0s0
        | seteuidgetuid –> r0e0s0
        | seteuid –> error;

state r1e1s0 :
        | setuid0 –> r1e0s0
        | setuid –> error
        | seteuid0 –> r1e0s0
        | seteuid –> error;

state r1e0s1 :
        | setuid0 –> r0e0s0
        | setuidgeteuid –> r0e0s0
        | setuidgetuid –> r1e1s1
        | setuid –> error
        | seteuidgetuid –> r1e1s1
        | seteuid –> error
        | badcall –> violation;

state r0e1s1 :
        | setuid0 –> r0e0s1
        | setuidgetuid –> r0e0s1
        | setuid –> error
        | seteuid0 –> r0e0s1
        | seteuidgetuid –> r0e0s1;

state r0e0s1 :
        | setuid0 –> r0e0s0
        | setuidgetuid –> r0e0s0
        | setuidgeteuid –> r0e0s0
        | setuid –> error
        | seteuid –> error
        | badcall –> violation;

state r1e1s1 :
        | setuid –> error
        | seteuid –> error;

state error :
        | badcall –> violation;

accept state violation;
```

Figure 6.13: Specification for the full process privilege model

and largest automaton reported in [CDW04]. This property demonstrates that, in practice, the representative function sets do not exhibit worst case size— while the set $F_M^{\equiv}$ could contain millions of elements for an 11 state automaton, in this case, there are only 58 distinct representative functions.

We report in Table 6.1 the number of lines of code for each package, the number of executables for each package, and the time to check the property for the executables in

Benchmark	Size	Programs	Banshee (s)	MOPS (s)
VixieCron 3.0.1	4k	2	.52	.57
At 3.1.8	6k	2	.52	.62
Sendmail 8.12.8	222k	1	2.3	5.1
Apache 2.0.40	229k	1	.6	.7

Table 6.1: Benchmark data for process privilege experiment

the package for both Banshee and MOPS.[4] Each executable in a package is checked separately. Our analysis times show that our algorithm's scalability and performance is very good, and that the bidirectional solver is usable for realistic applications.

[4]The experiment was performed on a 2.0 GHz Intel Core Duo machine with 512 Gb of memory.

Chapter 7

Related Work

There are several frameworks whose expressive power is similar to that of the mixed constraint framework presented here. Datalog is a database query language based on logic programming [CGT89]. Datalog has recently received some attention as a specification language for static analyses. We note that the subset of pure set constraints implemented in BANSHEE is equivalent to chain datalog [Yan90] and also context-free language reachability [MR97]. There are also obvious connections to bottom-up logic [McA02].

Implementations of these frameworks have been applied to solve static analysis problems. The bddbddb system is a deductive database that uses a binary-decision diagram library as its back-end [WL04]. Using Datalog as a high level language, bddbddb hides low-level BDD operations from the analysis designer. Other toolkits that use BDD back-ends include CrocoPat [BNL03] and Jedd [LH04]. One disadvantage to these systems is that BDD-based backends cannot be treated as a "black box"; using them effectively requires subtle understanding of the internal BDD representation (in particular, variable orderings). BDD-based algorithms also have exponential worst-case performance. As we have shown, a class of reachability problems more general

than those handled by current BDD-based methods can be solved in polynomial time.

Parametric regular path queries are a declarative way of specifying graph queries as regular expression patterns [LRY+04]. Regular path queries are not as powerful as set constraints, though the use of parameters to correlate related data may be a useful addition to our framework.

Many researchers have formulated program analyses as CFL reachability problems. The author first became aware of this connection from work by Reps et al., who formulated interprocedural dataflow analysis as a Dyck-CFL graph, and introduced an algorithm to answer the all-pairs reachability problem for these graphs [RHS95]. Follow-up work introduced a demand-driven algorithm for solving this problem [HRS95], though we have not so far seen a fully incremental algorithm described. Many implementations of Dyck-CFL reachability are based on this algorithm. Rehof et al. show how to use CFL reachability as an implementation technique for polymorphic flow analysis [RF01]. CFL reachability is used as an alternative to repeated copying and simplification of systems of instantiation constraints. The first application of Rehof et al.'s work was to polymorphic points-to analysis [DLFR01]. Other applications of Dyck-CFL reachability include field-sensitive points-to analysis, shape analysis [Rep95], interprocedural slicing [HRB90], and debugging systems of unification constraints [CH03].

Pushdown systems [EHRS00] are transition systems where the configurations consist of a control state and a stack. As we have seen, the regularly annotated constraint framework (in conjunction with the Dyck-CFL reachability reduction from Chapter 5) can efficiently express pushdown systems. Our work also addresses the problem of separate analysis of pushdown systems, and also illustrates a connection between pushdown systems and type-based flow analysis.

Weighted pushdown systems (WPDS) label transitions with values from a domain of weights [RSJ03]. Weighted pushdown reachability computes the meet-over-

all-paths value for paths that meet certain properties. WPDS have been used to solve various interprocedural dataflow analysis problems—the weight domains are general enough to compute numerical properties (e.g., for constant propagation), which cannot be expressed using our annotations. On the other hand, WPDS focus on checking a single, but extended, context-free property, while annotated constraints naturally express a combination of a context-free and any number of regular reachability properties. The exact relationship between WPDS and regularly annotated constraints is not clear.

Other analyses that demand more expensive algorithms, e.g.; path sensitive analyses, cannot be expressed with the polynomial time algorithms we present here. Ramalingam and Reps have compiled a categorized bibliography on incremental computation [RR93]. We are not aware of previous work on incrementalizing set constraints, though work on incrementalizing transitive closure is abundant and addresses some of the same issues [DI00; Rod03]. The CLA (compile, link, analyze) [HT01b] approach to analyzing large codebases supports a form of file-granularity incrementality similar to traditional compilers: modified files can be re-compiled and linked to any unchanged object files. This approach has some advantages. For example, since CLA doesn't save any analysis results, object file formats are simpler, and there is no need to make the analysis persistent. However, CLA defers all of its analysis work until after the link phase, so the only savings would be the cost of parsing and producing the object files.

Regularly annotated set constraints are partly inspired by the annotated inclusion constraints presented in [RMR01; MR05]. We have shown how to incorporate infinite regular languages as annotations, and we believe that finite state automata are a more natural specification language for annotations than the *concat* and *match* operators used in prior work. All of the annotation languages used in [MR05] can be expressed in terms of finite state automata.

The combination of polymorphic recursion and non-structural subtyping that we consider in Section 6.6 was first considered in [FRD99]. The solution proposed in [FRD99] has the disadvantage that polymorphism on data types is achieved by copying constraints on data types. While there is no implementation of this algorithm, the general experience with constraint-copying implementations is that they are slow [FFA00]. For this reason we consider our approach, which relies on regular annotations rather than copying constraints for polymorphism, to be a more practical algorithm for this class of analyses.

Chapter 8

A Summary of Banshee

In this chapter we describe the BANSHEE implementation, which is publicly available from `http://banshee.sourceforge.net`. All file names in this chapter refer to files in the version 1.02 distribution.

8.1 Banshee Specification Language Syntax

Specification files consist of a single *specification*, drawn from the following grammar:

specification ::= `specification` *spcid* `:` *hdrid* `=` `spec` *dataspec* `end`

dataspec ::= `data` $exprid_1$ `:` $sort_1\langle sortopts\rangle$ $\langle$`=` $conspec_1\rangle_1 \cdots$
`and` $exprid_n$ `:` $sort_n\langle sortopts\rangle$ $\langle$`=` $conspec_n\rangle_n$
| $dataspec_1$ $dataspec_2$

conspec ::= *conid* $\langle$`of` *consig*$\rangle$
| *grpid*`<`*conid*`>` `of` *consig*
| $conspec_1$ `|` $conspec_2$

consig ::= $bconsig_0$ `*` $\cdots$ `*` $bconsig_n$

bconsig ::= *vnc* *exprid*

vnc ::= `+`|`-`|`=`

sortopts ::= `[`$option_1$`,` $\cdots$ `,`$option_n$`]`

sort ::= `term` | `setIF` | `row(`*exprid*`)`

By convention, BANSHEE specification files have extension `.bsp`. Expression identifiers (*exprid*), constructor identifiers (*conid*), group identifiers(*grpid*) specification identifiers (*spcid*), and header identifiers (*hdrid*) must only be used once in a `.bsp` file. Identifiers consist of an upper- or lowercase letter followed by a sequence of letters, digits, or underscores ('_').

8.2 Banshee Annotation Language Syntax

Annotations consist of a single *automaton*, drawn from the following grammar:

automaton ::= $state_1$; $\cdots$ $state_n$;

state ::= $stateopt_1 \cdots stateopt_n$ `state` *stateid* `:` *transitions*

| $stateopt_1 \cdots stateopt_n$ `state` *stateid*

stateopt ::= `start`

| `accept`

transitions ::= *transition*

| *transitions transition*

transition ::= `|` *symbol* `->` *stateid*

symbol ::= *symid*

| *symid* `<` *paramid* `>`

| _

By convention, BANSHEE annotation files have extension `.ban`.

State identifiers (*stateid*), symbol identifiers (*symid*), and parameter identifiers (*paramid*) must only be used once in a `.ban` file. Identifiers consist of an upper- or lowercase letter followed by a sequence of letters, digits, or underscores ('_').

8.3 iBanshee: The Banshee Interpreter

*i*BANSHEE is an interpreter that provides an easy way to explore BANSHEE's constraint language. *i*BANSHEE allows the designer to interact with BANSHEE by typing in constraints directly. *i*BANSHEE also acts as a lightweight interface to BANSHEE for clients written in languages other than C.

8.3.1 Expressions in iBanshee

We first show how to declare constructors and variables in *i*Banshee. Both constructors and variables must be declared before use. *i*Banshee constructors must begin with an upper or lower case letter, followed by a string of letters, numbers, or underscores. Declaring a constant is simple: [1]

```
[0] > c : setIF
constructor: c
```

This declares a constant c. The colon followed by `setIF` is a sort declaration in *i*Banshee.

Declaring an n-ary constructor is slightly more involved, because we need to specify the constructor's arity. We do so with a comma separated list of sort declarations:

```
[0] > f(+setIF,+setIF) : (setIF)
constructor: f
```

This declares a binary constructor f. The plus signs before the sort declarations in this example declare the variances of the constructor arguments.

Variables are declared just as constants. However, *i*Banshee syntactically distinguishes variables from constructors by forcing you to begin each variable name with a tick (`'`). For example:

```
[0] > 'x : setIF
var: 'x
```

declares a variable called `'x`.

[1] The `[0] >` is *i*Banshee's prompt. The line immediately following a line starting with a prompt is *i*Banshee's output.

Note that *i*Banshee only requires declaration of constructors and variables. Other terms need not be declared explicitly. In other words, once `f` and `'x` have been defined as in the preceeding examples, expressions such as `f('x,'x)` can be referenced without prior declaration. Next, we discuss the syntax for declaring constraints between expressions.

8.3.2 Constraints in iBanshee

Creating a system of constraints in *i*Banshee is straightforward. After defining a set of constructors and variables, a language of expressions over those constructors and variables is available for use in constraints. We'll continue with an *i*Banshee program that builds on the previous example:

```
[0] > f(+setIF,+setIF):setIF
constructor: f
[0] > c : setIF
constructor: c
[0] > g(+setIF) : setIF
constructor: g
[0] > 'x : setIF
var: 'x
[0] > 'y : setIF
var: 'y
[0] > f('x,g('x)) <= f('y,'y)
[1] > c <= 'x
```

Since Banshee solves constraints online, queries can be made at any point after a constraint has been added. We can do so using *i*Banshee's query commands.

*i*Banshee commands always begin with `!`. The first command is `!tlb`, which stands for *transitive lower bounds*. This command allows one to read off the least solution of the constraints:

```
[2] > !tlb 'x
{c}
[2] > !tlb 'y
{c, g('x)}
```

With a little investigation we see that this is exactly the least solution.

8.3.3 Backtracking

Backtracking allows the user to "roll back" the state of a constraint system at any point in time. In *i*Banshee, the number displayed at the prompt tells the current "version" of the constraint system. Backtracking allows the constraint system to be rolled back to any previous version, by specifying the version number:

Let's work with the previous example:

```
[0] > f(+setIF,=term):setIF
constructor: ref
[0] > 'x : setIF
[0] > 'y : term
[0] > 'z : setIF
[0] > c : term
[0] > f('x, c) <= f('z,'y)
[1] > !ecr 'y
c
```

Notice that the version number is incremented after the addition of the constraint `f('x,c) <= f('z,'y)`. That means that the constraint system's version prior to the addition of that constraint is `0`, and the version after the constraint addition is `1`. In *i*BANSHEE, backtracking is accomplished by the command `!undo [i]`, where `i` is the version of the constraint system to backtrack to:

```
[1] > !undo 0
[0] > !ecr 'y
'y
```

After the `!undo` command, the constraint system reverts to its state just before the constraint was added. Since `'y` is unconstrained, its equivalence class representative is itself.

8.3.4 Persistence

Constraint systems can be saved or loaded to disk. The complete internal representation of a constraint system is saved, so after restoration all operations are legal (including backtracking).

8.4 Using iBanshee

8.4.1 iBanshee Commands

For completeness, we summarize the commands available in *i*BANSHEE.

- `!help` Print the quick reference
- `!tlb e` Print the transitive lower bounds of `e`
- `!ecr e` Print the equivalence class representative of `e`

- `!undo i` Roll back the constraint system to its state at time `i`
- `!trace i` Set the trace level to depth `i`. This command prints constraints for recursive calls to the constraint solver up to `i` levels deep, which is useful for debugging
- `!quit` Exits *i*Banshee
- `!save "filename"` Saves the current constraint system
- `!load "filename"` Loads the constraint system saved using the `save` command
- `!rsave` Save the current constraint system (the filename is currently hardcoded) using region serialization, which is much faster than using `!save`
- `!rload` Load the constraint system saved with `rsave`
- `!exit` Exits *i*Banshee

8.4.2 iBanshee Syntax

Here we summarize *i*Banshee's syntax.

```
ident            : [A-Z a-z]([A-Z a-z 0-9 _])*
integers (i)     : [0-9]+

Variables (v)    : '{ident}
Constructors (c) : {ident}
Labels (l)       : {ident}
Names (n)        : {ident}

Expressions (e)  : v | c | n | c(e1,...,en) | e1 && e2 | e1 || e2
```

```
                | <l1=e1,...,ln=en [| e]> | 0:s | 1:s | _:s
                | pat(c,i,e) | proj(c,i,e) | ( e ) | c^-i(e)

sorts           : basesort | row(basesort)

basesort        : setIF | term

Var decl        : v : sort
Constructor decl : c(s1,...,sn) : basesort
Name decl       : n = e
Sig (s)         : + sort | - sort | = sort

Constraints     : e1 <= e2 | e1 == e2
```

8.5 The Nonspecialized Interface

BANSHEE includes a nonspecialized C library (`[engine/libnsengine.a]`) that provides a direct interface to the constraint solver.

To use BANSHEE as a nonspecialized library, these steps should be followed:

- The analysis designer should include the header file `[engine/nonspec.h]` in their application. In general, this should be the only BANSHEE header included.
- The client should call `nonspec_init` before calling any other BANSHEE functions.
- The client should be linked with the libraries `[engine/libnsengine.a]` and `[libcompat/libregions.a]`.

The header [engine/nonspec.h] contains the complete nonspecialized BANSHEE API.

8.6 The CFL Reachability Interface

We have implemented the Dyck-CFL reachability reduction described in Chapter 5 and have included it with BANSHEE.

The API for Dyck CFL reachability using BANSHEE is generated from the specification file [dyckcfl/dyckcfl_terms.bsp].

See [tests/dyckcfl-spec-test.c] and the associated `make` target in the `tests` directory for a complete example.

Chapter 9

Conclusion

In this dissertation, we have devised novel solutions to many of the practical limitations of previous program analysis toolkits. Specifically, we have shown how a simple heuristic based on backtracking can be used to incrementalize a mixed constraint solver. We have shown how to make the solver persistent (thereby enabling integration into build systems that support separate compilation) using region-based memory management and tracked references. We have also demonstrated a partial evaluation technique that allows the analysis designer to specify their analysis using a concise specification language.

We have also shown that a much broader class of program analyses can be expressed (and moreover efficiently implemented) within the mixed constraint framework. First, we showed that a new reduction from set constraints to Dyck context-free language reachability and demonstrated that a substantial flow analysis application can be solved as efficiently with our generic constraint solver as with a hand-written solver optimized specifically for that analysis. Second, we showed how to add annotations to the constraint solver. Annotations provide a mechanism for incorporating an additional dimension of precision to mixed constraint-based analyses. By basing our

annotations on regular languages, we are able to provide a semantics for annotated constraints. We are also able to extend our specialization technique to the annotations, using finite state machines as a natural, declarative specification language.

Bibliography

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[AFFS98] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, 1998.

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA, 2004. ACM Press.

[BNL03] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Simple and efficient relational querying of software structures. In *Proceedings of the 10th Working Conference on Reverse Engineering*, page 216. IEEE Computer Society, 2003.

[CDW04] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, San Diego, CA, February 4–6, 2004.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CH03] Venkatesh Choppella and Christopher T. Haynes. Source-tracking unification. In Franz Baader, editor, *19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Computer Science*, pages 458–472, Miami, Florida, United States, July 2003. Springer.

[CW02] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM con-*

ference on Computer and communications security, pages 235–244, New York, NY, USA, 2002. ACM Press.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[DI00] C. Demetrescu and G. F. Italiano. Fully dynamic transitive closure: Breaking through the $O(n^2)$ barrier. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 381. IEEE Computer Society, 2000.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS '01: The 8th International Static Analysis Symposium*, Lecture Notes in Computer Science, Paris, France, July 16–18 2001. Springer-Verlag.

[DSST86] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 109–121, 1986.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification*, pages 232–247, 2000.

[FA97] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 114–126, London, United Kingdom, 1997. Springer-Verlag.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[FFA00] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In Jens Palsberg, editor, *Static Analysis, Seventh International Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 175–198, Santa Barbara, CA, June/July 2000. Springer-Verlag.

[FFSA98] Manuel Fähndrich, Jefrrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Montreal, Canada, June 1998.

[FRD99] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. From polymorphic subtyping to cfl reachability: Context-sensitive flow analysis using instantiation constraints. Technical Report MSR-TR-99-84, Microsoft Research, 1999.

[GI91] Zvi Galil and Giuseppe F. Italiano. A note on set union with arbitrary deunions. *Information Processing Letters*, 37(6):331–335, 1991.

[Hei92] Nevin Heintze. *Set Based Program Analysis*. PhD dissertation, Carnegie Mellon University, Department of Computer Science, October 1992.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 104–115. ACM Press, 1995.

[HT01a] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[HT01b] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[JMT99] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on security and Privacy*, 1999.

[LH04] Ondřej Lhoták and Laurie Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM Press, 2004.

[LRY+04] Yanhong A. Liu, Tom Rothamel, Fuxiang Yu, Scott D. Stoller, and Nanjun Hu. Parametric regular path queries. In *Proceedings of the ACM*

SIGPLAN 2004 Conference on Programming Language Design and Implementation, 2004.

[McA02] David McAllester. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537, 2002.

[Mos96] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, 1996.

[MR97] David Melski and Thomas Reps. Interconvertbility of set constraints and context-free language reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 74–89. ACM Press, 1997.

[MR00] David Melski and Thomas Reps. Interconvertibility of a class of set constraints and context-free-language reachability. *Theoretical Computer Science*, 248:29–98, 2000.

[MR05] Ana Milanova and Barbara G. Ryder. Annotated inclusion constraints for precise flow analysis. In *IEEE International Conference on Software Maintenance*, September 2005.

[NIS02] NIST. The economic impacts of inadequate infrastructure for software testing, May 2002. Planning Report 02-3.

[Pal95] Jens Palsberg. Efficient inference of object types. *Information and Computation*, (123):198–209, 1995.

[Rep95] Thomas Reps. Shape analysis as a generalized path problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 1–11. ACM Press, 1995.

[Rep98] Thomas Reps. Program analysis via graph reachability. In *Information and Software Technology*, pages 701–726, 1998.

[Rep00] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. In *ACM Trans. Prorgram. Lang. Syst.*, volume 22, pages 162–186, 2000.

[RF01] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, London, United Kingdom, January 2001.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 43–55, 2001.

[Rod03] Liam Roditty. A faster and simpler fully dynamic transitive closure. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 404–412. Society for Industrial and Applied Mathematics, 2003.

[RR93] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 502–510. ACM Press, 1993.

[RSJ03] Thomas Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proc. 10th Int. Static Analysis Symp.*, pages 189–213, 2003.

[SFA00] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–95. ACM Press, 2000.

[SH97] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the Conference on Programming Language Design and Implementation*. ACM Press, June 2004.

[WT89] J. Westbrook and R. E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM Journal on Computing*, 18(1):1–11, 1989.

[Yan90] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 230–242. ACM Press, 1990.