Regularly Annotated Set Constraints [*]

John Kodumal
UC Berkeley and Coverity, Inc.
jkodumal@coverity.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

A general class of program analyses are a combination of context-free and regular language reachability. We define *regularly annotated set constraints*, a constraint formalism that captures this class. Our results extend the class of reachability problems expressible naturally in a single constraint formalism, including such diverse applications as interprocedural dataflow analysis, precise type-based flow analysis, and pushdown model checking.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages

General Terms Algorithms, Design, Experimentation, Languages, Theory

Keywords Set constraints, context-free language reachability, flow analysis, annotated inclusion constraints, pushdown model checking

1. Introduction

Many program analyses reduce to 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 [20], context-free reachability [23], and the cubic-time fragment of set constraints [10, 1] are all formalisms that describe such analyses. These three approaches are closely related: set constraint solvers are implemented using optimized dynamic transitive closure algorithms [7, 27, 11] and one efficient implementation of context-free reachability is based on a reduction to set constraints [15]. Representative program analysis problems solvable with these methods include polymorphic flow analysis [21] and field-sensitive points-to analysis [26].

There are more complex analysis problems where multiple reachability properties must be satisfied simultaneously; e.g., one can easily define problems that require matching of both function calls/returns and data type constructors/destructors. Unfortunately, analysis problems requiring satisfying two or more context-free properties simultaneously are undecidable [22]. A general, decidable class of reachability properties is the combination of a context-free language with any number of regular languages. A number of natural analysis problems fall into this class [3, 5, 13, 12].

In this paper we extend set constraints to express program analyses involving 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. Our approach builds on an idea first introduced in [25], in which terms and constraints can be annotated with a word from some language. We introduce *regularly annotated set constraints*, where each constructor of a term and each constraint can be annotated with a word from a regular language. The principal contributions of this paper are:

- We introduce regularly annotated set constraints and give a formal semantics (Section 2). Previous work on annotated constraints has not addressed semantics, probably because the applications use only finite languages [19]. Because our annotations are drawn from infinite regular languages, annotations are not bounded in size and understanding even the termination of a constraint solver requires formalization.
- We present algorithms for solving regularly annotated set constraints. Unlike the unannotated case, where all natural solving strategies have the same asymptotic complexity, for annotated constraints we can give algorithms for either *forward* or *backward* solving that are asymptotically faster than *bidirectional* solving (Section 5). If the finite state machine for the regular annotations is small, the bidirectional strategy is practical. This complexity difference shows that for analysis problems reducible to annotated set constraints, whole program analysis, which can be done with forwards or backwards solutions, is apparently more efficient than separate, compositional analysis, which requires bidirectional solving.
- We solve an open problem: We combine previous results [15] with annotations to express a type-based flow analysis supporting polymorphic recursion and non-structural subtyping in a label flow analysis (Section 7). We also 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 (Section 6).

2. Annotated Constraints

This section introduces the syntax and semantics of regularly annotated set constraints, gives an algorithm for solving such constraints, and works through an example.

[*] This work was supported by NSF grants CCR-0085949 and CCF-0430378 and equipment donations from Intel and Dell.

2.1 Set Constraints

Let $c, d, \ldots \in C$ be a set of constructors; each constructor c has arity $a(c)$. Constructors inductively define a *domain* T:

$$T = \{c(t_1, \ldots, t_{a(c)}) | t_i \in T \wedge c \in C\} \cup \{\bot\}$$

Constants (arity zero constructors) and $\bot$ are the base cases of T. There is a standard ordering $\leq$ on elements of T:

$$\begin{array}{rcl} & \bot \leq t & \\ t_1 \leq t'_1 \wedge \ldots \wedge t_n \leq t'_n & \Leftarrow & c(t_1, \ldots, t_{a(c)}) \leq c(t'_1, \ldots, t'_{a(c)}) \end{array}$$

The value $\bot$ represents, as usual, undefined (e.g., non-terminating) terms. The reason for using a domain with $\bot$ instead of the simpler set of ground terms without $\bot$ is that we want constructors to be *non-strict* (i.e., $c(t, \bot) \neq \bot$); we explain why in Section 3.

The class of *set expressions* we consider in this paper are set variables, constructors applied to set variables, and projections:

$$se ::= \mathcal{X} \mid c(\mathcal{X}_1, \ldots, \mathcal{X}_{a(c)}) \mid c^{-i}(\mathcal{X})$$

Set expressions denote *downward-closed* subsets of the domain. A set $T' \subseteq T$ is downward-closed if

$$\forall t \in T.\ (t \in T' \wedge t' \leq t) \Rightarrow t' \in T'$$

Using downward-closed sets simply says that we do not attempt to reason about whether terms are fully defined (have no non-$\bot$ sub-terms). An *assignment* ρ is a mapping from set variables to downward-closed subsets of T. Assignments are extended to set expressions in the natural way:

$$\begin{array}{rcl} \rho(c(\mathcal{X}_1, \ldots, \mathcal{X}_{a(c)})) & = & \{c(t_1, \ldots, t_{a(c)}) | t_i \in \rho(\mathcal{X}_i)\} \\ \rho(c^{-i}(\mathcal{X})) & = & \{t_i | c(t_1, \ldots, t_{a(c)}) \in \rho(\mathcal{X})\} \end{array}$$

A *set constraint* is an inclusion constraint $se_1 \subseteq se_2$ between any two set expressions, except that projections may not appear on the right-hand side of a constraint. An assignment ρ is a *solution* of a constraint $se_1 \subseteq se_2$ if $\rho(se_1) \subseteq \rho(se_2)$.

2.2 Regularly Annotated Set Constraints

Let $M = (\Sigma, S, s_0, \delta, S_{accept})$ be a minimized deterministic finite state automaton (DFA), where Σ is the input alphabet, S the set of states, s_0 and S_{accept} the start state and set of final states respectively, and $\delta(w, s) = s'$ is the machine's transition function. We refer to $L(M)$, the language accepted by M, whenever we need to appeal to a language characterization. Because regular languages are closed under products, it is sufficient to deal only with a single machine representing the product of all the regular reachability properties for a given application.

2.3 Annotated Terms

The first step in our extension is to define the domain of terms. Intuitively, 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 are needed on every constructor in a term, not just at the root; 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 in $L(M)$. The required language of partial words depends on the particular algorithm for computing solutions of the constraints. A *forwards* (*backwards*) solver propagates constructors from lower (upper) bounds to upper (lower) bounds; it turns out that a forwards (backwards) solver should admit prefixes (suffixes) of words in $L(M)$ (see Section 5). A *bidirectional* solver may propagate information either forwards from lower bounds to upper bounds or backwards from upper bounds to lower bounds, and so a bidirectional solver should accept arbitrary substrings of words in $L(M)$. In this paper we focus on bidirectional solving, as it is technically the most interesting case.

The *M-annotated domain* over constructors $c, d, \ldots \in C$ and finite automaton M is

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

Let M^{sub} be the minimal DFA 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 need an operation that appends a word to all levels of an annotated term:

$$\begin{array}{rcl} c^w(t_1, \ldots, t_{a(c)}) \cdot w' & = & c^{ww'}(t_1 \cdot w', \ldots, t_{a(c)} \cdot w') \\ \bot \cdot w' & = & \bot \end{array}$$

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

2.4 Annotated Set Constraints

An *M-regularly annotated set constraint* is an inclusion constraint $se_1 \subseteq^x se_2$, where se_1, se_2 are set expressions and $x \in \Sigma \cup \{\epsilon\}$. We often abbreviate $se_1 \subseteq^\epsilon se_2$ by dropping the annotation $se_1 \subseteq se_2$. Note set expressions are not themselves annotated; we do not need to burden the analysis designer with annotating set expressions, because it is possible to infer the needed set expression annotations during constraint resolution.

We next define assignments ρ mapping set expressions to sets of annotated ground terms. We extend set expressions with *word set variables* attached to each constructor:

$$se ::= \mathcal{X} \mid c^\alpha(\mathcal{X}_1, \ldots, \mathcal{X}_{a(c)}) \mid c^{-i}(\mathcal{X})$$

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.

$$\begin{array}{rcl} \rho(c^\alpha(\mathcal{X}_1, \ldots, \mathcal{X}_{a(c)})) & = & \{c^w(t_1, \ldots, t_{a(c)}) | w \in \rho(\alpha) \wedge t_i \in \rho(\mathcal{X}_i)\} \\ \rho(c^{-i}(\mathcal{X})) & = & \{t_i | c^w(t_1, \ldots, t_{a(c)}) \in \rho(\mathcal{X})\} \end{array}$$

An assignment ρ is a solution of a system of annotated constraints $\{se_1 \subseteq^w se_2\}$ if $\rho(se_1) \cdot w \subseteq \rho(se_2)$ for all constraints. Solutions may assign arbitrary sets to the word and term variables, provided they satisfy the constraints. We now show a restricted family of solutions, the *M-regular solutions*, are sufficient to characterize all solutions. We define the following congruence on words in $L(M^{sub})$:

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

Theorem 2.1. $w \equiv_M w' \implies \forall s \in S.\ \delta(w, s) = \delta(w', s)$

Proof. This result follows from the Myhill-Nerode theorem; we include the proof because it is useful in understanding our constraint resolution algorithm. Assume $w \equiv_M w'$ yet there is an $s \in S$ 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, implying there is a word x with $\delta(x, s_0) = s$. Minimality also implies the existence of a word y where $\delta(y, s_i) \in S_{accept}$ and $\delta(y, s_k) \notin S_{accept}$, or vice versa. Without loss of generality, assume the first case. Then $xwy \in L(M)$ but $xw'y \notin L(M)$, contradicting the assumption that $w \equiv_M w'$. $\square$

Thus, each word equivalence class W defines a unique *representative function* from states to states: $f(s) = \delta(w, s)$ for $w \in W$. Associated with every automaton M is a finite set of representative functions $F_M^{\overline{\equiv}} = \bigcup_i F_M^i$, computed inductively as follows:

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

The representative function for the word ϵ in any machine is the identity f_ϵ mapping each state to itself. We appeal to this function characterization of equivalence classes shortly.

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

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

An assignment ρ is *M-regular* if

$$\begin{array}{rcl} w \in \rho(\alpha) \wedge w \equiv_M w' & \Longrightarrow & w' \in \rho(\alpha) \\ t \in \rho(X) \wedge t \equiv_M t' & \Longrightarrow & t' \in \rho(X) \end{array}$$

Lemma 2.2. If $t \equiv_M t'$ and $t \cdot w \in T^{M^{sub}}$ then $t \cdot w \equiv_M t' \cdot w$.

Proof. By induction on the structure of t. Without loss of generality, assume $t = c^x(t_1, \ldots, t_{a(c)})$. Because $t \equiv_M t'$, we know $t' = c^y(t'_1, \ldots, t'_{a(c)})$ where $x \equiv_M y$ and $t_i \equiv_M t'_i$.

$$\begin{array}{rcl} c^x(t_1, \ldots, t_{a(c)}) \cdot w & = & \text{def. of } \cdot \\ c^{xw}(t_1 \cdot w, \ldots, t_{a(c)} \cdot w) & \equiv_M & \\ c^{xw}(t'_1 \cdot w, \ldots, t'_{a(c)} \cdot w) & \equiv_M & \\ c^{yw}(t'_1 \cdot w, \ldots, t'_{a(c)} \cdot w) & = & \text{def. of } \cdot \\ c^y(t'_1, \ldots, t'_{a(c)}) \cdot w & & \end{array}$$

For the second step, $t \cdot w \in T^{M^{sub}}$ implies that, for each i, $t_i \cdot w \in T^{M^{sub}}$ and therefore we can apply the induction hypothesis to conclude $t_i \cdot w \equiv_M t'_i \cdot w$. For the third step, observe $x \equiv_M y$ implies that $xw \equiv_M yw$. □

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

Theorem 2.3. If ρ is a solution of a system of *M*-annotated constraints, then its M-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 $t \cdot w \in \rho'(e_2)$.

$$\begin{array}{rcl} t \in \rho'(e_1) & \Rightarrow & \text{def. of regular completion} \\ \exists t' \equiv_M t.\, t' \in \rho(e_1) & \Rightarrow & \rho \text{ is a solution} \\ t' \cdot w \in \rho(e_2) & \Rightarrow & \rho' \geq \rho \\ t' \cdot w \in \rho'(e_2) & \Rightarrow & \\ t \cdot w \in \rho'(e_2) & & \end{array}$$

The last step follows from Lemma 2.2, using $t' \equiv_M t$ 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 and solutions range over subsets of $T^{M^{sub}}$). □

Because the regular completion of every solution is also a solution, it suffices to compute only regular solutions. Furthermore, in the regular solutions, each word set variable represents a set of full equivalence classes, and by Theorem 2.1 each equivalence class corresponds to a unique representative function. Thus, we can replace word set variables with representative function variables and each annotation with the representative function for the annotation's equivalence class and deal with finite sets of representative functions instead of infinite sets of words. A mapping $\rho(\alpha) = \{f_1, f_2, \ldots\}$ corresponds to the regular solution

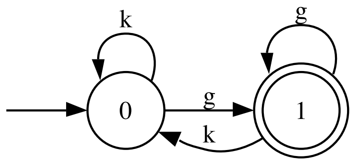

Figure 1. Finite state automaton M_{1bit} for the 1-bit language.

$$\rho(\alpha) = \{w \mid \exists f \in \{f_1, f_2, \ldots\}. \forall s \in S. f(s) = \delta(s, w)\}$$

From here on we 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 the input automaton is M_{1bit} shown in Figure 1.

Example 2.4. Consider the following constraint system:

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

Let f_g be the representative function containing the word g; $f_g(0) = 1$ and $f_g(1) = 1$. The solution for the function variables is $\alpha, \beta = \{f_\epsilon\}$, $\gamma = \{f_g\}$ and for set variables (the downward-closure of) $\mathcal{Y}, \mathcal{W} = \{c^{f_g}\}, \mathcal{X} = \{o^{f_g}(c^{f_g})\}$, and $\mathcal{Z} = \{o^{f_g}(c^{f_g})\}$. Note that the solution instantiates the constraint

$$o^\beta(\mathcal{W}) \subseteq^g \mathcal{X}$$

to

$$o^{f_\epsilon}(c^{f_g}) \subseteq^{f_g} o^{f_g}(c^{f_g})$$

The left-hand side illustrates that annotations on different levels of constructors within a term are not always the same.

3. Solving Annotated Constraints

Our resolution algorithm uses a standard, two-phase approach:

- The first phase non-deterministically 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?

3.1 Resolution Rules

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

$$\begin{array}{rl} c^\alpha(\mathcal{X}_1, \ldots, \mathcal{X}_{a(c)}) \subseteq^f c^\beta(\mathcal{Y}_1, \ldots, \mathcal{Y}_{a(c)}) & \Rightarrow \\ \bigwedge_i \mathcal{X}_i \subseteq^f \mathcal{Y}_i \wedge f \circ \alpha \subseteq \beta & \\ & \\ c^\alpha(\ldots) \subseteq^f d^\beta(\ldots) & \Rightarrow \\ \text{no solution} & \end{array}$$

The first rule propagates inclusions between constructed terms to components and produces *representative function constraints* on function variables annotating constructor expressions: the function annotations β on the right-hand side constructor expression must 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 $F_{\overline{M}}^{\equiv}$ of representative functions is known and fixed for a given machine M, these function constraints are themselves simple examples of set constraints. The second resolution rule simply recognizes manifestly inconsistent constraints where the top-level constructors do not match.

The third rule resolves projection constraints.

$$c^{\alpha}(\ldots, \mathcal{X}_i, \ldots) \subseteq \mathcal{Y} \;\wedge\; c^{-i}(\mathcal{Y}) \subseteq \mathcal{Z} \quad \Rightarrow \quad \mathcal{X}_i \subseteq \mathcal{Z}$$

These rules should be interpreted as follows: whenever there are constraints satisfying the left-hand side of the rule, the right-hand side constraint is added to the system of constraints. Because constraints are only added the rules are automatically sound (no new solutions are created by the rules). It is easy to check that the rules are also complete (each rule preserves all solutions of the original system of constraints). These rules are not complete for a semantics that uses strict constructors (i.e., if one component of a constructor expression is $\perp$ then the entire construction collapses to $\perp$), and indeed this is the reason we have chosen to use the non-strict semantics here. Constraints with at least one binary strict constructor are much more expensive to solve [2]; in practice, all implementations we know of assume a non-strict semantics for constructors.

Finally, a transitive closure rule propagates annotations by composing them:

$$se_1 \subseteq^{f} \mathcal{X} \subseteq^{g} se_2 \Rightarrow se_1 \subseteq^{g \circ f} se_2$$

Because M is minimized, no work need be done propagating annotations that are necessarily non-accepting, obviating the need for a *match* operation as in [19]. Returning to Example 2.4, the solved form of this system is:

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

Note 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 $f_g \circ f_g = f_g$ (see Figure 1).

Lemma 3.1. Constraint resolution 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 functions on a finite set of states to itself is bounded, the total number of possible constraints is also bounded. □

3.2 Queries

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

More precisely, we say that constraints $\mathcal{C}_1$ *entail* constraints $\mathcal{C}_2$, written $\mathcal{C}_1 \models \mathcal{C}_2$, if every solution of $\mathcal{C}_1$ is a solution of $\mathcal{C}_2$. 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 in t and $F_{accept} = \{f \mid f \in F_{\overline{M}}^{\equiv} \wedge f(s_0) \in S_{accept}\}$. Intuitively, F_{accept} is the subset of representative functions $F_{\overline{M}}^{\equiv}$ that lead to an accept state from the initial state; these functions represent full words in $L(M)$. We can now explain important aspects of querying annotated constraints:

- 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 [10]; queries in this case are particularly easy to solve. For example, for a constant c^{α}, the entailment

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

  holds if and only if a constraint $c^{\alpha} \subseteq^{f} \mathcal{X}$ is present in the solved form system $\mathcal{C}$ where $f(s_0) \in S_{accept}$.
- Demand driven solvers essentially move all of the work of resolution to queries [11]. As an optimization, our solver does not generate function constraints or annotate constructors at all during resolution and this decision has important performance advantages (see Section 8). For our queries, the representative function constraints needed to answer a query can be reconstructed as part of the entailment computation itself.
- Our applications use 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 constructors must be annotated in certain states. Such a query can be used to search for the existence of a term denoting an error in the program (e.g when checking finite state properties). 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 2.4, 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 2.4.

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 $L(M)$. Solving in the larger domain $T^{M^{sub}}$ still guarantees termination and also preserves all entailment queries using words in $L(M)$.

3.3 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 [12]. 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 `gens` and `kills` are idempotent. Figure 1 shows the finite state automaton M_{1bit} for a single dataflow fact. For this language, $F_{\overline{M}}^{\equiv} = \{f_{\epsilon}, f_g, f_k\}$, since $f_g \circ f_g = f_g$ and $f_k \circ f_g = f_{\epsilon}$ and so on. Thus, we need not track arbitrary sequences of `gens` and `kills`; it suffices to track just three different possible annotations. An n-bit language can be derived from a product construction.

4. Complexity

We sketch a complexity argument for the constraint resolution algorithm described in Section 3. A system of constraints containing no annotations can be solved in $O(n^3)$ time, where n is the size (measured in number of symbols) of the system. Each of 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 compute a new number of lower and upper bounds. Consider a particular lower bound $se \subseteq \mathcal{X}$ (the argument for upper bounds is similar). There may be one lower bound $se \subseteq^f \mathcal{X}$ between se and $\mathcal{X}$ for each distinct equivalence class f annotating the constraints, so the problem is to bound the number of distinct representative functions. 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.

Thus, each variable in an annotated system can have up to $n \cdot |F_M^{\equiv}|$ lower bounds and $n \cdot |F_M^{\equiv}|$ upper bounds. With representative functions as the annotations on constraints, function composition can be implemented as a precomputed table and so $f \circ g$ can be computed in constant time via table lookup. Thus, 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 $O(n^3\,|F_M^{\equiv}|^2)$. This generic argument may be sharpened for the constraints generated by a particular application.

Now we relate complexity in terms of $|F_M^{\equiv}|$ to $|S|$, the number of states in M. One can build an adversarial 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 each of the $|S|^{|S|}$ possible functions from domain S to range S. One such machine, given in Figure 2, uses three operations *rotate*, *swap*, and *merge* to generate the entire space of functions as follows:

- *rotate* maps state i to state $i+1$, with wraparound.
- *swap* maps state 1 to state 2, and state 2 to state 1, while mapping every other state to itself. Every sequence of rotates and swaps generates a permutation of the n states of the automaton, and every permutation is generated by some sequence of rotates and swaps.
- *merge* allows the generation of information-losing functions (functions that are not injective) by mapping state 1 to itself, state 2 to state 1, and any other state to itself. Using an appropriate sequence of swaps and rotates first, any pair of function outputs can be merged.

This machine is clearly a pathologically bad case, yet the possibility of having an implementation that is superexponential in the size of the automata specification means that there could be examples that are simply too costly to implement using bidirectional solving. On the other hand, we have looked at a number of fairly complex finite state properties and have not run into this limitation in practice. In the next subsection, we discuss alternative solver strategies that trade off the ability to do separate analysis for improved asymptotic complexity.

5. Alternative Solution Strategies

We have assumed constraint resolution operates by applying resolution rules in any order until the system is solved. We call this strategy *bidirectional solving* because the constraints are "extended" either forwards or backwards via transitive closure. Two natural

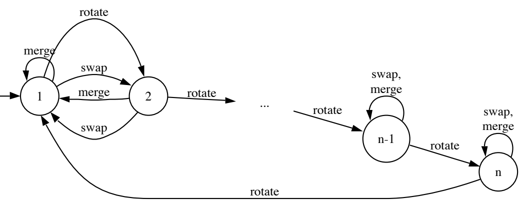

Figure 2. A machine M with a small alphabet where $|F_M^{\equiv}|$ is large.

alternatives are *forward* solving and *backward* solving. 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: pushing $f(c)$ forwards adds $f(c) \subseteq \mathcal{Y}$ and discovers $f(c) \subseteq f(\mathcal{Z})$. A backwards solver only applies transitive closure by pushing upper bound sinks towards lower bound sources: pushing $f(\mathcal{Z})$ backwards adds $\mathcal{X} \subseteq f(\mathcal{Z})$ to discover the same fact $f(c) \subseteq f(\mathcal{Z})$. Bidirectional solvers can add both new constraints, in any order.

5.1 Tradeoffs

Bidirectional solving has two advantages. Bidirectional solving enables separate analysis, because the closure rules do not need all sources and sinks to be present to solve the available constraints. A related advantage is that constraints can be solved online. Unidirectional solvers defer most processing until the entire constraint graph is built.

On the other hand, unidirectional solvers can naturally be made demand driven. For example, solving forward from a constant can selectively answer the query "For what set of variables must this constant appear in every solution?" Bidirectional solvers typically compute a representation of all solutions prior to the query phase.

The choice of solver directionality also affects the complexity of constraint resolution. 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 words in $L(M)$. In the forwards case, we relax the congruence $\equiv_M$ to a right congruence $\equiv_r$:

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

Here, $F_M^{\equiv_r}$ only distinguishes functions that map s_0 to one of the different $|S|$ states in M, by the following analog of Theorem 2.1:

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

The construction for the backwards case is symmetric, using a left congruence in place of a right congruence.

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 $F_M^{\equiv_r}$. Notice that the forwards solver always computes new annotations as $g \circ f$, where $f \in F_M^{\equiv_r}$ and $g \in F_M^{\equiv}$. Furthermore, the resulting function $g \circ f$ is always in $F_M^{\equiv_r}$. In the bidirectional case, both the initial and derived annotations are 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 of M. In the unidirectional case, i is exactly $|S|$. In the bidirectional case, i may be as much as $|S|^{|S|}$.

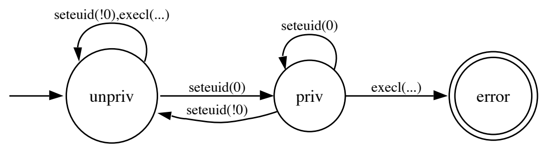

Figure 3. Automaton for process privilege.

f_0	:	unpriv	$\rightarrow$	priv		f_2	:	unpriv	$\rightarrow$	unpriv
		priv	$\rightarrow$	priv				priv	$\rightarrow$	error
		error	$\rightarrow$	error				error	$\rightarrow$	error
f_1	:	unpriv	$\rightarrow$	unpriv		f_{error}	:	unpriv	$\rightarrow$	error
		priv	$\rightarrow$	unpriv				priv	$\rightarrow$	error
		error	$\rightarrow$	error				error	$\rightarrow$	error

Figure 4. A few of the representative functions in $F_M^{\equiv}$ for the process privilege model.

6. 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 for finding security bugs in C code [5]. Following the approach of [5], 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()` function. A program that violates this property may give an untrusted program full access to the system. Figure 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.

6.1 Modeling Programs with Constraints

We now show how to find violations of temporal safety properties using annotated constraints. With 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. The annotations are program statements relevant to the security property (i.e., the statements labeling transitions in Figure 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 $o_i^{-1}(\mathcal{F}_{exit}) \subseteq \mathcal{S}_i$, where $\mathcal{F}_{entry}$ (resp. $\mathcal{F}_{exit}$) is the node representing the entry (resp. 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.

6.2 Checking for Security Violations

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 *positive-negative reachability* (PN reachability [15]), which allows partially matched call-return paths (e.g., a path from a caller to callee that does not return to the caller is partially matched because it has the constructor but not the canceling projection). 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, the sequence of constructors in the term represent a possible runtime stack) that leads to the error.

6.3 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. The surface syntax constraints for this example are in the left column below. Translating the example to our internal syntax, we compute the set $F_M^{\equiv}$ and replace word annotations with representative functions in the right column below (relevant functions are shown in Figure 4):

$$\begin{array}{cc}
pc \subseteq \mathcal{S}_1 & pc^{f_\epsilon} \subseteq \mathcal{S}_1 \\
\mathcal{S}_1 \subseteq^{seteuid(0)} \mathcal{S}_2 & \mathcal{S}_1 \subseteq^{f_0} \mathcal{S}_2 \\
\mathcal{S}_2 \subseteq \mathcal{S}_3 & \mathcal{S}_2 \subseteq \mathcal{S}_3 \\
\mathcal{S}_2 \subseteq \mathcal{S}_4 & \mathcal{S}_2 \subseteq \mathcal{S}_4 \\
\mathcal{S}_3 \subseteq^{seteuid(!0)} \mathcal{S}_5 & \mathcal{S}_3 \subseteq^{f_1} \mathcal{S}_5 \\
\mathcal{S}_4 \subseteq \mathcal{S}_5 & \mathcal{S}_4 \subseteq \mathcal{S}_5 \\
\mathcal{S}_5 \subseteq^{execl(\ldots)} \mathcal{S}_6 & \mathcal{S}_5 \subseteq^{f_2} \mathcal{S}_6
\end{array}$$

Constraint resolution 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 $pc^{f_{error}}$ is in $\mathcal{S}_6$, the constraint variable corresponding to the program point after the `execl` call, indicating the presence of a possible security vulnerability.

6.4 Parametric Annotations

Pushdown model checking applications sometimes require a limited ability to correlate pieces of data. An excellent example is an analysis that tracks the opening and closing of files; the automaton for this analysis is shown in Figure 5. The annotations `x =`

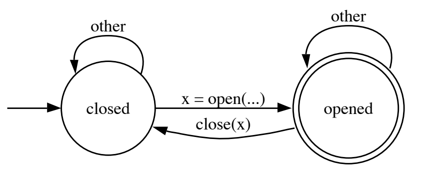

Figure 5. Automaton for tracking file state.

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

Figure 6. Example C program that manipulates file descriptors.

`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 we show a simple program that manipulates two file descriptors $\texttt{fd}_1$ and $\texttt{fd}_2$. We would like an analysis that determines that $\texttt{fd}_2$ remains open at the end of the program, but not $\texttt{fd}_1$. Type systems and dataflow analyses naturally incorporate this kind of information using a type environment or symbolic map which can separately track the types of multiple names, and we take a similar approach. Adding this capability allows us to faithfully reproduce the functionality of a particular pushdown model checker, MOPS [4], that we use for our experiments (see Section 8).

To separate the states of each of the file descriptors in the program, the automaton in Figure 5 must be instantiated for each possible file descriptor that occurs in the input program. However, our approach precludes explicit instantiation because we compile away the automaton statically, before the input program is available (recall Section 4).

Instead of explicit instantiation, we perform instantiations on-the-fly by maintaining 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:

$$[(x : \texttt{fd}_1) \mapsto f;\ (x : \texttt{fd}_2) \mapsto g \mid r_\phi]$$

This particular substitution environment contains two entries: one where parameter x is instantiated to $\texttt{fd}_1$ and another where x is instantiated to $\texttt{fd}_2$. The residual function is r_ϕ. The intuition is that this substitution tracks two copies of the automation in Figure 5. If any new instantiations are added via composition (e.g. $(x : \texttt{fd}_3)$), the residual r_ϕ is 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

$$\begin{array}{lllll} f_1 & : & \text{closed} & \rightarrow & \text{opened} \\ & & \text{opened} & \rightarrow & \text{opened} \\ f_2 & : & \text{closed} & \rightarrow & \text{closed} \\ & & \text{opened} & \rightarrow & \text{closed} \end{array} \qquad \begin{array}{lll} \phi_1 & = & [(x : \texttt{fd}_1) \mapsto f_1 \mid f_\epsilon] \\ \phi_2 & = & [(x : \texttt{fd}_2) \mapsto f_1 \mid f_\epsilon] \\ \phi_3 & = & [(x : \texttt{fd}_1) \mapsto f_2 \mid f_\epsilon] \end{array}$$

Figure 7. A few of the representative functions and substitution environments for the file state example.

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 allude to this by dropping the brackets when writing an empty substitution environment: e.g. $[\ \mid r]$ is simply written r. Before discussing how to handle multiple parameters, we walk through the example from Figure 6.

6.4.1 Example

In the surface syntax, the program in Figure 6 results in the system of constraints shown in the left column below. Internally, the constraint system is translated into the system of constraints in the right column (Figure 7 gives the definitions of the relevant representative functions and substitution environments):

$$\begin{array}{cc} pc \subseteq \mathcal{S}_1 & pc^{f_\epsilon} \subseteq \mathcal{S}_1 \\ \mathcal{S}_1 \subseteq^{open(\texttt{fd}_1)} \mathcal{S}_2 & \mathcal{S}_1 \subseteq^{\phi_1} \mathcal{S}_2 \\ \mathcal{S}_2 \subseteq^{open(\texttt{fd}_2)} \mathcal{S}_3 & \mathcal{S}_2 \subseteq^{\phi_2} \mathcal{S}_3 \\ \mathcal{S}_3 \subseteq^{close(\texttt{fd}_1)} \mathcal{S}_4 & \mathcal{S}_3 \subseteq^{\phi_3} \mathcal{S}_4 \end{array}$$

These constraints imply $pc \subseteq^{\phi_3 \circ \phi_2 \circ \phi_1} \mathcal{S}_4$ where:

$$\phi_3 \circ \phi_2 \circ \phi_1 = [(x : \texttt{fd}_1) \mapsto f_2;\ (x : \texttt{fd}_2) \mapsto f_1 | f_\epsilon]$$

6.4.2 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[1] 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.

7. 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 algorithms for context-sensitive, field-sensitive flow analysis. A proof by Reps shows the general problem to be undecidable [22]. As mentioned previously, the core issue is the arbitrary interleaving of two matching properties: function calls and

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

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

$$\frac{\Gamma \vdash e : \sigma_1 \times^{\mathcal{L}} \sigma_2}{\Gamma \vdash e.i : \sigma_i}\ (\text{Proj } i = 1, 2) \qquad \frac{\Gamma, x : \sigma, f : \sigma \rightarrow \sigma' \vdash e : \sigma'}{\Gamma \vdash f(x : \tau) : \tau' = e : \sigma \rightarrow \sigma'}\ \text{(Def)}$$

$$\frac{\Gamma \vdash e : \sigma \quad \vdash \sigma \leq \sigma'}{\vdash e : \sigma'}\ \text{(Sub)} \qquad \frac{\Gamma(f) = \sigma \quad \sigma \preceq^i_+ \sigma'}{\Gamma \vdash f^i : \sigma'}\ \text{(Inst)}$$

Figure 8. Type rules for polymorphic recursive system.

$$\frac{tl(\sigma) \subseteq tl(\sigma')}{\sigma \leq \sigma'}\ \text{(Sub)} \qquad \frac{o_i^{-1}(tl(\sigma)) \subseteq tl(\sigma')}{\sigma \preceq^i_+ \sigma'}\ \text{(Pos)}$$

$$\frac{o_i(tl(\sigma')) \subseteq tl(\sigma)}{\sigma \preceq^i_- \sigma'}\ \text{(Neg)} \qquad \frac{\sigma_1 \preceq^i_- \sigma_3 \quad \sigma_2 \preceq^i_+ \sigma_4}{\sigma_1 \rightarrow \sigma_2 \preceq^i_+ \sigma_3 \rightarrow \sigma_4}\ \text{(Fun)}$$

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

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

Figure 9. Constraint generation for polymorphic recursive system.

returns, and type constructors and destructors. Viewing Reps' result in a type-based setting, 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 7.6), 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 a previously known reduction strategy to model context-free language reachability of function matchings as a set constraint problem [15]. 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.

7.1 Source Language

The analysis operates on the following source language:

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

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:

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

Function tl returns the top-level label of a labeled type.

7.2 Type Rules and Constraint Generation

Figure 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.

The constraint generation rules are shown in Figure 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.[2] Constraints between substructures of types are discovered automatically as needed during constraint resolution.

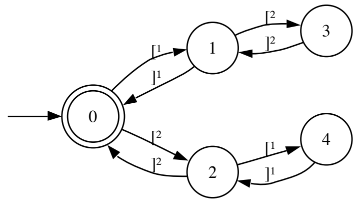

Figure 10. Finite state automaton for single level pairs.

7.2.1 Function Call Matching with Terms

We apply a known reduction to model the matching of function calls and returns [15]. The result in [21] shows that this is equivalent to polymorphic recursive treatment of functions.

7.2.2 Type Constructor Matching with Annotations

Annotations are used to model the matching language of type constructors and destructors. For example, in $(x^{\mathcal{X}}, y^{\mathcal{Y}})^{\mathcal{P}}.1^{\mathcal{Z}}$, 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 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 10 we show the finite state

[2] 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 for brevity.

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

Figure 11. Non-structural subtyping example.

$$\begin{tikzcd}
 & \mathcal{P} \arrow[rrr, bend left] & & & \mathcal{H} \arrow[dd, dashed, "o_i^{-1}"] & \\
\mathcal{A} \arrow[ur, "{[}^1_{int}"] & & \mathcal{Y} \arrow[ul, "{[}^2_{int}"'] & & & \\
 & & & & \mathcal{T} \arrow[dl, "{]}^1_{int}"'] \arrow[dr, "{]}^2_{int}"] & \\
 & & \mathcal{B} \arrow[uu, dashed, "o_i"] & \mathcal{U} & & \mathcal{V}
\end{tikzcd}$$

Figure 12. Constraint graph for the program in Figure 11 (only relevant edges are shown).

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.

7.3 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 every solution of $\mathcal{Y}$. This query yields answers for *matched* flow, and this approach can be extended to partially-matched reachability through functions using PN reachability [15].

7.4 An Example

Consider the program in Figure 11 (from [8]). Non-structural subtyping assigns `pair` type $int^{\mathcal{Y}} \to \beta^{\mathcal{H}}$ along with constraint $\beta = int^{\mathcal{A}} \times^{\mathcal{P}} int^{\mathcal{Y}}$. Figure 12 shows a simplified constraint graph for this program. Flow from $\mathcal{B}$ to $\mathcal{V}$ is captured by constraints:

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

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

7.5 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 this is 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 $pt(\texttt{x}) = pt(\texttt{y}) = \{a, b\}$, and the analysis reports 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 and downward closure elided) representing points-to sets for the above program:

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

Intersecting the solutions for X and Y reveals that there are no common terms; hence the two variables are not aliased. Thus, the constraint solutions themselves encode 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 can destroy the precision of the analysis. Stack-aware alias queries use the call stack to disambiguate object allocation sites, yielding yet another dimension of precision for context-sensitive alias analysis. While stack-aware queries are not likely to be as significant as other recent approaches to context sensitivity (for example, object sensitivity), the benefit comes with almost no cost, as stack-aware alias queries are encoded in the constraint solutions already.

7.6 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 projections $o_i^{-1}(\ldots)$ 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 $pair^{-1}(\ldots)$ and $pair^{-2}(\ldots)$ to deconstruct a pair. This more natural representation can improve performance, as edge additions that would be discovered twice using unary constructors can be discovered once instead using a binary constructor. With this approach, the constraint system for the example program in Figure 11 is as follows:

$$\begin{array}{rcl}
\mathcal{B} & \subseteq^{[_i} & \mathcal{Y} \\
pair(\mathcal{A}, \mathcal{Y}) & \subseteq & \mathcal{H} \\
\mathcal{H} & \subseteq^{]_i} & \mathcal{T} \\
pair^{-2}(\mathcal{T}) & \subseteq & \mathcal{V}
\end{array}$$

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

8. Experimental Results

We have implemented regularly annotated set constraints in the publicly available BANSHEE toolkit [16, 14], which allows the specification of program analyses using multiple, mixed constraint formalisms [6]. We have added support for annotations to BANSHEE's existing implementation of the Set sort. Some of the technical details (such as handling projection merging [27] and cycle elimination [7] in the presence of annotations) are similar to those addressed in [25]; 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 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 3.2). 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 is reduced to table lookups by the specializer.

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 set constraints. We chose a security property (Property 1 from [4]) and checked several sensitive software packages for security violations using the approach outlined in Section 6. The property we checked is a complete model of the simple process privilege property described in Section 6. The complete model contains 11 states and 9 different alphabet symbols. It is the most complex property and largest automaton reported in [4]. This property demonstrates that, in practice, the representative function sets may 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 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 the package for both BANSHEE and MOPS.[3] 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. Performance is even better considering that MOPS uses significant hand-coded optimizations external to the core push-down model checker. In particular, for improved performance MOPS first slices the program to extract the security-relevant portion in a pre-pass, resulting in a much smaller program to be checked by the push-down implementation, whereas our implementation simply generates and solves constraints for the entire program.

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 1. Benchmark data for process privilege experiment.

9. Related Work

Regularly annotated set constraints generalize the annotated inclusion constraints presented in [25, 19]. 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. We also believe that the annotation languages used in [19] can be expressed in terms of finite state automata.

Parametric regular path queries are a declarative way of specifying graph queries as regular expression patterns [18]. Regular path queries are not as powerful as set constraints, though the use of parameters to correlate related data may be a useful addition. The combination of polymorphic recursion and non-structural subtyping was first considered in [8]. The solution proposed in [8] has the disadvantage that polymorphism on data types is achieved by copying constraints. While there is no implementation of this algorithm, the general experience with constraint-copying implementations is that they are slow [9]. 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. However, our results also suggest that a bidirectional solver is unlikely to scale for this problem, as the number of states of the DFA grows at least with the size of the largest type in the program being analyzed. We believe a whole-program analysis using a forwards or backwards solver (in the style of [11]) would scale; unfortunately BANSHEE includes only bidirectional solvers and currently no forwards or backwards solvers for set constraints are publicly available.

Weighted pushdown systems (WPDS) label transitions with values from a domain of weights [24]. 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.

Binary Decision Diagrams (BDDs) have been successful as an alternative to graph reachability for program analysis applications [28, 17]. BDD-based algorithms have exponential worst-case complexity, whereas annotated set constraints can be solved in polynomial time (but only using forwards or backwards solvers). Also, at

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

least to date BDD-based approaches have not been integrated with context-free properties.

10. Conclusion

We have described a formalism that extends set constraints with annotations drawn from a regular language, allowing the expression of reachability problems involving simultaneous context-free and regular properties. We have shown how to express applications as diverse as type-based flow analysis, interprocedural dataflow analysis, and pushdown model checking.

References

[1] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proc. of the Second International Workshop on Types in Compilation*, 1998.

[2] A. Aiken, D. Kozen, M. Vardi, and E. Wimmers. The complexity of set constraints. In *Proc. of the 7th Workshop on Computer Science Logic*, pages 1–17. Springer-Verlag, 1994.

[3] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proc. of the Symp. on Theory of Computing*, pages 202–211, 2004.

[4] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Proc. of the 11th Annual Network and Distributed System Security Symp.*, pages 171–185, Feb. 4–6, 2004.

[5] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proc. of the 9th ACM Conf. on Computer and Communications Security*, pages 235–244, 2002.

[6] M. Fähndrich and A. Aiken. Program analysis using mixed term and set constraints. In *Proc. of the 4th International Symp. on Static Analysis*, pages 114–126. Springer-Verlag, 1997.

[7] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 85–96, June 1998.

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

[9] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In *Proc. of the Static Analysis Symposium*, pages 175–198, June 2000.

[10] N. Heintze. *Set Based Program Analysis*. PhD dissertation, Carnegie Mellon University, Department of Computer Science, Oct. 1992.

[11] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of c code in a second. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 254–263, 2001.

[12] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proc. of the Symp. on Foundations of Software Engineering*, pages 104–115. ACM Press, 1995.

[13] T. Jensen, D. L. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proc. of the 1999 IEEE Symp. on security and Privacy*, 1999.

[14] J. Kodumal. Banshee: A toolkit for constructing constraint-based analyses. http://banshee.sourceforge.net, 2005.

[15] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proc. of the Conf. on Programming Language Design and Implementation*, pages 207–218, 2004.

[16] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *Proc. of the 12th International Static Analysis Symposium*, pages 218–234, Sept. 2005.

[17] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Proc. of the Conf. on Programming Language Design and Implementation*, 2004.

[18] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Proc. of the Conf. on Programming Language Design and Implementation*, 2004.

[19] A. Milanova and B. Ryder. Annotated inclusion constraints for precise flow analysis. In *IEEE International Conf. on Software Maintenance*, Sept. 2005.

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

[21] J. Rehof and M. Fähndrich. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In *Proc. of the Symp. on Principles of Programming Languages*, pages 54–66, Jan. 2001.

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

[23] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the Symp. on Principles of Programming Languages*, pages 49–61, Jan. 1995.

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

[25] A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java using annotated constraints. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 43–55, 2001.

[26] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for Java. In *Proc. of the Conf. on Object-Oriented Programs, Systems, Languages, and Applications*, 2005.

[27] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proc. of the Symp. on Principles of Programming Languages*, pages 81–95, 2000.

[28] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the Conf. on Programming Language Design and Implementation*, June 2004.