Regularly Annotated Set Constraints

John Kodumal
EECS Department
University of California, Berkeley
jkodumal@cs.berkeley.edu

Alex Aiken
Computer Science Department
Stanford University
aiken@cs.stanford.edu

Abstract

A general class of program analyses can be characterized as a combination of context-free and regular language reachability. We define *regularly annotated set constraints*, a constraint formalism that captures this class of analysis problems. We give a constraint resolution algorithm and show experimentally that an implementation of our approach is both efficient and scalable. Our results considerably 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 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 [17], context-free reachability [20], and the cubic-time fragment of set constraints [8, 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 [5, 24, 9] and the most efficient general implementation of context-free reachability is based on a reduction to set constraints [12]. Representative program analysis problems that can be solved with these methods include polymorphic flow analysis [18] and field-sensitive points-to analysis [23].

There are 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 [19]. 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 [2, 4, 11, 10].

In this paper 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 [22], 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. The principal contributions of this paper are as follows:

- 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 [16]. 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 [16], and the regular language formulation is amenable to automatic generation of the constraint resolution rules.
- We give a novel and very efficient constraint resolution algorithm for solving regularly annotated set constraints. Our preliminary experiments show that this algorithm is not only fast and scalable in theory, but also in practice.
- We show how to combine previous results [12] with annotations to express a type-based flow analysis that supports polymorphic recursion and non-structural subtyping in a label flow analysis. 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.

The remainder of this paper is organized as follows: in Section 2 we introduce the new constraint formalism and provide a

semantics. Sections 3 and 4 present applications of our formalism. Section 5 describes our implementation and presents experimental results. Section 6 contains related work, and Section 7 concludes.

2. Annotated Constraints

In this section, we introduce the syntax and semantics of regularly annotated set constraints, give an algorithm for solving such constraints, and work through an example.

2.1 Set Constraints

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

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

Note that constructors may be constants (arity zero); the constants form the base case of the definition of T.

Set expressions are terms over set variables $\mathcal{X}, \mathcal{Y}, \ldots$:

$$se ::= \mathcal{X} \mid c(se_1, \ldots, se_{a(c)})$$

The meaning of a set expression is a set of ground terms. Let ρ be an assignment mapping set variables to subsets of T. Then

$$\rho(c(se_1, \ldots, se_{a(c)})) = \{c(t_1, \ldots, t_{a(c)}) | t_i \in \rho(se_i)\}$$

Set constraints are inclusion constraints $se_1 \subseteq se_2$ between set expressions. An assignment ρ is a *solution* of the constraint if $\rho(se_1) \subseteq \rho(se_2)$.

2.2 Regularly Annotated Set Constraints

We extend set constraints as follows. Let Σ be a finite alphabet, and let M be a finite state automaton on Σ. While regular languages and finite automata are equivalent, it is technically more convenient to work with automata. 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.

2.2.1 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 $L(M)$, the language of 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 any substring of a word in $L(M)$.

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 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 $Q \cdot w = \{t \cdot w | t \in Q\}$.

2.2.2 Annotated Set Constraints

A *regularly annotated set constraint* is an inclusion constraint $se_1 \subseteq_w se_2$, where se_1, se_2 are set expressions and $w \in L(M^{sub})$. We normally abbreviate $se_1 \subseteq_\epsilon se_2$ by dropping the annotation $se_1 \subseteq se_2$.

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:

$$se ::= \mathcal{X} \mid c^\alpha(se_1, \ldots, se_{a(c)})$$

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.

$$\rho(c^\alpha(se_1, \ldots, se_{a(c)})) \quad = \quad \{c^w(t_1, \ldots, t_{a(c)}) | w \in \rho(\alpha) \wedge t_i \in \rho(se_i)\}$$

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 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. For automaton M^{sub}, let S be the set of states, $s_0 \in S$ be the start state, and $\delta : \Sigma^* \times S \to S$ be the transition function. We say w and w' are *equivalent*, $w \equiv w'$ if $\delta(w, s_0) = \delta(w', s_0)$. We 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

$$w \in \rho(\alpha) \wedge w \equiv w' \quad \Leftrightarrow \quad w' \in \rho(\alpha)$$

$$t \in \rho(X) \wedge t \equiv t' \quad \Leftrightarrow \quad t' \in \rho(X)$$

Lemma 2.1. 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 $xw \equiv yw$ because $\delta(xw, s_0)$ must be defined (again, because $t \cdot w \in T^{M^{sub}}$) and δ is a function. The last step is another application of the definition of $\cdot$.

□

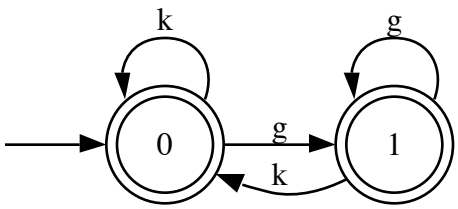

Figure 1. Finite state automaton M_{1bit} for the single fact bit-vector language. M_{1bit}^{sub} is the same automaton.

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

Proof. Consider any constraint $se_1 \subseteq_w se_2$ and term $t \in \rho'(se_1)$; the goal is to show $t \cdot w \in \rho'(se_2)$.

$$\begin{array}{rcl} t \in \rho'(se_1) & \Rightarrow & \\ t' \in \rho(se_1) & \Rightarrow & \text{for some } t' \equiv t \\ t' \cdot w \in \rho(se_2) & \Rightarrow & \\ t' \cdot w \in \rho'(se_2) & \Rightarrow & \\ t \cdot w \in \rho'(se_2) & & \end{array}$$

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 $\rho(se_1)$ 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 2.1, using the fact that $t' \equiv t$ from the first step and the fact that $t' \cdot w \in \rho(se_2)$ 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 constraints the regular completion is also a solution, it suffices to compute only the regular solutions. Thus Theorem 2.2 suggests the following alternate characterization of the word variables on constructors: Since the only interesting sets of words are full equivalence classes corresponding to the states of M^{sub}, we can treat word variables as state variables and map them to sets of states of M^{sub}. This shift of perspective has the important advantage that we can now deal with finite sets of states instead of potentially infinite sets of words. A mapping $\rho(\alpha) = \{s_1, s_2, \ldots\}$ corresponds to the regular solution $\rho(\alpha) = \{w | \delta(s_0, w) \in \{s_1, s_2, \ldots\}\}$.

From here on we will refer to these sets of states, not sets of words, in constructor annotations; we use the term *state variables* instead of word variables for clarity. Our algorithm infers the sets of states 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 1. It happens that for M_{1bit} the machine M_{1bit}^{sub} is exactly the same.

Example 1. *Consider the following constraint system:*

$$\begin{array}{rcl} c^{\alpha} & \subseteq_g & \mathcal{W} \\ o^{\beta}(\mathcal{W}) & \subseteq_g & \mathcal{X} \\ \mathcal{X} & \subseteq_{\epsilon} & o^{\gamma}(\mathcal{Y}) \\ o^{\gamma}(\mathcal{Y}) & \subseteq_{\epsilon} & \mathcal{Z} \end{array}$$

We have annotated each constructor with a state variable. Map the state variables as follows, where the states are given by the machine M_{1bit}^{sub} accepting substrings of $L(M_{1bit})$, also given in Figure 1: $\alpha, \beta = \{0\}$, $\gamma = \{1\}$. Set variables are mapped as follows: $\mathcal{Y}, \mathcal{W} = \{c^1\}, \mathcal{X} = \{o^1(c^1)\}$, and $\mathcal{Z} = \{o^1(c^1)\}$. It is easy to see that this assignment is a solution of the constraints.

2.3 Constraint Solving

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 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 $c^s \in X$ for some annotated term c^s and set variable X?

2.3.1 Resolution Rules

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

$$\begin{array}{rcl} c^{\alpha}(se_1, \ldots, se_{a(c)}) \subseteq_w c^{\beta}(se'_1, \ldots, se'_{a(c)}) & \Rightarrow & \\ \bigwedge_i se_i \subseteq_w se'_i \wedge \delta(w, \alpha) \subseteq \beta & & \\ & & \\ c^{\alpha}(\ldots) \subseteq_w d^{\beta}(\ldots) & \Rightarrow & \\ & \text{no solution} & \end{array}$$

The first rule propagates inclusions between constructed terms to inclusions on the components. Recall that we require that all constraints are annotated with words in $L(M^{sub})$. Because this rule does not generate any new annotations, it preserves this invariant. The other part of the first rule produces *state constraints* between the state variables annotating constructor expressions: the possible state annotations β on the right-hand side constructor expression are constrained to contain at least $\delta(w, \alpha)$, the set of automaton states reachable from states in α on word w (we define $\delta(w, Q)$, where Q is a set of states, to be $\{\delta(w, s) | s \in Q\}$). Because states are just constants (zero-ary constructors) and the transition function δ is known and fixed for a given application, these state 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 annotations together. To ensure termination, we must bound the maximum length of an annotation, and here we make essential use of the finite state automaton M^{sub}. 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. Thus, we can use the pumping length of the automaton as a bound on word length. Whenever the concatenation of two words $w \cdot w'$ exceeds p, the pumping length of M^{sub}, there must exist a word w'' whose length is at most p, so that a run of M^{sub} on w'' ends in the same state as a run on $w \cdot w'$. Since we are checking properties against M^{sub}, we can ignore any constraint path not in $L(M^{sub})$, which also preserves the invariant that all annotations are drawn

from $L(M^{sub})$. The transitive closure rule that reflects these observations is:

$$se_1 \subseteq_w \mathcal{X} \subseteq_{w'} se_2 \Rightarrow se_1 \subseteq_{pump(w \cdot w')} se_2 \text{ if } w \cdot w' \in L(M^{sub})$$

The operation $pump(w)$ picks a word of length at most the pumping length of M^{sub}, so that a run of M^{sub} on w ends in the same state as a run of $pump(w)$ on M^{sub}. Algorithmically, the $pump$ operation is implemented by running the automaton on the word and pruning out any letters occurring on cyclic paths in the run (we describe a more efficient implementation in Section 2.5). The membership check $w \cdot w' \in L(M^{sub})$ guarantees that we only consider constraint paths that the automaton M might eventually accept; it serves the same purpose as the more specialized $match$ operation in [16].

Returning to Example 1, the solved form of this system is:

$$\begin{array}{rcl} c^\alpha & \subseteq_g & \mathcal{W} \\ o^\beta(\mathcal{W}) & \subseteq_g & \mathcal{X} \\ \mathcal{X} & \subseteq & o^\gamma(\mathcal{Y}) \\ o^\gamma(\mathcal{Y}) & \subseteq & \mathcal{Z} \\ o^\beta(\mathcal{W}) & \subseteq_g & o^\gamma(\mathcal{Y}) \\ \mathcal{W} & \subseteq_g & \mathcal{Y} \\ c^\alpha & \subseteq_g & \mathcal{Y} \\ \\ \delta(g, \beta) & \subseteq & \gamma \end{array}$$

Notice that the transitive constraints $c^\alpha \subseteq_g \mathcal{W} \subseteq_g \mathcal{Y}$ result in the constraint $c^\alpha \subseteq_g \mathcal{Y}$ because $pump(gg) = g$ for the machine in Figure 1.

Lemma 2.3. Constraint resolution applying the transitive closure and constructor rules terminates and preserves all solutions of the constraints.

Proof. (Sketch) The interesting case is the transitive closure rule. The number of possible constraints is a function of the maximum annotation length and the number of set expressions. As the resolution rules do not create any new set expressions and the length of the longest annotation is bounded by the $pump$ operation, the total number of possible constraints is also bounded. To prove that all solutions are preserved, we use the fact that the $pump(w \cdot w') \equiv w \cdot w'$ to show that every regular solution of the constraints is preserved by adding the transitive constraint. □

2.3.2 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 $L(M)$. Note that for queries we are interested in annotations in our original language $L(M)$, not $L(M^{sub})$.

More precisely, we say that a system of constraints C_1 entails a system of constraints C_2, written $C_1 \models C_2$, if every solution of C_1 is a solution of C_2. Let C be the solved system of constraints. The formalization of the simple query is:

$$C \wedge s_0 \subseteq \alpha \wedge s_0 \subseteq \beta \ldots \models t \subseteq_w \mathcal{X}$$

where s_0 is the start state of M^{sub}, $\alpha, \beta, \ldots$ are the state variables appearing t, and $w \in L(M)$. We can now explain a number of important aspects of querying annotated set constraints:

- Because all of our constructors are monotonic, constraint systems have least solutions and to check the entailment it suffices to check that $t \subseteq_w \mathcal{X}$ holds in the least solution of $C \wedge s_0 \subseteq \alpha \wedge s_0 \subseteq \beta \ldots$.[1] Without the state constraints $s_0 \subseteq \alpha \wedge \ldots$, the least solution of the constraints would assign the empty set to every state variable, as the constraints generated by the constructor rule (recall Section 2.3.1) do not require any state 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 state variables to include particular states 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 [8]; queries in this case are particularly easy to solve. For example, for a constant c^α, the entailment
$$C \wedge s_0 \subseteq \alpha \models c^\alpha \subseteq_w \mathcal{X}$$
holds if and only if the constraint $c^\alpha \subseteq_w \mathcal{X}$ is present in the solved form system C. Our implementation is based on a strategy that is not completely eager; it does not compute some transitive constraints. This design saves a great deal of space and time in solving by using a sparser representation, but also requires a little more work to answer queries [5].
- Demand driven solvers essentially move all of the work of resolution to queries [9]. As another optimization, our implementation solves automata state constraints on demand. Our solver actually does not generate state constraints or annotate constructors at all during resolution and this has important performance advantages (see Section 5). For our queries, the automata state 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 1, let C_1 be the solved form of the constraints. The query

$$C_1 \wedge 0 \subseteq \alpha \wedge 0 \subseteq \beta \models o^\beta(c^\alpha) \subseteq_g \mathcal{Z}$$

is true. The least solution of $C_1 \wedge 0 \subseteq \alpha \wedge 0 \subseteq \beta$ is the assignment given in Example 1.

We can now explain in more detail why we solve constraints over $T^{M^{sub}}$ instead of T^M. Because the solving phase does not know the queries, it cannot know which constructors are expected to be in which states. 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.

2.4 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 [10]. For an analysis that tracks n facts, we pick an alphabet Σ partitioned into two sets $G = \{g_1, \ldots, g_n\}$ and $K =$

[1] 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.

$\{k_1, \ldots, k_n\}$ (`gens` and `kills`, respectively). The idea is that a `gen` followed by a matching `kill`, as in the word $g_i k_i$, cancel, and that `gens` and `kills` are idempotent. Figure 1 shows the finite state automaton M_{1bit} for a single dataflow fact. For this language, the pumping length is 1: in the automaton M^{sub}_{1bit}, $pump(gg) = g$, $pump(gk) = \epsilon$, etc. Thus, we do not need to keep track of arbitrary sequences of `gens` and `kills` in constraint resolution; it suffices to track just the state of M^{sub}_{1bit}. An n-bit language can be derived from a product construction.

2.5 Complexity

We sketch a generic complexity argument for the constraint resolution algorithm described Section 2.3. 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 $se \subseteq \mathcal{X}$ (the argument for upper bounds is the same). In an annotated constraint system there may be many lower bounds between se and $\mathcal{X}$, one for each distinct word w that can annotate the constraints, so the problem is to bound the number of distinct word annotations $se \subseteq_w \mathcal{X}$. Let s be the number states in automaton M^{sub}. We claim that there are at most $O(s)$ constraints of the form $se \subseteq_w \mathcal{X}$ in a solved system. To see this, note that the $pump$ operation equates words that lead to the same state in M^{sub}. Thus, it suffices to annotate constraints with states instead of words, and there are only s distinct states. In the n-bit language, for instance, this approach automatically exploits order independence of distinct bits: If 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 $n \cdot s$ lower bounds and $n \cdot s$ upper bounds. With states as the annotations on constraints, new annotations (the $pump$ 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 $O(n^2 s^2)$ work. The total complexity is therefore $O(n^3 s^2)$. Note that this is a generic argument that can usually be sharpened for the constraints generated by a particular application.

3. 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 [19]; as mentioned in Section 1, 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 3.6), 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 reduction strategy described in previous work [12] 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 Section 6).

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

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

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

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

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

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

Figure 2. Type rules for polymorphic recursive system

3.1 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:

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

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

3.2 Type Rules and Constraint Generation

Figure 2 shows the type system for the polymorphic recursive analysis. The rules for variables, pairs, and pair projection are

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

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

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

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

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

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

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

Figure 3. Constraint generation for polymorphic recursive system

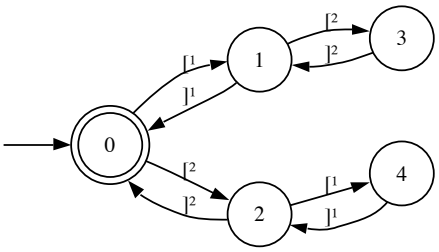

Figure 4. Finite state automaton for single level pairs

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

Figure 5. Non-structural subtyping example

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 3. 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. The rules in Figure 3 use one new form of set expression, a *projection* expression, which we have omitted until now for simplicity. For a constructor of arity n there are n projection operators, one for each argument position. The important property of projections is given by the following equation:

$$c^{-i}(c^{\alpha}(se_1, \ldots, se_i, \ldots, se_n)) = se_i$$

Projections in set constraints are well understood and easily incorporated into the algorithms in Section 2.

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.

3.2.1 Function Call Matching with Terms

As mentioned previously, we apply a result from [12] to model the matching of function calls and returns. Briefly, we create a unary constructor o_i for each function instantiation site i, and add a constraint $o_i(\mathcal{W}) \subseteq \mathcal{X}$ to model the flow from an actual parameter labeled with $\mathcal{W}$ to a formal parameter labeled with $\mathcal{X}$. Flow from the function's return label (say $\mathcal{Y}$) to the label at the function's return site (say $\mathcal{Z}$) is modeled by a constraint $o_i^{-1}(\mathcal{Y}) \subseteq \mathcal{Z}$. The result in [18] shows that this is equivalent to polymorphic recursive treatment of functions.

[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 only for brevity.

3.2.2 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. This is the 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 4 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.

3.3 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 [12]).

3.4 An Example

Consider the program in Figure 5 (taken from [6]). Non-structural subtyping can assign `pair` the type $int^{\mathcal{Y}} \to \beta^{\mathcal{H}}$ along with the constraint $\beta = int^{\mathcal{A}} \times^{\mathcal{P}} int^{\mathcal{Z}}$. Figure 6 shows a slightly simplified constraint graph for this program. Flow from $\mathcal{B}$ to $\mathcal{V}$ is captured by the constraints:

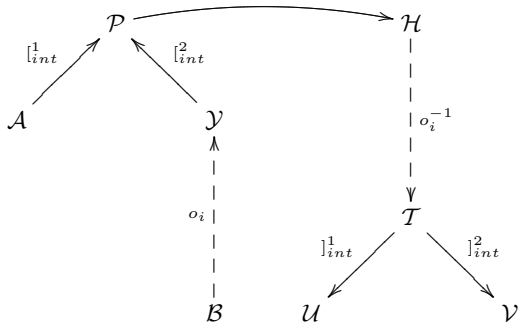

Figure 6. Constraint graph for the program in Figure 5 (only relevant edges are shown)

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

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

3.5 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:

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

Intersecting the solutions for X and Y reveals that there are no common 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.

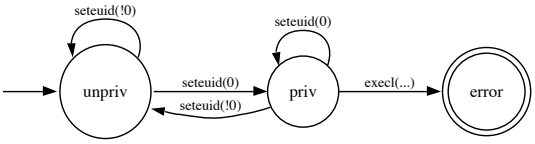

Figure 7. Automaton for process privilege.

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.

3.6 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 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 [12]. With this approach, the constraint system for the example program in Figure 5 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 $\mathcal{B} \subseteq \mathcal{V}$.

4. 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 [4].

Following the approach of [4], 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 intersect the languages $L(M)$ 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 7 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.

4.1 Modeling Programs with Constraints

We now show how to find violations of temporal safety properties using annotated constraints. 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 7). 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 $\mathcal{F}_{entry}$ ($\mathcal{F}_{exit}$) 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.

4.2 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). 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.

4.3 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 constraints for this example are as follows:

$$
\begin{gathered}
pc^{unpriv} \subseteq \mathcal{S}_1 \\
\mathcal{S}_1 \subseteq_{seteuid(0)} \mathcal{S}_2 \\
\mathcal{S}_2 \subseteq \mathcal{S}_3 \\
\mathcal{S}_2 \subseteq \mathcal{S}_4 \\
\mathcal{S}_3 \subseteq_{seteuid(!0)} \mathcal{S}_5 \\
\mathcal{S}_4 \subseteq \mathcal{S}_5 \\
\mathcal{S}_5 \subseteq_{execl(\ldots)} \mathcal{S}_6
\end{gathered}
$$

The constraints imply that pc^{priv} is in the least solution of $\mathcal{S}_4$, the constraint variable corresponding to the `execl` call, indicating the presence of a possible security vulnerability.

5. Implementation

We have added a preliminary implementation of regularly annotated set constraints to the BANSHEE toolkit [13] by adding support for annotations to BANSHEE's existing implementation of set constraints. Many of the technical details (such as handling projection merging [24] and cycle elimination [5]) are similar to those addressed in [22]; we omit them here. BANSHEE also uses *projection patterns* in place of pure projections (following the approach in [12]); the expressive power is the same but projection patterns are better suited to an implementation.

The $pump$ function is currently implemented by hand for each finite state automaton. Automatically producing the $pump$ function is straightforward, and we don't believe that our manual efforts would improve the scalability over an automatically generated implementation. 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.

Our implementation omits state variables on set expressions completely during constraint solving and instead does all of the calculations involving machine states during queries (recall Section 2.3.2). By omitting state 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 word operations in the transitive closure rule is minimal (recall Section 2.5).

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 5 from [3]) and checked several sensitive software packages for security violations using the approach outlined in Section 4. We also simultaneously checked for violations of the simple process privilege property described in Section 4. The additional property we checked is as follows:

1. A program should call `umask(077)` before calls to `mkstemp`
2. Never call the functions `tmpnam`, `tempnam`, `tmpfile`

Programs that violate condition 1 are vulnerable to race condition attacks, and violations of condition 2 allow all users to read temporary files, which may not be secure.[3]

We report in Table 1 the number of lines of code for each package (using David Wheeler's `sloccount` tool [26]), the number of executables for each package, and the time to check the property for all executables in the package. Each executable in a package is checked separately; note that the `OpenSSL` package, though it has fewer lines of code than the `Bind` package, has the largest single program that we check. The experiment was performed on a 2.80

[3] [3] also checks to see if the parameter to `mkstemp` is reused. We do not check this property because we have not implemented so-called *pattern variables*, which allow syntactic matching of program constructs.

Benchmark	KLOC	Programs	Time (s)
At 3.1.8-33	3	2	.15
OpenSSH 3.5p1-6	45	1	1.24
Postfix 2.2.5	71	40	17.2
Bind 9.2.2	203	30	50.1
OpenSSL 0.9.6e	120	1	216

Table 1. Benchmark data for experiment

GHz Intel Xeon machine with 4 Gb of memory. Our analysis times show that our algorithm's scalability and performance is very good, and even this preliminary implementation is usable for realistic applications. Our times are also faster than those reported for MOPS, in most cases by more than an order of magnitude.[4] This comparison of wall clock times is not fair (the MOPS experiments were done on a 1.5 GHz machine, and we have not exactly replicated the experiments) but does show that our general implementation of annotated set constraints is at least competitive with hand-written versions.

6. Related Work

Regularly annotated set constraints are partly inspired by the annotated inclusion constraints presented in [22, 16]. 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 [16] 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 [15]. 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.

The combination of polymorphic recursion and non-structural subtyping that we consider in Section 3 was first considered in [6]. The solution proposed in [6] 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 [7]. 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.

Weighted pushdown systems (WPDS) label transitions with values from a domain of weights [21]. 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 utilized as an alternative to graph reachability for program analysis applications [25, 14]. As in our approach, analyses are specified with a high level language. BDDs are used as a back-end to compactly represent what would otherwise be intractably large relations. One disadvantage is that BDD-based toolkits 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 and at least to date have not been integrated with context-free properties. As we have shown here, a class of reachability problems more general than those handled by current BDD-based methods can be solved in polynomial time.

Other analyses that demand more expensive algorithms, e.g. path sensitive analyses, cannot be expressed with the polynomial time algorithms we present here.

[4] The comparison is even more favorable when we consider that MOPS compacts its input control flow graphs to eliminate property-irrelevant program statements, reducing CFGs by orders of magnitude. We perform no such compaction.

7. Conclusion

In this paper, we have described a new formalism that extends set constraints with annotations drawn from a regular language. We have shown how to express applications as diverse as type-based flow analysis, interprocedural dataflow analysis, and pushdown model checking within this formalism. We have implemented an algorithm for solving regularly annotated constraints within BANSHEE, our constraint-based program analysis toolkit. Preliminary experiments with our implementation suggest that it scales well with good performance.

References

[1] 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.

[2] R. 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.

[3] H. Chen, D. Dean, and D. 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, Feb. 4–6, 2004.

[4] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, New York, NY, USA, 2002. ACM Press.

[5] M. Fähndrich, J. S. Foster, Z. Su, and A. 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.

[6] 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.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In J. 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.

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

[9] N. Heintze and O. 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.

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

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

[12] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 207–218. ACM Press, 2004.

[13] J. Kodumal and A. Aiken. Banshee: A scalable constraint-based analysis toolkit. In *SAS '05: Proceedings of the 12th International Static Analysis Symposium*, pages 218–234. London, United Kingdom, September 2005.

[14] O. Lhoták and L. 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.

[15] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

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

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

[18] J. Rehof and M. 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.

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

[20] T. Reps, S. Horwitz, and M. 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.

[21] 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.

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

[23] M. Sridharan, D. Gopan, L. Shan, and R. Bodik. Demand-driven points-to analysis for java. In *Proceedings of the twentieth conference on object-oriented programs, systems, languages, and applications*, 2005.

[24] Z. Su, M. Fähndrich, and A. 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.

[25] J. Whaley and M. 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.

[26] D. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount, 2005.