Componential Set-Based Analysis

Cormac Flanagan
cormac@cs.rice.edu

Matthias Felleisen
matthias@cs.rice.edu

Rice University*

Abstract

Set-based analysis is a constraint-based whole program analysis that is applicable to functional and object-oriented programming languages. Unfortunately, the analysis is useless for large programs, since it generates descriptions of data flow relationships that grow quadratically in the size of the program.

This paper presents componential set-based analysis, which is faster and handles larger programs without any loss of accuracy over set-based analysis. The design of the analysis exploits a number of theoretical results concerning constraint systems, including a completeness result and a decision algorithm concerning the observable equivalence of constraint systems. Experimental results validate the practicality of the analysis.

1 The Effectiveness of Set-Based Analysis

Rice's Scheme program development environment provides a static debugger, MrSpidey, which analyzes a program and, using the results of this analysis, checks the soundness of all computational primitives [9]. If a primitive operation may fault due to a violation of its invariant, MrSpidey highlights the program operation so that the programmer can investigate the potential fault site before running the program. Using the graphical explanation facilities of MrSpidey, the programmer can determine whether this fault will really happen or whether the corresponding correctness proof is beyond the analysis's capabilities.

MrSpidey's program analysis is a constraint-based system similar to Heintze's set-based analysis [11]. The analysis consists of two co-mingled phases: a *derivation* phase, during which MrSpidey derives constraints describing the data flow relationships of the analyzed program, and a *solution* phase, during which MrSpidey solves the constraints. The solution conservatively approximates the set of values that may be returned by each program expression.

In practice, MrSpidey has proven highly effective for pedagogic programming, which includes programs of several hundred to a couple of thousand lines of code. It becomes less useful, however, for debugging larger programs due to limitations in the underlying analysis, which has an $O(n^3)$ worst-case time bound. The constant on the cubic element is small, but it becomes dominant for programs of several thousand lines.

The bottleneck is due to the excessive size of the constraint systems that describe a program's data flow relationships. If we could simplify these constraint systems without affecting the data flow relationships that they denote, then we could reduce the analysis times. That is, by first simplifying the constraint system for each program component (*e.g.* module or package), we could solve the combined system of constraints in less time. Furthermore, if we saved each simplified constraint system in a *constraint file*, then we could exploit those saved constraints in future runs of the analysis to avoid reprocessing components that have not changed.

The simplification of constraint systems raises both interesting theoretical and practical questions. On the theoretical side, we need to ensure that simplification preserves the observable behavior of a constraint system. In this paper, we provide a complete characterization of observable behavior and, in the course of this development, establish a close connection between this observable equivalence of constraint systems and the equivalence of regular tree grammars (RTGs).[1] Exploiting this connection, we develop a complete algorithm for deciding the equivalence of constraint systems. Unfortunately, the algorithm is PSPACE-hard.

Fortunately, a minimized constraint system is only optimal but not necessary for practical purposes. The

*This work was partially supported by NSF grants CCR-9633746 and CCR-9619756, and a Lodieska Stockbridge Vaughan Fellowship.

[1] A number of researchers, including Reynolds [18], Jones and Muchnick [14], Heintze [11], Aiken [2], and Cousot and Cousot [3] previously exploited the relationship between RTGs and the *least solution* of a constraint system. We present an additional result, namely a connection between RTGs and the observable behavior (*i.e.*, the *entire solution space*) of constraint systems.

practical question concerns finding approximate algorithms for simplifying constraint systems that would make MrSpidey more useful. To answer this question, we exploit the correspondence between the minimization problems for RTGs and constraint systems to adapt a variety of algorithms for simplifying RTGs to the problem of simplifying constraint systems. Based on these simplification algorithms, we develop a *componential*,[2] or component-wise, variant of set-based analysis. Experimental results verify the effectiveness of the simplification algorithms and the corresponding flavors of the analysis. The simplified constraint systems are typically at least an order of magnitude smaller than the original systems, and these reductions in size result in significant gains in the speed of the analysis.

We expect that some of our theoretical and practical results as well as the techniques will carry over to other constraint-based systems, such as the conditional type system of Aiken *et al.* [2], Eifrig *et al.*'s object-oriented type system [5], or Pottier's or Smith *et al.*'s subtyping simplification algorithms [17, 21].

The presentation proceeds as follows. Section 2 describes an idealized source language. Sections 3 and 4 present the theoretical underpinnings of the new analysis. Section 5 introduces practical constraint simplification algorithms and Sections 6 and 7 discuss how these algorithms perform in a realistic program analysis system. Section 8 discusses related work, and Section 9 describes directions for future research.

2 The Source Language

For simplicity, we derive our analysis for a λ-calculus-like language with constants and labeled expressions. It is straightforward to extend the analysis to a realistic language including assignments, recursive data structures, objects and modules along the lines described in an earlier report [7].

Expressions in the language are either variables, values, function applications, **let**-expressions, or labeled expressions: see figure 1. We use labels to identify those program expressions whose values we wish to predict. Values include basic constants and functions. Functions have identifying tags so that MrSpidey can reconstruct a call-graph from the results of the analysis. We use **let**-expressions to introduce polymorphic bindings, and hence restrict these bindings to syntactic values [23]. We work with the usual conventions and terminology of the λ_v-calculus when discussing syntactic issues. In particular, the substitution operation $M[x \leftarrow V]$ replaces all free occurrences of x within M by V, and Λ^0 denotes the set of closed terms, also called *programs*.

[2]**componential** *a.* of or pertaining to components; *spec.* (*Ling.*) designating the analysis of distinctive sound units or grammatical elements into phonetic or semantics components (*New Shorter Oxford English Dictionary*, Clarendon Press, 1993)

Syntax:

$$\begin{array}{lclcll} M & \in & \Lambda & = & x \mid V \mid (M\ M) \mid M^l & \text{(Expressions)} \\ & & & & \mid (\mathbf{let}\ (x\ M)\ M) & \\ V & \in & \mathit{Value} & = & b \mid (\lambda^t x.M) & \text{(Values)} \\ x & \in & \mathit{Vars} & = & \{x, y, z, \ldots\} & \text{(Variables)} \\ b & \in & \mathit{BConst} & & & \text{(Basic constants)} \\ t & \in & \mathit{Tag} & & & \text{(Function tags)} \\ l & \in & \mathit{Label} & & & \text{(Expression labels)} \end{array}$$

Evaluator:

$$\begin{array}{rcl} \mathit{eval} : \Lambda^0 & \longrightarrow & \mathit{Value} \cup \{\bot\} \\ \mathit{eval}(M) & = & V \quad \text{if } M \longmapsto^* V \end{array}$$

Reduction Rules:

$$\begin{array}{rcll} \mathcal{E}[\ ((\lambda^t x.M)\ V)\ ] & \longrightarrow & \mathcal{E}[\ M[x \mapsto V]\ ] & (\beta_v) \\ \mathcal{E}[\ (\mathbf{let}\ (x\ V)\ M)\ ] & \longrightarrow & \mathcal{E}[\ M[x \mapsto V]\ ] & (\beta_{let}) \\ \mathcal{E}[\ V^l\ ] & \longrightarrow & \mathcal{E}[\ V\ ] & (\mathit{unlabel}) \end{array}$$

Evaluation Contexts:

$$\mathcal{E} = [\ ] \mid (\mathcal{E}\ M) \mid (V\ \mathcal{E}) \mid (\mathbf{let}\ (x\ \mathcal{E})\ M) \mid \mathcal{E}^l$$

Figure 1: The source language Λ: syntax and semantics

We specify the meaning of programs via the reduction semantics based on the rules described in figure 1. The reduction rules β_v and β_{let} are conventional, and the *unlabel* rule removes the label from an expression once its value is needed.

3 Set-Based Analysis

Conceptually, set-based analysis consists of two phases: a *specification* phase and a *solution* phase.[3] During the specification phase, the analysis tool derives *constraints* on the sets of values that program expressions may assume. These constraints describe the data flow relationships of the analyzed program. During the solution phase, the analysis produces finite descriptions of the potentially infinite sets of values that satisfy these constraints. The result provides an approximate set of values for each labeled expression in the program.

3.1 The Constraint Language

To simplify the derivation of the constraint simplification algorithms, we formulate our constraint language in terms of type selectors, instead of the more usual

[3]Cousot and Cousot showed that set-based analysis can alternatively be formulated as an abstract interpretation computed by chaotic iteration [3].

type constructors:

$$\begin{array}{rcrcl} \tau & \in & \mathit{SetExp} & = & \alpha \mid c \mid \mathtt{dom}(\tau) \mid \mathtt{rng}(\tau) \\ \alpha,\beta,\gamma & \in & \mathit{SetVar} & \supset & \mathit{Label} \\ c & \in & \mathit{Const} & = & \mathit{BConst} \cup \mathit{Tag} \end{array}$$

A *set expression* τ is either a set variable; a constant; or one of the "selector" expressions $\mathtt{dom}(\tau)$ or $\mathtt{rng}(\tau)$. By using selector expressions, we can specify each "quantum" of the program's data flow behavior independently; using constructors would combine several of these quanta into one constraint. The meta-variables α, β, γ range over set variables, and we include program labels in the collection of set variables. Constants include both basic constants and function tags. A *constraint* $\mathcal{C}$ is an inequality $\tau_1 \leq \tau_2$ relating two set expressions.

Intuitively, each set expression denotes a set of run-time values, and a constraint $[\tau_1 \leq \tau_2]$ indicates that the value set denoted by τ_1 is contained in the value set denoted by τ_2. A *constraint system* $\mathcal{S}$ is a collection of constraints. A *simple constraint system* is a collection of *simple constraints*, which have the form:

$$\begin{array}{rcl} c \leq \beta \mid & \alpha \leq \beta & \mid \alpha \leq \mathtt{dom}(\beta) \\ \mid \mathtt{rng}(\alpha) \leq \beta \mid & \mathtt{dom}(\alpha) \leq \beta & \mid \alpha \leq \mathtt{rng}(\beta) \end{array}$$

In some cases, we are interested in constraints that only mention certain set variables. The *restriction* of a constraint system to a collection of set variables E is:

$$\mathcal{S}\mid_E \;=\; \{\mathcal{C} \in \mathcal{S} \mid \mathcal{C} \text{ only mentions set variables in } E\}$$

3.2 Semantics of Constraints

A set expression denotes a collection of values, which is represented as a triple $X = \langle C, D, R\rangle$. The first component $C \in \mathcal{P}(\mathit{Const})$[4] is a set of basic constants and function tags, and represents a set of run-time values (relative to a given program) according to the relation $V\ \mathit{in}\ C$:

$$\begin{array}{rcl} b\ \mathit{in}\ C & \text{iff} & b \in C \\ (\lambda^t x.M)\ \mathit{in}\ C & \text{iff} & t \in C \end{array}$$

The second and third components of X denote the possible argument values (*dom*) and result values (*rng*) of functions in X, respectively. Since these two components also denote value sets, the appropriate model for set expressions is the solution of the equation:[5]

$$\mathcal{D} = \mathcal{P}(\mathit{Const}) \times \mathcal{D} \times \mathcal{D}$$

[4] $\mathcal{P}$ denotes the power-set constructor.

[5] The set $\mathcal{D}$ is equivalent to the set of all infinite binary trees with each node labeled with an element of $\mathcal{P}(\mathit{Const})$. This set can be formally defined as the set of total functions $f : \{\mathtt{dom},\mathtt{rng}\}^* \longrightarrow \mathcal{P}(\mathit{Const})$, and the rest of the development can be adapted *mutandis mutatis* [16]. For clarity, we present our results using the more intuitive notation instead.

We use the functions $\mathit{const} : \mathcal{D} \longrightarrow \mathcal{P}(\mathit{Const})$ and dom, $\mathit{rng} : \mathcal{D} \longrightarrow \mathcal{D}$ to extract the respective components of an element of $\mathcal{D}$.

We order the elements of $\mathcal{D}$ according to a relation that is contravariant in the argument component, since the information about argument values at an application needs to flow *backward* along data-flow paths to the formal parameter of the corresponding function definitions. Thus $\langle C_1, D_1, R_1\rangle \sqsubseteq \langle C_2, D_2, R_2\rangle$ if and only if $C_1 \subseteq C_2$, $D_2 \sqsubseteq D_1$, and $R_1 \sqsubseteq R_2$. The set $\mathcal{D}$ forms a complete lattice under this ordering, with top and bottom elements being the solutions to the equations $\top = \langle \mathit{Const}, \bot, \top\rangle$ and $\bot = \langle \emptyset, \top, \bot\rangle$, respectively.

The semantics of set expressions is defined with respect to a *set environment* ρ, which maps each set variable to an element of $\mathcal{D}$. We extend the domain of set environments from set variables to set expressions in the natural manner:

$$\begin{array}{rcl} \rho : \mathit{SetExp} & \longrightarrow & \mathcal{D} \\ \rho(c) & = & \langle \{c\}, \top, \bot\rangle \\ \rho(\mathtt{dom}(\tau)) & = & \mathit{dom}\,(\rho(\tau)) \\ \rho(\mathtt{rng}(\tau)) & = & \mathit{rng}\,(\rho(\tau)) \end{array}$$

An environment ρ *satisfies* a constraint $\mathcal{C} = [\tau_1 \leq \tau_2]$ (written $\rho \models \mathcal{C}$) if $\rho(\tau_1) \sqsubseteq \rho(\tau_2)$. Similarly, ρ satisfies $\mathcal{S}$, or ρ is a solution of $\mathcal{S}$ (written $\rho \models \mathcal{S}$) if $\rho \models \mathcal{C}$ for each $\mathcal{C} \in \mathcal{S}$. The *solution space* of a constraint system $\mathcal{S}$ is $\mathit{Soln}(\mathcal{S}) = \{\rho \mid \rho \models \mathcal{S}\}$. A constraints set $\mathcal{S}_1$ *entails* $\mathcal{S}_2$ (written $\mathcal{S}_1 \models \mathcal{S}_2$) iff $\mathit{Soln}(\mathcal{S}_1) \subseteq \mathit{Soln}(\mathcal{S}_2)$, and $\mathcal{S}_1$ is *observably equivalent to* $\mathcal{S}_2$ (written $\mathcal{S}_1 \cong \mathcal{S}_2$) iff $\mathcal{S}_1 \models \mathcal{S}_2$ and $\mathcal{S}_2 \models \mathcal{S}_1$.

The *restriction* of a solution space to a collection of variables E is:

$$\mathit{Soln}(\mathcal{S})\mid_E = \{\rho \mid \exists \rho' \in \mathit{Soln}(\mathcal{S}).\, \forall \alpha \in E.\, \rho(\alpha) = \rho'(\alpha)\}$$

We extend the notion of restriction to entailment and observable equivalence of constraint systems:

- If $\mathit{Soln}(\mathcal{S}_1)\mid_E \subseteq \mathit{Soln}(\mathcal{S}_2)\mid_E$, then $\mathcal{S}_1$ *entails* $\mathcal{S}_2$ *with respect to* E (written $\mathcal{S}_1 \models_E \mathcal{S}_2$).
- If $\mathcal{S}_1 \models_E \mathcal{S}_2$ and $\mathcal{S}_2 \models_E \mathcal{S}_1$ then that $\mathcal{S}_1$ and $\mathcal{S}_2$ are *observably equivalent with respect to* E (written $\mathcal{S}_1 \cong_E \mathcal{S}_2$) .

3.3 Deriving Constraints

The specification phase of set-based analysis derives constraints on the sets of values that program expressions may assume. Following Aiken *et al.* and Palsberg and O'Keefe, we formulate this derivation as a subtype system [2, 16].

The derivation proceeds in a syntax-directed manner according to the constraint derivation rules presented

$$\Gamma \cup \{x : \alpha\} \vdash x : \alpha, \emptyset \qquad (var)$$

$$\Gamma \vdash b : \alpha, \{b \leq \alpha\} \qquad (const)$$

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash M^l : \alpha, \mathcal{S} \cup \{\alpha \leq l\}} \qquad (label)$$

$$\frac{\Gamma \cup \{x : \alpha_1\} \vdash M : \alpha_2, \mathcal{S}}{\Gamma \vdash (\lambda^t x.M) : \alpha, \mathcal{S} \cup \{t \leq \alpha,\ \mathtt{dom}(\alpha) \leq \alpha_1,\ \alpha_2 \leq \mathtt{rng}(\alpha)\}} \qquad (abs)$$

$$\frac{\Gamma \vdash M_i : \beta_i, \mathcal{S}_i}{\Gamma \vdash (M_1\ M_2) : \alpha, \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\beta_2 \leq \mathtt{dom}(\beta_1),\ \mathtt{rng}(\beta_1) \leq \alpha\}} \qquad (app)$$

$$\frac{\begin{array}{c}\Gamma \vdash V : \alpha, \mathcal{S}_V \\ A = \mathit{Vars}(\mathcal{S}_V) \setminus (FV[\mathit{rng}(\Gamma)] \cup \mathit{Label}) \\ \Gamma \cup \{x : \forall A. (\alpha, \mathcal{S}_V)\} \vdash M : \beta, \mathcal{S}\end{array}}{\Gamma \vdash (\mathbf{let}\ (x\ V)\ M) : \beta, \mathcal{S}} \qquad (let)$$

$$\frac{\psi \text{ is a substitution of fresh vars for } A}{\Gamma \cup \{x : \forall A. (\alpha, \mathcal{S}_V)\} \vdash x : \psi(\alpha), \psi(\mathcal{S}_V)} \qquad (inst)$$

Figure 2: Constraint derivation rules.

in figure 2. Each rule infers a judgement of the form $\Gamma \vdash M : \alpha, \mathcal{S}$, where the *set variable context* Γ maps the free variables of M either to set variables or constraint schemas (see below); α names the value set of M; and the constraint system $\mathcal{S}$ describes the data flow relationships of M, using α.

The rules (*var*) and (*const*) are straightforward. The rule (*label*) records the value set of a labeled expression in the appropriate label. The rule (*abs*) for functions records the function's tag, and also propagates values from the function's domain into its formal parameter and from the function's body into its range. The rule (*app*) for applications propagates values from the argument expression into the domain of the applied function and from the range of that function into the result of the application expression.

The rule (*let*) produces a *constraint schema* $\sigma = \forall A. (\alpha, \mathcal{S})$ for polymorphic, **let**-bound values [20, 2, 23]. The set variable α names the result of the expression, the constraint system $\mathcal{S}$ describes the data flow relationships of the expression, and the set A contains those internal set variables of the constraint system that must be duplicated at each reference to the **let**-bound variable via the rule (*inst*). We use $FV[\mathit{rng}(\Gamma)]$ to denote the free set variables in the range of Γ. The free set variables of a schema $\sigma = \forall A. (\alpha, \mathcal{S})$ are those in $\mathcal{S}$ but not in A, and the free variables of a set variable is simply the set variable itself.

3.4 Set Based Analysis

Every constraint system admits the trivial solution $\rho^\top$ where $\rho^\top(\alpha) = \top_s$ and $\top_s = \langle \mathit{Const}, \top_s, \top_s \rangle$. Since $\top_s$ represents the set of *all* run-time values, this solution is highly approximate and utterly useless. Fortunately, constraint systems typically yield many additional solutions that more accurately characterize the value sets of program expressions.

For example, consider the program $P = (\lambda^t x.x)$, which yields the constraint system:

$$\{t \leq \alpha_P, \mathtt{dom}(\alpha_P) \leq \alpha_x, \alpha_x \leq \mathtt{rng}(\alpha_P)\}$$

In addition to the trivial solution described above, this constraint system admits a number of other solutions, including:

$$\begin{array}{rcl}\rho_1 & = & \{\alpha_P \mapsto \langle \{t\}, \bot, \bot \rangle, \alpha_x \mapsto \bot\} \\ \rho_2 & = & \{\alpha_P \mapsto \langle \{t\}, \top, \top \rangle, \alpha_x \mapsto \top\}\end{array}$$

The solution ρ_1 more accurately describes the program's run-time value sets than ρ_2. Yet these two solutions are incomparable under the ordering $\sqsubseteq$ (pointwise extended to environments), since it models the flow of values through a program, but does not rank environments according to their accuracy.

Therefore we introduce a second ordering $\sqsubseteq_s$ on $\mathcal{D}$ that properly ranks environments according to their accuracy. This ordering is covariant in the domain position, *i.e.*, $\langle C_1, D_1, R_1 \rangle \sqsubseteq_s \langle C_2, D_2, R_2 \rangle$ if and only if $C_1 \subseteq C_2$, $D_1 \sqsubseteq_s D_2$, and $R_1 \sqsubseteq_s R_2$.

Under this ordering, a constraint system $\mathcal{S}$ has both a maximal solution ($\rho^\top$ above) and a minimal solution. The minimal solution exists because the greatest lower bound $\sqcap_s$ with respect to $\sqsubseteq_s$ of two solutions is also a solution [11]. We use $\mathit{LeastSoln}(\mathcal{S})$ to denote this least solution, and define set-based analysis as the function that extracts the basic constants and function tags for each labeled expression from $\mathit{LeastSoln}(\mathcal{S})$.

Definition 3.1. (*sba*) If $\emptyset \vdash P : \alpha, \mathcal{S}$, then:

$$\mathit{sba}(P)(l) = \mathit{const}(\mathit{LeastSoln}(\mathcal{S})(l))$$

The solution $\mathit{sba}(P)$ conservatively approximates the value sets for each labeled expression.

Theorem 3.2 (Correctness of *sba*) *If* $P \longmapsto^* \mathcal{E}[\ V^l\ ]$ *then* V *in* $\mathit{sba}(P)(l)$.

This result follows from a subject reduction proof along the lines of Wright and Felleisen [22] and Palsberg [15] and is contained in a related report [8].

3.5 Computing the Least Solution

To compute $\mathit{sba}(P)$, we close the constraint system for P under the rules Θ described in figure 3. Intuitively,

$$\frac{c \leq \beta \qquad \beta \leq \gamma}{c \leq \gamma} \qquad (s_1)$$

$$\frac{\alpha \leq \mathtt{rng}(\beta) \qquad \beta \leq \gamma}{\alpha \leq \mathtt{rng}(\gamma)} \qquad (s_2)$$

$$\frac{\mathtt{dom}(\beta) \leq \alpha \qquad \beta \leq \gamma}{\mathtt{dom}(\gamma) \leq \alpha} \qquad (s_3)$$

$$\frac{\alpha \leq \mathtt{rng}(\beta) \qquad \mathtt{rng}(\beta) \leq \gamma}{\alpha \leq \gamma} \qquad (s_4)$$

$$\frac{\alpha \leq \mathtt{dom}(\beta) \qquad \mathtt{dom}(\beta) \leq \gamma}{\alpha \leq \gamma} \qquad (s_5)$$

Figure 3: The inference rule system Θ.

these rules infer all the data flow paths in the program, and propagate values along those paths. Specifically, the rules (s_1), (s_2), and (s_3) propagate information about constants, function domains and function ranges forward along the data flow paths of the program. These data flow paths are described by constraints of the form $\beta \leq \gamma$. The rule (s_4) constructs the data flow paths from actual to formal parameters for each function call, and the rule (s_5) similarly constructs data flow paths from function bodies to corresponding call sites. We write $\mathcal{S} \vdash_\Theta \mathcal{C}$ if $\mathcal{S}$ proves $\mathcal{C}$ via the rules Θ, and use $\Theta(\mathcal{S})$ to denote the closure of $\mathcal{S}$ under Θ, *i.e.*, the set $\{\mathcal{C} \mid \mathcal{S} \vdash_\Theta \mathcal{C}\}$.

MrSpidey uses a worklist algorithm to compute the closure of $\mathcal{S}$ under Θ efficiently. The worklist keeps track of all *eligible* inference rules whose antecedents are in $\mathcal{S}$ but whose consequent may not be in $\mathcal{S}$. The algorithm repeatedly removes an inference rule from the worklist, adds its consequent to $\mathcal{S}$, if necessary, and then adds to the worklist all inference rules that are made eligible by the addition of that consequent. The process iterates until the worklist is empty, at which point $\mathcal{S}$ is closed under Θ. The complete algorithm can be found in an earlier technical report [7].

This closure process propagates all information concerning the possible constants for labeled expressions into constraints of the form $c \leq l$. Hence, we can infer $sba(P)$ from $\Theta(\mathcal{S})$ according to the following theorem.

Theorem 3.3 *If* $P \in \Lambda^0$ *and* $\emptyset \vdash P : \alpha, \mathcal{S}$ *then:*

$$sba(P)(l) = \{c \mid [c \leq l] \in \Theta(\mathcal{S})\}$$

4 Observable Equivalence of Constraints

The traditional set-based analysis we have just described has proven highly effective for programs of up to a couple of thousand lines of code. Unfortunately, it is useless for larger programs due to its nature as a whole program analysis and due to the size of the constraint systems it produces, which are quadratic in the size of (large) programs. Storing these constraint systems in memory is beyond the capabilities of most machines.

To overcome this problem, we develop algorithms for simplifying constraints systems. Applying these simplification algorithms to each program component significantly reduces both the time and space required by the overall analysis.

The following subsection shows that constraint simplification does not affect the analysis results provided the simplified system is *observably equivalent* to the original system. Subsection 4.2 presents a complete proof-theoretic formulation of observable equivalence, and subsection 4.3 exploits this formulation to develop an algorithm for deciding the observable equivalence of constraint systems. The insights provided by this development lead to the practical constraint simplification algorithms of section 5.

4.1 Conditions on Constraint Simplification

Let us consider a program P containing a program component M. Suppose the constraint derivations for M concludes $\Gamma \vdash M : \alpha, \mathcal{S}_1$, where $\mathcal{S}_1$ is the constraint system for M. Our goal is to replace $\mathcal{S}_1$ by a simpler constraint system without changing $sba(P)$.

Since the constraint derivation process is compositional, the constraint derivation for the entire program concludes $\emptyset \vdash P : \beta, \mathcal{S}_C \cup \mathcal{S}_1$, where $\mathcal{S}_C$ is the constraint system for the context surrounding M. The combined constraint system $\mathcal{S}_C \cup \mathcal{S}_1$ describes the space of solutions for the entire program, which is the intersection of the two respective solution spaces:

$$Soln(\mathcal{S}_C \cup \mathcal{S}_1) = Soln(\mathcal{S}_C) \cap Soln(\mathcal{S}_1)$$

and hence $Soln(\mathcal{S}_1)$ describes at least all the properties of $\mathcal{S}_1$ relevant to the analysis. However, $Soln(\mathcal{S}_1)$ may describe solutions for set variables that are not relevant to the analysis of P. In particular,

- $sba(P)$ only references the solutions for labels; and
- the only interactions between $\mathcal{S}_C$ and $\mathcal{S}_1$ are due to the set variables $\{\alpha\} \cup FV[rng(\Gamma)]$.

Thus the only properties of $\mathcal{S}_1$ relevant to the analysis is the solution space for its *external set variables*

$$E = Label \cup \{\alpha\} \cup FV[rng(\Gamma)]$$

For our original problem, this means that we want a constraint system $\mathcal{S}_2$ whose solution space restricted to E is equivalent to that of $\mathcal{S}_1$ restricted to E:

$$Soln(\mathcal{S}_1)\mid_E = Soln(\mathcal{S}_2)\mid_E;$$

or, with the notation from section 3, $\mathcal{S}_1$ and $\mathcal{S}_2$ are observably equivalent on E:

$$\mathcal{S}_1 \cong_E \mathcal{S}_2$$

We can translate this compaction idea into an additional rule for the constraint derivation system:

$$\frac{\begin{array}{c}\Gamma \vdash_{\cong} M : \alpha, \mathcal{S}_1 \\ \mathcal{S}_1 \cong_E \mathcal{S}_2 \text{ where } E = Label \cup FV[rng(\Gamma)] \cup \{\alpha\}\end{array}}{\Gamma \vdash_{\cong} M : \alpha, \mathcal{S}_2} \quad (\cong)$$

This rule is *admissible* in that any derivation (denoted using $\vdash_{\cong}$) in the extended constraint derivation system produces information that is equivalent to the information produced by the original analysis.

Lemma 4.1 *If $\emptyset \vdash_{\cong} P : \alpha, \mathcal{S}$, then:*

$$sba(P)(l) = const(LeastSoln(\mathcal{S})(l))$$

4.2 Proof-Theoretic Characterization

Since the new derivation rule ($\cong$) involves the semantic notion of observably equivalent constraint systems, it cannot be used directly. To make this rule useful, we must first reformulate the observable equivalence relation as a syntactic proof system.

The key properties of the observational equivalence relation are reflections of the properties of the ordering relation ($\sqsubseteq$) and the functions *dom* and *rng*, respectively. We can reify these properties into a syntactic proof system via the following inference rules:

$$\alpha \leq \alpha \quad (reflex) \qquad \frac{\tau_1 \leq \tau \quad \tau \leq \tau_2}{\tau_1 \leq \tau_2} \quad (trans')$$

$$\frac{\kappa_1 \leq \kappa_2}{\begin{array}{c}\mathtt{rng}(\kappa_1) \leq \mathtt{rng}(\kappa_2) \\ \mathtt{dom}(\kappa_2) \leq \mathtt{dom}(\kappa_1)\end{array}} \quad (compat)$$

where we restrict κ to non-constant set expressions to avoid inferring useless tautologies:

$$\kappa ::= \alpha \mid \mathtt{dom}(\kappa) \mid \mathtt{rng}(\kappa)$$

Many of the inferred constraints lie outside of the original language of simple constraints. The extended language of *compound constraints* is:

$$\mathcal{C} \quad ::= \quad c \leq \kappa \mid \kappa \leq \kappa$$

While this proof system obviously captures the properties of $\sqsubseteq$, it does not lend itself to an efficient implementation. Specifically, checking if two potential antecedents of ($trans'$) contain the same set expression τ involves comparing two potentially large set expressions. Hence we use an alternative proof system that can easily be implemented, yet infers the same constraints as the above. The alternative system consists of the inference rules Ψ described in Figure 4, together with the rules Θ from Figure 3. The rules ($compose_{1\ldots4}$) replace a reference to a set variable by an upper or lower (non-constant) bound for that variable, as appropriate. The rule ($trans$) of Ψ provides a weaker characterization of transitivity than the previous rule ($trans'$), but the additional rules compensate for this weakness.

$$\frac{\alpha \leq \mathtt{rng}(\beta) \qquad \beta \leq \kappa}{\alpha \leq \mathtt{rng}(\kappa)} \quad (compose_1)$$

$$\frac{\alpha \leq \mathtt{dom}(\beta) \qquad \beta \geq \kappa}{\alpha \leq \mathtt{dom}(\kappa)} \quad (compose_2)$$

$$\frac{\alpha \geq \mathtt{rng}(\beta) \qquad \beta \geq \kappa}{\alpha \geq \mathtt{rng}(\kappa)} \quad (compose_3)$$

$$\frac{\alpha \geq \mathtt{dom}(\beta) \qquad \beta \leq \kappa}{\alpha \geq \mathtt{dom}(\kappa)} \quad (compose_4)$$

$$\alpha \leq \alpha \quad (reflex)$$

$$\frac{\tau_1 \leq \alpha \qquad \alpha \leq \tau_2}{\tau_1 \leq \tau_2} \quad (trans)$$

$$\frac{\kappa_1 \leq \kappa_2}{\begin{array}{c}\mathtt{rng}(\kappa_1) \leq \mathtt{rng}(\kappa_2) \\ \mathtt{dom}(\kappa_2) \leq \mathtt{dom}(\kappa_1)\end{array}} \quad (compat)$$

Figure 4: The inference rule system Ψ.

The proof system $\Theta \cup \Psi$ is sound and complete in that it infers all true compound constraints.

Lemma 4.2 (Soundness and Completeness of $\Theta\Psi$) *For a simple constraint system $\mathcal{S}$ and compound constraint $\mathcal{C}$, $\mathcal{S} \vdash_{\Psi\Theta} \mathcal{C}$ if and only if $\mathcal{S} \models \mathcal{C}$.*

This lemma implies that $\Psi\Theta(\mathcal{S})$, which denotes the closure of $\mathcal{S}$ with respect to $\Theta \cup \Psi$, contains exactly those (compound) constraints that hold in all environments in $Soln(\mathcal{S})$. For a collection of external set variables E, $\Psi\Theta(\mathcal{S})\mid_E$ contains all (compound) constraints that hold in all environments in $Soln(\mathcal{S})\mid_E$.

Lemma 4.3 *$\mathcal{S} \cong_E \Psi\Theta(\mathcal{S})\mid_E$.*

We could use this result to define a proof-theoretic equivalent of restricted entailment as follows:

$$\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2 \text{ iff } \Psi\Theta(\mathcal{S}_1)\mid_E \supseteq \Psi\Theta(\mathcal{S}_2)\mid_E$$

and then show that $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ if and only if $\mathcal{S}_1 \models_E \mathcal{S}_2$. However, a variant of the above definition yields a relation that is easier to compute. Specifically, suppose $\Psi\Theta(\mathcal{S}_1)\mid_E$ contains the constraint $[\mathtt{rng}(\tau_1) \leq \mathtt{rng}(\tau_2)]$ inferred by (*compat*). Then, since $\mathit{Vars}(\tau_1) \cup \mathit{Vars}(\tau_2) \subseteq E$, the corresponding antecedent $[\tau_1 \leq \tau_2]$ is also in $\Psi\Theta(\mathcal{S})\mid_E$, and therefore:

$$\Psi\Theta(\mathcal{S})\mid_E \setminus \{\mathtt{rng}(\tau_1) \leq \mathtt{rng}(\tau_2)\} \quad \cong_E \quad \Psi\Theta(\mathcal{S})\mid_E$$

Put differently, because (*compat*) does not eliminate any variables, any (*compat*)-consequent in $\Psi\Theta(\mathcal{S})\mid_E$ is subsumed by its antecedent. If we define:

$$\Pi = \Psi \setminus \{\mathit{compat}\}$$

then this argument implies that $\Psi\Theta(\mathcal{S})\mid_E \cong_E \Pi\Theta(\mathcal{S})\mid_E$. Hence we get the following lemma.

Lemma 4.4 $\mathcal{S} \cong_E \Pi\Theta(\mathcal{S})\mid_E$.

Together, lemmas 4.2 and 4.4 provide the basis to introduce proof-theoretic equivalents of restricted entailment and observable equivalence:

- $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ iff $\Psi\Theta(\mathcal{S}_1)\mid_E \supseteq \Pi\Theta(\mathcal{S}_2)\mid_E$,
- $\mathcal{S}_1 =^E_{\Psi\Theta} \mathcal{S}_2$ iff $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ and $\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$.

The two relations completely characterize restricted entailment and observable equivalence.

Theorem 4.5 (Soundness and Completeness)

1. $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ *if and only if* $\mathcal{S}_1 \models_E \mathcal{S}_2$.
2. $\mathcal{S}_1 =^E_{\Psi\Theta} \mathcal{S}_2$ *if and only if* $\mathcal{S}_1 \cong_E \mathcal{S}_2$.

4.3 Deciding Observable Equivalence

The relation $=^E_{\Psi\Theta}$ completely characterizes the model-theoretic observable equivalence relation $\cong_E$, but for an implementation of the extended constraint derivation system we need a decision algorithm for $=^E_{\Psi\Theta}$.

Given $\mathcal{S}_1$ and $\mathcal{S}_2$ closed under Θ, this algorithm needs to verify that $\Psi(\mathcal{S}_1)\mid_E = \Psi(\mathcal{S}_2)\mid_E$. The naive approach to enumerate and to compare the two constraint systems does not work, since they are infinite. For example, if $\mathcal{S} = \{\alpha \leq \mathtt{rng}(\alpha)\}$, then $\Psi(\mathcal{S})$ is the infinite set $\{\alpha \leq \mathtt{rng}(\alpha), \alpha \leq \mathtt{rng}(\mathtt{rng}(\alpha)), \ldots\}$.

Fortunately, the infinite constraint systems inferred by Ψ exhibit a regular structure, which we exploit to decide observable equivalence as follows. First, we generate regular grammars describing the upper and lower bounds for each set variable. Second, we extend these grammars to regular tree grammars (RTGs) describing *all* constraints in $\Pi(\mathcal{S}_1)\mid_E$ and $\Pi(\mathcal{S}_2)\mid_E$, excluding those constraints inferred via *compat*, which we cannot describe in this manner. Third, we use these RTGs to decide entailment by checking if $\Psi(\mathcal{S}_1)\mid_E \supseteq \Pi(\mathcal{S}_2)\mid_E$ via an adaptation of an RTG containment algorithm. To decide observable equivalence, we simply check entailment in both directions. These steps are described in more detail below.

Regular Grammars: Our first step is to describe the lower and upper non-constant bounds for each set variable. Technically, we want to describe the following two languages of types:

$$\{\kappa \mid [\kappa \leq \alpha] \in \Psi(\mathcal{S}) \text{ and } \mathit{Vars}(\kappa) \subseteq E\}$$
$$\{\kappa \mid [\alpha \leq \kappa] \in \Psi(\mathcal{S}) \text{ and } \mathit{Vars}(\kappa) \subseteq E\}$$

for each set variable α. Both languages are generated by a regular grammar $G_r(\mathcal{S}, E)$. The grammar contains the non-terminals α_U and α_L, for each α in $\mathcal{S}$, which generate the above lower and upper bounds of α, respectively.

The productions of the grammar are determined by $\mathcal{S}$ and Ψ. To illustrate this idea, suppose $\mathcal{S}$ contains $[\alpha \leq \mathtt{rng}(\beta)]$. Then, for each upper bound κ of β, the rule (*compose*$_1$) infers the upper bound $\mathtt{rng}(\kappa)$ of α. Since, by induction, β's upper bounds are generated by β_U, the production $\alpha_U \mapsto \mathtt{rng}(\beta_U)$ generates the corresponding upper bounds of α. More generally, the collection of productions $\{\alpha_U \mapsto \mathtt{rng}(\beta_U) \mid [\alpha \leq \mathtt{rng}(\beta)] \in \mathcal{S}\}$ describes all bounds inferred via (*compose*$_1$). Bounds inferred via the remaining (*compose*) rules can be described in a similar manner.

Bounds inferred via the rule (*reflex*) imply the production rules $\alpha_U \mapsto \alpha$, $\alpha_L \mapsto \alpha$ for $\alpha \in E$. The rule (*compat*) cannot generate constraints of the form $[\kappa \leq \alpha]$ or $[\alpha \leq \kappa]$. Finally, consider the rule (*trans*), and suppose this rule infers an upper bound τ on α. This bound must be inferred from an upper bound τ on β, based on the antecedent $[\alpha \leq \beta]$. Hence the productions $\{\alpha_U \mapsto \beta_U \mid [\alpha \leq \beta] \in \mathcal{S}\}$ generate all upper bounds inferred via (*trans*). In a similar fashion, the productions $\{\beta_L \mapsto \alpha_L \mid [\alpha \leq \beta] \in \mathcal{S}\}$ generate all lower bounds inferred via (*trans*).

Definition 4.6. (*Regular Grammar* $G_r(\mathcal{S}, E)$) Let $\mathcal{S}$ be a simple constraint system and E a collection of set variables. The regular grammar $G_r(\mathcal{S}, E)$ consists of the non-terminals $\{\alpha_L, \alpha_U \mid \alpha \in \mathit{Vars}(\mathcal{S})\}$ and the following productions:

$$\begin{array}{ll}
\alpha_U \mapsto \alpha,\ \alpha_L \mapsto \alpha & \forall\, \alpha \in E \\
\alpha_U \mapsto \beta_U,\ \beta_L \mapsto \alpha_L & \forall\, [\alpha \leq \beta] \in \mathcal{S} \\
\alpha_U \mapsto \mathtt{dom}(\beta_L) & \forall\, [\alpha \leq \mathtt{dom}(\beta)] \in \mathcal{S} \\
\alpha_U \mapsto \mathtt{rng}(\beta_U) & \forall\, [\alpha \leq \mathtt{rng}(\beta)] \in \mathcal{S} \\
\beta_L \mapsto \mathtt{dom}(\alpha_U) & \forall\, [\mathtt{dom}(\alpha) \leq \beta] \in \mathcal{S} \\
\beta_L \mapsto \mathtt{rng}(\alpha_L) & \forall\, [\mathtt{rng}(\alpha) \leq \beta] \in \mathcal{S}
\end{array}$$

∎

The grammar $G_r(\mathcal{S}, E)$ describes two languages for each set variable: the upper and lower non-constant bounds. Specifically, if $\mapsto_G^*$ denotes a derivation in the grammar G, and $\mathcal{L}_G(x)$ denotes the language $\{\tau \mid x \mapsto_G^* \tau\}$ generated by a non-terminal x, then the following lemma holds.

Lemma 4.7 *If $G = G_r(\mathcal{S}, E)$, then:*

$$\begin{array}{rcl} \mathcal{L}_G(\alpha_L) & = & \{\kappa \mid [\kappa \leq \alpha] \in \Psi(\mathcal{S}) \textit{ and } \mathit{Vars}(\kappa) \subseteq E\} \\ \mathcal{L}_G(\alpha_U) & = & \{\kappa \mid [\alpha \leq \kappa] \in \Psi(\mathcal{S}) \textit{ and } \mathit{Vars}(\kappa) \subseteq E\} \end{array}$$

Proof: We prove each containment relation by induction on the appropriate derivation. ∎

Regular Tree Grammars: The grammar $G_r(\mathcal{S}, E)$ does not describe all constraints in $\Pi(\mathcal{S}) \mid_E$. In particular, it does not describe constraints of the form $[c \leq \tau]$ and constraints inferred by (*trans*) or (*compat*). To represent the constraint system $\Pi(\mathcal{S}) \mid_E$, we extend the grammar $G_r(\mathcal{S}, E)$ to a *regular tree grammar* $G_t(\mathcal{S}, E)$. It combines upper and lower bounds for set variables in the same fashion as the (*trans*) rule, and also generates constraints of the form $[c \leq \tau]$ where appropriate.

Definition 4.8. (*Regular Tree Grammar* $G_t(\mathcal{S}, E)$) The RTG $G_t(\mathcal{S}, E)$ extends the grammar $G_r(\mathcal{S}, E)$ with the root non-terminal R and the additional productions:

$$\begin{array}{ll} R \mapsto [\alpha_L \leq \alpha_U] & \forall\ \alpha \in \mathit{Vars}(\mathcal{S}) \\ R \mapsto [c \leq \alpha_U] & \forall\ [c \leq \alpha] \in \mathcal{S} \end{array}$$

where $[\cdot \leq \cdot]$ is viewed as a binary constructor. ∎

The grammar $G_t(\mathcal{S}, E)$ describes all constraints in $\Pi(\mathcal{S}) \mid_E$.

Lemma 4.9 *If $G = G_t(\mathcal{S}, E)$, then $\Pi(\mathcal{S}) \mid_E = \mathcal{L}_G(R)$.*

Before we can exploit the grammar representation of $\Pi(\mathcal{S}) \mid_E$, we must still prove that the closure under $\Theta \cup \Pi \cup \{\mathit{compat}\}$ can be performed in a sequential manner. The following lemma justifies this staging of the closure algorithm.

Lemma 4.10 *For any simple constraint system $\mathcal{S}$:*

$$\Psi\Theta(\mathcal{S}) = \Psi(\Theta(\mathcal{S})) = \mathit{compat}(\Pi(\Theta(\mathcal{S})))$$

The Entailment Algorithm: We can check entailment based on lemmas 4.9 and 4.10 as follows. Given $\mathcal{S}_1$ and $\mathcal{S}_2$, we close them under Θ and then have:

$$\begin{array}{rcl} & \mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1 & \\ \iff & \Psi\Theta(\mathcal{S}_2) \mid_E \supseteq \Pi\Theta(\mathcal{S}_1) \mid_E & \text{by defn } \vdash^E_{\Psi\Theta} \\ \iff & \Psi(\Theta(\mathcal{S}_2)) \mid_E \supseteq \Pi(\Theta(\mathcal{S}_1)) \mid_E & \text{by lemma 4.10} \\ \iff & \Psi(\mathcal{S}_2) \mid_E \supseteq \Pi(\mathcal{S}_1) \mid_E & \text{as } \mathcal{S}_i = \Theta(\mathcal{S}_i) \\ \iff & \mathit{compat}(\Pi(\mathcal{S}_2) \mid_E) \supseteq \Pi(\mathcal{S}_1) \mid_E & \text{by lemma 4.10} \\ \iff & \mathit{compat}(\mathcal{L}_{G_2}(R)) \supseteq \mathcal{L}_{G_1}(R) & \text{by lemma 4.9} \\ & \text{where } G_i = G_t(\mathcal{S}_i, E) & \end{array}$$

The Entailment Algorithm

In the following, $\mathcal{P}_{\text{fin}}$ denotes the finite power-set constructor.

$$\text{Let:}\quad \begin{array}{rclcrcl} G_1 & = & G_r(\mathcal{S}_1, E) & & L_i & = & \{\alpha_L \mid \alpha \in \mathit{Vars}(\mathcal{S}_i)\} \\ G_2 & = & G_t(\mathcal{S}_2, E) & & U_i & = & \{\alpha_U \mid \alpha \in \mathit{Vars}(\mathcal{S}_i)\} \end{array}$$

Assume G_1 and G_2 are pre-processed to remove ϵ-transitions. For $C \in \mathcal{P}_{\text{fin}}(L_2 \times U_2)$, define:

$$\mathcal{L}(C) = \{[\tau_L \leq \tau_U] \mid \langle \alpha_L, \beta_U \rangle \in C,\ \alpha_L \mapsto_{G_2} \tau_L,\ \beta_U \mapsto_{G_2} \tau_U\}$$

The relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\cdot,\cdot,\cdot,\cdot]$ is defined as the largest relation on $L_1 \times U_1 \times \mathcal{P}_{\text{fin}}(L_2 \times U_2) \times \mathcal{P}_{\text{fin}}(L_2 \times U_2)$ such that if:

$$\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D] \qquad \alpha_L \mapsto_{G_1} X \qquad \beta_U \mapsto_{G_1} Y$$

then one of the following cases hold:

1. $\mathcal{L}([X \leq Y]) \subseteq \mathcal{L}(C \cup D)$.
2. $X = \mathtt{rng}(\alpha'_L)$, $Y = \mathtt{rng}(\beta'_U)$ and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha'_L, \beta'_U, C, D']$, where:
$$\begin{array}{rl} D' = \{\langle \gamma'_L, \delta'_U \rangle \mid & \langle \gamma_L, \delta_U \rangle \in C \cup D, \\ & \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U)\} \end{array}$$
3. $X = \mathtt{dom}(\alpha'_U)$, $Y = \mathtt{dom}(\beta'_L)$ and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\beta'_L, \alpha'_U, C, D']$, where:
$$\begin{array}{rl} D' = \{\langle \delta'_L, \gamma'_U \rangle \mid & \langle \gamma_L, \delta_U \rangle \in C \cup D, \\ & \gamma_L \mapsto_{G_2} \mathtt{dom}(\gamma'_U), \delta_U \mapsto_{G_2} \mathtt{dom}(\delta'_L)\} \end{array}$$

The *computable entailment relation* $\mathcal{S}_2 \vdash^E_{\text{alg}} \mathcal{S}_1$ holds if and only if $\forall \alpha \in \mathit{Vars}(\mathcal{S}_1)$:

$$\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \alpha_U, \{\langle \gamma_L, \gamma_U \rangle \mid \gamma \in \mathit{Vars}(\mathcal{S}_2)\}, \emptyset]$$

Figure 5: The computable entailment relation $\vdash^E_{\text{alg}}$

The containment question $\mathcal{L}_{G_2}(R) \supseteq \mathcal{L}_{G_1}(R)$ can be decided via an RTG containment algorithm. To decide the more difficult question:

$$\mathit{compat}(\mathcal{L}_{G_2}(R)) \supseteq \mathcal{L}_{G_1}(R)$$

we adapt an RTG containment algorithm to allow for constraints inferred via (*compat*) on $\mathcal{L}_{G_2}(R)$.

The extended algorithm is presented in Figure 5. It first computes the largest relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}$ such that $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ holds if and only if:

$$\mathcal{L}([\alpha_L \leq \beta_U]) \subseteq \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D)$$

where α_L, β_U describe collections of types; C, D describe collections of constraints; and $\mathcal{L}([\alpha_L \leq \beta_U])$ denotes the language $\{[\tau_L \leq \tau_U] \mid \alpha_L \mapsto^* \tau_L,\ \beta_U \mapsto^* \tau_U\}$. The first case in the definition of $\mathcal{R}$ uses an RTG containment algorithm to detect if $\mathcal{L}([\alpha_L \leq \beta_U]) \subseteq \mathcal{L}(C) \cup \mathcal{L}(D)$. The two remaining cases handle constraints of the form $[\mathtt{rng}(\alpha'_L) \leq \mathtt{rng}(\beta'_U)]$ or $[\mathtt{dom}(\alpha'_U) \leq \mathtt{dom}(\beta'_L)]$, and allow for inferences via (*compat*). The relation $\mathcal{R}$ can be computed by starting with a maximal relation (true at every point), and then iteratively setting entries to false as required by figure 5, until the largest relation satisfying the definition is reached.

Based on this relation, the algorithm then defines a *computable entailment relation* $\vdash^E_{\text{alg}}$ on constraint systems. This relation is equivalent to $\vdash^E_\Psi$.

Theorem 4.11 $\mathcal{S}_2 \vdash^E_\Psi \mathcal{S}_1$ *if and only if* $\mathcal{S}_2 \vdash^E_{\text{alg}} \mathcal{S}_1$.

The entailment algorithm takes exponential time, since the size of $\mathcal{R}$ is exponential in the number of set variables in $\mathcal{S}_2$. Although faster algorithms for the entailment may exist, these algorithms must all be in PSPACE, because the containment problem on NFA's, which is PSPACE-complete [1], can be polynomially reduced to the entailment problem on constraint systems.

By using the entailment algorithm in both directions, we can now decide if two constraint systems are observable equivalent. Thus, given a constraint system, we can find a minimal, observably equivalent system by systematically generating *all* constraint systems in order of increasing size, until we find one observably equivalent to the original system. Of course, the process of computing the minimal equivalent system with this algorithm is far too expensive for use in practical program analysis systems.

5 Practical Constraint Simplification

Fortunately, to take advantage of the rule ($\cong$) in a program analysis tool, we do not need a completely minimized constraint system. Any simplifications in a constraint system produces corresponding reductions in the overall analysis time.

For this purpose, we exploit the connection between constraint systems and RTGs. By Lemmas 4.4 and 4.9, any transformation on constraint systems that preserves the language:

$$\mathcal{L}_{G_t(\Theta(\mathcal{S}),E)}(R)$$

also preserves the observable behavior of $\mathcal{S}$ with respect to E. Based on this observation, we transform a variety of existing algorithms for simplifying RTGs to algorithms for simplifying constraint systems. In the following subsections, we present the four most promising algorithms found so far. We use G to denote $G_t(\mathcal{S}, E)$, and we let X range over non-terminals and p over *paths*, which are sequences of the constructors `dom` and `rng`. Each algorithm assumes that the constraint system $\mathcal{S}$ is closed under Θ. Computing this closure corresponds to propagating data flow information locally within a program component. This step is relatively cheap, since program components are typically small (less than a few thousand lines of code).

5.1 Empty Constraint Simplification

A non-terminal X is *empty* if $\mathcal{L}_G(X) = \emptyset$. Similarly, a production is *empty* if it refers to empty non-terminals, and a constraint is *empty* if it only induces empty productions. Since empty productions have no effect on the language generated by G, an empty constraint in $\mathcal{S}$ can be deleted without changing $\mathcal{S}$'s observable behavior.

To illustrate this idea, consider the program component $P = (\lambda^g y.((\lambda^f x.1)\ y))$, where f and g are function tags. Although this example is unrealistic, it illustrates the behavior of our simplification algorithms. Analyzing P according to the constraint derivation rules yields a system $\mathcal{S}$ containing ten constraints. Closing $\mathcal{S}$ under Θ yields an additional three constraints. Figure 6 displays the resulting constraint system $\Theta(\mathcal{S})$, together with the corresponding grammar $G_t(\Theta(\mathcal{S}), \{\alpha^P\})$. An inspection of this grammar shows that the set of non-empty non-terminals is:

$$\{{\alpha^P}_L, {\alpha^P}_U, {\alpha^y}_L, {\alpha^a}_U, {\alpha^r}_L, {\alpha^1}_U, {\alpha^x}_L, R\}$$

Five of the constraints in $\Theta(\mathcal{S})$ are empty, and are removed by this simplification algorithm, yielding a simplified system of eight non-empty constraints.

5.2 Unreachable Constraint Simplification

A non-terminal X is *unreachable* if there is no production $R \mapsto [Y \le Z]$ or $R \mapsto [Z \le Y]$ such that $\mathcal{L}_G(Y) \ne \emptyset$ and $Z \rightarrow^*_G p(X)$. Similarly, a production is *unreachable* if it refers to unreachable non-terminals, and a constraint is *unreachable* if it only induces unreachable productions. Unreachable productions have no effect on the language $\mathcal{L}_G(R)$, and hence unreachable constraints in $\mathcal{S}$ can be deleted without changing the observable behavior of $\mathcal{S}$.

In the above example, the reachable non-terminals are ${\alpha^1}_U$, ${\alpha^a}_U$ and ${\alpha^g}_U$. Three of the constraints are unreachable, and are removed by this algorithm, yielding a simplified system with five reachable constraints.

5.3 Removing ϵ-Constraints

A constraint of the form $[\alpha \le \beta] \in \mathcal{S}$ is an *ϵ-constraint*. Suppose $\alpha \notin E$ and the only upper bound on α in $\mathcal{S}$ is the ϵ-constraint $[\alpha \le \beta]$, *i.e.*, there are no other constraints of the form $\alpha \le \tau$, $\texttt{rng}(\alpha) \le \gamma$, or $\gamma \le \texttt{dom}(\alpha)$ in $\mathcal{S}$. Then, for any solution ρ of $\mathcal{S}$, the set environment

Constraints	Production Rules		Non-empty	Reachable
$f \leq \alpha^f$	$R \mapsto [f \leq {\alpha^f}_U]$			
$\mathtt{dom}(\alpha^f) \leq \alpha^x$	${\alpha^x}_L \mapsto \mathtt{dom}({\alpha^f}_U)$			
$1 \leq \alpha^1$	$R \mapsto [1 \leq {\alpha^1}_U]$		$1 \leq \alpha^1$	$1 \leq \alpha^1$
$\alpha^1 \leq \mathtt{rng}(\alpha^f)$	${\alpha^1}_U \mapsto \mathtt{rng}({\alpha^f}_U)$			
$\mathtt{rng}(\alpha^f) \leq \alpha^a$	${\alpha^a}_L \mapsto \mathtt{rng}({\alpha^f}_L)$			
$\alpha^y \leq \alpha^r$	${\alpha^y}_U \mapsto {\alpha^r}_U$	${\alpha^r}_L \mapsto {\alpha^y}_L$	$\alpha^y \leq \alpha^r$	
$\alpha^r \leq \mathtt{dom}(\alpha^f)$	${\alpha^r}_U \mapsto \mathtt{dom}({\alpha^f}_L)$			
$g \leq \alpha^P$	$R \mapsto [g \leq {\alpha^P}_U]$		$g \leq \alpha^P$	$g \leq \alpha^P$
$\mathtt{dom}(\alpha^P) \leq \alpha^y$	${\alpha^y}_L \mapsto \mathtt{dom}({\alpha^P}_U)$		$\mathtt{dom}(\alpha^P) \leq \alpha^y$	
$\alpha^a \leq \mathtt{rng}(\alpha^P)$	${\alpha^a}_U \mapsto \mathtt{rng}({\alpha^P}_U)$		$\alpha^a \leq \mathtt{rng}(\alpha^P)$	$\alpha^a \leq \mathtt{rng}(\alpha^P)$
$\alpha^r \leq \alpha^x$	${\alpha^r}_U \mapsto {\alpha^x}_U$	${\alpha^x}_L \mapsto {\alpha^r}_L$	$\alpha^r \leq \alpha^x$	
$\alpha^1 \leq \alpha^a$	${\alpha^1}_U \mapsto {\alpha^a}_U$	${\alpha^a}_L \mapsto {\alpha^1}_L$	$\alpha^1 \leq \alpha^a$	$\alpha^1 \leq \alpha^a$
$1 \leq \alpha^a$	$R \mapsto [1 \leq {\alpha^a}_U]$		$1 \leq \alpha^a$	$1 \leq \alpha^a$
	${\alpha^P}_L \mapsto \alpha^P$	${\alpha^P}_U \mapsto \alpha^P$		

Figure 6: The original constraint system, grammar and simplified constraint systems for $P = (\lambda^g y.((\lambda^f x.1)\ y))$

ρ' defined by:

$$\rho'(\delta) = \begin{cases} \rho(\delta) & \text{if } \delta \not\equiv \alpha \\ \rho(\beta) & \text{if } \delta \equiv \alpha \end{cases}$$

is also a solution of $\mathcal{S}$. Therefore we can replace all occurrences of α in $\mathcal{S}$ by β while still preserving the observable behavior $\mathit{Soln}(\mathcal{S})\mid_E$. This substitution transforms the constraint $[\alpha \leq \beta]$ to the tautology $[\beta \leq \beta]$, which can be deleted. Dually, if $[\alpha \leq \beta] \in \mathcal{S}$ with $\beta \notin E$ and β having no other lower bounds, then we can replace β by α, again eliminating the constraint $[\alpha \leq \beta]$.

To illustrate this idea, consider the remaining constraints for P. In this system, the only upper bound for the set variable α^1 is the ϵ-constraint $[\alpha^1 \leq \alpha^a]$. Hence this algorithm replaces all occurrences of α^1 by α^a, which further simplifies this constraint system into:

$$\{1 \leq \alpha^a, \alpha^a \leq \mathtt{rng}(\alpha^P), g \leq \alpha^P\}$$

This system is the smallest simple constraint system observably equivalent to the original system $\Theta(\mathcal{S})$.

5.4 Hopcroft's Algorithm

The previous algorithm *merges* set variables under certain circumstances, and only when they are related by an ϵ-constraint. We would like to identify more general circumstances under which set variables can be merged. To this end, we define a *valid unifier* for $\mathcal{S}$ to be an equivalence relation $\sim$ on the set variables of $\mathcal{S}$ such that we can merge the set variables in each equivalence class of $\sim$ without changing the observable behavior of $\mathcal{S}$. Using a model-theoretic argument, we can show that an equivalence relation $\sim$ is a valid unifier for $\mathcal{S}$ if for all solutions $\rho \in \mathit{Soln}(\mathcal{S})$ there exists another solution $\rho' \in \mathit{Soln}(\mathcal{S})$ such that ρ' agrees with ρ on E and $\rho'(\alpha) = \rho'(\beta)$ for all $\alpha \sim \beta$.

A natural strategy for generating ρ' from ρ is to map each set variable to the least upper bound of the set variables in its equivalence class:

$$\rho'(\alpha) = \bigsqcup_{\alpha' \sim \alpha} \rho(\alpha')$$

Figure 7 describes sufficient conditions to ensure that ρ' is a solution of $\mathcal{S}$, and hence that $\sim$ is a valid unifier for $\mathcal{S}$. To produce an equivalence relation satisfying these conditions, we use a variant of Hopcroft's $O(n \lg n)$ time

1. Use a variant of Hopcroft's algorithm [12] to compute an equivalence relation $\sim$ on the set variables of $\mathcal{S}$ that satisfies the following conditions:
   (a) Each set variable in E is in an equivalence class by itself.
   (b) If $[\alpha \leq \beta] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \exists \beta \sim \beta'$ such that $[\alpha' \leq \beta'] \in \mathcal{S}$.
   (c) If $[\alpha \leq \mathtt{rng}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \exists \beta \sim \beta'$ such that $[\alpha' \leq \mathtt{rng}(\beta')] \in \mathcal{S}$.
   (d) If $[\mathtt{rng}(\alpha) \leq \beta] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \exists \beta \sim \beta'$ such that $[\mathtt{rng}(\alpha') \leq \beta'] \in \mathcal{S}$.
   (e) If $[\alpha \leq \mathtt{dom}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \forall \beta \sim \beta'$ such that $[\alpha' \leq \mathtt{dom}(\beta')] \in \mathcal{S}$.
2. Merge set variables according to their equivalence class.

Figure 7: The *Hopcroft* algorithm

algorithm [12] for computing an equivalence relation on states in a DFA and then merge set variables according to their equivalence class.[6]

5.5 Simplification Benchmarks

To test the effectiveness of the simplification algorithms, we extended MrSpidey with the four algorithms that we have just described: *empty*, *unreachable*, *ϵ-removal*, and *Hopcroft*. Each algorithm also implements the preceding simplification strategies. The first three algorithms are linear in the number of non-empty constraints in the system, and *Hopcroft* is log-linear.

We tested the algorithms on the constraint systems for nine program components on a 167MHz Sparc Ultra 1 with 326M of memory, using the MzScheme byte code compiler [10]. The results are described in figure 8. The second column gives the number of lines in each program component, and the third column gives the number of constraints in the original (unsimplified) constraint system after closing it under the rules Θ. The remaining columns describe the behavior of each simplification algorithm, presenting the factor by which the number of constraints was reduced, and the time (in milliseconds) required for this simplification.

The results demonstrate the effectiveness and efficiency of our simplification algorithms. The resulting constraint systems are typically at least an order of magnitude smaller than the original system. The cost of these algorithms is reasonable, particularly considering that they were run on a byte code compiler. As expected, the more sophisticated algorithms are more effective, but are also more expensive.

6 Componential Set-Based Analysis

Equipped with the simplification algorithms, we return to our original problem of developing a componential set-based analysis. The new analysis tool processes programs in three steps.

1. For each component in the program, the analysis derives and simplifies the constraint system for that component and saves the simplified system in a *constraint file*, for use in later runs of the analysis. The simplification is performed with respect to the external variables of the component, *excluding* expression labels, in order to minimize the size of the simplified system. Thus, the simplified system only needs to describe how the component interacts with the rest of the program, and the simplification algorithm can discard constraints that are only necessary to infer local value set invariants. These discarded constraints are reconstructed later as needed.

   This step can be skipped for each program component that has not changed since the last run of the analysis, since its constraint file can be used instead.

2. The analysis combines the simplified constraint systems of the *entire* program and closes the combined collection of constraints under Θ, thus propagating data flow information between the constraint systems for the various program components.

3. Finally, to reconstruct the full analysis results for the program component that the programmer is focusing on, the analysis tool combines the constraint system from the second step with the unsimplified constraint system for that component. It closes the resulting system under Θ, which yields appropriate value set invariants for each labeled expression in the component.

[6] A similar development based on the definition $\rho'(\alpha) = \sqcap\{\rho(\alpha') \mid \alpha \sim \alpha'\}$ results in an alternative algorithm, which is less effective in practice.

The new analysis can easily process programs that consist of many components. For its first step, it eliminates all those constraints that have only local relevance, thus producing a small combined constraint system for the entire program. As a result, the analysis tool can solve the combined system more quickly and using less space than traditional set-based analysis [11]. Finally, it recreates as much precision as traditional set-based analysis as needed on a per-component basis.

The new analysis performs extremely in an interactive setting because it exploits the saved constraint files where possible and thus avoids re-processing many program components unnecessarily.

We implemented four variants of this analysis. Each analysis uses a particular simplification algorithm to simplify the constraint systems for the program components.

6.1 Benchmarks

We tested the componential analyses with five benchmark programs, ranging from 1,200 to 17,000 lines. For comparison purposes, we also analyzed each benchmark with the *standard* set-based analysis that performs no simplification. The analyses handled library functions in a context-sensitive, polymorphic manner according to the constraint derivation rules (*let*) and (*inst*) to avoid merging information between unrelated calls to these

Definition	lines	size	*empty* factor	*empty* time	*unreachable* factor	*unreachable* time	*ε-removal* factor	*ε-removal* time	*Hopcroft* factor	*Hopcroft* time
`map`	5	221	3	<10	6	20	11	30	13	30
`reverse`	6	287	4	<10	8	20	20	10	20	30
`substring`	8	579	12	10	64	10	64	10	96	20
`qsort`	41	1387	15	<10	15	30	58	50	66	40
`unify`	89	2921	10	10	11	80	55	120	65	150
`hopcroft`	201	8429	25	10	42	100	118	100	124	200
`check`	237	21854	4	50	4	1150	26	370	168	510
`escher-fish`	493	30509	187	10	678	40	678	40	678	80
`scanner`	1209	59215	3	180	17	840	45	2450	57	2120

Figure 8: Behavior of the constraint simplification algorithms.

functions. The remaining functions were analyzed in a context-insensitive, monomorphic manner. The results are documented in figure 9.

The third column in the figure shows the maximum size of the constraint system generated by each analysis, and also shows this size as a percentage of the constraint system generated by the *standard* analysis. The analyses based on the simplification algorithms produce significantly smaller constraint systems, and can also analyze more programs, such as `sba` and `poly`, for which the *standard* analysis exhausted heap space.

The fourth column shows the time required to analyze each program from scratch, without using any existing constraint files.[7] The analyses that exploit constraint simplification yield significant speed-ups over the *standard* analysis because they manipulate much smaller constraint systems. The results indicate that, for these benchmarks, the *ε-removal* algorithm yields the best trade-off between efficiency and effectiveness of the simplification algorithms. The additional simplification performed by the more expensive *Hopcroft* algorithm is out-weighed by the overhead of running the algorithm. The tradeoff may change as we analyze larger programs.

To test the responsiveness of the componential analyses in an interactive setting based on an analyze-debug-edit cycle, we re-analyzed each benchmark after changing a randomly chosen component in that benchmark. The re-analysis times are shown in the fifth column of figure 9. These times show an order-of-magnitude improvement in analysis times over the original, *standard* analysis, since the saved constraint files are used to avoid reanalyzing all of the unchanged program components. For example, the analysis of `zodiac`, which used to take over two minutes, now completes in under four seconds. Since practical debugging sessions using MrSpidey typically involve repeatedly analyzing the project each time the source code of one module is modified, *e.g.*, when a bug is identified and eliminated, using separate analysis substantially improves the usability of MrSpidey.

[7] These times exclude scanning and parsing time.

Program (# lines)	Analysis	Number of constraints		Analysis/ Re-analysis time (s)		File size (bytes)
`scanner` (1253)	*standard*	61K		14.1	7.7	572K
	empty	24K	(39%)	12.0	3.1	189K
	unreachable	15K	(25%)	9.7	2.0	39K
	ε-removal	14K	(23%)	9.5	1.7	28K
	Hopcroft	14K	(23%)	10.4	1.7	25K
`zodiac` (3419)	*standard*	704K		133.4	110.6	1634K
	empty	62K	(9%)	34.1	8.1	328K
	unreachable	21K	(3%)	28.8	4.5	169K
	ε-removal	13K	(2%)	28.8	3.8	147K
	Hopcroft	11K	(2%)	31.4	3.8	136K
`nucleic` (3432)	*standard*	333K		83.9	51.2	2882K
	empty	90K	(27%)	52.8	17.8	592K
	unreachable	68K	(20%)	48.4	14.6	386K
	ε-removal	56K	(17%)	48.3	13.1	330K
	Hopcroft	56K	(17%)	60.9	13.2	328K
`sba` (11560)	*standard*	>5M	*	*	*	*
	empty	1908K	(<38%)	181.5	65.5	1351K
	unreachable	105K	(<2%)	149.5	43.3	920K
	ε-removal	76K	(<2%)	147.1	42.2	770K
	Hopcroft	65K	(<1%)	156.8	41.1	716K
`poly` (17661)	*standard*	>5M	*	*	*	*
	empty	>5M	*	*	*	*
	unreachable	201K	(<4%)	259.6	26.9	1517K
	ε-removal	68K	(<1%)	239.6	13.3	1038K
	Hopcroft	38K	(<1%)	254.1	10.9	907K

* indicates the analysis exhausted heap space

Figure 9: Behavior of the componential analyses.

The disk-space required to store the constraint files is shown in column six. Even though these files use a straight-forward, text-based representation, their size is typically within a factor of two or three of the corresponding source file.

Program	lines	*copy* analysis	Relative time of *smart* polymorphic analyses				Mono. analysis
			empty	*unreachable*	ϵ*-removal*	*Hopcroft*	
`lattice`	215	4.2s	39%	36%	35%	38%	42%
`browse`	233	2.5s	76%	76%	76%	81%	75%
`splay`	265	7.9s	75%	73%	70%	72%	83%
`check`	281	50.1s	21%	23%	14%	14%	23%
`graphs`	621	2.8s	85%	85%	82%	87%	82%
`boyer`	624	4.3s	46%	46%	49%	50%	40%
`matrix`	744	7.5s	64%	57%	51%	52%	45%
`maze`	857	6.2s	64%	59%	58%	61%	54%
`nbody`	880	39.6s	57%	25%	25%	26%	28%
`nucleic`	3335	*	* 243s	* 42s	* 42s	* 44s	* 36s

* indicates the *copy* analysis exhausted heap space,
and the table contains absolute times for the other analyses

Figure 10: Times for the smart polymorphic analyses, relative to the *copy* analysis.

7 Efficient Polymorphic Analysis

The constraint simplification algorithms also enables an efficient polymorphic, or context-sensitive, analysis. To avoid merging information between unrelated calls to functions that are used in a polymorphic fashion, a polymorphic analysis duplicates the function's constraints at each call site. We extended MrSpidey with five polymorphic analyses. The first analysis is *copy*, which duplicates the constraint system for each polymorphic reference via a straightforward implementation of the rules (*let*) and (*inst*).[8] The remaining four analyses are *smart* analyses that simplify the constraint system for each polymorphic definition.

We tested the analyses using a standard set of benchmarks [13]. The results of the test runs are documented in figure 10. The second column shows the number of lines in each benchmark; the third column presents the time for the *copy* analysis; and columns four to seven show the times for each smart polymorphic analysis, as a percentage of the *copy* analysis time. For comparison purposes, the last column shows the relative time of the original, but less accurate, monomorphic analysis.

The results again demonstrate the effectiveness of our constraint simplification algorithms. The smart analyses that exploit constraint simplification are always significantly faster and can analyze more programs than the *copy* analysis. For example, while *copy* exhausts heap space on the `nucleic` benchmark, all smart analyses successfully analyzed this benchmark.

Again, it appears that the ϵ*-removal* analysis yields the best trade-off between efficiency and effectiveness of the simplification algorithms. This analysis provides the additional accuracy of polymorphism without much additional cost over the coarse, monomorphic analysis. With the exception of the benchmarks `browse`, `splay` and `graphs`, which do not re-use many functions in a polymorphic fashion, this analysis is a factor of 2 to 4 times faster than the *copy* analysis, and it is also capable of analyzing larger programs.

[8] We also implemented a polymorphic analysis that re-analyzes a definition at each reference, but found its performance to be comparable to, and sometimes worse than, the *copy* analysis.

8 Competitive Work

Fähndrich and Aiken [6] examine constraint simplification for an analysis based on a more complex constraint language. They develop a number of heuristic algorithms for constraint simplification, which they test on programs of up to 6000 lines. Their fastest approach yields a factor of 3 saving in both time and space, but is slow in absolute times compared to other analyses.

Pottier [17] studies an ML-style language with a subtype system based on constraints, and and presents an incomplete algorithm for deciding entailment on constraint systems. He proposes some *ad hoc* algorithms for simplifying constraints, but does not present results on the cost or effectiveness of these algorithms.

Eifrig, Smith and Trifonov [5, 21] describe a subtyping relation between constrained types that are similar to our constraint systems, and they present an incomplete decision algorithm for subtyping. They describe three algorithms for simplifying constraint systems, two of which which are similar to the *empty* and ϵ*-removal* algorithms, and the third is a special case of the *Hopcroft* algorithm. They do not present results on the cost or effectiveness of these algorithms.

Duesterwald *et al* [4] describe algorithms for simplifying data flow equations. These algorithms are similar to the ϵ*-removal* and *Hopcroft* algorithms. Their approach only preserves the greatest solution of the equation system and assumes that the control flow graph is already known. Hence it cannot be used to analyze programs in a componential manner or to analyze programs with advanced control-flow mechanisms such as first-class functions and virtual methods. The paper does not present results on the cost or effectiveness of these algorithms.

9 Future Work

All our constraint simplification algorithms preserve the observable behavior of constraint systems, and thus do not affect the accuracy of the analysis. If we were willing to tolerate a less accurate analysis, we could choose a compressed constraint system that does not preserve the observable behavior of the original, but only *entails* that behavior. This approach allows the use of much smaller constraint systems, and hence yields a faster analysis.

A promising approach for deriving such approximate constraint systems is to rely on a programmer-provided *signature* describing the behavior of each program component, and to derive the new constraint system from that signature. After checking the entailment condition to verify that signature-based constraints correctly approximates the behavior of the module, we could use those constraints in the remainder of the analysis. Since the signature-based constraints are smaller than the derived ones, this approach could significantly reduce analysis times for large projects. We are investigating this approach for developing a typed module language on top of Scheme.

References

[1] Aho, A., J. Hopcroft and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

[2] Aiken, A., Wimmers, E. L., and Lakshman, T. K. Soft typing with conditional types. In *Proceedings of the ACM Sigplan Conference on Principles of Programming Languages* (1994), pp. 163–173.

[3] Cousot, P., and Cousot, R. Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the 1995 Conference on Functional Programming and Computer Architecture* (1995), pp. 170–181.

[4] Duesterwald, E., Gupta, R., and Soffa, M. L. Reducing the cost of data flow analysis by congruence partitioning. In *International Conference on Compiler Construction* (April 1994).

[5] Eifrig, J., Smith, S., and Trifonov, V. Sound polymorphic type inference for objects. In *Conference on Object-Oriented Programming Systems, Languages, and Applications* (1995).

[6] Fähndrich, M., and Aiken, A. Making set-constraint based program analyses scale. Technical Report UCB/CSD-96-917, University of California at Berkeley, 1996.

[7] Flanagan, C., and Felleisen, M. Set-based analysis for full Scheme and its use in soft-typing. Technical Report TR95-254, Rice University, 1995.

[8] Flanagan, C., and Felleisen, M. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR-96-266, Rice University, 1996.

[9] Flanagan, C., Flatt, M., Krishnamurthi, S., Weirich, S., and Felleisen, M. Finding bugs in the web of program invariants. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (1996), pp. 23–32.

[10] Flatt, M. *MzScheme Reference Manual.* Rice University.

[11] Heintze, N. Set-based analysis of ML programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994), pp. 306–317.

[12] Hopcroft, J. E. An n log n algorithm for minimizing the states of a finite automaton. *The Theory of Machines and Computations* (1971), 189–196.

[13] Jagannathan, S., and Wright, A. K. Effective flow analysis for avoiding run-time checks. In *Proc. 2nd International Static Analysis Symposium, LNCS 983* (September 1995), Springer-Verlag, pp. 207–224.

[14] Jones, N., and Muchnick, S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages* (January 1982), pp. 66–74.

[15] Palsberg, J. Closure analysis in constraint form. *Transactions on Programming Languages and Systems 17*, 1 (1995), 47–62.

[16] Palsberg, J., and O'Keefe, P. A type system equivalent to flow analysis. In *Proceedings of the ACM SIGPLAN '95 Conference on Principles of Programming Languages* (1995), pp. 367–378.

[17] Pottier, F. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming* (1996), pp. 122–133.

[18] Reynolds, J. Automatic computation of data set defintions. *Information Processing'68* (1969), 456–461.

[19] Shivers, O. *Control-flow Analysis of Higher-Order Languages, or Taming Lambda.* PhD thesis, Carnegie-Mellon University, 1991.

[20] Tofte, M. Type inference for polymorphic references. *Information and Computation 89*, 1 (November 1990), 1–34.

[21] Trifonov, V., and Smith, S. Subtyping constrained types. In *Third International Static Analysis Symposium (LNCS 1145)* (1996), pp. 349–365.

[22] Wright, A., and Felleisen, M. A syntactic approach to type soundness. *Information and Computation 115*, 1 (1994), 38–94.

[23] Wright, A. K. Simple imperative polymorphism. *Lisp and Symbolic Computation 8*, 4 (Dec. 1995), 343–356.