Componential Set-Based Analysis

CORMAC FLANAGAN
Compaq Systems Research Center
and
MATTHIAS FELLEISEN
Rice University

Set-based analysis (SBA) produces good predictions about the behavior of functional and object-oriented programs. The analysis proceeds by inferring *constraints* that characterize the data flow relationships of the analyzed program. Experiences with MrSpidey, a static debugger based on SBA, indicate that SBA can adequately deal with programs of up to a couple of thousand lines of code. SBA fails, however, to cope with larger programs because it generates systems of constraints that are at least linear, and possibly quadratic, in the size of the analyzed program.

This article presents theoretical and practical results concerning methods for reducing the size of constraint systems. The theoretical results include a proof-theoretic characterization of the *observable behavior* of constraint systems for program components, and a complete algorithm for deciding the observable equivalence of constraint systems. In the course of this development we establish a close connection between the observable equivalence of constraint systems and the equivalence of regular-tree grammars. We then exploit this connection to adapt a variety of algorithms for simplifying grammars to the problem of simplifying constraint systems.

Based on the resulting algorithms, we have developed *componential set-based analysis*, a modular and polymorphic variant of SBA. Experimental results verify the effectiveness of the simplification algorithms and the componential analysis. The simplified constraint systems are typically an order of magnitude smaller than the original systems. These reductions in size produce significant gains in the speed of the analysis.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, symbolic execution*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*(static) debugger, optimizer*; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*(soft) type structure*

General Terms: Algorithms, Languages, Performance, Theory

Additional Key Words and Phrases: constraint-based analysis, program analysis, Scheme, soft typing, static debugging

Authors' addresses: C. Flanagan, Compaq Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301; email: flanagan@pa.dec.com; M. Felleisen, Department of Computer Science, Rice University, 6100 South Main, Houston, TX 77005-1892; email: matthias@cs.rice.edu.

1. THE EFFECTIVENESS OF SET-BASED ANALYSIS

Rice's Scheme program development environment provides a static debugger, MrSpidey, which statically analyzes a program and, using the results of this analysis, checks the soundness of all primitive operations [Flanagan et al. 1996]. If an operation may fault due to a violation of its precondition, MrSpidey highlights the program operation so that the programmer can investigate the potential fault site before running the program.

MrSpidey's underlying program analysis is a constraint-based system similar to Heintze's set-based analysis of ML programs [Heintze 1994]. The analysis consists of two co-mingled phases: a *derivation* phase, which derives constraints describing the data flow relationships of the analyzed program, and a *solution* phase, which solves these constraints. The solution conservatively approximates the set of possible values for each program expression.

In practice, MrSpidey has proven highly effective for pedagogic programming, which includes programs of several hundreds to 2,000 lines of code. It also works reasonably well on some programs of up to several thousand lines in length. However, it becomes less useful for debugging large programs, for two reasons:

—Set-based analysis has an $O(n^3)$ worst-case time bound. The constant on the cubic element in the polynomial is small, but it becomes noticeable for programs of several thousand lines.

—Large programming projects tend to reuse functions in a polymorphic fashion. To avoid merging information between unrelated calls to such functions, the analysis must duplicate the function's constraint system at each corresponding call site. This duplication is expensive because of the size of the constraint system.

A closer look at these two obstacles suggests that the major limitation of set-based analysis is the size of the constraint system that it generates. If we could reduce the size of a constraint system without affecting the solution space that it denotes, we could simplify constraint systems for program components at intermediate stages during the analysis and thus reduce the analysis time. By simplifying the constraint system for each module, we could significantly reduce the cost of solving the combined set of constraints for a modularized program; similarly, by simplifying the constraint system for a polymorphic function definition, we could reduce the cost of duplicating that constraint system at each polymorphic reference.

The simplification of constraint systems raises both interesting theoretical and practical questions. On the theoretical side, we need to ensure that simplification preserves the solution space, or *observable behavior*, of a constraint system. In this article, we provide a proof-theoretic characterization of observable behavior and establish a close connection between the observable equivalence of constraint systems and the equivalence of regular tree grammars (RTGs).[1] Exploiting this connection, we develop algorithms for deciding the observable equivalence of constraint systems, and for finding a minimal constraint system observably equivalent

[1] A number of researchers, including Reynolds [1969], Jones and Muchnick [1982], Heintze [1994], Aiken [1994], and Cousot and Cousot [1995], previously exploited the relationship between RTGs and the *least solution* of a constraint system. We present a different result, namely a connection between RTGs and the observable behavior (i.e., the *entire solution space*) of constraint systems.

to a given system. Unfortunately, both of these problems are PSPACE-hard.

Since a minimal constraint system is only optimal but not necessary for practical purposes, the practical question concerns finding approximate algorithms for simplifying constraint systems that would make MrSpidey more useful on large programs. To answer this question, we exploit the correspondence between RTGs and constraint systems to adapt a variety of algorithms for simplifying RTG to the problem of simplifying constraint systems. Based on these simplification algorithms, we develop a *componential*,[2] or componentwise, variant of set-based analysis. Experimental results verify the effectiveness of the simplification algorithms and the componential analysis. The simplified constraint systems are typically an order of magnitude smaller than the original systems. These reductions in size produce significant gains in the speed of the analysis.

The presentation of our results proceeds as follows. Section 2 introduces an idealized source language, and Section 3 reviews the traditional set-based analysis of that language. Section 4 formalizes the notion of constraint system equivalence, based on the denotational semantics of constraints. Section 5 describes a logic for constraint system equivalence, and Section 6 uses this logic to develop a connection between RTGs and the constraint systems. Section 7 exploits this connection to derive a number of practical constraint simplification algorithms. Sections 8 and 9 discuss how these algorithms perform in a realistic program analysis system. Section 10 discusses related work. Section 11 describes directions for future research. Appendix A outlines how the analysis extends to additional language features such as data-structures, assignments, and nonlocal control operators. Appendix B contains proofs of various theorems and lemmas. Appendix C presents the algorithm for deciding constraint system equivalence. Appendix D contains an index of notations used in the article.

Notation. We use $A \overset{\circ}{\longrightarrow} B$ to denote the set of partial maps from A to B, $\mathcal{P}$ to denote the power-set constructor, and $\mathcal{P}_{\text{fin}}$ to denote the finite power-set constructor.

2. THE SOURCE LANGUAGE

For simplicity, we present our results for a λ-calculus-like language with constants and labeled expressions. It is straightforward to extend the analysis to a realistic language with assignments, recursive data structures, objects, and nonlocal control operators along the lines outlined in Appendix A and described in more detail in the first author's dissertation [Flanagan 1997].

2.1 Syntax

Expressions in our language are either variables, values, function applications, labeled expressions, or **let**-expressions: see Figure 1. 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 [Wright 1995]. We use labels to identify those program expressions whose values we wish to predict. We work with the usual conventions and terminology of

[2]**componential** *a.* of or pertaining to components.

Syntax:

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

Fig. 1. The source language Λ.

the λ_v-calculus [Plotkin 1975] 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*.

2.2 Semantics

We specify the meaning of programs based on three notions of reduction:

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

The β_v and β_{let} rules are the conventional rules for the λ-calculus. The *unlabel* rule simply removes the label from an expression once its value is needed.

An *evaluation context* $\mathcal{E}$ is an expression containing a hole $[\,]$ in place of the next subterm to be evaluated:

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

For example, in the term $(N\ M)$, the next expression to be evaluated lies within N, and thus the definition of evaluation contexts includes the clause $(\mathcal{E}\ M)$. An evaluation context always contains a single hole $[\,]$, and we use the notation $\mathcal{E}[\ M\ ]$ to denote the term produced by filling the hole in $\mathcal{E}$ with the term M.

The standard reduction relation $\longmapsto$ is the compatible closure [Barendregt 1984, ch.#2] of $\longrightarrow$ with respect to evaluation contexts:

$$\mathcal{E}[\ M\ ] \longmapsto \mathcal{E}[\ N\ ] \qquad \Longleftrightarrow \qquad M \longrightarrow N.$$

The relation $\longmapsto^*$ is the reflexive, transitive closure of $\longmapsto$. The semantics of the language is defined via the partial function *eval* on programs:

$$
\begin{array}{rcll}
\mathit{eval} : \Lambda^0 & \longrightarrow\!\!\!\!\!\!\circ\ \ & \mathit{Value} & \\
\mathit{eval}(M) & = & V & \text{if } M \longmapsto^* V.
\end{array}
$$

3. A REVIEW OF HEINTZE'S SET-BASED ANALYSIS

Set-based analysis consists of two phases: a *derivation* phase and a *solution* phase.[3] The derivation phase derives *constraints* on the sets of values that program expressions may assume. These constraints describe the data flow relationships of the

[3] Cousot and Cousot [1995] showed that the results of set-based analysis can alternatively be computed via an abstract interpretation based on chaotic iteration.

analyzed program. The solution phase solves these constraints to yield a conservative approximation of the set of possible values for each labeled expression in the program.

3.1 The Constraint Language

A *constraint* is simply an inequality between set expressions. Each set expression denotes a set of values, and the constraint denotes the corresponding set containment relationship. A *set expression* τ is either a constant, a set variable, or one of the selector expressions $\texttt{dom}(\tau)$ or $\texttt{rng}(\tau)$:

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

The selector expression $\texttt{rng}(\tau)$ denotes the set of values returned by functions in τ; similarly, $\texttt{dom}(\tau)$ denotes the set of values to which these functions are applied. Constants include both basic constants and function tags. Set variables include program labels as a strict subset.

An *atomic constraint* $\mathcal{C}$ is one of the following inequalities between set expressions, and an *atomic constraint system* $\mathcal{S}$ is a collection of atomic constraints:

$$
\begin{array}{rcrcl}
\mathcal{C} \in & \mathit{AtomicCon} = & c \leq \beta & \mid & \alpha \leq \beta \\
& & \mid\ \alpha \leq \texttt{dom}(\beta) & \mid & \texttt{dom}(\alpha) \leq \beta \\
& & \mid\ \alpha \leq \texttt{rng}(\beta) & \mid & \texttt{rng}(\alpha) \leq \beta
\end{array}
$$

$$
\mathcal{S} \in \mathit{AtomicConSystem} = \mathcal{P}_{\text{fin}}(\mathit{AtomicCon}).
$$

We use $\mathit{SetVar}(\mathcal{S})$ to denote the collection of set variables in a constraint system $\mathcal{S}$. For clarity, we sometimes enclose constraints inside square brackets, as in $[\tau_1 \leq \tau_2]$.

Our constraint language is based on the "selectors" $\texttt{dom}(\tau)$ or $\texttt{rng}(\tau)$ instead of the more usual "constructors" $(\tau_1 \rightarrow \tau_2)$ [Aiken et al. 1994]. For example, we describe a function's behavior via the two constraints $\texttt{dom}(\alpha) \leq \alpha_1$ and $\alpha_2 \leq \texttt{rng}(\alpha)$, instead of the combined constraint $(\alpha_1 \rightarrow \alpha_2) \leq \alpha$. By using selectors, we can specify each "quantum" of the program's data-flow behavior independently, which aids in the development of constraint simplification algorithms.

3.2 Deriving Constraints

The derivation phase of set-based analysis derives atomic constraints on the sets of values that program expressions may assume. Following Aiken et al. [1994] and Palsberg and O'Keefe [1995], we formulate this derivation as a proof system. Each proof rule infers a judgment of the form $\Gamma \vdash M : \alpha, \mathcal{S}$, where

(1) the *derivation context* Γ maps the free variables of the expression M to either set variables or *constraint schemas* (see below);
(2) α names the value set of M; and
(3) $\mathcal{S}$ is a system of atomic constraints describing the data-flow relationships of M.

The derivation proceeds in a syntax-directed manner according to the constraint derivation rules presented in Figure 2. The rule (*var*) extracts the appropriate set variable α for a particular program variable x from the derivation context. The

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

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

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

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

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

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

$$\frac{\psi \text{ is a substitution of set variables for } \overline{\alpha}}{\Gamma \cup \{x : \forall\overline{\alpha}.\,(\alpha_V, \mathcal{S}_V)\} \vdash x : \beta, \psi(\mathcal{S}_V) \cup \{\psi(\alpha_V) \leq \beta\}} \qquad (inst)$$

Fig. 2. Constraint derivation rules.

rule (*const*) generates the constraint $b \leq \alpha$, which ensures that the value set for a constant expression contains that constant. The rule (*label*) records the possible values of a labeled expression M^l in the label l.

The rule (*abs*) for functions records the function's tag and 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* $\sigma = \forall\overline{\alpha}.\,(\beta, \mathcal{S})$ for polymorphic, **let**-bound values [Aiken et al. 1994; Tofte 1990]. The set variable β names the result of the value; the system of atomic constraints $\mathcal{S}$ describes the data-flow relationships of the value, using β; and the set $\overline{\alpha} = \{\alpha_1, \ldots, \alpha_n\}$ 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*).

The derivation context Γ maps program variables to either set variables or constraint schemas:

$$\begin{array}{rcl} \Gamma \in & \mathit{DerivCtxt} & = \mathit{Var} \longrightarrow\!\!\!\!\!\circ\ (\mathit{SetVar} \cup \mathit{ConSchema}) \\ \sigma \in & \mathit{ConSchema} & = \forall\overline{\alpha}.\,(\beta, \mathcal{S}). \end{array}$$

We use $FV[\mathit{range}(\Gamma)]$ to denote the free set variables in the range of Γ. The free set variables of a constraint schema $\forall\overline{\alpha}.\,(\beta, \mathcal{S})$ are those in $\mathit{SetVar}(\mathcal{S}) \cup \{\beta\}$ but not

$$\frac{c \leq \beta \qquad \beta \leq \gamma}{c \leq \gamma} \tag{s_1}$$

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

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

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

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

Fig. 3. The rules $\Theta = \{s_1, \ldots, s_5\}$.

in $\overline{\alpha}$, and the free variables of a set variable are simply the set variable itself.

Many of the constraint derivation rules introduce new set variables. For example, the rule (*const*) introduces the new set variable α. Whenever this rule is applied, we need to choose a fresh set variable for α that is not used elsewhere in the constraint derivation. Choosing a fresh set variable in this manner yields a more accurate analysis.

3.3 Solving Constraint Systems

The derivation phase generates a system of atomic constraints that describes the data-flow behavior of the analyzed program. To solve this constraint system, we close it under the inference rules Θ described in Figure 3. Intuitively, these rules infer all the data-flow paths in the program, which are described by constraints of the form $\beta \leq \gamma$ (for $\beta, \gamma \in \mathit{SetVar}$), and propagate values along those data-flow 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. The rule (s_4) constructs the data-flow paths from function bodies to corresponding call sites for each function call, and the rule (s_5) similarly constructs data-flow paths from actual to formal parameters. 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 $\{\mathcal{C} \mid \mathcal{S} \vdash_\Theta \mathcal{C}\}$.

The analysis tool uses a worklist algorithm to compute the closure of $\mathcal{S}$ under Θ efficiently. The worklist keeps track of all constraints in $\mathcal{S}$ whose consequences under Θ may not be in $\mathcal{S}$. The algorithm repeatedly removes a constraint from the worklist, and for each consequence under Θ that is not already in $\mathcal{S}$, it adds that consequence both to $\mathcal{S}$ and to the worklist. The process iterates until the worklist is empty, at which point $\mathcal{S}$ is closed under Θ. The complete algorithm can be found in the first author's dissertation [Flanagan 1997].

This closure process propagates all information concerning the possible constants for labeled expressions into constraints of the form $c \leq l$. We define the set-based analysis of a program as a function that returns the set of possible constants (i.e., basic constants and function tags) for each labeled expression.

Definition 3.1. ($sba : \Lambda^0 \longrightarrow (Label \longrightarrow \mathcal{P}(Const))$). If $P \in \Lambda^0$ and $\emptyset \vdash P : \alpha, \mathcal{S}$ then

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

The set of constants C returned by $sba(P)(l)$ describes a collection of run-time values (i.e., basic constants and functions) according to the relation V **in** C. Essentially, this relation converts between function tags and tagged λ-expressions:

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

The solution $sba(P)$ conservatively approximates the value sets for each labeled expression in the program, as described by the following theorem.

THEOREM 3.2. (CORRECTNESS OF *sba*). *If* $P \longmapsto^* \mathcal{E}[\, V^l \,]$ *then* V **in** $sba(P)(l)$.

PROOF. We prove this property using a subject reduction proof, following Wright and Felleisen [1994] and Palsberg [1995]. The complete proof is contained in the first author's dissertation [Flanagan 1997]. □

4. TOWARD SIMPLIFYING CONSTRAINTS FOR PROGRAM COMPONENTS

The traditional set-based analysis 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 the large size of the constraint systems it produces for these programs. Since large programs are typically structured as a collection or a hierarchy of components (e.g., modules, classes, or functions), it is natural to try and exploit this hierarchical structure in the analysis of these programs.

To illustrate this idea, consider a program P containing a component M, where M may be a module, class, or function definition. Suppose the constraint derivation 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, say $\mathcal{S}_2$, without changing the results of the analysis. This idea is easily expressed as an additional constraint derivation rule

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}_1 \qquad \mathcal{S}_1 \sim \mathcal{S}_2}{\Gamma \vdash M : \alpha, \mathcal{S}_2} \qquad (\cong)$$

but the precise meaning of the equivalence relation $\mathcal{S}_1 \sim \mathcal{S}_2$ remains to be determined.

Since the goal of this new rule is to replace one constraint system by a simpler system without changing the behavior of the analysis as a whole, the situation is analogous to program optimization, where a compiler replaces one program fragment by a faster fragment without changing the behavior of the program as a whole. For program optimization, we use the language semantics as a source of insight for code transformations. To develop a better intuition for constraint systems, we develop a denotational semantics for constraints, study its structure, and exploit it for ideas concerning the simplification of constraints for program components.

The first subsection presents the denotational semantics of constraints. Each constraint system denotes a space of solutions for the set variables. By ranking these solutions according to their accuracy, we can characterize set-based analysis in terms of the most accurate solution, which is the topic of the second subsection. Finally, in the third and last subsection, we return to the problem of analyzing program components and state a precise version of the equivalence relation $\mathcal{S}_1 \sim \mathcal{S}_2$ in terms of the denotational semantics of constraints.

4.1 The Meaning of Set Constraints

Intuitively, a set expression τ denotes a set of values, and a constraint $\tau_1 \leq \tau_2$ denotes a corresponding set containment relationship. We formalize this meaning of constraints by mapping syntactic set expressions onto a semantic domain. We first describe the precise structure of the semantic domain, and then describe the mapping from set expressions to that domain.

The Semantic Domain. A set expression denotes a set of values. For our idealized language, a value set consists of basic constants and functions, and we therefore choose to represent it as a triple $X = \langle C, D, R \rangle$. The first component $C \in \mathcal{P}(Const)$ is a set of basic constants and function tags. The second and third components of X denote the possible arguments and results of functions in X, respectively. Since these last two components also denote value sets, the appropriate model for set expressions is the (nonempty) solution of the equation:

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

The solution $\mathcal{D}$ is equivalent to the set of all infinite binary trees with each node labeled with an element of $\mathcal{P}(Const)$.[4] We use the functions $const : \mathcal{D} \longrightarrow \mathcal{P}(Const)$ and $dom, rng : \mathcal{D} \longrightarrow \mathcal{D}$ to extract the respective components of an element of $\mathcal{D}$.

We order the elements of $\mathcal{D}$ according to a relation $\sqsubseteq$ that is antimonotonic in the domain position. We choose this ordering because information about argument values at a call site needs to flow *backward* along data-flow paths to the formal parameter of the corresponding function definitions. To illustrate this idea, consider a program that binds a function f to a program variable g. This behavior is described in the semantic domain as the inequality $X_f \sqsubseteq X_g$, where X_f and X_g describe the values sets for f and g respectively. Since the argument set for f must contain all values to which g is applied, the inequality $dom\,(X_g) \sqsubseteq dom\,(X_f)$ must also hold. Thus the domain $\mathcal{D}$ should satisfy the inference rule

$$\frac{X_f \sqsubseteq X_g}{dom\,(X_g) \sqsubseteq dom\,(X_f)},$$

which is why the ordering $\sqsubseteq$ needs to be antimonotonic in the domain element.

We define $\sqsubseteq$ as the greatest relation satisfying

$$\langle C_1, D_1, R_1 \rangle \sqsubseteq \langle C_2, D_2, R_2 \rangle \text{ iff } C_1 \subseteq C_2, D_2 \sqsubseteq D_1, R_1 \sqsubseteq R_2.$$

[4]The set $\mathcal{D}$ can be formally defined as the set of total functions $f : \{\mathtt{dom}, \mathtt{rng}\}^* \longrightarrow \mathcal{P}(Const)$, and the rest of the development can be adapted *mutandis mutatis* [Palsberg and O'Keefe 1995]. For clarity, we present our results using the more intuitive equational definition instead. Also, we can analyze languages with additional data structures by extending $\mathcal{D}$ to infinite *n-ary* trees, where n is the number of selectors (e.g.. `dom`, `rng`) corresponding to the extended language.

Under this ordering, the set $\mathcal{D}$ forms a complete lattice; the top and bottom elements and the least upper bound and greatest lower bound operations are defined recursively as

$$\begin{aligned}
\top &= \langle Const, \bot, \top \rangle \\
\bot &= \langle \emptyset, \top, \bot \rangle \\
\langle C_1, D_1, R_1 \rangle \sqcup \langle C_2, D_2, R_2 \rangle &= \langle C_1 \cup C_2, D_1 \sqcap D_2, R_1 \sqcup R_2 \rangle \\
\langle C_1, D_1, R_1 \rangle \sqcap \langle C_2, D_2, R_2 \rangle &= \langle C_1 \cap C_2, D_1 \sqcup D_2, R_1 \sqcap R_2 \rangle .
\end{aligned}$$

The Semantics of Constraints. Since set expressions contain variables, the meaning of a set expression depends on a *set environment* ρ, which defines the meaning of those variables

$$\rho \in SetEnv = SetVar \longrightarrow \mathcal{D}.$$

Given a set environment ρ, the meaning of set expressions is defined by the following unique extension of ρ from $SetVar$ to $SetExp$:

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

The semantics of constraints is easiest to define for the following full constraint language, which is an extension of the atomic constraint language considered earlier.

$$\begin{aligned}
\mathsf{C} \in \quad & Constraint = \tau_1 \leq \tau_2 \\
\mathsf{S} \in \quad & ConstraintSystem = \mathcal{P}_{\text{fin}}(Constraint)
\end{aligned}$$

We continue to use the calligraphic letters $\mathcal{C}$ and $\mathcal{S}$ as metavariables ranging over atomic constraints and systems of atomic constraints, respectively, and use the sans serif letters C and S as metavariables ranging over full constraints and full constraint systems, respectively.

We say that a set environment ρ *satisfies* a constraint $\tau_1 \leq \tau_2$ (written $\rho \models \tau_1 \leq \tau_2$) if $\rho(\tau_1) \sqsubseteq \rho(\tau_2)$. Figure 4 introduces a number of additional definitions concerning the semantics of constraint systems. The entailment relation $\models$ on constraint systems is obviously reflexive and transitive. Note that a restricted solution space $Soln(\mathsf{S}) \mid_E$ actually contains *more* set environments than in $Soln(\mathsf{S})$, since these additional environments can specify arbitrary domain elements for set variables that are *not* in E.

4.2 Ranking Solutions

A constraint system may have multiple solutions. To illustrate this idea, consider the program $P = (\lambda^t x.x)$. The constraint derivation rules of Figure 2 yield the following constraint system for P:

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

This constraint system admits the trivial solution $\rho^{\top_s}$ defined by

$$\begin{aligned}
\rho^{\top_s}(\alpha) &= \top_s \quad \forall \alpha \in SetVar \\
\top_s &= \langle Const, \top_s, \top_s \rangle .
\end{aligned}$$

Notation	Meaning	Pronunciation
$\rho \models \tau_1 \leq \tau_2$	$\rho(\tau_1) \sqsubseteq \rho(\tau_2)$	ρ satisfies $\tau_1 \leq \tau_2$
$\rho \models \mathsf{S}$	$\forall \mathsf{C} \in \mathsf{S}.\ \rho \models \mathsf{C}$	ρ satisfies S
$Soln(\mathsf{S})$	$\{\rho \mid \rho \models \mathsf{S}\}$	solution space of S
$\mathsf{S}_1 \models \mathsf{S}_2$	$Soln(\mathsf{S}_1) \subseteq Soln(\mathsf{S}_2)$	S_1 entails S_2
$\mathsf{S}_1 \cong \mathsf{S}_2$	$\mathsf{S}_1 \models \mathsf{S}_2$ and $\mathsf{S}_2 \models \mathsf{S}_1$	S_1 is observably equivalent to S_2
$Soln(\mathsf{S}) \mid_E$	$\{\rho \mid \exists \rho' \in Soln(\mathsf{S})$ such that $\rho(\alpha) = \rho'(\alpha)\ \forall \alpha \in E\}$	restriction of $Soln(\mathsf{S})$ to a collection of set variables E
$\mathsf{S}_1 \models_E \mathsf{S}_2$	$Soln(\mathsf{S}_1) \mid_E \subseteq Soln(\mathsf{S}_2) \mid_E$	S_1 entails S_2 with respect to E
$\mathsf{S}_1 \cong_E \mathsf{S}_2$	$\mathsf{S}_1 \models_E \mathsf{S}_2$ and $\mathsf{S}_2 \models_E \mathsf{S}_1$	S_1 and S_2 are observably equivalent with respect to E
$\mathsf{S} \mid_E$	$\{\mathsf{C} \in \mathsf{S} \mid \mathsf{C}$ only mentions set variables in $E\}$	restriction of S to E

Fig. 4. Notations concerning the semantics of set constraints.

The domain element $\top_s$ represents all run-time values, including functions that can take any argument and return any result. Hence, this solution is highly approximate and thus utterly useless. Fortunately, the constraint system admits a number of other solutions, including

$$\begin{aligned}
\rho_1 &= \{\, \alpha_P \mapsto \langle \{t\}, \bot, \bot \rangle, \quad \alpha_x \mapsto \bot, \alpha_M \mapsto \bot \,\} \\
\rho_2 &= \{\, \alpha_P \mapsto \langle \{t\}, \top, \top \rangle, \quad \alpha_x \mapsto \top, \alpha_M \mapsto \top \,\} \\
\rho_3 &= \{\, \alpha_P \mapsto \langle \{t, c_1\}, X, X \rangle, \alpha_x \mapsto X, \alpha_M \mapsto X \,\}
\end{aligned}$$

where $X = \langle \{c_2\}, \bot, \bot \rangle$, and c_1 and c_2 are arbitrary constants.

If we assume P to be the entire program, the function tagged t is never applied, and hence the true set of run-time values for x is simply the empty set. The solution ρ_1 describes this situation more accurately than either ρ_2 or ρ_3. Yet these three solutions are incomparable under the ordering $\sqsubseteq$, which models the flow of values through a program, but does not rank set environments according to their accuracy.

Therefore we introduce an alternative ordering $\sqsubseteq_s$ on $\mathcal{D}$ that ranks environments according to their accuracy. This ordering is monotonic in the domain position and is defined as the greatest relation satisfying

$$\langle C_1, D_1, R_1 \rangle \sqsubseteq_s \langle C_2, D_2, R_2 \rangle \text{ iff } C_1 \subseteq C_2, D_1 \sqsubseteq_s D_2, R_1 \sqsubseteq_s R_2.$$

As with the original ordering $\sqsubseteq$, the set $\mathcal{D}$ forms a complete lattice under this alternative ordering $\sqsubseteq_s$; the associated top and bottom elements and the least upper bound and greatest lower bound operations are defined recursively as

$$\begin{aligned}\top_s &= \langle Const, \top_s, \top_s\rangle \\ \bot_s &= \langle \emptyset, \bot_s, \bot_s\rangle \\ \langle C_1, D_1, R_1\rangle \sqcup_s \langle C_2, D_2, R_2\rangle &= \langle C_1 \cup C_2, D_1 \sqcup_s D_2, R_1 \sqcup_s R_2\rangle \\ \langle C_1, D_1, R_1\rangle \sqcap_s \langle C_2, D_2, R_2\rangle &= \langle C_1 \cap C_2, D_1 \sqcap_s D_2, R_1 \sqcap_s R_2\rangle.\end{aligned}$$

We extend the ordering relation $\sqsubseteq_s$ pointwise from $\mathcal{D}$ to set environments. Using the extended ordering on set environments, a system of atomic constraints $\mathcal{S}$ has both a maximal solution and a minimal solution. The maximal solution is always the set environment $\rho^{\top_s}$ described above, and we use $LeastSoln(\mathcal{S})$ to denote the minimal solution. This minimal solution exists because the greatest lower bound of the solution space (with respect to $\sqcap_s$) is also a solution.

LEMMA 4.1. *Every system of atomic constraints has a solution that is least with respect to* $\sqsubseteq_s$ *[Heintze 1994].*

PROOF. Let $\mathcal{S}$ be a system of atomic constraints, and define $\rho_{min} = \bigsqcap_s Soln(\mathcal{S})$, where $\bigsqcap_s$ is the pointwise extension of $\sqcap_s$ to set environments. We prove that $\rho_{min} \in Soln(\mathcal{S})$ by a case analysis showing that ρ_{min} satisfies each constraint $\mathcal{C} \in \mathcal{S}$. Hence ρ_{min} is the least solution of $\mathcal{S}$ with respect to $\sqsubseteq_s$. □

Using this result, we can now express the set-based analysis of a program in terms of the denotational semantics of its constraint system.

LEMMA 4.2. *If* $P \in \Lambda^0$ *and* $\emptyset \vdash P : \alpha, \mathcal{S}$ *then*

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

PROOF.

$$\begin{aligned} c \in sba(P)(l) &\iff \mathcal{S} \vdash_\Theta c \leq l && \text{by definition 3.1} \\ &\iff \mathcal{S} \models c \leq l && \text{by following lemma 4.3} \\ &\iff \forall \rho \in Soln(\mathcal{S}).\ c \in const(\rho(l)) \\ &\iff c \in \bigcap(\{const(\rho(l)) \mid \rho \in Soln(\mathcal{S})\}) \\ &\iff c \in const(\textstyle\bigsqcap_s(\{\rho \mid \rho \in Soln(\mathcal{S})\})(l)) \\ &\iff c \in const(LeastSoln(\mathcal{S})(l)) \end{aligned}$$ □

The previous proof relies on the following lemma, which states that closing a system of atomic constraints under Θ propagates all information concerning the possible constants for labeled expressions into constraints of the form $c \leq l$.

LEMMA 4.3. (SOUNDNESS AND COMPLETENESS OF Θ). *If* $\mathcal{S}$ *is a system of atomic constraints, then*

$$\mathcal{S} \vdash_\Theta c \leq \alpha \quad \iff \quad \mathcal{S} \models c \leq \alpha.$$

PROOF. See Appendix B.1. □

4.3 Conditions for Constraint System Equivalence

We now return to our original problem, which is to determine how to replace one constraint system by a simpler one without changing the analysis results. Consider again the situation where a program P contains a component M whose constraint derivation concludes

$$\Gamma \vdash M : \alpha, \mathcal{S}_1.$$

Let the context surrounding M be C, i.e., $P = C[M]$. Since the constraint derivation process is compositional, the constraint derivation for the entire program concludes

$$\emptyset \vdash P : \beta, \mathcal{S}_1 \cup \mathcal{S}_C,$$

where $\mathcal{S}_C$ is the constraint system for C. By Lemma 4.2, the results of analyzing P can only depend on the solution space for the combined constraint system $\mathcal{S}_1 \cup \mathcal{S}_C$. But this is the same as the intersection of the two respective solution spaces:

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

Hence $\mathit{Soln}(\mathcal{S}_1)$ describes at least all the properties of $\mathcal{S}_1$ relevant to the analysis, but it may also describe solutions for set variables that are not relevant to the analysis. In particular:

—$\mathit{sba}(P)$ only references the solutions for label variables; and

—an inspection of the constraint derivation rules shows that the only interactions between $\mathcal{S}_C$ and $\mathcal{S}_1$ are due to the set variables in $\{\alpha\} \cup FV[\mathit{range}(\Gamma)]$.

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

$$E = \mathit{Label} \cup \{\alpha\} \cup FV[\mathit{range}(\Gamma)].$$

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:

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

or, with the notation from Figure 4, $\mathcal{S}_1$ and $\mathcal{S}_2$ are observably equivalent on E:

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

We can now refine the constraint derivation rule $(\cong)$ based on this notion of equivalence:

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}_1 \qquad \mathcal{S}_1 \cong_E \mathcal{S}_2 \text{ where } E = \mathit{Label} \cup \{\alpha\} \cup FV[\mathit{range}(\Gamma)]}{\Gamma \vdash M : \alpha, \mathcal{S}_2} \quad (\cong)$$

The refined rule is *admissible* in that the use of this rule does not change the analysis results.

Lemma 4.4. (Admissibility of $(\cong)$). *If* $\emptyset \vdash_{\cong} P : \alpha, \mathcal{S}$ *then*

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

Proof. See Appendix B.1. □

5. THE LOGIC OF CONSTRAINT SYSTEM EQUIVALENCE

The new derivation rule $(\cong)$ involves the semantic notion of observably equivalent constraint systems. To make this rule useful, we need a strategy or algorithm for finding an observably equivalent but simpler version of a given constraint system. Because algorithms can only be based on *syntactic* entities (as opposed to *semantic* notions such as observable equivalence), our first step in the development of such an algorithm is to reformulate the observable equivalence relation as a syntactic proof system.

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

$$\alpha \leq \alpha \quad (\mathit{reflex}) \qquad \frac{\tau_1 \leq \tau \quad \tau \leq \tau_2}{\tau_1 \leq \tau_2} \quad (\mathit{trans}_\tau) \qquad \frac{\kappa_1 \leq \kappa_2}{\begin{array}{c}\mathtt{rng}(\kappa_1) \leq \mathtt{rng}(\kappa_2) \\ \mathtt{dom}(\kappa_2) \leq \mathtt{dom}(\kappa_1)\end{array}} \quad (\mathit{compat})$$

The metavariables $\kappa, \kappa_1, \kappa_2$ in (compat) range over nonconstant set expressions:

$$\kappa, \kappa_1, \kappa_2 = \alpha \mid \mathtt{dom}(\kappa) \mid \mathtt{rng}(\kappa) \ .$$

The restriction on (compat) avoids inferring useless tautologies. For example, without this restriction, the constraint $c \leq \alpha$ would yield the constraint $\mathtt{rng}(c) \leq \mathtt{rng}(\alpha)$ via (compat), which is a tautology since the range of a constant is $\perp$.

The rules (reflex) and (trans_τ) capture the reflexivity and transitivity of the ordering relation $\sqsubseteq$; (compat) expresses the monotonicity and antimonotonicity of the functions *rng* and *dom*, respectively. We write $\mathcal{S} \vdash_\Delta \mathcal{C}$ if $\mathcal{S}$ proves $\mathcal{C}$ via the rules Δ, and use $\Delta(\mathcal{S})$ to denote the closure of $\mathcal{S}$ under Δ, i.e., the set $\{\mathcal{C} \mid \mathcal{S} \vdash_\Delta \mathcal{C}\}$.

Since many of the Δ-inferred constraints lie outside of the original language of atomic constraints, we define an extended *compound constraint* language:

$$\begin{array}{rcl} \mathbf{C} \in \mathit{CmpdConstraint} &=& c \leq \kappa \mid \kappa \leq \kappa \\ \mathbf{S} \in \mathit{CmpdConSystem} &=& \mathcal{P}_{\mathrm{fin}}(\mathit{CmpdConstraint}) \end{array}$$

We use the boldface roman letters $\mathbf{C}$ and $\mathbf{S}$ as metavariables ranging over compound constraints and systems of compound constraints, respectively.

The proof system Δ completely captures the relevant properties of the ordering $\sqsubseteq$ and the functions *rng* and *dom*. That is, Δ is both sound and complete.

LEMMA 5.1. (SOUNDNESS AND COMPLETENESS OF Δ). *For a system* $\mathbf{S}$ *of compound constraints and a compound constraint* $\mathbf{C}$,

$$\mathbf{S} \vdash_\Delta \mathbf{C} \iff \mathbf{S} \models \mathbf{C}.$$

PROOF. See Appendix B.2. □

This lemma implies that $\Delta(\mathcal{S})$ contains exactly those compound constraints that hold in all environments in $\mathit{Soln}(\mathcal{S})$. Hence, if we consider a collection of external set variables E, then $\Delta(\mathcal{S})\mid_E$ contains all compound constraints that hold in all environments in $\mathit{Soln}(\mathcal{S})\mid_E$. Therefore the following lemma holds.

LEMMA 5.2. *For a system* $\mathbf{S}$ *of compound constraints,* $\mathbf{S} \cong_E \Delta(\mathbf{S})\mid_E$.

PROOF. See Appendix B.2. □

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

$$\mathcal{S}_1 \vdash^E_\Delta \mathcal{S}_2 \text{ if and only if } \Delta(\mathcal{S}_1)\mid_E \supseteq \Delta(\mathcal{S}_2)\mid_E$$

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

$$\frac{\alpha \leq \mathtt{rng}(\beta) \qquad \beta \leq \kappa}{\alpha \leq \mathtt{rng}(\kappa)} \quad (\mathit{compose}_1) \qquad\qquad \frac{\alpha \leq \mathtt{dom}(\beta) \qquad \beta \geq \kappa}{\alpha \leq \mathtt{dom}(\kappa)} \quad (\mathit{compose}_2)$$

$$\frac{\alpha \geq \mathtt{rng}(\beta) \qquad \beta \geq \kappa}{\alpha \geq \mathtt{rng}(\kappa)} \quad (\mathit{compose}_3) \qquad\qquad \frac{\alpha \geq \mathtt{dom}(\beta) \qquad \beta \leq \kappa}{\alpha \geq \mathtt{dom}(\kappa)} \quad (\mathit{compose}_4)$$

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

Fig. 5. The inference rule system Ψ.

and then show that $\mathcal{S}_1 \vdash^E_\Delta \mathcal{S}_2$ if and only if $\mathcal{S}_1 \models_E \mathcal{S}_2$. However, this definition based on the proof system Δ does not lend itself to an efficient implementation. Specifically, checking if two potential antecedents of (trans_τ) contain the same set expression τ involves comparing two potentially large set expressions. Hence we develop an alternative proof system that is more suitable for an implementation, yet infers the same constraints as Δ.

The alternative system consists of the inference rules Ψ described in Figure 5, together with the rules Θ from Figure 3. The rules (reflex) and (compat) of Ψ are those of Δ. The rules $(\mathit{compose}_{1\ldots4})$ of Ψ replace a reference to a set variable by an upper or lower (nonconstant) bound for that variable, as appropriate. The rule (trans_α) of Ψ provides a weaker characterization of transitivity than the previous rule (trans_τ), but, provided we start with a system of atomic constraints, the additional rules, Θ and $(\mathit{compose}_{1\ldots4})$, compensate for this weakness. That is, suitable combinations of these additional rules allow us to infer any constraint that could be inferred by the rule (trans_τ).

Lemma 5.3. (Equivalence of Proof Systems). *For a system of atomic constraints $\mathcal{S}$,*

$$\Delta(\mathcal{S}) = \Psi\Theta(\mathcal{S}).$$

Proof. See Appendix B.2. □

We could define a proof-theoretic equivalent of restricted entailment based on $\Psi\Theta$ as follows:

$$\mathcal{S}_1 \vdash^E_\Delta \mathcal{S}_2 \text{ if and only if } \Psi\Theta(\mathcal{S}_1)\mid_E \supseteq \Psi\Theta(\mathcal{S}_2)\mid_E,$$

but this approach is still needlessly inefficient. In particular, because (compat) does not eliminate any variables, any (compat)-consequent in $\Psi\Theta(\mathcal{S}_2)\mid_E$ is subsumed by its antecedent. Hence if we define

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

then this argument implies that the following lemma holds.

Lemma 5.4. *For any system $\mathcal{S}$ of atomic constraints, $\Psi\Theta(\mathcal{S})\mid_E \cong \Pi\Theta(\mathcal{S})\mid_E$.*

PROOF. See Appendix B.2. □

Together, Lemmas 5.2, 5.3, and 5.4 provide the basis of a proof-theoretic equivalent of restricted entailment and observable equivalence that is also suitable for implementation.

Definition 5.5. ($\vdash^E_{\Psi\Theta}$, $=^E_{\Psi\Theta}$).

(1) $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ if and only if $\Psi\Theta(\mathcal{S}_1)\mid_E \supseteq \Pi\Theta(\mathcal{S}_2)\mid_E$,

(2) $\mathcal{S}_1 =^E_{\Psi\Theta} \mathcal{S}_2$ if and only if $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ and $\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$.

The two relations $\vdash^E_{\Psi\Theta}$ and $=^E_{\Psi\Theta}$ completely characterize restricted entailment and observable equivalence of systems of atomic constraints.

THEOREM 5.6. (SOUNDNESS AND COMPLETENESS OF $\vdash^E_{\Psi\Theta}$ AND $=^E_{\Psi\Theta}$).

(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* $\mathcal{S}_1 \cong_E \mathcal{S}_2$.

PROOF. We prove the first part of this theorem as follows:

$$
\begin{array}{lll}
 & \mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2 & \\
\iff & \Psi\Theta(\mathcal{S}_1)\mid_E \supseteq \Pi\Theta(\mathcal{S}_2)\mid_E & \\
\iff & \mathit{Soln}(\Psi\Theta(\mathcal{S}_1)\mid_E) \supseteq \mathit{Soln}(\Pi\Theta(\mathcal{S}_2)\mid_E) & \\
\iff & \mathit{Soln}(\Psi\Theta(\mathcal{S}_1)\mid_E)\mid_E \supseteq \mathit{Soln}(\Pi\Theta(\mathcal{S}_2)\mid_E)\mid_E & \\
\iff & \mathit{Soln}(\Psi\Theta(\mathcal{S}_1)\mid_E)\mid_E \supseteq \mathit{Soln}(\Psi\Theta(\mathcal{S}_2)\mid_E)\mid_E & \text{by Lemma 5.4} \\
\iff & \mathit{Soln}(\Delta(\mathcal{S}_1)\mid_E)\mid_E \supseteq \mathit{Soln}(\Delta(\mathcal{S}_2)\mid_E)\mid_E & \text{by Lemma 5.3} \\
\iff & \mathit{Soln}(\mathcal{S}_1)\mid_E \supseteq \mathit{Soln}(\mathcal{S}_2)\mid_E & \text{by Lemma 5.2} \\
\iff & \mathcal{S}_1 \models_E \mathcal{S}_2 &
\end{array}
$$

The second part of this theorem follows from part 1. □

6. THE DECIDABILITY OF THE LOGIC: THE THEORY OF SIMPLIFICATION

A correct constraint simplification algorithm must preserve the observable behavior of constraint systems as defined by the proof-theoretic characterization $\mathcal{S}_1 =^E_{\Psi\Theta} \mathcal{S}_2$. We continue our search for such simplification algorithms by further investigating the properties of the relation $\mathcal{S}_1 =^E_{\Psi\Theta} \mathcal{S}_2$.

The first part of this investigation is the development of a decision algorithm for $\mathcal{S}_1 =^E_{\Psi\Theta} \mathcal{S}_2$. This decision algorithm immediately allows us to simplify constraint systems by systematically generating *all* constraint systems in order of increasing size, until we find one observably equivalent to the original system. Although this naive simplification strategy is inefficient, it does serve to highlight the relevance of the decision algorithm in solving the constraint simplification problem. In particular, the practical constraint simplification strategies of the next section are based on insights gained by the development of the decision algorithm.

We formulate the decision algorithm to work on systems of atomic constraints, since these are the constraints used in the analysis. Given two systems of atomic constraints $\mathcal{S}_1$ and $\mathcal{S}_2$, the decision algorithm needs to verify that $\Psi\Theta(\mathcal{S}_1)\mid_E = \Psi\Theta(\mathcal{S}_2)\mid_E$. The following lemma shows that the closure $\Psi\Theta(\cdot)$ can be performed in a staged manner. In particular, Π does not create any additional opportunities for rules in Θ, and (*compat*) does not create any additional opportunities for Π or Θ.

LEMMA 6.1. (STAGING). *For any system of atomic constraints $\mathcal{S}$,*

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

PROOF. See Appendix B.3. □

This lemma implies that if $\mathcal{S}_1$ and $\mathcal{S}_2$ are first closed under Θ, then the decision algorithm only 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 $\Psi(\mathcal{S}_1)\mid_E$ and $\Psi(\mathcal{S}_2)\mid_E$ does not work, since they are typically infinite. For example, if $\mathcal{S} = \{\alpha \leq \texttt{rng}(\alpha)\}$, then $\Psi(\mathcal{S})$ is the infinite set $\{\alpha \leq \texttt{rng}(\alpha), \alpha \leq \texttt{rng}(\texttt{rng}(\alpha)), \ldots\}$. Fortunately, the infinite constraint systems inferred by Ψ exhibit a regular structure, which we exploit as follows:

(1) For each $\mathcal{S}_i$ (where $i = 1$ or 2) we generate a regular grammar describing the upper and lower nonconstant bounds for each set variable in $\mathcal{S}_i$.
(2) We extend these regular grammars to regular tree grammars (RTGs) describing all constraints in $\Pi(\mathcal{S}_1)\mid_E$ and $\Pi(\mathcal{S}_2)\mid_E$. This representation allows us to use a standard RTG containment algorithm to decide if $\Pi(\mathcal{S}_1)\mid_E \supseteq \Pi(\mathcal{S}_2)\mid_E$.
(3) Based on the RTG containment algorithm, we develop a decision algorithm for the more difficult entailment question $\Psi(\mathcal{S}_1)\mid_E \supseteq \Pi(\mathcal{S}_2)\mid_E$ by allowing for the additional (*compat*) inferences on $\mathcal{S}_1$.

By checking entailment in both directions, we can decide if two constraint systems are observably equivalent. These steps are described in more detail below.

6.1 Regular Grammars

Our first step is to transform each constraint system $\mathcal{S}_i$ (for $i = 1, 2$) into a corresponding regular grammar. This regular grammar, denoted $G_r(\mathcal{S}, E)$, contains two nonterminals α_L and α_U, for each set variable α in $\mathcal{S}_i$. These nonterminals generate the following two languages of lower and upper nonconstant bounds of α, respectively:

$$\{\kappa \mid [\kappa \leq \alpha] \in \Pi(\mathcal{S}) \text{ and } SetVar(\kappa) \subseteq E\}$$
$$\{\kappa \mid [\alpha \leq \kappa] \in \Pi(\mathcal{S}) \text{ and } SetVar(\kappa) \subseteq E.\}$$

To illustrate the derivation of these grammars, consider the program component $M = (\lambda x. (x\ 1))$, and take $E = \{\alpha\}$. A simplified[5] constraint system $\mathcal{S}_M$ for M is described in Figure 6, together with the productions in the corresponding regular grammar. This grammar describes the upper and lower nonconstant bounds for each set variable in $\Pi(\mathcal{S}_M)\mid_E$. For example, the productions

$$\begin{aligned} \delta_L &\mapsto \texttt{rng}(\beta_L) \\ \beta_L &\mapsto \texttt{dom}(\alpha_U) \\ \alpha_U &\mapsto \alpha \end{aligned}$$

imply that $\delta_L \mapsto^* \texttt{rng}(\texttt{dom}(\alpha))$, or alternatively that $\texttt{rng}(\texttt{dom}(\alpha)) \leq \delta$.

[5] We use a simplified version of M's constraint system for a more concise explanation.

Constraints $\mathcal{S}_M$	Grammar $G_r(\mathcal{S}_M, E)$	Additional productions in $G_t(\mathcal{S}_M, E)$
$\mathtt{dom}(\alpha) \leq \beta$	$\beta_L \mapsto \mathtt{dom}(\alpha_U)$	
$1 \leq \gamma$		$R \mapsto [1 \leq \gamma_U]$
$\gamma \leq \mathtt{dom}(\beta)$	$\gamma_U \mapsto \mathtt{dom}(\beta_L)$	
$\mathtt{rng}(\beta) \leq \delta$	$\delta_L \mapsto \mathtt{rng}(\beta_L)$	
$\delta \leq \mathtt{rng}(\alpha)$	$\delta_U \mapsto \mathtt{rng}(\alpha_U)$	
	$\alpha_L \mapsto \alpha \quad \alpha_U \mapsto \alpha$	$R \mapsto [\alpha_L \leq \alpha_U]\ \forall \alpha \in SetVar(\mathcal{S}_M)$

Fig. 6. The constraint system, regular grammar, and RTG for $M = (\lambda x. (x\ 1))$.

The productions of the grammar are determined by $\mathcal{S}_M$ and Π. For example, the constraint $[\mathtt{rng}(\beta) \leq \delta] \in \mathcal{S}_M$ implies that for each lower bound κ of β, the rule $(compose_3)$ infers the lower bound $\mathtt{rng}(\kappa)$ of δ. Since, by induction, β's lower bounds are generated by β_L, the production $\delta_L \mapsto \mathtt{rng}(\beta_L)$ generates the corresponding lower bounds of δ.

More generally, the collection of productions

$$\{\delta_L \mapsto \mathtt{rng}(\beta_L) \mid \text{for any } \beta, \delta \text{ with } [\mathtt{rng}(\beta) \leq \delta] \in \mathcal{S}\}$$

describes all bounds inferred via $(compose_3)$ on a system $\mathcal{S}$ of atomic constraints. Bounds inferred via the remaining $(compose)$ rules can be described in a similar manner. Bounds inferred via the rule $(reflex)$ imply the productions $\alpha_U \mapsto \alpha$ and $\alpha_L \mapsto \alpha$ for $\alpha \in E$. Finally, consider the rule $(trans_\alpha)$, and suppose this rule infers an upper bound τ on α. This bound must be inferred from an upper bound τ on β, using the additional 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 $(trans_\alpha)$. Similarly, the productions $\{\beta_L \mapsto \alpha_L \mid [\alpha \leq \beta] \in \mathcal{S}\}$ generate all lower bounds inferred via $(trans_\alpha)$.

Definition 6.2. (Regular Grammar $G_r(\mathcal{S}, E)$). Let $\mathcal{S}$ be a system of atomic constraints and E a collection of set variables. The regular grammar $G_r(\mathcal{S}, E)$ consists of the nonterminals $\{\alpha_L, \alpha_U \mid \alpha \in SetVar(\mathcal{S})\}$ 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 $G_r(\mathcal{S}, E)$ generates two languages for each set variable that describe the upper and lower nonconstant bounds for that set variable. 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 nonterminal X, then the following lemma holds.

Lemma 6.3. *Let $\mathcal{S}$ be a system of atomic constraints and E a collection of set variables. If $G = G_r(\mathcal{S}, E)$, then*

$$\begin{array}{rcl}
\mathcal{L}_G(\alpha_L) & = & \{\kappa \mid [\kappa \leq \alpha] \in \Pi(\mathcal{S}) \text{ and } SetVar(\kappa) \subseteq E\} \\
\mathcal{L}_G(\alpha_U) & = & \{\kappa \mid [\alpha \leq \kappa] \in \Pi(\mathcal{S}) \text{ and } SetVar(\kappa) \subseteq E\}.
\end{array}$$

PROOF. For each language, we prove that the inclusion holds in each direction by induction on the appropriate derivation and by case analysis on the last step in that derivation. □

6.2 Regular Tree Grammars

For a system of atomic constraints $\mathcal{S}$, the grammar $G_r(\mathcal{S}, E)$ does not describe all constraints in $\Pi(\mathcal{S}) \mid_E$. In particular:

—$G_r(\mathcal{S}, E)$ does not describe constraints of the form $[c \leq \tau]$. Thus, for example, the regular grammar for $\mathcal{S}_M$ does not describe the constraint $[1 \leq \mathtt{dom}(\beta)]$ in $\Pi(\mathcal{S}_M) \mid_E$.

—$G_r(\mathcal{S}, E)$ does not describe constraints inferred by the $(trans_\alpha)$ rule that are not bounds of the form $[\kappa \leq \alpha]$ or $[\alpha \leq \kappa]$. Thus, for example, the regular grammar for $\mathcal{S}_M$ describes the constraints $\mathtt{rng}(\mathtt{dom}(\alpha)) \leq \delta$ and $\delta \leq \mathtt{rng}(\alpha)$ in $\Pi(\mathcal{S}_M) \mid_E$, but it does not describe the $trans_\alpha$-consequent $[\mathtt{rng}(\mathtt{dom}(\alpha)) \leq \mathtt{rng}(\alpha)]$ of those constraints, which is also in $\Pi(\mathcal{S}_M) \mid_E$.

For an arbitrary constraint system $\mathcal{S}$, we represent the constraint system $\Pi(\mathcal{S}) \mid_E$ by extending the grammar $G_r(\mathcal{S}, E)$ to a *regular tree grammar* $G_t(\mathcal{S}, E)$. The extended grammar combines upper and lower bounds for set variables in the same fashion as the $(trans_\alpha)$ rule and generates constraints of the form $[c \leq \tau]$ where appropriate.

Definition 6.4. (REGULAR TREE GRAMMAR $G_t(\mathcal{S}, E)$). The regular tree grammar $G_t(\mathcal{S}, E)$ extends the grammar $G_r(\mathcal{S}, E)$ with the root nonterminal R and the additional productions

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

The constructor $[\cdot \leq \cdot]$ is binary.

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

LEMMA 6.5. *Let* $G = G_t(\mathcal{S}, E)$. *Then* $\Pi(\mathcal{S}) \mid_E = \mathcal{L}_G(R)$.

PROOF. See Appendix B.3. □

The grammar $G_t(\mathcal{S}_M, E)$ for the example program component M is described in Figure 6. This grammar yields all constraints in $\Pi(\mathcal{S}_M) \mid_E$. For example, the productions

$$R \mapsto [1 \leq \gamma_U] \qquad \gamma_U \mapsto \mathtt{dom}(\beta_L) \qquad \beta_L \mapsto \mathtt{dom}(\alpha_U)$$

imply that $R \mapsto^* [1 \leq \mathtt{dom}(\mathtt{dom}(\alpha))]$, or that the argument to the function M is applied to the constant 1.

6.3 The Entailment Algorithm

We check entailment based on Lemma 6.5 as follows. Given $\mathcal{S}_1$ and $\mathcal{S}_2$, we close them under Θ and then have

$$\begin{array}{rll} & \mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1 & \\ \iff & \Psi\Theta(\mathcal{S}_2) \mid_E \supseteq \Pi\Theta(\mathcal{S}_1) \mid_E & \text{by defn } \vdash^E_{\Psi\Theta} \\ \iff & \Psi(\Theta(\mathcal{S}_2)) \mid_E \supseteq \Pi(\Theta(\mathcal{S}_1)) \mid_E & \text{by lemma 6.1} \\ \iff & \Psi(\mathcal{S}_2) \mid_E \supseteq \Pi(\mathcal{S}_1) \mid_E & \text{as } \mathcal{S}_i = \Theta(\mathcal{S}_i) \\ \iff & \mathit{compat}(\Pi(\mathcal{S}_2) \mid_E) \supseteq \Pi(\mathcal{S}_1) \mid_E & \text{by lemma 6.1} \\ \iff & \mathit{compat}(\mathcal{L}_{G_2}(R)) \supseteq \mathcal{L}_{G_1}(R) & \text{by lemma 6.5, where } G_i = G_t(\mathcal{S}_i, E). \end{array}$$

The containment question

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

can be decided via a standard RTG containment algorithm [Gécseg and Steinby 1984]. To decide the more difficult question

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

we extend the RTG containment algorithm to allow for constraints inferred via (*compat*) on the language $\mathcal{L}_{G_2}(R)$. Unfortunately, the extended algorithm takes exponential time, and hence is not directly useful in program analysis systems. Because the algorithm is mostly of theoretical interest, we defer its presentation to Appendix C.

Faster algorithms for the entailment problem may exist, but these algorithms are all in PSPACE, because the containment problem on NFAs, which is PSPACE-complete [Aho et al. 1974], can be polynomially reduced to the entailment problem on constraint systems.

Although the entailment algorithm is computationally expensive, we can still, in theory, decide if two constraint systems are observably equivalent by running the entailment algorithm in both directions. In addition, given a constraint system, we can, again, in theory, find a minimal, observably equivalent system using the naive systematic search strategy outlined in the introduction to this section. Of course, the process of computing the minimal equivalent system with this algorithm is far too expensive for use in practical program analysis systems, which is why we now turn our attention to the development of more practical constraint simplification algorithms.

7. CONSTRAINT SIMPLIFICATION: THE PRACTICE

To take advantage of the rule ($\cong$) in program analysis algorithms, we do not need a completely minimized constraint system. Any *simplifications* in a constraint system produce corresponding reductions in the time and space required for the overall analysis. Hence we concentrate on finding fast algorithms that simplify constraint systems.

For this purpose, we exploit the connection between constraint systems and RTGs. By Lemma 6.5, any transformation on constraint systems that preserves the language

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

Constraints	Production Rules	Nonempty	Reachable
$f \leq \alpha^f$	$R \mapsto [f \leq \alpha^f_U]$		
$\mathtt{dom}(\alpha^f) \leq \alpha^x$	$\alpha^x_L \mapsto \mathtt{dom}(\alpha^f_U)$		
$1 \leq \alpha^1$	$R \mapsto [1 \leq \alpha^1_U]$	$1 \leq \alpha^1$	$1 \leq \alpha^1$
$\alpha^1 \leq \mathtt{rng}(\alpha^f)$	$\alpha^1_U \mapsto \mathtt{rng}(\alpha^f_U)$		
$\mathtt{rng}(\alpha^f) \leq \alpha^a$	$\alpha^a_L \mapsto \mathtt{rng}(\alpha^f_L)$		
$\alpha^y \leq \alpha^r$	$\alpha^y_U \mapsto \alpha^r_U$ $\alpha^r_L \mapsto \alpha^y_L$	$\alpha^y \leq \alpha^r$	
$\alpha^r \leq \mathtt{dom}(\alpha^f)$	$\alpha^r_U \mapsto \mathtt{dom}(\alpha^f_L)$		
$g \leq \alpha^M$	$R \mapsto [g \leq \alpha^M_U]$	$g \leq \alpha^M$	$g \leq \alpha^M$
$\mathtt{dom}(\alpha^M) \leq \alpha^y$	$\alpha^y_L \mapsto \mathtt{dom}(\alpha^M_U)$	$\mathtt{dom}(\alpha^M) \leq \alpha^y$	
$\alpha^a \leq \mathtt{rng}(\alpha^M)$	$\alpha^a_U \mapsto \mathtt{rng}(\alpha^M_U)$	$\alpha^a \leq \mathtt{rng}(\alpha^M)$	$\alpha^a \leq \mathtt{rng}(\alpha^M)$
$\alpha^r \leq \alpha^x$	$\alpha^r_U \mapsto \alpha^x_U$ $\alpha^x_L \mapsto \alpha^r_L$	$\alpha^r \leq \alpha^x$	
$\alpha^1 \leq \alpha^a$	$\alpha^1_U \mapsto \alpha^a_U$ $\alpha^a_L \mapsto \alpha^1_L$	$\alpha^1 \leq \alpha^a$	$\alpha^1 \leq \alpha^a$
$1 \leq \alpha^a$	$R \mapsto [1 \leq \alpha^a_U]$	$1 \leq \alpha^a$	$1 \leq \alpha^a$
	$\alpha^M_L \mapsto \alpha^M$ $\alpha^M_U \mapsto \alpha^M$		

Fig. 7. The constraints, RTG and simplified constraints for $M = (\lambda^g y.((\lambda^f x.1)\ y))$.

also preserves the observable behavior of $\mathcal{S}$ with respect to E. Based on this observation, we have adapted a variety of existing algorithms for simplifying regular tree grammars 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 nonterminals 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 too small to trigger the cubic time behavior of the analysis.

7.1 Removing Empty Constraints

A nonterminal X is *empty* if $\mathcal{L}_G(X) = \emptyset$. Similarly, a production is *empty* if it refers to empty nonterminals, 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.

Let us illustrate this idea with the program component $M = (\lambda^g y.((\lambda^f x.1)\ y))$. Although this example is simplistic, it illustrates the behavior of our simplification algorithms. The solved constraint system $\mathcal{S}_M$ for M is shown in Figure 7, together with the corresponding grammar $G_t(\mathcal{S}_M, E)$ where $E = \{\alpha^M\}$. An inspection of this grammar shows that the set of nonempty nonterminals is

$$\{\alpha^M_L, \alpha^M_U, \alpha^y_L, \alpha^a_U, \alpha^r_L, \alpha^1_U, \alpha^x_L, R\}.$$

Five of the constraints in $\mathcal{S}_M$ are empty and are removed by this simplification algorithm, yielding a simplified system of eight nonempty constraints.

(1) Use a variant of Hopcroft's algorithm [Hopcroft 1971] 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 $[\alpha \leq \texttt{rng}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha' \; \exists \beta \sim \beta'$ such that $[\alpha' \leq \texttt{rng}(\beta')] \in \mathcal{S}$.
   (d) If $[\texttt{rng}(\alpha) \leq \beta] \in \mathcal{S}$ then $\forall \alpha \sim \alpha' \; \exists \beta \sim \beta'$ such that $[\texttt{rng}(\alpha') \leq \beta'] \in \mathcal{S}$.
   (e) If $[\alpha \leq \texttt{dom}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha' \; \forall \beta \sim \beta'$ such that $[\alpha' \leq \texttt{dom}(\beta')] \in \mathcal{S}$.

(2) Merge set variables according to their equivalence class.

Fig. 8. The *Hopcroft* algorithm.

7.2 Removing Unreachable Constraints

A nonterminal X is *unreachable* if there is no production $R \mapsto [Y \leq Z]$ or $R \mapsto [Z \leq Y]$ such that $\mathcal{L}_G(Y) \neq \emptyset$ and $Z \rightarrow_G^* p(X)$. Similarly, a production is *unreachable* if it refers to unreachable nonterminals; 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 nonterminals are α_U^1, α_U^a, and α_U^g. Three of the constraints are unreachable and are removed by this algorithm, yielding a simplified system with five reachable constraints.

7.3 Removing ϵ-Constraints

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

$$\{1 \leq \alpha^a,\ \alpha^a \leq \texttt{rng}(\alpha^M),\ g \leq \alpha^M\}.$$

For this example, this algorithm yields the smallest system of atomic constraints that is observably equivalent to the original system $\Theta(\mathcal{S})$.

7.4 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

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

Fig. 9. Behavior of the constraint simplification algorithms.

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,[6] we can show that an equivalence relation $\sim$ is a valid unifier for $\mathcal{S}$ if for all solutions $\rho \in Soln(\mathcal{S})$ there exists another solution $\rho' \in Soln(\mathcal{S})$ 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 8 describes sufficient conditions to ensure that ρ' generated in this manner 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 $O(n \log n)$ time algorithm [Hopcroft 1971] for computing an equivalence relation on states in a DFA and then merge set variables according to their equivalence class.[7]

The following theorem states that this simplification algorithm preserves the observable behavior of constraint systems.

Theorem 7.1. (Correctness of the Hopcroft Algorithm). *Let $\mathcal{S}$ be a system of atomic constraints with external variables E; let $\sim$ be an equivalence relation on the set variables in a constraint system $\mathcal{S}$ satisfying conditions (a) to (e) from Figure 8; let the substitution f map each set variable to a representation element of its equivalence class; and let $\mathcal{S}' = f(\mathcal{S})$, i.e., $\mathcal{S}'$ denotes the constraint system $\mathcal{S}$ with set variables merged according to their equivalence class. Then $\mathcal{S} \cong_E \mathcal{S}'$.*

Proof. Let ρ be a solution of $\mathcal{S}$. Define ρ' by

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

[6] Unlike the previous simplification algorithms, the development of this algorithm does not exploit the connection between constraint systems and RTGs.

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

The set environments ρ and ρ' agree on E by condition (a) on $\sim$, and we can show that $\rho' \models \mathcal{C}$ for all $\mathcal{C} \in \mathcal{S}$ by a case analysis on $\mathcal{C}$. $\square$

7.5 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 in the sequence also implements the preceding simplification algorithms.

We tested the algorithms on the constraint systems for nine program components. These experiments were run on a 167MHz Sparc Ultra 1 with 326MB of memory, using the MzScheme byte-code compiler [Flatt 1997]. The results are described in Figure 9. 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 of our simplification algorithms. The resulting constraint systems are typically at least an order of magnitude smaller than the original system. As expected, the more sophisticated algorithms are more effective, but are also more expensive.

8. COMPONENTIAL SET-BASED ANALYSIS

Equipped with the simplification algorithms, we can now turn our attention to our original problem: extending set-based analysis to handle significantly larger programs. These larger programs are typically constructed as a collection of program components. Exploiting this component-based structure is the key to analyzing such programs efficiently.

Componential set-based analysis 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, and the component's 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.

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

($*$ indicates the analysis exhausted heap space)

Fig. 10. Behavior of the modular analyses.

(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.

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 far less space than Heintze's set-based analysis. Finally, it recreates as much precision as traditional set-based analysis as needed on a per-component basis.

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

8.1 Experimental Results

We implemented four variants of componential set-based analysis. Each analysis uses a particular simplification algorithm from Section 7 to simplify the constraint systems for the program components. We tested these 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 results are documented in Figure 10.

The benchmark programs were written in the MzScheme [Flatt 1997] dialect of Scheme, which has a module system [Flatt and Felleisen 1998]. The componential analyses treated each `module` as a separate component. The analyses handled library primitives (such as *car*) in a polymorphic manner according to the constraint derivation rules (*let*) and (*inst*), in order to avoid merging information between unrelated calls to these functions. Treating these primitives in a monomorphic manner would have produced an analysis that was overly coarse and hence not very useful for our intended application (static debugging). In this experiment, user-defined functions were analyzed in a monomorphic manner; the following subsection describes the use of constraint simplification in a polymorphic analysis.

The fourth column in the figure shows the maximum size of the constraint system generated by each analysis and 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 analyze more programs, such as `sba` and `poly`, for which the *standard* analysis exhausted the available heap space.

The fifth column shows the time required to analyze each program from scratch, without using any existing constraint files.[8] 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 trade-off may change as we analyze yet larger programs.

To test the responsiveness of the componential analyses in an interactive setting based on an analyze-debug-edit cycle, we reanalyzed each benchmark after changing a randomly chosen component in that benchmark. The reanalysis times are shown in the sixth column of Figure 10. Even in the absence of constraint simplification, there is some advantage in caching the intermediate constraint systems between runs, but much of this advantage is lost in the time taken to read and write large constraint files. In the presence of constraint simplification, this approach yields 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 debugging sessions with MrSpidey typically require analyzing the project many times, e.g., when a bug is identified and eliminated, using separate analysis substantially improves the usability of MrSpidey.

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

[8] These times exclude scanning and parsing time.

Program	lines	*copy* analysis	Relative time of *smart* polymorphic analyses: *empty*	*unreachable*	ϵ-*removal*	*Hopcroft*	Mono. analysis
`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)

Fig. 11. Times for the smart polymorphic analyses, relative to the *copy* analysis.

9. EFFICIENT POLYMORPHIC ANALYSIS

The constraint simplification algorithms also enable 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*).[9] 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 [Jagannathan and Wright 1995]. The results of the test runs are documented in Figure 11. 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 reuse 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.

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

10. RELATED WORK

Our results were developed in 1995–1996 and published at PLDI'97. In the meantime, a number of researchers have investigated the problem of constraint simplification in order to derive faster and more scalable analyses and type systems.

Deutsch and Heintze [1996] examine constraint simplification for set-based analysis. They propose two simplification algorithms, which are analogous to our empty and unreachable constraint simplification algorithms, but do not present results on the cost or effectiveness of these simplification algorithms.

Fähndrich and Aiken [1996] 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 program analyses.

Pottier [1996] studies an ML-style language with subtyping. Performing type inference on this language produces subtype constraints that are similar to our constraints. Pottier defines an entailment relation on constraints, and presents an incomplete algorithm for deciding entailment. In addition, he proposes some *ad hoc* algorithms for simplifying constraints. He does not report any results on the cost or effectiveness of these algorithms.

Trifonov and Smith [1996] describe a subtyping relation between constrained types, which are similar to our constraint systems, and they present an incomplete decision algorithm for subtyping. They do not discuss constraint simplification. Eifrig et al. [1995] discuss constraint simplification in the context of type inference for objects. They present three algorithms for simplifying constraint systems, two of 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 [1994] 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 or virtual methods. The article does not present results on the cost or effectiveness of these algorithms.

11. FUTURE WORK

All our constraint simplification algorithms preserve the observable behavior of constraint systems, and thus do not effect the accuracy of the analysis. If we were willing to tolerate less accurate analysis results, we could choose a compressed constraint system that does not preserve the observable behavior of the original system, but only *entails*, or conservatively approximates, that behavior. This approach could yield significant savings in both time and space.

A promising approach for deriving such approximate constraint systems is to rely on a programmer-provided *signature* describing the behavior of a 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 component, we could use those constraints in the remainder of the analysis. Since the signature-based constraints are expected to be 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.

APPENDIX

A. EXTENDING SET-BASED ANALYSIS

Realistic programming languages provide a variety of additional facilities on top of the idealized core language Λ. These facilities typically include compound data structures, assignments, and nonlocal control operators such as first-class continuations or exceptions. This section discusses the extension of set-based analysis to encompass these additional features of practical programming languages. This extension also suggests how componential set-based analysis can be adapted to other safe languages such as Java.

A.1 Additional Selectors

Most of the additional programming constructs mentioned above introduce additional kinds of values into the language. Modeling these additional values in the analysis requires the introduction of additional selectors into the constraint language and the corresponding extension of the underlying domain $\mathcal{D}$ and the set of operations and relations defined on $\mathcal{D}$.

To simplify this process, we first abstract over the collection of selectors in the constraint language. The constraint language currently contains a single monotonic selector, **rng**, and a single antimonotonic selector, **dom**. We generalize the constraint language with two sets, Sel^+ and Sel^-, of monotonic and antimonotonic selectors, respectively, which are currently defined as the singletons:

$$\begin{aligned} Sel^+ &= \{\texttt{rng}\} \\ Sel^- &= \{\texttt{dom}\} \end{aligned}$$

We use the metavariables $\texttt{sel}^+$, $\texttt{sel}^-$, and $\texttt{sel}$ to range over selectors in Sel^+, Sel^-, and $Sel^+ \cup Sel^-$, respectively. Expressed in terms of these metavariables, the language of set expressions becomes

$$\tau \in SetExp = \alpha \mid c \mid \texttt{sel}^+(\tau) \mid \texttt{sel}^-(\tau),$$

and an *atomic constraint* is of the form

$$\begin{array}{rcrcr} \mathcal{C} \in AtomicCon = & & c \leq \beta & \mid & \alpha \leq \beta \\ & \mid & \alpha \leq \texttt{sel}^-(\beta) & \mid & \texttt{sel}^-(\alpha) \leq \beta \\ & \mid & \alpha \leq \texttt{sel}^+(\beta) & \mid & \texttt{sel}^+(\alpha) \leq \beta. \end{array}$$

These constraints have their expected semantics on an extended domain $\mathcal{D}$ that contains a product for each selector in the constraint language:

$$\mathcal{D} = \mathcal{P}(Const) \times \underbrace{\mathcal{D} \times \cdots \times \mathcal{D}}_{\texttt{sel}^- \in Sel^-} \times \underbrace{\mathcal{D} \times \cdots \times \mathcal{D}}_{\texttt{sel}^+ \in Sel^+}$$

This reformulation simplifies the process of extending the analysis to cope with additional programming constructs. The remainder of the derivation of the analysis

$$\frac{\Gamma \vdash M_i : \alpha_i, \mathcal{S}_i}{\Gamma \vdash (\textbf{cons}\ M_1\ M_2) : \beta, \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\alpha_1 \leq \texttt{car}(\beta), \alpha_2 \leq \texttt{cdr}(\beta)\}} \quad (cons)$$

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash (\textbf{car}\ M) : \beta, \mathcal{S} \cup \{\texttt{car}(\alpha) \leq \beta\}} \quad (car)$$

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash (\textbf{cdr}\ M) : \beta, \mathcal{S} \cup \{\texttt{cdr}(\alpha) \leq \beta\}} \quad (cdr)$$

Fig. 12. Constraint derivation rules for pairs.

can be adapted to the modified formulation, *mutandis mutatis*.

A.2 Analysis of Pairs

Let Λ^p be the following extension of Λ with immutable pairs.

$$\begin{array}{rcl} M \in \Lambda^p & = & \ldots \mid (\textbf{cons}\ M\ M) \mid (\textbf{car}\ M) \mid (\textbf{cdr}\ M) \\ V \in \mathit{Value} & = & \ldots \mid (\textbf{cons}\ V\ V) \end{array}$$

Semantics. The additional syntactic forms have their usual Scheme semantics, which we formalize with two additional notions of reduction

$$\begin{array}{rcll} (\textbf{car}\ (\textbf{cons}\ V_1\ V_2)) & \longrightarrow & V_1 & (car) \\ (\textbf{cdr}\ (\textbf{cons}\ V_1\ V_2)) & \longrightarrow & V_2 & (cdr) \end{array}$$

and with an extended notion of evaluation contexts

$$\mathcal{E} = \ldots \mid (\textbf{cons}\ \mathcal{E}\ M) \mid (\textbf{cons}\ V\ \mathcal{E}) \mid (\textbf{car}\ \mathcal{E}) \mid (\textbf{cdr}\ \mathcal{E}).$$

The standard reduction relation $\longmapsto$ and the evaluator *eval* for the extended language Λ^p is defined in the usual manner, following Section 2.

Analysis. The analysis of the extended language Λ^p requires two additional monotonic selectors `car` and `cdr`:

$$\begin{array}{rcl} \mathit{Sel}^+ & = & \{\texttt{rng},\ \texttt{car},\ \texttt{cdr}\} \\ \mathit{Sel}^- & = & \{\texttt{dom}\} \end{array}$$

These additional selectors yield corresponding products in the domain $\mathcal{D}$. Each element $X \in \mathcal{D}$ is now a 5-tuple $\langle C, D, R, A_1, A_2\rangle$, where the additional components A_1 and A_2 describe the possible `car` and `cdr` fields of pairs represented by X. We extend the relation V **in** X to describe the pairs represented by an element $X = \langle C, D, R, A_1, A_2\rangle$ in $\mathcal{D}$ as follows:

$$\begin{array}{rcl} b\ \textbf{in}\ X & \text{iff} & b \in C \\ (\lambda^t x.M)\ \textbf{in}\ X & \text{iff} & t \in C \\ (\textbf{cons}\ V_1\ V_2)\ \textbf{in}\ X & \text{iff} & V_1\ \textbf{in}\ A_1, V_2\ \textbf{in}\ A_2 \end{array}$$

The constraint derivation rules for the new syntactic forms are described in Figure 12. The rule (*cons*) records the possible values for each component of the pair. The rules (*car*) and (*cdr*) extract the appropriate component from the set variable for the argument expression.

The set-based analysis function *sba* for the extended language Λ^p is defined following Definition 3.1. As in Section 3.3, we can compute $sba(P)$ from the closure of $\mathcal{S}$ under Θ:

$$\begin{array}{rl} sba(P)(l) = & \{b \mid \mathcal{S} \vdash_\Theta b \leq l\} \\ & \cup\ \{(\lambda^t x.M) \mid \mathcal{S} \vdash_\Theta t \leq l\} \\ & \cup\ \{(\mathbf{cons}\ V_1\ V_2) \mid \ \mathcal{S} \vdash_\Theta \alpha_1 \leq \mathtt{car}(l),\ V_1 \in sba(P)(\alpha_1) \\ & \qquad\qquad\qquad\qquad \mathcal{S} \vdash_\Theta \alpha_2 \leq \mathtt{cdr}(l),\ V_2 \in sba(P)(\alpha_2)\} \end{array}$$

A.3 Analysis of First-Class Continuations

Consider the following language Λ^{cc}, which extends Λ^p with first-class continuations:

$$M \in \Lambda^{cc} = \ \ldots \mid (\mathbf{abort}\ M) \mid (\mathbf{callcc}^t\ M)$$

An **abort**-expression evaluates its subexpression, and returns the resulting value as the result of the entire computation. The **callcc**-expression $(\mathbf{callcc}^t\ M)$ first evaluates its argument M to a function, then *captures* the current evaluation context (or *continuation*) surrounding the expression, and applies the function produced by M to this evaluation context. An invocation of a captured evaluation context causes the current evaluation context to be discarded and replaced by the captured context. Just like a function expression, a **callcc**-expression has an identifying tag so that MrSpidey can reconstruct the textual source of the corresponding continuation values from the results of the analysis.

Semantics. We define the semantics of the **abort** and **callcc** constructs by extending the standard reduction relation with the following rules for aborting and capturing evaluation contexts:

$$\begin{array}{rcll} \mathcal{E}[\ (\mathbf{abort}\ M)\ ] & \longmapsto & M & \quad (\mathit{abort}) \\ \mathcal{E}[\ (\mathbf{callcc}^t\ M)\ ] & \longmapsto & \mathcal{E}[\ (M\ (\lambda^t x.(\mathbf{abort}\ \mathcal{E}[\ x\ ])))\ ] & \quad (\mathit{callcc}) \end{array}$$

The evaluator for the extended language is defined in the usual manner, following Section 2.

Analysis. Figure 13 introduces the additional derivation rules for **abort** and **callcc** expressions. An **abort** expression never returns, so the derivation rule (*abort*) introduces a fresh type variable for these expressions. The least solution (under $\sqsubseteq_s$) for this type variable is $\perp_s$, denoting the empty set of values.

The rule (*callcc*) introduces a new type variable δ to denote the captured continuation. The rule records that

(1) the type variable δ contains the tag t from the **callcc** expression;

(2) δ is the argument to the function (denoted by α) that is returned by M;

(3) the result value this function becomes the result of the **callcc** expression;

(4) argument values to δ are also returned as results of the **callcc** expression.

In addition, the rule adds the "dummy" constraint $\gamma \leq \mathtt{rng}(\delta)$. This dummy constraint is required in order that the constraint derivation rules satisfy the subject reduction lemma. That is, the (*callcc*) reduction rule produces a contractum containing the syntactic term $(\lambda^t x.(\mathbf{abort}\ \mathcal{E}[\ x\ ]))$, which is not present in the (*callcc*)-redex. Applying the constraint derivation rules to this contractum yields a number

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash (\textbf{abort}\ M) : \beta, \mathcal{S}} \qquad (abort)$$

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash (\textbf{callcc}^t\ M) : \beta, \mathcal{S} \cup \left\{ \begin{array}{rcl} t & \leq & \delta \\ \delta & \leq & \texttt{dom}(\alpha) \\ \texttt{rng}(\alpha) & \leq & \beta \\ \texttt{dom}(\delta) & \leq & \beta \\ \gamma & \leq & \texttt{rng}(\delta) \end{array} \right\}} \qquad (callcc)$$

Fig. 13. Constraint derivation rules for first-class continuations.

of constraints, including the constraint $\gamma \leq \texttt{rng}(\delta)$, where γ describes the value set for $(\textbf{abort}\ \mathcal{E}[\ x\ ])$, and δ describes the value set for the λ-expression. The subject reduction lemma requires that this constraint is entailed by the constraint system for the (*callcc*)-redex. In order to satisfy this requirement, we include that constraint in the redex's constraint system.

The set-based analysis function *sba* for the extended language Λ^{cc} is defined as in Definition 3.1 and is computed in the usual fashion based on the closure of the derived constraint system under Θ.

A.4 Analysis of Assignable Variables

Next we consider the set-based analysis of a language with assignable variables. Let $\Lambda^!$ be the following extension of Λ^p:

$$\begin{array}{rcll} M \in \Lambda^! & = & \ldots \mid (\textbf{letrec}\ (D^*)\ M) \mid (\textbf{set!}\ z\ M) \mid z & \text{(Expressions)} \\ D \in \mathit{Defines} & = & (\textbf{define}\ z\ V) & \text{(Definitions)} \\ z \in \mathit{AssignVar} & & & \text{(Assign. Variables)} \end{array}$$

The extended language contains assignable variables, in addition to the regular, immutable variables. These assignable variables are introduced by a **letrec**-expression $(\textbf{letrec}\ (D^*)\ M)$, where D^* is a sequence of definitions of the form $(\textbf{define}\ z\ V)$. Each assignable variable in D^* is bound in the entire **letrec**-expression, and we work with the usual conventions concerning α-renaming for assignable variables. An assignment expression $(\textbf{set!}\ z\ M)$ first evaluates M to some value, assigns the variable z to that value, and then returns the value.

Semantics. We evaluate programs within an enclosing **letrec** containing a *heap* and an expression. The *heap* is a sequence of definitions containing all currently defined assignable variables:

$$H \in \mathit{Heap} = D^*$$

All references and assignments to assignable variables operate on this heap. We use the functional notation $H(z)$ to extract the value bound to z in the heap H.

To allow the evaluation of subexpressions inside the **set!** form, we extend the notion of evaluation contexts:

$$\mathcal{E} = \ldots \mid (\textbf{set!}\ z\ \mathcal{E})$$

$$\frac{\begin{array}{c}\Gamma \cup \{z_i : \alpha_i\} \vdash V_i : \beta_i, \mathcal{S}_i \\ \Gamma \cup \{z_i : \alpha_i\} \vdash M : \gamma, \mathcal{S}\end{array}}{\begin{array}{c}\Gamma \vdash (\textbf{letrec}\ ((\textbf{define}\ z_1\ V_1) \ldots (\textbf{define}\ z_n\ V_n))\ M) \\ : \gamma, \mathcal{S} \cup \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n \cup \{\beta_i \leq \alpha_i \mid 1 \leq i \leq n\}\end{array}} \quad (\mathit{letrec})$$

$$\Gamma \cup \{z : \beta\} \vdash z : \alpha, \mathcal{S} \cup \{\alpha \leq \beta\} \quad (\mathit{ref})$$

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash (\textbf{set!}\ z\ M) : \alpha, \mathcal{S} \cup \{\alpha \leq \Gamma(z)\}} \quad (\mathit{set!})$$

Fig. 14. Constraint derivation rules for assignable variables.

We extend the standard reduction relation with the following additional cases for the new syntactic forms. To evaluate an internal **letrec**, we lift its definitions out into the global heap, ensuring that the appropriate hygiene conditions are satisfied:

$$\begin{array}{llr}
(\textbf{letrec}\ (H)\ \mathcal{E}[\ (\textbf{letrec}\ (D^*)\ M)\ ]) & & \\
\quad \longmapsto (\textbf{letrec}\ (H \cup D^*)\ \mathcal{E}[\ M\ ]) & & (\mathit{letrec}) \\
\quad \text{if } \mathit{dom}(H) \cap \mathit{dom}(D^*) = \emptyset & & \\
(\textbf{letrec}\ (H)\ \mathcal{E}[\ z\ ]) \longmapsto (\textbf{letrec}\ (H)\ \mathcal{E}[\ V\ ]) & \text{if } H(z) = V & (\mathit{ref}) \\
(\textbf{letrec}\ (H \cup (\textbf{define}\ z\ V))\ \mathcal{E}[\ (\textbf{set!}\ z\ V')\ ]) & & \\
\quad \longmapsto (\textbf{letrec}\ (H \cup (\textbf{define}\ z\ V'))\ \mathcal{E}[\ V'\ ]) & & (\mathit{set!}) \\
(\textbf{letrec}\ (H)\ \mathcal{E}[\ M\ ]) \longmapsto (\textbf{letrec}\ (H)\ \mathcal{E}[\ M'\ ]) & \text{if } M \longrightarrow M' & (\mathit{compat})
\end{array}$$

The semantics of the extended language is defined via the partial function *eval* on programs. This evaluator now returns a pair consisting of a heap and a value, where the heap provides bindings for the assignable variables in the value.

$$\begin{array}{lcll}
\mathit{eval} : \Lambda^0 & \longrightarrow\!\!\!\!\!\circ & \mathit{Heap} \times \mathit{Value} & \\
\mathit{eval}(M) & = & \langle H, V \rangle & \text{if } (\textbf{letrec}\ ()\ M) \longmapsto^* (\textbf{letrec}\ (H)\ V)
\end{array}$$

Analysis. The analysis of $\Lambda^!$ is based on the additional constraint derivation rules described in Figure 14. The rule (*letrec*) extends the derivation context Γ to map each assignable variable z_i to a fresh set variable α_i and generates constraints for both the defined values and the **letrec**-body using the extended derivation context. The rule (*set!*) propagates all possible assigned values into the value set for the assigned variable. A constraint derivation context now maps immutable variables to either set variables or constraint schemas, as before, and now also maps assignable variables to set variables.

The set-based analysis function *sba* for the extended language $\Lambda^!$ can be defined and computed in the usual fashion.

B. PROOFS

B.1 Proofs for Section 4

LEMMA 4.3. (SOUNDNESS AND COMPLETENESS OF Θ). *If $\mathcal{S}$ is a system of atomic constraints, then*

$$\mathcal{S} \vdash_\Theta c \leq \alpha \quad \Longleftrightarrow \quad \mathcal{S} \models c \leq \alpha.$$

PROOF. The soundness of Θ is straightforward. To prove the completeness of Θ, assume $\mathcal{S} \models c \leq \alpha$. Let ρ be any fixpoint of the functional F defined as

$$\begin{array}{rcl} F : \mathit{SetEnv} & \longrightarrow & \mathit{SetEnv} \\ F(\rho)(\alpha) & = & \langle\, \{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha\}, \\ & & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \alpha \leq^* \delta,\ \gamma \leq \mathtt{dom}(\delta)\}, \\ & & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha)\}\ \rangle \end{array}$$

where $\mathcal{S} \vdash_\Theta \alpha \leq^* \delta$ means there exists some $\delta_1, \ldots, \delta_n$ with $\alpha \equiv \delta_1$ and $\delta_n \equiv \delta$ such that

$$\mathcal{S} \vdash_\Theta \delta_i \leq \delta_{i+1} \qquad \text{for } 1 \leq i < n.$$

Note. The asymmetry between the definition of the domain and range components of $F(\rho)(\alpha)$ arises from the rules Θ. The rule (s_2) propagates set variables denoting results of functions in α forward along data-flow paths into constraints of the form $\gamma \leq \mathtt{rng}(\alpha)$. However, the same propagation does not occur for set variables denoting arguments to functions in α. Hence this propagation is performed in the definition of $F(\rho)(\alpha)$ by finding all γ such that $\gamma \leq \mathtt{dom}(\delta)$ and $\alpha \leq^* \delta$.

If $\rho \models \mathcal{S}$, then $\rho \models c \leq \alpha$, and hence $\mathcal{S} \vdash_\Theta c \leq \alpha$ by the definition of ρ, as required. Thus it just remains to prove that $\rho \models \mathcal{S}$. We proceed by case analysis on constraints $\mathcal{C} \in \mathcal{S}$.

—Suppose $\mathcal{C} = [\alpha \leq \beta]$. We need to show that the correct ordering holds between the components of $\rho(\alpha)$ and $\rho(\beta)$. For the first component, since $\mathcal{C} \in \mathcal{S}$, by (s_1) we have

$$\begin{array}{lrcl} & \{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha\} & \subseteq & \{c \mid \mathcal{S} \vdash_\Theta c \leq \beta\} \\ \therefore & \mathit{const}\,(\rho(\alpha)) & \subseteq & \mathit{const}\,(\rho(\beta))\,. \end{array}$$

For the second (domain) component, by (s_1)

$$\begin{array}{lrcl} & \mathcal{S} \vdash_\Theta \beta \leq^* \delta & \Rightarrow & \mathcal{S} \vdash_\Theta \alpha \leq^* \delta \\ \therefore & \{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \beta \leq^* \delta,\ \gamma \leq \mathtt{dom}(\delta)\} & \subseteq & \{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \alpha \leq^* \delta,\ \gamma \leq \mathtt{dom}(\delta)\} \\ \therefore & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \beta \leq^* \delta,\ \gamma \leq \mathtt{dom}(\delta)\} & \sqsubseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \alpha \leq^* \delta,\ \gamma \leq \mathtt{dom}(\delta)\} \\ \therefore & \mathit{dom}\,(\rho(\beta)) & \sqsubseteq & \mathit{dom}\,(\rho(\alpha))\,. \end{array}$$

For the third (range) component, by (s_2)

$$\begin{array}{lrcl} & \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha) & \Rightarrow & \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta) \\ \therefore & \{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha)\} & \subseteq & \{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta)\} \\ \therefore & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha)\} & \sqsubseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta)\} \\ \therefore & \mathit{rng}\,(\rho(\alpha)) & \sqsubseteq & \mathit{rng}\,(\rho(\beta))\,. \end{array}$$

Hence $\rho(\alpha) \sqsubseteq \rho(\beta)$.

—Suppose $\mathcal{C} = [\mathtt{dom}(\alpha) \leq \beta]$.

$$\begin{array}{rcll} \rho(\mathtt{dom}(\alpha)) & = & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \alpha \leq^* \delta,\ \gamma \leq \mathtt{dom}(\delta)\} & \text{by definition of } \rho \\ & \sqsubseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \mathtt{dom}(\delta) \leq \beta,\ \gamma \leq \mathtt{dom}(\delta)\} & \text{by } (s_3) \\ & \sqsubseteq & \bigsqcup_s\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \beta\} & \text{by } (s_5) \\ & \sqsubseteq & \rho(\beta) & \end{array}$$

—The remaining cases for $\mathcal{C}$ hold by similar reasoning.

Hence $\rho \models \mathcal{S}$, and the lemma holds. □

The following lemma demonstrates that the rule $(\cong)$ is *admissible* in that any derivation in the extended constraint derivation system produces information equivalent to that produced by the original analysis.

LEMMA 4.4. (ADMISSIBILITY OF $(\cong)$). *If* $\emptyset \vdash_\cong P : \alpha, \mathcal{S}$ *then*

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

PROOF. This lemma follows from the property

> if $\Gamma \vdash_\cong M : \alpha, \mathcal{S}_1$, and $E = Label \cup \{\alpha\} \cup FV[range(\Gamma)]$, then there exists $\mathcal{S}_2$ such that $\Gamma \vdash M : \alpha, \mathcal{S}_2$ and $\mathcal{S}_1 \cong_E \mathcal{S}_2$.

We prove this hypothesis by induction on the derivation $\Gamma \vdash_\cong M : \alpha, \mathcal{S}_1$, and by case analysis on the last step in the derivation.

—Suppose $\Gamma \vdash_\cong M : \alpha, \mathcal{S}_1$ via $(\cong)$ because $\Gamma \vdash_\cong M : \alpha, \mathcal{S}_3$ and $\mathcal{S}_3 \cong_E \mathcal{S}_1$. By induction, $\Gamma \vdash M : \alpha, \mathcal{S}_4$ where $\mathcal{S}_3 \cong_E \mathcal{S}_4$. Since $\cong_E$ is an equivalence relation, $\mathcal{S}_1 \cong_E \mathcal{S}_2$, and hence the lemma holds.

—Otherwise the proof of each case holds based on the induction hypothesis. □

B.2 Proofs for the Logic of Constraint System Equivalence

The following proofs require a number of auxiliary definitions.

Definition B.3. (PATHS).

—A *path* $p, q, r \in Path$ is a sequence of the constructors `dom` and `rng`. We use ϵ to denote the empty sequence, and $p.q$ to denote the concatenation of the paths p and q.

—The *arity* of a path p, denoted πp, is the number of `dom`'s in p, taken modulo 2. If πp is 0, we say p is *monotonic*; otherwise p is *antimonotonic*.

—For a path p, the notation $p(\tau)$ denotes the set expression τ enclosed in the `dom`'s and `rng`'s of p, i.e., if $p \equiv \mathtt{rng.dom}$, then $p(\alpha) \equiv \mathtt{rng}(\mathtt{dom}(\alpha))$.

—For a path p and a domain element $X \in \mathcal{D}$, the notation $p(X)$ extracts the component of X at the position p. This notation is formalized as follows:

$$\begin{array}{rcl} \epsilon(X) & = & X \\ (\mathtt{rng}.p)(X) & = & rng\,(p(X)) \\ (\mathtt{dom}.p)(X) & = & dom\,(p(X)) \end{array}$$

—The relations $\leq_0$ and $\leq_1$ denote $\leq$ and $\geq$, respectively.

—The relations $\subseteq_0$ and $\subseteq_1$ denote $\subseteq$ and $\supseteq$, respectively.

Lemma 5.1. (Soundness and Completeness of Δ). *For a system* $\mathbf{S}$ *of compound constraints and a compound constraint* $\mathbf{C}$*,*

$$\mathbf{S} \vdash_\Delta \mathbf{C} \quad \Longleftrightarrow \quad \mathbf{S} \models \mathbf{C}.$$

Proof. The soundness of Δ is straightforward. To demonstrate the completeness of Δ, we assume $\mathbf{S} \models \mathbf{C}$ and prove that $\mathbf{S} \vdash_\Delta \mathbf{C}$ by case analysis on $\mathbf{C}$.

(1) Suppose $\mathbf{C} = [c \leq \kappa]$. Define ρ by

$$\forall p \in \mathit{Path}.\ \forall \alpha \in \mathit{SetVar}.\ \mathit{const}(p(\rho(\alpha))) = \{c \mid \mathbf{S} \vdash_\Delta c \leq p(\alpha)\}.$$

This solution ρ is actually the $\mathit{LeastSoln}(\mathbf{S})$. For the purposes of this proof, however, it is sufficient to prove a weaker result, specifically that $\rho \models \mathbf{S}$. We prove this result by a case analysis showing that ρ satisfies every constraint $\mathbf{C}' \in \mathbf{S}$.

(a) Suppose $\mathbf{C}' = [c' \leq q(\beta)]$. Then, by the definition of ρ, $c' \in \mathit{const}(\rho(q(\beta)))$, and hence $\rho \models c' \leq q(\beta)$.

(b) Suppose $\mathbf{C}' = [p(\alpha) \leq q(\beta)]$. We need to show that $\rho(p(\alpha)) \sqsubseteq \rho(q(\beta))$. We prove this inequality by showing that for any path r:

$$\mathit{const}(r(\rho(p(\alpha)))) \subseteq_{\pi r} \mathit{const}(r(\rho(q(\beta)))).$$

If r is monotonic, then

$$\begin{aligned} \mathit{const}(r(\rho(p(\alpha)))) &= \{c \mid \mathbf{S} \vdash_\Delta c \leq r(p(\alpha))\} \\ &\subseteq \{c \mid \mathbf{S} \vdash_\Delta c \leq r(q(\beta))\} \\ &\quad\ \text{via } (\mathit{trans}_\tau) \text{ from } \mathbf{S} \vdash_\Delta r(p(\alpha)) \leq r(q(\beta)) \\ &\quad\ \text{which follows from } [p(\alpha) \leq q(\beta)] \in \mathbf{S} \text{ via } (\mathit{compat}) \\ &= \mathit{const}(r(\rho(q(\beta)))). \end{aligned}$$

The case where r is antimonotonic follows by a similar argument.

Hence $\rho \models \mathbf{S}$, and, in particular, $\rho \models c \leq \kappa$. Since $\kappa = p(\alpha)$ for some p and α, then we have that

$$c \in \mathit{const}(p(\rho(\alpha))) = \{c \mid \mathbf{S} \vdash_\Delta c \leq p(\alpha)\}.$$

Hence, $\mathbf{S} \vdash_\Delta c \leq \kappa$, as required.

(2) Suppose $\mathbf{C} = [\kappa_1 \leq \kappa_2]$. Let c be a constant not used in $\mathbf{S}$ or $\mathbf{C}$; let $\mathbf{S}' = \mathbf{S} \cup \{c \leq \kappa_1\}$; and let $\rho = \mathit{LeastSoln}(\mathbf{S}')$. Since $\rho \models \mathbf{C}$, we have that

$$\rho \models \{c \leq \kappa_1,\ \kappa_1 \leq \kappa_2\}.$$

Hence $\rho \models c \leq \kappa_2$, and by the first part of this proof, $\mathbf{S}' \vdash_\Delta c \leq \kappa_2$.
We now show that

$$\text{for any } \kappa',\ \mathbf{S}' \vdash_\Delta c \leq \kappa' \text{ if and only if } \mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa'.$$

The right-to-left implication is straightforward. We prove the left-to-right implication by induction on the derivation of $\mathbf{S}' \vdash_\Delta c \leq \kappa'$:

(a) Suppose $\mathbf{S}' \vdash_\Delta c \leq \kappa'$ because $[c \leq \kappa'] \in \mathbf{S}'$. Then $\kappa' = \kappa_1$ because c is unique, and $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa_1$ via (*reflex*), as required.

(b) If $[c \leq \kappa'] \notin \mathbf{S}'$, then $\mathbf{S}' \vdash_\Delta c \leq \kappa'$ must be derived via the rule (trans_τ) based on the antecedents $\mathbf{S}' \vdash_\Delta \{c \leq \kappa'', \kappa'' \leq \kappa'\}$. By induction, $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa''$. Hence $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa'$ via (trans_τ), as required.

Since $\mathbf{S}' \vdash_\Delta c \leq \kappa_2$, the above hypothesis implies that $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa_2$, as required. $\square$

LEMMA 5.2. *For a system* $\mathbf{S}$ *of compound constraints,* $\mathbf{S} \cong_E \Delta(\mathbf{S}) \mid_E$.

PROOF. We need to show that $\mathbf{S} \cong_E \Delta(\mathbf{S}) \mid_E$, i.e.,

$$Soln(\mathbf{S}) \mid_E = Soln(\Delta(\mathbf{S}) \mid_E) \mid_E.$$

Since Δ is sound

$$Soln(\mathbf{S}) \mid_E = Soln(\Delta(\mathbf{S})) \mid_E \subseteq Soln(\Delta(\mathbf{S}) \mid_E) \mid_E,$$

because the solution space increases as the constraints $\Delta(\mathbf{S})$ are restricted to E.

To show the containment in the other direction, assume $\rho \models \Delta(\mathbf{S}) \mid_E$. Without loss of generality, assume $\rho(\alpha) = \bot_s$ for all $\alpha \notin E$. We extend ρ to a superenvironment ρ' that satisfies $\mathbf{S}$ as follows:

$$\forall p \in Path.\ \forall \alpha \in SetVar.\ const(p(\rho'(\alpha))) = \bigcup\{const(\rho(\tau)) \mid \mathbf{S} \vdash_\Delta \tau \leq p(\alpha)\}$$

We show that $\rho' \models \mathbf{S}$ by case analysis on the constraints $\mathbf{C} \in \mathbf{S}$.

—Suppose $\mathbf{C} = [c \leq q(\beta)]$. Then

$$\begin{array}{rcl} const(q(\rho'(\beta))) & = & \bigcup\{const(\rho(\tau)) \mid \mathbf{S} \vdash_\Delta \tau \leq q(\beta)\} \\ & \supseteq & \{c\} \end{array}$$

as required.

—Suppose $\mathbf{C} = [p(\alpha) \leq q(\beta)]$. Then for any path r, $\mathbf{S} \vdash_\Delta r(p(\alpha)) \leq_{\pi r} r(q(\beta))$ via (*compat*). Hence

$$\bigcup\{\tau \mid \mathbf{S} \vdash_\Delta \tau \leq r(p(\alpha))\} \subseteq_{\pi r} \bigcup\{\tau \mid \mathbf{S} \vdash_\Delta \tau \leq r(q(\beta))\},$$

and therefore

$$\begin{array}{rcl} const(\rho'(r(p(\alpha)))) & = & \bigcup\{const(\rho(\tau)) \mid \mathbf{S} \vdash_\Delta \tau \leq r(p(\alpha))\} \\ & \subseteq_{\pi r} & \bigcup\{const(\rho(\tau)) \mid \mathbf{S} \vdash_\Delta \tau \leq r(q(\beta))\} \\ & = & const(\rho'(r(q(\beta)))). \end{array}$$

Hence

$$\rho'(p(\alpha)) \sqsubseteq \rho'(q(\beta)).$$

And hence $\rho' \models \mathbf{C}$, as required.

Thus $\rho' \models \mathbf{S}$. It remains to show that ρ and ρ' agree on E. Let $\alpha \in E$ and $r \in Path$. Then

$$\begin{array}{rcl} const(\rho'(r(\alpha))) & = & \bigcup\{const(\rho(\tau)) \mid \mathbf{S} \vdash_\Delta \tau \leq r(\alpha)\} \\ & & \text{by definition of } \rho' \\ & = & \bigcup\{const(\rho(\tau)) \mid \mathbf{S} \vdash_\Delta \tau \leq r(\alpha), SetVar(\tau) \subseteq E\} \\ & & \text{since } \rho(\beta) = \bot_s \text{ for } \beta \notin E \\ & & \text{and hence } \rho(\tau) = \bot_s \text{ for } SetVar(\tau) \not\subseteq E \\ & = & \bigcup\{const(\rho(\tau)) \mid \tau \leq r(\alpha) \in \Delta(\mathbf{S}) \mid_E\} \\ & = & const(\rho(r(\alpha))), \end{array}$$

since $[r(\alpha) \leq r(\alpha)] \in \Delta(\mathbf{S}) \mid_E$ by (*reflex*) and (*compat*), and for $[\tau \leq r(\alpha)] \in \Delta(\mathbf{S}) \mid_E$, $const(\rho(\tau)) \subseteq const(\rho(r(\alpha)))$. Thus ρ and ρ' agree on E, and the lemma holds. $\square$

LEMMA 5.3. (EQUIVALENCE OF PROOF SYSTEMS). *For a system of atomic constraints $\mathcal{S}$,*

$$\Delta(\mathcal{S}) = \Psi\Theta(\mathcal{S}).$$

PROOF. The proof of the inclusion $\Psi\Theta(\mathcal{S}) \subseteq \Delta(\mathcal{S})$ is straightforward.

We prove the converse inclusion $\Delta(\mathcal{S}) \subseteq \Psi\Theta(\mathcal{S})$ by induction on the numer of non $\Psi\Theta$-steps in the derivation of $\mathbf{C} \in \Delta(\mathcal{S})$. Again, for the base case, if $\mathbf{C} \in \Delta(\mathcal{S})$ because $\mathbf{C} \in \mathcal{S}$, then $\mathbf{C} \in \Psi\Theta(\mathcal{S})$. Otherwise we proceed by case analysis on the last rule used in the derivation of $\mathbf{C} \in \Delta(\mathcal{S})$.

(reflex), (compat). These rules are also in Ψ, and by induction, the antecedents are in $\Psi\Theta(\mathcal{S})$; hence $\mathbf{C} \in \Psi\Theta(\mathcal{S})$.

$(trans_\tau)$. The last step in the derivation must be

$$\frac{\tau_1 \le \tau \qquad \tau \le \tau_2}{\tau_1 \le \tau_2} \quad .(trans_\tau)$$

We proceed by case analysis on τ to show that $[\tau_1 \le \tau_2] \in \Psi\Theta(\mathcal{S})$.

(1) The case $\tau = c$ is impossible, since $[\tau_1 \le c]$ is not a compound constraint.

(2) If $\tau \in SetVar$, then $[\tau_1 \le \tau_2] \in \Psi\Theta(\mathcal{S})$ via $(trans_\alpha)$.

(3) Suppose $\tau = \mathtt{rng}(\tau')$. If $\tau' \in SetVar$ then $[\tau_1 \le \tau_2] \in \Psi\Theta(\mathcal{S})$ via (s_4).
Otherwise $\tau_1 \le \mathtt{rng}(\tau')$ and $\mathtt{rng}(\tau') \le \tau_2$ are not atomic constraints, and we proceed by considering the derivation of these constraints in $\Psi\Theta(\mathcal{S})$.
Suppose the last step in the derivation of $\mathcal{S} \vdash_{\Psi\Theta} \tau_1 \le \mathtt{rng}(\tau')$ is via $(trans_\alpha)$:

$$\frac{\tau_1 \le \gamma \qquad \gamma \le \mathtt{rng}(\tau')}{\tau_1 \le \mathtt{rng}(\tau')} \quad (trans_\alpha)$$

Then $\mathcal{S} \vdash_{\Psi\Theta} \gamma \le \mathtt{rng}(\tau')$, $\mathcal{S} \vdash_{\Delta} \mathtt{rng}(\tau') \le \tau_2$, and hence $\mathcal{S} \vdash_{\Delta} \gamma \le \tau_2$ via a shorter proof, so by induction $\mathcal{S} \vdash_{\Psi\Theta} \gamma \le \tau_2$, and hence $\mathcal{S} \vdash_{\Psi\Theta} \tau_1 \le \tau_2$ via $(trans_\alpha)$.
The case where last step in the derivation of $\mathcal{S} \vdash_{\Psi\Theta} \mathtt{rng}(\tau') \le \tau_2$ is via $(trans_\alpha)$ holds by similar reasoning.
Otherwise, the last step in the derivation of $\tau_1 \le \mathtt{rng}(\tau')$ is either via

$$\frac{\alpha_1 \le \mathtt{rng}(\beta_1) \qquad \beta_1 \le \tau'}{\alpha_1 \le \mathtt{rng}(\tau')} \quad (compose_1)$$

where $\tau_1 = \alpha_1$, or

$$\frac{\tau_1' \le \tau'}{\mathtt{rng}(\tau_1') \le \mathtt{rng}(\tau')} \quad (compat)$$

where $\tau_1 = \mathtt{rng}(\tau_1')$. Similarly, the last step in the derivation of $\mathtt{rng}(\tau') \le \tau_2$ is either via

$$\frac{\tau' \le \beta_2 \qquad \mathtt{rng}(\beta_2) \le \alpha_2}{\mathtt{rng}(\tau') \le \alpha_2} \quad (compose_3)$$

where $\tau_2 = \alpha_2$, or

$$\frac{\tau' \le \tau_2'}{\mathtt{rng}(\tau') \le \mathtt{rng}(\tau_2')} \quad (compat)$$

where $\tau_2 = \mathtt{rng}(\tau_2')$. We consider the four possible combinations for the derivations of $\tau_1 \leq \mathtt{rng}(\tau')$ and $\mathtt{rng}(\tau') \leq \tau_2$:

(a) Suppose $\tau_1 \leq \mathtt{rng}(\tau')$ is inferred via $(compose_1)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(compose_3)$. Then $\{\beta_1 \leq \tau', \tau' \leq \beta_2\} \subseteq \Delta(\mathcal{S})$, and therefore $[\beta_1 \leq \beta_2] \in \Delta(\mathcal{S})$ via $(trans_\tau)$. By induction, $[\beta_1 \leq \beta_2] \in \Psi\Theta(\mathcal{S})$, and the following derivation then shows that $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$.

$$\dfrac{\dfrac{\alpha_1 \leq \mathtt{rng}(\beta_1) \qquad \beta_1 \leq \beta_2}{\alpha_1 \leq \mathtt{rng}(\beta_2)}\,(s_2) \qquad \mathtt{rng}(\beta_2) \leq \alpha_2}{\alpha_1 \leq \alpha_2}\,(s_4)$$

(b) Suppose $\tau_1 \leq \mathtt{rng}(\tau')$ is inferred via $(compose_1)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(compat)$. Then $\{\beta_1 \leq \tau', \tau' \leq \tau_2'\} \subseteq \Delta(\mathcal{S})$, and therefore $[\beta_1 \leq \tau_2'] \in \Delta(\mathcal{S})$ via $(trans_\tau)$. By induction, $[\beta_1 \leq \tau_2'] \in \Psi\Theta(\mathcal{S})$, and the following derivation shows that $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$.

$$\dfrac{\alpha_1 \leq \mathtt{rng}(\beta_1) \qquad \beta_1 \leq \tau_2'}{\alpha_1 \leq \mathtt{rng}(\tau_2')}\,(compose_1)$$

(c) Suppose $\tau_1 \leq \mathtt{rng}(\tau')$ is inferred via $(compat)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(compose_3)$. This case holds by reasoning similar to the previous case.

(d) Suppose $\tau_1 \leq \mathtt{rng}(\tau')$ is inferred via $(compat)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(compat)$. Then $\{\tau_1' \leq \tau', \tau' \leq \tau_2'\} \subseteq \Delta(\mathcal{S})$, and therefore $[\tau_1' \leq \tau_2'] \in \Delta(\mathcal{S})$ via $(trans_\tau)$. By induction, $[\tau_1' \leq \tau_2'] \in \Psi\Theta(\mathcal{S})$, and therefore a $(compat)$-inference shows that $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$.

There are no other possibilities for the derivations of $\tau_1 \leq \mathtt{rng}(\tau')$ and $\mathtt{rng}(\tau') \leq \tau_2$.

(4) Suppose $\tau = \mathtt{dom}(\tau')$. This case holds by reasoning similar to the previous case where $\tau = \mathtt{rng}(\tau')$.

There are no other possibilities for τ.

There are no other possibilities for the derivation of $\mathbf{C} \in \Delta(\mathcal{S})$. Hence $\Delta(\mathcal{S}) \subseteq \Psi\Theta(\mathcal{S})$. □

Lemma 5.4. *For any system $\mathcal{S}$ of atomic constraints, $\Psi\Theta(\mathcal{S}) \mid_E \cong \Pi\Theta(\mathcal{S}) \mid_E$.*

Proof. Since the rule $(compat)$ does not create any Π or Θ opportunities, $\Psi\Theta(\mathcal{S}) = compat(\Pi\Theta(\mathcal{S}))$, and hence we just need to show that

$$compat(\Pi\Theta(\mathcal{S})) \mid_E \cong \Pi\Theta(\mathcal{S}) \mid_E .$$

Now

$$\begin{array}{rcl} compat(\Pi\Theta(\mathcal{S})) & \supseteq & \Pi\Theta(\mathcal{S}) \\ \therefore\ compat(\Pi\Theta(\mathcal{S})) \mid_E & \supseteq & \Pi\Theta(\mathcal{S}) \mid_E \\ \therefore\ compat(\Pi\Theta(\mathcal{S})) \mid_E & \models & \Pi\Theta(\mathcal{S}) \mid_E . \end{array}$$

To prove the converse, let $\rho \models \Pi\Theta(\mathcal{S}) \mid_E$. If $\rho \not\models compat(\Pi\Theta(\mathcal{S})) \mid_E$, then let $\mathbf{C}$ be the constraint in $compat(\Pi\Theta(\mathcal{S})) \mid_E$ with the smallest derivation such that $\rho \not\models \mathbf{C}$. Then the last step in the derivation of $\mathbf{C}$ must be via $(compat)$. Let $\mathbf{C}'$ be the antecedent of this rule in $compat(\Pi\Theta(\mathcal{S}))$. Then $SetVar(\mathbf{C}') = SetVar(\mathbf{C}) \subseteq E$, and hence $\mathbf{C}' \in compat(\Pi\Theta(\mathcal{S})) \mid_E$ with a smaller derivation. Therefore $\rho \models \mathbf{C}'$, and hence since (*compat*) is sound, $\rho \models \mathbf{C}$. Thus $\rho \models compat(\Pi\Theta(\mathcal{S})) \mid_E$, as required. □

B.3 Proofs for the Decidability of the Logic

LEMMA 6.1. (STAGING). *For any system of atomic constraints* $\mathcal{S}$,

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

PROOF. The equality $\Psi(\Theta(\mathcal{S})) = compat(\Pi(\Theta(\mathcal{S})))$ holds, since (*compat*) does not create any Π or Θ opportunities.

The inclusion $\Psi\Theta(\mathcal{S}) \supseteq \Psi(\Theta(\mathcal{S}))$ obviously holds. To prove the inclusion $\Psi\Theta(\mathcal{S}) \subseteq \Psi(\Theta(\mathcal{S}))$ holds, we suppose $\mathcal{S} \vdash_{\Psi\Theta} \mathbf{C}$, and prove $\mathbf{C} \in \Psi(\Theta(\mathcal{S}))$ by induction on the derivation $\mathcal{S} \vdash_{\Psi\Theta} \mathbf{C}$, and by case analysis on the last step in this derivation.

—Suppose $\mathcal{S} \vdash_{\Psi\Theta} \mathbf{C}$ via some rule in Ψ. By induction, the antecedents of this rule are in $\Psi(\Theta(\mathcal{S}))$, and hence $\mathbf{C} \in \Psi(\Theta(\mathcal{S}))$.

—Suppose $\mathcal{S} \vdash_{\Psi\Theta} \mathbf{C}$ via one of the rules (s_1), (s_2), or (s_3). These rules are subsumed by $(trans_\alpha)$, $(compose_1)$, and $(compose_4)$, and hence this case is subsumed by the previous case.

—Suppose $\mathcal{S} \vdash_{\Psi\Theta} \mathbf{C}$ via (s_4), based on the antecedents $\{\alpha \leq \mathtt{rng}(\beta),\ \mathtt{rng}(\beta) \leq \gamma\}$. By induction, these antecedents are in $\Psi(\Theta(\mathcal{S}))$. An examination of Ψ shows that Ψ can only infer $[\alpha \leq \mathtt{rng}(\beta)]$ if there exists α', β' such that $\Theta(\mathcal{S})$ contains the constraints

$$\alpha \leq^* \alpha' \qquad \alpha' \leq \mathtt{rng}(\beta') \qquad \beta' \leq^* \beta.$$

Similarly, Ψ can only infer $[\mathtt{rng}(\beta) \leq \gamma]$ if there exists β'', γ' such that $\Theta(\mathcal{S})$ contains the constraints

$$\beta \leq^* \beta'' \qquad \mathtt{rng}(\beta'') \leq \gamma' \qquad \gamma' \leq^* \gamma.$$

Hence

$\mathcal{S} \vdash_\Theta \alpha' \leq \mathtt{rng}(\beta'')$	via multiple applications of (s_2)
$\mathcal{S} \vdash_\Theta \alpha' \leq \gamma'$	via (s_4)
$\Theta(\mathcal{S}) \vdash_\Psi \alpha \leq \gamma$	via multiple applications of $(trans_\alpha)$.

—The case for (s_5) holds by reasoning similar to the previous case. □

LEMMA 6.5. *Let* $G = G_t(\mathcal{S}, E)$. *Then* $\Pi(\mathcal{S}) \mid_E = \mathcal{L}_G(R)$.

PROOF. We prove that $\Pi(\mathcal{S}) \mid_E \subseteq \mathcal{L}_G(R)$ by case analysis on $\mathbf{C} \in \Pi(\mathcal{S}) \mid_E$.

—Suppose $\mathbf{C} = [\alpha \leq \kappa]$. Then by Lemma 6.3, $\alpha_U \mapsto_G^* \kappa$. Since $SetVar(\mathbf{C}) \subseteq E$, $\alpha \in E$, and hence $\alpha_L \mapsto_G \alpha$. Thus $R \mapsto_G [\alpha_L \leq \alpha_U] \mapsto_G^* [\alpha \leq \kappa]$, and hence $[\alpha \leq \kappa] \in \mathcal{L}_G(R)$.

—The case where $\mathbf{C} = [\kappa \leq \alpha]$ follows by similar reasoning.

—Suppose $\mathbf{C} = [c \leq \kappa]$. If $\mathbf{C} \in \mathcal{S}$, then $\kappa = \alpha$, $\alpha \in E$, and $R \mapsto_G [c \leq \alpha_U] \mapsto_G [c \leq \alpha]$ as required.

If $\mathbf{C} \notin \mathcal{S}$, then an examination of the inference rules in Π shows that $\mathbf{C}$ can only be inferred via $(trans_\alpha)$, based on the antecedents $[c \leq \alpha]$ and $[\alpha \leq \kappa]$ in

$\Pi(\mathcal{S}) \mid_E$. By Lemma 6.3, $\alpha_L \mapsto^*_G c$ and $\alpha_U \mapsto^*_G \kappa$. Hence $R \mapsto [c \leq \kappa]$, and hence $[c \leq \kappa] \in \mathcal{L}_G(R)$, as required.

—Finally, consider $\mathbf{C} = [\kappa_1 \leq \kappa_2]$, where $\kappa_1, \kappa_2 \notin \mathit{SetVar}$. An examination of the inference rules in Π shows that $\mathbf{C}$ can only be inferred via (trans_α), based on the antecedents $[\kappa_1 \leq \alpha]$ and $[\alpha \leq \kappa_2]$. By Lemma 6.3, $\alpha_L \mapsto^*_G \kappa_1$ and $\alpha_U \mapsto^*_G \kappa_2$. Hence $R \mapsto [\kappa_1 \leq \kappa_2]$, and hence $[\kappa_1 \leq \kappa_2] \in \mathcal{L}_G(R)$, as required.

We prove that $\Pi(\mathcal{S}) \mid_E \supseteq \mathcal{L}_G(R)$ by case analysis on $\mathbf{C} \in \mathcal{L}_G(R)$.

—Suppose $\mathbf{C} = [\kappa_1 \leq \kappa_2]$. Then for some α, $\alpha_L \mapsto^*_G \kappa_1$ and $\alpha_U \mapsto^*_G \kappa_2$. By Lemma 6.3, $\{\kappa_1 \leq \alpha,\ \alpha \leq \kappa_2\} \subseteq \Pi(\mathcal{S})$ and $\mathit{SetVar}(\kappa_i) \subseteq E$. Hence $[\kappa_1 \leq \kappa_2] \in \Pi(\mathcal{S}) \mid_E$, as required.

—Otherwise $\mathbf{C} = [c \leq \kappa]$. Then for some α, $[c \leq \alpha] \in \mathcal{S}$ and $\alpha_U \mapsto^*_G \kappa$. By Lemma 6.3, $\{\alpha \leq \kappa\} \subseteq \Pi(\mathcal{S})$ and $\mathit{SetVar}(\kappa) \subseteq E$. Hence $[c \leq \kappa] \in \Pi(\mathcal{S}) \mid_E$, as required. □

C. THE ENTAILMENT ALGORITHM

The algorithm for deciding the restricted entailment of constraint systems is presented in Figure 15. Given two systems of atomic constraints $\mathcal{S}_1$ and $\mathcal{S}_2$ and a collection E of external set variables, the algorithm decides if $\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$. Based on the reasoning of Section 6.3, the relation $\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$ holds if and only if

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

where $G_i = G_t(\Theta(\mathcal{S}_i), E)$. To decide if $\mathcal{L}_{G_1}(R) \subseteq \mathit{compat}(\mathcal{L}_{G_2}(R))$, the algorithm first computes a 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 are nonterminals describing set expressions; 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 relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}$ is defined as the largest relation satisfying the conditions in Figure 15. It is computed by starting with the maximal relation (true at every point), and then iteratively setting entries to false, until the largest relation satisfying these conditions is reached. The first condition in the definition of $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}$ uses an RTG containment algorithm to detect if $\mathcal{L}([\alpha_L \leq \beta_U]) \subseteq \mathcal{L}(C) \cup \mathcal{L}(D)$. The following two conditions 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).

Based on the relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}$, the algorithm then defines a *computable entailment relation* $\mathcal{S}_2 \vdash^E_{\text{alg}} \mathcal{S}_1$ on constraint systems. This relation holds if and only if $\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$.

THEOREM C.1. (CORRECTNESS OF THE ENTAILMENT ALGORITHM). *$\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$ if and only if $\mathcal{S}_2 \vdash^E_{\text{alg}} \mathcal{S}_1$.*

PROOF. We prove this theorem based on the following invariant concerning the relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\cdot,\cdot,\cdot,\cdot]$:

$$\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D] \iff \mathcal{L}_{G_1}([\alpha_L \leq \beta_U]) \subseteq \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D) \qquad (1)$$

The Entailment Algorithm

For $i \in \{1, 2\}$ let:

$$\begin{aligned} G_i &= G_t(\Theta(\mathcal{S}_i), E) \\ L_i &= \{\alpha_L \mid \alpha \in SetVar(\mathcal{S}_i)\} \\ U_i &= \{\alpha_U \mid \alpha \in SetVar(\mathcal{S}_i)\} \end{aligned}$$

and let G_i be pre-processed to remove ϵ-transitions.
For $C \in \mathcal{P}_{\mathrm{fin}}(L_2 \times U_2)$, define

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

The relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\cdot,\cdot,\cdot,\cdot]$ is defined as the largest relation on

$$L_1 \times U_1 \times \mathcal{P}_{\mathrm{fin}}(L_2 \times U_2) \times \mathcal{P}_{\mathrm{fin}}(L_2 \times U_2)$$

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) $\mathcal{L}_{G_1}([X \leq Y]) \subseteq \mathcal{L}_{G_2}(C \cup D)$.

(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

$$D' = \{\langle \gamma'_L, \delta'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U)\}$$

(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

$$D' = \{\langle \delta'_L, \gamma'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{dom}(\gamma'_U), \delta_U \mapsto_{G_2} \mathtt{dom}(\delta'_L)\}$$

(4) In no other cases does $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ hold.

The *computable entailment relation* $\mathcal{S}_2 \vdash^E_{\mathrm{alg}} \mathcal{S}_1$ holds if and only if:

(1) $\forall R \mapsto_{G_1} [\alpha_L \leq \alpha_U].\ \mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \alpha_U, \{\langle \gamma_L, \gamma_U \rangle \mid \gamma \in SetVar(\mathcal{S}_2)\}, \emptyset]$, and

(2) $\forall R \mapsto_{G_1} [c \leq \alpha_U].\ \mathcal{L}_{G_1}(\alpha_U) \subseteq \mathcal{L}_{G_2}(\{\gamma_U \mid R \mapsto_{G_2} [c \leq \gamma_U]\})$.

Fig. 15. The computable entailment relation $\vdash^E_{\mathrm{alg}}$.

Now also assume that $\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$. Then $\Pi\Theta(\mathcal{S}_1) \mid_E \subseteq compat(\Pi\Theta(\mathcal{S}_2)) \mid_E$. By Lemma 6.5, $\Pi\Theta(\mathcal{S}_i) \mid_E = \mathcal{L}_{G_i}(R)$, and hence

$$\mathcal{L}_{G_1}(R) \subseteq compat(\mathcal{L}_{G_2}(R)).$$

Thus, for all $R \mapsto_{G_1} [\alpha_L \leq \alpha_U]$

$$\begin{aligned} \mathcal{L}_{G_1}([\alpha_L \leq \alpha_U]) &\subseteq compat(\mathcal{L}_{G_2}(R)) \\ \therefore\ \mathcal{L}_{G_1}([\alpha_L \leq \alpha_U]) &\subseteq compat(\mathcal{L}_{G_2}(\{\langle \gamma_L, \gamma_U \rangle \mid \gamma \in SetVar(\mathcal{S}_2)\})). \end{aligned}$$

Hence

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

Also, from $\mathcal{L}_{G_1}(R) \subseteq compat(\mathcal{L}_{G_2}(R))$, we have that for all $R \mapsto_{G_1} [c \leq \alpha_U]$

$$\begin{aligned} \mathcal{L}_{G_1}([c \leq \alpha_U]) &\subseteq compat(\mathcal{L}_{G_2}(R)) \\ \therefore\ \mathcal{L}_{G_1}([c \leq \alpha_U]) &\subseteq \mathcal{L}_{G_2}(R) \\ \therefore\ \mathcal{L}_{G_1}(\alpha_U) &\subseteq \mathcal{L}_{G_2}(\{\gamma_U \mid R \mapsto_{G\ 2} [c \leq \gamma_U]\}). \end{aligned}$$

Hence $\mathcal{S}_2 \vdash^E_{\text{alg}} \mathcal{S}_1$ holds. The proof of the converse implication that $\mathcal{S}_2 \vdash^E_{\text{alg}} \mathcal{S}_1$ implies $\mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1$ proceeds by a similar argument.

It remains to show that (1) holds. To prove the left-to-right direction, suppose that $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ holds and

$$\begin{array}{lll} \alpha_L & \mapsto_{G_1} X & \mapsto^*_{G_1} \tau_L \\ \beta_U & \mapsto_{G_1} Y & \mapsto^*_{G_1} \tau_U. \end{array}$$

We prove by induction on τ_L that

$$\mathcal{L}([\tau_L \leq \tau_U]) \subseteq \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D).$$

One of three cases in the definition of $\mathcal{R}$ must hold.

(1) $\mathcal{L}([\alpha_L \leq \beta_U]) \subseteq \mathcal{L}(C \cup D)$. This case is trivial.

(2) In this case

$$\begin{array}{lll} X = \texttt{rng}(\alpha'_L) & \alpha'_L \mapsto^*_{G_1} \tau'_L & \tau_L = \texttt{rng}(\tau'_L) \\ Y = \texttt{rng}(\beta'_U) & \beta'_U \mapsto^*_{G_1} \tau'_U & \tau_U = \texttt{rng}(\tau'_U) \end{array}$$

and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha'_L, \beta'_U, C, D']$, where

$$D' = \{\langle \gamma'_L, \delta'_U\rangle \mid \langle \gamma_L, \delta_U\rangle \in C \cup D, \gamma_L \mapsto_{G_2} \texttt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \texttt{rng}(\delta'_U)\}.$$

By induction, $[\tau'_L \leq \tau'_U] \in \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D')$.

—If $[\tau'_L \leq \tau'_U] \in \mathcal{L}(D')$ then there exists $\langle \gamma'_L, \delta_U\rangle \in D$ such that $\gamma'_L \mapsto^*_{G_2} \tau'_L$ and $\delta'_U \mapsto^*_{G_2} \tau'_U$. By the definition of D', there exists $\langle \gamma_L, \delta_U\rangle \in C \cup D$ such that $\gamma_L \mapsto^*_{G_2} \tau_L$ and $\delta_U \mapsto^*_{G_2} \tau_U$. Therefore $[\tau_L \leq \tau_U] \in \mathcal{L}(C \cup D)$, as required.

—If $[\tau'_L \leq \tau'_U] \in \mathit{compat}(\mathcal{L}(C))$ then $[\tau_L \leq \tau_U] \in \mathit{compat}(\mathcal{L}(C))$, as required.

(3) The proof for the third case of the definition of $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\cdot,\cdot,\cdot,\cdot]$ is similar to that for the second case.

To prove the right-to-left direction, suppose

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

and that the relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ does not hold. Hence there exists X, Y such that $\alpha_L \mapsto_{G_1} X$ and $\beta_U \mapsto_{G_1} Y$ and none of the three conditions in Figure 15 hold. Furthermore, since $\mathcal{R}$ is the largest relation satisfying the conditions in Figure 15, there exists a finite proof that none of the three conditions hold.

Of all possible such *counterexamples* $\langle \alpha_L, \beta_U, X, Y, C, D\rangle$, we pick the one with the smallest proof that the relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ does not hold, and proceed by case analysis on the last step in this proof.

—Suppose $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ does not hold because of condition (1). Then $\mathcal{L}([X \leq Y]) \not\subseteq \mathcal{L}(C \cup D)$, which contradicts the assumptions above.

—Suppose $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ does not hold because of condition (2). Then $X = \texttt{rng}(\alpha'_L)$ and $Y = \texttt{rng}(\beta'_U)$. Consider any pair of set expressions τ_L and τ_U such that $\alpha'_L \mapsto^*_{G_1} \tau_L$ and $\beta'_U \mapsto^*_{G_1} \tau_U$. We consider the two possibilities for $[\texttt{rng}(\tau_L) \leq \texttt{rng}(\tau_U)] \in \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D)$ separately.

—If $[\mathtt{rng}(\tau_L) \leq \mathtt{rng}(\tau_U)] \in \mathcal{L}(C) \cup \mathcal{L}(D)$, then there exists $\langle \gamma_L, \delta_U \rangle \in C \cup D$ such that

$$\gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma_L') \mapsto^*_{G_2} \mathtt{rng}(\tau_L)$$
$$\delta_U \mapsto_{G_2} \mathtt{rng}(\delta_U') \mapsto^*_{G_2} \mathtt{rng}(\tau_U).$$

Hence $[\tau_L \leq \tau_U] \in \mathcal{L}(D')$, where

$$D' = \{\langle \gamma_L', \delta_U' \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma_L'), \delta_U \mapsto_{G_2} \mathtt{rng}(\delta_U')\}.$$

—Otherwise $[\mathtt{rng}(\tau_L) \leq \mathtt{rng}(\tau_U)] \in compat(\mathcal{L}(C)) \setminus \mathcal{L}(C)$, and hence $[\tau_L \leq \tau_U] \in compat(\mathcal{L}(C))$.

Hence

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

The proof that $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ does not hold cannot rely on a smaller proof that $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L', \beta_U', C, D']$ does not hold, since that would yield a counterexample with a smaller proof.

—The case where $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ does not hold because of condition (3) is also impossible via reasoning similar to the previous case.

Thus the invariant (1) is true, and thus the theorem holds. $\square$

D. NOTATIONS

Symbol	**Meaning**
$\rightharpoonup$	Partial map constructor
$\mathcal{P}$	Power set constructor
$\mathcal{P}_{\text{fin}}$	Finite power set constructor
$M \in \Lambda$	Terms
$V \in Value$	Values
$x \in Var$	Variables
$b \in BasicConst$	Basic constants
$t \in Tag$	Function tags
$l \in Label$	Labels
$\beta_v, \beta_{let}, unlabel$	Reduction rules
$\longrightarrow$	Reduction relation
$\mathcal{E}$	Evaluation contexts
$\longmapsto, \longmapsto^*$	Standard reduction relation
$eval$	Evaluator
$\tau \in SetExp$	Set expressions
$\alpha, \beta, \gamma \in SetVar$	Set variables
$c \in Const$	Constants
$\mathtt{dom}, \mathtt{rng}$	Selectors
$\mathcal{C} \in AtomicCon$	Atomic constraints
$\mathsf{S} \in AtomicConSystem$	System of atomic constraints
$SetVar$	Set variables in a constraint system

$\Gamma \in \mathit{DerivCtxt}$	Set variable context
$\Gamma \vdash M : \alpha, \mathcal{S}$	Constraint derivation rules
$\sigma \in \mathit{ConSchema}$	Constraint schema
$FV[\mathit{range}(\Gamma)]$	Free variables in the range of Γ
$\vdash_\Theta$	Deduction via Θ
$\Theta = \{s_1, \ldots, s_n\}$	Inference rules
sba	Analysis function
in	Values described by constants
$\mathcal{D}$	Domain for constraints
$\mathit{const}, \mathit{dom}, \mathit{rng}$	Extract components of element of $\mathcal{D}$
$\sqsubseteq, \top, \bot, \sqcup, \sqcap$	Ordering, elements and operations on $\mathcal{D}$
$\rho \in \mathit{SetEnv}$	Set environment
$\mathsf{C} \in \mathit{Constraint}$	Constraints
$\mathsf{S} \in \mathit{ConstraintSystem}$	Constraint systems
$\models$	Satisfies, or entails
$\mathit{Soln}(\mathsf{S})$	Solution space
$\cong$	Observable equivalence
$\mathit{Soln}(\mathsf{S}) \mid_E$	Restricted solution space
$\models_E$	Restricted entailment
$\cong_E$	Restricted observable equivalence
$\mathsf{S} \mid_E$	Restriction of a constraint system
$\sqsubseteq_s, \top_s, \bot_s, \sqcup_s, \sqcap_s$	Alternative ordering on domain
$\mathit{LeastSoln}$	Least Solution
E	External variables
Δ	Inference rules on constraint systems
κ	Nonconstant set expression
$\mathbf{C} \in \mathit{CmpdConstraint}$	Compound constraint
$\mathbf{S} \in \mathit{CmpdConSystem}$	System of compound constraints
Ψ	Inference rules on constraint systems
Π	Inference rules on constraint systems
$\vdash^E_{\Psi\Theta}, =^E_{\Psi\Theta}$	Relations on constraint systems
G	Grammar
$G_r(\mathcal{S}, E)$	Function producing regular grammar
α_L, α_U	Grammar nonterminals
$\mathcal{L}_G(X)$	Language for nonterminal X in G
$G_t(\mathcal{S}, E)$	Function producing RTG
R	Root nonterminal

Sel^+, Sel^-	Sets of selectors
$\texttt{sel}^+, \texttt{sel}^-, \texttt{sel}$	Selectors
Λ^p	Language Λ plus pairs
$\texttt{car}, \texttt{cdr}$	Selectors for pairs
Λ^{cc}	Language Λ^p plus continuations
$\Lambda^!$	Language Λ^p plus assignments
$D \in \mathit{Defines}$	Definitions
$z \in \mathit{AssignVar}$	Assignable variables
$H \in \mathit{Heap}$	Heap of definitions
$\leq^*$	Transitive closure of $\leq$
$p, q, r \in \mathit{Path}$	Paths
arity, π	Arity function
$\leq_i$	Either $\leq$ or $\geq$
$\subseteq_i$	Either $\subseteq$ or $\supseteq$
$\mathcal{R}$	Relation for computing entailment

ACKNOWLEDGMENT

We thank the anonymous referees who made several useful suggestions.

REFERENCES

AHO, A., J. HOPCROFT AND J. ULLMAN. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

AIKEN, A., WIMMERS, E. L., AND LAKSHMAN, T. K. 1994. Soft typing with conditional types. In *Proceedings of the Symposium on the Principles of Programming Languages*. 163–173.

BARENDREGT, H.P. 1984. *The Lambda Calculus: Its Syntax and Semantics*, Revised ed. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland, Amsterdam.

COUSOT, P. AND COUSOT, R. 1995. Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. In *Proceedings on Functional Programming and Computer Architecture*. 170–181.

DEUTSCH, A. AND HEINTZE, N. 1996. Partial solving of set constraints. Unpublished manuscript.

DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. 1994. Reducing the cost of data flow analysis by congruence partitioning. In *International Conference on Compiler Construction*.

EIFRIG, J., SMITH, S., AND TRIFONOV, V. 1995. Sound polymorphic type inference for objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*.

FÄHNDRICH, M. AND AIKEN, A. 1996. Making set-constraint based program analyses scale. Technical Report UCB/CSD-96-917, University of California at Berkeley.

FLANAGAN, C. 1997. Effective static debugging via componential set-based analysis. Ph.D. thesis, Rice University, Houston, Texas.

FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., WEIRICH, S., AND FELLEISEN, M. 1996. Finding bugs in the web of program invariants. In *Programming Language Design and Implementation*. 23–32.

Flatt, M. 1997. MzScheme reference manual. Technical Report TR97-280, Rice University.

Flatt, M. and Felleisen, M. 1998. Units: Cool modules for HOT languages. In *Proceedings of the Symposium on the Programming Language Design and Implementation.*

Gécseg, F. and Steinby, M. 1984. *Tree Automata.* Akadémiai Kiadó, Budapest.

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

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

Jagannathan, S. and Wright, A. K. 1995. Effective flow analysis for avoiding run-time checks. In **2nd** *International Static Analysis Symposium, Lecture Notes in Computer Science, vol. 983.* Springer-Verlag, 207–224.

Jones, N. and Muchnick, S. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Principles of Programming Languages.* 66–74.

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

Palsberg, J. and O'Keefe, P. 1995. A type system equivalent to flow analysis. In *Proceedings of the Symposium on the Principles of Programming Languages.* 367–378.

Plotkin, G. D. 1975. Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci. 1,* 125–159.

Pottier, F. 1996. Simplifying subtyping constraints. In *Proceedings of the International Conference on Functional Programming.* 122–133.

Reynolds, J. 1969. Automatic computation of data set defintions. *Information Processing'68*, 456–461.

Tofte, M. 1990. Type inference for polymorphic references. *Info. Comput. 89,* 1 (November), 1–34.

Trifonov, V. and Smith, S. 1996. Subtyping constrained types. In **3rd** *International Static Analysis Symposium, Lecture Notes in Computer Science, vol. 1145.* 349–365.

Wright, A. and Felleisen, M. 1994. A syntactic approach to type soundness. *Info. Comput. 115,* 1, 38–94.

Wright, A. K. 1995. Simple imperative polymorphism. *Lisp Symbol. Comput. 8,* 4 (Dec.), 343–356.

Received November 1997; accepted April 1998