Modular and Polymorphic Set-Based Analysis: Theory and Practice

Cormac Flanagan
Matthias Felleisen

Rice COMP TR96-266

November 1996

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Copyright ©1996 by

Cormac Flanagan and Matthias Felleisen

Modular and Polymorphic Set-Based Analysis: Theory and Practice

(revision 1.3)

Cormac Flanagan Matthias Felleisen

Department of Computer Science,
Rice University,
Houston, TX 77251-1892

Contact email: cormac@cs.rice.edu

Abstract

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 Rice's program development environment, which includes a static debugger based on SBA, indicate that SBA can deal with programs of up to a couple of thousand lines of code. However, SBA does not cope with larger programs because it generates large systems of constraints for these programs. These constraint systems are at least linear, and possibly quadratic, in the size of the analyzed program.

This paper presents theoretical and practical results concerning methods for reducing the size of constraint systems. The theoretical results include a complete proof-theoretic characterization of the observable behavior of a constraint system, which we use to 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 practical problem of simplifying constraint systems. Based on the resulting algorithms, we develop a *componential set-based analysis*, a modular and polymorphic variant of SBA. Experimental results validate the practicality of the simplification algorithms and the analysis. The simplified constraint systems are typically at least an order of magnitude smaller than the original systems, which significantly reduces both the time and space required by the analysis.

Contents

1 The Effectiveness of Set-Based Analysis **1**

2 The Source Language **2**

3 Set-Based Analysis **3**
- 3.1 The Constraint Language . . . 4
- 3.2 The Lattice $\mathcal{D}$ . . . 4
- 3.3 Semantics of Constraints . . . 5
- 3.4 Deriving Constraints . . . 6
- 3.5 Soundness of the Derived Constraints . . . 7
- 3.6 Set Based Analysis . . . 7
- 3.7 Computing the Least Solution . . . 8

4 Observable Equivalence of Constraint Systems **9**
- 4.1 Conditions for Constraint Simplification . . . 9
- 4.2 Proof-Theoretic Characterization of Observable Equivalence . . . 10
- 4.3 Deciding Observable Equivalence . . . 13

5 Practical Constraint System Simplification **16**
- 5.1 Empty Constraint Simplification . . . 17
- 5.2 Unreachable Constraint Simplification . . . 18
- 5.3 Removing ϵ-Constraints . . . 18
- 5.4 Hopcroft's Algorithm . . . 18
- 5.5 Simplification Benchmarks . . . 19

6 Componential Set-Based Analysis **20**
- 6.1 Benchmarks . . . 21

7 Efficient Polymorphic Analysis **22**

8 Competitive Work **23**

9 Future Work **24**

A Proofs **25**
- A.1 Subject Reduction Proof . . . 25
- A.2 Proofs for Computing Set-Based Analysis . . . 29
- A.3 Proofs for Observable Equivalence . . . 33
- A.4 Correctness of the Entailment Algorithm . . . 43
- A.5 Correctness of the Hopcroft Algorithm . . . 45

B Notations **47**

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 [8]. If an operation may fault due to a violation of its invariant, MrSpidey highlights the program operation so that the programmer can investigate the potential fault site before running the program. Using the graphical explanation facilities of MrSpidey, the programmer can determine whether this fault will really happen or whether the corresponding correctness proof is beyond the analysis's capabilities.

MrSpidey's program analysis is a constraint-based system similar to Heintze's set-based analysis of ML programs [10]. The analysis consists of two co-mingled phases: a *specification* phase, during which MrSpidey derives constraints describing the data flow relationships, and a *solution* phase, during which MrSpidey solves the constraints. The solution conservatively approximates the set of values that may result from 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. Although the constant on the cubic element in the polynomial is small, it is noticeable for programs of several thousand lines.

- Large programming projects tend to re-use functions in a polymorphic fashion. To avoid merging information between unrelated calls to such functions, the analysis must duplicate the constraints for each corresponding call site. This duplication is expensive because the size of the constraint set is at best linear, and possibly quadratic, in the size of the function.

A closer look at these two obstacles quickly reveals that the major limitation of set-based analysis is the size of the constraint system describing the data flow relationships of a program. If we could reduce the size of constraint systems without affecting the solution space that they denote, we could simplify constraint sets at intermediate stages during the specification phase and thus reduce the analysis time. In particular, after simplifying the constraint set of a polymorphic function, MrSpidey would duplicate a much smaller constraint set at each polymorphic reference than the plain analysis; similarly, producing a simplified constraint set for a module would substantially reduce the cost of solving the combined set of constraints for a modularized program.

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

[1] A number of researchers, including Reynolds [18], Jones and Muchnick [14], Heintze [10], Aiken [2], and Cousot and Cousot [3] previously exploited the relationship between RTGs and the *least solution* of a

gorithm for deciding the equivalence of constraint systems. Unfortunately, the problem is PSPACE-hard.

Fortunately, a minimized constraint set is only optimal but not necessary for practical purposes. The practical question concerns finding approximate algorithms for simplifying constraint sets that would make MrSpidey more useful. To answer this question, we exploit the correspondence between the minimization problem for RTGs and constraint sets 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 component-wise, variant of set-based analysis. Experimental results verify the effectiveness of these algorithms and the corresponding flavors of the analysis. The simplified constraint systems are typically at least an order of magnitude smaller than the original constraint set. Moreover, these reductions in size result in significant gains in the speed of the analysis.

We expect that some of our theoretical and practical results as well as the techniques will carry over to other constraint-based systems, such as the conditional type system of Aiken *et al.* [2], Eifrig *et al.*'s object-oriented type system [5], or Pottier's or Smith *et al.*'s subtyping simplification algorithms [17, 20].

The presentation of our results proceeds as follows. Section 2 describes an idealized source language. Section 3 describes our set-based analysis. Section 4 characterizes the relationship between constraint systems and RTGs, and section 5 exploits this connection to derive a number of practical constraint simplification algorithms. Sections 6 and 7 discuss how these algorithms perform in a realistic program analysis system. Section 8 discusses related work, and section 9 describes directions for future research. Appendix A contains proof of various theorems and lemmas. Appendix B contains an index of notations used in the paper.

2 The Source Language

Syntax:

M	$\in$	Λ	$=$	$x \mid V \mid (M\ M) \mid (\mathbf{let}\ (x\ M)\ M) \mid M^l$	(Expressions)
V	$\in$	*Value*	$=$	$b \mid (\lambda^t x.M)$	(Values)
x	$\in$	*Vars*	$=$	$\{x, y, z, \ldots\}$	(Variables)
b	$\in$	*BConst*			(Basic Constants)
t	$\in$	*Tag*			(Function Tags)
l	$\in$	*Label*			(Expression Labels)

Figure 1: The source language Λ

constraint system. We present an additional result, namely a connection between RTGs and the observable behavior (*i.e.*, the *entire solution space*) of constraint systems.

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

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 including assignments, recursive data structures, objects and modules along the lines described in an earlier report [7].

Expressions in the language are either variables, values, function applications, **let**-expressions, or labeled expressions. We use labels to identify those program expressions whose values we wish to predict. Values include basic constants and functions. Functions have identifying tags so that MrSpidey can reconstruct a call-graph from the results of the analysis. We use **let**-expressions to introduce polymorphic bindings, and hence restrict these bindings to syntactic values [22]. We work with the usual conventions and terminology of the λ_v-calculus when discussing syntactic issues. In particular, the substitution operation $M[x \leftarrow V]$ replaces all free occurrences of x within M by V, and Λ^0 denotes the set of closed terms, also called *programs*.

We specify the meaning of programs based upon the *notions of reduction* β_v, β_{let} and *unlabel*:

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

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:

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

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 $(\mathcal{E}\ M)$.

The reduction relation $\longmapsto$ is the compatible closure 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 partial function *eval* on programs is defined as:

$$\begin{array}{lcll} \mathit{eval} : \Lambda^0 & \longrightarrow_p & \mathit{Value} & \\ \mathit{eval}(M) & = & V & \text{if } M \longmapsto^* V \end{array}$$

3 Set-Based Analysis

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

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

3.1 The Constraint Language

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

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

A *set expression* τ is either a set variable; a constant; or one of the "selector" expressions $\mathtt{dom}(\tau)$ or $\mathtt{rng}(\tau)$. By using selector expressions, we can specify each "quantum" of the program's data flow behavior independently; using constructors would combine several of these quanta into one constraint.[4] The meta-variables α, β, γ range over set variables, and we include program labels in the collection of set variables. Constants include both basic constants and function tags. A *constraint* $\mathcal{C}$ is an inequality $\tau_1 \leq \tau_2$ relating two set expressions.

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

$$c \leq \beta \;\mid\; \alpha \leq \beta \;\mid\; \alpha \leq \mathtt{dom}(\beta) \;\mid\; \mathtt{rng}(\alpha) \leq \beta \;\mid\; \mathtt{dom}(\alpha) \leq \beta \;\mid\; \alpha \leq \mathtt{rng}(\beta)$$

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

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

3.2 The Lattice $\mathcal{D}$

A set expression denotes a collection of values. For our sample language, such a collection consists of basic constants and functions, and is represented as a triple $X = \langle C, D, R \rangle$. The first component $C \in \mathcal{P}(\mathit{Const})$[5] is a set of basic constants and function tags, and represents a set of run-time values (relative to a given program) according to the relation $V \ \mathit{in}\ C$:

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

The second and third components of X denote the possible argument values (dom) and result values (rng) of functions in X, respectively. Since these two components also denote value sets, the appropriate model for set expressions is the solution of the equation:[6]

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

[4] For example, we specify a function's behavior via the two constraints $\{\mathtt{dom}(\alpha) \leq \alpha_1,\ \alpha_2 \leq \mathtt{rng}(\alpha)\}$ instead of the combined constraint $(\alpha_1 \to \alpha_2) \leq \alpha$.

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

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

We use the functions $const : \mathcal{D} \longrightarrow \mathcal{P}(Const)$ 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 that is contravariant in the second position, since the information about argument values at an application needs to flow *backward* along data-flow paths to the formal parameter of the corresponding function definitions:

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

The set $\mathcal{D}$ forms a complete lattice under this ordering; the top and bottom elements are the solutions to the equations $\top = \langle Const, \bot, \top\rangle$ and $\bot = \langle \emptyset, \top, \bot\rangle$, respectively. The least upper bound and greatest lower bound operations are recursively defined as:

$$\begin{array}{rcl} \langle C_1, D_1, R_1\rangle \sqcup \langle C_2, D_2, R_2\rangle & = & \langle C_1 \cup C_2, D_1 \sqcap D_2, R_1 \sqcup R_2\rangle \\ \langle C_1, D_1, R_1\rangle \sqcap \langle C_2, D_2, R_2\rangle & = & \langle C_1 \cap C_2, D_1 \sqcup D_2, R_1 \sqcap R_2\rangle \end{array}$$

3.3 Semantics of Constraints

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

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

The collection of set environments forms a complete lattice $SetEnv$ under the pointwise extension of the ordering relation $\sqsubseteq$ on $\mathcal{D}$. An environment ρ *satisfies* a constraint $\mathcal{C} = [\tau_1 \leq \tau_2]$ (written $\rho \models \mathcal{C}$) if $\rho(\tau_1) \sqsubseteq \rho(\tau_2)$. Similarly, ρ satisfies $\mathcal{S}$, or ρ is a solution of $\mathcal{S}$ (written $\rho \models \mathcal{S}$) if $\rho \models \mathcal{C}$ for each $\mathcal{C} \in \mathcal{S}$. The *solution space* of a constraint system $\mathcal{S}$ is:

$$Soln(\mathcal{S}) = \{\rho \mid \rho \models \mathcal{S}\}$$

A constraints set $\mathcal{S}_1$ *entails* $\mathcal{S}_2$ (written $\mathcal{S}_1 \models \mathcal{S}_2$) iff $Soln(\mathcal{S}_1) \subseteq Soln(\mathcal{S}_2)$, and $\mathcal{S}_1$ is *observably equivalent to* $\mathcal{S}_2$ (written $\mathcal{S}_1 \cong \mathcal{S}_2$) iff $\mathcal{S}_1 \models \mathcal{S}_2$ and $\mathcal{S}_2 \models \mathcal{S}_1$.

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

$$Soln(\mathcal{S})\,|_E \quad = \quad \{\rho \mid \exists \rho' \in Soln(\mathcal{S}) \text{ such that } \rho(\alpha) = \rho'(\alpha)\ \forall \alpha \in E\}$$

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

Definition 3.1. (*Restricted Entailment, Restricted Observable Equivalence*)

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

∎

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

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

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

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

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

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

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

Figure 2: Constraint derivation rules.

3.4 Deriving Constraints

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

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

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

The rule (*let*) produces a *constraint schema* $\sigma = \forall A.(\alpha, \mathcal{S})$ for polymorphic, **let**-bound expressions [19, 2]. The set variable α names the result of the expression, the constraint system $\mathcal{S}$ describes the data flow relationships of the expression, using α, and the set A contains those internal set variables of the constraint system that must be duplicated at each reference to the **let**-bound variable via the rule (*inst*). We use $FV[\mathit{rng}(\Gamma)]$ to denote the free set variables in the range of Γ. The free set variables of a schema $\sigma = \forall A.(\alpha, \mathcal{S})$ are

those in $\mathcal{S}$ but not in A, and the free variables of a set variable is simply the set variable itself.

3.5 Soundness of the Derived Constraints

Let P be a program such that $\emptyset \vdash P : \alpha, \mathcal{S}$. Typically, $\mathcal{S}$ has many solutions. Each solution ρ correctly approximates the value sets of labeled expressions in P. That is, if ρ is a solution of $\mathcal{S}$, $P \longmapsto^* P'$, and P' contains an expression V^l, then V is a possible value of some expression M^l in the source program P and V *in* $const(\rho(l))$. We prove this property using a subject reduction proof, following Wright and Felleisen [21][7] and Palsberg [15].

Main Lemma 3.2 (Soundness) *If* $\emptyset \vdash P : \alpha, \mathcal{S}$ *and* $\rho \models \mathcal{S}$ *and* $P \longmapsto^* \mathcal{E}[\ V^l\ ]$ *then* V *in* $const(\rho(l))$.

Proof: See Appendix A.1.

3.6 Set Based Analysis

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

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

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

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

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

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

Therefore we introduce a second ordering $\sqsubseteq_s$ on $\mathcal{D}$ that properly ranks environments according to their accuracy. This ordering is covariant in the domain position, *i.e.*:

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

The maximal and minimal elements of $\mathcal{D}$ under $\sqsubseteq_s$ are the solutions to the equations $\top_s = \langle Const, \top_s, \top_s \rangle$ and $\bot_s = \langle \emptyset, \bot_s, \bot_s \rangle$, respectively. The least upper bound and greatest lower bound operations are recursively defined as:

$$\begin{array}{rcl} \langle C_1, D_1, R_1 \rangle \sqcup_s \langle C_2, D_2, R_2 \rangle & = & \langle C_1 \cup C_2, D_1 \sqcup_s D_2, R_1 \sqcup_s R_2 \rangle \\ \langle C_1, D_1, R_1 \rangle \sqcap_s \langle C_2, D_2, R_2 \rangle & = & \langle C_1 \cap C_2, D_1 \sqcap_s D_2, R_1 \sqcap_s R_2 \rangle \end{array}$$

[7] The proof is extensible to the full imperative language.

Under the ordering $\sqsubseteq_s$, a constraint system has both a maximal solution ($\rho^\top$ above) and a minimal solution. The minimal solution exists because the greatest lower bound $\sqcap_s$ with respect to $\sqsubseteq_s$ of two solutions is also a solution [10].

Lemma 3.3 (Least Solution of Constraint Systems) *Every constraint system has a solution that is least with respect to* $\sqsubseteq_s$.

Proof: See Appendix A.2.

Using Lemma 3.3, it makes sense to define $\mathit{LeastSoln}(\mathcal{S})$ as the least solution of the constraint system $\mathcal{S}$ under the ordering $\sqsubseteq_s$, and set-based analysis as the function that extracts the basic constants and function tags for each labeled expression from this least solution.

Definition 3.4. (*sba*) If $\emptyset \vdash P : \alpha, \mathcal{S}$, then $\mathit{sba}(P)(l) = \mathit{const}(\mathit{LeastSoln}(\mathcal{S})(l))$. ∎

By Lemma 3.2, $\mathit{sba}(P)$ correctly characterizes the possible value sets for each labeled expression.

Theorem 3.5 *If* $P \longmapsto^* \mathcal{E}[\ V^l\ ]$ *then* V *in* $\mathit{sba}(P)(l)$.

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

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

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

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

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

Figure 3: The Rules $\Theta = \{s_1, \ldots, s_5\}$

3.7 Computing the Least Solution

To compute $\mathit{sba}(P)$, we close the constraint system for P under the rules Θ described in figure 3. Intuitively, these rules infer all the data flow paths in the program and propagate values along those paths. Specifically, the rules (s_1), (s_2), and (s_3) propagate information about constants, function domains and function ranges forward along the data flow paths of the program. These data flow paths are described by constraints of the form $\beta \leq \gamma$.

The rule (s_4) constructs the data flow paths from actual to formal parameters for each function call, and the rule (s_5) similarly constructs data flow paths from function bodies to corresponding call sites. We write $\mathcal{S} \vdash_\Theta \mathcal{C}$ if $\mathcal{S}$ proves $\mathcal{C}$ via the rules Θ, and use $\Theta(\mathcal{S})$ to denote the closure of $\mathcal{S}$ under Θ, *i.e.*, the set $\{\mathcal{C} \mid \mathcal{S} \vdash_\Theta \mathcal{C}\}$.

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

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

Theorem 3.6 *If* $P \in \Lambda^0$ *and* $\emptyset \vdash P : \alpha, \mathcal{S}$ *then* $sba(P)(l) = \{c \mid \mathcal{S} \vdash_\Theta c \leq l\}$.

Proof: See Appendix A.2.

4 Observable Equivalence of Constraint Systems

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

To avoid this problem, we develop algorithms for simplifying constraints systems. Applying these simplification algorithms at strategic points during the constraint derivation, *e.g.*, after module definitions or polymorphic function definitions, significantly reduces both the time and space required by the overall analysis.

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

4.1 Conditions for Constraint Simplification

Let us consider a program P containing a program component M, where M may be a module definition or a polymorphic function definition. Suppose the constraint derivations for M concludes $\Gamma \vdash M : \alpha, \mathcal{S}_1$, where $\mathcal{S}_1$ is the constraint system for M. Our goal is to replace $\mathcal{S}_1$ by a simpler constraint system, say S_2, without changing $sba(P)$.

Since the constraint derivation process is compositional, the constraint derivation for the entire program will conclude $\emptyset \vdash P : \beta, \mathcal{S}_C \cup \mathcal{S}_1$, where $\mathcal{S}_C$ is the constraint system

for the context surrounding M. The union of the sets $\mathcal{S}_C$ and $\mathcal{S}_1$ describes the space of solutions for the entire program, which means that the space of solutions is the intersection of the two respective solution spaces:

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

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

- $sba(P)$ 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 $\{\alpha\} \cup FV[rng(\Gamma)]$.

Thus the only properties of $\mathcal{S}_1$ relevant to the analysis is the solution space for its *external set variables* $E = Label \cup \{\alpha\} \cup FV[rng(\Gamma)]$.

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

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

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

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

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

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

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

Lemma 4.1 *If* $\emptyset \vdash_{\cong} P : \alpha, \mathcal{S}$, *then* $sba(P)(l) = const(LeastSoln(\mathcal{S})(l))$.

Proof: See Appendix A.3.

4.2 Proof-Theoretic Characterization of Observable Equivalence

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

The key properties of the observational equivalence relation are reflections of the reflexivity and transitivity of the ordering relation ($\sqsubseteq$) and the monotonicity and anti-monotonicity

$$\frac{\alpha \leq \mathtt{rng}(\beta) \qquad \beta \leq \kappa}{\alpha \leq \mathtt{rng}(\kappa)} \quad (compose_1) \qquad \frac{\alpha \leq \mathtt{dom}(\beta) \qquad \beta \geq \kappa}{\alpha \leq \mathtt{dom}(\kappa)} \quad (compose_2)$$

$$\frac{\alpha \geq \mathtt{rng}(\beta) \qquad \beta \geq \kappa}{\alpha \geq \mathtt{rng}(\kappa)} \quad (compose_3) \qquad \frac{\alpha \geq \mathtt{dom}(\beta) \qquad \beta \leq \kappa}{\alpha \geq \mathtt{dom}(\kappa)} \quad (compose_4)$$

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

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

Figure 4: The Inference Rule System Ψ

of the functions *rng* and *dom*, respectively. We can reify these properties into a syntactic proof system via the following inference rules:

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

where we restrict meta-variables $\kappa, \kappa_1, \kappa_2$ to non-constant set expressions to avoid inferring useless tautologies:

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

The rules (*reflex*) and (*trans'*) capture the reflexivity and transitivity of the ordering relation $\sqsubseteq$; (*compat*) reflects the monotonicity and anti-monotonicity of the functions *rng* and *dom*, respectively. Many of the inferred constraints lie outside of the original language of simple constraints. The extended language of *compound constraints* is:

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

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

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

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

Proof: See Appendix A.3.

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

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

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

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

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

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

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

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

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

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

Proof: See Appendix A.3.

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

- $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ if and only if $\Psi\Theta(\mathcal{S}_1) \mid_E \supseteq \Pi\Theta(\mathcal{S}_2) \mid_E$,
- $\mathcal{S}_1 =^E_{\Psi\Theta} \mathcal{S}_2$ 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 completely characterize restricted entailment and observable equivalence of constraint systems.

Theorem 4.5 (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: See Appendix A.3.

4.3 Deciding Observable Equivalence

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

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

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

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

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

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

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

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

Definition 4.6. (*Regular Grammar* $G_r(\mathcal{S}, E)$) Let $\mathcal{S}$ be a simple constraint system and E a collection of set variables. The regular grammar $G_r(\mathcal{S}, E)$ consists of the non-terminals

$\{\alpha_L, \alpha_U \mid \alpha \in \mathit{Vars}(\mathcal{S})\}$ and the following productions:

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

∎

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

Lemma 4.7 *Let $G = G_r(\mathcal{S}, E)$. Then:*

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

Proof: See Appendix A.3.

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

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

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

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

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

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

Proof: See Appendix A.3.

Before we can exploit the grammar representation of $\Pi(\mathcal{S})\mid_E$, we must prove that the closure under $\Theta \cup \Pi \cup \{\mathit{compat}\}$ can be performed in a staged manner. The following lemma justifies this staging of the closure algorithm. In particular, it states that Π does not create any additional opportunities for rules in Θ, and (*compat*) does not create any additional opportunities for Π or Θ.

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

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

Proof: See Appendix A.3.

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

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

The containment question:

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

can be decided via an RTG containment algorithm. To decide the more difficult question:

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

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

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

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

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

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

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

Proof: See Appendix A.4.

The entailment algorithm takes exponential time, since the size of $\mathcal{R}$ is exponential in the number of set variables in $\mathcal{S}_2$. Although faster algorithms for the entailment may

The Entailment Algorithm

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

$$\begin{array}{lcllcl} G_1 & = & G_r(\mathcal{S}_1, E) & L_i & = & \{\alpha_L \mid \alpha \in \mathit{Vars}(\mathcal{S}_i)\} \\ G_2 & = & G_l(\mathcal{S}_2, E) & U_i & = & \{\alpha_U \mid \alpha \in \mathit{Vars}(\mathcal{S}_i)\} \end{array}$$

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

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

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

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

then one of the following cases hold:

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

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

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

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

exist, these algorithms must all be in PSPACE, because the containment problem on NFA's, which is PSPACE-complete [1], can be polynomially reduced to the entailment problem on constraint systems.

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

5 Practical Constraint System Simplification

Fortunately, 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 produces corresponding reductions in the time and space required for the overall analysis.

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

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

also preserves the observable behavior of $\mathcal{S}$ with respect to E. Based on this observation, we have transformed 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 non-terminals and p over *paths*, which are sequences of the constructors `dom` and `rng`. Each algorithm assumes that the constraint system $\mathcal{S}$ is closed under Θ. Computing this closure corresponds to propagating data flow information locally within a program component. This step is relatively cheap, since program components are typically small (less than a few thousand lines of code).

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

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

5.1 Empty Constraint Simplification

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

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

corresponding grammar $G_t(\Theta(\mathcal{S}), E)$ where $E = \{\alpha^P\}$. An inspection of this grammar shows that the set of non-empty non-terminals is:

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

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

5.2 Unreachable Constraint Simplification

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

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

5.3 Removing ϵ-Constraints

A constraint of the form $[\alpha \leq \beta] \in \mathcal{S}$ 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 $\alpha \leq \tau$, $\mathtt{rng}(\alpha) \leq \gamma$, or $\gamma \leq \mathtt{dom}(\alpha)$ 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 P. In this system, the only upper bound for the set variable α^1 is the ϵ-constraint $[\alpha^1 \leq \alpha^a]$. Hence this algorithm replaces all occurrences of α^1 by α^a, which further simplifies this constraint system into:

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

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

5.4 Hopcroft's Algorithm

The previous algorithm *merges* set variables under certain circumstances, and only when they are related by an ϵ-constraint. We would like to identify more general circumstances under which set variables can be merged. To this end, we define a *valid unifier* for $\mathcal{S}$ to be an equivalence relation $\sim$ on the set variables of $\mathcal{S}$ such that we can merge the set variables

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

Figure 7: The *Hopcroft* algorithm

in each equivalence class of $\sim$ without changing the observable behavior of $\mathcal{S}$. Using a model-theoretic argument, we can show that an equivalence relation $\sim$ is a valid unifier for $\mathcal{S}$ if for all solutions $\rho \in \mathit{Soln}(\mathcal{S})$ there exists another solution $\rho' \in \mathit{Soln}(\mathcal{S})$ such that ρ' agrees with ρ on E and $\rho'(\alpha) = \rho'(\beta)$ for all $\alpha \sim \beta$.

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

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

Figure 7 describes sufficient conditions to ensure that ρ' is a solution of $\mathcal{S}$, and hence that $\sim$ is a valid unifier for $\mathcal{S}$. To produce an equivalence relation satisfying these conditions, we use a variant of Hopcroft's $O(n \lg n)$ time algorithm [12] for computing an equivalence relation on states in a DFA and then merge set variables according to their equivalence class.[8]

5.5 Simplification Benchmarks

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

We tested the algorithms on the constraint systems for nine program components on a 167MHz Sparc Ultra 1 with 326M of memory, using the MzScheme byte code compiler [9]. The results are described in figure 8. The second column gives the number of lines in each program component, and the third column gives the number of constraints in the original (unsimplified) constraint system after closing it under the rules Θ. The remaining columns describe the behavior of each simplification algorithm, presenting the factor by

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

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

Figure 8: Behavior of the constraint simplification algorithms.

which the number of constraints was reduced, and the time (in milliseconds) required for this simplification.

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

6 Componential Set-Based Analysis

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

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

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

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

3. Finally, to reconstruct the full analysis results for the program component that the programmer is focusing on, the analysis tool combines the constraint system from the

second step with the unsimplified constraint system for that component. It closes the resulting system under Θ, which yields appropriate value set invariants for each labeled expression in the component.

The new analysis can easily process programs that consist of many components. For its first step, it eliminates all those constraints that have only local relevance, thus producing a small combined constraint system for the entire program. As a result, the analysis tool can solve the combined system more quickly and using less space than traditional set-based analysis [10]. Finally, it recreates as much precision as traditional set-based analysis as needed on a per-component basis.

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

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

6.1 Benchmarks

We tested the componential analyses with five benchmark programs, ranging from 1,200 to 17,000 lines. For comparison purposes, we also analyzed each benchmark with the *standard* set-based analysis that performs no simplification. The analyses handled library functions in a context-sensitive, polymorphic manner according to the constraint derivation rules (*let*) and (*inst*) to avoid merging information between unrelated calls to these functions. The remaining functions were analyzed in a context-insensitive, monomorphic manner. The results are documented in figure 9.

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

The fifth column shows the time required to analyze each program from scratch, without using any existing constraint files.[9] 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 outweighed by the overhead of running the algorithm. The tradeoff may change as we analyze larger programs.

To test the responsiveness of the componential analyses in an interactive setting based on an analyze-debug-edit cycle, we re-analyzed each benchmark after changing a randomly chosen component in that benchmark. The re-analysis times are shown in the sixth column of figure 9. These times show an order-of-magnitude improvement in analysis times over the original, *standard* analysis, since the saved constraint files are used to avoid reanalyzing all of the unchanged program components. For example, the analysis of `zodiac`, which used

[9] These times exclude scanning and parsing time.

Program	lines	Analysis	Num. of constraints	Analysis time	Re-analysis time	Constraint file size (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

Figure 9: Behavior of the modular analyses.

to take over two minutes, now completes in under four seconds. Since practical debugging sessions using MrSpidey typically involve repeatedly analyzing the project each time the source code of one module is modified, *e.g.*, when a bug is identified and eliminated, using separate analysis substantially improves the usability of MrSpidey.

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

7 Efficient Polymorphic Analysis

The constraint simplification algorithms also enables an efficient polymorphic, or context-sensitive, analysis. To avoid merging information between unrelated calls to functions that are used in a polymorphic fashion, a polymorphic analysis duplicates the function's constraints at each call site. We extended MrSpidey with five polymorphic analyses. The first analysis is *copy*, which duplicates the constraint system for each polymorphic reference via a

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

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

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

straightforward implementation of the rules (*let*) and (*inst*).[10] The remaining four analyses are *smart* analyses that simplify the constraint system for each polymorphic definition.

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

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

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

8 Competitive Work

Fähndrich and Aiken [6] examine constraint simplification for an analysis based on a more complex constraint language. They develop a number of heuristic algorithms for constraint simplification, which they test on programs of up to 6000 lines. Their fastest approach

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

yields a factor of 3 saving in both time and space, but is slow in absolute times compared to other program analyses.

Pottier [17] 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 [20] 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, Smith and Trifonov [5] discuss constraint simplification in the context of type inference for objects. They present three algorithms for simplifying constraint systems, two of which which are similar to the *empty* and *ϵ-removal* algorithms, and the third is a special case of the *Hopcroft* algorithm. They do not present results on the cost or effectiveness of these algorithms.

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

9 Future Work

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

A promising approach for deriving such approximate constraint systems is to rely on a programmer-provided *signature* describing the behavior of a polymorphic function or module, and to derive the new constraint system from that signature. After checking the entailment condition to verify that signature-based constraints correctly approximates the behavior of the module, we could use those constraints in the remainder of the analysis. Since the signature-based constraints are 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.

A Proofs

A.1 Subject Reduction Proof

Main Lemma 3.2 *If* $\emptyset \vdash P : \alpha, \mathcal{S}$ *and* $\rho \models \mathcal{S}$ *and* $P \longmapsto^* \mathcal{E}[\ V^l\ ]$ *then* V *in* $const(\rho(l))$.

Proof: Lemma A.1 shows that standard reduction steps preserve entailment. Hence $\emptyset \vdash \mathcal{E}[\ V^l\ ] : \alpha, \mathcal{S}'$ where $\mathcal{S} \models \mathcal{S}'$. This derivation contains a subderivation concluding:

$$\frac{\Gamma \vdash V : \beta, \mathcal{S}_V}{\Gamma \vdash V^l : \beta, \mathcal{S}_V \cup \{\beta \leq l\}} \quad (label)$$

Since $\rho \models \mathcal{S}_V$, V *in* $const\,(\rho(l))$. ∎

Lemma A.1 (Subject Reduction for $\longmapsto$) *If* $\Gamma \vdash M_1 : \alpha, \mathcal{S}_1$ *and* $M_1 \longmapsto M_2$, *then* $\Gamma \vdash M_2 : \alpha, \mathcal{S}_2$ *where* $\mathcal{S}_1 \models \mathcal{S}_2$.

Proof: Follows from the Subject Reduction Lemma A.2 and the Replacement Lemma A.6. ∎

Lemma A.2 (Subject Reduction for $\longrightarrow$) *If* $\Gamma \vdash M_1 : \alpha, \mathcal{S}_1$ *and* $M_1 \longrightarrow M_2$, *then* $\Gamma \vdash M_2 : \alpha, \mathcal{S}_2$ *such that* $\mathcal{S}_1 \models \mathcal{S}_2$.

Proof: The proof proceeds by case analysis according to the relation $M_1 \longrightarrow M_2$.

- Suppose $M_1 \longrightarrow M_2$ via (β_v). Then:

$$\begin{aligned} M_1 &= ((\lambda^t x.N)\, V) \\ M_2 &= N[x \mapsto V] \end{aligned}$$

The typing derivation on M_1 must be of the following form:

$$\frac{\dfrac{\Gamma \cup \{x : \beta_x\} \vdash N : \beta_N, \mathcal{S}_N}{\Gamma \vdash (\lambda^t x.N) : \beta_t, \mathcal{S}_t}\ (abs) \quad \Gamma \vdash V : \beta_V, \mathcal{S}_V}{\Gamma \vdash M_1 : \alpha, \mathcal{S}_1} \quad (app)$$

where:

$$\begin{aligned} \mathcal{S}_t &= \mathcal{S}_N \cup \{t \leq \beta_t,\ \mathtt{dom}(\beta_t) \leq \beta_x,\ \beta_N \leq \mathtt{rng}(\beta_t)\} \\ \mathcal{S}_1 &= \mathcal{S}_N \cup \mathcal{S}_V \cup \{\beta_V \leq \mathtt{dom}(\beta_t), \mathtt{rng}(\beta_t) \leq \alpha\} \end{aligned}$$

By the Substitution Lemma A.3:

$$\Gamma \vdash M_2 : \beta_N, \mathcal{S}_2' \qquad \text{where} \qquad \mathcal{S}_N \cup \mathcal{S}_V \cup \{\beta_V \leq \beta_x\} \models \mathcal{S}_2'$$

By the Flow Lemma A.5:

$$\Gamma \vdash M_2 : \alpha, \mathcal{S}_2 \qquad \text{where} \qquad \mathcal{S}_2' \cup \{\beta_N \leq \alpha\} \models \mathcal{S}_2$$

Since $\mathcal{S}_1 \supseteq \mathcal{S}_N \cup \mathcal{S}_V$, we have that $\mathcal{S}_1 \models \mathcal{S}_N \cup \mathcal{S}_V$. Also:

$$\{\beta_V \leq \mathtt{dom}(\beta_t),\ \mathtt{dom}(\beta_t) \leq \beta_x\} \quad \models \quad \{\beta_V \leq \beta_x\}$$

and hence $\mathcal{S}_1 \models \mathcal{S}_2'$. Furthermore,

$$\{\beta_N \leq \mathtt{rng}(\beta_t),\ \mathtt{rng}(\beta_t) \leq \alpha\} \quad \models \quad \{\beta_N \leq \alpha\}$$

and therefore $\mathcal{S}_1 \models \mathcal{S}_2$, as required.

- Suppose $M_1 \longrightarrow M_2$ via (β_{let}). Then:

$$\begin{aligned} M_1 &= (\textbf{let}\ (x\ V)\ N) \\ M_2 &= N[x \mapsto V] \end{aligned}$$

The typing derivation on M_1 must be of the following form:

$$\frac{\begin{array}{c}\Gamma \vdash V : \alpha_V, \mathcal{S}_V \\ A = \mathit{Vars}(\mathcal{S}_V) \setminus (\mathit{FV}[\mathit{rng}(\Gamma)] \cup \mathit{Label}) \\ \Gamma \cup \{x : \forall A.\, (\alpha_V, \mathcal{S}_V)\} \vdash N : \beta, \mathcal{S}\end{array}}{\Gamma \vdash M_1 : \beta, \mathcal{S}} \quad (\mathit{let})$$

By the Polymorphic Substitution Lemma A.4:

$$\Gamma \vdash M_2 : \beta, \mathcal{S}$$

as required.

- Suppose $M_1 \longrightarrow M_2$ via $(\mathit{unlabel})$. Then:

$$\begin{aligned} M_1 &= V^l \\ M_2 &= V \end{aligned}$$

The typing derivation on M_1 must be of the following form:

$$\frac{\Gamma \vdash V : \alpha, \mathcal{S}_V}{\Gamma \vdash V^l : \alpha, \mathcal{S}_V \cup \{\alpha \leq l\}} \quad (\mathit{label})$$

Hence $\Gamma \vdash V : \alpha, \mathcal{S}_V$.

∎

Lemma A.3 (Substitution) *If*

$$\begin{array}{c}\Gamma \cup \{x : \alpha_x\} \vdash N : \alpha_N, \mathcal{S}_N \\ \Gamma \vdash V : \alpha_V, \mathcal{S}_V\end{array}$$

then

$$\Gamma \vdash N[x \mapsto V] : \alpha_N, \mathcal{S} \quad \textit{where} \quad \mathcal{S}_N \cup \mathcal{S}_V \cup \{\alpha_V \leq \alpha_x\} \models \mathcal{S}$$

Proof: The proof proceeds by induction on the length of the proof that:

$$\Gamma \cup \{x : \alpha_x\} \vdash N : \alpha_N, \mathcal{S}_N$$

and by case analysis on the last step.

- (*var*): Suppose $N = x$. Then $\alpha_N = \alpha_x$, $S_N = \emptyset$, $N[x \mapsto V] = V$ and this case holds via the Flow Lemma A.5.

  Otherwise $N = y \neq x$, $N[x \mapsto V] = y$, $\Gamma \vdash y : \Gamma(y), \emptyset$ and this case trivially holds since $\mathcal{S} = \emptyset$.

- (*const*): This case trivially holds since $\mathcal{S} = \emptyset$.

- (*abs*): In this case $N = (\lambda^t y.M)$, and the typing derivation is:

$$\frac{\Gamma \cup \{x : \alpha_x, y : \alpha_y\} \vdash M : \alpha_M, \mathcal{S}_M}{\Gamma \cup \{x : \alpha_x\} \vdash (\lambda^t y.M) : \alpha_N, \mathcal{S}_N} \quad (abs)$$

  where:

$$\mathcal{S}_N = \mathcal{S}_M \cup \{t \leq \alpha_N,\ \mathtt{dom}(\alpha_N) \leq \alpha_y,\ \alpha_M \leq \mathtt{rng}(\alpha_N)\}$$

  If $x = y$, then $N[x \mapsto V] = N$ and this case holds.

  Otherwise, $N[x \mapsto V] = (\lambda^t y.M[x \mapsto V])$. By induction,

$$\Gamma \cup \{x : \alpha_x, y : \alpha_y\} \vdash M[x \mapsto V] : \alpha_M, \mathcal{S}'_M$$

  where:

$$\mathcal{S}_M \cup \mathcal{S}_V \cup \{\alpha_V \leq \alpha_x\} \models \mathcal{S}'_M$$

  Hence, via (*abs*):

$$\Gamma \vdash N[x \mapsto V] : \alpha_N, \mathcal{S}$$

  where:

$$\mathcal{S} = \mathcal{S}'_M \cup \{t \leq \alpha_N,\ \mathtt{dom}(\alpha_N) \leq \alpha_y,\ \alpha_M \leq \mathtt{rng}(\alpha_N)\}$$

  Since $\mathcal{S}_N \cup \mathcal{S}_V \cup \{\alpha_V \leq \alpha_x\} \models \mathcal{S}'_M$, the lemma holds for this case.

- (*app*): In this case $M = (M_1\ M_2)$, and the typing derivation is:

$$\frac{\Gamma \cup \{x : \alpha_x\} \vdash M_i : \beta_i, \mathcal{S}_i}{\Gamma \cup \{x : \alpha_x\} \vdash (M_1\ M_2) : \alpha_N, \mathcal{S}_N} \quad (app)$$

  where

$$\mathcal{S}_N = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\beta_2 \leq \mathtt{dom}(\beta_1), \mathtt{rng}(\beta_1) \leq \alpha_N\}$$

  By induction,

$$\Gamma \vdash M_i[x \mapsto V] : \beta_i, \mathcal{S}'_i$$

  where

$$\mathcal{S}_i \cup \mathcal{S}_V \cup \{\alpha_V \leq \alpha_x\} \models \mathcal{S}'_i$$

Hence

$$\Gamma \vdash M[x \mapsto V] : \alpha_N, \mathcal{S}$$

where

$$\mathcal{S} = \mathcal{S}'_1 \cup \mathcal{S}'_2 \cup \{\beta_2 \leq \mathtt{dom}(\beta_1), \mathtt{rng}(\beta_1) \leq \alpha_N\}$$

Obviously, $\mathcal{S}_N \cup \mathcal{S}_V \cup \{\alpha_V \leq \alpha_x\} \models \mathcal{S}$, and the lemma holds for this case.

- (*label*): This case is straightforward.

∎

Lemma A.4 (Polymorphic Substitution) *If*

$$\begin{gathered}\Gamma \vdash V : \alpha_V, \mathcal{S}_V \\ A = \mathit{Vars}(\mathcal{S}_V) \setminus (FV[\mathit{rng}(\Gamma)] \cup \mathit{Label}) \\ \Gamma \cup \{x : \forall A.(\alpha_V, \mathcal{S}_V)\} \vdash N : \alpha_N, \mathcal{S}_N\end{gathered}$$

then

$$\Gamma \vdash N[x \mapsto V] : \alpha_N, \mathcal{S}_N$$

Proof: The proof proceeds by induction on the length of the proof that:

$$\Gamma \cup \{x : \forall A.(\alpha_V, \mathcal{S}_V)\} \vdash N : \alpha_N, \mathcal{S}_N$$

and by case analysis on the last step.

- (*var*): Suppose $N = x$. Let ψ be the substitution of fresh variables for A used in the (*inst*) derivation rule for N. Then $\alpha_N = \psi(\alpha_V)$, $S_N = \psi(\mathcal{S}_V)$ and $N[x \mapsto V] = V$. Hence if $\mathcal{D}$ is the derivation concluding $\Gamma \vdash V : \alpha_V, \mathcal{S}_V$, then $\psi(\mathcal{D})$ is an analogous derivation concluding $\Gamma \vdash N[x \mapsto V] : \alpha_N, \mathcal{S}_N$.

  Otherwise $N = y \neq x$, $N[x \mapsto V] = y$ and this case trivially holds since $\mathcal{S} = \emptyset$.

- (*const*): This case trivially holds since $\mathcal{S} = \emptyset$.

- (*abs*): In this case $N = (\lambda^t y.M)$, and the typing derivation is:

$$\frac{\Gamma \cup \{x : \forall A.(\alpha_V, \mathcal{S}_V), y : \alpha_y\} \vdash M : \alpha_M, \mathcal{S}_M}{\Gamma \cup \{x : \forall A.(\alpha_V, \mathcal{S}_V)\} \vdash (\lambda^t y.M) : \alpha_N, \mathcal{S}_N} \quad (abs)$$

  where:

$$\mathcal{S}_N = \mathcal{S}_M \cup \{t \leq \alpha_N,\ \mathtt{dom}(\alpha_N) \leq \alpha_y,\ \alpha_M \leq \mathtt{rng}(\alpha_N)\}$$

  If $x = y$, then $N[x \mapsto V] = N$ and this case holds.

  Otherwise, $N[x \mapsto V] = (\lambda^t y.M[x \mapsto V])$. By induction,

$$\Gamma \cup \{x : \forall A.(\alpha_V, \mathcal{S}_V), y : \alpha_y\} \vdash M[x \mapsto V] : \alpha_M, \mathcal{S}_M$$

  Hence, via (*abs*):

$$\Gamma \vdash N[x \mapsto V] : \alpha_N, \mathcal{S}_N$$

  as required.

- (*app*): In this case $M = (M_1\ M_2)$, and the typing derivation is:

$$\frac{\Gamma \cup \{x : \forall A.(\alpha_V, \mathcal{S}_V)\} \vdash M_i : \beta_i, \mathcal{S}_i}{\Gamma \cup \{x : \forall A.(\alpha_V, \mathcal{S}_V)\} \vdash (M_1\ M_2) : \alpha_N, \mathcal{S}_N} \quad (app)$$

  where

$$\mathcal{S}_N = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\beta_2 \leq \mathtt{dom}(\beta_1), \mathtt{rng}(\beta_1) \leq \alpha_N\}$$

  By induction,

$$A \vdash M_i[x \mapsto V] : \beta_i, \mathcal{S}_i$$

  Hence

$$\Gamma \vdash M[x \mapsto V] : \alpha_N, \mathcal{S}_N$$

  as required.

- (*label*): This case is straightforward.

∎

Lemma A.5 (Flow) *If $\Gamma \vdash M : \alpha, \mathcal{S}$ then for all $\beta \in SetVar$ not used in $\mathcal{S}$, $\Gamma \vdash M : \beta, \mathcal{S}'$ with $\mathcal{S} \cup \{\alpha \leq \beta\} \models \mathcal{S}'$.*

Proof: Replace α by β in the derivation $\Gamma \vdash M : \alpha, \mathcal{S}$. ∎

Lemma A.6 (Replacement) *If:*

1. *D is a deduction concluding $\Gamma \vdash \mathcal{E}[\ M_1\ ] : \alpha, \mathcal{S}_1$,*
2. *D_1 is a subdeduction of D concluding $\Gamma' \vdash M_1 : \beta, \mathcal{S}_1'$,*
3. *D_1 occurs in D in the position corresponding to the hole ($[\ ]$) in $\mathcal{E}$, and*
4. *$\Gamma' \vdash M_2 : \beta, \mathcal{S}_2'$ where $\mathcal{S}_1' \models \mathcal{S}_2'$,*

then $\Gamma \vdash \mathcal{E}[\ M_2\ ] : \alpha, \mathcal{S}_2$ where $\mathcal{S}_1 \models \mathcal{S}_2$.

Proof: Follows Hindley and Seldin [11:page 181]. ∎

Lemma A.7 (Value Typing) *If $\Gamma \vdash V : \alpha, \mathcal{S}$ and $\rho \models \mathcal{S}$, then V in $const(\rho(\alpha))$.*

A.2 Proofs for Computing Set-Based Analysis

Lemma 3.3 *Every constraint system has a solution that is least with respect to $\sqsubseteq_s$.*

Proof: Let $\mathcal{S}$ be a simple constraint system, and define $\rho = \sqcap_s Soln(\mathcal{S})$, using the pointwise extension of $\sqcap_s$ to set environments. We prove by case analysis that ρ satisfies any constraint $\mathcal{C} \in \mathcal{S}$.

- The case where $\mathcal{C} = [\alpha \leq \beta]$ follows from Lemma A.8.

- Suppose $\mathcal{C} = [c \leq \beta]$. Then $c \in const\,(\rho'(\beta))$ for all $\rho' \in Soln(\mathcal{S})$, therefore $c \in const\,(\rho(\beta))$ and $\rho \models \mathcal{C}$.
- Suppose $\mathcal{C} = [\alpha \leq \mathtt{rng}(\beta)]$. Then:

$$\begin{array}{rcll}
\rho(\alpha) & = & \displaystyle\sqcap_s \atop {\scriptstyle \rho' \in Soln(\mathcal{S})} & \rho'(\alpha) \\
& \sqsubseteq & \displaystyle\sqcap_s \atop {\scriptstyle \rho' \in Soln(\mathcal{S})} & \rho'(\mathtt{rng}(\beta)) \\
& & \multicolumn{2}{l}{\text{by Lemma A.8, since } \rho'(\alpha) \sqsubseteq \rho'(\mathtt{rng}(\beta))} \\
& = & rng \left(\displaystyle\sqcap_s \atop {\scriptstyle \rho' \in Soln(\mathcal{S})} \right. & \left. \rho'(\beta) \right) \qquad \text{by definition of } \sqcap_s \\
& = & rng\,(\rho(\beta)) & \\
& = & \rho(\mathtt{rng}(\beta)) &
\end{array}$$

  Hence $\rho \models \alpha \leq \mathtt{rng}(\beta)$.

The remaining cases are similar. Hence the set of environments satisfying $\mathcal{S}$ has a least element $\sqcap_s \{\rho \mid \rho \models \mathcal{S}\}$. ∎

Lemma A.8 *Let I be an index set, and let $x_i, y_i \in \mathcal{D}$ for all $i \in I$.*

- *If $x_i \sqsubseteq y_i$ for all $i \in I$, then:*

$$\sqcap_{i \in I}\ x_i \ \sqsubseteq\ \sqcap_{i \in I}\ y_i$$

$$\sqcup_{i \in I}\ x_i \ \sqsubseteq\ \sqcup_{i \in I}\ y_i$$

- *If $x_i \sqsubseteq_s y_i$ for all $i \in I$, then:*

$$\sqcap_{i \in I}\ x_i \ \sqsubseteq_s\ \sqcap_{i \in I}\ y_i$$

$$\sqcup_{i \in I}\ x_i \ \sqsubseteq_s\ \sqcup_{i \in I}\ y_i$$

Proof: The proof is based on the interpretation of $\mathcal{D}$ as the set of total functions

$$f : \{0,1\}^* \longrightarrow \mathcal{P}(Const)$$

and proceeds by showing the appropriate relation holds between the sets of constant elements at any path in $\{0,1\}^*$. ∎

Theorem 3.6 *If $P \in \Lambda^0$ and $\emptyset \vdash P : \alpha, \mathcal{S}$ then $sba(P)(l) = \{c \mid \mathcal{S} \vdash_\Theta c \leq l\}$.*

Proof: Follows from Definition 3.4 and Lemma A.9 ∎

Lemma A.9 *For any constraint system $\mathcal{S}$, $const(LeastSoln(\mathcal{S})(\alpha)) = \{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha\}$.*

Proof: We proof this theorem via an alternative definition of $LeastSoln(\mathcal{S})$ in terms of the inference rules Θ. Specifically, $LeastSoln'$ is defined by:

$$LeastSoln'(\mathcal{S}) = \bigsqcup_{i=0,\infty}^{s} F^i(\bot_s)$$

where

- $\bot_s$ denotes the environment $\bot_s(\alpha) = \bot_s$,
- the functional F is:

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

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

$$\mathcal{S} \vdash_\Theta \{\delta_i \leq \delta_{i+1}, \mid 1 \leq i < n\}$$

Let ρ be a fixpoint of F. We prove $\rho \models \mathcal{S}$ 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 corresponding components of $\rho(\alpha)$ and $\rho(\beta)$. For the first component, by (s_1):

$$\begin{array}{rrcl} & \{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha\} & \subseteq & \{c \mid \mathcal{S} \vdash_\Theta c \leq \beta\} \\ \Rightarrow & const\,(\rho(\alpha)) & \sqsubseteq & const\,(\rho(\beta)) \end{array}$$

  For the second (domain) component, by (s_3):

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

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

$$\begin{array}{rrcl} & [\mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha)] & \Rightarrow & [\mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta)] \\ \Rightarrow & \{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha)\} & \subseteq & \{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta)\} \\ \Rightarrow & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha)\} & \sqsubseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta)\} \\ \Rightarrow & rng\,(\rho(\alpha)) & \sqsubseteq & rng\,(\rho(\beta)) \end{array}$$

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

- Suppose $\mathcal{C} = [c \leq \beta]$:

$$\begin{array}{rcl} \rho(\beta) & \sqsupseteq & \langle \{c \mid \mathcal{S} \vdash_\Theta c \leq \beta\}, \top, \bot \rangle \\ & \sqsupseteq & \langle \{c\}, \top, \bot \rangle \\ & = & \rho(c) \end{array}$$

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

$$\begin{array}{rrcl} & \{\rho(\alpha)\} & \subseteq & \{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta)\} \\ \Rightarrow & \rho(\alpha) & \sqsubseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\beta)\} \\ & & = & \rho(\mathtt{rng}(\beta)) \end{array}$$

- Suppose $\mathcal{C} = [\mathtt{rng}(\alpha) \leq \beta]$. Then

$$\begin{array}{rrcll} & \rho(\beta) & \sqsupseteq & \rho(\gamma) & \forall \mathcal{S} \vdash_\Theta \gamma \leq \beta \\ \Rightarrow & \rho(\beta) & \sqsupseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \beta\} & \\ & & \sqsupseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{rng}(\alpha)\} & \text{by } (s_4) \\ & & = & \rho(\mathtt{rng}(\alpha)) & \end{array}$$

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

$$\begin{array}{rcl} \rho(\alpha) & \sqsubseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \gamma \leq \mathtt{dom}(\beta)\} \\ & \sqsubseteq & \bigsqcup\{\rho(\gamma) \mid \mathcal{S} \vdash_\Theta \beta \leq^* \delta,\ \gamma \leq \mathtt{dom}(\delta)\} \\ & = & \rho(\mathtt{dom}(\beta)) \end{array}$$

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

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

Thus fixpoints of F are solutions of $\mathcal{S}$, and hence $\mathit{LeastSoln}'(\mathcal{S})$ is a solution of $\mathcal{S}$. Therefore:

$$\mathit{const}\,(\mathit{LeastSoln}(\mathcal{S})(\alpha)) \subseteq \mathit{const}\,(\mathit{LeastSoln}(\mathcal{S})(\alpha)) = \{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha\}$$

Since the rules Θ are sound, $\{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha\} \subseteq \mathit{const}\,(\mathit{LeastSoln}(\mathcal{S})(\alpha))$. Hence

$$\mathit{const}\,(\mathit{LeastSoln}(\mathcal{S})(\alpha)) = \{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha\}$$

as required.

∎

A.3 Proofs for Observable Equivalence

The proofs of this section require a number of auxiliary definitions.

Definition A.10. (*path, arity,* $\leq_i$, `sel`, $\overline{\mathcal{C}}$, *upper/lower bound,* $\mathcal{I}$)

- A *path* p is a sequence of the constructors `dom` and `rng`.
- The *arity* of a path p, denoted πp, is the number of $\texttt{dom}(\cdot)$'s in p, taken modulo 2. If πp is 0, we say p is monotonic, otherwise p is anti-monotonic.
- The relations $\leq_0$ and $\leq_1$ denote $\leq$ and its inverse respectively.
- The variable `sel` ranges over paths of length 1.
- We use the notation $\overline{\mathcal{C}}$ to map compound constraints of the form $[\alpha \leq_i p(\beta)]$ to a corresponding simple constraint system. In each case in the definition, γ is a fresh type variable, and $\tau \notin \mathit{SetVar}$.

$$\begin{array}{rcl}
\overline{\alpha \leq \beta} & = & \{\alpha \leq \beta\} \\
\overline{[\alpha \leq \texttt{rng}(\tau)]} & = & \{\alpha \leq \texttt{rng}(\gamma)\} \cup \overline{[\gamma \leq \tau]} \\
\overline{[\alpha \leq \texttt{dom}(\tau)]} & = & \{\alpha \leq \texttt{dom}(\gamma)\} \cup \overline{[\tau \leq \gamma]} \\
\overline{[\texttt{rng}(\tau) \leq \alpha]} & = & \{\texttt{rng}(\gamma) \leq \alpha\} \cup \overline{[\tau \leq \gamma]} \\
\overline{[\texttt{dom}(\tau) \leq \alpha]} & = & \{\texttt{dom}(\gamma) \leq \alpha\} \cup \overline{[\gamma \leq \tau]}
\end{array}$$

- An *upper bound* for α is a constraint of the form $p(\alpha) \leq_i \tau$, where $\pi(p) \neq i$.
- An *lower bound* for α is a constraint of the form $p(\alpha) \leq_i \tau$, where $\pi(p) = i$.
- The function $\mathcal{I}$ is defined as:

$$\begin{array}{rcl}
\mathcal{I} : \mathcal{D} \times \mathit{path} & \longrightarrow & \mathcal{D} \\
\mathcal{I}(X, \epsilon) & = & X \\
\mathcal{I}(X, \texttt{dom}(p)) & = & \mathcal{I}(\langle \emptyset, X, \bot_s \rangle, p) \\
\mathcal{I}(X, \texttt{rng}(p)) & = & \mathcal{I}(\langle \emptyset, \bot_s, X \rangle, p)
\end{array}$$

∎

Lemma 4.1 *If* $\emptyset \vdash_{\cong} P : \alpha, \mathcal{S}$, *then* $\mathit{sba}(P)(l) = \mathit{const}(\mathit{LeastSoln}(\mathcal{S})(l))$. .

Proof: Follows from Lemma A.11. ∎

Lemma A.11 *If* $\Gamma \vdash_{\cong} M : \alpha, \mathcal{S}_1$, *and* $E = FV[\mathit{rng}(\Gamma)] \cup \{\alpha\} \cup \mathit{Label}$, *then there exists* $\mathcal{S}_2$ *such that* $\Gamma \vdash M : \alpha, \mathcal{S}_2$ *and* $\mathcal{S}_1 \cong_E \mathcal{S}_2$.

Proof: The proof proceeds by induction on the derivation $\Gamma \vdash_{\cong} M : \alpha, \mathcal{S}_1$, and by case analysis on the last step in the derivation.

- If the last step in the derivation $\Gamma \vdash_{\cong} M : \alpha, \mathcal{S}_1$ uses a derivation rule other that $(\cong)$, then the lemma holds based on the induction hypothesis.
- 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 $\mathcal{S}_3 \cong_E \mathcal{S}_4$. Hence the lemma holds.

∎

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

Proof of Lemma 4.2: The soundness of $\Theta \cup \Psi$ is straightforward. To demonstrate the completeness of $\Theta \cup \Psi$, we assume $\mathcal{S} \models \mathcal{C}$ and prove that $\mathcal{S} \vdash_{\Psi\Theta} \mathcal{C}$ by case analysis on $\mathcal{C}$.

- Suppose $\mathcal{C} = [p(\alpha) \leq q(\beta)]$, where p and q are paths. Let c be a constant not used in $\mathcal{S}$ or $\mathcal{C}$, let γ and δ be fresh variables, and let:

$$\begin{array}{rcl} \mathcal{S}_p & = & \overline{[\gamma \leq p(\alpha)]} \\ \mathcal{S}_q & = & \overline{[q(\beta) \leq \delta]} \\ \mathcal{S}' & = & \mathcal{S} \cup \{c \leq \gamma\} \cup \mathcal{S}_p \cup \mathcal{S}_q \\ \rho & = & \mathit{LeastSoln}(\mathcal{S}') \end{array}$$

  Since $\rho \models \mathcal{C}$, by Lemma A.15:

$$\rho \models \{c \leq \gamma, \gamma \leq p(\alpha), p(\alpha) \leq q(\beta), q(\beta) \leq \delta\}$$

  Hence $\rho \models c \leq \delta$, and by Lemma A.9, $\mathcal{S}' \vdash_\Theta c \leq \delta$. This last constraint must be derived from the constraints $\mathcal{S} \cup \mathcal{S}_p \cup \mathcal{S}_q \vdash_\Theta \gamma \leq^* \delta$ via (s_1), and hence via the Tracing Lemma A.12, $\mathcal{S} \vdash_{\Psi\Theta} p(\alpha) \leq q(\beta)$ as required.

- Suppose $\mathcal{C} = [c \leq q(\beta)]$. Let δ be a fresh variable, and let:

$$\begin{array}{rcl} \mathcal{S}' & = & \mathcal{S} \cup \overline{[q(\beta) \leq \delta]} \\ \rho & = & \mathit{LeastSoln}(\mathcal{S}') \end{array}$$

  Since $\rho \models \mathcal{C}$:

$$\rho \models \{c \leq q(\beta), q(\beta) \leq \delta\}$$

  Hence $\rho \models c \leq \delta$, and by Lemma A.9, $\mathcal{S}' \vdash_\Theta c \leq \delta$. This last constraint must be derived because there exists some γ such that $[c \leq \gamma] \in \mathcal{S}$ and $\mathcal{S}' \vdash_\Theta \gamma \leq^* \delta$. By the One Side Tracing Lemma A.13, $\mathcal{S} \vdash_{\Psi\Theta} \gamma \leq q(\beta)$. Hence $\mathcal{S} \vdash_{\Psi\Theta} c \leq q(\beta)$.

∎

Lemma A.12 (Tracing Lemma) *Let $\mathcal{S}$ is a simple constraint system. If*

$$\begin{array}{c}\gamma \notin \mathit{Vars}(\mathcal{S}) \cup \{\alpha, \beta, \delta\} \\ \delta \notin \mathit{Vars}(\mathcal{S}) \cup \{\alpha, \beta, \gamma\} \\ \mathcal{S} \cup \overline{[\gamma \leq p(\alpha)]} \cup \overline{[q(\beta) \leq \delta]} \vdash_\Theta \gamma \leq^* \delta\end{array}$$

then:

$$\mathcal{S} \vdash_{\Psi\Theta} [p(\alpha) \leq q(\beta)]$$

Proof: The proof proceeds by induction on the length of p. Let:

$$\begin{array}{rcl}\mathcal{S}_p & = & \overline{[\gamma \leq p(\alpha)]} \\ \mathcal{S}_q & = & \overline{[q(\beta) \leq \delta]} \\ \mathcal{S}' & = & \mathcal{S} \cup \mathcal{S}_p \cup \mathcal{S}_q\end{array}$$

We proceed by case analysis on the possibilities for $\mathcal{S}' \vdash_\Theta \gamma \leq^* \delta$.

- Suppose $\mathcal{S}' \vdash_\Theta \{\gamma \leq^* \delta\}$ because $\mathcal{S}' \vdash_\Theta \{\gamma \leq \gamma',\ \gamma' \leq^* \delta',\ \delta' \leq \delta\}$.

  Each variable t in $[\gamma' \leq^* \delta']$ has both upper and lower ϵ-constraints $\{t_1 \leq t, t \leq t_2\} \subseteq \Theta(\mathcal{S}')$, and hence by lemma A.16 (1), $t \notin (\mathit{Vars}(\mathcal{S}_p) \cup \mathit{Vars}(\mathcal{S}_q)) \setminus \{\alpha, \beta\}$. Hence by Lemma A.16 (2), the constraints $[\gamma \leq^* \delta]$ cannot depend on $\mathcal{S}_p$ or $\mathcal{S}_q$, and thus $\mathcal{S} \vdash_\Theta \{\gamma' \leq^* \delta'\}$.

  Also by Lemma A.16 (2), $[\gamma \leq \gamma']$ cannot depend on $\mathcal{S}_q$, and therefore $\mathcal{S} \cup \mathcal{S}_p \vdash_\Theta \{\gamma \leq \gamma'\}$. The One Side Tracing Lemma A.13 then implies that $\mathcal{S} \vdash_{\Pi\Theta} p(\alpha) \leq \gamma'$. A similar argument shows that $\mathcal{S} \vdash_{\Pi\Theta} \delta' \leq q(\beta)$.

  Hence, via multiple applications of (*trans*):

  $$\mathcal{S} \vdash_{\Pi\Theta} p(\alpha) \leq q(\beta)$$

  as required.

- Otherwise $\mathcal{S}' \vdash_\Theta \gamma \leq \delta$. This constraint must be inferred by one of the inference rules (s_4) or (s_5). Assume the constraint is inferred via (s_4). Then for some δ':

  $$\mathcal{S}' \vdash_\Theta \{\gamma \leq \mathtt{rng}(\delta'),\ \mathtt{rng}(\delta') \leq \delta\}$$

  An examination of the inference system Θ shows that Θ never infers constraints of the form $[\mathtt{rng}(\delta') \leq \delta]$. Hence $[\mathtt{rng}(\delta') \leq \delta] \in \mathcal{S}'$, and therefore $\delta \notin \mathit{Vars}(\mathcal{S} \cup \mathcal{S}_p)$, $[\mathtt{rng}(\delta') \leq \delta] \in \mathcal{S}_q$. Thus $q = \mathtt{rng}.q'$ and $\mathcal{S}_q = \{\mathtt{rng}(\delta') \leq \delta\} \cup \overline{[q'(\beta) \leq \delta]}$.

  The constraint $\mathcal{S}' \vdash_\Theta \gamma \leq \mathtt{rng}(\delta')$ must be derived from the constraints $[\gamma \leq \mathtt{rng}(\gamma')] \in \mathcal{S}'$ and $\mathcal{S}' \vdash_\Theta \gamma' \leq^* \delta'$ via multiple applications of (s_2). Since $\gamma \notin \mathit{Vars}(\mathcal{S} \cup \mathcal{S}_q)$, $[\gamma \leq \mathtt{rng}(\gamma')] \in \mathcal{S}_p$. Thus $p = \mathtt{rng}.p'$ and $\mathcal{S}_p = \{\gamma \leq \mathtt{rng}(\gamma')\} \cup \overline{[\gamma' \leq p'(\alpha)]}$.

  By Lemma A.17 (2), the only constraints in $\Theta(\mathcal{S}')$ whose derivation relies on $[\mathtt{rng}(\delta') \leq \delta]$ or $[\gamma \leq \mathtt{rng}(\gamma)']$ must be lower bounds on δ or upper bounds on γ, respectively. Since the constraints $[\gamma' \leq^* \delta']$ are of neither form, it follows that:

  $$\mathcal{S} \cup \overline{[\gamma' \leq p'(\alpha)]} \cup \overline{[q'(\beta) \leq \delta']} \vdash_\Theta \gamma' \leq^* \delta'$$

Hence by induction:

$$\mathcal{S} \vdash_{\Psi\Theta} p'(\alpha) \leq q'(\beta)$$

and the constraint:

$$\mathcal{S} \vdash_{\Psi\Theta} p(\alpha) \leq q(\beta)$$

follows via an application of the (*compat*) rule.

The case where $\mathcal{S}' \vdash_{\Theta} \gamma \leq \delta$ via (s_5) follows by similar reasoning.

∎

Lemma A.13 (One Side Tracing Lemma) *Let $\mathcal{S}$ is a simple constraint system not mentioning α. If*

$$\mathcal{S} \cup \overline{[\alpha \leq_i p(\beta)]} \vdash_{\Theta} \alpha \leq_i \delta$$

then

$$\mathcal{S} \vdash_{\Pi\Theta} p(\beta) \leq_i \delta$$

Proof: The proof requires a stronger invariant that if:

$$\mathcal{S} \cup \overline{[\alpha \leq_i p(\beta)]} \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$$

where $j = (i + \pi q) \bmod 2$, then

$$\mathcal{S} \vdash_{\Pi\Theta} q(p(\beta)) \leq_j \delta$$

The proof of this invariant proceeds by induction on the length of p.

- If $p = \epsilon$, then $\mathcal{S}' = \mathcal{S} \cup \{\alpha \leq_i \beta\}$. Since $\mathcal{S} \vdash_{\Pi} \mathcal{S} \cup \{\beta \leq_i \beta\}$, any constraint that can be inferred on α can also be inferred on β via a similar derivation. Hence:

  $$\mathcal{S} \vdash_{\Pi\Theta} q(\beta) \leq_j \delta$$

- Otherwise $p = \texttt{sel}.p'$, and for some γ:

  $$\mathcal{S}' = \mathcal{S} \cup \overline{[\gamma \leq_k p'(\beta)]} \cup \{\alpha \leq_i \texttt{sel}(\gamma)\}$$

  where $k = (i + \pi(\texttt{sel})) \bmod 2$. Since $\mathcal{S}' \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$, by the One Step Tracing Lemma A.14:

  $$\mathcal{S} \cup \overline{[\gamma \leq_k p'(\beta)]} \vdash_{\Pi\Theta} q(\texttt{sel}(\gamma)) \leq_j \delta$$

  and hence by induction:

  $$\mathcal{S} \vdash_{\Pi\Theta} q(\texttt{sel}(p'(\beta))) \leq_j \delta$$

  as required.

∎

Lemma A.14 (One Step Tracing Lemma) *Suppose $\mathcal{S}$ is a simple constraint collection and α does not occur in $\mathcal{S}$. If*

$$\mathcal{S} \cup \{\alpha \leq_i \mathtt{sel}(\beta)\} \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$$

where $j = (i + \pi q) \bmod 2$ and $\delta \not\equiv \alpha$, then

$$\mathcal{S} \vdash_{\Pi\Theta} q(\mathtt{sel}(\beta)) \leq_j \delta$$

Proof: Let $\mathcal{S}' = \mathcal{S} \cup \{\alpha \leq_i \mathtt{sel}(\beta)\}$. The proof proceeds by induction on the derivation $\mathcal{S}' \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$ and by case analysis on the last step in the derivation.

- Suppose $\mathcal{S}' \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$ via the rule (*trans*) based on the antecedents:

  $$\mathcal{S}' \vdash_{\Pi\Theta} \{q(\alpha) \leq_j \delta',\ \delta' \leq_j \delta\}$$

  By Lemma A.17 (1), α cannot have both upper and lower bounds in $\Pi\Theta(\mathcal{S}')$, hence $\delta' \not\equiv \alpha$. By induction $\mathcal{S} \vdash_{\Pi\Theta} q(\mathtt{sel}(\beta)) \leq_j \delta'$.

  Furthermore, by Lemma A.17 (2), $\mathcal{S} \vdash_{\Pi\Theta} \delta' \leq_j \delta$, and therefore via (*trans*):

  $$\mathcal{S} \vdash_{\Pi\Theta} q(\mathtt{sel}(\beta)) \leq_j \delta$$

  as required.

- Suppose $\mathcal{S}' \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$ via one of the (*compose*) rules based on the antecedents:

  $$\mathcal{S}' \vdash_{\Pi\Theta} \{q'(\alpha) \leq_k \alpha',\ \mathtt{sel}'(\alpha') \leq_j \delta\}$$

  where $q = \mathtt{sel}'.q'$ and $k = (j + \pi(\mathtt{sel}')) \bmod 2$. By Lemma A.17 (1), α cannot have both upper and lower bounds in $\Pi\Theta(\mathcal{S}')$, hence $\alpha' \not\equiv \alpha$. Hence applying the induction hypothesis to $\mathcal{S}' \vdash_{\Pi\Theta} q'(\alpha) \leq_k \alpha'$ shows that:

  $$\mathcal{S} \vdash_{\Pi\Theta} q'(\mathtt{sel}(\beta)) \leq_k \alpha'$$

  By Lemma A.17 (2), $\mathcal{S} \vdash_{\Pi\Theta} \mathtt{sel}'(\alpha') \leq_j \delta$, and therefore via one of the (*compose*) rules:

  $$\mathcal{S} \vdash_{\Pi\Theta} \mathtt{sel}'(q'(\mathtt{sel}(\beta))) \leq_j \delta$$

  as required.

- The case where $\mathcal{S}' \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$ via (s_2) or (s_3) are subsumed by the (*compose*) case.

There are no other ways to infer that $\mathcal{S}' \vdash_{\Pi\Theta} q(\alpha) \leq_j \delta$.

∎

Lemma A.15 *For $\mathcal{C} = [\alpha \leq_i p(\beta)]$, $\overline{\mathcal{C}} \models \mathcal{C}$.*

Proof: By induction on the size of $\mathcal{C}$ and case analysis on the definition of $\overline{\mathcal{C}}$. ∎

Lemma A.16 *Let $\mathcal{S}$ be a simple constraint system not mentioning γ and let $\mathcal{S}_p = \overline{[\gamma \leq_i p(\alpha)]}$.*

1. *Each $\beta \in Vars(\mathcal{S}_p) \setminus \{\alpha\}$ cannot have both upper and lower ϵ-constraints in $\Theta(\mathcal{S} \cup \mathcal{S}_p)$.*
2. *Each constraint in $\Theta(\mathcal{S} \cup \mathcal{S}_p) \setminus \Theta(\mathcal{S})$ must mention a variable in $Vars(\mathcal{S}_p) \setminus \{\alpha\}$.*

Lemma A.17 *Let $\mathcal{S}$ be a simple constraint system not mentioning γ and let $\mathcal{S}_p = \{\gamma \leq_i \texttt{sel}(\alpha)\}$.*

1. *γ cannot have both upper and lower ϵ-constraints in $\Theta(\mathcal{S} \cup \mathcal{S}_p)$.*
2. *Each constraint in $\Theta(\mathcal{S} \cup \mathcal{S}_p) \setminus \Theta(\mathcal{S})$ must be an upper bound (if $i = 0$) or a lower bound (if $i = 1$) on γ.*

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

Proof: If $\rho \in Soln(\mathcal{S}) \mid_E$, then

$$\begin{array}{rcll} \rho & \in & Soln(\Psi\Theta(\mathcal{S})) \mid_E & \text{by Lemma 4.2} \\ & \subseteq & Soln(\Psi\Theta(\mathcal{S}) \mid_E) \mid_E & \end{array}$$

since the solution space increases as the constraints are restricted to E.

Conversely, assume $\mathcal{S} = \Theta(\mathcal{S})$ and suppose $\rho \models \Psi(\mathcal{S}) \mid_E$. Wlog, assume $\rho(\alpha) = \bot_s$ for all $\alpha \notin E$. We define a super-environment ρ' of ρ via $\Psi(\mathcal{S})$ as:

$$\rho'(\alpha) = \bigsqcup\{\mathcal{I}(\rho(\tau), p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha)\}$$

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

- Suppose $\mathcal{C} = [c \leq \alpha]$. Then

$$\begin{array}{rcl} \rho'(\alpha) & = & \bigsqcup\{\mathcal{I}(\rho(\tau), p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha)\} \\ & \sqsupseteq & \bigsqcup\{\mathcal{I}(\rho(c), \epsilon)\} \\ & = & \rho(c) \end{array}$$

- Suppose $\mathcal{C} = [\alpha \leq \beta]$:

$$\begin{array}{rcrcl} & & \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha) & \Rightarrow & \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\beta) \\ & & & & \text{by completeness of } \Theta \cup \Psi \\ & & & & \text{and the Staging Lemma 4.10} \\ \Rightarrow & & \{\mathcal{I}(\rho(\tau), p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha)\} & \subseteq & \{\mathcal{I}(\rho(\tau), p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\beta)\} \\ \Rightarrow & & \bigsqcup\{\mathcal{I}(\rho(\tau), p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha)\} & \sqsubseteq & \bigsqcup\{\mathcal{I}(\rho(\tau), p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\beta)\} \\ \Rightarrow & & \rho'(\alpha) & \subseteq & \rho'(\beta) \end{array}$$

- Suppose $\mathcal{C} = [\alpha \leq \mathtt{rng}(\beta)]$. Then:

$$\begin{array}{rcll} \rho'(\mathtt{rng}(\beta)) & = & \mathtt{rng}(\bigsqcup\{\mathcal{I}(\rho(\tau),p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\beta)\}) & \\ & = & \bigsqcup\{\mathcal{I}(\rho(\tau),p') \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p'} p'(\mathtt{rng}(\beta))\} & \\ & \sqsupseteq & \bigsqcup\{\rho(\alpha)\} & \\ & \sqsupseteq & \rho'(\alpha) & \text{via } (\textit{reflex}) \end{array}$$

- Suppose $\mathcal{C} = [\mathtt{rng}(\alpha) \leq \beta]$: Then

$$\begin{array}{rrcll} & \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\mathtt{rng}(\alpha)) & \Rightarrow & \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\beta) & \\ & & & \text{by completeness of } \Theta \cup \Psi & \\ & & & \text{and the Staging Lemma 4.10} & \\ \Rightarrow & \rho'(\mathtt{rng}(\alpha)) & = & \mathtt{rng}(\bigsqcup\{\mathcal{I}(\rho(\tau),p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha)\}) & \\ & & = & \bigsqcup\{\mathcal{I}(\rho(\tau),p') \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p'} p'(\mathtt{rng}(\alpha))\} & \\ & & \sqsubseteq & \bigsqcup\{\mathcal{I}(\rho(\tau),p') \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p'} p'(\beta)\} & \\ & & \sqsubseteq & \rho'(\beta) & \end{array}$$

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

$$\begin{array}{rcll} \rho'(\mathtt{dom}(\beta)) & = & \mathtt{dom}(\sqcap\{\mathcal{I}(\rho(\tau),p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\beta)\}) & \\ & = & \bigsqcup\{\mathcal{I}(\rho(\tau),p') \mid \mathcal{S} \vdash_\Psi \tau \geq_{\pi p'} p'(\mathtt{dom}(\beta))\} & \\ & \sqsupseteq & \bigsqcup\{\rho(\alpha)\} & \\ & \sqsupseteq & \rho'(\alpha) & \end{array}$$

- Suppose $\mathcal{C} = [\mathtt{dom}(\alpha) \leq \beta]$: Then:

$$\begin{array}{rrcll} & \mathcal{S} \vdash_\Psi \tau \geq_{\pi p} p(\mathtt{dom}(\alpha)) & \Rightarrow & \mathcal{S} \vdash_\Psi \tau \geq_{\pi p} p(\beta) & \\ & & & \text{by completeness of } \Theta \cup \Psi & \\ & & & \text{and the Staging Lemma 4.10} & \\ \Rightarrow & \rho'(\mathtt{dom}(\alpha)) & = & \mathtt{dom}(\bigsqcup\{\mathcal{I}(\rho(\tau),p) \mid \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha)\}) & \\ & & = & \sqcap\{\mathcal{I}(\rho(\tau),p') \mid \mathcal{S} \vdash_\Psi \tau \geq_{\pi p'} p'(\mathtt{dom}(\alpha))\} & \\ & & \sqsubseteq & \sqcap\{\mathcal{I}(\rho(\tau),p') \mid \mathcal{S} \vdash_\Psi \tau \geq_{\pi p'} p'(\beta)\} & \\ & & \sqsubseteq & \rho'(\beta) & \end{array}$$

Thus $\rho \models \mathcal{S}$. It remains to show that ρ and ρ' agree on E. Let $\alpha \in E$. Since Ψ includes $(\textit{reflex})$, $\rho(\alpha) \sqsubseteq \rho'(\alpha)$. Also:

$$\begin{array}{rrcll} & \rho'(\alpha) & \sqsubseteq & \rho(\alpha) & \\ \Leftarrow & \mathcal{I}(\rho(\tau),p) & \sqsubseteq & \rho(\alpha) & \forall \mathcal{S} \vdash_\Psi \tau \leq_{\pi p} p(\alpha) \\ \Leftarrow & \rho(\tau) & \sqsubseteq_{\pi p} & \rho(p(\alpha)) & \text{by Lemma A.18} \\ \Leftarrow & \rho & \models & \tau \leq_{\pi p} p(\alpha) & \\ \Leftarrow & \rho & \models & \Psi(\mathcal{S})\,|_E & \end{array}$$

Thus ρ and ρ' agree on E, and the lemma holds. ∎

Lemma A.18 *If p is a path and f_p denotes the corresponding function on $\mathcal{D}$, then for all $x \in \mathcal{D}$, $f_p(\mathcal{I}(x,p)) = x$.*

Proof: By induction on the length of p, and by case analysis on the top constructor in p.

- $p = \epsilon$. Trivial
- $p = \mathtt{dom}(p')$.

$$\begin{array}{rcl} f_p(\mathcal{I}(x,p)) & = & dom\,(f_{p'}(\mathcal{I}(x,\mathtt{dom}(p')))) \\ & = & dom\,(f_{p'}(\mathcal{I}(\langle\emptyset,x,\bot_s\rangle,p'))) \\ & = & dom\,(\langle\emptyset,x,\bot_s\rangle) \\ & = & x \end{array}$$

- The case where $p = \mathtt{rng}(p')$ is similar.

∎

Lemma A.19 *If* $\mathcal{S} \vdash_{\Psi\Theta} \alpha \leq_i \tau$ *then* $\mathcal{S} \vdash_{\Pi\Theta} \alpha \leq_i \tau$.

Theorem 4.5 [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:

1. Suppose $\mathcal{S}_1 \models_E \mathcal{S}_2$. Then

$$\begin{array}{rll} & \mathcal{S}_1 \models_E \Pi\Theta(\mathcal{S}_2) & \text{by the soundness of } \Theta \cup \Pi \\ \Rightarrow & \mathcal{S}_1 \models_E \Pi\Theta(\mathcal{S}_2)\,|_E & \\ \Rightarrow & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2)\,|_E.\ \ \mathcal{S}_1 \models \mathcal{C} & \\ \Rightarrow & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2)\,|_E.\ \ \mathcal{S}_1 \vdash_{\Psi\Theta} \mathcal{C} & \text{by Lemma 4.2} \\ \Rightarrow & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2)\,|_E.\ \ \mathcal{C} \in \Psi\Theta(\mathcal{S}_1)\,|_E & \\ \Rightarrow & \Psi\Theta(\mathcal{S}_1)\,|_E \supseteq \Pi\Theta(\mathcal{S}_2)\,|_E & \\ \Rightarrow & \mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2 & \end{array}$$

   Conversely, suppose $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$. Then

$$\begin{array}{rl} & \Psi\Theta(\mathcal{S}_1)\,|_E \supseteq \Pi\Theta(\mathcal{S}_2)\,|_E \\ \Rightarrow & \mathit{Soln}(\Psi\Theta(\mathcal{S}_1)\,|_E) \subseteq \mathit{Soln}(\Pi\Theta(\mathcal{S}_2)\,|_E) \\ \Rightarrow & \mathit{Soln}(\Psi\Theta(\mathcal{S}_1)\,|_E)\,|_E \subseteq \mathit{Soln}(\Pi\Theta(\mathcal{S}_2)\,|_E)\,|_E \\ \Rightarrow & \mathit{Soln}(\mathcal{S}_1)\,|_E \subseteq \mathit{Soln}(\mathcal{S}_2)\,|_E \\ & \text{by Lemma 4.4, since } \mathit{Soln}(\mathcal{S}_i)\,|_E = \mathit{Soln}(\Pi\Theta(\mathcal{S}_i)\,|_E)\,|_E \\ \Rightarrow & \mathcal{S}_1 \models_E \mathcal{S}_2 \end{array}$$

2. Follows from part 1.

∎

Lemma 4.7 *Let $G = G_r(\mathcal{S}, E)$. Then:*

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

Proof: We prove the left-to-right inclusion by induction on the derivation of the constraint $\mathcal{C} \in \Psi(\mathcal{S})$, and by case analysis on the last step in that derivation.

- Suppose $\mathcal{C} \in \Psi(\mathcal{S})$ because $\mathcal{C} \in \mathcal{S}$. We proceed by case analysis on $\mathcal{C}$.
  - Suppose $\mathcal{C} = [\alpha \leq \beta]$. Then $\alpha \in E$, so $\beta_L \mapsto \alpha_L \mapsto \alpha$, and hence $\alpha \in \mathcal{L}_G(\beta_L)$. Similarly, $\beta \in \mathcal{L}_G(\alpha_U)$.

  The remaining cases for $\mathcal{C}$ follow by similar reasoning.
- Suppose $\mathcal{C} = [\alpha \leq \mathtt{rng}(\kappa)]$ is inferred via $(\mathit{compose}_1)$ from the antecedents $[\alpha \leq \mathtt{rng}(\beta)]$ and $[\beta \leq \kappa]$. Then $\alpha_U \mapsto \mathtt{rng}(\beta_U)$, and by induction $\beta_U \mapsto^* \kappa$. Hence $\alpha_U \mapsto \mathtt{rng}(\kappa)$, as required.

The remaining cases follow by similar reasoning.

We prove the right-to-left inclusion by induction on the derivation $\alpha_L \mapsto^* \kappa$ or $\alpha_U \mapsto^* \kappa$, and by case analysis on the last step in the derivation. The reasoning for each case is straightforward. ∎

Lemma 4.9 *Let $G = G_t(\mathcal{S}, E)$. Then $\Pi(\mathcal{S})\,|_E = \mathcal{L}_G(R)$.*

Proof: We prove the left-to-right inclusion by case analysis on $\mathcal{C} \in \Pi(\mathcal{S})\,|_E$.

- Suppose $\mathcal{C} = [\alpha \leq \kappa]$. Then by Lemma 4.7, $\alpha_U \mapsto_G^* \kappa$. Since $\mathit{Vars}(\mathcal{C}) \subseteq E$, $\alpha \in E$, and hence $\alpha_L \mapsto_G \alpha$. Thus $R \mapsto_G^* [\alpha \leq \kappa]$, and hence $[\alpha \leq \kappa] \in \mathcal{L}_G(R)$.
- The case where $\mathcal{C} = [\kappa \leq \alpha]$ follows by similar reasoning.
- Suppose $\mathcal{C} = [c \leq \kappa]$. If $\mathcal{C} \in \mathcal{S}$, then $\kappa = \alpha \in E$, and $R \mapsto_G [c \leq \alpha_U] \mapsto_G [c \leq \alpha]$, as required.

  If $\mathcal{C} \notin \mathcal{S}$, then an examination of the inference rules in Π shows that $\mathcal{C}$ can only be inferred via (trans), based on the antecedents $[c \leq \alpha]$ and $[\alpha \leq \kappa]$. By Lemma 4.7, $\alpha_L \mapsto_G^* c$ and $\alpha_U \mapsto_G^* \kappa$. Hence $R \mapsto [c \leq \kappa]$, and hence $[c \leq \kappa] \in \mathcal{L}_G(R)$, as required.
- Otherwise $\mathcal{C} = [\kappa_1 \leq \kappa_2]$, where $\kappa_1, \kappa_2 \notin \mathit{SetVar}$. An examination of the inference rules in Π shows that $\mathcal{C}$ can only be inferred via (trans), based on the antecedents $[\kappa_1 \leq \alpha]$ and $[\alpha \leq \kappa_2]$. By Lemma 4.7, $\alpha_L \mapsto_G^* \kappa_1$ 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 the right-to-left inclusion by case analysis on $\mathcal{C} \in \mathcal{L}_G(R)$.

- Suppose $\mathcal{C} = [\kappa_1 \leq \kappa_2]$. Then for some α, $\alpha_L \mapsto_G^* \kappa_1$ and $\alpha_U \mapsto_G^* \kappa_2$. By Lemma 4.7, $\{\kappa_1 \leq \alpha,\ \alpha \leq \kappa_2\} \subseteq \Psi(\mathcal{S})$ and $\mathit{Vars}(\kappa_i) \subseteq E$. By Lemma 4.10, $\{\kappa_1 \leq \alpha,\ \alpha \leq \kappa_2\} \subseteq \Pi(\mathcal{S})$. Hence $[\kappa_1 \leq \kappa_2] \in \Pi(\mathcal{S})\,|_E$, as required.

- Otherwise $\mathcal{C} = [c \leq \kappa]$. Then for some α, $[c \leq \alpha] \in \mathcal{S}$ and $\alpha_U \mapsto_G^* \kappa$. By Lemma 4.7, $\{\alpha \leq \kappa\} \subseteq \Psi(\mathcal{S})$ and $\mathit{Vars}(\kappa) \subseteq E$. By Lemma 4.10, $\{\alpha \leq \kappa\} \subseteq \Pi(\mathcal{S})$. Hence $[c \leq \kappa] \in \Pi(\mathcal{S})\,|_E$, as required.

∎

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

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

Proof: The equality $\Psi(\Theta(\mathcal{S})) = \mathit{compat}(\Pi(\Theta(\mathcal{S})))$ 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 inclusion $\Psi\Theta(\mathcal{S}) \subseteq \Psi(\Theta(\mathcal{S}))$ holds, we suppose $\mathcal{S} \vdash_{\Psi\Theta} \mathcal{C}$, and prove that $\Theta(\mathcal{S}) \vdash_\Psi \mathcal{C}$ by induction on the derivation $\mathcal{S} \vdash_{\Psi\Theta} \mathcal{C}$, and by case analysis on the last step in this derivation.

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

- Suppose $\mathcal{S} \vdash_{\Psi\Theta} \mathcal{C}$ via one of the rules (s_1), (s_2) or (s_3). These rules are subsumed by (*trans*), ($\mathit{compose}_1$) and ($\mathit{compose}_4$), and hence this case is subsumed by the previous case.

- Suppose $\mathcal{S} \vdash_{\Psi\Theta} \mathcal{C}$ 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*)

- The case for (s_5) holds by similar reasoning.

∎

The Entailment Algorithm

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

$$\begin{array}{lcllcl} G_1 & = & G_r(\mathcal{S}_1, E) & L_i & = & \{\alpha_L \mid \alpha \in \mathit{Vars}(\mathcal{S}_i)\} \\ G_2 & = & G_l(\mathcal{S}_2, E) & U_i & = & \{\alpha_U \mid \alpha \in \mathit{Vars}(\mathcal{S}_i)\} \end{array}$$

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

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

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

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

then one of the following cases hold:

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

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

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

Figure 11: The Computable Entailment Relation $\vdash^E_{\text{alg}}$

A.4 Correctness of the Entailment Algorithm

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

Proof: The definitions of the entailment relation and the relation $\mathcal{R}$ are shown in figure 11. We first prove that:

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

To prove the left-to-right direction, suppose $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ and:

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

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 algorithm must hold.

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

2. In this case:

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

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

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

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 11 hold. Furthermore, since $\mathcal{R}$ is the largest relation satisfying the conditions in figure 11, there exists a finite proof that none of the three conditions hold.

Of all possible such *counter-examples* $\langle \alpha_L, \beta_U, X, Y, C, D \rangle$, 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 $X = \mathtt{rng}(\alpha'_L)$ and $Y = \mathtt{rng}(\beta'_U)$. 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 $[\mathtt{rng}(\tau_L) \leq \mathtt{rng}(\tau_U)] \in \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D)$ 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:
$$\begin{array}{c} \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L) \mapsto^*_{G_2} \mathtt{rng}(\tau_L) \\ \delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U) \mapsto^*_{G_2} \mathtt{rng}(\tau_U) \end{array}$$
    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 $[\mathtt{rng}(\tau_L) \leq \mathtt{rng}(\tau_U)] \in \mathit{compat}(\mathcal{L}(C)) \setminus \mathcal{L}(C)$, and hence $[\tau_L \leq \tau_U] \in \mathit{compat}(\mathcal{L}(C))$.

Hence

$$\mathcal{L}([\alpha'_L \leq \beta'_U]) \subseteq \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D')$$

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 counter-example with a smaller proof. Hence $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha'_L, \beta'_U, C, D']$ holds, which implies that $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ holds via the condition two.

- 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 similar reasoning.

Thus the invariant

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

is true. Therefore:

$$\begin{array}{ll} & \mathcal{S}_2 \vdash^E_{\mathrm{alg}} \mathcal{S}_1 \\ \Longleftrightarrow & \forall \alpha \in \mathit{Vars}(\mathcal{S}_1).\ \mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \alpha_U, \{\langle \gamma_L, \gamma_U \rangle \mid \gamma \in \mathit{Vars}(\mathcal{S}_2)\}, \emptyset] \\ \Longleftrightarrow & \forall \alpha \in \mathit{Vars}(\mathcal{S}_1).\ \mathcal{L}([\alpha_L \leq \alpha_U]) \subseteq \mathit{compat}(\mathcal{L}(\{\langle \gamma_L, \gamma_U \rangle \mid \gamma \in \mathit{Vars}(\mathcal{S}_2)\})) \\ \Longleftrightarrow & \Pi(\mathcal{S}_1) \subseteq \mathit{compat}(\Pi(\mathcal{S}_2)) \\ \Longleftrightarrow & \mathcal{S}_2 \vdash^E_{\Psi} \mathcal{S}_1 \end{array}$$

Hence the entailment algorithm is correct.

∎

A.5 Correctness of the Hopcroft Algorithm

Theorem A.20 *Let $\mathcal{S}$ be a simple constraint system with external variables E, and let $\sim$ be an equivalence relation on the set variables in a constraint system $\mathcal{S}$ such that:*

1. *Each set variable in E is in an equivalence class by itself.*
2. *If $[\alpha \leq \beta] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \exists \beta \sim \beta'$ such that $[\alpha' \leq \beta'] \in \mathcal{S}$.*
3. *If $[\alpha \leq \mathtt{rng}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \exists \beta \sim \beta'$ such that $[\alpha' \leq \mathtt{rng}(\beta')] \in \mathcal{S}$.*
4. *If $[\mathtt{rng}(\alpha) \leq \beta] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \exists \beta \sim \beta'$ such that $[\mathtt{rng}(\alpha') \leq \beta'] \in \mathcal{S}$.*
5. *If $[\alpha \leq \mathtt{dom}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha'\ \forall \beta \sim \beta'$ such that $[\alpha' \leq \mathtt{dom}(\beta')] \in \mathcal{S}$.*

Let $\mathcal{S}'$ 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

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

Obviously ρ, ρ' agree on E from condition 1. We claim that $\rho' \models \mathcal{C}$ for all $\mathcal{C} \in \mathcal{S}$ by case analysis on $\mathcal{C}$.

- Suppose $\mathcal{C} = [\alpha \leq \mathtt{rng}(\beta)]$. Then $\forall \alpha \sim \alpha' \; \exists \beta \sim \beta'$ such that:

$$\rho(\alpha') \quad \sqsubseteq \quad \rho(\mathtt{rng}(\beta'))$$

Hence $\forall \alpha \sim \alpha'$:

$$\rho(\alpha') \quad \sqsubseteq \quad \bigsqcup_{\beta' \sim \beta} \rho(\mathtt{rng}(\beta'))$$

and therefore:

$$\bigsqcup_{\alpha' \sim \alpha} \rho(\alpha') \quad \sqsubseteq \quad \bigsqcup_{\beta' \sim \beta} \rho(\mathtt{rng}(\beta'))$$

Hence:

$$\begin{array}{rcl} \rho'(\alpha) & = & \displaystyle\bigsqcup_{\alpha' \sim \alpha} \rho(\alpha') \\ & \sqsubseteq & \displaystyle\bigsqcup_{\beta' \sim \beta} \rho(\mathtt{rng}(\beta')) \\ & = & \displaystyle rng \left(\bigsqcup_{\beta' \sim \beta} \rho(\beta') \right) \\ & = & rng\,(\rho'(\beta)) \\ & = & \rho'(\mathtt{rng}(\beta)) \end{array}$$

and thus $\rho' \models \mathcal{C}$, as required.

- The cases where $\mathcal{C} = [\alpha \leq \beta]$ and $\mathcal{C} = [\mathtt{rng}(\alpha) \leq \beta]$ follow by similar reasoning.
- Suppose $\mathcal{C} = [\alpha \leq \mathtt{dom}(\beta)]$. Then $\forall \alpha \sim \alpha' \; \forall \beta \sim \beta'$ such that:

$$\rho(\alpha') \quad \sqsubseteq \quad \rho(\mathtt{dom}(\beta'))$$

Hence $\forall \alpha \sim \alpha'$:

$$\rho(\alpha') \quad \sqsubseteq \quad \bigsqcap_{\beta' \sim \beta} \rho(\mathtt{dom}(\beta'))$$

and therefore:

$$\bigsqcup_{\alpha' \sim \alpha} \rho(\alpha') \quad \sqsubseteq \quad \bigsqcap_{\beta' \sim \beta} \rho(\mathtt{dom}(\beta'))$$

Hence:

$$\begin{array}{rcl} \rho'(\alpha) & = & \displaystyle\bigsqcup_{\alpha' \sim \alpha} \rho(\alpha') \\ & \sqsubseteq & \displaystyle\bigsqcap_{\beta' \sim \beta} \rho(\mathtt{dom}(\beta')) \\ & = & \displaystyle dom \left(\bigsqcup_{\beta' \sim \beta} \rho(\beta') \right) \\ & = & dom\,(\rho'(\beta)) \\ & = & \rho'(\mathtt{dom}(\beta)) \end{array}$$

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

$$\begin{array}{rcl}
\rho'(\mathtt{dom}(\beta)) & = & dom\,(\rho'(\beta)) \\
& = & dom\left(\bigsqcup_{\beta' \sim \beta} \rho(\beta')\right) \\
& = & \bigsqcap_{\beta' \sim \beta} \rho(dom\,(\beta')) \\
& \sqsubseteq & \rho(dom\,(\beta)) \\
& \sqsubseteq & \rho(\alpha) \\
& \sqsubseteq & \bigsqcup_{\alpha' \sim \alpha} \rho(\alpha') \\
& = & \rho'(\alpha)
\end{array}$$

- Suppose $\mathcal{C} = [c \leq \alpha]$. Then

$$\begin{array}{rcl}
\rho'(\alpha) & = & \bigsqcup_{\alpha' \sim \alpha} \rho(\alpha') \\
& \sqsupseteq & \rho(\alpha) \\
& = & c
\end{array}$$

∎

B Notations

Symbol	Meaning	Section	Page
$M \in \Lambda$	Terms	2	2
$V \in \mathit{Value}$	Values	2	2
$x \in \mathit{Vars}$	Variables	2	2
$b \in \mathit{BConst}$	Basic constants	2	2
$t \in \mathit{Tag}$	Function tags	2	2
$l \in \mathit{Label}$	Labels	2	2
$\beta_v, \beta_{let}, \mathit{unlabel}$	Reduction rules	2	3
$\mathcal{E}$	Evaluation contexts	2	3
$\longmapsto, \longmapsto^*$	Standard reduction relation	2	3
eval	Evaluator	2	3
$\tau \in \mathit{SetExp}$	Set expressions	3.1	4
$\alpha, \beta, \ldots \in \mathit{SetVar}$	Set variables	3.1	4
$c \in \mathit{Const}$	Constants	3.1	4
$\mathtt{dom}, \mathtt{rng}$	Type expression constructors	3.1	4
$\mathcal{C}$	Constraints	3.1	4
$\mathcal{S}$	Constraint systems	3.1	4

$\mathcal{S}\mid_E$	Restriction of a constraint system	3.1	4
$\mathcal{D}, \sqsubseteq, \top, \bot, \sqcup, \sqcap$	Domain, ordering, elements and operations	3.2	4
$\mathcal{P}$	Power set constructor	3.1	4
in	Values described by constants	3.2	4
const, *dom*, *rng*	Extract components of element of $\mathcal{D}$	3.2	4
$\rho \in \mathit{SetEnv}$	Set environment	3.3	5
$\mathit{Soln}(\mathcal{S})$	Solution space	3.3	5
$\models, \models_E$	Satisfies, or entails	3.3	5
$\mathit{Soln}(\mathcal{S})\mid_E$	Restriction	3.3	5
$\cong_E$	Observable equivalence	3.1	5
$\vdash$	Constraint derivation rules	3.4	6
Γ	Set variable context	3.4	6
σ	Constraint schema	3.4	6
$\sqsubseteq_s, \top_s, \bot_s, \sqcup_s, \sqcap_s$	Alternative ordering on domain	3.4	6
sba	Analysis function	3.6	8
LeastSoln	Least Solution	3.6	8
$\Theta = \{s_1, \ldots, s_n\}$	Inference rules	3.6	8
$\vdash$	Deduction	3.7	9
E	External variables	4.1	10
$FV[\mathit{rng}(\Gamma)]$	Free variables in Γ	3.4	7
Ψ, Π	Inference rules on constraint systems	4.2	11
G	Grammar	4.6	14
G_r	Function producing regular grammar	4.6	14
α_L, α_U	Grammar non-terminals	4.6	14
$\mathcal{L}_G(X)$	Language for X in G	4.6	14
G_t	Function producing RTG	4.8	14
R	Root non-terminal	4.8	14
p, q	Paths	A.3	33
π	Arity function	A.3	33
$\leq_i$	$\leq$ or $\geq$	A.3	33
`sel`	Either `dom` or `rng`	A.3	33
$\overline{\mathcal{C}}$	Converts compound to simple constraints	A.3	33
$\mathcal{I}$	Injection function	A.3	33
$\leq^*$	Transitive closure of $\leq$	A.2	31

References

[1] Aho, A., J. Hopcroft and J. Ullman. *The Design and Analysis of Computer*

Algorithms. Addison-Wesley, Reading, Mass., 1974.

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

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

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

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

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

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

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

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

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

[11] Hindley, R. J., and Seldin, J. P. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

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

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

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

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

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

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

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

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

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

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

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