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

Modular and Polymorphic Set-Based Analysis: Theory and Practice

(revision 1.4)

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 Source Language 4

3.1.1 Syntax 4

3.1.2 Semantics 4

3.2 The Constraint Language 5

3.3 The Meaning of Set Constraints 6

3.3.1 The Semantic Domain 6

3.3.2 The Semantics of Constraints 7

3.4 Deriving Constraints 8

3.5 Soundness of the Derived Constraints 10

3.6 Solving Set Constraints 11

3.6.1 Computing the Least Solution 13

4 Constraint Simplification **13**

4.1 Conditions for Constraint Simplification 14

4.2 The Proof Theory of Observable Equivalence 15

4.3 Deciding Observable Equivalence 18

4.4 Regular Grammars 18

4.5 Regular Tree Grammars 20

4.6 Staging 21

4.7 The Entailment Algorithm 21

4.8 Practical Constraint System Simplification 23

4.8.1 Empty Constraint Simplification 24

4.8.2 Unreachable Constraint Simplification 24

4.8.3 Removing ϵ-Constraints 25

4.8.4 Hopcroft's Algorithm 25

4.9 Simplification Benchmarks 26

5 Componential Set-Based Analysis **27**

5.1 Componential Set-Based Analysis 27

5.2 Experimental Results 28

5.3 User Interface for Multi-File Programs 30

5.4 Efficient Polymorphic Analysis 30

6 Competitive Work **31**

7 Future Work **32**

A Proofs **32**
A.1 Subject Reduction Proof 32
A.2 Proofs for Computing Set-Based Analysis 39
A.3 Proofs for Conditions for Constraint Simplification 42
A.4 Proofs for Proof Theory of Observable Equivalence 43
A.5 Proofs for Deciding Observable Equivalence 49
A.6 Correctness of the Entailment Algorithm 52
A.7 Correctness of the Hopcroft Algorithm 55

B Notations **56**

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	$\in$	*BasicConst*			(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 a related report [**?**].

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 $(\mathcal{E}\ M)$. An evaluation context always contains a single hole $[\,]$, and we use the notation $\mathcal{E}[\ M\ ]$ to denote term produced by filling the hole in $\mathcal{E}$ with the term M.

The standard 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 semantics of the language is defined via the partial function *eval* on programs:

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

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

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

relationships of the analyzed program. During the solution phase, the analysis produces finite descriptions of the potentially infinite sets of values that satisfy these constraints. The result provides an approximate set of values for each labeled expression in the program.

3.1 The Source Language

We develop the analysis for an idealized, λ-calculus-like language Λ with constants and labeled expressions. This section introduces the syntax and semantics of Λ.

3.1.1 Syntax

Syntax:

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

Figure 2: The source language Λ

Expressions in the language are either variables, values, applications, **let**-expressions, or labeled expressions: see figure 2. Values include basic constants and functions. Each function has an identifying tag so that MrSpidey can reconstruct the textual source of function values from the results of the analysis. We use **let**-expressions to introduce polymorphic bindings, and hence restrict these bindings to syntactic values [22]. We use labels to identify those program expressions whose values we wish to predict.

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*.

3.1.2 Semantics

We specify the meaning of programs based upon three *notions of reduction*:

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

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 sub-term to be evaluated:

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

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

The *standard reduction relation* $\longmapsto$ is the compatible closure of $\longrightarrow$ with respect to evaluation contexts:

$$\mathcal{E}[\ M\ ] \longmapsto \mathcal{E}[\ N\ ] \qquad \text{iff} \qquad M \longrightarrow N$$

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

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

3.2 The Constraint Language

To simplify the later derivation of the constraint simplification algorithms (see chapter **??**), we express the constraint language in terms of selectors, instead of the more usual constructors. Specifically, a *set expression* τ is either a set variable; a constant; or one of the "selector" expressions $\mathtt{dom}(\tau)$ or $\mathtt{rng}(\tau)$:

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

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

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* $\mathsf{C} \in \mathit{Constraint}$ is an inequality $\tau_1 \le \tau_2$ relating two set expressions.

$$\mathsf{C} \ \in \ \mathit{Constraint} \ = \ \tau_1 \le \tau_2$$

We sometimes enclose constraints inside square brackets for clarity: $[\tau_1 \le \tau_2]$. A *constraint system* $\mathsf{S} \in \mathit{ConstraintSystem}$ is a collection of constraints.

$$\mathsf{S} \ \in \ \mathit{ConstraintSystem} \ = \ \mathcal{P}_{\text{fin}}(\mathit{Constraint})$$

We use $\mathit{SetVar}(\mathsf{S})$ to denote the collection of set variables in a constraint system S. In some cases, the relevant constraints in a constraint system are those that only mention certain set variables. The *restriction* of a constraint system to a collection of set variables E is:

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

3.3 The Meaning of Set Constraints

Intuitively, each set expression τ denotes a set of run-time values, and each constraint $\tau_1 \leq \tau_2$ denotes a corresponding set containment relationship. We formalize the meaning of set constraints by mapping syntactic set expressions onto a semantic domain. The next subsection describes the precise structure of the semantic domain, and the second subsection describes the mapping from set expressions to that domain.

3.3.1 The Semantic Domain

A set expression denotes a collection of values. For our sample language, the collection consists of basic constants and functions and is therefore best represented as a triple $X = \langle C, D, R\rangle$. The first component $C \in \mathcal{P}(Const)$[4] is a set of basic constants and function tags. The second and third components of X denote the possible arguments and results of functions in X, respectively. Since these two components also denote value sets, the appropriate model for set expressions is the solution of the equation:

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

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

Each element of $\mathcal{D}$ represents a set of run-time values (relative to a given program) according to the set of basic constants and function tags in its first component. The set of values represented by an element $X \in \mathcal{D}$ is defined through the relation V *in* X:

$$\begin{aligned} b \ \textit{in}\ \langle C, D, R\rangle \quad &\text{iff} \quad b \in C \\ (\lambda^t x.M) \ \textit{in}\ \langle C, D, R\rangle \quad &\text{iff} \quad t \in C \end{aligned}$$

We order the elements of $\mathcal{D}$ according to a relation that is anti-monotonic in the domain position:

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

This ordering is anti-monotonic in the domain position because information about argument values at an application needs to flow *backward* along data-flow paths to the formal parameter of the corresponding function definitions. To illustrate this idea, consider a program that binds a function f to a program variable g. This behavior is described in the semantic domain as the inequality $X_f \sqsubseteq X_g$, where X_f and X_g describe the values sets for f and g respectively. Since the possible argument set for f must contain all values to which g is

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

[5] We can analyze languages with additional data structures by extending $\mathcal{D}$ to infinite *n-ary* trees, where n is the number of selectors (*e.g.*. `dom`, `rng`) corresponding to the extended language.

applied, the inequality $dom\,(X_g) \sqsubseteq dom\,(X_f)$ must also hold. Thus the domain $\mathcal{D}$ should satisfy the inference rule:

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

which is why the ordering $\sqsubseteq$ needs to be anti-monotonic in the domain element.

Under the defined ordering, the set $\mathcal{D}$ forms a complete lattice; the top and bottom elements are the solutions to the equations:

$$\begin{array}{rcl} \top & = & \langle Const, \bot, \top \rangle \\ \bot & = & \langle \emptyset, \top, \bot \rangle \end{array}$$

respectively. The least upper bound and greatest lower bound operations are recursively defined as:

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

3.3.2 The Semantics of Constraints

The semantics of a constraint system is defined with respect to a *set environment* ρ, which specifies an element of $\mathcal{D}$ for each set variable in a constraint system:

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

The collection of set environments forms a complete lattice $SetEnv$ under the pointwise extension of the ordering relation $\sqsubseteq$ on $\mathcal{D}$.

For each set environment ρ, we define the following unique extension ρ^* that specifies a meaning for set expressions:

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

Where there is no confusion, we remove the asterisk and simply use ρ to denote ρ^*.

A set environment ρ *satisfies* a constraint $\mathsf{C} = [\tau_1 \leq \tau_2]$ (written $\rho \models \mathsf{C}$) if $\rho(\tau_1) \sqsubseteq \rho(\tau_2)$. Similarly, ρ satisfies S, or ρ is a solution of S (written $\rho \models \mathsf{S}$) if $\rho \models \mathsf{C}$ for each $\mathsf{C} \in \mathsf{S}$. The relation $\models$ is obviously reflexive and transitive. The *solution space* of a constraint system S is:

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

A constraints set S_1 *entails* S_2 (written $\mathsf{S}_1 \models \mathsf{S}_2$) iff $Soln(\mathsf{S}_1) \subseteq Soln(\mathsf{S}_2)$, and S_1 is *observably equivalent to* S_2 (written $\mathsf{S}_1 \cong \mathsf{S}_2$) iff $\mathsf{S}_1 \models \mathsf{S}_2$ and $\mathsf{S}_2 \models \mathsf{S}_1$.

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

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

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

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

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

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

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

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

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

Figure 3: Constraint derivation rules.

There are actually *more* set environments in the restricted solution space, since these additional environments can specify arbitrary domain elements for all set variables that are not in E.

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

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

- If $Soln(\mathsf{S}_1) \mid_E \subseteq Soln(\mathsf{S}_2) \mid_E$, then S_1 *entails* S_2 *with respect to* E (written $\mathsf{S}_1 \models_E \mathsf{S}_2$).
- If $\mathsf{S}_1 \models_E \mathsf{S}_2$ and $\mathsf{S}_2 \models_E \mathsf{S}_1$ then that S_1 and S_2 are *observably equivalent with respect to* E (written $\mathsf{S}_1 \cong_E \mathsf{S}_2$) .

∎

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.* [2] and Palsberg and O'Keefe [16], we formulate this derivation as a proof system.

The derivation proceeds in a syntax-directed manner according to the constraint derivation rules presented in figure 3. Each rule infers a judgment of the form $\Gamma \vdash M : \alpha, \mathcal{S}$, where:

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

The constraint derivation rules only generates a certain subset of the constraint language, called *simple constraints*. Simple constraints have the form:

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

$$\mathcal{S} \in \mathit{SimpleConSystem} = \mathcal{P}_{\text{fin}}(\mathit{SimpleCon})$$

A collection of such simple constraints forms a *simple constraint system*. We use the calligraphic letters $\mathcal{C}$ and $\mathcal{S}$ as meta-variables ranging over simple constraints and simple constraint systems, respectively.

The constraint derivation rule (*var*) derives appropriate constraints for a variable reference x. This rule generates the constraint $\beta \leq \alpha$, where β describes the value set of x, and α denoting the value set for this reference to x. The constraint derivation rule (*const*) generates the constraint $b \leq \alpha$, which ensures that the value set for a constant expression contains that constant. The rule (*label*) records the possible values of a labeled expression M^l in the label l.

The rule (*abs*) for functions records the function's tag, and 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 correctness of the rules (*abs*) and (*app*) relies on the anti-monotonicity of the underlying ordering $\sqsubseteq$ in the domain position.

The rule (*let*) produces a *constraint schema* $\sigma = \forall\overline{\alpha}.(\beta, \mathcal{S})$ for polymorphic, **let**-bound values [2, 19]. The set variable β names the result of the value; the simple constraint system $\mathcal{S}$ describes the data-flow relationships of the value, using β; and the set $\overline{\alpha} = \{\alpha_1, \ldots, \alpha_n\}$ contains those internal set variables of the constraint system that must be duplicated at each reference to the **let**-bound variable via the rule (*inst*).

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

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

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

Many of the constraint derivation rules contain *meta* set variables. For example, the rule (*const*):

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

mentions the meta set variable α. Any time this rule is applied, we need to choose an appropriate set variable for this meta variable. Choosing a fresh set variable not used elsewhere in the derivation yields a more accurate analysis. A *most general constraint derivation* is one that always uses fresh set variable for these meta variables, and a *most general constraint system* for an expression is one produced by a most general constraint derivation. However, the use of fresh variables is not strictly necessary for the correctness of the analysis. As an extreme example, we could perform the entire analysis using a single set variable, although this would yield extremely coarse results, and would be of no practical use. But the ability to consider constraint derivations that re-use certain set variables significantly simplifies the subject reduction proofs of the following section.

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 of $\mathcal{S}$ correctly approximates the value sets of labeled expressions in P. That is, if ρ is a solution of $\mathcal{S}$ and V is a possible value of some expression M^l in P, then V *in* $\rho(l)$. We prove this property using a subject reduction proof [7], following Wright and Felleisen [21] and Palsberg [15].

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

Proof: The Subject Reduction for $\longmapsto$ Lemma (3.3) shows that standard reduction steps preserve entailment. Hence, since $P \longmapsto^* \mathcal{E}[\, V^l \,]$, there exists some $\mathcal{S}'$ such that $\emptyset \vdash \mathcal{E}[\, V^l \,] : \alpha, \mathcal{S}'$ and $\mathcal{S} \models \mathcal{S}'$. The derivation of this judgment must contain a sub-derivation concluding:

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

Except for the rule (let), each application of a constraint derivation rule can only extend the constraint system produced by its sub-derivation. Since definition of evaluation contexts does not contain a clause for **let**-expressions, there cannot be any **let**-expressions on the spine from V^l to $\mathcal{E}[\, V^l \,]$. Hence $\mathcal{S}_V \cup \{\beta \leq l\} \subseteq \mathcal{S}'$.

Since $\rho \models \mathcal{S}$, $\mathcal{S} \models \mathcal{S}'$, and $\mathcal{S}' \supseteq \mathcal{S}_V \cup \{\beta \leq l\}$, we have that $\rho \models \mathcal{S}_V \cup \{\beta \leq l\}$. Hence V *in* $\rho(\beta)$ by the Value Typing Lemma 3.7. But $\rho(\beta) \sqsubseteq \rho(l)$, hence V *in* $\rho(l)$, as required. ∎

The proof of the above result relies on the following lemma showing that standard reduction steps preserves the entailment of the derived constraint systems.

Lemma 3.3 (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 3.4 and the Replacement Lemma 3.5. ∎

Lemma 3.4 (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: See Appendix A.1. ∎

Lemma 3.5 (Replacement) *If:*

1. *D is a deduction concluding $\Gamma \vdash \mathcal{E}[\ M_1\ ] : \alpha, \mathcal{S}_1$,*
2. *D_1 is a sub-deduction 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 the proof idea of Hindley and Seldin [11:page 181]. ∎

The Flow Lemma describes conditions under which we can replace the result set variable returned by the constraint derivation rules.

Lemma 3.6 (Flow) *If $\Gamma \vdash M : \alpha, \mathcal{S}$ then for all $\gamma \in SetVar$, $\Gamma \vdash M : \gamma, \mathcal{S}'$ with $\mathcal{S} \cup \{\alpha \leq \gamma\} \models \mathcal{S}'$.*

Proof: See Appendix A.1. ∎

The Value Typing Lemma simply states that any solution to the constraint system for a syntactic value always corresponds to a value set invariant that includes that value.

Lemma 3.7 (Value Typing) *If $\Gamma \vdash V : \alpha, \mathcal{S}$ and $\rho \models \mathcal{S}$, then V in $\rho(\alpha)$.*

Proof: By considering the two cases $V = b$ and $V = (\lambda^t x.M)$. ∎

3.6 Solving Set Constraints

Every simple constraint system admits the trivial solution $\rho^{\top_s}$ defined by $\rho^{\top_s}(\alpha) = \top_s$ where $\top_s$ is recursively defined as $\top_s = \langle Const, \top_s, \top_s \rangle$. The domain element $\top_s$ represents the set of all run-time values, including functions that can take any value as an argument, and return any value as a result.

Lemma 3.8 *If $\mathcal{S}$ is a simple constraint system then $\rho^{\top_s} \models \mathcal{S}$.*

Proof: By a case analysis showing that $\rho^{\top_s} \models \mathcal{C}$ for any simple constraint $\mathcal{C}$. ∎

Since $\top_s$ represents all run-time values, this solution is highly approximate and thus utterly useless. Fortunately, simple constraint systems yield many additional solutions that more accurately characterize the value sets of program expressions.

To illustrate this idea, consider the program $P = (\lambda^t x.x)$. According to the constraint derivation rules of figure 3, this program yields the constraint system:

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

In addition to the trivial solution $\rho^{\top_s}$, this constraint system admits a number of other solutions, including:

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

where $X = \langle\{c_2\}, \bot, \bot\rangle$, and c_1 and c_2 are arbitrary constants. Because we assume P to be the entire program, the function tagged t is never applied, and hence the set of run-time values for x is simply the empty set. The solution ρ_1 describes this (empty) set of run-time values of x more accurately than either ρ_2 or ρ_3. Yet these three solutions are incomparable under the ordering $\sqsubseteq$, since the ordering models the flow of values through a program, but does not rank set environments according to their accuracy.

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

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

The maximal and minimal elements of $\mathcal{D}$ under $\sqsubseteq_s$ are the solutions to the equations:

$$\begin{array}{rcl} \top_s & = & \langle \mathit{Const}, \top_s, \top_s\rangle \\ \bot_s & = & \langle \emptyset, \bot_s, \bot_s\rangle \end{array}$$

respectively. The least upper bound and greatest lower bound operations are recursively defined as:

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

Under the ordering $\sqsubseteq_s$, a simple constraint system has both a maximal solution ($\rho^{\top_s}$ 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 3.9 (Least Solution of Simple Constraint Systems) *Every simple constraint system has a solution that is least with respect to $\sqsubseteq_s$.*

Proof: See Appendix A.2. ∎

Using Lemma 3.9, it makes sense to define $\mathit{LeastSoln}(\mathcal{S})$ as the least solution of the simple constraint system $\mathcal{S}$ under the ordering $\sqsubseteq_s$. Since this solution yields the most accurate invariants consistent with the constraints $\mathcal{S}$, we define set-based analysis as the function that extracts the possible values for each labeled expression from this least solution.

Definition 3.10. $(\mathit{sba} : \Lambda^0 \longrightarrow (\mathit{Label} \longrightarrow \mathcal{P}(\mathit{Value})))$ If $\emptyset \vdash P : \alpha, \mathcal{S}$ is a most general derivation, then

$$\mathit{sba}(P)(l) = \{V \mid V \textit{ in } \mathit{LeastSoln}(\mathcal{S})(l)\}$$

∎

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

Theorem 3.11 *If $P \longmapsto^* \mathcal{E}[\, V^l \,]$ then $V \in \mathit{sba}(P)(l)$.*

Proof: Follows from Lemma 3.2. ∎

3.6.1 Computing the Least Solution

To compute $sba(P)$, we derive the most general constraint system for P and close that constraint system under the rules Θ described in figure 4. Intuitively, these rules infer all the data-flow paths in the program, which are described by constraints of the form $\beta \leq \gamma$ (for $\beta, \gamma \in SetVar$), and propagate values along those data-flow paths. Specifically, the rules (s_1), (s_2), and (s_3) propagate information about constants, function domains and function ranges forward along the data-flow paths of the program. The rule (s_4) constructs the data-flow paths from 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 $\{\mathcal{C} \mid \mathcal{S} \vdash_\Theta \mathcal{C}\}$. An algorithm for computing $\Theta(\mathcal{S})$ is included in the next section.

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

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

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

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

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

Figure 4: The rules $\Theta = \{s_1, \ldots, s_5\}$

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 3.12 *If $P \in \Lambda^0$ and $\emptyset \vdash P : \alpha, \mathcal{S}$ is a most general constraint derivation then:*

$$\begin{aligned} sba(P)(l) = \quad & \{b \mid \mathcal{S} \vdash_\Theta b \leq l\} \\ \cup \quad & \{(\lambda^t x.M) \mid \mathcal{S} \vdash_\Theta t \leq l\} \end{aligned}$$

Proof: See Appendix A.2. ∎

4 Constraint Simplification

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 and manipulating these constraint systems can be extremely expensive.

To overcome this problem, we have developed algorithms for *simplifying* constraint systems. A simplified version of a constraint system contains significantly fewer constraints, yet still preserves the essential characteristics of the original system. Applying these simplification algorithms at strategic points during the constraint derivation, *e.g.*, to the constraint system for a module definition or a polymorphic function definition, significantly reduces both the time and space required by the overall analysis.

The following section shows that constraint simplification does not affect the analysis results, provided the simplified system is *observably equivalent* to the original system. Section 4.2 presents a proof-theoretic formulation of observable equivalence, and section 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 4.8.

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 $\mathcal{S}_2$, without changing the results of the analysis.

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

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

where $\mathcal{S}_C$ is the constraint system for C. The union of the sets $\mathcal{S}_C$ and $\mathcal{S}_1$ describes the space of solutions for the entire program, which is the same as the intersection of the two respective solution spaces:

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

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

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

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

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

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

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

or, with the notation from section 3.2, $\mathcal{S}_1$ 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:

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

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 (Admissibility of $(\cong)$) *If $\emptyset \vdash_{\cong} P : \alpha, \mathcal{S}$ is a most general constraint derivation then:*

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

Proof: See Appendix A.3. ∎

4.2 The Proof Theory of Observable Equivalence

Since the new derivation rule $(\cong)$ involves the *semantic* notion of observably equivalent constraint systems, it cannot be used in an algorithmic manner. To make this rule useful, we must first reformulate the observable equivalence relation (or some conservative approximation thereof) 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 of the functions *rng* and *dom*, respectively. We can reify these properties into a syntactic proof system via the following inference rules Δ:

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

The meta-variables $\kappa, \kappa_1, \kappa_2$ range over non-constant set expressions:

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

This restriction avoids inferring useless tautologies. For example, without this restriction, the constraint $c \leq \alpha$ would yield the constraint $\mathtt{rng}(c) \leq \mathtt{rng}(\alpha)$ via (*compat*), which is a tautology since $\mathtt{rng}(c) = \bot$.

The rules (*reflex*) and ($trans_\tau$) capture the reflexivity and transitivity of the ordering relation $\sqsubseteq$; (*compat*) reflects the monotonicity and anti-monotonicity of the functions *rng* and *dom*, respectively. Since many of the inferred constraints lie outside of the original language of simple constraints, we define an extended *compound constraint* language that includes all of the inferred constraints:

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

We use the boldface roman letters $\mathbf{C}$ and $\mathbf{S}$ as meta-variables ranging over compound constraints and compound constraint systems, respectively.

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

Lemma 4.2 (Soundness and Completeness of Δ) *For a compound constraint system* $\mathbf{S}$ *and a compound constraint* $\mathbf{C}$*:*

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

Proof: See Appendix A.4. ∎

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

Lemma 4.3 *For a compound constraint system* $\mathbf{S}$, $\mathbf{S} \cong_E \Delta(\mathbf{S}) \mid_E$.

Proof: See Appendix A.4. ∎

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

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

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

The alternative system consists of the inference rules Ψ described in figure 5, together with the rules Θ from figure 4. 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 ($trans_\alpha$) of Ψ provides a weaker characterization of transitivity than the previous rule ($trans_\tau$), but, provided we start from with a simple constraint system, the additional rules, Θ and ($compose_{1\ldots4}$), compensate for this weakness. That is, suitable combinations of these additional rules allow us to infer any constraint that could be inferred by the rule ($trans_\tau$).

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

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

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

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

Figure 5: The Inference Rule System Ψ

Lemma 4.4 (Equivalence of Proof Systems) *For a simple constraint system $\mathcal{S}$:*

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

Proof: See Appendix A.4. ∎

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, this definition is needlessly inefficient. Because (*compat*) does not eliminate any variables, any (*compat*)-consequent in $\Psi\Theta(\mathcal{S}_2) \mid_E$ is subsumed by its antecedent. If we define:

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

then this argument implies that $\Psi\Theta(\mathcal{S}_2) \mid_E \cong \Pi\Theta(\mathcal{S}_2) \mid_E$. Hence we get the following lemma.

Lemma 4.5 $\Psi\Theta(\mathcal{S}) \mid_E \cong \Pi\Theta(\mathcal{S}) \mid_E$.

Proof: See Appendix A.4. ∎

Together, Lemmas 4.3, 4.4 and 4.5 provide the basis to introduce proof-theoretic equivalents of restricted entailment and observable equivalence.

Definition 4.6. $(\vdash^E_{\Psi\Theta}, =^E_{\Psi\Theta})$

- $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ 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 $\vdash^E_{\Psi\Theta}$ and $=^E_{\Psi\Theta}$ completely characterize restricted entailment and observable equivalence of constraint systems.

Theorem 4.7 (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.4. ∎

4.3 Deciding Observable Equivalence

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

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

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

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

4.4 Regular Grammars

Our first step is to describe, for each set variable α in a simple constraint system $\mathcal{S}$, the following two languages of the lower and upper non-constant bounds of α:

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

These languages are generated by a regular grammar. For each set variable α, the grammar contains the non-terminals α_L and α_U, which generate the above lower and upper bounds of α, respectively.

To illustrate this idea consider the program component $P = (\lambda^g y.((\lambda^f x.1)\ y))$, where f and g are function tags, and take $E = \{\alpha^P\}$. The constraint system $\mathcal{S}_P$ for P (closed under Θ) is described in figure 6, together with the productions in the corresponding regular grammar. This grammar describes the upper and lower non-constant bounds for each set variable in $\Pi(\mathcal{S}_P)\ |_E$. For example, the productions:

$$\begin{array}{lcl} \alpha_L^x & \mapsto & \alpha_L^r \\ \alpha_L^r & \mapsto & \mathtt{dom}(\alpha_U^P) \\ \alpha_U^P & \mapsto & \alpha^P \end{array}$$

imply that $\alpha_L^x \mapsto^* \mathtt{dom}(\alpha_P)$. This lower bound for x means that the value set for x must include all values to which the function P is applied.

Constraints $\mathcal{S}_P$	Grammar $G_r(\mathcal{S}_P, E)$	Additional productions in $G_t(\mathcal{S}_P, E)$
$f \leq \alpha^f$		$R \mapsto [f \leq \alpha_U^f]$
$\mathtt{dom}(\alpha^f) \leq \alpha^x$	$\alpha_L^x \mapsto \mathtt{dom}(\alpha_U^f)$	
$1 \leq \alpha^1$		$R \mapsto [1 \leq \alpha_U^1]$
$\alpha^1 \leq \mathtt{rng}(\alpha^f)$	$\alpha_U^1 \mapsto \mathtt{rng}(\alpha_U^f)$	
$\mathtt{rng}(\alpha^f) \leq \alpha^a$	$\alpha_L^a \mapsto \mathtt{rng}(\alpha_L^f)$	
$\alpha^y \leq \alpha^r$	$\alpha_U^y \mapsto \alpha_U^r \quad \alpha_L^r \mapsto \alpha_L^y$	
$\alpha^r \leq \mathtt{dom}(\alpha^f)$	$\alpha_U^r \mapsto \mathtt{dom}(\alpha_L^f)$	
$g \leq \alpha^P$		$R \mapsto [g \leq \alpha_U^P]$
$\mathtt{dom}(\alpha^P) \leq \alpha^y$	$\alpha_L^y \mapsto \mathtt{dom}(\alpha_U^P)$	
$\alpha^a \leq \mathtt{rng}(\alpha^P)$	$\alpha_U^a \mapsto \mathtt{rng}(\alpha_U^P)$	
$\alpha^r \leq \alpha^x$	$\alpha_U^r \mapsto \alpha_U^x \quad \alpha_L^x \mapsto \alpha_L^r$	
$\alpha^1 \leq \alpha^a$	$\alpha_U^1 \mapsto \alpha_U^a \quad \alpha_L^a \mapsto \alpha_L^1$	
$1 \leq \alpha^a$		$R \mapsto [1 \leq \alpha_U^a]$
	$\alpha_L^P \mapsto \alpha^P \quad \alpha_U^P \mapsto \alpha^P$	$R \mapsto [\alpha_L \leq \alpha_U]\ \forall \alpha \in SetVar(\mathcal{S}_P)$

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

The productions of the grammar are determined by $\mathcal{S}_P$ and Π. For example, the constraint $[\alpha^1 \leq \mathtt{rng}(\alpha^f)] \in \mathcal{S}_P$ implies that for each upper bound κ of α^f, the rule $(compose_1)$ infers the upper bound $\mathtt{rng}(\kappa)$ of α^1. Since, by induction, α^f's upper bounds are generated by α_U^f, the production $\alpha_U^1 \mapsto \mathtt{rng}(\alpha_U^f)$ generates the corresponding upper bounds of α^1.

More generally, the collection of productions:

$$\{\alpha_U \mapsto \mathtt{rng}(\beta_U) \mid [\alpha \leq \mathtt{rng}(\beta)] \in \mathcal{S}\}$$

describes all bounds inferred via $(compose_1)$ on a simple constraint system $\mathcal{S}$. Bounds inferred via the remaining $(compose)$ rules can be described in a similar manner. Bounds

inferred via the rule (*reflex*) imply the productions $\alpha_U \mapsto \alpha$ and $\alpha_L \mapsto \alpha$ for $\alpha \in E$. Finally, consider the rule ($trans_\alpha$), and suppose this rule infers an upper bound τ on α. This bound must be inferred from an upper bound τ on β, using the additional antecedent $[\alpha \leq \beta]$. Hence the productions $\{\alpha_U \mapsto \beta_U \mid [\alpha \leq \beta] \in \mathcal{S}\}$ generate all upper bounds inferred via ($trans_\alpha$). In a similar fashion, the productions $\{\beta_L \mapsto \alpha_L \mid [\alpha \leq \beta] \in \mathcal{S}\}$ generate all lower bounds inferred via ($trans_\alpha$).

Definition 4.8. (*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 $\{\alpha_L, \alpha_U \mid \alpha \in SetVar(\mathcal{S})\}$ and the following productions:

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

∎

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

Lemma 4.9 *Let* $G = G_r(\mathcal{S}, E)$. *Then:*

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

Proof: See Appendix A.5. ∎

4.5 Regular Tree Grammars

The grammar $G_r(\mathcal{S}, E)$ does not describe all constraints in $\Pi(\mathcal{S})\ |_E$. In particular:

- $G_r(\mathcal{S}, E)$ does not describe constraints of the form $[c \leq \tau]$. Thus, for example, the regular grammar for the example program component P does not describe the constraint $[1 \leq \mathtt{rng}(\alpha^P)]$ in $\Pi(\mathcal{S}_P)\ |_E$.
- $G_r(\mathcal{S}, E)$ does not describe constraints inferred by the ($trans_\alpha$) rule that are not bounds of the form $[\kappa \leq \alpha]$ or $[\alpha \leq \kappa]$. To illustrate this idea, consider the program $Q = (\lambda^t x.x)$ whose constraint system is:

  $$\mathcal{S}_Q = \{t \leq \alpha^Q, \mathtt{dom}(\alpha^Q) \leq \alpha^x, \alpha^x \leq \mathtt{rng}(\alpha^Q)\}\ .$$

  The regular grammar $G_r(\mathcal{S}_Q, E)$ for Q describes the constraints $\{\mathtt{dom}(\alpha^Q) \leq \alpha^x,\ \alpha^x \leq \mathtt{rng}(\alpha^Q)\}$ in $\Pi(\mathcal{S}_Q)\ |_E$, but it does not describe the $trans_\alpha$ consequent $[\mathtt{dom}(\alpha^Q) \leq \mathtt{rng}(\alpha^Q)]$ of those constraints, which is also in $\Pi(\mathcal{S}_Q)\ |_E$.

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

Definition 4.10. (*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:

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

where $[\cdot \le \cdot]$ 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 4.11 *Let* $G = G_t(\mathcal{S},E)$. *Then* $\Pi(\mathcal{S})\mid_E = \mathcal{L}_G(R)$.

Proof: See Appendix A.5. ∎

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

$$R \mapsto [1 \le \alpha^a_U] \qquad \alpha^a_U \mapsto \mathtt{rng}(\alpha^P_U) \qquad \alpha^P_U \mapsto \alpha^P$$

imply that $R \mapsto^* [1 \le \mathtt{rng}(\alpha^P)]$, or that the constant 1 is returned as a possible result of the function P.

4.6 Staging

Before we can exploit the grammar representation of $\Pi(\mathcal{S})\mid_E$, we must prove that the closure under $\Theta \cup \Pi \cup \{compat\}$ 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 4.12 (Staging) *For any simple constraint system* $\mathcal{S}$*:*

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

Proof: See Appendix A.5. ∎

4.7 The Entailment Algorithm

We can check entailment based on lemmas 4.11 and 4.12 as follows. Given $\mathcal{S}_1$ and $\mathcal{S}_2$, we close them under Θ and then have:

$$\begin{array}{cll} & \mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1 & \\ \iff & \Psi\Theta(\mathcal{S}_2)\mid_E \supseteq \Pi\Theta(\mathcal{S}_1)\mid_E & \text{by defn } \vdash^E_{\Psi\Theta} \\ \iff & \Psi(\Theta(\mathcal{S}_2))\mid_E \supseteq \Pi(\Theta(\mathcal{S}_1))\mid_E & \text{by lemma 4.12} \\ \iff & \Psi(\mathcal{S}_2)\mid_E \supseteq \Pi(\mathcal{S}_1)\mid_E & \text{as } \mathcal{S}_i = \Theta(\mathcal{S}_i) \\ \iff & compat(\Pi(\mathcal{S}_2)\mid_E) \supseteq \Pi(\mathcal{S}_1)\mid_E & \text{by lemma 4.12} \\ \iff & compat(\mathcal{L}_{G_2}(R)) \supseteq \mathcal{L}_{G_1}(R) & \text{by lemma 4.11} \\ & \text{where } G_i = G_t(\mathcal{S}_i,E) & \end{array}$$

The Entailment Algorithm

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

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

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

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

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

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

then one of the following cases hold:

1. $\mathcal{L}([X \leq Y]) \subseteq \mathcal{L}(C \cup D)$.
2. $X = \texttt{rng}(\alpha'_L)$, $Y = \texttt{rng}(\beta'_U)$ 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 7: The computable entailment relation $\vdash^E_{\text{alg}}$

The containment question:

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

can be decided via a standard 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 7. 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 4.13 (Correctness of the Entailment Algorithm) *$\mathcal{S}_2 \vdash^E_\Psi \mathcal{S}_1$ if and only if $\mathcal{S}_2 \vdash^E_{\mathrm{alg}} \mathcal{S}_1$.*

Proof: See Appendix A.6. ∎

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

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

4.8 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 Lemma 4.11, 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 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 8: The constraint system, grammar and simplified systems for $P = (\lambda^g y.((\lambda^f x.1)\ y))$

4.8.1 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.

Let us illustrate this idea with the program component $P = (\lambda^g y.((\lambda^f x.1)\ y))$ we considered earlier. Although this example is unrealistic, it illustrates the behavior of our simplification algorithms. The solved constraint system $\mathcal{S}_P$ for P is shown in figure 8, together with the corresponding grammar $G_t(\mathcal{S}_P, E)$ 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 $\mathcal{S}_P$ are empty, and are removed by this simplification algorithm, yielding a simplified system of eight non-empty constraints.

4.8.2 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 α^1_U, α^a_U and α^g_U. Three of the constraints are unreachable, and are removed by this algorithm, yielding a simplified system with five reachable constraints.

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 9: The *Hopcroft* algorithm

4.8.3 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 $\Theta(\mathcal{S})$.

4.8.4 Hopcroft's Algorithm

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

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

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

Figure 9 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.[6]

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

Theorem 4.14 (Correctness of the Hopcroft Algorithm) *Let $\mathcal{S}$ be a simple constraint system with external variables E; let $\sim$ be an equivalence relation on the set variables in a constraint system $\mathcal{S}$ satisfying conditions (a) to (e) from figure 9; let the substitution f map each set variable to a representation element of its equivalence class; and let $\mathcal{S}' = f(\mathcal{S})$,* i.e., *$\mathcal{S}'$ denotes the constraint system $\mathcal{S}$ with set variables merged according to their equivalence class. Then $\mathcal{S} \cong_E \mathcal{S}'$.*

Proof: See Appendix A.7. ∎

4.9 Simplification Benchmarks

			empty		*unreachable*		*ϵ-removal*		*Hopcroft*	
Definition	lines	size	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 10: Behavior of the constraint simplification algorithms.

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.

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

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 compiler [9]. The results are described in figure 10. The second column gives the number of lines in each program component, and the third column gives the number of constraints in the original (unsimplified) constraint system after closing it under the rules Θ. The remaining columns describe the behavior of each simplification algorithm, presenting the factor by which the number of constraints was reduced, and the time (in milliseconds) required for this simplification. Since MzScheme is a byte code compiler, porting the simplification algorithms to a native code compiler could be expected to produce a speed-up of roughly a factor of 5.

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.

5 Componential Set-Based Analysis

Equipped with the simplification algorithms, we can now return to our original problem of extending set-based analysis to handle significantly larger programs. These programs are typically constructed as a collection of program components (*e.g.* modules, packages or files). Exploiting this component-based structure is the key to analyzing such programs efficiently.

The following section describes componential set-based analysis. Section 5.2 presents experimental results on the effectiveness of the analysis, and section 5.3 describes how MrSpidey presents the analysis results for multi-component programs to the programmer.

The constraint simplification algorithms also enables an efficient polymorphic, or context-sensitive, analysis that only duplicates a simplified constraint system for each reference to a polymorphic function. A description of this polymorphic analysis is presented in section **??**, together with experimental results on the behavior of the analysis.

5.1 Componential Set-Based Analysis

Componential set-based analysis processes programs in three steps.

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

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

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

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

5.2 Experimental Results

We implemented four variants of componential set-based analysis. Each analysis uses a particular simplification algorithm from chapter **??** to simplify the constraint systems for the program components. We tested these analyses with five benchmark programs, ranging from 1,200 to 17,000 lines. For comparison purposes, we also analyzed each benchmark with the *standard* set-based analysis that performs no simplification. The 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 11.

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.[7] The analyses that exploit constraint simplification yield significant speed-ups over the *standard* analysis because they manipulate much smaller constraint systems. The results indicate that, for these benchmarks, the ϵ*-removal* algorithm yields the best trade-off between efficiency and effectiveness of the simplification algorithms. The additional simplification performed by the more expensive *Hopcroft* algorithm is outweighed by the overhead of running the algorithm. The tradeoff may change as we analyze larger programs.

[7] 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 11: Behavior of the modular analyses.

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 11. These times show an order-of-magnitude improvement in analysis times over the original, *standard* analysis, since the saved constraint files are used to avoid reanalyzing all of the unchanged program components. For example, the analysis of `zodiac`, which used to take over two minutes, now completes in under four seconds. Since practical debugging sessions using MrSpidey typically involve repeatedly analyzing the project each time the source code of one module is modified, *e.g.*, when a bug is identified and eliminated, using separate analysis substantially improves the usability of MrSpidey.

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.

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

5.3 User Interface for Multi-File Programs

We extended MrSpidey's user interface to cope with programs consisting of multiple source files, or components. MrSpidey first analyses the program, using the componential set-based analysis described above, and then displays an annotated version of the program's main file with the usual static debugging mark-ups. The programmer can also view annotated versions of any other source file by using the `File|Open ...` dialog box (shown in Figure **??**) to select the source file of interest.

In multi-file programs, the source (or destination) of an arrow may sometimes refer to a program point in a separate file. In this case MrSpidey draws an arrow originating (or terminating) in the left margin of the program, as shown in figure **??**. Clicking on the arrow provides the option to zoom to and highlight the term at the other end of the arrow, as shown in figures **??** and **??**. These facilities are useful for following the flow of values through multi-file programs.

5.4 Efficient Polymorphic Analysis

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

We tested the analyses using a standard set of benchmarks [13]. The results of the test runs are documented in figure 12. The second column shows the number of lines in each

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

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.

6 Competitive Work

A number of researchers have investigated the problem of constraint simplification in order to derive faster and more scalable analyses and type systems.

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

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

7 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

Lemma 3.4 *(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 is of the form:

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

where:

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

By the Substitution Lemma A.1:

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

By the Flow Lemma 3.6:

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

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:

$$\begin{array}{rcccl} \mathcal{S}_1 & \supseteq & \{\beta_V \leq \mathtt{dom}(\beta_t),\ \mathtt{dom}(\beta_t) \leq \beta_x\} & \models & \{\beta_V \leq \beta_x\} \\ \mathcal{S}_1 & \supseteq & \{\beta_N \leq \mathtt{rng}(\beta_t),\ \mathtt{rng}(\beta_t) \leq \alpha\} & \models & \{\beta_N \leq \alpha\} \end{array}$$

Hence $\mathcal{S}_1 \models \mathcal{S}_2$, 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 is of the form:

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

By the Subject Reduction for **let** Lemma A.2:

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

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 is of the form:

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

Hence $\Gamma \vdash V : \alpha, \mathcal{S}_V$, and by the Flow Lemma 3.6, $\Gamma \vdash V : \beta, \mathcal{S}_2$ where $\mathcal{S}_1 \models \mathcal{S}_2$.

∎

Lemma A.1 (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 number of **let**-expressions in N, and on the size of N:

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

If $x \notin FV[N]$, then $N[x \mapsto V] = N$, $\mathcal{S} = \mathcal{S}_N$ and the lemma trivially holds.

Otherwise we proceed by case analysis on the constraint derivation rule used in the last step in the derivation.

- (*var*): Since $x \in FV[N]$, $N = x$. Hence $S_N = \{\alpha_x \leq \alpha_N\}$, $N[x \mapsto V] = V$ and this case holds via the Flow Lemma 3.6.
- (*const*): This case cannot occur since $x \in FV[N]$.
- (*label*): This case is straightforward.
- (*abs*): In this case $N = (\lambda^t y.M)$, and the constraint 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:

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

  Since $x \in FV[N]$, $x \neq y$. Hence $N[x \mapsto V] = (\lambda^t y.M[x \mapsto V])$. 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 $\mathcal{S}_N \cup \mathcal{S}_V \cup \{\alpha_V \leq \alpha_x\} \models \mathcal{S}$, the lemma holds for this case.

- (*app*): In this case $M = (M_1\ M_2)$, and the constraint 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.

- (*let*): In this case $N = (\textbf{let } (y\ W)\ M)$, where $W \in \mathit{Value}$. Hence:

  $$\Gamma \cup \{x : \alpha_x\} \vdash (\textbf{let } (y\ W)\ M) : \alpha_N, \mathcal{S}_N$$

  and therefore by the following Subject Reduction for **let** Lemma A.2:

  $$\Gamma \cup \{x : \alpha_x\} \vdash M[y \mapsto W] : \alpha_N, \mathcal{S}_N$$

  By induction:

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

  where $\mathcal{S}_N \cup \mathcal{S}_V \cup \{\alpha_V \leq \alpha_x\} \models \mathcal{S}$. Since:

  $$M[y \mapsto W][x \mapsto N] = M[x \mapsto N][y \mapsto W[x \mapsto N]]$$

  we have that:

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

  and therefore, by the Subject Reduction for **let** Lemma A.2:

  $$\Gamma \vdash (\textbf{let } (y\ W[x \mapsto V])\ M[x \mapsto V]) : \alpha_N, \mathcal{S}$$

  or, equivalently:

  $$\Gamma \vdash (\textbf{let } (y\ W)\ M)[x \mapsto V] : \alpha_N, \mathcal{S}$$

  and thus the lemma holds in this case.

- (*inst*): This case cannot occur since $x \in FV[N]$ and x is bound to a set variable in the derivation context.

∎

The constraint derivation rules uses constraint schemas to accurately analyze polymorphic **let**-expressions. The constraint system for a **let**-expression is actually *equivalent* to the constraint system for the corresponding β_{let}-expanded expression[9]. The following lemma demonstrates this equivalence of constraint systems.

Lemma A.2 (Subject Reduction for let)

$$\Gamma \vdash (\textbf{let } (x\ V)\ N) : \alpha_N, \mathcal{S}_N$$

if and only if

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

[9]This equivalence contrasts with the situation for the other reduction rules, where the constraint system for the redex only *entails* the constraint system for the contractum, as shown in the Subject Reduction for $\longrightarrow$ Lemma 3.4.

Proof: The derivation $\Gamma \vdash (\textbf{let } (x\ V)\ N) : \alpha_N, \mathcal{S}_N$ holds if and only if:

$$\Gamma \vdash V : \alpha_V, \mathcal{S}_V$$
$$\overline{\alpha} = SetVar(\mathcal{S}_V) \setminus (FV[range(\Gamma)] \cup Label)$$
$$\sigma = \forall \overline{\alpha}.(\alpha_V, \mathcal{S}_V)$$
$$\Gamma \cup \{x : \sigma\} \vdash N : \alpha_N, \mathcal{S}_N$$

The proof of both directions proceeds by induction on the number of **let**-expressions in N, and on the size of N:

If $x \notin FV[N]$, then $N[x \mapsto V] = N$ and the lemma trivially holds.

Otherwise we proceed by case analysis on the constraint derivation rule used in the last step in the derivation:

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

- (var): This case cannot occur since $x \in FV[N]$ implies $N = x$, but x is bound to a schema in the derivation context and so the rule $(inst)$ applies.
- $(const)$: This case cannot occur since $x \in FV[N]$.
- (abs): In this case $N = (\lambda^t y.M)$, and the typing derivation is:

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

  where:

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

  Since $x \in FV[N]$, $x \neq y$. Hence $N[x \mapsto V] = (\lambda^t y.M[x \mapsto V])$. By induction,

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

  Hence, via (abs):

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

  The reasoning for the converse direction is similar.

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

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

  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,

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

  Hence

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

  as required. The reasoning for the converse direction is similar.

- (*label*): This case is straightforward.
- (*let*): In this case $N = (\textbf{let}\ (y\ W)\ M)$, where $W \in \mathit{Value}$. Hence:

  $$\Gamma \cup \{x : \sigma\} \vdash (\textbf{let}\ (y\ W)\ M) : \alpha_N, \mathcal{S}_N$$

  Since M has fewer **let**-expressions than N, by induction:

  $$\Gamma \cup \{x : \sigma\} \vdash M[y \mapsto W] : \alpha_N, \mathcal{S}_N$$

  Since $M[y \mapsto W]$ has fewer **let**-expressions than N, by induction:

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

  Since:

  $$M[y \mapsto W][x \mapsto N] = M[x \mapsto N][y \mapsto W[x \mapsto N]]$$

  we have that:

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

  Since $M[x \mapsto V][y \mapsto W[x \mapsto V]]$ is smaller than N, by induction:

  $$\Gamma \vdash (\textbf{let}\ (y\ W[x \mapsto V])\ M[x \mapsto V]) : \alpha_N, \mathcal{S}_N$$

  or, equivalently:

  $$\Gamma \vdash (\textbf{let}\ (y\ W)\ M)[x \mapsto V] : \alpha_N, \mathcal{S}_N$$

  and thus the lemma holds in this case.
- (*inst*): Since $x \in FV[N]$, $N = x$, and the derivation on N must be:

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

  where ψ is a substitution of fresh variables for $\overline{\alpha}$ and $\mathcal{S}_N = \psi(\mathcal{S}_V) \cup \{\psi(\alpha_V) \leq \alpha_N\}$. If $\mathcal{D}$ is the derivation concluding

  $$\Gamma \vdash V : \alpha_V, \mathcal{S}_V$$

  then $\psi(\mathcal{D})$ is an analogous derivation concluding

  $$\Gamma \vdash V : \psi(\alpha_V), \psi(\mathcal{S}_V)$$

  Now $N[x \mapsto V] = V$, and by the Flow Lemma 3.6:

  $$\Gamma \vdash V : \alpha_N, \mathcal{S}_N$$

  as required.

∎

Lemma 3.6 *(Flow).* *If* $\Gamma \vdash M : \alpha, \mathcal{S}$ *then for all* $\gamma \in \mathit{SetVar}$, $\Gamma \vdash M : \gamma, \mathcal{S}'$ *with* $\mathcal{S} \cup \{\alpha \leq \gamma\} \models \mathcal{S}'$.

Proof: By induction on the derivation $\Gamma \vdash M : \alpha, \mathcal{S}$ and by case analysis on the last step in this derivation.

- (*var*): In this case the derivation for $M = x$ is:

$$\Gamma' \cup \{x : \beta\} \vdash x : \alpha, \mathcal{S}$$

where $\mathcal{S} = \{\beta \leq \alpha\}$. For any $\gamma \in SetVar$, let $\mathcal{S}' = \{\beta \leq \gamma\}$, and then:

$$\Gamma' \cup \{x : \beta\} \vdash x : \gamma, \mathcal{S}'$$

with $\mathcal{S} \cup \{\alpha \leq \gamma\} \models \mathcal{S}'$, as required.

- (*const*): This case follows by reasoning similar to the (*var*) case.

- (*label*): The derivation for $M = N^l$ must conclude:

$$\frac{\Gamma \vdash N : \beta, \mathcal{S}_N}{\Gamma \vdash N^l : \alpha, \mathcal{S}} \quad (label)$$

where $\mathcal{S} = \mathcal{S}_N \cup \{\beta \leq l, \beta \leq \alpha\}$. Let $\mathcal{S}' = \{\beta \leq l, \beta \leq \gamma\}$, and then:

$$\frac{\Gamma \vdash N : \beta, \mathcal{S}_N}{\Gamma \vdash N^l : \gamma, \mathcal{S}'} \quad (label)$$

with $\mathcal{S} \cup \{\alpha \leq \gamma\} \models \mathcal{S}'$.

- (*abs*): The derivation for $M = (\lambda^t x.N)$ must conclude:

$$\frac{\Gamma \cup \{x : \alpha_1\} \vdash N : \alpha_2, \mathcal{S}_N}{\Gamma \vdash (\lambda^t x.N) : \alpha, \mathcal{S}} \quad (abs)$$

where $\mathcal{S} = \{t \leq \alpha, \mathtt{dom}(\alpha) \leq \alpha_1, \alpha_2 \leq \mathtt{rng}(\alpha)\}$.
Let $\mathcal{S} = \{t \leq \gamma, \mathtt{dom}(\gamma) \leq \alpha_1, \alpha_2 \leq \mathtt{rng}(\gamma)\}$, and then:

$$\frac{\Gamma \cup \{x : \alpha_1\} \vdash N : \alpha_2, \mathcal{S}_N}{\Gamma \vdash (\lambda^x N.:)\gamma, \mathcal{S}} \quad (abs)$$

with $\mathcal{S} \cup \{\alpha \leq \gamma\} \models \mathcal{S}'$.

- (*app*): This case follows by reasoning similar to the (*app*) case.

- (*let*): This case follows by induction.

- (*abs*): The derivation for $M = x$ must be:

$$\Gamma \cup \{x : \forall \overline{\alpha}.\, (\alpha_V, \mathcal{S}_V)\} \vdash x : \alpha, \mathcal{S}$$

where $\mathcal{S} = \psi(\mathcal{S}_V) \cup \{\psi(\alpha_V) \leq \alpha\}$, and ψ is a substitution of set variables for $\overline{\alpha}$.
Let $\mathcal{S}' = \psi(\mathcal{S}_V) \cup \{\psi(\alpha_V) \leq \gamma\}$. Then

$$\Gamma \cup \{x : \forall \overline{\alpha}.\, (\alpha_V, \mathcal{S}_V)\} \vdash x : \gamma, \mathcal{S}'$$

with $\mathcal{S} \cup \{\alpha \leq \gamma\} \models \mathcal{S}'$.

∎

A.2 Proofs for Computing Set-Based Analysis

Lemma 3.9 *(Least Solution of Simple Constraint Systems). Every simple 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 that $\rho \in Soln(\mathcal{S})$ by showing that ρ satisfies any constraint $\mathcal{C} \in \mathcal{S}$. The proof proceeds by case analysis on $\mathcal{C}$.

- The case where $\mathcal{C} = [\alpha \leq \beta]$ follows from Lemma A.3.
- 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:

$$\begin{array}{rcll}
\rho(\alpha) & = & \displaystyle\sqcap_s \atop \rho' \in Soln(\mathcal{S}) & \rho'(\alpha) \\
& \sqsubseteq & \displaystyle\sqcap_s \atop \rho' \in Soln(\mathcal{S}) & \rho'(\mathtt{rng}(\beta)) \\
& & \multicolumn{2}{l}{\text{by Lemma A.3, since } \rho'(\alpha) \sqsubseteq \rho'(\mathtt{rng}(\beta))} \\
& = & rng \left(\displaystyle\sqcap_s \atop \rho' \in Soln(\mathcal{S}) \right. & \left. \rho'(\beta) \right) \qquad \text{by definition of } \sqcap_s \\
& = & rng\,(\rho(\beta)) & \\
& = & \rho(\mathtt{rng}(\beta)) &
\end{array}$$

  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 $\sqcap_s \{\rho \mid \rho \models \mathcal{S}\}$. ∎

The following lemma describes some properties about how the two orderings $\sqsubseteq$ and $\sqsubseteq_s$ defined on $\mathcal{D}$ interact.

Lemma A.3 *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:*

$$\sqcap_s \atop i \in I \; x_i \quad \sqsubseteq \quad \sqcap_s \atop i \in I \; y_i$$

$$\bigsqcup_s \atop i \in I \; x_i \quad \sqsubseteq \quad \bigsqcup_s \atop i \in I \; y_i$$

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

$$\sqcap \atop i \in I \; x_i \quad \sqsubseteq_s \quad \sqcap \atop i \in I \; y_i$$

$$\bigsqcup \atop i \in I \; x_i \quad \sqsubseteq_s \quad \bigsqcup \atop i \in I \; y_i$$

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

$$f : \{\mathtt{dom}, \mathtt{rng}\}^* \longrightarrow \mathcal{P}(Const)$$

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

To prove the first relation, assume $x_i \sqsubseteq y_i$ for all $i \in I$, and let p be a path in $\{\mathtt{dom}, \mathtt{rng}\}^*$. If p is monotonic, then $p(x_i) \sqsubseteq p(y_i)$. Hence:

$$\begin{aligned} p\left(\sqcap_{\substack{s \\ i\in I}} x_i\right) &= \sqcap_{\substack{s \\ i\in I}} p(x_i) \\ &\sqsubseteq \sqcap_{\substack{s \\ i\in I}} p(y_i) \\ &= p\left(\sqcap_{\substack{s \\ i\in I}} y_i\right) \end{aligned}$$

Conversely, if p is anti-monotonic, then $p(x_i) \sqsupseteq p(y_i)$. Hence:

$$\begin{aligned} p\left(\sqcap_{\substack{s \\ i\in I}} x_i\right) &= \sqcap_{\substack{s \\ i\in I}} p(x_i) \\ &\sqsupseteq \sqcap_{\substack{s \\ i\in I}} p(y_i) \\ &= p\left(\sqcap_{\substack{s \\ i\in I}} y_i\right) \end{aligned}$$

Hence

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

as required. ∎

Theorem 3.12 *If $P \in \Lambda^0$ and $\emptyset \vdash P : \alpha, \mathcal{S}$ is a most general constraint derivation then:*

$$\begin{aligned} sba(P)(l) = \quad & \{b \mid \mathcal{S} \vdash_\Theta b \leq l\} \\ \cup \quad & \{(\lambda^t x.M) \mid \mathcal{S} \vdash_\Theta t \leq l\} \end{aligned}$$

Proof:

$$\begin{aligned} \mathcal{S} \vdash_\Theta c \leq \alpha &\iff \mathcal{S} \models c \leq \alpha && \text{by lemma A.4} \\ &\iff \forall \rho \models \mathcal{S}.\ \rho \models c \leq \alpha \\ &\iff \forall \rho \in Soln(\mathcal{S}).\ c \in const(\rho(\alpha)) \\ &\iff c \in \bigcap(\{const(\rho(\alpha)) \mid \rho \in Soln(\mathcal{S})\}) \\ &\iff c \in const(\sqcap(\{\rho(\alpha) \mid \rho \in Soln(\mathcal{S})\})) \\ &\iff c \in const(\sqcap(\{\rho \mid \rho \in Soln(\mathcal{S})\})(\alpha)) \\ &\iff c \in const(LeastSoln(\mathcal{S})(\alpha)) \end{aligned}$$

The correctness of this theorem then follows from definition 3.10. ∎

Lemma A.4 (Soundness and Completeness of Θ) *For any simple constraint system $\mathcal{S}$:*

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

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

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

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

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

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

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

- Suppose $\mathcal{C} = [\alpha \leq \beta]$. We need to show that the correct ordering holds between the corresponding components of $\rho(\alpha)$ and $\rho(\beta)$. For the first component, by (s_1), which is the first rule in Θ:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.3 Proofs for Conditions for Constraint Simplification

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

Lemma 4.1 *(Admissibility of ($\cong$)). If $\emptyset \vdash_\cong P : \alpha, \mathcal{S}$ is a most general constraint derivation then:*

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

Proof: This lemma follows from the induction hypothesis:

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

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

- 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 $\mathcal{S}_3 \cong_E \mathcal{S}_4$. Since $\cong_E$ is an equivalence relation, $\mathcal{S}_1 \cong_E \mathcal{S}_2$, and hence the lemma holds.

∎

A.4 Proofs for Proof Theory of Observable Equivalence

The following proofs require a number of auxiliary definitions.

Definition A.5. (*Paths*)

- A *path* $p, q \in Path$ is a sequence of the constructors `dom` and `rng`. We use ϵ to denote the empty sequence, and $p.q$ to denote the concatenation of the paths p and q.
- The *arity* of a path p, denoted πp, is the number of `dom`'s in p, taken modulo 2. If πp is 0, we say p is *monotonic*, otherwise p is *anti-monotonic*.
- For a path p, the notation $p(\tau)$ denotes the set expression τ enclosed in the `dom`'s and `rng`'s of p, *i.e.*, if $p \equiv \texttt{rng.dom}$, then $p(\alpha) \equiv \texttt{rng}(\texttt{dom}(\alpha))$.
- The relations $\leq_0$ and $\leq_1$ denote $\leq$ and $\geq$, respectively.
- The relations $\sqsubseteq_0$ and $\sqsubseteq_1$ denote $\sqsubseteq$ and $\sqsupseteq$, respectively.
- The relations $\sqsupseteq_0$ and $\sqsupseteq_1$ denote $\sqsupseteq$ and $\sqsubseteq$, respectively.
- The relations $\subseteq_0$ and $\subseteq_1$ denote $\subseteq$ and $\supseteq$, respectively.
- The operations $\bigsqcup_0$ and $\bigsqcup_1$ denote $\bigsqcup$ and $\sqcap$, respectively.
- For a path p and a domain element $X \in \mathcal{D}$, the notation $p(X)$ extracts the component of X at the position p. This notation is formalized as follows:

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

- For a path p and a domain element $X \in \mathcal{D}$, the notation $X@p$ is defined as follows:

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

∎

Lemma 4.2 *(Soundness and Completeness of Δ). For a compound constraint system* $\mathbf{S}$ *and a compound constraint* $\mathbf{C}$*:*

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

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

- Suppose $\mathbf{C} = [c \leq \kappa]$. Define ρ by:

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

  We prove $\rho \models \mathbf{S}$ by a case analysis showing that ρ satisfies every constraint $\mathbf{C}' \in \mathbf{S}$.

  - Suppose $\mathbf{C}' = [c \leq q(\beta)]$. Then, by the definition of ρ, $c \in \mathit{const}(\rho(q(\beta)))$, and hence $\rho \models c \leq q(\beta)$.
  - Suppose $\mathbf{C}' = [p(\alpha) \leq q(\beta)]$. We need to show that $\rho(p(\alpha)) \sqsubseteq \rho(q(\beta))$. We prove this inequality by showing that for any path r:

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

    If r is monotonic, then:

$$\begin{array}{rl} & \mathit{const}(r(\rho(p(\alpha)))) \\ = & \mathit{const}(r(p(\rho(\alpha)))) \\ = & \{c \mid \mathbf{S} \vdash_\Delta c \leq r(p(\alpha))\} \\ \subseteq & \{c \mid \mathbf{S} \vdash_\Delta c \leq r(q(\beta))\} \\ & \text{via } (\mathit{trans}_\tau), \text{ since } [p(\alpha) \leq q(\beta)] \in \mathbf{S} \\ & \text{and hence } \mathbf{S} \vdash_\Delta r(p(\alpha)) \leq r(q(\beta)) \text{ via } (\mathit{compat}) \\ = & \mathit{const}(r(q(\rho(\beta)))) \\ = & \mathit{const}(r(\rho(q(\beta)))) \end{array}$$

    The case where r is anti-monotonic follows by a similar argument.

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

$$\begin{array}{rcl} c & \in & \mathit{const}(\rho(p(\alpha))) \\ & = & \mathit{const}(p(\rho(\alpha))) \\ & = & \{c \mid \mathbf{S} \vdash_\Delta c \leq p(\alpha)\} \end{array}$$

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

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

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

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

  We now show that for any κ', $\mathbf{S}' \vdash_\Delta c \leq \kappa'$ if and only if $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa'$. We prove this hypothesis by induction on the derivation of $\mathbf{S}' \vdash_\Delta c \leq \kappa'$.

  - Suppose $\mathbf{S}' \vdash_\Delta c \leq \kappa'$ because $[c \leq \kappa'] \in \mathbf{S}'$. Then $\kappa' = \kappa_1$, and by the rule (*reflex*), $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa_1$, as required.
  - If $[c \leq \kappa'] \notin \mathbf{S}'$, then $\mathbf{S}' \vdash_\Delta c \leq \kappa'$ must be derived via the rule (trans_τ) based on the antecedents $\mathbf{S}' \vdash_\Delta \{c \leq \kappa'', \kappa'' \leq \kappa'\}$. By induction, $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa''$. Hence $\mathbf{S} \vdash_\Delta \kappa_1 \leq \kappa'$ via (trans_τ), as required.

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

∎

Lemma 4.3 *For a compound constraint system* $\mathbf{S}$, $\mathbf{S} \cong_E \Delta(\mathbf{S}) \mid_E$.

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

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

Since the rules Δ are sound:

$$\begin{array}{rcl} \mathit{Soln}(\mathbf{S}) \mid_E & = & \mathit{Soln}(\Delta(\mathbf{S})) \mid_E \\ & \subseteq & \mathit{Soln}(\Delta(\mathbf{S}) \mid_E) \mid_E \end{array}$$

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

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

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

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

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

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

  as required.

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

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

  Therefore:

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

  Hence:

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

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

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

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

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

Lemma A.6 *For any $p \in Path$ and $X \in \mathcal{D}$, $p(X@p) = X$.*

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

Lemma 4.4 *(Equivalence of Proof Systems). For a simple constraint system $\mathcal{S}$:*

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

Proof: We show that $\Psi\Theta(\mathcal{S}) \subseteq \Delta(\mathcal{S})$ by induction on the derivation of $\mathbf{C} \in \Psi\Theta(\mathcal{S})$. For the base case, if $\mathbf{C} \in \Psi\Theta(\mathcal{S})$ because $\mathbf{C} \in \mathcal{S}$, then $\mathbf{C} \in \Delta(\mathcal{S})$. Otherwise we proceed by case analysis on the last rule used in the derivation of $\mathbf{C} \in \Psi\Theta(\mathcal{S})$.

- ($compose_1$): In this case $\mathbf{C} = [\alpha \leq \mathtt{rng}(\kappa)]$ is derived from the antecedents $\{\alpha \leq \mathtt{rng}(\beta),\ \beta \leq \kappa\} \in \Psi\Theta(\mathcal{S})$. By induction, these antecedents are also in $\Delta(\mathcal{S})$, and hence the following derivation shows that $\mathbf{C} \in \Delta(\mathcal{S})$:

$$\frac{\alpha \leq \mathtt{rng}(\beta) \qquad \dfrac{\beta \leq \kappa}{\mathtt{rng}(\beta) \leq \mathtt{rng}(\kappa)}\ (compat)}{\alpha \leq \mathtt{rng}(\kappa)}\ (trans_\tau)$$

- $(compose_2)$, $(compose_3)$, $(compose_3)$: These cases follow by similar reasoning.
- $(reflex)$, $(trans_\alpha)$, $(compat)$: These rules are either equivalent to or subsumed by corresponding rules in Δ.
- (s_1), (s_4), (s_5): For these cases $\mathbf{C} \in \Delta(\mathcal{S})$ via $(trans_\tau)$.
- (s_2), (s_3): These rules are special cases of the rules $(compose_1)$ and $(compose_4)$, respectively.

There are no other possibilities for the derivation $\mathbf{C} \in \Psi\Theta(\mathcal{S})\backslash\Delta(\mathcal{S})$, and hence $\Psi\Theta(\mathcal{S}) \subseteq \Delta(\mathcal{S})$.

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

- $(reflex)$, $(compat)$: These rules are also in Ψ and, by induction, the antecedents are in $\Psi\Theta(\mathcal{S})$, hence $\mathbf{C} \in \Psi\Theta(\mathcal{S})$.
- $(trans_\tau)$: The last step in the derivation must be:

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

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

  - The case $\tau = c$ is impossible, since $[\tau_1 \leq c]$ is not a compound constraint.
  - If $\tau \in SetVar$, then $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$ via $(trans_\alpha)$.
  - Suppose $\tau = \mathtt{rng}(\tau')$. If $\tau' \in SetVar$ then $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$ via (s_4).
    Otherwise $\tau_1 \leq \mathtt{rng}(\tau')$ and $\mathtt{rng}(\tau') \leq \tau_2$ are not simple constraints, and we proceed by considering the derivation of these constraints in $\Delta(\mathcal{S})$. The last step in the derivation of $\tau_1 \leq \mathtt{rng}(\tau')$ is either via:

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

    where $\tau_1 = \alpha_1$, or:

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

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

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

    where $\tau_2 = \alpha_2$, or:

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

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

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

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

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

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

* Suppose $\tau_1 \leq \mathtt{rng}(\tau')$ is inferred via $(compat)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(compose_3)$. This case holds by similar reasoning to the previous case.
* Suppose $\tau_1 \leq \mathtt{rng}(\tau')$ is inferred via $(compat)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(compat)$. Then $\{\tau_1' \leq \tau', \tau' \leq \tau_2'\} \subseteq \Delta(\mathcal{S})$, and therefore $[\tau_1' \leq \tau_2'] \in \Delta(\mathcal{S})$ via $(trans_\alpha)$. By induction, $[\tau_1' \leq \tau_2'] \in \Psi\Theta(\mathcal{S})$, and therefore a $(compat)$-inference shows that $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$.

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

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

There are no other possibilities for τ.

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

Lemma 4.5 $\Psi\Theta(\mathcal{S}) \mid_E \cong \Pi\Theta(\mathcal{S}) \mid_E$.

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

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

Now:

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

To prove the converse direction, let $\rho \models \Pi\Theta(\mathcal{S}) \mid_E$. If $\rho \not\models compat(\Pi\Theta(\mathcal{S})) \mid_E$, then let $\mathbf{C}$ be the constraint in $compat(\Pi\Theta(\mathcal{S})) \mid_E$ with the smallest derivation such that $\rho \not\models \mathbf{C}$. Then the last step in the derivation of $\mathbf{C}$ must be via $(compat)$. Let $\mathbf{C}'$ be the antecedent of this rule in

$compat(\Pi\Theta(\mathcal{S}))$. Then $SetVar(\mathbf{C}') = SetVar(\mathbf{C}) \subseteq E$, and hence $\mathbf{C}' \in compat(\Pi\Theta(\mathcal{S}))\ |_E$ with a smaller derivation. Therefore $\rho \models \mathbf{C}'$, and hence since (*compat*) is sound, $\rho \models \mathbf{C}$. Thus $\rho \not\models compat(\Pi\Theta(\mathcal{S}))\ |_E$, as required. ∎

Theorem 4.7 [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* $\mathcal{S}_1 \cong_E \mathcal{S}_2$.

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

$$
\begin{array}{lll}
 & \mathcal{S}_1 \models_E \Delta(\mathcal{S}_2) & \text{by the soundness of } \Delta \\
\therefore & \mathcal{S}_1 \models_E \Delta(\mathcal{S}_2)\ |_E & \\
\therefore & \forall \mathcal{C} \in \Delta(\mathcal{S}_2)\ |_E.\ \ \mathcal{S}_1 \models \mathcal{C} & \\
\therefore & \forall \mathcal{C} \in \Delta(\mathcal{S}_2)\ |_E.\ \ \mathcal{S}_1 \vdash_\Delta \mathcal{C} & \text{by Lemma 4.2} \\
\therefore & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2)\ |_E.\ \ \mathcal{S}_1 \vdash_\Delta \mathcal{C} & \text{by Lemma 4.4} \\
\therefore & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2)\ |_E.\ \ \mathcal{S}_1 \vdash_{\Psi\Theta} \mathcal{C} & \text{by Lemma 4.4} \\
\therefore & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2)\ |_E.\ \ \mathcal{C} \in \Psi\Theta(\mathcal{S}_1)\ |_E & \\
\therefore & \Psi\Theta(\mathcal{S}_1)\ |_E \supseteq \Pi\Theta(\mathcal{S}_2)\ |_E & \\
\therefore & \mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2 &
\end{array}
$$

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

$$
\begin{array}{ll}
 & \Psi\Theta(\mathcal{S}_1)\ |_E \supseteq \Pi\Theta(\mathcal{S}_2)\ |_E \\
\therefore & Soln(\Psi\Theta(\mathcal{S}_1)\ |_E) \subseteq Soln(\Pi\Theta(\mathcal{S}_2)\ |_E) \\
\therefore & Soln(\Psi\Theta(\mathcal{S}_1)\ |_E)\ |_E \subseteq Soln(\Pi\Theta(\mathcal{S}_2)\ |_E)\ |_E \\
\therefore & Soln(\mathcal{S}_1)\ |_E \subseteq Soln(\mathcal{S}_2)\ |_E \\
 & \text{by Lemmas 4.3, 4.4 and 4.5, since } Soln(\mathcal{S}_i)\ |_E = Soln(\Pi\Theta(\mathcal{S}_i)\ |_E)\ |_E \\
\therefore & \mathcal{S}_1 \models_E \mathcal{S}_2
\end{array}
$$

2. Follows from part 1.

A.5 Proofs for Deciding Observable Equivalence

We repeat definition 4.8 here, to avoid having to refer back to the original definition earlier in the text.

Definition 4.8 (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 $\{\alpha_L, \alpha_U \mid \alpha \in SetVar(\mathcal{S})\}$ and the following productions:

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

∎

Lemma 4.9 *Let $G = G_r(\mathcal{S}, E)$. Then:*

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

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

- Suppose $\mathbf{C} \in \Psi(\mathcal{S})$ because $\mathbf{C} \in \mathcal{S}$. We proceed by case analysis on $\mathbf{C}$.

  - Suppose $\mathbf{C} = [\alpha \leq \beta]$. Then $\alpha \in E$, so $\beta_L \mapsto \alpha_L$ and $\alpha_L \mapsto \alpha$ are productions in the grammar. Hence $\alpha \in \mathcal{L}_G(\beta_L)$. Similarly, $\beta \in \mathcal{L}_G(\alpha_U)$.

  The remaining cases for $\mathbf{C}$ follow by similar reasoning.

- Suppose $\mathbf{C} = [\alpha \leq \mathtt{rng}(\kappa)]$ is inferred via $(compose_1)$ 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. ∎

We repeat definition 4.10 here, to avoid having to refer back to the original definition earlier in the text.

Definition 4.10 (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:

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

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

Lemma 4.11 *Let $G = G_t(\mathcal{S}, E)$. Then $\Pi(\mathcal{S}) \mid_E = \mathcal{L}_G(R)$.*

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

- Suppose $\mathbf{C} = [\alpha \leq \kappa]$. Then by Lemma 4.9, $\alpha_U \mapsto_G^* \kappa$. Since $SetVar(\mathbf{C}) \subseteq E$, $\alpha \in E$, and hence $\alpha_L \mapsto_G \alpha$. Thus $R \mapsto_G [\alpha_L \leq \alpha_U] \mapsto_G^* [\alpha \leq \kappa]$, and hence $[\alpha \leq \kappa] \in \mathcal{L}_G(R)$.

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

- Suppose $\mathbf{C} = [c \leq \kappa]$. If $\mathbf{C} \in \mathcal{S}$, then $\kappa = \alpha$, $\alpha \in E$, and

  $$R \mapsto_G [c \leq \alpha_U] \mapsto_G [c \leq \alpha]$$

  as required.

  If $\mathbf{C} \notin \mathcal{S}$, then an examination of the inference rules in Π shows that $\mathbf{C}$ can only be inferred via $(trans_\alpha)$, based on the antecedents $[c \leq \alpha]$ and $[\alpha \leq \kappa]$. By Lemma 4.9, $\alpha_L \mapsto_G^* c$ 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 $\mathbf{C} = [\kappa_1 \leq \kappa_2]$, where $\kappa_1, \kappa_2 \notin SetVar$. An examination of the inference rules in Π shows that $\mathbf{C}$ can only be inferred via $(trans_\alpha)$, based on the antecedents $[\kappa_1 \leq \alpha]$ and $[\alpha \leq \kappa_2]$. By Lemma 4.9, $\alpha_L \mapsto_G^* \kappa_1$ 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 $\mathbf{C} \in \mathcal{L}_G(R)$.

- Suppose $\mathbf{C} = [\kappa_1 \leq \kappa_2]$. Then for some α, $\alpha_L \mapsto_G^* \kappa_1$ and $\alpha_U \mapsto_G^* \kappa_2$. By Lemma 4.9, $\{\kappa_1 \leq \alpha,\ \alpha \leq \kappa_2\} \subseteq \Psi(\mathcal{S})$ and $SetVar(\kappa_i) \subseteq E$. By Lemma 4.12, $\{\kappa_1 \leq \alpha,\ \alpha \leq \kappa_2\} \subseteq \Pi(\mathcal{S})$. Hence $[\kappa_1 \leq \kappa_2] \in \Pi(\mathcal{S})\ |_E$, as required.

- Otherwise $\mathbf{C} = [c \leq \kappa]$. Then for some α, $[c \leq \alpha] \in \mathcal{S}$ and $\alpha_U \mapsto_G^* \kappa$. By Lemma 4.9, $\{\alpha \leq \kappa\} \subseteq \Psi(\mathcal{S})$ and $SetVar(\kappa) \subseteq E$. By Lemma 4.12, $\{\alpha \leq \kappa\} \subseteq \Pi(\mathcal{S})$. Hence $[c \leq \kappa] \in \Pi(\mathcal{S})\ |_E$, as required.

∎

Lemma 4.12 (Staging) *For any simple constraint system $\mathcal{S}$:*

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

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

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

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

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

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

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

The Entailment Algorithm

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

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

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

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

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

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

then one of the following cases hold:

1. $\mathcal{L}([X \leq Y]) \subseteq \mathcal{L}(C \cup D)$.
2. $X = \mathtt{rng}(\alpha'_L)$, $Y = \mathtt{rng}(\beta'_U)$ and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha'_L, \beta'_U, C, D']$, where
$$D' = \{\langle \gamma'_L, \delta'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U)\}$$
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 13: The computable entailment relation $\vdash^E_{\text{alg}}$

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_α)

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

∎

A.6 Correctness of the Entailment Algorithm

Theorem 4.13 *(Correctness of the Entailment Algorithm).* $\mathcal{S}_2 \vdash^E_\Psi \mathcal{S}_1$ *if and only if* $\mathcal{S}_2 \vdash^E_{\text{alg}} \mathcal{S}_1$.

Proof: The definitions of the computable entailment relation and the relation $\mathcal{R}$ are shown in figure 13. We prove this theorem based on the following invariant concerning the relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\cdot,\cdot,\cdot,\cdot]$:

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

Assume this relation holds, and that $\mathcal{S}_2 \vdash^E_\Psi \mathcal{S}_1$. Then $\Pi(\mathcal{S}_1) \mid_E \subseteq \mathit{compat}(\Pi(\mathcal{S}_2)) \mid_E$. By lemma 4.11, $\Pi(\mathcal{S}_i) \mid_E = \mathcal{L}_{G_i}(R)$, and hence:

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

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

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

Hence:

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

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

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

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

It remains to show that the invariant concerning $\mathcal{R}$ holds. 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 definition of $\mathcal{R}$ must hold.

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

2. In this case:

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

   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 13 hold. Furthermore, since $\mathcal{R}$ is the largest relation satisfying the conditions in figure 13, 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 one. 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:
    $$\gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L) \mapsto^*_{G_2} \mathtt{rng}(\tau_L)$$
    $$\delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U) \mapsto^*_{G_2} \mathtt{rng}(\tau_U)$$
    Hence $[\tau_L \leq \tau_U] \in \mathcal{L}(D')$, where:
    $$D' = \{\langle \gamma'_L, \delta'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U)\}$$
  - Otherwise $[\mathtt{rng}(\tau_L) \leq \mathtt{rng}(\tau_U)] \in \mathit{compat}(\mathcal{L}(C)) \setminus \mathcal{L}(C)$, 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.

- 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 on $\mathcal{R}$ is true, and thus the lemma holds. ∎

A.7 Correctness of the Hopcroft Algorithm

Theorem 4.14 *(Correctness of the Hopcroft Algorithm). Let $\mathcal{S}$ be a simple constraint system with external variables E; let $\sim$ be an equivalence relation on the set variables in a constraint system $\mathcal{S}$ satisfying conditions (a) to (e) from figure 9; let the substitution f map each set variable to a representation element of its equivalence class; and let $\mathcal{S}' = f(\mathcal{S})$,* i.e., *$\mathcal{S}'$ denotes the constraint system $\mathcal{S}$ with set variables merged according to their equivalence class. Then $\mathcal{S} \cong_E \mathcal{S}'$.*

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

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

Obviously ρ, ρ' agree on E by condition (a) on $\sim$. 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 for all α' such that $\alpha \sim \alpha'$ there exists β' such that $\beta \sim \beta'$ and:

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

  Hence for all $\alpha \sim \alpha'$:

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

  and therefore:

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

  Hence:

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

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

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

and therefore:

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

Hence:

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

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

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

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

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

∎

B Notations

Symbol	Meaning	Section	Page
$M \in \Lambda$	Terms	2	4
$V \in \mathit{Value}$	Values	2	4
$x \in \mathit{Vars}$	Variables	2	4
$b \in \mathit{BasicConst}$	Basic constants	2	4

$t \in Tag$	Function tags	2	4
$l \in Label$	Labels	2	4
$\beta_v, \beta_{let}, unlabel$	Reduction rules	3.1.2	4
$\mathcal{E}$	Evaluation contexts	3.1.2	4
$\longmapsto, \longmapsto^*$	Standard reduction relation	3.1.2	5
$eval$	Evaluator	3.1.2	5
$\tau \in SetExp$	Set expressions	3.2	5
$\alpha, \beta, \ldots \in SetVar$	Set variables	3.2	5
$c \in Const$	Constants	3.2	5
`dom`, `rng`	Type expression constructors	3.2	5
$\mathcal{C}$	Constraints	3.2	5
$\mathcal{S}$	Constraint systems	3.2	5
$\mathcal{S} \mid_E$	Restriction of a constraint system	3.2	5
$\mathcal{D}, \sqsubseteq, \top, \bot, \sqcup, \sqcap$	Domain, ordering, elements and operations	3.3.1	6
$\mathcal{P}$	Power set constructor	3.2	5
in	Values described by constants	3.3.1	6
$const, dom, rng$	Extract components of element of $\mathcal{D}$	??	??
$\rho \in SetEnv$	Set environment	3.3.2	7
$Soln(\mathcal{S})$	Solution space	3.3.2	7
$\models, \models_E$	Satisfies, or entails	3.3.2	7
$Soln(\mathcal{S}) \mid_E$	Restriction	3.3.2	7
$\cong_E$	Observable equivalence	3.1	8
$\vdash$	Constraint derivation rules	3.4	8
Γ	Set variable context	3.4	8
σ	Constraint schema	3.4	9
$\sqsubseteq_s, \top_s, \bot_s, \sqcup_s, \sqcap_s$	Alternative ordering on domain	3.4	9
sba	Analysis function	3.10	12
$LeastSoln$	Least Solution	3.6	12
$\Theta = \{s_1, \ldots, s_n\}$	Inference rules	3.6.1	13
$\vdash$	Deduction	3.6.1	13
E	External variables	4.1	14
$FV[rng(\Gamma)]$	Free variables in Γ	3.4	9
Ψ, Π	Inference rules on constraint systems	4.2	17
G	Grammar	4.8	20
G_r	Function producing regular grammar	4.8	20
α_L, α_U	Grammar non-terminals	4.8	20
$\mathcal{L}_G(X)$	Language for X in G	4.8	20
G_t	Function producing RTG	4.10	21

R	Root non-terminal	4.10	21
p, q	Paths	A.4	43
π	Arity function	A.4	43
$\leq_i$	$\leq$ or $\geq$	A.4	43
`sel`	Either `dom` or `rng`	A.4	43
$\overline{C}$	Converts compound to simple constraints	A.4	43
$\cdot@\cdot$	Injection function	A.4	43
$\leq^*$	Transitive closure of $\leq$	A.2	41

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.