RICE UNIVERSITY

# Effective Static Debugging via Componential Set-Based Analysis

by

**Cormac Flanagan**

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

**Doctor of Philosophy**

APPROVED, THESIS COMMITTEE:

Matthias Felleisen
Professor of Computer Science

Robert S. Cartwright
Professor of Computer Science

J. E. Dennis
Noah Harding Professor of Computational
and Applied Mathematics

Houston, Texas

May, 1997

# Abstract

## Effective Static Debugging via Componential Set-Based Analysis

by Cormac Flanagan

Sophisticated software systems are inherently complex. Understanding, debugging and maintaining such systems requires inferring high-level characteristics of the system's behavior from a myriad of low-level details. For large systems, this quickly becomes an extremely difficult task.

MrSpidey is a static debugger that augments the programmers ability to deal with such complex systems. It statically analyzes the program and uses the results of the analysis to identify and highlight any program operation may cause a run-time fault. The programmer can then investigate each potential fault site and, using the graphical explanation facilities of MrSpidey, determine if the fault will really happen or whether the corresponding correctness proof is beyond the analysis's capabilities. In practice, MrSpidey has proven to be an effective tool for debugging program under development and understanding existing programs.

The key technology underlying MrSpidey is componential set-based analysis. This is a constraint-based, whole-program analysis for object-oriented and functional programs. The analysis first processes each program component (eg. module or package) independently, generating and simplifying a constraint system describing the data flow behavior of that component. The analysis then combines and solves these simplified constraint systems to yield invariants characterizing the run-time behavior of the entire program. This component-wise approach yields an analysis that handles significantly larger programs than previous analyses of comparable accuracy.

The simplification of constraint systems raises a number of questions. In particular, we need to ensure that simplification preserves the observable behavior, or solution space, of a constraint system. This dissertation provides a complete proof-theoretic and algorithmic characterization of the observable behavior of constraint systems, and establishes a close connection between the observable equivalence of constraint systems and the equivalence of regular tree grammars. We exploit this connection to develop a complete algorithm for deciding the observable equivalence of constraint systems, and to adapt a variety of algorithms for simplifying regular tree grammars to the problem of simplifying constraint systems. The resulting constraint simplification algorithms yield an order of magnitude reduction in the size of constraint systems for typical program expressions.

Acknowledgments

Many people have contributed to making my graduate career rewarding and enjoyable. First, and foremost, I would like to thank my advisor, Matthias Felleisen. He got me started in research, taught me a great deal about programming language semantics and pragmatics (with the occasional detour into philosophy or economics), spent hours giving me feedback on research ideas and papers, and inspired much of MrSpidey's user interface. I would also like to thank my other committee members: Robert (Corky) Cartwright, for valuable feedback on this and other research, and for the class he taught on fully abstract denotational semantics, which led to some of the key ideas of this thesis; and John Dennis, for taking time off his already busy schedule to serve on my committee.

My research environment at Rice was valuably enriched by my collogues in the programming languages theory group. I would like to thank all the members for their various contributions, including Matthew, Shriram, and Robby, who provided the DrScheme infrastructure without which the development of MrSpidey would have been impossible, and Amr, Andrew, Bruce, Mike and John, who helped me get started on research when I arrived at Rice.

Outside the programming languages group there are many other people at Rice whose valuable support and friendship helped me see this thesis through to completion, and who made my time at Rice more enjoyable.

Finally, this research would not have been possible with the support of Rice University, the Professional Activities Committee of the Association for Computing Machinery's Special Interest Group on Programming Languages, and the National Science Foundation.

I gratefully acknowledge all this help.

# Contents

# Chapter 1

# The Need for Static Debugging

Sophisticated software systems are inherently complex. A typical program such as a compiler or a word processor contains an enormous amount of detail. Developing, maintaining, or debugging this kind of system requires inferring high-level characteristic of the system's behavior from a large number of low-level details. For large systems, this quickly becomes an extremely difficult task, particularly since programmers, being human, have finite limits in the amount of complexity they can manage.

MrSpidey is a static debugger for Scheme that augments the programmers ability to deal with such complex systems. It statically analyzes the program and uses the results of the analysis to identify and highlight any program operation that may cause a run-time error. The programmer can then investigate each potential fault site and, using the graphical explanation facilities of MrSpidey, determine if the fault will really happen or whether the corresponding correctness proof is beyond the analysis's capabilities. In practice, MrSpidey has proven to be an effective tool for debugging a variety of programs, including a staged interpreter, a hardware verifier, and portions of Rice's Scheme program development environment.

The following section describes the kinds of program errors that MrSpidey helps identify; the next section shows how MrSpidey presents information concerning these potential errors to the programmer; and section 1.3 describes the underlying analysis that MrSpidey uses to infer that information. The last section provides an overview of the rest of the thesis.

## 1.1 Reliability and Safety

A reliable program does not mis-apply program operations. Addition always operates on numbers, not strings. Concatenation works with strings, not numbers. To avoid the abuse of program operations, most languages impose a restrictive type system, which forbids the syntactic formation of certain faulty program phrases. However, type systems are too coarse to solve the general problem, which includes indexing an array outside of its proper bounds, division by zero, dereferencing of null pointers, and jumping to non-function pointers. These problems are beyond the capabilities of standard type systems, and different languages deal with such run-time errors in different ways.

*Unsafe languages* like C [31] ignore the problem and leave it to the programmer to insert checks where appropriate. As a result C programs are notoriously prone to inexplicable crashes [33], or, worse, inexplicable, correct-looking results. In contrast, *safe languages* such as SML [34], Scheme [5], and Java [22] equip program operations with appropriate run-time checks where necessary. These checks guarantee that a misapplied program operation immediately raises an error signal, instead of returning a random bit-pattern. Although this solution ensures that programs don't produce random results, it is unsatisfactory because certain errors are not signaled until run-time.

A better approach is to verify the pre-conditions of each program operation statically. If we can prove that that a particular program operation is only applied to appropriate arguments, we say that that operation is *safe*. In the absence of such a proof, we have to consider that operation *unsafe*, since it may be applied to inappropriate arguments. Verifying the safety of program operations requires inferring invariants describing the sets of values that may occur at different points in the program. These invariants could be inferred manually, but for large systems this quickly becomes an extremely difficult and error-prone task. What is needed instead is a static analysis tool that assists the programmer in inferring invariants and reasoning about the safety of program operations. We call this kind of tool a *static debugger*.

Past research on static debuggers mainly focused on the synthesis of the invariants [3, 8]. However, the presentation and, in particular, the explanation of these invariants were neglected. We believe that synthesizing invariants is not enough. Instead, a programmer must be able to inspect the invariants *and* browse their underlying proof. Then, if some invariant contains an unexpected element, the programmer can determine whether the element results from a flaw in the program or approximations introduced by the proof system.

## 1.2 MrSpidey

MrSpidey is a static debugger for Scheme that allows the programmer to browse program invariants and their derivations. On demand, MrSpidey statically analyzes the program and uses the resulting invariants to identify and highlight program operations that are not provably safe (according to MrSpidey's underlying proof system). Associated pop-up menus provide access to additional information, including:

- a value-set invariant for each expression and variable, and
- a graphical explanation for each invariant.

The programmer can investigate each unsafe operation and determine whether (a) the fault will really happen, or (b) the corresponding correctness proof is beyond MrSpidey's capabilities. MrSpidey's graphical explanation facilities make it easy to distinguish these two situations.

As an illustration of MrSpidey's explanatory capabilities, consider the program `sum.ss` shown in figure 1.1. The program defines a function that sums binary trees. A tree is either a leaf node, represented as a number, or an internal node, represented as a cons-cell containing two subtrees.

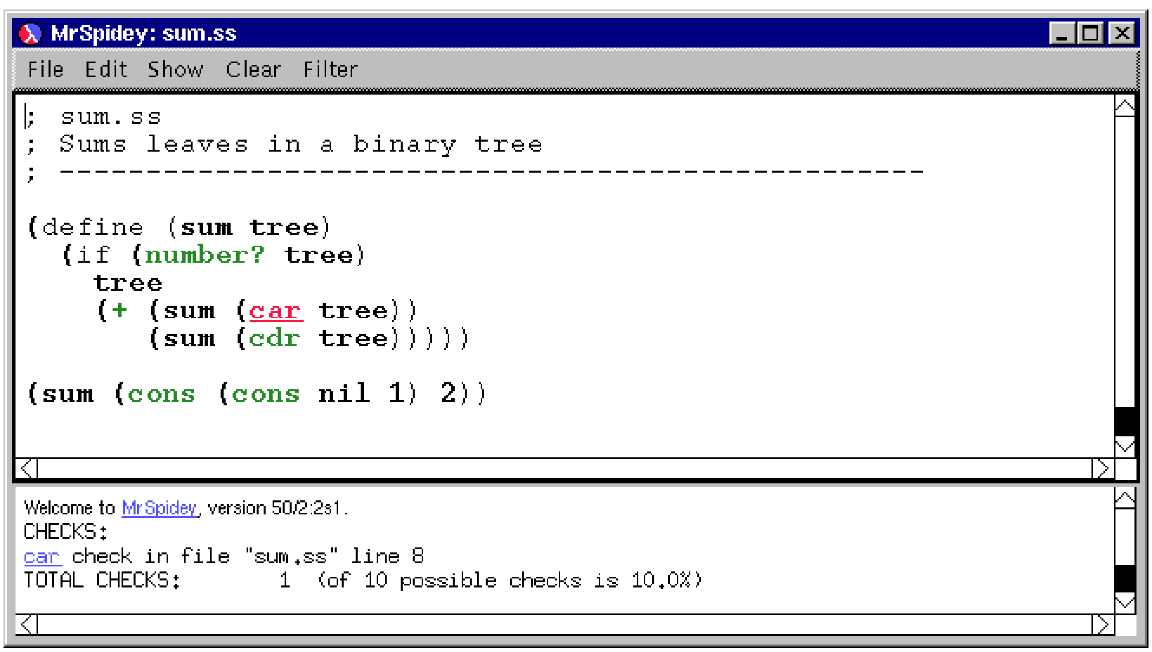


**Figure 1.1** The program `sum.ss`

The mark-ups in in figure 1.1 indicate that the operation `+` is provably safe, since it is not underlined. Conversely, `car`, which is underlined, is unsafe. That is, MrSpidey's proof system is unable to verify that this operation is only applied to appropriate arguments. Indeed, the proof system yields an invariant for `car`'s argument `tree` that contains values that are inappropriate arguments for `car`. Clicking on the argument variable `tree` displays that invariant, as shown in figure 1.2. The invariant shows that `tree` may be bound to the value `nil`, which is outside the domain of `car`. Using MrSpidey's explanatory facilities to investigate the source of this erroneous value results in the display shown in figure 1.3. The displayed arrows show how `nil` originated in the argument to `sum`. Since the argument is not actually a correct binary tree, we have identified the original source of the problem, and can now fix the program by providing a correct tree as the argument to `sum`.

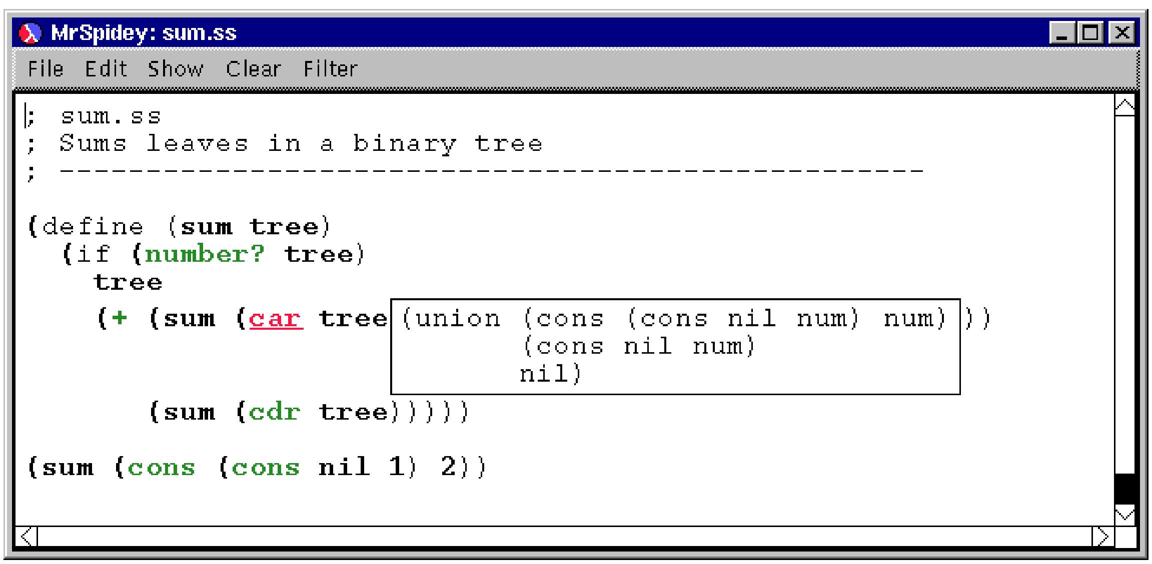


**Figure 1.2** The value set invariant for `tree`

Although `sum.ss` is a trivial program, MrSpidey also works reasonably well on substantially larger programs. The key problem in developing a static debugger for large programs is developing an underlying proof system that can infer accurate invariants for such programs. MrSpidey is based on a componential,* or component-wise, analysis that can effective handle programs of up to tens of thousands of lines of code.

*__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)

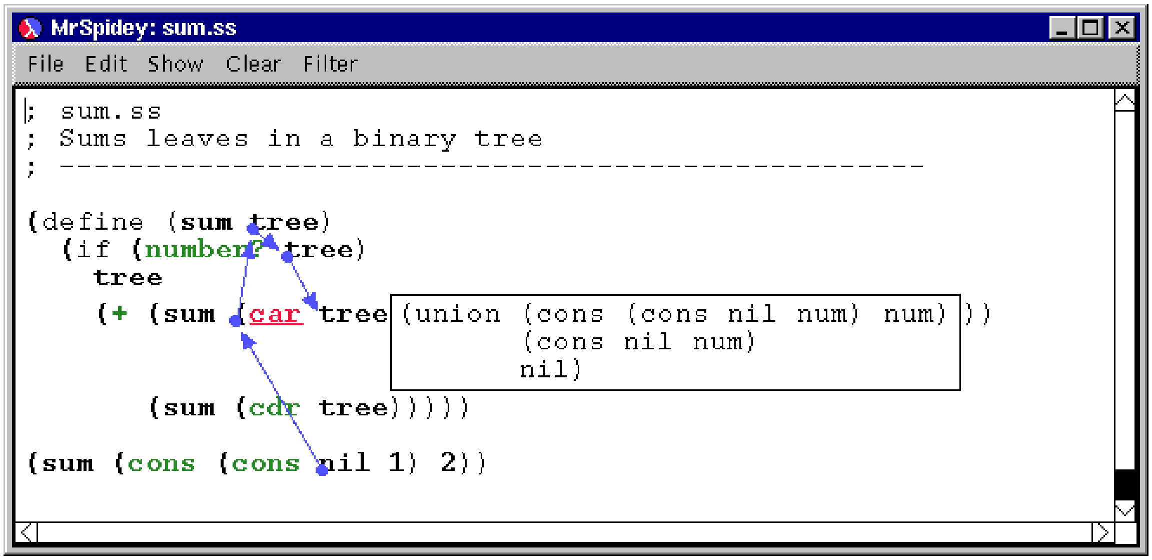


**Figure 1.3** The flow of `nil`

## 1.3 Componential Set-Based Analysis

The key technology underlying MrSpidey is componential set-based analysis. This analysis is a derivative of Heintze's set-based analysis of ML programs [24]. We begin with a discussion of Heintze's analysis and its limitations, and then describe how we can develop a componential variant of Heintze's analysis that overcomes those limitations.

### 1.3.1 Set-Based Analysis

Heintze's original set-based analysis is a constraint-based, whole-program analysis for functional and object-oriented programming languages. It consists of two co-mingled phases: a *specification* phase and a *solution* phase. During the specification phase, the analysis tool derives *constraints* on the sets of values that program expressions may assume. These constraints describe the data-flow relationships amongst the expressions in 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 expression in the program.

We initially used set-based analysis as the underlying proof system for MrSpidey, for the following three reasons:

- Set-based analysis produces accurate program invariants for Scheme-like languages, even in the presence of complex control-flow and data-flow patterns.

- Set-based analysis is intuitive to the programmer. The analysis interprets program operations as naïve set-theoretic operations on sets of run-time values, and propagates these sets of values along the program's data-flow paths, in a manner that is easily understood by the programmer.

- By exploiting the set-based analysis algorithm, we can provide a supporting explanation for each invariant produced by the analysis.

In practice, set-based analysis has proven highly effective for debugging pedagogic programming, which includes programs of several hundreds to 2,000 lines of code. However, it becomes less useful for debugging larger programs for three 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 becomes 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 call site. This duplication is expensive because the size of the constraint system is at best linear, and possibly quadratic, in the size of the function.

- MrSpidey presents value set invariants to the programmer that are computed from the information produced by the analysis, and as the constraints get larger these invariants become extremely verbose.

A closer look at these three 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 develop an algorithm for reducing the size of a constraint system without affecting the solution space that it denotes, we could simplify constraint systems at intermediate stages and thus reduce the analysis time. In particular, simplifying the constraint system for each module would substantially reduce the cost of solving the combined system of constraints for a modularized program; similarly, simplifying the constraint system of a polymorphic function would substantially reduce the cost of duplicating that constraint system at each polymorphic reference.

### 1.3.2 Constraint Simplification

The simplification of constraint systems raises both interesting theoretical and practical questions. On the theoretical side, we need to ensure that simplification preserves the observable behavior, or solution space, of a constraint system. In this dissertation, we provide a complete proof-theoretic and algorithmic characterization of the observable behavior of constraint systems. In the course of this development, we establish a close connection between the observable equivalence of constraint systems and the equivalence of regular tree grammars (RTGs).† Exploiting this connection, we develop a complete algorithm for deciding the observable equivalence of constraint systems. Unfortunately, the problem is PSPACE-hard, and hence the algorithm takes exponential time.

Fortunately, a minimized constraint system is not necessary for our purposes. The practical question concerns finding algorithms for simplifying, though not necessarily minimizing, constraint systems. To answer this question, we exploit the correspondence between the minimization problems for RTGs and constraint systems to adapt a variety of algorithms for simplifying RTGs to the problem of simplifying constraint systems. The resulting constraint simplification algorithms yield an order of magnitude reduction in the size of constraint systems for typical program expressions.

### 1.3.3 Componential Set-Based Analysis

We exploit these simplification algorithms to develop a componential set-based analysis algorithms. The componential analysis first processes each program component independently, deriving and simplifying the constraint system for that component and saves the simplified system in a *constraint file*, for use in later runs of the analysis. This step can be skipped for each program component that has not changed since the last run of the analysis, since the component's constraint file can be used instead.

The analysis then combines and solves these simplified constraint systems, thus propagating data-flow information between the constraint systems for the various program components. The resulting solution yields invariants characterizing the run-time behavior of the entire program.

†A number of researchers, including Reynolds [39], Jones and Muchnick [29], Heintze [24], Aiken [2], and Cousot and Cousot [7] previously exploited the relationship between RTGs and the *least solution* of a constraint system. We present an additional result, namely a connection between RTGs and the observable behavior (*i.e.*, the *entire solution space*) of constraint systems.

This component-wise approach yields an analysis that can handle significantly larger programs than previous analyses of comparable accuracy. The new analysis also 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.

## 1.4 Thesis Overview

This dissertation establishes that the results of set-based analysis can be effectively used to statically debug real programs. In the course of this investigation we present a new derivation of set-based analysis that is significantly more extensible that Heintze's original derivation; we show how to present the invariants and derivations of the analysis to the programmer in a natural and easily-accessible manner; and we evaluate the effectiveness of the resulting static debugging system on a variety of programs, including a staged interpreter and a hardware verifier.

Since real programs also tend to be large, we develop a componential variant of set-based analysis that performs significantly better on large programs. The development of this analysis requires an investigation of the observable equivalence of constraint systems. We provide a complete proof-theoretic and algorithmic characterization of the observable behavior of constraint systems and establish a close connection between the observable equivalence of constraint systems and the equivalence of RTGs. We then exploit this connection to develop a complete algorithm for deciding the observable equivalence of constraint systems, and to adapt a variety of algorithms for simplifying RTGs to the problem of simplifying constraint systems.

The next chapter describes set-based analysis in the context of an idealized $\lambda$-calculus-like language. This analysis is extended in chapter 3 to a number of additional features typically found in real program languages, such as pairs (or other compound data structures), first-class continuations, assignable variables, mutable data structures, modules and classes.

Chapter 4 explains how MrSpidey uses the analysis results to compute useful static debugging information, and chapter 5 describes how this information is presented to the programmer in a natural and intuitive manner.

Chapters 6 and 7 introduce componential set-based analysis. Chapter 6 describes our investigation of the observable equivalence of constraint systems and the development of the constraint simplification algorithms. Chapter 7 shows how to exploit these simplification algorithms in a componential analysis. These two chapters also contain experimental results describing the behavior of both the simplification algorithms and the componential analysis.

Chapter 8 evaluates the effectiveness of MrSpidey on a variety of programs, including a staged interpreter and a hardware verifier. Chapter 9 describes related work on static debugging, constraint simplification and program analysis. The final chapter examines of some of the problems and limitations of MrSpidey, and suggests directions for future work.

The thesis includes six appendices. The first three appendices present proofs for chapters 2, 4, and 6, respectively. The fourth appendix is a reference manual for MrSpidey. The fifth appendix describes a number of details concerning the implementation of MrSpidey, and the sixth appendix contains a list of the mathematical notations and symbols used in this dissertation.

# Chapter 2

# Set-Based Analysis

MrSpidey's underlying analysis is derived from Heintze's set-based analysis of ML programs [23, 24]. The analysis is a constraint-based, whole-program analysis for functional and object-oriented programming languages. It consists of two co-mingled phases: a *specification* phase and a *solution* phase.* During the specification phase, the analysis tool derives *constraints* on the sets of values that program expressions may assume. These constraints describe the data flow relationships amongst the expressions in 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.

In this chapter, we provide a formal description of set-based analysis for a idealized, $\lambda$-calculus-like language with constants. The following chapter extends the analysis to realistic language features including pairs, first-class continuations, assignable variables, mutable data structures, modules and classes. Since it is difficult to extend Heintze's development of set-based analysis beyond a functional core language, we develop Heintze's ideas using an alternative formulation and semantics. Specifically, whereas Heintze's development is based on "natural" semantics, which cannot easily accommodate non-local control operators and destructive data structure manipulations, our alternative formulation is based on an extensible reduction semantics. This change of framework also simplifies the derivation of the constraint simplification algorithms.

Section 2.1 introduces our idealized, $\lambda$-calculus-like language. Section 2.2 introduces the constraint language, and section 2.3 defines the meaning of those constraints. Section 2.4 describes how we derive appropriate constraints for a program, and section 2.5 proves the correctness of the derived constraints. Section 2.6 shows

---

*Cousot and Cousot showed that set-based analysis can alternatively be formulated as an abstract interpretation computed by chaotic iteration [7].

how to solve the derived constraints to yield information about the program's run-time behavior. Section 2.7 contains an outline implementation of set-based analysis.

## 2.1 The Source Language

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

### 2.1.1 Syntax

Syntax:

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

**Figure 2.1** The source language $\Lambda$

Expressions in the language are either variables, values, applications, **let**-expressions, or labeled expressions: see figure 2.1. 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 [48]. We use labels to identify those program expressions whose values we wish to predict.

We work with the usual conventions and terminology of the $\lambda_v$-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 $\Lambda^0$ denotes the set of closed terms, also called *programs*.

### 2.1.2 Semantics

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

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

The $\beta_v$ and $\beta_{let}$ rules are the conventional rules for the $\lambda$-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:

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

## 2.2 The Constraint Language

To simplify the later derivation of the constraint simplification algorithms (see chapter 6), we express the constraint language in terms of selectors, instead of the more usual constructors. Specifically, a *set expression* $\tau$ is either a set variable; a constant; or one of the "selector" expressions $\texttt{dom}(\tau)$ or $\texttt{rng}(\tau)$:

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

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 $\{\mathtt{dom}(\alpha) \leq \alpha_1,\ \alpha_2 \leq \mathtt{rng}(\alpha)\}$ instead of the combined constraint $\{(\alpha_1 \rightarrow \alpha_2) \leq \alpha\}$.

The meta-variables $\alpha, \beta, \gamma$ 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 $\tau_1 \leq \tau_2$ relating two set expressions.

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

We sometimes enclose constraints inside square brackets for clarity: $[\tau_1 \leq \tau_2]$. 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 $\mathsf{S}$. 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:

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

## 2.3 The Meaning of Set Constraints

Intuitively, each set expression $\tau$ 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.

### 2.3.1 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 $C \in \mathcal{P}(\mathit{Const})^{\dagger}$ is a set of basic constants and

---

$^{\dagger}\mathcal{P}$ denotes the power-set constructor.

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[‡] with each node labeled with an element of $\mathcal{P}(Const)$. This set can be formally defined as the set of total functions $f : \{\mathtt{dom}, \mathtt{rng}\}^* \longrightarrow \mathcal{P}(Const)$, and the rest of the development can be adapted *mutandis mutatis* [36]. For clarity, we present our results using the more intuitive equational definition instead.

We use the functions *const*, *dom*, and *rng* to extract the respective components of an element in $\mathcal{D}$:

$$\begin{array}{rcl} const : \mathcal{D} & \longrightarrow & \mathcal{P}(Const) \\ const\,(\langle C, D, R\rangle) & = & C \\ \\ dom : \mathcal{D} & \longrightarrow & \mathcal{D} \\ dom\,(\langle C, D, R\rangle) & = & D \\ \\ rng : \mathcal{D} & \longrightarrow & \mathcal{D} \\ rng\,(\langle C, D, R\rangle) & = & R \end{array}$$

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 $V \ in\ X$:

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

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

[‡]In order to analyze languages with additional data structures, we later extend $\mathcal{D}$ to infinite *n-ary* trees, where $n$ is the number of selectors (*e.g.*. `dom`, `rng`) corresponding to the extended language.

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 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:

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

### 2.3.2 The Semantics of Constraints

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

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

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 $\rho$, we define the following unique extension $\rho^*$ that specifies a meaning for set expressions:

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

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

A set environment $\rho$ *satisfies* a constraint $\mathsf{C} = [\tau_1 \leq \tau_2]$ (written $\rho \models \mathsf{C}$) if $\rho(\tau_1) \sqsubseteq \rho(\tau_2)$. Similarly, $\rho$ satisfies $\mathsf{S}$, or $\rho$ is a solution of $\mathsf{S}$ (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 $\mathsf{S}$ is:

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

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

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

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 2.3.1.** (*Restricted Entailment, Restricted Observable Equivalence*)

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

∎

## 2.4 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 [36], 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 2.2. Each rule infers a judgment of the form $\Gamma \vdash M : \alpha, \mathcal{S}$, where:

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

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

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

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

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

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

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

**Figure 2.2** Constraint derivation rules.

1. the *derivation context* $\Gamma$ maps the free variables of the expression $M$ to either set variables or *constraint schemas* (see below);

2. $\alpha$ 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 $\alpha$.

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

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

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 $\beta$ describes the value set of $x$, and $\alpha$ 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, 42]. The set variable $\beta$ names the result of the value; the simple constraint system $\mathcal{S}$ describes the data-flow relationships of the value, using $\beta$; 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 $\Gamma$ 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 $\Gamma$. 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 $\alpha$. 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.

## 2.5 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 $\rho$ of $\mathcal{S}$ correctly approximates the value sets of labeled expressions in $P$. That is, if $\rho$ 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 [13], following Wright and Felleisen [47] and Palsberg [35].

**Main Lemma 2.5.1** *(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 (2.5.2) 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:

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

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 2.5.6. 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 2.5.2** *(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 2.5.3 and the Replacement Lemma 2.5.4. ∎

**Lemma 2.5.3** *(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 2.5.4** *(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 [25:page 181]. ∎

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

**Lemma 2.5.5** *(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:** 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 2.5.6** *(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 $V = (\lambda^t x.M)$. ∎

## 2.6 Solving Set Constraints

Every simple constraint system admits the trivial solution $\rho^{\top_s}$ defined by:

$$\rho^{\top_s}(\alpha) \quad = \quad \top_s$$

where $\top_s$ is recursively defined as:

$$\top_s \quad = \quad \langle \mathit{Const}, \top_s, \top_s \rangle$$

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 2.6.1** 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 2.2, 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 $\rho_1$ describes this (empty) set of run-time values of $x$ more accurately than either $\rho_2$ or $\rho_3$. 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:

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

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 [24].

**Lemma 2.6.2** *(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 2.6.2, it makes sense to define *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 2.6.3.** ($sba : \Lambda^0 \longrightarrow (Label \longrightarrow \mathcal{P}(Value))$) If $\emptyset \vdash P : \alpha, \mathcal{S}$ is a most general derivation, then

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

∎

By Lemma 2.5.1, $sba(P)$ correctly characterizes the possible value sets for each labeled expression.

**Theorem 2.6.4** If $P \longmapsto^* \mathcal{E}[\ V^l\ ]$ then $V \in sba(P)(l)$.

**Proof:** Follows from Lemma 2.5.1. ∎

### 2.6.1 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 $\Theta$ described in figure 2.3. Intuitively, these rules infer all the data-flow paths in the program, which are described by constraints of the form $\beta \leq \gamma$ (for $\beta, \gamma \in SetVar$), 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 $\Theta$, and use $\Theta(\mathcal{S})$ to denote the closure of $\mathcal{S}$ under $\Theta$, *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.

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 2.6.5** 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. ∎

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

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

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

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

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

**Figure 2.3** The rules $\Theta = \{s_1, \ldots, s_5\}$

## 2.7 An Implementation of Set-Based Analysis

We conclude this chapter with an outline implementation in MzScheme [17] of the analysis described above. The implementation is presented in figures 2.4, 2.5 and 2.6.

### 2.7.1 Representation of constraint systems

Since the most computationally-intensive part of the analysis is closing the constraint system under the rules $\Theta$, we choose a representation for constraint systems that simplifies this task. If we take a closer look at $\Theta$, we see that each rule in $\Theta$ combines a lower and upper bound for some set variable $\alpha$ into a single constraint, where the lower bound on $\alpha$ must be of the form:

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

and the upper bound on $\alpha$ must be of the form:

$$\alpha \leq \beta \qquad \beta \leq \mathtt{dom}(\alpha) \qquad \mathtt{rng}(\alpha) \leq \beta$$

We represent the above lower and upper bounds for each set variable $\alpha$ separately. That is, a constraint system is represented as two mappings: one mapping from set variables to their lower bounds, and a second mapping from set variables to their upper bounds. Each kind of lower bound on $\alpha$ corresponds to a structure in the implementation:

$$\begin{array}{rcll} c & \leq & \alpha & \qquad\qquad \texttt{c-leq-this} \\ \beta & \leq & \texttt{rng}(\alpha) & \qquad\qquad \texttt{x-leq-rng-this} \\ \texttt{dom}(\alpha) & \leq & \beta & \qquad\qquad \texttt{dom-this-leq-x} \end{array}$$

In a similar fashion, each kind of upper bound on $\alpha$ also corresponds to a structure in the implementation:

$$\begin{array}{rcll} \alpha & \leq & \beta & \qquad\qquad \texttt{this-leq-x} \\ \beta & \leq & \texttt{dom}(\alpha) & \qquad\qquad \texttt{x-leq-dom-this} \\ \texttt{rng}(\alpha) & \leq & \beta & \qquad\qquad \texttt{rng-this-leq-x} \end{array}$$

### 2.7.2 Closing constraint systems

Set-based analysis consists of a specification phase and a solution phase. In this implementation, these two phases are co-mingled in that the constraint derivation algorithm keeps the derived constraint system closed under $\Theta$. That is, whenever the constraint derivation algorithm adds a new constraint to the current constraint system, any consequences of the new constraint under $\Theta$ are also added to the constraint system.

Constraint systems are extended via the functions *add-lower-bound+close!* and *add-upper-bound+close!*. The function *combine!* checks if a new consequent can be inferred from a lower and upper bound for a set variable.

### 2.7.3 Deriving constraints

The function *derive* is a straightforward implementation of the constraint derivation rules described in figure 2.2. This function takes as arguments a constraint derivation context *gamma*, an expression $M$, and a constraint system $S$. The function then extends $S$ with additional constraints and returns a set variable denoting the result value set of $M$. For simplicity, the implementation does not support **let**-expressions or constraint schemas, but these are straightforward to add.

```
;; ---- Main analysis function
(define (sba M)
  (let ([S (create-constraint-system)])
    (derive (make-empty-derivation-context) M S)
    S))

;; ---- Abstract syntax structures
(define-struct Var (x))
(define-struct Const (b))
(define-struct Lam (t x M))
(define-struct App (fn arg))
(define-struct Labeled (M l))

;; ---- Deriving constraints
(define (derive gamma M S)
  (let ([alpha (gen-set-var)])
    (match M
      [($ Var x)
        (add-upper-bound+close! S (lookup gamma x) (make-this-leq-x alpha))]
      [($ Const b)
        (add-lower-bound+close! S alpha (make-c-leq-this b))]
      [($ Labeled M l)
        (let ([beta (derive gamma M S)])
          (add-upper-bound+close! S beta (make-this-leq-x l))
          (add-upper-bound+close! S beta (make-this-leq-x alpha)))]
      [($ Lam t x N)
        (let* ( [alpha1 (gen-set-var)]
                [alpha2 (derive (extend gamma x alpha1) N S)])
          (add-lower-bound+close! S alpha (make-c-leq-this t))
          (add-lower-bound+close! S alpha (make-dom-this-leq-x alpha1))
          (add-lower-bound+close! S alpha (make-x-leq-rng-this alpha2)))]
      [($ App fn arg)
        (let ( [beta1 (derive gamma fn S)]
               [beta2 (derive gamma arg S)])
          (add-upper-bound+close! S beta1 (make-x-leq-dom-this beta2))
          (add-upper-bound+close! S beta1 (make-rng-this-leq-x alpha)))])
    alpha))
```

**Figure 2.4** Deriving constraints

```
;; —— Structures for constraints
(define-struct c-leq-this (c))
(define-struct dom-this-leq-x (x))
(define-struct x-leq-rng-this (x))

(define-struct this-leq-x (x))
(define-struct x-leq-dom-this (x))
(define-struct rng-this-leq-x (x))

;; —— Extending and re-closing constraint system
;; For clarity, we have not abstracted over the following two functions
(define (add-lower-bound+close! S alpha low-bound)
  (unless (has-lower-bound? S alpha low-bound)
    (add-lower-bound! S alpha low-bound)
    (for-each
      (lambda (up-bound) (combine! S low-bound up-bound))
      (upper-bounds S alpha))))

(define (add-upper-bound+close! S alpha up-bound)
  (unless (has-upper-bound? S alpha up-bound)
    (add-upper-bound! S alpha up-bound)
    (for-each
      (lambda (low-bound) (combine! S low-bound up-bound))
      (lower-bounds S alpha))))

;; —— Adding consequents of two constraints
(define (combine! S low-bound up-bound)
  (match (list low-bound up-bound)
    ;; The following cases applies rules (s1) through (s5), respectively
    [(($ c-leq-this _) ($ this-leq-x gamma))
     (add-lower-bound+close! S gamma low-bound)]
    [(($ dom-this-leq-x _) ($ this-leq-x gamma))
     (add-lower-bound+close! S gamma low-bound)]
    [(($ x-leq-rng-this _) ($ this-leq-x gamma))
     (add-lower-bound+close! S gamma low-bound)]
    [(($ x-leq-rng-this alpha) ($ rng-this-leq-x gamma) )
     (add-upper-bound+close! S alpha (make-this-leq-x gamma))]
    [(($ dom-this-leq-x gamma) ($ x-leq-dom-this alpha) )
     (add-upper-bound+close! S alpha (make-this-leq-x gamma))]
    [_ (void)]))
```

**Figure 2.5** Representing, extending and closing a constraint system

```
;; —— Functions for manipulating constraint systems

(define (gen-set-var) ...)
;; Creates a fresh set variable

(define (create-constraint-system) ...)
;; Creates an empty constraint system

(define (add-lower-bound! S alpha low-bound) ...)
;; Extends the constraint system S with an additional lower bound for alpha

(define (has-lower-bound? S alpha low-bound) ...)
;; Checks if the constraint system S already has the given lower bound for alpha

(define (lower-bounds S alpha) ...)
;; Returns a list of lower bounds for alpha in the constraint system S

(define (add-upper-bound! S alpha up-bound) ...)
(define (has-upper-bound? S alpha up-bound) ...)
(define (upper-bounds S alpha) ...)
;; Ditto for upper bounds

;; —— Functions for manipulating derivation contexts

(define (make-empty-derivation-context) ...)
;; creates an empty derivation context

(define (extend gamma x alpha) ...)
;; extends the derivation context gamma to map x to alpha

(define (lookup gamma x) ...)
;; looks up the binding for x in the derivation context gamma
```

**Figure 2.6** Manipulating constraint systems and derivation contexts

# Chapter 3

# Extending Set-Based Analysis

Realistic programming languages provide a variety of additional facilities on top of the idealized core language $\Lambda$. These facilities typically include pairs (or other compound data structures such as C's `struct`s or Pascal's `record`s), assignable variables, mutable data structures, and possibly objects, modules and non-local control operators. Since MrSpidey is designed to assist in the development of realistic Scheme programs, we need to extend the underlying set-based analysis to encompass these additional features of practical programming languages. This extension also suggests how MrSpidey can be adapted to other safe languages such as Java.

## 3.1 Additional Selectors

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

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

$$\begin{array}{rcl} Sel^+ & = & \{\mathtt{rng}\} \\ Sel^- & = & \{\mathtt{dom}\} \end{array}$$

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

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

and a *simple constraint* is of the form:

$$\begin{array}{rcll} \mathcal{C} & \in & \mathit{SimpleCon} & = \quad c \leq \beta \\ & & & \mid \alpha \leq \beta \\ & & & \mid \alpha \leq \texttt{sel}^-(\beta) \\ & & & \mid \texttt{sel}^+(\alpha) \leq \beta \\ & & & \mid \texttt{sel}^-(\alpha) \leq \beta \\ & & & \mid \alpha \leq \texttt{sel}^+(\beta) \end{array}$$

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

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

This reformulation simplifies the process of extending the analysis to cope with additional programming constructs. The remainder of the derivation from chapter 2 can be adapted to the modified formulation, *mutandis mutatis*. In particular, the appropriate inference rules $\Theta$ for the modified formulation are described in figure 3.1.

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

$$\frac{\alpha \leq \texttt{sel}^+(\beta) \qquad \beta \leq \gamma}{\alpha \leq \texttt{sel}^+(\gamma)} \qquad (s_2)$$

$$\frac{\texttt{sel}^-(\beta) \leq \alpha \qquad \beta \leq \gamma}{\texttt{sel}^-(\gamma) \leq \alpha} \qquad (s_3)$$

$$\frac{\alpha \leq \texttt{sel}^+(\beta) \qquad \texttt{sel}^+(\beta) \leq \gamma}{\alpha \leq \gamma} \qquad (s_4)$$

$$\frac{\alpha \leq \texttt{sel}^-(\beta) \qquad \texttt{sel}^-(\beta) \leq \gamma}{\alpha \leq \gamma} \qquad (s_5)$$

**Figure 3.1** The adapted rules $\Theta = \{s_1, \ldots, s_5\}$

## 3.2 Analysis of Pairs

Let $\Lambda^p$ be the following extension of $\Lambda$ with immutable pairs.

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

$$\begin{array}{rcrcl} V & \in & \mathit{Value} & = & \ldots \\ &&&& \mid (\textbf{cons}\ V\ V) \end{array}$$

### 3.2.1 Semantics

The additional syntactic forms have their usual Scheme semantics, which we formalize via two additional notions of reduction:

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

To allow the evaluation of sub-expressions inside the syntactic forms **cons**, **car** and **cdr**, we extend the notion of evaluation contexts as follows:

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

The standard reduction relation $\longmapsto$ and the evaluator *eval* for the extended language $\Lambda^p$ can now be defined in the usual manner, following section 2.1.2.

### 3.2.2 Analysis

The analysis of the extended language $\Lambda^p$ requires two additional monotonic selectors `car` and `cdr`:

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

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

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

The constraint derivation rules for the new syntactic forms are described in figure 3.2. The rule (*cons*) records both the pairs tag and the possible values for each component of the pair. The rules (*car*) and (*cdr*) extract the appropriate component from the set variable for the argument expression.

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

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

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

**Figure 3.2** Constraint derivation rules for pairs

The soundness proof for set-based analysis is easily adapted to the extended language. The Replacement (2.5.4), Value Typing (2.5.6), and Soundness (2.5.1) Lemmas are straightforward to adapt, and we extend the Subject Reduction for $\longrightarrow$ Lemma (2.5.3) as follows.

**Lemma 3.2.1** *(Subject Reduction for $\longrightarrow$ on $\Lambda^p$).* 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 (*car*). Then:

$$\begin{array}{rcl} M_1 & = & (\mathbf{car}\ (\mathbf{cons}\ V_1\ V_2)) \\ M_2 & = & V_1 \end{array}$$

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

$$\dfrac{\dfrac{\Gamma \vdash V_i : \alpha_i, \mathcal{S}'_i}{\Gamma \vdash (\textbf{cons}\ V_1\ V_2) : \beta, \mathcal{S}'_1 \cup \mathcal{S}'_2 \cup \mathcal{S}}\ (cons)}{\Gamma \vdash M_1 : \gamma, \mathcal{S}'_1 \cup \mathcal{S}'_2 \cup \mathcal{S} \cup \mathcal{S}'}\ (car)$$

where:

$$\begin{array}{rcl} \mathcal{S} & = & \{t \leq \beta, \alpha_1 \leq \texttt{car}(\beta), \alpha_2 \leq \texttt{cdr}(\beta)\} \\ \mathcal{S}' & = & \{\texttt{car}(\beta) \leq \gamma\} \end{array}$$

Hence $\mathcal{S} \cup \mathcal{S}' \models \{\alpha_1 \leq \gamma\}$, and by the Flow Lemma 2.5.5, $\Gamma \vdash V_1 : \gamma, \mathcal{S}_2$ where $\mathcal{S}'_1 \cup \mathcal{S} \cup \mathcal{S}' \models \mathcal{S}_2$, as required.

- The case for the reduction rule ($cdr$) is similar.

∎

The set-based analysis function $sba$ for the extended language $\Lambda^p$ is defined following definition 2.6.3. As in section 2.6, we can compute $sba(P)$ from the closure of $\mathcal{S}$ under $\Theta$:

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

## 3.3 Analysis of First-Class Continuations

Consider the following language $\Lambda^{cc}$, which extends $\Lambda^p$ with first-class continuations in addition to functional core language and pairs:

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

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

### 3.3.1 Semantics

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

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

The evaluator for the extended language is defined in the usual manner, following section 2.1.2.

### 3.3.2 Analysis

Figure 3.3 introduces the additional derivation rules for **abort** and **callcc** expressions. An **abort** expression never returns, so the derivation rule (*abort*) introduces a fresh type variable for these expressions. The least solution (under $\sqsubseteq_s$) for this type variable is $\perp_s$, denoting the empty set of values.

The rule (*callcc*) introduces a new type variable $\delta$ to denote the captured continuation. The rule records that:

1. the type variable $\delta$ contains the tag $t$ from the **callcc** expression;
2. $\delta$ is the argument to the function (denoted by $\alpha$) that is returned by $M$;
3. the result value this function becomes the result of the **callcc** expression;
4. argument values to $\delta$ are also returned as results of the **callcc** expression.

In addition, the rule adds the "dummy" constraint $\gamma \leq \texttt{rng}(\delta)$. This dummy constraint is required in order that the constraint derivation rules satisfy the subject reduction lemma. That is, the (*callcc*) reduction rule produces a contractum containing the syntactic term $(\lambda^t x.(\textbf{abort}\ \mathcal{E}[\ x\ ]))$, which is not present in the (*callcc*)-redex. Applying the constraint derivation rules to this contractum yields a number of constraints, including the constraint $\gamma \leq \texttt{rng}(\delta)$, where $\gamma$ describes the value set for $(\textbf{abort}\ \mathcal{E}[\ x\ ])$, and $\delta$ describes the value set for the $\lambda$-expression. The subject reduction lemma requires that this constraint is entailed by the constraint system for the (*callcc*)-redex. In order to satisfy this requirement, we include that constraint in the redex's constraint system.

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

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

**Figure 3.3** Constraint derivation rules for first-class continuations

The Replacement (2.5.4), Value Typing (2.5.6), and Soundness (2.5.1) Lemmas are easily adapted to the extended language. We extend the Subject Reduction for $\longmapsto$ Lemma (2.5.2) to account for the additional standard reduction rules as follows.

**Lemma 3.3.1** *(Subject Reduction for $\longmapsto$ on $\Lambda^{cc}$).* If $\emptyset \vdash M_1 : \alpha, \mathcal{S}_1$ and $M_1 \longmapsto M_2$, then $\emptyset \vdash M_2 : \beta, \mathcal{S}_2$ such that $\mathcal{S}_1 \models \mathcal{S}_2$.

**Proof:** The proof proceeds by case analysis according to the relation $M_1 \longmapsto M_2$. Since the cases for the $(\beta_v)$ and $(unlabel)$ rules are the same as before, we only consider the cases for the additional rules $(abort)$ and $(callcc)$.

- Suppose $M_1 \longmapsto M_2$ via $(abort)$. Then:

  $$M_1 \;=\; \mathcal{E}[\ (\mathbf{abort}\ M_2)\ ]$$

  The derivation on $M_1$ must include a sub-derivation concluding $\Gamma \vdash M_2 : \beta, \mathcal{S}_2$ where $\mathcal{S}_2 \subseteq \mathcal{S}_1$. Since $M_1$ is closed, so is $M_2$, and hence $\emptyset \vdash M_2 : \beta, \mathcal{S}_2$.

- Suppose $M_1 \longmapsto M_2$ via $(callcc)$. Then:

  $$\begin{array}{rcl} M_1 & = & \mathcal{E}[\ (\mathbf{callcc}^t\ M)\ ] \\ M_2 & = & \mathcal{E}[\ (M\ (\lambda^t x.(\mathbf{abort}\ \mathcal{E}[\ x\ ])))\ ] \end{array}$$

  The derivation on $M_1$ must include a sub-derivation concluding:

  $$\frac{\Gamma \vdash M : \alpha_M, \mathcal{S}_M}{\Gamma \vdash (\mathbf{callcc}^t\ M) : \beta, \mathcal{S}_M \cup \mathcal{S}_C} \quad (callcc)$$

where:

$$\mathcal{S}_C = \left\{ \begin{array}{rcl} t & \leq & \delta \\ \delta & \leq & \mathtt{dom}(\alpha_M) \\ \mathtt{rng}(\alpha_M) & \leq & \beta \\ \mathtt{dom}(\delta) & \leq & \beta \\ \gamma & \leq & \mathtt{rng}(\delta) \end{array} \right\}$$

Let $\mathcal{S}_{\mathcal{E}}$ be the additional constraints generated by the derivation on $M_1$ due to the context $\mathcal{E}$ surrounding $(\mathbf{callcc}^t\ V)$. Then $\mathcal{S}_1 = \mathcal{S}_{\mathcal{E}} \cup \mathcal{S}_M \cup \mathcal{S}_C$, and the following derivation holds:

$$\cfrac{\cfrac{\cfrac{\Gamma \cup \{x : \beta\} \vdash x : \beta, \{\beta \leq \beta\}}{\Gamma \cup \{x : \beta\} \vdash \mathcal{E}[\ x\ ] : \alpha, \mathcal{S}_{\mathcal{E}}}}{\Gamma \cup \{x : \beta\} \vdash (\mathbf{abort}\ \mathcal{E}[\ x\ ]) : \gamma, \mathcal{S}_{\mathcal{E}}}\ (abort)}{\Gamma \vdash (\lambda^t x.(\mathbf{abort}\ \mathcal{E}[\ x\ ])) : \delta, \mathcal{S}_L}\ (abs)$$

where:

$$\mathcal{S}_L = \mathcal{S}_{\mathcal{E}} \cup \{t \leq \delta, \mathtt{dom}(\delta) \leq \beta, \gamma \leq \mathtt{rng}(\delta)\}$$

Hence:

$$\Gamma \vdash (M\ (\lambda^t x.(\mathbf{abort}\ \mathcal{E}[\ x\ ]))) : \beta, \mathcal{S}_2'$$

where:

$$\mathcal{S}_2' = \mathcal{S}_L \cup \mathcal{S}_M \cup \{\delta \leq \mathtt{dom}(\alpha_M), \mathtt{rng}(\alpha_M) \leq \beta\}$$

Therefore, by the Replacement Lemma 2.5.4:

$$\Gamma \vdash M_2 : \beta', \mathcal{S}_2$$

where:

$$\mathcal{S}_2 = \mathcal{S}_{\mathcal{E}} \cup \mathcal{S}_2'$$

Since $\mathcal{S}_1$ contains $\mathcal{S}_2$, $\mathcal{S}_1 \models \mathcal{S}_2$, and the lemma holds for this case.

∎

The set-based analysis function *sba* for the extended language $\Lambda^{cc}$ is defined as in definition 2.6.3, and can be computed in the usual fashion based on the closure of the derived constraint system under $\Theta$.

## 3.4 Analysis of Assignable Variables

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

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

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

### 3.4.1 Semantics

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

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

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

To allow the evaluation of sub-expressions inside the **set!** form, we extend the notion of evaluation contexts as follows:

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

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

$$\begin{array}{lll} (\textbf{letrec}\ (H)\ \mathcal{E}[\ (\textbf{letrec}\ (D^*)\ M)\ ]) & & \\ \quad \longmapsto & (\textbf{letrec}\ (H \cup D^*)\ \mathcal{E}[\ M\ ]) & (\mathit{letrec}) \\ & \text{if } \mathit{dom}(H) \cap \mathit{dom}(D^*) = \emptyset & \end{array}$$

$$(\textbf{letrec}\ (H)\ \mathcal{E}[\ z\ ]) \quad \longmapsto \quad (\textbf{letrec}\ (H)\ \mathcal{E}[\ V\ ]) \quad \text{if } H(z) = V \qquad (\mathit{ref})$$

$$\begin{array}{lll} (\textbf{letrec}\ (H \cup (\textbf{define}\ z\ V))\ \mathcal{E}[\ (\textbf{set!}\ z\ V')\ ]) & & \\ \quad \longmapsto & (\textbf{letrec}\ (H \cup (\textbf{define}\ z\ V'))\ \mathcal{E}[\ V'\ ]) & (\mathit{set!}) \end{array}$$

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

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

### 3.4.2 Analysis

The analysis of the extended language $\Lambda^!$ is based on the additional constraint derivation rules described in figure 3.4. The rule (*letrec*) extends the derivation context $\Gamma$ to map each assignable variable $z_i$ to a fresh set variable $\alpha_i$ and generates constraints for both the defined values and the **letrec**-body using the extended derivation context. The rule (*set!*) simply propagates all possible assigned values into the value set for the assigned variable. A constraint derivation context now maps variables to either set variables or constraint schemas, as before, and now also maps assignable variables to set variables.

Adapting the soundness proof for the extended analysis is straightforward. The Replacement (2.5.4), Value Typing (2.5.6), and Soundness (2.5.1) Lemmas are easily adapted to the extended language. It is straightforward to extend the Subject Reduction for $\longmapsto$ Lemma (2.5.2) with cases for the additional reduction rules. The set-based analysis function *sba* for the extended language $\Lambda^!$ can be defined and computed in the usual fashion.

## 3.5 Analysis of Assignable Boxes

Next, we extend our sample language with assignable boxes, where are somewhat more difficult to analyze than assignable variables. Let $\Lambda^b$ be the following extension

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

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

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

**Figure 3.4** Constraint derivation rules for assignable variables

of $\Lambda^!$:

$$\begin{array}{llllll} M & \in & \Lambda^b & = & \ldots & \text{(Expressions)} \\ & & & & \mid (\textbf{box}\ M) & \\ & & & & \mid (\textbf{unbox}\ M) & \\ & & & & \mid (\textbf{set-box!}\ M\ M) & \\ & & & & & \\ V & \in & \mathit{Value} & = & \ldots & \text{(Values)} \\ & & & & \mid \langle \texttt{box}\ z \rangle & \end{array}$$

### 3.5.1 Semantics

The additional syntactic forms have their usual Scheme semantics. The value $\langle \texttt{box}\ z \rangle$ denotes a box containing the value bound to the assignable variable (or *location*) $z$. We formalize this semantics by extending the standard reduction relation as follows:

$$\begin{array}{lll} (\textbf{letrec}\ (H)\ \mathcal{E}[\ (\textbf{box}\ V)\ ]) & & \\ \quad \longmapsto\ (\textbf{letrec}\ (H \cup (\textbf{define}\ z\ V))\ \mathcal{E}[\ \langle \texttt{box}\ z \rangle\ ]) & & (\mathit{box}) \\ \quad\quad \text{if } z \notin \mathit{dom}(H) & & \end{array}$$

$$\begin{array}{lll} (\textbf{letrec}\ (H)\ \mathcal{E}[\ (\textbf{unbox}\ \langle \texttt{box}\ z \rangle)\ ]) & & \\ \quad \longmapsto\ (\textbf{letrec}\ (H)\ \mathcal{E}[\ V\ ]) \quad \text{if } H(z) = V & & (\mathit{unbox}) \end{array}$$

$$\begin{array}{lll} (\textbf{letrec}\ (H \cup (\textbf{define}\ z\ V))\ \mathcal{E}[\ (\textbf{set-box!}\ \langle \texttt{box}\ z \rangle\ V')\ ]) & & \\ \quad \longmapsto\ (\textbf{letrec}\ (H \cup (\textbf{define}\ z\ V'))\ \mathcal{E}[\ V'\ ]) & & (\mathit{set\text{-}box!}) \end{array}$$

The rule (*box*) allocates a new assignable variable, or *location*, $z$, that is not already bound in the heap, binds this location to the boxed value in the heap, and returns a new kind of value, $\langle \texttt{box}\ z \rangle$ as the result of the **box** expression. The rules (*unbox*) and (*set-box!*) are analogous to the rules (*ref*) and (*set!*) described in the previous section, except that the new rules operate on box values of the form $\langle \texttt{box}\ z \rangle$, instead of directly on assignable variables.

To allow the evaluation of sub-expressions inside the syntactic forms **box**, **unbox** and **set-box!**, we extend the notion of evaluation contexts as follows:

$$\mathcal{E} = \ldots \mid (\textbf{box}\ \mathcal{E}) \mid (\textbf{unbox}\ \mathcal{E}) \mid (\textbf{set-box!}\ \mathcal{E}\ M) \mid (\textbf{set-box!}\ V\ \mathcal{E})$$

The semantics of the extended language is defined via the partial function *eval* on programs, in the same manner as in the previous section.

### 3.5.2 Analysis

The assignable boxes of the extended language $\Lambda^b$ are first-class values. Unlike assignable variables, these boxes can be passed around between various parts of the analyzed program. This flexibility makes assignable boxes significantly more difficult to analyze than assignable variables. [*]

As a first attempt, we could try to analyze boxes by extending the constraint language with an single additional selector, $\texttt{box}$, such that $\texttt{box}(\tau)$ describes the set of values contained in the boxes denoted by $\tau$. Unfortunately, this approach does not work very well. To illustrate the problem with this approach, consider the data-flow path in the analyzed program represented by the constraint $\alpha \leq \beta$. In order to ensure that boxed values flow forward along this data-flow path in the appropriate manner, we need to require that $\texttt{box}(\alpha) \leq \texttt{box}(\beta)$, which implies that $\texttt{box}$ is monotonic. But now suppose that a **set-box!** operation is performed on the value set described by

[*]We could alternatively define the semantics of the additional syntactic forms in $\Lambda^b$ by macro-expanding them into the following $\Lambda^!$ expressions:

$$\begin{array}{rcl}
(\textbf{box}\ M) & \longrightarrow & (\textbf{letrec}\ ([z\ M])\ (\textbf{cons}\ (\lambda d.\, z)\ (\lambda x.\, (\textbf{begin}\ (\textbf{set!}\ z\ x)\ x)))) \\
(\textbf{unbox}\ M) & \longrightarrow & ((\textbf{car}\ M)\ 1) \\
(\textbf{set-box!}\ M_1\ M_2) & \longrightarrow & ((\textbf{cdr}\ M_1)\ M_2)
\end{array}$$

The analysis of these macro-expanded forms yields results are analogous to that produced by the analysis described in this section for the original forms.

$\beta$, thus increasing the set of possible boxed values at $\beta$. To insure that this effect is reflected in the boxed values of $\alpha$, we need to require that $\texttt{box}(\beta) \leq \texttt{box}(\alpha)$, which implies `box` is also *anti-monotonic*. Hence the constraint $\alpha \leq \beta$ can hold if and only if $\texttt{box}(\alpha) = \texttt{box}(\beta)$. In practice, this means that any two boxed values that reach the same program point are *unified*. Unfortunately, our experience with Soft Scheme [45] indicates that the results of unification-based analyses are often unintuitive and difficult to explain.

In order to produce a more intuitive and accurate analysis, we introduce the notion of *split boxes*[†], which uses the following two additional selectors for the analysis of boxes:

1. a monotonic selector $\texttt{box}^+$ that models how boxed values flow forward along data-flow paths;
$$Sel^+ \;=\; \{\ldots, \texttt{box}^+\}$$
and

2. an anti-monotonic selector $\texttt{box}^-$ that models how the values assigned to boxes flow *backward* along data-flow paths.
$$Sel^- \;=\; \{\ldots, \texttt{box}^-\}$$

Appropriate constraint derivation rules for the new syntactic forms, based on the two selectors $\texttt{box}^+$ and $\texttt{box}^-$, are described in figure 3.5. The rule (*box*) introduces a set variable $\delta$ to describe the set of values possibly contained in the newly constructed box. The set expression $\texttt{box}^+(\beta)$ describes values that are contained in the box, and $\texttt{box}^-(\beta)$ describes values that are assigned to the box. Both the initial value of the box ($\alpha$) and any values assigned to the box ($\texttt{box}^-(\beta)$) must be contained in $\delta$, and $\delta$ is contained in the set of values ($\texttt{box}^+(\beta)$) that can be extracted from the box. The rules (*unbox*) and (*set-box!*) are straightforward.

Proving the soundness of the new analysis is straightforward. The Replacement (2.5.4), Value Typing (2.5.6), and Soundness (2.5.1) Lemmas are easily adapted to the extended language, and the Subject Reduction for $\longmapsto$ Lemma (2.5.2) can be extended with cases for the additional reduction rules. The set-based analysis function *sba* for $\Lambda^b$ is defined and computed in the usual fashion.

---

[†]This idea was also independently discovered by Scott Smith and Valerie Trifinov, and by Francois Pottier (private communications, ICFP'96), and earlier by Reynolds [40].

$$\frac{\Gamma \vdash M : \alpha, \mathcal{S}}{\Gamma \vdash (\textbf{box}\ M) : \beta, \mathcal{S} \cup \left\{ \begin{array}{rcl} \alpha & \leq & \delta \\ \texttt{box}^-(\beta) & \leq & \delta \\ \delta & \leq & \texttt{box}^+(\beta) \end{array} \right\}} \quad (box)$$

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

$$\frac{\Gamma \vdash M_i : \alpha_i, \mathcal{S}_i}{\Gamma \vdash (\textbf{set-box!}\ M_1\ M_2) : \beta, \mathcal{S}_1 \cup \mathcal{S}_2 \cup \{\alpha_2 \leq \texttt{box}^-(\alpha_1), \alpha_2 \leq \beta\}} \quad (set\text{-}box!)$$

**Figure 3.5** Constraint derivation rules for boxes

## 3.6 Analysis of Units

Realistic programming languages typically have a module system to help organize large software systems. In MzScheme [17], the module system is based on units [19]. In this section we consider the analysis of a simplified version of MzScheme's unit system. The language $\Lambda^u$ extends $\Lambda^!$ as follows:

$$\begin{array}{lclcll} M & \in & \Lambda^u & = & \ldots & \text{(Expressions)} \\ & & & & \mid (\textbf{link}^t\ M\ M) & \\ & & & & \mid (\textbf{invoke}\ M\ z) & \\ & & & & & \\ V & \in & \mathit{Value} & = & \ldots & \text{(Values)} \\ & & & & \mid (\textbf{unit}^t\ w_1\ D^*\ M\ w_2) & \end{array}$$

A unit expression $(\textbf{unit}^t\ w_1\ D^*\ M\ w_2)$ consists of:

1. an identifying tag $t$;
2. a single imported assignable variable $w_1$;
3. a series of definitions $D^*$ binding assignable variables to syntactic values;
4. an expression $M$ that may mutate those assignable variables, if required; and

5. an exported assignable variable $w_2$.

Assignments to the imported variable are syntactically disallowed.

A **link** clause contains an identifying tag and two sub-expressions. These sub-expressions should evaluate to units, which the **link** clause then combines into a single (compound) unit. Both units and link clauses have identifying tags so that MrSpidey can reconstruct the textual source of various unit values from the results of the analysis. An **invoke** clause consists of an expression that should evaluate to a unit, and an assignable variable that becomes the imported variable of the unit, after which the definitions in the unit are evaluated in order.

### 3.6.1 Semantics

We specify the semantics of the new syntactic forms via the reduction rules:

$$\begin{array}{ll} (\mathbf{link}^t\ (\mathbf{unit}^{t_1}\ w_1\ D_1^*\ M_1\ w_2)\ (\mathbf{unit}^{t_2}\ w_2\ D_2^*\ M_2\ w_3)) & \\ \qquad\qquad \longrightarrow\ (\mathbf{unit}^t\ w_1\ D_1^*.D_2^*\ (\texttt{begin}\ M_1\ M_2)\ w_3) & (\beta_{link}) \end{array}$$

$$\begin{array}{ll} (\mathbf{invoke}\ (\mathbf{unit}^t\ z\ D^*\ M\ w)\ z) & \\ \qquad\qquad \longrightarrow\ (\mathbf{letrec}\ (D^*)\ (\texttt{begin}\ M\ w)) & (invoke) \end{array}$$

The rule $(\beta_{link})$ for **link** expressions combines two units to produce a compound unit. The imported variable of the compound unit is the imported variable of the first sub-unit, the exported variable of the first sub-unit is connected to the imported variable of the second sub-unit, and the exported variable of the second sub-unit becomes the exported variable of the compound unit. The syntax $(\texttt{begin}\ M_1\ M_2)$ is shorthand for $((\lambda^t d.M_2)\ M_1)$, where $d \notin FV[M_2]$. The rule $(invoke)$ converts a unit value into a **letrec**-expression, which is then evaluated as described in the previous section.

We augment the definition of evaluation contexts to permit evaluation of the sub-expressions in **link** and **invoke** forms:

$$\mathcal{E} = \ldots\ |\ (\mathbf{link}^t\ \mathcal{E}\ M)\ |\ (\mathbf{link}^t\ V\ \mathcal{E})\ |\ (\mathbf{invoke}\ \mathcal{E}\ z)$$

This semantics can easily be extended to handle mutually-recursive units by allowing units to have two imported variables, such that a **link** clause connects the exported variable of the first sub-unit to one of the imported variables of the second unit, and vice-versa.

### 3.6.2 Analysis

The analysis of the extended language $\Lambda^u$ requires additional selectors `ui` and `ue` that extract the imported and exported value sets of a unit:

$$\begin{array}{rcl} Sel^{+} & = & \{\ldots, \mathtt{ue}\} \\ Sel^{-} & = & \{\ldots, \mathtt{ui}\} \end{array}$$

The additional constraint derivation rules for the new syntactic forms are described in figure 3.6. The rule (*unit*) for unit expressions introduces a number of additional set variables, including a set variable $\gamma_1$ for the imported variable $w_1$; set variables $\alpha_1, \ldots, \alpha_n$ for the defined variables $z_1, \ldots, z_n$; set variables $\beta_1, \ldots, \beta_n$ for the initial values $V_1, \ldots, V_n$; and a set variable $\gamma$ denoting the result of the unit expression. The rule (*unit*) then ensures that

- the set variable $\gamma$ includes the identifying tag $t$ of the unit;
- the imported value set of the unit is contained in the value set of the imported variable;
- the value set of the exported variable $w_2$ is contained in the exported value set of the unit; and
- the set variable $\beta_i$ (describing $V_i$) is contained in $\alpha_i$ (describing $z_i$).

The rule (*link*) ensures that the set variable $\alpha$ for the link expression includes the identifying tag $t$; that the imported value set of the resulting compound unit is contained in the imported value set of the first unit; that the exported value set of the first unit is contained in the imported value set of the second unit; and that the exported value set of the second unit is contained in the exported value set of the resulting compound unit.

The rule (*invoke*) ensures that the value set of the argument variable is contained in the imported value set of the unit; and that the exported value set of the unit becomes the result value set of the invoke expression.

We extend the relation $V\ in\ X$ describing the set of run-time values represented by an element $X \in \mathcal{D}$ (relative to a given program) with the following additional clause for unit values:

$$(\mathbf{unit}^t\ x\ D^*\ M\ y)\ in\ \langle C, \ldots \rangle \quad \text{iff} \quad t \in C$$

$$\frac{\begin{array}{c}\Gamma' = \Gamma \cup \{w_1 : \gamma_1, z_1 : \alpha_1, \ldots, z_n : \alpha_n\} \\ \Gamma' \vdash V_i : \beta_i, \mathcal{S}_i \quad \text{for } i = 1, \ldots, n \\ \Gamma' \vdash M : \gamma_M, \mathcal{S}_M\end{array}}{\begin{array}{l}\Gamma \vdash (\textbf{unit}^t\ w_1\ (\textbf{define}\ z_1\ V_1) \ldots (\textbf{define}\ z_n\ V_n)\ M\ w_2) \\ \quad : \gamma, \mathcal{S}_M \cup \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n \cup \left\{\begin{array}{rcll} t & \leq & \gamma & \\ \texttt{ui}(\gamma) & \leq & \gamma_1 & \\ \Gamma(w_2) & \leq & \texttt{ue}(\gamma) & \\ \beta_i & \leq & \alpha_i & 1 \leq i \leq n \end{array}\right\}\end{array}} \quad (\mathit{unit})$$

$$\frac{\Gamma \vdash M_i : \beta_i, \mathcal{S}_i \quad \text{for } i = 1, 2}{\Gamma \vdash (\textbf{link}^t\ M_1\ M_2) : \alpha, \mathcal{S}_1 \cup \mathcal{S}_2 \cup \left\{\begin{array}{rcl} t & \leq & \alpha \\ \texttt{ui}(\alpha) & \leq & \texttt{ui}(\beta_1) \\ \texttt{ue}(\beta_1) & \leq & \texttt{ui}(\beta_2) \\ \texttt{ue}(\beta_2) & \leq & \texttt{ue}(\alpha) \end{array}\right\}} \quad (\mathit{link})$$

$$\frac{\Gamma \vdash M : \beta, \mathcal{S}}{\Gamma \vdash (\textbf{invoke}\ M\ z) : \alpha, \mathcal{S} \cup \{\Gamma(z) \leq \texttt{ui}(\beta),\ \texttt{ue}(\beta) \leq \alpha\}} \quad (\mathit{invoke})$$

**Figure 3.6** Constraint derivation rules for units

The soundness proof for set-based analysis can be adapted to the extended language.

## 3.7 Analysis of Classes

In this section, we outline the extension of set-based analysis to a simple class system. Let $\Lambda^c$ be the following extension of $\Lambda^u$:

$$\begin{array}{rcll} M & \in & \Lambda^c = & \ldots \\ & & & \mid (\texttt{class}\ N\ (z_1 \ldots z_k)\ [z_{k+1}\ V_{k+1}] \ldots [z_n\ V_n]) \\ & & & \mid (\texttt{make-obj}\ M) \\ & & & \mid M.z \end{array}$$

In the class expression:

$$(\texttt{class}\ N\ (z_1 \ldots z_k)\ [z_{k+1}\ V_{k+1}] \ldots [z_n\ V_n])$$

$N$ describes the super-class; $z_1, \ldots, z_k$ are instance variables inherited from that super-class; and $z_{k+1}, \ldots, z_n$ are additional instance variables in the new class. These new instance variables are initialized to the values $V_{k+1}, \ldots, V_n$, respectively.

The special form (`make-obj` $M$) creates new objects of the class described by $M$. Although the `make-obj` form does not allow arguments to be passed to the class, such arguments can be simulated by extending the class with additional instance variables that are initialized to the appropriate argument values. The special form $M.z$ extracts the value bound to the instance variable $z$ in the object described by $M$.

### 3.7.1 Semantics

We do not provide a formal semantics for this class system. Instead, we refer the interested reader to a paper on the formal semantics of a related class system [20].

### 3.7.2 Analysis

The analysis of programs in $\Lambda^c$ requires that we add a number of additional selectors to the constraint language. For describing the behavior of class values, we introduce an additional, monotonic selector, $\texttt{cl-obj}(\cdot)$, such that $\texttt{cl-obj}(\tau)$ describes objects produced by classes in $\tau$.

We also introduce additional selectors for describing the behavior of objects. Objects are similar to the assignable boxes of section 3.5, except that whereas boxes have a single assignable field, an object may have multiple assignable instance variables. The analysis of boxes required two additional selectors, $\texttt{box}^+$ and $\texttt{box}^-$, which model how boxed values flow both forward and backward along data-flow paths. In a similar manner, for each instance variable $z$, we introduce two additional selectors, $\texttt{ivar}_z^+$ and $\texttt{ivar}_z^-$. These additional selectors model how values for an instance variable $z$ of an object flow both forward and backward along the data-flow paths of the object.

The additional constraint derivation rules for classes are described in figure 3.7. The rule (*class*) is the most complex, and introduces a number of new set variables, which have the following meaning:

- the set variable $\alpha_s$ describes the super-class;
- the set variables $\beta_1, \ldots, \beta_n$ describe the value set of the corresponding instance variables $z_1, \ldots, z_n$;

$$\frac{\begin{array}{c}\Gamma \vdash N : \alpha_s, \mathcal{S}_s \\ \Gamma[z_1 : \beta_1, \ldots, z_n : \beta_n] \vdash V_i : \gamma_i, \mathcal{S}_i \quad \text{for } i = k+1, \ldots, n\end{array}}{\begin{array}{l}\Gamma \vdash (\texttt{class}\ N\ (z_1 \ldots z_k)\ [z_{k+1}\ V_{k+1}] \ldots [z_n\ V_n]) \\ \quad : \alpha, \mathcal{S}_s \cup \mathcal{S}_{k+1} \cup \cdots \cup \mathcal{S}_n \cup \left\{ \begin{array}{rcl} \texttt{cl-obj}(\alpha_s) & \leq & \alpha_o \\ \texttt{ivar}^+_{z_i}(\alpha_o) & \leq & \beta_i \\ \beta_i & \leq & \texttt{ivar}^-_{z_i}(\alpha_o) \\ \gamma_i & \leq & \beta_i \\ \alpha_o & \leq & \texttt{cl-obj}(\alpha) \end{array} \right\}\end{array}} \quad (\textit{class})$$

$$\frac{\Gamma \vdash M : \beta, \mathcal{S}_s}{\Gamma \vdash (\texttt{make-obj}\ M) : \alpha, \mathcal{S} \cup \{\texttt{cl-obj}(\beta) \leq \alpha\}} \quad (\textit{make-obj})$$

$$\frac{\Gamma \vdash M : \beta, \mathcal{S}_s}{\Gamma \vdash M.z : \alpha, \mathcal{S} \cup \{\texttt{ivar}^+_z(\beta) \leq \alpha\}} \quad (\textit{ivar})$$

**Figure 3.7** Constraint derivation rules for classes

- the set variables $\gamma_{k+1}, \ldots, \gamma_n$ describe the value set of the corresponding initialization expressions $V_{k+1}, \ldots, V_n$; and
- the set variable $\alpha_o$ describes objects of the new class.

The rule (*class*) first derives constraints for the super-class expression $N$, and then derives constraints for each initialization expression $V_i$ in an appropriate derivation context. It then ensures that:

- that the current object ($\alpha_o$) contains instance variable values defined in the super class ($\alpha_s$), via the constraint $\texttt{cl-obj}(\alpha_s) \leq \alpha_o$;
- that the values in $\beta_i$ for the instance variable $z_i$ reflect the values from $\alpha_o$;
- that the initial value $V_1$, described by $\gamma_i$, is contained the value set described by $\beta_i$; and
- that the resulting class, described by $\alpha$, correctly refers to objects of the new class, which are described by $\alpha_o$.

The rule (*make-obj*) extracts the `cl-obj`$(\cdot)$ component from the set variable describing the class; and the rule (*ivar*) extracts the instance variable component from the set variable describing the object.

We do not provide a soundness proof for the extended analysis, but we conjecture the Subject Reduction for $\longmapsto$ Lemma (2.5.2) could be appropriately extended to the new analysis, based on suitable rewriting semantics for the extended language.

# Chapter 4

# Using Set-Based Analysis for Static Debugging

The constraint system inferred by set-based analysis provides information about the behavior of the analyzed program. MrSpidey uses this constraint system to:

- compute a compact and intuitive value set invariant, or *type*, for each expression in the program; and
- identify potentially *unsafe* program operations.

This chapter describes how this information is inferred from the results of the analysis.

## 4.1 The Type Language

The simple constraint system computed by set-based analysis is not suitable for presentation to the programmer. Simple constraint systems are well-suited for efficiently representing and manipulating information about a program's run-time behavior, but are difficult for programmers to interpret, for two reasons.

- The constraints are expressed in terms of "selectors", instead of the conventional "constructors", which are familiar to most programmers.
- Simple constraint systems use set variables extensively as *indirection* pointers. For example, the constraint $\alpha \leq \mathtt{rng}(\mathtt{rng}(\beta))$ is represented as the simple constraint system $\{\alpha \leq \mathtt{rng}(\gamma), \gamma \leq \mathtt{rng}(\beta)\}$ with an indirection through $\gamma$. This extensive use of indirections makes constraint systems difficult to interpret.

To avoid the need for programmers to interpret constraint systems, MrSpidey uses a conventional type language for communication with programmers. We present the type language in the context of the programming language $\Lambda^p$ described in section 3.2. This type language includes constants, set variables, the empty type $\perp_s$, functions, pairs, unions and recursive type definitions: see figure 4.1. The scope rules for recursive type definitions are analogous to Scheme's **letrec** construct. The notions of free and bound variables in types are defined in the usual manner, and we use $\mathit{Type}^0$ to denote the set of closed types.

$$
\begin{array}{rcll}
\omega & \in & \mathit{Type} & = \quad c \\
 & & & \mid \alpha \\
 & & & \mid \bot_s \\
 & & & \mid (\omega \ \rightarrow^T \ \omega) \\
 & & & \mid (\mathtt{cons}\ \omega\ \omega) \\
 & & & \mid \omega \cup \omega \\
 & & & \mid (\mathtt{rec}\ ([\alpha_1\ \omega_1]\ \ldots\ [\alpha_n\ \omega_n])\ \omega) \\
 & & & \\
T & \subseteq & \mathit{FnTag} &
\end{array}
$$

**Figure 4.1** The type language *Type*.

The semantics of closed types is specified via the meaning function $\mathcal{M}$ that maps each closed type into $\mathcal{D}$. The semantics of open types is specified via a related function $\mathcal{M}_\rho$, where the additional set environment $\rho$ specifies the meaning of the free variables in the type. These two functions are defined in figure 4.2. The semantics of the types $c$, $\alpha$, and $\bot_s$ are straightforward. The type $(\omega_1 \ \rightarrow^T \ \omega_2)$ denotes an element of $\mathcal{D}$ with function tags $T$ and whose `dom` and `rng` components are described by $\omega_1$ and $\omega_2$, respectively. The type $(\mathtt{cons}\ \omega_1\ \omega_2)$ denotes an element of $\mathcal{D}$ with whose `car` and `cdr` components are denoted by $\omega_1$ and $\omega_2$, respectively. The type $\omega_1 \cup \omega_2$ denotes the union (or least upper bound) of $\omega_1$ and $\omega_2$. The recursive type $(\mathtt{rec}\ ([\alpha_1\ \omega_1]\ \ldots\ [\alpha_n\ \omega_n])\ \omega)$ denotes the element of $\mathcal{D}$ described by $\omega$, where each $\alpha_i$ is bound to the element denoted by $\omega_i$.

## 4.2 Computing Type Information

MrSpidey infers a type for each program expression from the constraint system $\mathcal{S}$ computed by set-based analysis. The least solution $\mathit{LeastSoln}(\mathcal{S})$ of this constraint system is a set environment mapping set variables to elements of the domain $\mathcal{D}$. For each labeled expression $M^l$ in the program, $\mathit{LeastSoln}(\mathcal{S})(l)$ describes the set of possible run-time values of $M^l$. To communicate this value set to the programmer,

$$
\begin{array}{rcl}
\mathcal{M}[\cdot] : \mathit{Type}^0 & \longrightarrow & \mathcal{D} \\
\mathcal{M}[\omega] & = & \mathcal{M}_\rho[\omega] \\
 & & \text{(the choice of } \rho \text{ does not affect the defn.)} \\
\\
\mathcal{M}.[\cdot] : \mathit{SetEnv} \times \mathit{Type} & \longrightarrow & \mathcal{D} \\
\mathcal{M}_\rho[c] & = & \langle \{c\}, \bot_s, \bot_s, \bot_s, \bot_s \rangle \\
\mathcal{M}_\rho[\alpha] & = & \rho(\alpha) \\
\mathcal{M}_\rho[\bot_s] & = & \bot_s \\
\mathcal{M}_\rho[(\omega_1 \;\rightarrow^T\; \omega_2)] & = & \langle T, \rho(\omega_1), \rho(\omega_2), \bot_s, \bot_s \rangle \\
\mathcal{M}_\rho[(\texttt{cons}\ \omega_1\ \omega_2)] & = & \langle \{\texttt{pair}\}, \bot_s, \bot_s, \mathcal{M}_\rho[\omega_1], \mathcal{M}_\rho[\omega_2] \rangle \\
\mathcal{M}_\rho[\omega_1 \cup \omega_2] & = & \mathcal{M}_\rho[\omega_1] \sqcup_s \mathcal{M}_\rho[\omega_2] \\
\mathcal{M}_\rho[(\texttt{rec}\ ([\alpha_1\ \omega_1]\ \ldots\ [\alpha_n\ \omega_n])\ \omega)] & = & \mathcal{M}_{\rho'}[\omega] \\
 & & \text{where } \rho' = \mathit{lfp}(\lambda\rho.\ \rho[\alpha_i \mapsto \mathcal{M}_\rho[\omega_i]]) \\
 & & \text{under the ordering } \sqsubseteq_s
\end{array}
$$

**Figure 4.2** The semantics of types.

MrSpidey computes a type invariant describing $\mathit{LeastSoln}(\mathcal{S})(l)$ from the program's constraint system. This type invariant is computed in three steps.

### Step 1

MrSpidey first uses one of the simplification algorithms from section 6.4 to simplify the constraint system $\mathcal{S}$ with respect to the external variable $l$, while preserving $\mathit{LeastSoln}(\mathcal{S})(l)$.

### Step 2

MrSpidey then computes a type from the simplified constraint system according to the function $\mathit{MkType} : \mathit{SimpleConSystem} \times \mathit{SetVar} \longrightarrow \mathit{Type}^0$:

$$
\begin{array}{rcl}
\mathit{MkType}(\mathcal{S}, \alpha) & = & (\texttt{rec}\ ([\alpha_1\ \omega_1]\ \ldots\ [\alpha_n\ \omega_n])\ \alpha) \\
 & & \text{where } \{\alpha_1, \ldots, \alpha_n\} = \mathit{SetVar}(\mathcal{S}) \\
 & & \text{and } \omega_i = \mathit{MkType}'(\mathcal{S}, \alpha_i)
\end{array}
$$

The auxiliary function$\mathit{MkType}'$ maps a constraint system $\mathcal{S}$ and a set variable $\alpha_i$ to an open type that may refer to other set variables from $\mathcal{S}$. The function $\mathit{MkType}$

then links the open types for each set variable in $\mathcal{S}$ together in a single `rec` type, thus producing an appropriate closed type.

The auxiliary function $\mathit{MkType}' : \mathit{SimpleConSystem} \times \mathit{SetVar} \longrightarrow \mathit{Type}$ is defined as:

$$
\begin{array}{lrcl}
 & \mathit{MkType}'(\mathcal{S}, \alpha) & = & \omega_b \cup \omega_p \cup \omega_f \\
 & & & \\
\text{where} & \omega_b & = & \{b \in \mathit{BasicConst} \mid \mathcal{S} \vdash_\Theta b \leq \alpha\} \\
 & \omega_p & = & (\texttt{cons}\ \omega_{\texttt{car}}\ \omega_{\texttt{cdr}}) \\
 & \omega_{\texttt{car}} & = & \{\beta \mid \mathcal{S} \vdash_\Theta \beta \leq \texttt{car}(\alpha)\} \\
 & \omega_{\texttt{cdr}} & = & \{\beta \mid \mathcal{S} \vdash_\Theta \beta \leq \texttt{cdr}(\alpha)\} \\
 & \omega_f & = & (\omega_{\texttt{dom}} \ \to^T \ \omega_{\texttt{rng}}) \\
 & T & = & \{t \in \mathit{FnTag} \mid \mathcal{S} \vdash_\Theta t \leq \alpha\} \\
 & \omega_{\texttt{dom}} & = & \{\beta \mid \mathcal{S} \vdash_\Theta \alpha_i \leq^* \delta, \beta \leq \texttt{dom}(\delta)\} \\
, & \omega_{\texttt{rng}} & = & \{\beta \mid \mathcal{S} \vdash_\Theta \beta \leq \texttt{rng}(\alpha)\}
\end{array}
$$

The auxiliary function $\mathit{MkType}'$ returns a union of three types. The first type $\omega_b$ describes the basic constants in $\mathit{LeastSoln}(\mathcal{S})(\alpha)$; the second type $\omega_p$ describes pairs in $\mathit{LeastSoln}(\mathcal{S})(\alpha)$; and the last type $\omega_f$ describes the function tags and argument and results sets in $\mathit{LeastSoln}(\mathcal{S})(\alpha)$. In the definition above, we use a set of types to denote the corresponding union type, *i.e.*, the set of types $\{\omega_1, \omega_2, \omega_3\}$ denotes the union type $\omega_1 \cup \omega_2 \cup \omega_3$, and the empty set denotes the type $\bot_s$.

The type produced by $\mathit{MkType}(\mathcal{S}, \alpha)$ correctly describes $\mathit{LeastSoln}(\mathcal{S})(\alpha)$.

**Lemma 4.2.1** *(Correctness of MkType).* If $\mathcal{S}$ is a simple constraint system then:

$$\mathcal{M}[\mathit{MkType}(\mathcal{S}, \alpha)] = \mathit{LeastSoln}(\mathcal{S})(\alpha)$$

**Proof:** See Appendix B.1. ∎

**Step 3**

Finally, MrSpidey uses the following reductions on *Type* to simplify type expressions.

$$
\begin{array}{rcl}
\omega \cup \bot_s & \longrightarrow & \omega \\
\bot_s \cup \omega & \longrightarrow & \omega \\
(\texttt{cons}\ \bot_s\ \bot_s) & \longrightarrow & \bot_s \\
(\bot_s\ \rightarrow^{\emptyset}\ \bot_s) & \longrightarrow & \bot_s \\
(\texttt{rec}\ ()\ \omega) & \longrightarrow & \omega \\
(\texttt{rec}\ ([\alpha_1\ \omega_1] \ldots [\alpha_n\ \omega_n])\ \omega) & \longrightarrow & \\
\multicolumn{3}{r}{(\texttt{rec}\ ([\alpha_1\ \omega_1] \ldots [\alpha_{i-1}\ \omega_{i-1}][\alpha_{i+1}\ \omega_{i+1}] \ldots [\alpha_n\ \omega_n])\ \omega)[\alpha_i \mapsto \omega_i]} \\
 & & \text{provided } \alpha_i \notin \mathit{SetVar}(\omega_i)
\end{array}
$$

MrSpidey produces a type that is in normal form with respect to these reductions, each of which preserves the meaning of type expressions.

**Lemma 4.2.2** *(Correctness of Type Reductions).* If $\omega_1 \longrightarrow \omega_2$ then $\mathcal{M}_\rho[\omega_1] = \mathcal{M}_\rho[\omega_2]$ for any set environment $\rho$.

**Proof:** By case analysis on the reductions $\omega_1 \longrightarrow \omega_2$. ∎

The type produced by this three-step processes provides a compact description of the corresponding value-set invariant, and is more easily understood by the programmer than the original constraint system produced by set-based analysis.

## 4.3 Identifying Unsafe Operations

Unsafe program operations are a natural starting point in the static debugging process. MrSpidey identifies these unsafe operations using the results of set-based analysis. We consider the problem of identifying unsafe operations in the language $\Lambda^p$. To identify these unsafe operations, MrSpidey considers in turn each operation in the program.

- Suppose the operation is of the form (**car** $M^l$). Then MrSpidey considers the set of constants $\{c \mid [c \leq l] \in \mathcal{S}\}$, where $\mathcal{S}$ is the constraint system for the program closed under $\Theta$. If this constant set only contains the tag `pair`, then, according to the definition of the relation $V\ \mathit{in}\ X$, the value set invariant for $M^l$ contains only pairs. Hence the operation (**car** $M^l$) is *safe*, and can never raise an error during an execution.

Conversely, if the constant set contains additional constants such as function tags or basic constants, then the expression $M^l$ may return values other than pairs, in which case operation (**car** $M^l$) is *unsafe*, since it may raise an error at run-time.

- Operations of the form (**cdr** $M^l$) are dealt with in a similar manner.

- Suppose the operation is an application of the form ($M^l$ $N$). If the constant set $\{c \mid [c \leq l] \in \mathcal{S}\}$ for $M^l$ only contains function tags, then $M^l$ only returns functions, and the application will always succeed.

  Conversely, if the constant set contains additional constants such as basic constants or the tag `pair`, then $M^l$ may return values other than functions. Hence the application *unsafe*, and may raise an error at run-time.

Since **car**, **cdr** and function application are the only operations in the language that may cause errors in $\Lambda^p$, the above analysis will identify all unsafe operations.

Appendix E.5 describes how MrSpidey extends this idea to detect unsafe operations in full Scheme.

# Chapter 5

# User Interface to the Static Debugger

A useful static debugger must fit seamlessly into a programmer's work pattern, and should provide the programmer with useful information in a natural and easily accessible manner. For these reasons, we integrated MrSpidey with DrScheme, Rice's program development environment for Scheme.

On demand, MrSpidey analyzes the current project and uses the resulting constraint systems to infer useful static debugging information. Specifically, MrSpidey:

1. identifies unsafe program operations;
2. derives an appropriate type for each program expression; and
3. provides a graphical explanation of each derived invariant.

MrSpidey presents this information to the programmer using *program mark-ups*. These mark-ups are simple font and color changes that provide information about the analysis results without disturbing the familiar lexical and syntactic structure of the program. Additional information is available via a pop-up menu associated with each marked-up token. The programmer can thus browse through the derived information, and can resume program development based on an improved understanding of the program's execution behavior.

## 5.1 Displaying Unsafe Operations

Unsafe program operations that may raise run-time errors are natural starting points in the static debugging process. MrSpidey highlights these unsafe operations via font and color changes. Any primitive operation that may be applied to inappropriate arguments, thus raising a run-time error, is highlighted in red (or underlined on monochrome screens). Conversely, primitive operations that never raise errors are shown in green. Any function definition that may be applied to an incorrect number of arguments is highlighted by displaying the `lambda` keyword in red (again, underlined on monochrome screens), and any application expression where the function position may yield a non-function is highlighted by displaying the enclosing parentheses in red (or underlined). Figure 5.1 contains an example of each of these three kinds of unsafe operations.

MrSpidey also presents summary information describing each unsafe operation, together with a hyper-link to that operation. The **tab** key moves the focus forward to the next unsafe operation, and the **shift-tab** key moves the focus backward to the previous unsafe operation. By using these facilities, the programmer can easily inspect the unsafe operations in a program.

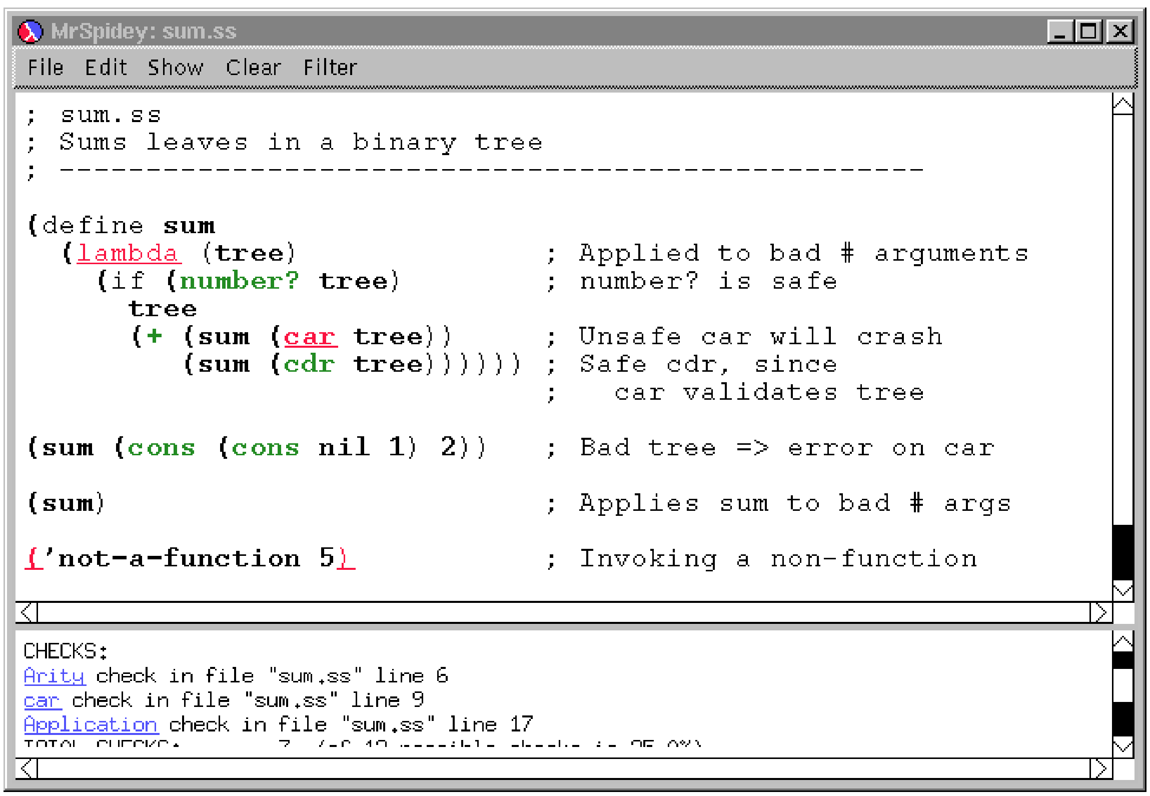


**Figure 5.1** Identifying unsafe operations

## 5.2 Pop-Up Menu

MrSpidey also provides a significant amount of additional information for each expression in the analyzed program. This information cannot be immediately displayed, since it would simply result in "information overload". Instead, MrSpidey provides this information on a demand-driven basis via a pop-up menu associated with each program variable and expression. Figure 5.2 shows the pop-up menu displayed by clicking on the variable *tree*. The information available through the menu is described in the following sections.

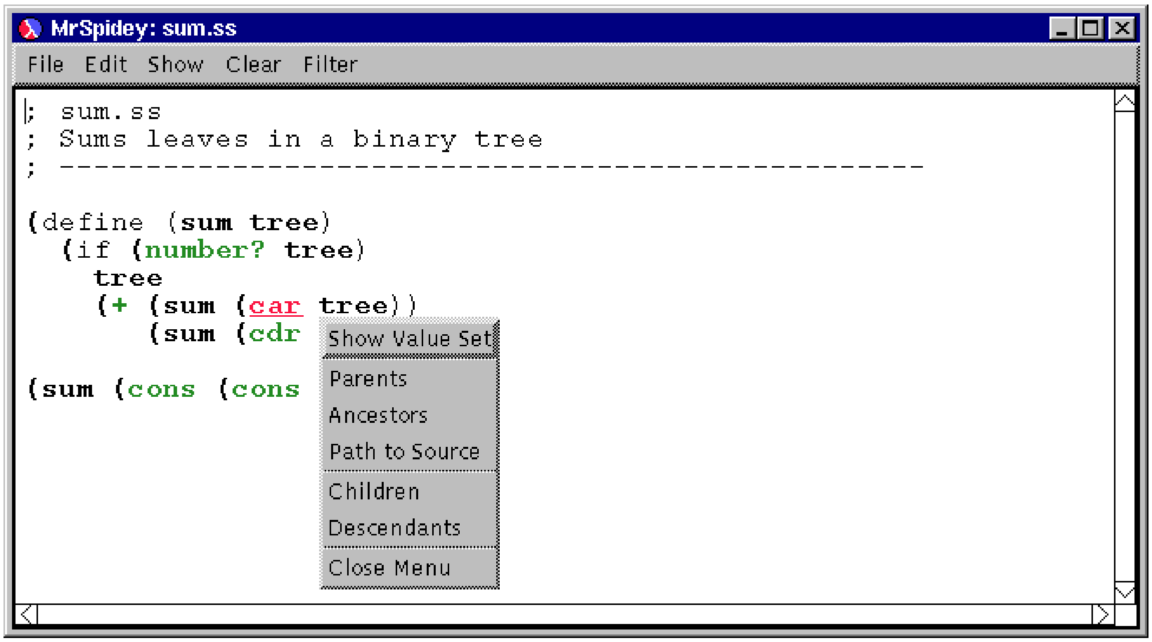


**Figure 5.2** The pop-up menu

## 5.3 Presenting Type Information

MrSpidey lazily computes a type for each program expression, as described in section 4.2. The selection of the Show Value Set option from an expression's menu causes MrSpidey to compute the corresponding type, and to display that type in a box inserted to the right of the expression in the buffer. For example, figure 5.3 shows the inferred type for the variable *tree*.

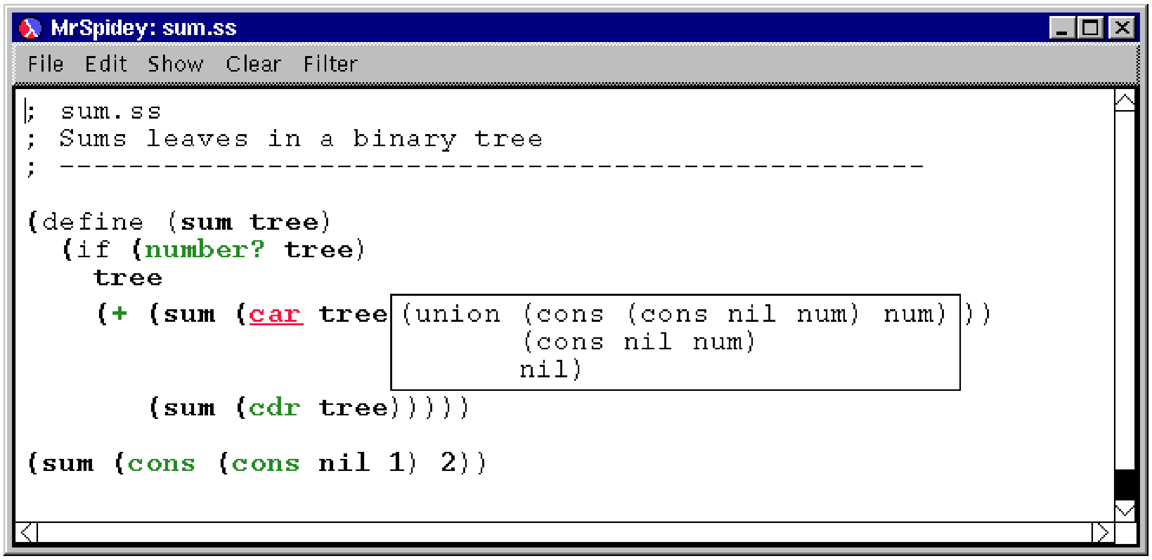


**Figure 5.3** Displaying type information

## 5.4 The Value Flow Browser

During the constraint derivation phase of the set-based analysis, MrSpidey infers subset constraints of the form $[\alpha \leq \beta]$ that describe the flow of values between various points in the program.

Since the type invariant of an expression is derived from the values propagated along the incoming subset constraints to that expression, the collection of all subset constraints for a program provides an explanation of the derivation of type invariants for that program. MrSpidey can display each subset constraint as an arrow overlaid on the program text. Because a large numbers of arrows would clutter the program text, these arrows are presented in a demand-driven fashion. The Parent option in the pop-up menu for an expression allows the programmer to view incoming edges for that expression in the value-flow graph. For example, figure 5.4 shows the incoming edges for the parameter *tree*.

Hyper-links associated with the head and tail of each arrow provide a fast means of navigating through textually distinct but semantically related parts of the program, which is especially useful on large programs. Clicking on the head of an arrow moves the focus to the term at the tail of the arrow, and vice versa.

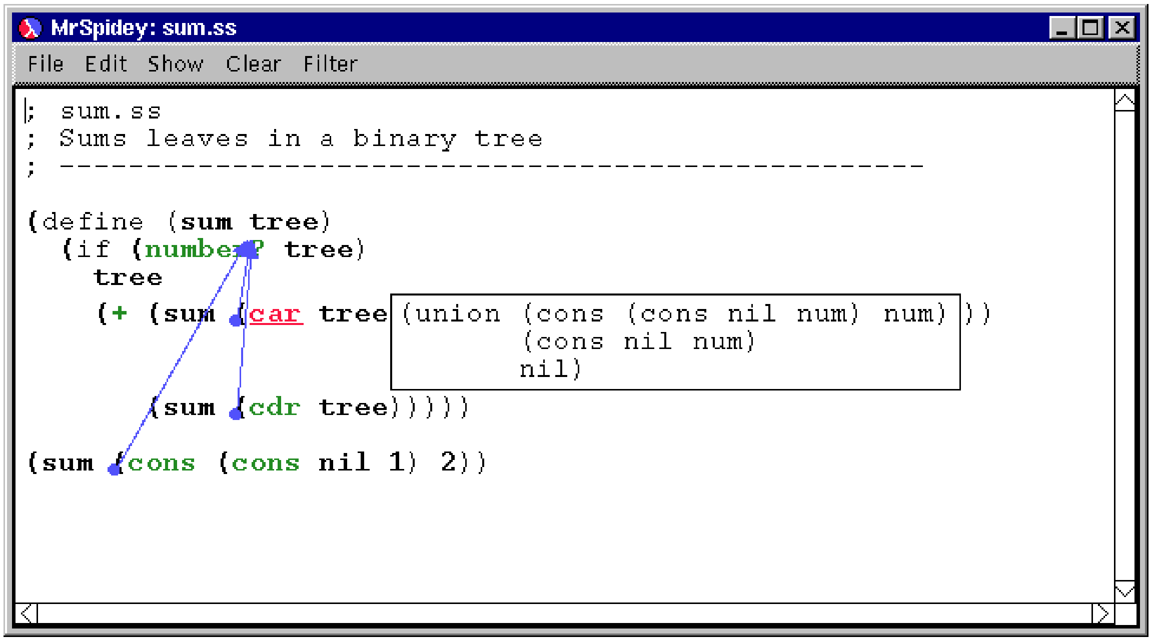


**Figure 5.4** Value source information

Using these facilities, a programmer who encounters a surprising value set invariant can proceed in an iterative fashion to expose arrows that describe the derivation of that invariant. To expedite this iterative process, MrSpidey also provides an Ancestors facility that automatically exposes all portions of the value flow graph that influence a particular invariant, thus providing the programmer with a complete explanation of the derivation of that invariant. For example, figure 5.5 shows the ancestors of the argument variable *tree*.

In some cases, the number of arrows presented by the ancestor facility is excessive. Since the programmer is typically only interested in a particular class of values, MrSpidey incorporates a *filter* facility that allows the programmer to restrict the displayed arrows to those that affect the flow of certain kinds of values. This facility is extremely useful for quickly understanding why a primitive operation may be applied to inappropriate argument values. Figure 5.6 shows how to configure the filter facility to consider only values corresponding to a particular constructor.

By using an appropriate combination of the ancestor and filter facilities, the programmer can quickly view the flow of a particular class of value through the program.

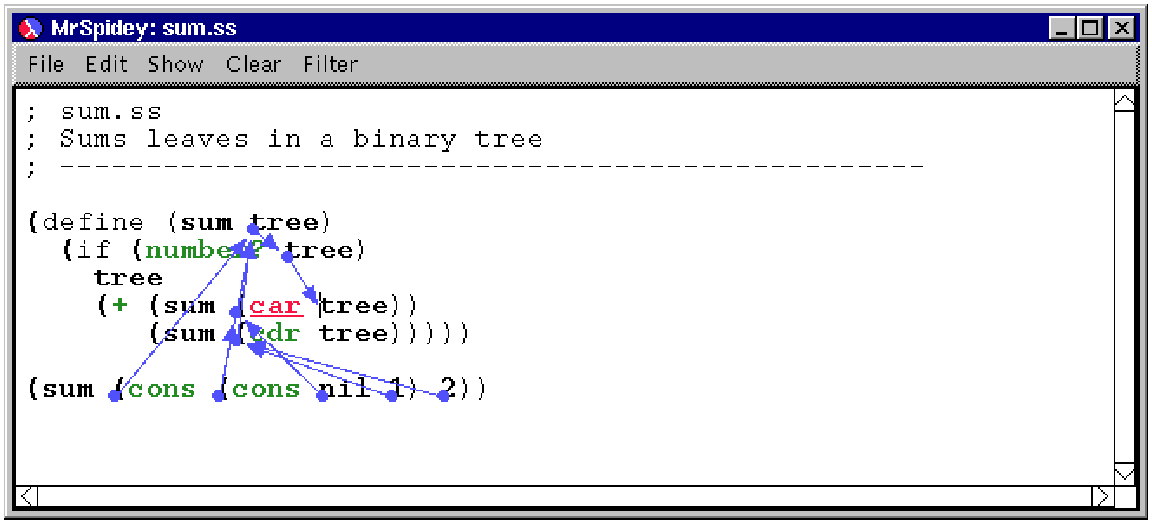


**Figure 5.5** Ancestors of tree

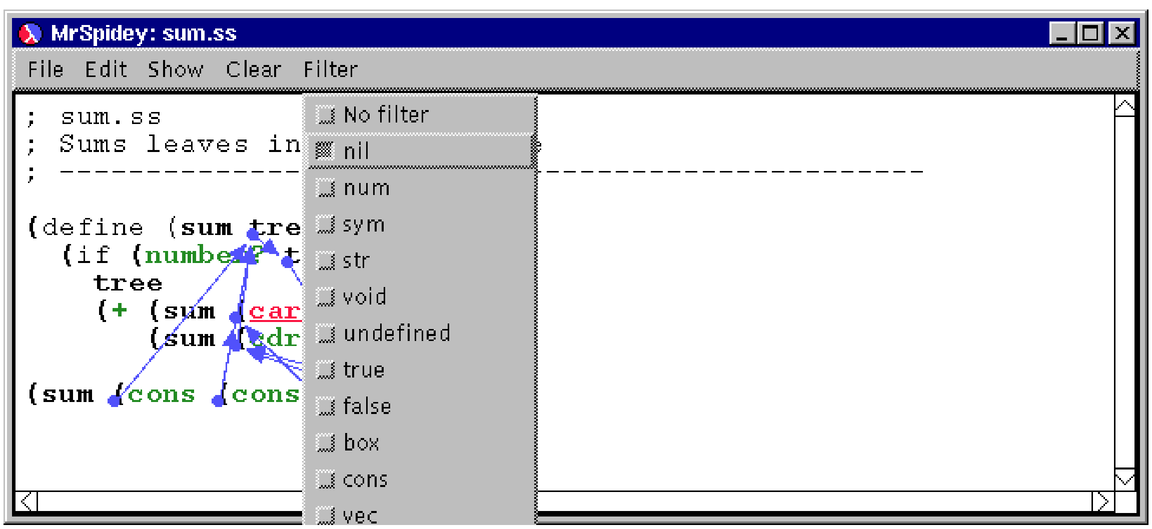


**Figure 5.6** Configuring the filter facility

For example, figure 5.7 shows the derivation of the `nil` component in the value set invariant for *tree*.

Finally, MrSpidey also provides Children and Descendants options on the pop-up menu for each expression. These options allow the programmer to view possible uses of the expression's return value.

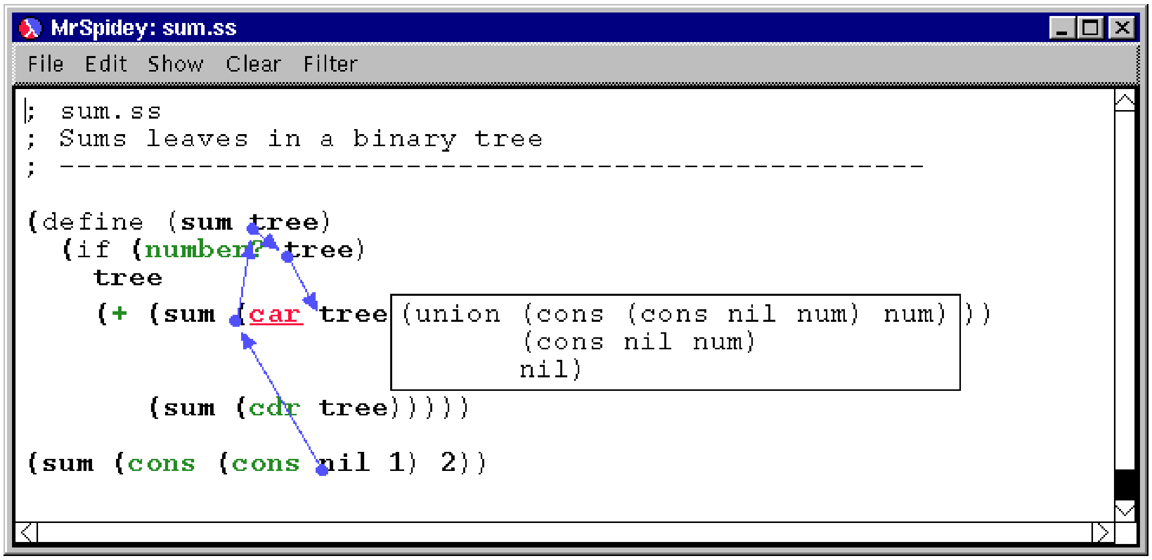


**Figure 5.7** Flow of `nil`

## 5.5 A Sample Debugging Session

To illustrate how the explanatory capabilities of MrSpidey can be used to identify and eliminate bugs, we describe how this tool could be used on the program `sum.ss`. When MrSpidey is invoked, it highlights the primitive operation **car** as unsafe (see Figure 5.1), indicating that this operation may raise a run-time error. Inspecting the type invariant for the operation argument, *tree*, shows that this set includes the inappropriate argument `nil` (see Figure 5.3). By using the ancestor and filter facilities, we can view how this erroneous value flows through the program, resulting in the display shown in figure 5.7. The displayed information makes it obvious that the error is caused by application of *sum* to the ill-formed tree (cons (cons nil 1) 2).

Although the example program `sum` presented in this section is trivial, it does provide a good example of the explanatory capabilities of MrSpidey. Before we can discuss how MrSpidey's explanatory capabilities scale to significantly larger programs, we must first deal with the problem of analyzing those larger programs, which is the topic of the following two chapters.

# Chapter 6

# Constraint Simplification

The traditional set-based analysis described in chapters 2 and 3 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 6.2 presents a proof-theoretic formulation of observable equivalence, and section 6.3 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 6.4.

## 6.1 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 $E$:

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

or, with the notation from section 2.2, $\mathcal{S}_1$ 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:

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

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 6.1.1** *(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 C.1. ∎

## 6.2 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 $\Delta$:

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

The 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 $(\mathit{compat})$, which is a tautology since $\mathtt{rng}(c) = \bot$.

The rules $(\mathit{reflex})$ and $(\mathit{trans}_\tau)$ capture the reflexivity and transitivity of the ordering relation $\sqsubseteq$; $(\mathit{compat})$ 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:

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

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

**Lemma 6.2.1** *(Soundness and Completeness of $\Delta$).* 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 C.2. ∎

This lemma implies that $\Delta(\mathcal{S})$ contains exactly those compound constraints that hold in all environments in $Soln(\mathcal{S})$. 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 $Soln(\mathcal{S}) \mid_E$. Therefore the following lemma holds.

**Lemma 6.2.2** For a compound constraint system $\mathbf{S}$, $\mathbf{S} \cong_E \Delta(\mathbf{S}) \mid_E$.

**Proof:** See Appendix C.2. ∎

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 $\Delta$ does not lend itself to an efficient implementation. Specifically, checking if two potential antecedents of $(trans_\tau)$ contain the same set expression $\tau$ 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 $\Delta$.

The alternative system consists of the inference rules $\Psi$ described in figure 6.1, together with the rules $\Theta$ from figure 2.3. The rules $(compose_{1\ldots4})$ of $\Psi$ 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 $\Psi$ are as described above. The rule $(trans_\alpha)$ of $\Psi$ 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, $\Theta$ and $(compose_{1\ldots4})$, compensate for this weakness. That is,

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

$$\frac{\alpha \le \mathtt{dom}(\beta) \qquad \beta \ge \kappa}{\alpha \le \mathtt{dom}(\kappa)} \qquad (compose_2)$$

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

$$\frac{\alpha \ge \mathtt{dom}(\beta) \qquad \beta \le \kappa}{\alpha \ge \mathtt{dom}(\kappa)} \qquad (compose_4)$$

$$\alpha \le \alpha \qquad (reflex)$$

$$\frac{\tau_1 \le \alpha \qquad \alpha \le \tau_2}{\tau_1 \le \tau_2} \qquad (trans_\alpha)$$

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

**Figure 6.1** The inference rule system $\Psi$

suitable combinations of these additional rules allow us to infer any constraint that could be inferred by the rule $(trans_\tau)$.

**Lemma 6.2.3** *(Equivalence of Proof Systems).* For a simple constraint system $\mathcal{S}$:

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

**Proof:** See Appendix C.2. ∎

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 6.2.4** $\Psi\Theta(\mathcal{S}) \mid_E \cong \Pi\Theta(\mathcal{S}) \mid_E$.

**Proof:** See Appendix C.2. ∎

Together, Lemmas 6.2.2, 6.2.3 and 6.2.4 provide the basis to introduce proof-theoretic equivalents of restricted entailment and observable equivalence.

**Definition 6.2.5.** ($\vdash^E_{\Psi\Theta}$, $=^E_{\Psi\Theta}$)

- $\mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2$ 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 6.2.6** *(Soundness and Completeness of $\vdash^E_{\Psi\Theta}$ and $=^E_{\Psi\Theta}$).*

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

**Proof:** See Appendix C.2. ∎

## 6.3 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 $\Theta$, 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 $\mathcal{S} = \{\alpha \leq \texttt{rng}(\alpha)\}$, then $\Psi(\mathcal{S})$ is the infinite set $\{\alpha \leq \texttt{rng}(\alpha), \alpha \leq \texttt{rng}(\texttt{rng}(\alpha)), \ldots\}$.

Fortunately, the infinite constraint systems inferred by $\Psi$ 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.

### 6.3.1 Regular Grammars

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

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

These languages are generated by a regular grammar. For each set variable $\alpha$, the grammar contains the non-terminals $\alpha_L$ and $\alpha_U$, which generate the above lower and upper bounds of $\alpha$, 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 $\Theta$) is described in figure 6.2, together with the productions in the corresponding regular grammar. This grammar describes the upper and lower non-constant bounds for each set variable in $\Pi(\mathcal{S}_P) \mid_E$. For example, the productions:

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

imply that $\alpha^x_L \mapsto^* \mathtt{dom}(\alpha_P)$. 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 $G_t(\mathcal{S}_P, E)$ |
|---|---|---|
| $f \leq \alpha^f$ | | $R \mapsto [f \leq \alpha^f_U]$ |
| $\mathtt{dom}(\alpha^f) \leq \alpha^x$ | $\alpha^x_L \mapsto \mathtt{dom}(\alpha^f_U)$ | |
| $1 \leq \alpha^1$ | | $R \mapsto [1 \leq \alpha^1_U]$ |
| $\alpha^1 \leq \mathtt{rng}(\alpha^f)$ | $\alpha^1_U \mapsto \mathtt{rng}(\alpha^f_U)$ | |
| $\mathtt{rng}(\alpha^f) \leq \alpha^a$ | $\alpha^a_L \mapsto \mathtt{rng}(\alpha^f_L)$ | |
| $\alpha^y \leq \alpha^r$ | $\alpha^y_U \mapsto \alpha^r_U \quad \alpha^r_L \mapsto \alpha^y_L$ | |
| $\alpha^r \leq \mathtt{dom}(\alpha^f)$ | $\alpha^r_U \mapsto \mathtt{dom}(\alpha^f_L)$ | |
| $g \leq \alpha^P$ | | $R \mapsto [g \leq \alpha^P_U]$ |
| $\mathtt{dom}(\alpha^P) \leq \alpha^y$ | $\alpha^y_L \mapsto \mathtt{dom}(\alpha^P_U)$ | |
| $\alpha^a \leq \mathtt{rng}(\alpha^P)$ | $\alpha^a_U \mapsto \mathtt{rng}(\alpha^P_U)$ | |
| | | |
| $\alpha^r \leq \alpha^x$ | $\alpha^r_U \mapsto \alpha^x_U \quad \alpha^x_L \mapsto \alpha^r_L$ | |
| $\alpha^1 \leq \alpha^a$ | $\alpha^1_U \mapsto \alpha^a_U \quad \alpha^a_L \mapsto \alpha^1_L$ | |
| $1 \leq \alpha^a$ | | $R \mapsto [1 \leq \alpha^a_U]$ |
| | | |
| | $\alpha^P_L \mapsto \alpha^P \quad \alpha^P_U \mapsto \alpha^P$ | $R \mapsto [\alpha_L \leq \alpha_U]\ \forall \alpha \in \mathit{SetVar}(\mathcal{S}_P)$ |

**Figure 6.2** 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 $\Pi$. For example, the constraint $[\alpha^1 \leq \mathtt{rng}(\alpha^f)] \in \mathcal{S}_P$ implies that for each upper bound $\kappa$ of $\alpha^f$, the rule $(compose_1)$ infers the upper bound $\mathtt{rng}(\kappa)$ of $\alpha^1$. Since, by induction, $\alpha^f$'s upper bounds are generated by $\alpha^f_U$, the production $\alpha^1_U \mapsto \mathtt{rng}(\alpha^f_U)$ generates the corresponding upper bounds of $\alpha^1$.

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 $\tau$ on $\alpha$. This bound must be inferred from an upper bound $\tau$ on $\beta$, 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 6.3.1.** (*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 6.3.2** 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 C.3. ∎

### 6.3.2 Regular Tree Grammars

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

- $G_r(\mathcal{S}, E)$ does not describe constraints of the form $[c \leq \tau]$. Thus, for example, the regular grammar for the example program component $P$ does not describe the constraint $[1 \leq \mathtt{rng}(\alpha^P)]$ in $\Pi(\mathcal{S}_P) \mid_E$.

- $G_r(\mathcal{S}, E)$ does not describe constraints inferred by the $(trans_\alpha)$ rule that are not bounds of the form $[\kappa \leq \alpha]$ or $[\alpha \leq \kappa]$. 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 $\Pi(\mathcal{S}_Q)\mid_E$, 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 $\Pi(\mathcal{S}_Q)\mid_E$.

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

**Definition 6.3.3.** (*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. ∎

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

**Lemma 6.3.4** Let $G = G_t(\mathcal{S}, E)$. Then $\Pi(\mathcal{S})\mid_E = \mathcal{L}_G(R)$.

**Proof:** See Appendix C.3. ∎

The grammar $G_t(\mathcal{S}_P, E)$ for the example program component $P$ is described in figure 6.2. This grammar yields all constraints in $\Pi(\mathcal{S}_P)\mid_E$. For example, the productions:

$$R \mapsto [1 \leq \alpha_U^a] \qquad \alpha_U^a \mapsto \mathtt{rng}(\alpha_U^P) \qquad \alpha_U^P \mapsto \alpha^P$$

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

### 6.3.3 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 $\Pi$ does not create any additional opportunities for rules in $\Theta$, and ($compat$) does not create any additional opportunities for $\Pi$ or $\Theta$.

**Lemma 6.3.5** *(Staging).* For any simple constraint system $\mathcal{S}$:

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

**Proof:** See Appendix C.3. ∎

### 6.3.4 The Entailment Algorithm

We can check entailment based on lemmas 6.3.4 and 6.3.5 as follows. Given $\mathcal{S}_1$ and $\mathcal{S}_2$, we close them under $\Theta$ and then have:

$$\begin{array}{rcl}
 & \mathcal{S}_2 \vdash^E_{\Psi\Theta} \mathcal{S}_1 & \\
\iff & \Psi\Theta(\mathcal{S}_2) \mid_E \supseteq \Pi\Theta(\mathcal{S}_1) \mid_E & \text{by defn } \vdash^E_{\Psi\Theta} \\
\iff & \Psi(\Theta(\mathcal{S}_2)) \mid_E \supseteq \Pi(\Theta(\mathcal{S}_1)) \mid_E & \text{by lemma 6.3.5} \\
\iff & \Psi(\mathcal{S}_2) \mid_E \supseteq \Pi(\mathcal{S}_1) \mid_E & \text{as } \mathcal{S}_i = \Theta(\mathcal{S}_i) \\
\iff & compat(\Pi(\mathcal{S}_2) \mid_E) \supseteq \Pi(\mathcal{S}_1) \mid_E & \text{by lemma 6.3.5} \\
\iff & compat(\mathcal{L}_{G_2}(R)) \supseteq \mathcal{L}_{G_1}(R) & \text{by lemma 6.3.4} \\
 & \text{where } G_i = G_t(\mathcal{S}_i, E) &
\end{array}$$

The containment question:

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

can be decided via a standard RTG containment algorithm [21]. To decide the more difficult question:

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

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 6.3. 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:

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

where $\alpha_L$, $\beta_U$ 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

**The Entailment Algorithm**

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

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

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

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

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

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

such that if:

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

then one of the following cases hold:

1. $\mathcal{L}_{G_1}([X \leq Y]) \subseteq \mathcal{L}_{G_2}(C \cup D)$.
2. $X = \mathtt{rng}(\alpha'_L)$, $Y = \mathtt{rng}(\beta'_U)$ and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha'_L, \beta'_U, C, D']$, where
$$D' = \{\langle \gamma'_L, \delta'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U)\}$$
3. $X = \mathtt{dom}(\alpha'_U)$, $Y = \mathtt{dom}(\beta'_L)$ and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\beta'_L, \alpha'_U, C, D']$, where
$$D' = \{\langle \delta'_L, \gamma'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{dom}(\gamma'_U), \delta_U \mapsto_{G_2} \mathtt{dom}(\delta'_L)\}$$
4. In no other cases does $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ hold.

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

1. $\forall R \mapsto_{G_1} [\alpha_L \leq \alpha_U].\ \mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \alpha_U, \{\langle \gamma_L, \gamma_U \rangle \mid \gamma \in \mathit{SetVar}(\mathcal{S}_2)\}, \emptyset]$, and
2. $\forall R \mapsto_{G_1} [c \leq \alpha_U].\ \mathcal{L}_{G_1}(\alpha_U) \subseteq \mathcal{L}_{G_2}(\{\gamma_U \mid R \mapsto_{G_2} [c \leq \gamma_U]\})$.

**Figure 6.3** The computable entailment relation $\vdash^E_{\text{alg}}$

(*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 6.3.6** *(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 C.4. ∎

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

## 6.4 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 6.3.4, 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 $\Theta$. 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 |
|---|---|---|---|
| $f \leq \alpha^f$ | $R \mapsto [f \leq \alpha_U^f]$ | | |
| $\mathtt{dom}(\alpha^f) \leq \alpha^x$ | $\alpha_L^x \mapsto \mathtt{dom}(\alpha_U^f)$ | | |
| $1 \leq \alpha^1$ | $R \mapsto [1 \leq \alpha_U^1]$ | $1 \leq \alpha^1$ | $1 \leq \alpha^1$ |
| $\alpha^1 \leq \mathtt{rng}(\alpha^f)$ | $\alpha_U^1 \mapsto \mathtt{rng}(\alpha_U^f)$ | | |
| $\mathtt{rng}(\alpha^f) \leq \alpha^a$ | $\alpha_L^a \mapsto \mathtt{rng}(\alpha_L^f)$ | | |
| $\alpha^y \leq \alpha^r$ | $\alpha_U^y \mapsto \alpha_U^r \quad \alpha_L^r \mapsto \alpha_L^y$ | $\alpha^y \leq \alpha^r$ | |
| $\alpha^r \leq \mathtt{dom}(\alpha^f)$ | $\alpha_U^r \mapsto \mathtt{dom}(\alpha_L^f)$ | | |
| $g \leq \alpha^P$ | $R \mapsto [g \leq \alpha_U^P]$ | $g \leq \alpha^P$ | $g \leq \alpha^P$ |
| $\mathtt{dom}(\alpha^P) \leq \alpha^y$ | $\alpha_L^y \mapsto \mathtt{dom}(\alpha_U^P)$ | $\mathtt{dom}(\alpha^P) \leq \alpha^y$ | |
| $\alpha^a \leq \mathtt{rng}(\alpha^P)$ | $\alpha_U^a \mapsto \mathtt{rng}(\alpha_U^P)$ | $\alpha^a \leq \mathtt{rng}(\alpha^P)$ | $\alpha^a \leq \mathtt{rng}(\alpha^P)$ |
| $\alpha^r \leq \alpha^x$ | $\alpha_U^r \mapsto \alpha_U^x \quad \alpha_L^x \mapsto \alpha_L^r$ | $\alpha^r \leq \alpha^x$ | |
| $\alpha^1 \leq \alpha^a$ | $\alpha_U^1 \mapsto \alpha_U^a \quad \alpha_L^a \mapsto \alpha_L^1$ | $\alpha^1 \leq \alpha^a$ | $\alpha^1 \leq \alpha^a$ |
| $1 \leq \alpha^a$ | $R \mapsto [1 \leq \alpha_U^a]$ | $1 \leq \alpha^a$ | $1 \leq \alpha^a$ |
| | $\alpha_L^P \mapsto \alpha^P \quad \alpha_U^P \mapsto \alpha^P$ | | |

**Figure 6.4** The constraint system, grammar and simplified systems for $P = (\lambda^g y.((\lambda^f x.1)\ y))$

### 6.4.1 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 6.4, 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:

$$\{\alpha_L^P, \alpha_U^P, \alpha_L^y, \alpha_U^a, \alpha_L^r, \alpha_U^1, \alpha_L^x, R\}$$

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.

### 6.4.2 Unreachable Constraint Simplification

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

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

### 6.4.3 Removing $\epsilon$-Constraints

A constraint of the form $[\alpha \leq \beta] \in \mathcal{S}$ is an *$\epsilon$-constraint*. Suppose $\alpha \notin E$ and the only upper bound on $\alpha$ in $\mathcal{S}$ is the $\epsilon$-constraint $[\alpha \leq \beta]$, *i.e.*, there are no other constraints of the form $\alpha \leq \tau$, $\texttt{rng}(\alpha) \leq \gamma$, or $\gamma \leq \texttt{dom}(\alpha)$ in $\mathcal{S}$. Then, for any solution $\rho$ of $\mathcal{S}$, the set environment $\rho'$ 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 $\alpha$ in $\mathcal{S}$ by $\beta$ while still preserving the observable behavior $\mathit{Soln}(\mathcal{S}) \mid_E$. This substitution transforms the constraint $[\alpha \leq \beta]$ to the tautology $[\beta \leq \beta]$, which can be deleted. Dually, if $[\alpha \leq \beta] \in \mathcal{S}$ with $\beta \notin E$ and $\beta$ having no other lower bounds, then we can replace $\beta$ by $\alpha$, 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 $\alpha^1$ is the $\epsilon$-constraint $[\alpha^1 \leq \alpha^a]$. Hence this algorithm replaces all occurrences of $\alpha^1$ by $\alpha^a$, 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 $\Theta(\mathcal{S})$.

### 6.4.4 Hopcroft's Algorithm

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

A natural strategy for generating $\rho'$ from $\rho$ 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 6.5 describes sufficient conditions to ensure that $\rho'$ 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 [27] for computing an equivalence relation on states in a DFA and then merge set variables according to their equivalence class.*

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

**Theorem 6.4.1** *(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 6.5; let

*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.

1. Use a variant of Hopcroft's algorithm [27] to compute an equivalence relation $\sim$ on the set variables of $\mathcal{S}$ that satisfies the following conditions:
   (a) Each set variable in $E$ is in an equivalence class by itself.
   (b) If $[\alpha \leq \beta] \in \mathcal{S}$ then $\forall \alpha \sim \alpha' \; \exists \beta \sim \beta'$ such that $[\alpha' \leq \beta'] \in \mathcal{S}$.
   (c) If $[\alpha \leq \texttt{rng}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha' \; \exists \beta \sim \beta'$ such that $[\alpha' \leq \texttt{rng}(\beta')] \in \mathcal{S}$.
   (d) If $[\texttt{rng}(\alpha) \leq \beta] \in \mathcal{S}$ then $\forall \alpha \sim \alpha' \; \exists \beta \sim \beta'$ such that $[\texttt{rng}(\alpha') \leq \beta'] \in \mathcal{S}$.
   (e) If $[\alpha \leq \texttt{dom}(\beta)] \in \mathcal{S}$ then $\forall \alpha \sim \alpha' \; \forall \beta \sim \beta'$ such that $[\alpha' \leq \texttt{dom}(\beta')] \in \mathcal{S}$.
2. Merge set variables according to their equivalence class.

**Figure 6.5** The *Hopcroft* algorithm

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 C.5. ∎

## 6.5 Simplification Benchmarks

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

We tested the algorithms on the constraint systems for nine program components on a 167MHz Sparc Ultra 1 with 326M of memory, using the MzScheme compiler [17]. The results are described in figure 6.6. 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 $\Theta$. 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,

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

**Figure 6.6** Behavior of the constraint simplification algorithms.

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.

# Chapter 7

# 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 7.2 presents experimental results on the effectiveness of the analysis, and section 7.3 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 7.4, together with experimental results on the behavior of the analysis.

## 7.1 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 $\Theta$, 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 $\Theta$, 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 [24]. 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.

## 7.2 Experimental Results

We implemented four variants of componential set-based analysis. Each analysis uses a particular simplification algorithm from chapter 6 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 7.1.

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.* 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 *$\epsilon$-removal* algorithm yields the best trade-off between efficiency and effectiveness of the simplification algorithms. The additional simplification performed by the more expensive *Hopcroft* algorithm is out-weighed by the overhead of running the algorithm. The tradeoff may change as we analyze larger programs.

To test the responsiveness of the componential analyses in an interactive setting based on an analyze-debug-edit cycle, we re-analyzed each benchmark after changing a randomly chosen component in that benchmark. The re-analysis times are shown in the sixth column of figure 7.1. 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.

## 7.3 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

*These times exclude scanning and parsing time.

| Program | lines | Analysis | Num. of constraints | Analysis time | Re-analysis time | Constraint file (bytes) |
|---|---|---|---|---|---|---|
| `scanner` | 1253 | *standard* | 61K | 14.1s | 7.7s | 572K |
| | | *empty* | 24K (39%) | 12.0s | 3.1s | 189K |
| | | *unreachable* | 15K (25%) | 9.7s | 2.0s | 39K |
| | | *$\epsilon$-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 |
| | | *$\epsilon$-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 |
| | | *$\epsilon$-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 |
| | | *$\epsilon$-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 |
| | | *$\epsilon$-removal* | 68K (<1%) | 239.6s | 13.3s | 1038K |
| | | *Hopcroft* | 38K (<1%) | 254.1s | 10.9s | 907K |

* indicates the analysis exhausted heap space

**Figure 7.1** Behavior of the modular analyses.

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 7.2) 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 7.3. Clicking on the arrow provides the option to zoom to and highlight the term at the

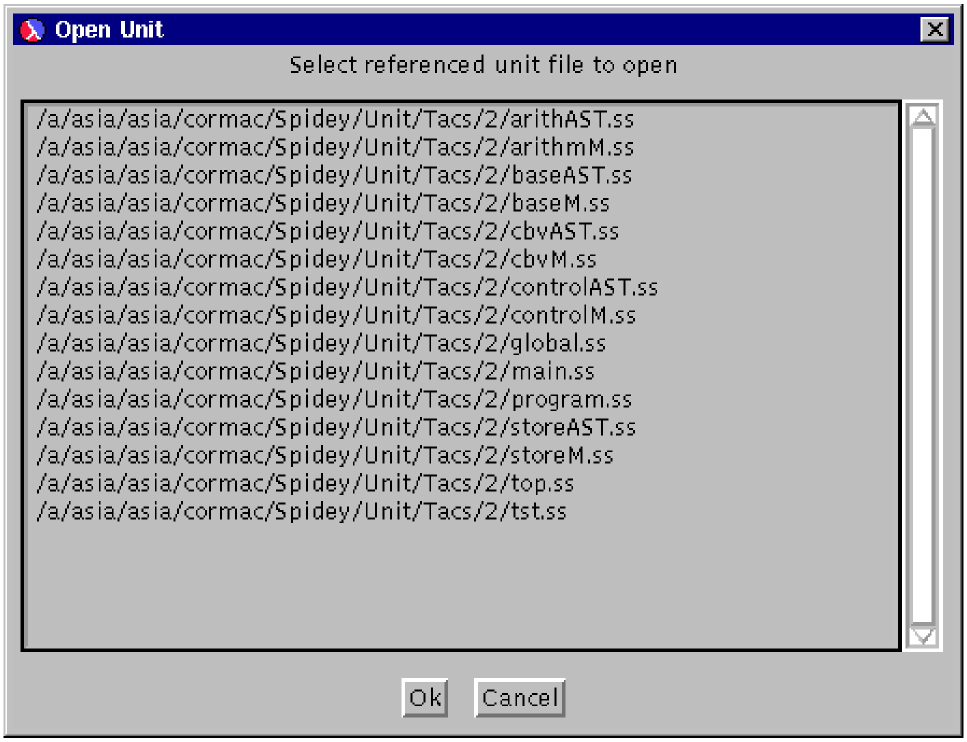


**Figure 7.2** The `File|Open ...` dialog box

other end of the arrow, as shown in figures 7.4 and 7.5. These facilities are useful for following the flow of values through multi-file programs.

## 7.4 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*).[†] The remaining four analyses are *smart* analyses that simplify the constraint system for each polymorphic definition.

[†]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.

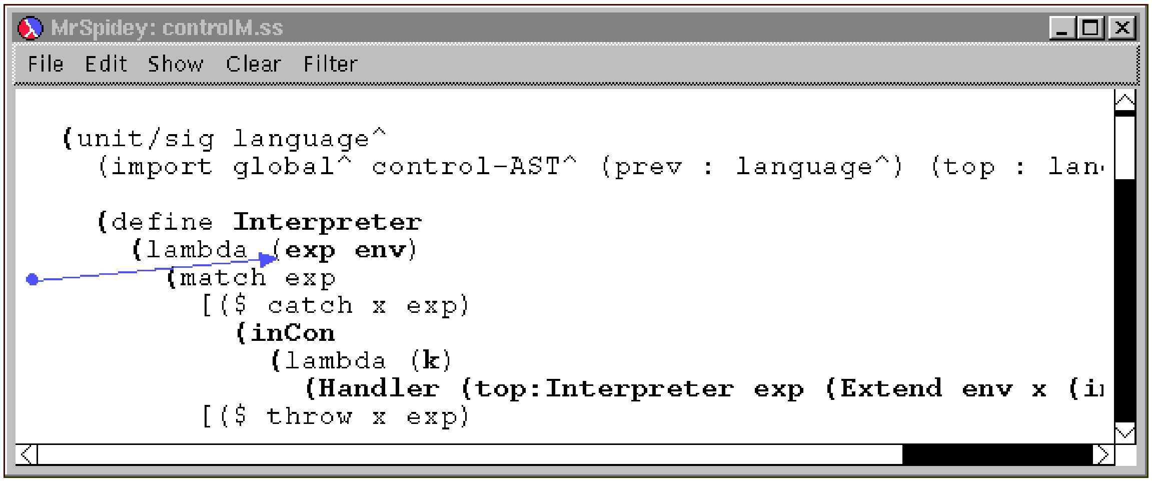


**Figure 7.3** Source in another file

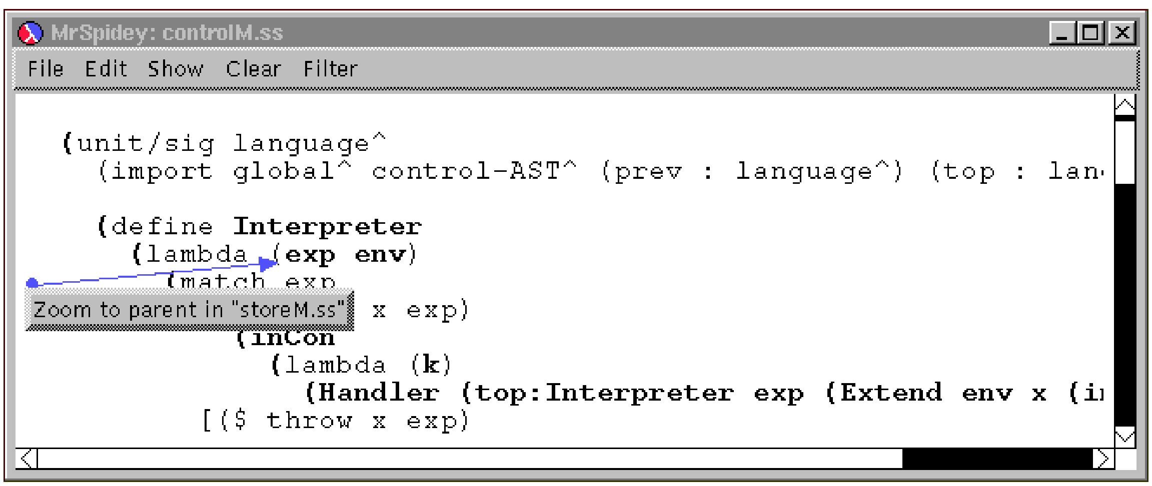


**Figure 7.4** Zooming to the source in another file

We tested the analyses using a standard set of benchmarks [28]. The results of the test runs are documented in figure 7.6. The second column shows the number of lines in each benchmark; the third column presents the time for the *copy* analysis; and columns four to seven show the times for each smart polymorphic analysis, as a percentage of the *copy* analysis time. For comparison purposes, the last column shows the relative time of the original, but less accurate, monomorphic analysis.

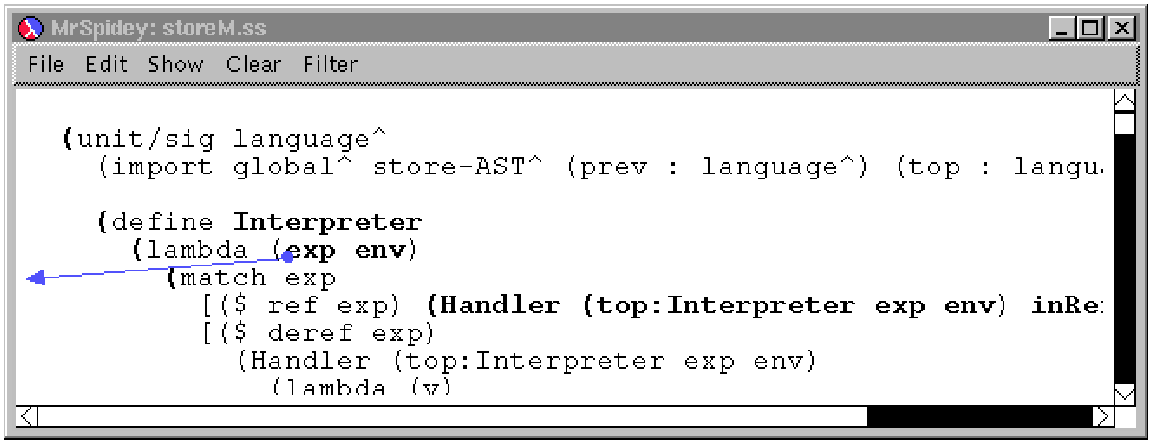


**Figure 7.5** The source in the other file

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

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

**Figure 7.6** Times for the smart polymorphic analyses, relative to the *copy* 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 $\epsilon$-*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.

# Chapter 8

# Evaluation of MrSpidey

We evaluated the usefulness of MrSpidey as static debugging tool using a number of programs.

## 8.1 Verifying a Web Server

Rice's web server software consists of a normal, fully-functional web server and a simple backup server. The backup server consists of a 76 line Scheme program that accepts connections to the web port, and returns a HTML page saying:

```
The Rice University computer science department's Web server has
been disconnected temporarily.
```

We used MrSpidey to statically debug the backup server. MrSpidey detected a single, potentially unsafe operation, where the analysis suggested that the end-of-file value could be returned by `read-line` and then passed as an argument to the operation `string-length`. An inspection of the program revealed that this behavior could never actually occur. After simplifying two lines of code, MrSpidey was able to verify the safety of the `string-length` operation, and produced the summary:

```
TOTAL CHECKS: 0 (of 56 possible checks is 0.0%)
```

## 8.2 Verifying gunzip

MzScheme's standard library contains Scheme code for inflating deflated (PKZIP's method 8 compressed) data. The code consists of a single 800 line file `inflate.ss`, translated directly from the `gzip` source distribution. MrSpidey initially reported that out of the 650 operations in the program, it was unable to verify the safety of 27 (or 5%) of them.

The majority of these unsafe operations were `vector-ref` operations that, according to MrSpidey's analysis, could be applied to non-vector values. We used MrSpidey's filter and path-to-source facilities to identify the source of these erroneous, non-vector values.

- Some of these non-vector values came from the third field in a structure called `huft`. This field contains a number in some situations and a vector in others. By splitting the field into separate fields for numeric and vector values, we reduced the unsafe operation count to 18.

- Some non-vector values came from a stack of *tables*, each of which should be a vector. However, the stack was initialized as a vector of zero's. By changing the initial value of the stack to a vector of empty vectors, we reduced the unsafe operation count by one to 17.

- The non-vector value `null` value was mistakenly passed as an argument to a function `huft_build`, instead of the empty vector. Changing the program to pass the empty vector instead reduced the unsafe operation count by two to 15.

- The non-vector value `null` also was the initial value of a variable that was later assigned to a vector. Initializing the variable to the empty vector reduced the unsafe operation count by three to 12.

Some of the remaining unsafe operations were caused by an extremely complicated return protocol from the function `huft_build`. The function returns a tuple where the type of some fields depends on the value of other fields. We simplified the return protocol of the function by using exceptions to handle erroneous conditions. This reduced the unsafe operation count by three to 9.

Minor rewriting to produce a cleaner coding style further reduced the unsafe operation count by two to 7. All of the remaining unsafe operations are actual errors that resulted from failing to detect and handle a truncated input file. Thus the original program would crash on a truncated input file with a message such as:

```
> (gunzip "~/tmp/t")
char=?: expects type <character> as 1st arg;
given \#<eof> (type <eof>); other args: \#\∅
```

By adding code to check of the end-of-file case, we finally reduced the unsafe operation count to 0, and the resulting statically debugged program gracefully handles truncated input files:

```
> (gunzip "~/tmp/t")
gunzip: Unexpected end of input file
```

## 8.3 Verifying an Extended Direct Semantics Interpreter

*Extended direct semantics* is a format for denotational language specifications that accommodates orthogonal extensions of a language without changing the denotations of existing phrases [4]. The semantics of a language is specified in this format using a tower of interpreters. The tower starts with a basic interpreter, which can only interpret certain trivial expressions. This basic interpreter is then composed with additional interpreters for the various constructs in the language.

We used MrSpidey to statically debug an interpreter expressed in this style. In addition to the basic interpreter, this program contains interpreters for:

- Arithmetic operations (integer constants, `add1` and `sub1`)
- Call-by-value functions (variables, functions and applications)
- Control operations (`catch` and `throw`)
- Assignments (`ref`, `deref` and `setref`)

Each interpreter consists of two units: one unit defining the abstract syntax structures and a second unit that specifies how to interpret the new abstract syntax. An additional five units specify global definitions and test cases, for a total of 15 units in 15 separate files.

Porting the program to MzScheme's unit system introduced an error, where one of the units was defined to take three input signatures:

(**unit/sig** (*Program*)
  (**import** *global^ language^ arith-AST^*)
  ...

but was only passed two signatures in the main `compound-unit` clause:

> ...
> [*PROGRAM*
>  : *Program^*
>  ((*reference-unit* "program.ss") *GLOBAL* *TOP*)]
> ...

When this program was analyzed, MrSpidey produced the warning:

```
Warning: Unit takes 47 imports, given 26 in file "program.ss" line 2
```

together with a hyper-link to the relevant unit.

After this bug was fixed, MrSpidey was able to verify the safety of the program, and produced the following summary:

```
main.ss         CHECKS:       0  (of 39 possible checks is 0.0%)
program.ss      CHECKS:       0  (of 4 possible checks is 0.0%)
top.ss          CHECKS:       0  (of 1 possible checks is 0.0%)
storeM.ss       CHECKS:       0  (of 161 possible checks is 0.0%)
controlM.ss     CHECKS:       0  (of 49 possible checks is 0.0%)
cbvM.ss         CHECKS:       0  (of 40 possible checks is 0.0%)
arithmM.ss      CHECKS:       0  (of 24 possible checks is 0.0%)
baseM.ss        CHECKS:       0  (of 21 possible checks is 0.0%)
storeAST.ss     CHECKS:       0  (of 11 possible checks is 0.0%)
controlAST.ss   CHECKS:       0  (of 5 possible checks is 0.0%)
arithAST.ss     CHECKS:       0  (of 2 possible checks is 0.0%)
cbvAST.ss       CHECKS:       0  (of 2 possible checks is 0.0%)
baseAST.ss      CHECKS:       0  (of 1 possible checks is 0.0%)
global.ss       CHECKS:       0  (of 41 possible checks is 0.0%)
tst.ss          CHECKS:       0  (of 46 possible checks is 0.0%)
```

Part of the reason that MrSpidey is so successful on this program is that the program had already been carefully written so that Soft Scheme [46] could verify its safety, and thus the style of the program was already well-suited for automatic analysis techniques.

## 8.4 Statically Debugging HHL

We used MrSpidey to statically debug a program under development. This program, called `HHL`, is a hardware verifier using heterogeneous logic. It consists of 3312 lines of Scheme code distributed over 12 files, and interfaces to the Omega calculator [38].

We used MrSpidey to analyze the entire program, and then concentrated on statically debugging one file, `prover.ss`, containing 500 lines of code. MrSpidey initially reported that out of 466 operations in the file, it was unable to verify the safety of 17 (or 4%) of them. Nine of these unsafe operations were caused by bugs in the program.

- Two unsafe `string-append` operations were caused by a variable being erroneously initialized with `void`, instead of with a string.

- An arity check was caused be a two-argument function being applied to a single argument.
- An unsafe `car` operation was applied to the result value of `read`, which is not necessarily a pair.
- Three other unsafe string operations were applied to the result of `read-line`, which can return the end-of-file value in addition to strings.
- On two occasions, the primitive `andmap` was applied to a single argument.

The remaining eight unsafe operations appear to be caused by limitations in the underlying analysis.

# Chapter 9

# Related Work

## 9.1 Static Debuggers

A number of interactive analysis tools and static debugging systems have been developed for various programming languages. Some address different concerns; none provide an explanation of the derived invariants.

Syntox [3] is a static debugger for a subset of Pascal. Like MrSpidey, it associates run-time invariants, *i.e.*, numeric ranges, with statements in the program. Because Syntox does not provide an explanation of these invariants, it is difficult for a programmer to decide whether an unexpected invariant is caused by a weakness in the proof system or a flaw in the program. In addition, the existing system processes only a first-order language, though Bourdoncle explains how to extend the analysis [3:Section 5].

Several environments [30, 6, 26, 44, 41] have been built for parallel programming languages to expose dependencies, thus allowing the programmer to tune programs to minimize these dependencies. In particular, the ParaScope [6, 30] and D editors [26] have many similarities to MrSpidey. Both MrSpidey and the editors provide information at varying levels of granularity; both retain source correlation through transformations; and both depict dependencies graphically. However, unlike MrSpidey, the editors process a language with extremely simple control- and data-flow facilities, and therefore do not need to provide a supporting explanation for the derived dependencies.

The Extended Static Checking system (ESC) [8] is a static debugger designed to detect program errors such as nil dereferences, out-of-bound array accesses and deadlocks, and race conditions. If an error may occur, then ESC returns a counter-example in the form of an assignment of values to program variables that can cause the error. ESC is based on an automatic theorem prover that is more powerful than MrSpidey's constraint-based approach, but which is also more expensive. Hence ESC cannot analyze large programs, and is restricted to working on a per-procedure basis.

Therefore, this approach requires that the programmer annotates the program with specifications for the interfaces between procedures, which significantly increases the start-up cost of using ESC. In contrast, MrSpidey can be immediately applied to existing programs without the need for any additional annotations.

Microsoft's Program Analysis Group has developed a static debugger for small C programs.* In a fashion similar to MrSpidey, this debugger analyzes programs and uses the resulting invariants to identify potential bugs. However, it cannot explain the derivation of the resulting invariants, and only works on small programs of up to a few hundred lines of code.

## 9.2 Constraint Simplification

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 [9] 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 [12] 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 [37] 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 [43] present constrained types that are similar to our constraint systems, and they describe an incomplete algorithm for deciding the subtyping relation between constrained types. They do not discuss constraint simplification. Eifrig, Smith and Trifonov [11] discuss constraint simplification in the context of type inference for objects. They present three algorithms for simplifying constraint systems,

*Personal communication: Daniel Weise (February 97).

two of which which are similar to the *empty* and $\epsilon$-*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* [10] describe algorithms for simplifying data-flow equations. These algorithms are similar to the $\epsilon$-*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.

# Chapter 10

# Limitations and Future Work

Although MrSpidey has proven to be an effective tool for statically debugging a variety of programs, including the programs described above, there are several aspects of MrSpidey that could be improved.

## 10.1 Size of Types

The constraint simplification algorithms can reduce the size of type invariants by a significant factor, but in some cases the resulting types are still excessively large. This problem could be partly remedied by developing more aggressive constraint simplification algorithms. However, more aggressive algorithms may not completely solve the problem, because for a given constraint system, even the smallest equivalent system may yield an excessively large type.

Traditional static type systems avoid this problem by introducing a new type name each time a new datatype is declared. For example, the Standard ML declaration corresponding to the binary trees we considered earlier is:

```
datatype tree = Leaf of int
              | Node of tree * tree
```

This declaration introduces the type name `tree` that can be used as a shorthand for describing any valid tree build using `Leaf` and `Node`. By comparison, in MrSpidey the corresponding type expression would need to explicate the combination of `Leaf` and `Node` constructors used to build the tree.

As a partial solution to this problem, MrSpidey allows the programmer to configure the type display algorithm to avoid displaying the types for the fields of a structure or for the instance variables of an object (see appendix D.2.2). Since most excessively large types involve either structures or objects, selecting this option keeps the size of types manageable, even for complicated programs.

## 10.2 Accuracy of the Analysis

Componential set-based analysis infers reasonably accurate invariants describing the value sets for program expressions, even in the presence of complex control-flow and data-flow patterns. However, since the actual run-time behavior of a program is undecidable, these invariants are necessarily approximate, and there are some situations where the invariants are overly coarse.

A serious limitation of MrSpidey is that it does not perform any analysis of integer subranges. Thus, MrSpidey cannot detect errors that may occur due to an array index being out-of-bounds. Other static debuggers [3, 8] can detect this kind of error, but it is not clear how well these debuggers scale up to large, complex programs.

## 10.3 State in the User Interface

For the most part, MrSpidey's graphical user interface presents static debugging information in a natural and intuitive manner. However, the filter facility (described in section 5.4) introduces some hidden state into the user interface. That is, although the currently selected filter affects the behavior of the interface, this filter is not displayed. Hence it is easy for the programmer to forget which filter is currently selected, and to be confused by the displayed value flow.

There are two possible solutions to this problem. The first is simply to display the currently selected filter. However, state can be confusing in a user interface, even when it is displayed. Therefore, a better solution would be develop an alternative interface that does not require this state information.

One such approach is to allow the programmer to click on the internal components of a type invariant, yielding a pop-up menu as before, and then to have options in that menu that display the value flow only for values corresponding to the selected component of the type. This approach would remove the need for state in the user interface, and would hopefully provide programmer with better access to the derivation of the program's invariants.

## 10.4 Signatures

All the constraint simplification algorithms we considered 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 simplified constraint system that does not preserve the observable behavior of the original system, but only *entails*, or conservatively approximates, that behavior. This approach allows the use of 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. The appropriate constraint derivation rule is:

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

Since the signature-based constraints are expected to be smaller than the derived ones, this approach could significantly reduce analysis times for large programs. It would also allow for a program component to be statically debugged with respect to its signature, without needing to access the entire source program.

# Appendix A

# Proofs for Chapter 3

## A.1 Subject Reduction Proof

**Lemma 2.5.3** *(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 $(\beta_v)$. 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:

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

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.1.1:

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

By the Flow Lemma 2.5.5:

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

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

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

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

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

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

  The typing derivation on $M_1$ is of the form:

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

  By the Subject Reduction for **let** Lemma A.1.2:

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

  as required.

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

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

  The typing derivation on $M_1$ is of the form:

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

  Hence $\Gamma \vdash V : \alpha, \mathcal{S}_V$, and by the Flow Lemma 2.5.5, $\Gamma \vdash V : \beta, \mathcal{S}_2$ where $\mathcal{S}_1 \models \mathcal{S}_2$.

∎

**Lemma A.1.1** *(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

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

**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 2.5.5.

- $(const)$: 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 $N = (\mathbf{let}\ (y\ W)\ M)$, where $W \in \mathit{Value}$. Hence:

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

  and therefore by the following Subject Reduction for **let** Lemma A.1.2:

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

  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.1.2:

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

  or, equivalently:

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

  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 $\beta_{let}$-expanded expression*. The following lemma demonstrates this equivalence of constraint systems.

**Lemma A.1.2** *(Subject Reduction for* **let***).*

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

if and only if

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

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

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

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:

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

---

*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 2.5.3.

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 $N = (\mathbf{let}\ (y\ W)\ M)$, where $W \in \mathit{Value}$. Hence:

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

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:

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

or, equivalently:

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

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 $\psi$ 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 2.5.5:

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

  as required.

∎

**Lemma 2.5.5** *(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:** 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 $\gamma \in \mathit{SetVar}$, 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 $\psi$ 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 2.6.2** *(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 $\rho = \bigcap_s Soln(\mathcal{S})$, using the pointwise extension of $\bigcap_s$ to set environments. We prove that $\rho \in Soln(\mathcal{S})$ by showing that $\rho$ 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.2.1.

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

$$\begin{array}{rll}
\rho(\alpha) & = & \displaystyle\bigcap_{\rho' \in Soln(\mathcal{S})}{}_s \; \rho'(\alpha) \\
& \sqsubseteq & \displaystyle\bigcap_{\rho' \in Soln(\mathcal{S})}{}_s \; \rho'(\texttt{rng}(\beta)) \\
& & \text{by Lemma A.2.1, since } \rho'(\alpha) \sqsubseteq \rho'(\texttt{rng}(\beta)) \\
& = & rng\left(\displaystyle\bigcap_{\rho' \in Soln(\mathcal{S})}{}_s \; \rho'(\beta)\right) \qquad \text{by definition of } \sqcap_s \\
& = & rng\,(\rho(\beta)) \\
& = & \rho(\texttt{rng}(\beta))
\end{array}$$

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

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

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

**Lemma A.2.1** 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:

$$\bigcap_{s\, i \in I} x_i \;\sqsubseteq\; \bigcap_{s\, i \in I} y_i$$

$$\bigsqcup_{s\, i \in I} x_i \;\sqsubseteq\; \bigsqcup_{s\, i \in I} y_i$$

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

$$\bigsqcap_{i \in I} x_i \;\sqsubseteq_s\; \bigsqcap_{i \in I} y_i$$

$$\bigsqcup_{i \in I} x_i \;\sqsubseteq_s\; \bigsqcup_{i \in I} y_i$$

**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:

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

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

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

Hence

$$\bigcap_{s\ i\in I} x_i \sqsubseteq \bigcap_{s\ i\in I} y_i$$

as required. ∎

**Theorem 2.6.5** *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:**

$$\begin{aligned} \mathcal{S} \vdash_\Theta c \leq \alpha &\iff \mathcal{S} \models c \leq \alpha && \text{by lemma A.2.2} \\ &\iff \forall \rho \models \mathcal{S}.\ \rho \models c \leq \alpha \\ &\iff \forall \rho \in \mathit{Soln}(\mathcal{S}).\ c \in \mathit{const}(\rho(\alpha)) \\ &\iff c \in \bigcap(\{\mathit{const}(\rho(\alpha)) \mid \rho \in \mathit{Soln}(\mathcal{S})\}) \\ &\iff c \in \mathit{const}(\sqcap(\{\rho(\alpha) \mid \rho \in \mathit{Soln}(\mathcal{S})\})) \\ &\iff c \in \mathit{const}(\sqcap(\{\rho \mid \rho \in \mathit{Soln}(\mathcal{S})\})(\alpha)) \\ &\iff c \in \mathit{const}(\mathit{LeastSoln}(\mathcal{S})(\alpha)) \end{aligned}$$

The correctness of this theorem then follows from definition 2.6.3. ∎

**Lemma A.2.2** *(Soundness and Completeness of $\Theta$).* For any simple constraint system $\mathcal{S}$:

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

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

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

where 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 $\Theta$. These rules propagate set variables denoting the result of functions in $\alpha$ 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 $\alpha$, and hence this propagation in performed in the definition of $F(\rho)(\alpha)$ by finding all $\gamma$ 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 $\rho$, 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 $\Theta$:

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

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

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

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

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

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

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

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

- 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. ∎

# Appendix B

# Proofs for Chapter 5

## B.1 Correctness of *MkType*

**Lemma 4.2.1** *(Correctness of MkType). If $\mathcal{S}$ is a simple constraint system then:*

$$\mathcal{M}[\mathit{MkType}(\mathcal{S}, \alpha)] = \mathit{LeastSoln}(\mathcal{S})(\alpha)$$

**Proof:** By the definition of $\mathcal{M}$ and *MkType*:

$$\mathcal{M}[\mathit{MkType}(\mathcal{S}, \alpha)] = \rho_s(\alpha)$$

where $\rho_s$ is the least fixed point under $\sqsubseteq_s$ of the functional:

$$\rho \mapsto \mathcal{M}_\rho[\mathit{MkType}'(\mathcal{S}, \alpha_i)]$$

where $\alpha_i$ ranges over $\mathit{SetVar}(\mathcal{S}) = \{\alpha_1, \ldots, \alpha_n\}$. Hence $\rho_s$ is the least solution to the system of equalities:

$$\rho(\alpha_i) = \mathcal{M}_\rho[\mathit{MkType}'(\mathcal{S}, \alpha_i)]$$

But (ignoring pairs to simplify the presentation):

$$\begin{aligned}\mathcal{M}_\rho[\mathit{MkType}'(\mathcal{S}, \alpha_i)] = \langle\ & \{b \mid \mathcal{S} \vdash_\Theta b \leq \alpha_i\} \cup \{t \mid \mathcal{S} \vdash_\Theta t \leq \alpha_i\}, \\ & \textstyle\bigsqcup_s \{\rho(\beta) \mid \mathcal{S} \vdash_\Theta \alpha_i \leq^* \delta, \beta \leq \mathtt{dom}(\delta)\}, \\ & \textstyle\bigsqcup_s \{\rho(\beta) \mid \mathcal{S} \vdash_\Theta \beta \leq \mathtt{rng}(\alpha_i)\}\ \rangle\end{aligned}$$

Equating each component of $\rho(\alpha_i)$ with the corresponding component of the above tuple, we see that $\rho_s$ must be the least solution to the system of equalities (1):

$$\begin{array}{rcl} \mathit{const}(\rho(\alpha_i)) & = & \{c \mid \mathcal{S} \vdash_\Theta c \leq \alpha_i\} \\ \mathit{dom}\,(\rho(\alpha_i)) & = & \bigsqcup_s \{\rho(\beta) \mid \mathcal{S} \vdash_\Theta \alpha_i \leq^* \delta, \beta \leq \mathtt{dom}(\delta)\} \\ \mathit{rng}\,(\rho(\alpha_i)) & = & \bigsqcup_s \{\rho(\beta) \mid \mathcal{S} \vdash_\Theta \beta \leq \mathtt{rng}(\alpha_i)\} \end{array}$$

An inductive argument shows that the system of equalities (1) holds if and only if the following system of equalities (2) also holds, where $p$ ranges over paths:

$$const(p(\rho(\alpha_i))) = \{c \mid \mathcal{S} \vdash_\Delta c \leq p(\alpha_i)\}$$

Hence $\rho_s$ is also the least solution to (2). But:

$$const(p(\rho(\alpha_i))) = \{c \mid \rho \models c \leq p(\alpha_i)\} ,$$

hence $\rho_s$ is the least solution for $\rho$ to the following condition (3):

$$\rho \models c \leq p(\alpha_i) \quad \Longleftrightarrow \quad \mathcal{S} \vdash_\Delta c \leq p(\alpha_i)$$

But (3) holds if and only if $\rho \models \mathcal{S}$. Therefore $LeastSoln(\mathcal{S}) = \rho_s$, since $\rho_s$ is the least solution to (3), and hence:

$$\mathcal{M}[MkType(\mathcal{S}, \alpha)] = LeastSoln(\mathcal{S})(\alpha)$$

as required. ∎

# Appendix C

# Proofs for Chapter 6

## C.1 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 6.1.1** *(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.

∎

## C.2 Proofs for Proof Theory of Observable Equivalence

The following proofs require a number of auxiliary definitions.

**Definition C.2.1.** (*Paths*)

- A *path* $p, q \in Path$ is a sequence of the constructors $\mathtt{dom}$ and $\mathtt{rng}$. We use $\epsilon$ 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 $\pi p$, is the number of $\mathtt{dom}$'s in p, taken modulo 2. If $\pi p$ is 0, we say $p$ is *monotonic*, otherwise $p$ is *anti-monotonic*.
- For a path $p$, the notation $p(\tau)$ denotes the set expression $\tau$ enclosed in the $\mathtt{dom}$'s and $\mathtt{rng}$'s of $p$, *i.e.*, if $p \equiv \mathtt{rng}.\mathtt{dom}$, then $p(\alpha) \equiv \mathtt{rng}(\mathtt{dom}(\alpha))$.
- 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:

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

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

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

∎

**Lemma 6.2.1** *(Soundness and Completeness of $\Delta$). 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 $\Delta$ is straightforward. To demonstrate the completeness of $\Delta$, 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 $\rho$ 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 $\rho$ satisfies every constraint $\mathbf{C}' \in \mathbf{S}$.

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

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

    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 $\alpha$, 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 $\kappa'$, $\mathbf{S}' \vdash_\Delta c \leq \kappa'$ 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 $(\mathit{trans}_\tau)$ 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 $(\mathit{trans}_\tau)$, 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 6.2.2** *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 $\Delta$ 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 $\rho$ to a super-environment $\rho'$ 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

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

as required.

- Suppose $\mathbf{C} = [p(\alpha) \leq q(\beta)]$. Then for any path $r$, $\mathbf{S} \vdash_\Delta r(p(\alpha)) \leq_{\pi r} r(q(\beta))$. 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 $\rho$ and $\rho'$ 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 $\rho$ and $\rho'$ agree on $E$, and the lemma holds. ∎

**Lemma C.2.2** 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 6.2.3** *(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 $\mathbf{C} \in \Delta(\mathcal{S})$:

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

- $(compose_2)$, $(compose_3)$, $(compose_3)$: These cases follow by similar reasoning.

- $(reflex)$, $(trans_\alpha)$, $(compat)$: These rules are either equivalent to or subsumed by corresponding rules in $\Delta$.

- $(s_1)$, $(s_4)$, $(s_5)$: 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 $\mathbf{C} \in \Psi\Theta(\mathcal{S}) \setminus \Delta(\mathcal{S})$, 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 $\Psi$ 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:

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

  We proceed by case analysis on $\tau$ 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 $\tau \in \mathit{SetVar}$, then $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$ via $(\mathit{trans}_\alpha)$.
- Suppose $\tau = \mathtt{rng}(\tau')$. If $\tau' \in \mathit{SetVar}$ 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:

$$\frac{\alpha_1 \leq \mathtt{rng}(\beta_1) \qquad \beta_1 \leq \tau'}{\alpha_1 \leq \mathtt{rng}(\tau')} \quad (\mathit{compose}_1)$$

where $\tau_1 = \alpha_1$, or:

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

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

$$\frac{\tau' \leq \beta_2 \qquad \mathtt{rng}(\beta_2) \leq \alpha_2}{\mathtt{rng}(\tau') \leq \alpha_2} \quad (\mathit{compose}_3)$$

where $\tau_2 = \alpha_2$, or:

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

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 $(\mathit{compose}_1)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(\mathit{compose}_3)$. 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 $(\mathit{trans}_\alpha)$. By induction, $[\beta_1 \leq \beta_2] \in \Psi\Theta(\mathcal{S})$, and the following derivation then shows that $[\tau_1 \leq \tau_2] \in \Psi\Theta(\mathcal{S})$.

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

  * Suppose $\tau_1 \leq \mathtt{rng}(\tau')$ is inferred via $(\mathit{compose}_1)$ and $\mathtt{rng}(\tau') \leq \tau_2$ is inferred via $(\mathit{compat})$. Then $\{\beta_1 \leq \tau', \tau' \leq \tau_2'\} \subseteq \Delta(\mathcal{S})$, and therefore

$[\beta_1 \le \tau_2'] \in \Delta(\mathcal{S})$ via $(trans_\alpha)$. By induction, $[\beta_1 \le \tau_2'] \in \Psi\Theta(\mathcal{S})$, and the following derivation shows that $[\tau_1 \le \tau_2] \in \Psi\Theta(\mathcal{S})$.

$$\frac{\alpha_1 \le \mathtt{rng}(\beta_1) \qquad \beta_1 \le \tau_2'}{\alpha_1 \le \mathtt{rng}(\tau_2')} \quad (compose_1)$$

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

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

– 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 $\tau$.

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 6.2.4** $\Psi\Theta(\mathcal{S}) \mid_E \cong \Pi\Theta(\mathcal{S}) \mid_E$.

**Proof:** Since the rule $(compat)$ does not create any $\Pi$ or $\Theta$ 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 $\mathbf{C}' \in compat(\Pi\Theta(\mathcal{S})) \mid_E$ with a smaller derivation. Therefore $\rho \models \mathbf{C}'$, and hence since $(compat)$ is sound, $\rho \models \mathbf{C}$. Thus $\rho \not\models compat(\Pi\Theta(\mathcal{S})) \mid_E$, as required. ∎

**Theorem 6.2.6 [Soundness and Completeness of $\vdash^E_{\Psi\Theta}$ and $=^E_{\Psi\Theta}$]**

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

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

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

$$\begin{array}{lll}
 & \mathcal{S}_1 \models_E \Delta(\mathcal{S}_2) & \text{by the soundness of } \Delta \\
\therefore & \mathcal{S}_1 \models_E \Delta(\mathcal{S}_2) \mid_E & \\
\therefore & \forall \mathcal{C} \in \Delta(\mathcal{S}_2) \mid_E .\ \ \mathcal{S}_1 \models \mathcal{C} & \\
\therefore & \forall \mathcal{C} \in \Delta(\mathcal{S}_2) \mid_E .\ \ \mathcal{S}_1 \vdash_\Delta \mathcal{C} & \text{by Lemma 6.2.1} \\
\therefore & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2) \mid_E .\ \ \mathcal{S}_1 \vdash_\Delta \mathcal{C} & \text{by Lemma 6.2.3} \\
\therefore & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2) \mid_E .\ \ \mathcal{S}_1 \vdash_{\Psi\Theta} \mathcal{C} & \text{by Lemma 6.2.3} \\
\therefore & \forall \mathcal{C} \in \Pi\Theta(\mathcal{S}_2) \mid_E .\ \ \mathcal{C} \in \Psi\Theta(\mathcal{S}_1) \mid_E & \\
\therefore & \Psi\Theta(\mathcal{S}_1) \mid_E \supseteq \Pi\Theta(\mathcal{S}_2) \mid_E & \\
\therefore & \mathcal{S}_1 \vdash^E_{\Psi\Theta} \mathcal{S}_2 &
\end{array}$$

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

$$\begin{array}{ll}
 & \Psi\Theta(\mathcal{S}_1) \mid_E \supseteq \Pi\Theta(\mathcal{S}_2) \mid_E \\
\therefore & Soln(\Psi\Theta(\mathcal{S}_1) \mid_E) \subseteq Soln(\Pi\Theta(\mathcal{S}_2) \mid_E) \\
\therefore & Soln(\Psi\Theta(\mathcal{S}_1) \mid_E) \mid_E \subseteq Soln(\Pi\Theta(\mathcal{S}_2) \mid_E) \mid_E \\
\therefore & Soln(\mathcal{S}_1) \mid_E \subseteq Soln(\mathcal{S}_2) \mid_E \\
 & \text{by Lemmas 6.2.2, 6.2.3 and 6.2.4, since } Soln(\mathcal{S}_i) \mid_E = Soln(\Pi\Theta(\mathcal{S}_i) \mid_E) \mid_E \\
\therefore & \mathcal{S}_1 \models_E \mathcal{S}_2
\end{array}$$

2. Follows from part 1.

## C.3 Proofs for Deciding Observable Equivalence

We repeat definition 6.3.1 here, to avoid having to refer back to the original definition earlier in the text.

**Definition 6.3.1** (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:

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

∎

**Lemma 6.3.2** *Let $G = G_r(\mathcal{S}, E)$. Then:*

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

**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 $\mathbf{C} = [\alpha \leq \texttt{rng}(\kappa)]$ is inferred via $(compose_1)$ from the antecedents $[\alpha \leq \texttt{rng}(\beta)]$ and $[\beta \leq \kappa]$. Then $\alpha_U \mapsto \texttt{rng}(\beta_U)$, and by induction $\beta_U \mapsto^* \kappa$. Hence $\alpha_U \mapsto \texttt{rng}(\kappa)$, 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 6.3.3 here, to avoid having to refer back to the original definition earlier in the text.

**Definition 6.3.3** (Regular Tree Grammar $G_t(\mathcal{S}, E)$) The regular tree grammar $G_t(\mathcal{S}, E)$ extends the grammar $G_r(\mathcal{S}, E)$ with the root non-terminal $R$ and the additional productions:

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

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

**Lemma 6.3.4** *Let* $G = G_t(\mathcal{S}, E)$*. Then* $\Pi(\mathcal{S})\ |_E = \mathcal{L}_G(R)$*.*

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

- Suppose $\mathbf{C} = [\alpha \leq \kappa]$. Then by Lemma 6.3.2, $\alpha_U \mapsto^*_G \kappa$. Since $\mathit{SetVar}(\mathbf{C}) \subseteq E$, $\alpha \in E$, 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 $\Pi$ shows that $\mathbf{C}$ can only be inferred via $(\mathit{trans}_\alpha)$, based on the antecedents $[c \leq \alpha]$ and $[\alpha \leq \kappa]$. By Lemma 6.3.2, $\alpha_L \mapsto^*_G c$ 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 $\kappa_1, \kappa_2 \notin \mathit{SetVar}$. An examination of the inference rules in $\Pi$ shows that $\mathbf{C}$ can only be inferred via $(\mathit{trans}_\alpha)$, based on the antecedents $[\kappa_1 \leq \alpha]$ and $[\alpha \leq \kappa_2]$. By Lemma 6.3.2, $\alpha_L \mapsto^*_G \kappa_1$ 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 $\alpha$, $\alpha_L \mapsto^*_G \kappa_1$ and $\alpha_U \mapsto^*_G \kappa_2$. By Lemma 6.3.2, $\{\kappa_1 \leq \alpha,\ \alpha \leq \kappa_2\} \subseteq \Psi(\mathcal{S})$ and $\mathit{SetVar}(\kappa_i) \subseteq E$. By Lemma 6.3.5, $\{\kappa_1 \leq \alpha,\ \alpha \leq \kappa_2\} \subseteq \Pi(\mathcal{S})$. Hence $[\kappa_1 \leq \kappa_2] \in \Pi(\mathcal{S})\ |_E$, as required.

- Otherwise $\mathbf{C} = [c \leq \kappa]$. Then for some $\alpha$, $[c \leq \alpha] \in \mathcal{S}$ and $\alpha_U \mapsto^*_G \kappa$. By Lemma 6.3.2, $\{\alpha \leq \kappa\} \subseteq \Psi(\mathcal{S})$ and $\mathit{SetVar}(\kappa) \subseteq E$. By Lemma 6.3.5, $\{\alpha \leq \kappa\} \subseteq \Pi(\mathcal{S})$. Hence $[c \leq \kappa] \in \Pi(\mathcal{S})\mid_E$, as required.

∎

**Lemma 6.3.5 (Staging)** *For any simple constraint system $\mathcal{S}$:*

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

**Proof:** The equality $\Psi(\Theta(\mathcal{S})) = \mathit{compat}(\Pi(\Theta(\mathcal{S})))$ holds since $(\mathit{compat})$ does not create any $\Pi$ or $\Theta$ 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 $\Psi$. 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 $(\mathit{trans}_\alpha)$, $(\mathit{compose}_1)$ and $(\mathit{compose}_4)$, 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 $\Psi$ shows that $\Psi$ can only infer $[\alpha \leq \mathtt{rng}(\beta)]$ if there exists $\alpha', \beta'$ such that $\Theta(\mathcal{S})$ contains the constraints:
$$\alpha \leq^* \alpha' \qquad \alpha' \leq \mathtt{rng}(\beta') \qquad \beta' \leq^* \beta$$
Similarly, $\Psi$ can only infer $[\mathtt{rng}(\beta) \leq \gamma]$ if there exists $\beta'', \gamma'$ 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 $(\mathit{trans}_\alpha)$ |

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

∎

## C.4 Correctness of the Entailment Algorithm

**Theorem 6.3.6** *(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:** The definitions of the computable entailment relation and the relation $\mathcal{R}$ are shown in figure C.1. We prove this theorem based on the following invariant concerning the relation $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\cdot,\cdot,\cdot,\cdot]$:

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

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 6.3.4, $\Pi(\mathcal{S}_i)\mid_E = \mathcal{L}_{G_i}(R)$, and hence:

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

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

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

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 $R \mapsto_{G_1} [c \le \alpha_U]$:

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

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:

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

**The Entailment Algorithm**

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

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

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

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

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

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

such that if:

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

then one of the following cases hold:

1. $\mathcal{L}_{G_1}([X \leq Y]) \subseteq \mathcal{L}_{G_2}(C \cup D)$.
2. $X = \mathtt{rng}(\alpha'_L)$, $Y = \mathtt{rng}(\beta'_U)$ and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha'_L, \beta'_U, C, D']$, where
$$D' = \{\langle \gamma'_L, \delta'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \mathtt{rng}(\delta'_U)\}$$
3. $X = \mathtt{dom}(\alpha'_U)$, $Y = \mathtt{dom}(\beta'_L)$ and $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\beta'_L, \alpha'_U, C, D']$, where
$$D' = \{\langle \delta'_L, \gamma'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \mathtt{dom}(\gamma'_U), \delta_U \mapsto_{G_2} \mathtt{dom}(\delta'_L)\}$$
4. In no other cases does $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \beta_U, C, D]$ hold.

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

1. $\forall R \mapsto_{G_1} [\alpha_L \leq \alpha_U]$. $\mathcal{R}_{\mathcal{S}_1,\mathcal{S}_2}[\alpha_L, \alpha_U, \{\langle \gamma_L, \gamma_U \rangle \mid \gamma \in \mathit{SetVar}(\mathcal{S}_2)\}, \emptyset]$, and
2. $\forall R \mapsto_{G_1} [c \leq \alpha_U]$. $\mathcal{L}_{G_1}(\alpha_U) \subseteq \mathcal{L}_{G_2}(\{\gamma_U \mid R \mapsto_{G_2} [c \leq \gamma_U]\})$.

**Figure C.1** The computable entailment relation $\vdash^E_{\text{alg}}$

We prove by induction on $\tau_L$ 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 C.1 hold. Furthermore, since $\mathcal{R}$ is the largest relation satisfying the conditions in figure C.1, 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 $X = \texttt{rng}(\alpha'_L)$ and $Y = \texttt{rng}(\beta'_U)$. Consider any pair of set expressions $\tau_L$ and $\tau_U$ such that $\alpha'_L \mapsto^*_{G_1} \tau_L$ and $\beta'_U \mapsto^*_{G_1} \tau_U$. We consider the two possibilities for $[\texttt{rng}(\tau_L) \leq \texttt{rng}(\tau_U)] \in \mathit{compat}(\mathcal{L}(C)) \cup \mathcal{L}(D)$ separately.

  - If $[\texttt{rng}(\tau_L) \leq \texttt{rng}(\tau_U)] \in \mathcal{L}(C) \cup \mathcal{L}(D)$, then there exists $\langle \gamma_L, \delta_U \rangle \in C \cup D$ such that:
  $$\begin{array}{l} \gamma_L \mapsto_{G_2} \texttt{rng}(\gamma'_L) \mapsto^*_{G_2} \texttt{rng}(\tau_L) \\ \delta_U \mapsto_{G_2} \texttt{rng}(\delta'_U) \mapsto^*_{G_2} \texttt{rng}(\tau_U) \end{array}$$
  Hence $[\tau_L \leq \tau_U] \in \mathcal{L}(D')$, where:
  $$D' = \{\langle \gamma'_L, \delta'_U \rangle \mid \langle \gamma_L, \delta_U \rangle \in C \cup D, \gamma_L \mapsto_{G_2} \texttt{rng}(\gamma'_L), \delta_U \mapsto_{G_2} \texttt{rng}(\delta'_U)\}$$

  - Otherwise $[\texttt{rng}(\tau_L) \leq \texttt{rng}(\tau_U)] \in \mathit{compat}(\mathcal{L}(C)) \setminus \mathcal{L}(C)$, 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. ∎

## C.5 Correctness of the Hopcroft Algorithm

**Theorem 6.4.1** *(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 6.5; 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 $\rho$ be a solution of $\mathcal{S}$. Define $\rho'$ by:

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

Obviously $\rho, \rho'$ 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 $\mathcal{C} = [\alpha \leq \texttt{rng}(\beta)]$. Then for all $\alpha'$ such that $\alpha \sim \alpha'$ there exists $\beta'$ such that $\beta \sim \beta'$ and:

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

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

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

  and therefore:

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

  Hence:

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

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

- The cases where $\mathcal{C} = [\alpha \leq \beta]$ and $\mathcal{C} = [\texttt{rng}(\alpha) \leq \beta]$ follow by similar reasoning.

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

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

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

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

and therefore:

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

Hence:

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

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

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

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

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

∎

# Appendix D

# MrSpidey Reference Manual

MrSpidey is an interactive, static debugger for Scheme designed to help programmers understand and debug complex programs. It automatically infers information about the run-time behavior of programs, and uses this information to identify potential "danger-points" in those programs. Specifically, MrSpidey:

- infers a *type*, or value set invariant, describing the set of possible values for each program expression;
- uses this information to identify *unsafe* program operations that may cause run-time errors; and
- provides a supporting graphical explanation for these invariants.

MrSpidey supports almost all of DrScheme, which is an extension of R4RS Scheme with structures, a module system, an object system, and a GUI toolbox. For further information on the technology underlying MrSpidey, see [14, 15, 16, 13].

**Thanks:** Many thanks to both Matthew Flatt and Robby Findler for MrEd, and to Shriram Krishnamurthi for Zodiac, his source-correlating macro-expander. MrSpidey crucially depends on both of these packages. Thanks also to Stephanie Weirich for work on the first implementation of MrSpidey, and to Matthias Felleisen, Corky Cartwright, Jeremy Buhler, and the Rice University Spring '96 COMP311 programming languages class for their feedback and help.

The typesetting sources for this manual are taken from *Reference Manual for wxWindows 1.60: a portable C++ GUI toolkit.*

## D.1 Using MrSpidey

MrSpidey is an integrated portion of DrScheme. To analyze the current program in DrScheme, click on DrScheme's Analyze button. MrSpidey analyzes the program and displays the analysis results in a new frame. This frame contains two sub-windows: a program window and a summary window. The display of these windows is controlled via the Show menu.

### D.1.1 The Program Window

The program window contains an annotated version of original program. The additional annotations present information about the results of the analysis, as described below.

#### Unsafe Operations

An *unsafe* operation is one which may be applied to inappropriate arguments, thus raising an error. Unsafe operations are a natural starting point for static debugging. MrSpidey highlights these unsafe operations via font and color changes, as follows:

- Any primitive operation that may be applied to inappropriate arguments, thus raising a run-time error, is highlighted in red (or underlined on monochrome screens).* Conversely, primitive operations that never raise errors are shown in green.

- Any function that may be applied to an incorrect number of arguments is highlighted by displaying the `lambda` keyword in red (or underlined).

- Any application expression where the function position may return a non-function is highlighted by displaying the enclosing parentheses in red (or underlined).

Figure D.1 contains examples of these three different kinds of unsafe operations. The tab key moves the focus forward to the next unsafe operation, and the shift-tab key moves the focus backward to the previous unsafe operation. These keystrokes make it easy to inspect the unsafe operations in a program.

---

*Certain unsafe operations, such as a **vector-ref** operation whose index argument may be out-of-range, are not detected. A list of such operations is contained in section D.6.

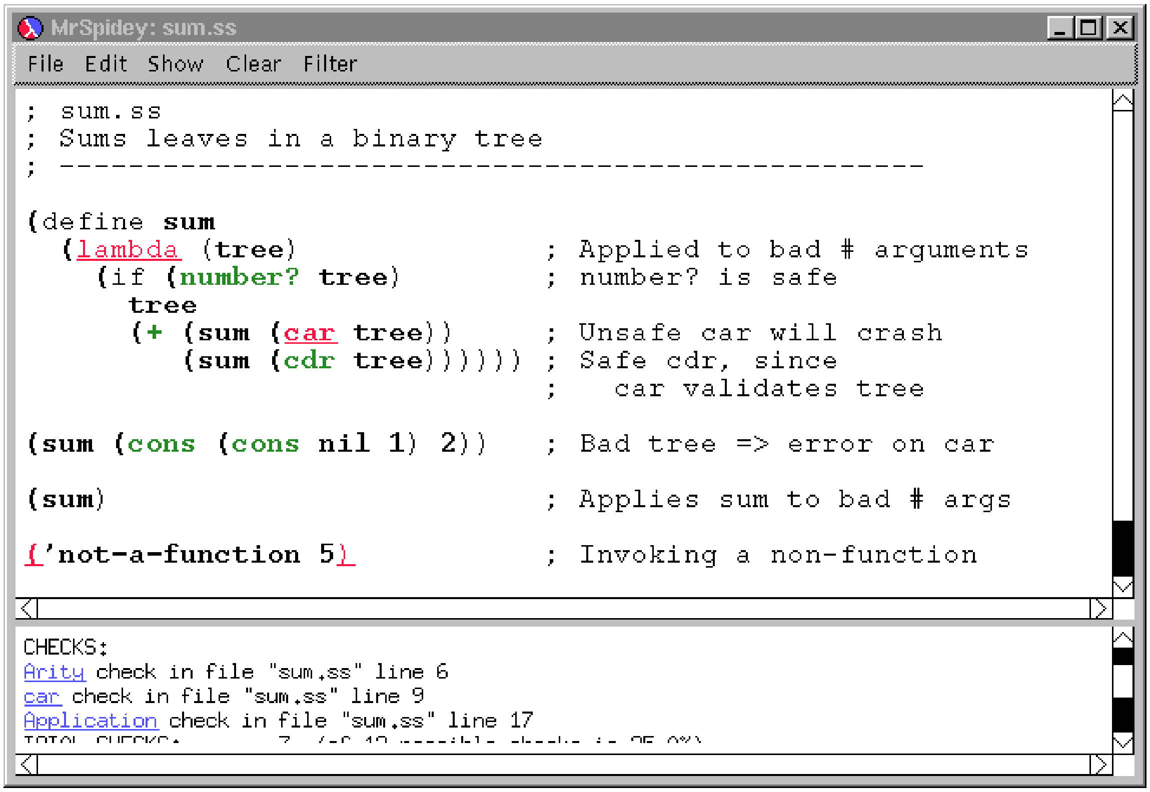


**Figure D.1** Identifying unsafe operations

### Popup Menus

MrSpidey also computes significant additional information for each analyzed expression. This information is available on a demand-driven basis via pop-up menus. MrSpidey associates a pop-up menu with all program variables, which are marked in bold, and also with the opening parenthesis of each expression, which is also marked in bold: see figure 5.2. Clicking on one of these bold tokens displays the associated menu, which then provides access to additional type and value flow information, as described below.

### Type Information

MrSpidey provides an inferred type for each program expression. To view the type of an expression, select the Show Value Set option of the expression's menu. Mrspidey then computes the expression's type and displays it in a box inserted to the right of

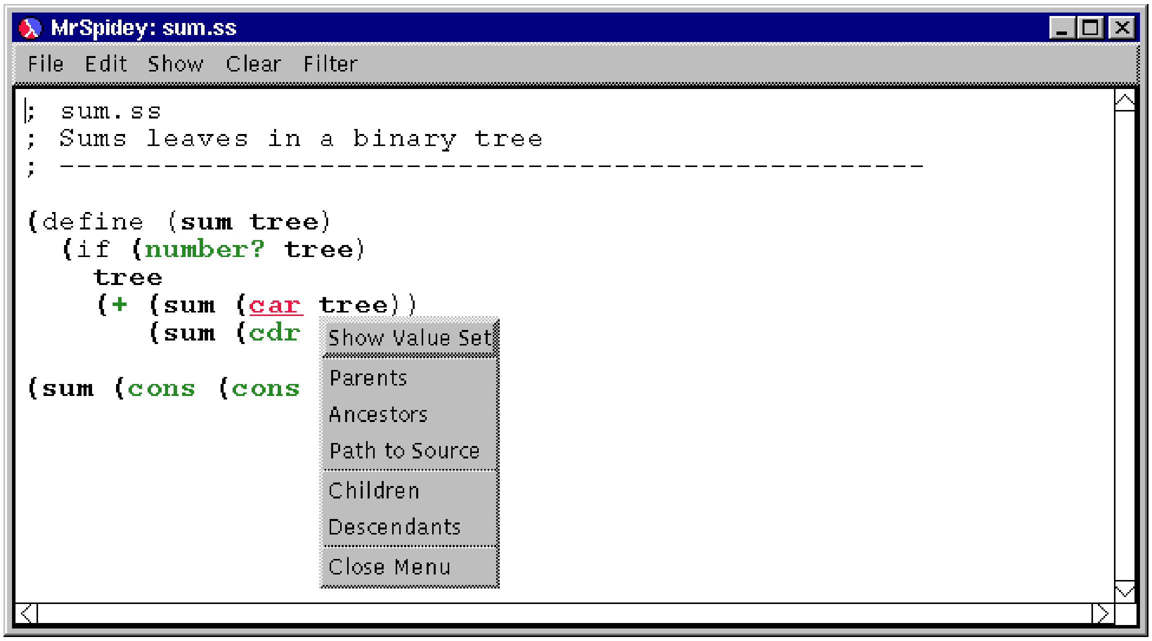


**Figure D.2** The pop-up menu

the expression, as illustrated in figure D.3. See Section D.4 for a complete description of the type language. The type box is deleted by selecting the Close Value Set option from the popup menu. Alternatively, selecting Clear | Types deletes all type boxes in the buffer.

### The Value Flow Browser

MrSpidey can also explain the derivation of each value set invariant, or type. This explanation describes how the flow of values through the program yields a particular value set invariant. The collection of all potential paths along which value may flow through the program forms the program's *value flow graph*. MrSpidey describes each edge in the data-flow graph as an arrow overlaid on the program text that connects the relevant points of the program. Because a large number of arrows would clutter the program text, these arrows are presented in a demand-driven fashion. Each expression's popup menu provides facilities for inspecting relevant portions of the value flow graph.

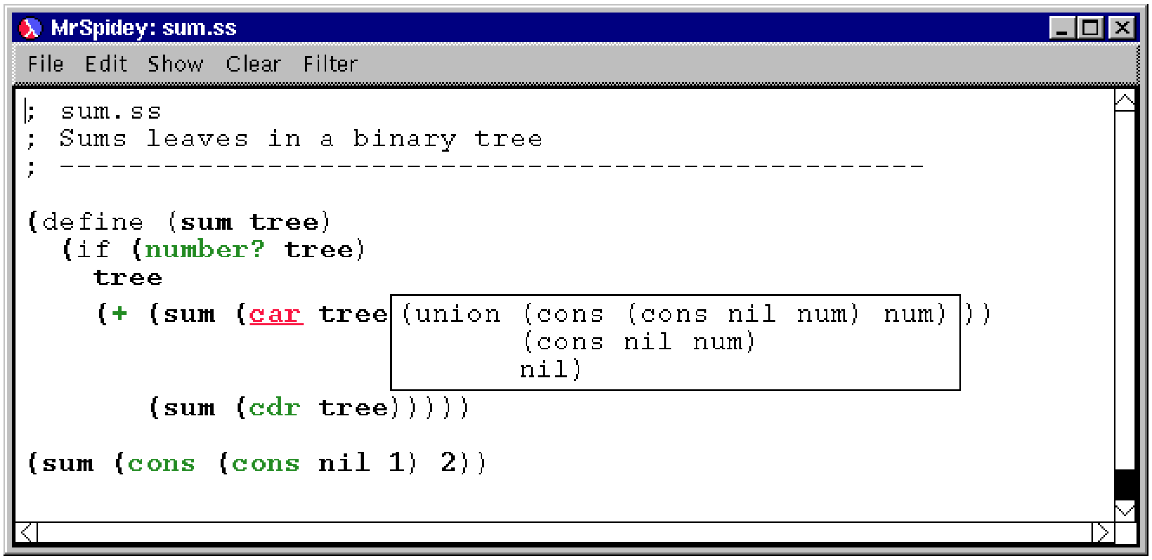


**Figure D.3** Displaying type information

- Parents: exposes arrows indicating the immediate parents of the current expression in the value flow graph. For example, figure D.4 shows the incoming edges for the parameter `tree` in the program `sum`.
- Ancestors: exposes all ancestors of the current expression in the value flow graph.
- Path to Source: finds the shortest path in the value flow graph from a constructor expression to the current expression.
- Children: exposes the immediate children of the current expression in the value flow graph.
- Descendants: exposes all descendants of the current expression in the value flow graph.

All of the above options can be customized to show only the flow of certain values by selecting the appropriate set of values from the Filter menu. This option is particularly useful in conjunction with the Path to Source facility for inspecting the flow of an unexpected value through the program. For example, in the program `sum`, the unexpected value for `tree` is `nil`. Setting the filter to this value, and then selecting Path to Source for `tree` results in the explanation described in figure D.5.

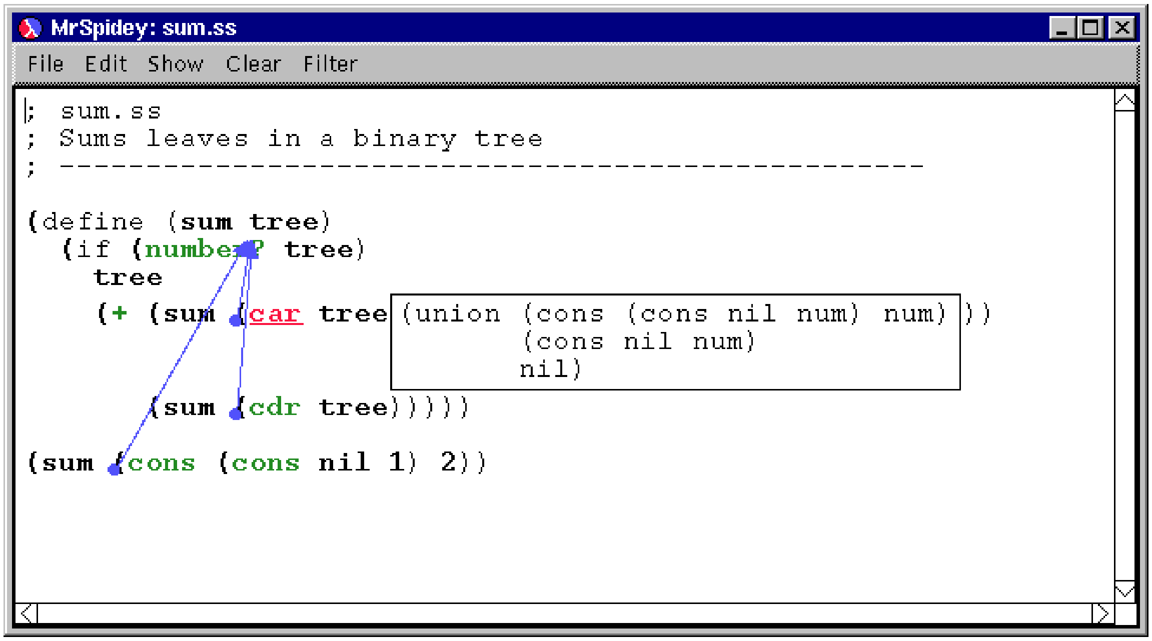


**Figure D.4** Parents of `tree`

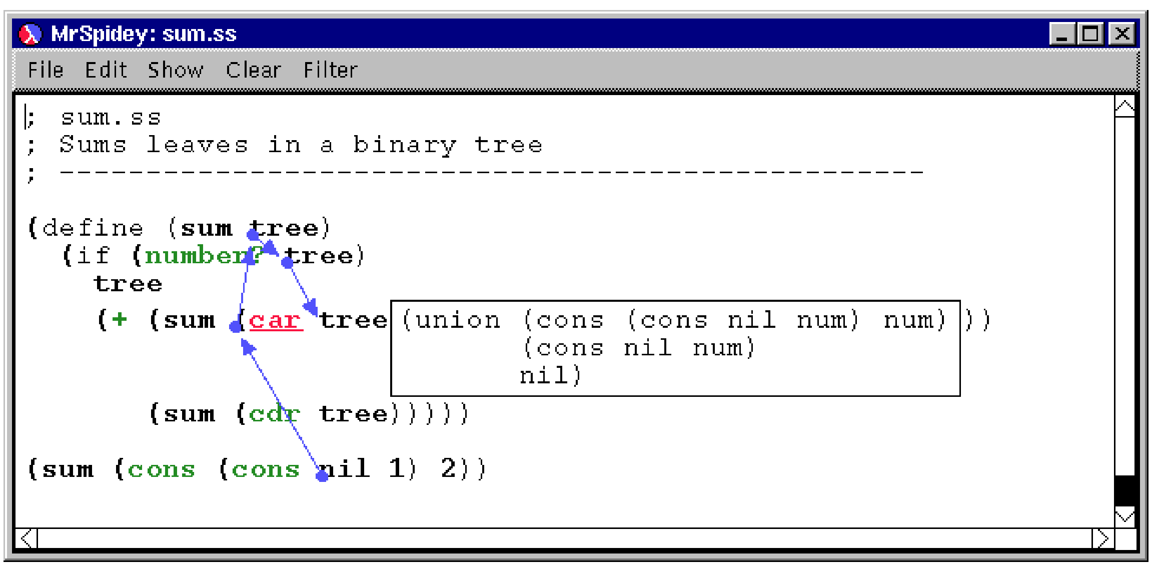


**Figure D.5** Flow of `nil`

Clicking on the head or tail of an arrow with the left mouse button moves the current focus to the term at the other end of the arrow, which can be useful for following the flow of values through large programs. Clicking on the head or tail of an arrow with the right mouse button deletes the arrow. Alternatively, selecting Clear|Arrows deletes all arrows in the buffer.

### D.1.2 The Summary Window

The summary window lists all unsafe operations in the program, together with hyperlinks to those operation. It counts the number of unsafe operations, and expresses that number as a percentage of the total number of operations in the program. This window also contains warning about unbound variables and failed type assertions. A typical summary window is show in figure D.1.

## D.2 Preferences

The Edit|Preferences ... menu item menu allows users to configure a variety of DrScheme options. Two of the preferences windows, MrSpidey Analysis and MrSpidey Type Display, control aspects of MrSpidey's behavior. (These windows are only available after MrSpidey is loaded.)

### D.2.1 MrSpidey Analysis Preferences Window

The MrSpidey Analysis preferences window configures MrSpidey's analysis of programs. An example of this window is shown in figure D.6, and the controls are described below.

- Accurate constant types: When this button is off (default), then character, symbolic and numeric constants are given the types `char`, `sym` or `num` respectively. If the button is on, then these constants are typed accurately, *i.e.*, the number 4 is assigned the type 4, etc.

- Constant merge size: Large quoted values in Scheme can yield large types that significantly increase the analysis time. To overcome this problem, MrSpidey generates *approximate* types for such constants. The Constant merge size slider controls how large constants can get before MrSpidey starts to approximate their type.

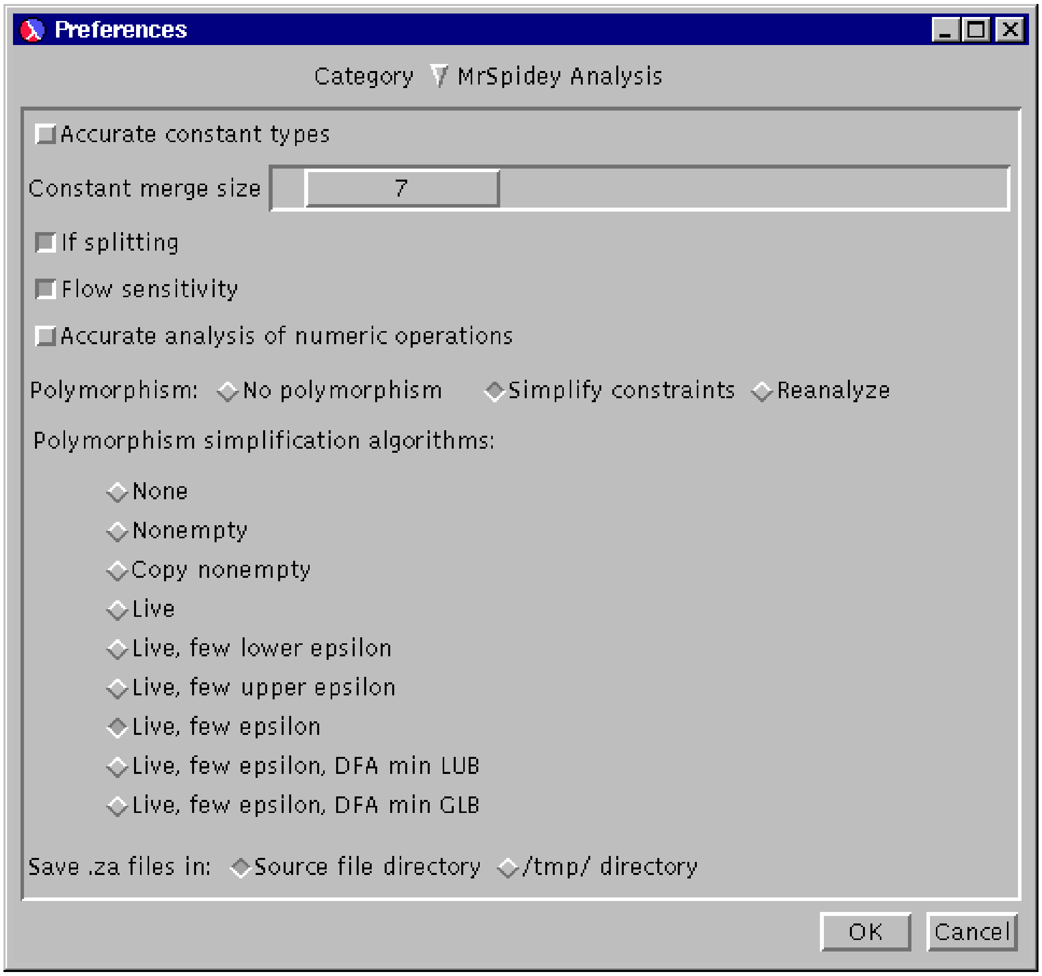


**Figure D.6** MrSpidey Analysis preferences window

- If splitting: If this button is on (default), MrSpidey is smart about conditional expressions such as (**if** (*number? x*) ...). MrSpidey only propagates numeric values for `x` in the then-part, and non-numbers in the else-part.

- Flow sensitivity: If this button is on (default), then after an expression such as `(car x)`, MrSpidey knows that the value of `x` must be a pair.

- Accurate analysis of numeric operations: When this button is off (default), then numeric operations such as `+` simply return the type `num`. If the button is on, then the numeric operations are typed more accurately, as described in subsection D.4.1.

- Polymorphism: This radio box controls how `polymorphic` expressions are analyzed.
  - No polymorphism: `polymorphic` annotation is ignored;
  - Simplify constraints: The constraint system for the polymorphic expression is simplified (see below); and
  - Reanalyze: The polymorphic expression is re-analyzed for each polymorphic reference.
- Polymorphism simplification algorithms: This radio box controls which constraint simplification algorithm is used to simplify the constraint system of polymorphic expression, provided the Polymorphism control is set to Simplify constraints. These constraint simplification algorithms are described in section 6.4.
- Save .za files in: MrSpidey generates and saves constraint (`.za`) files during the analysis of multi-file programs. This radio box controls where these constraint files are stored, either in the same directory as the corresponding source file (default), or in a temporary directory.

### D.2.2 MrSpidey Type Display **Preferences Window**

The MrSpidey Type Display preference window controls how MrSpidey computes and displays type information. An example of this window is shown in figure D.7, and the controls are described below.

- Show types as: Types can be displayed either as basic types (default), which just show the range of functions, or as type schemas, which show how the domain and range of a function are related (*e.g.* `(X1 -> X1)`). Basic types contain less information, but are more compact, which is an important benefit when working with large programs. With basic types, users can also choose whether or not to show instance variables and structure fields. For large programs, it is often best not to show these components, in order to produce reasonably compact types.
- Constraint simplification algorithms: This radio box controls which constraint simplification algorithm is used to simplify the constraint system of an expression, before converting that constraint system to a type. These constraint simplification algorithms are described in section 6.4.

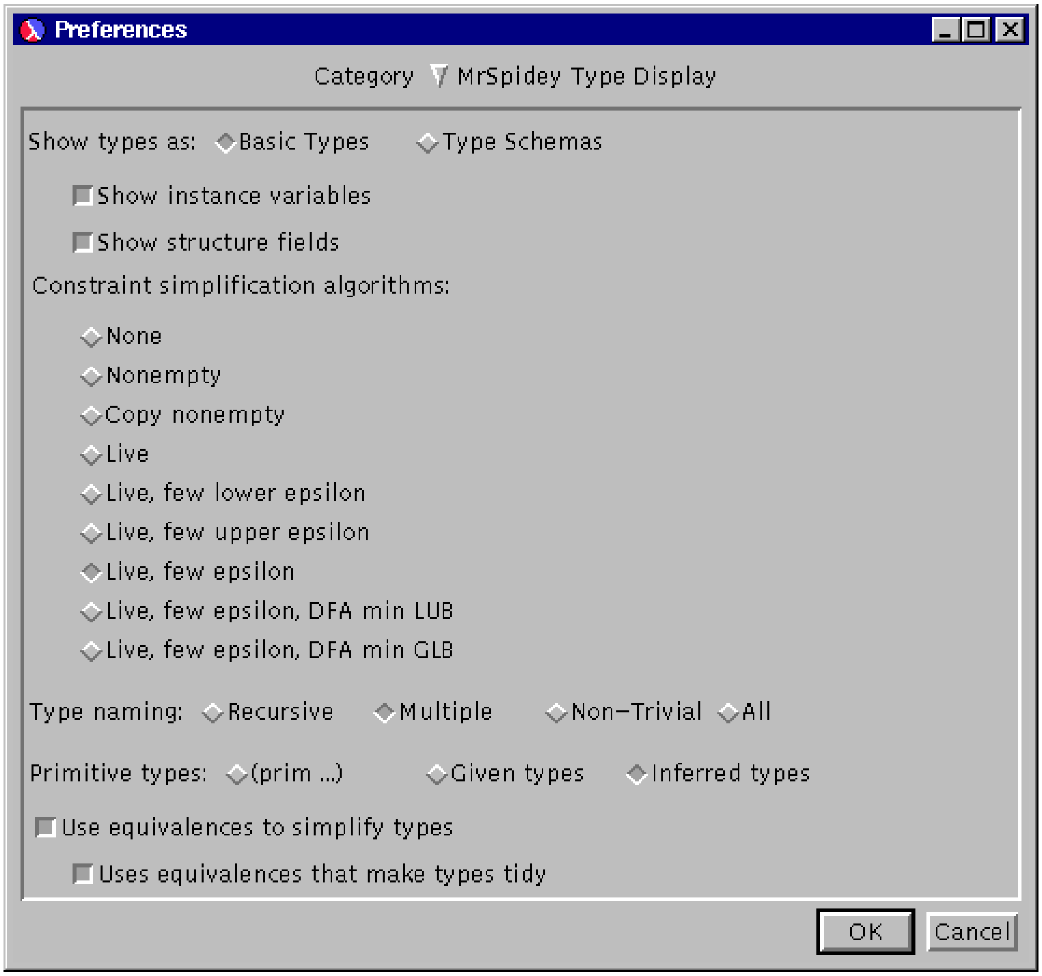


**Figure D.7** MrSpidey Type Display preferences window

- Type naming: This parameter controls what types are named in a `rec` type expression, and can be:
  - Recursive: Names just enough types to express recursive types.
  - Multiple: (default) Names every type that is referred to more than once
  - Non-Trivial: Names every type except trivial types such as `num`, `sym`, etc.
  - All: Names all types.
- Primitive types: This parameter can be:
  - (prim ...): Displays primitive types as `(prim car)` etc.

- Given types: Displays the given types of primitives, *e.g.*, `((cons a b) -> a)`.
- Inferred types: (default) Displays the inferred domain and range of primitive functions, *e.g.*: `((cons num 4) -> num)`.

- Use equivalences to simplify types: If this control is on (default), then a number of rewriting rules are used to simplify types before they are displayed.

- Use equivalences that make types tidy: If this control is on (default), then some of the type rewriting rules will merge types into disjoint unions, thus losing a certain amount of type information in order to produce a more compact type.

## D.3 Analysis of Large Programs

Large programs in DrScheme are typically split into multiple source files, where each source file contains a `unit` (or `unit/sig`) expression. The main file for the program then refers to each source file via the `reference-unit` (or `reference-unit/sig`) form, and links these multiple units together into a single compound-unit that is then invoked.

To provide a quick turn-around time when statically debugging such programs, MrSpidey uses a *componential* analysis to avoid re-analyzing source files where possible. When each referenced unit file is first analyzed, MrSpidey:

1. derives a constraint system that describes the data-flow behavior of the unit;

2. simplifies the constraint system while preserving the externally-visible information about the unit's data-flow behavior; and

3. saves the simplified constraint system in a *constraint file* named *file*`.za` (where *file*`.ss` is the name of the source file).

Although the use of constraint files does not reduce the time required for the first analysis, on subsequent analyses MrSpidey can use the saved constraint files to avoid the re-analysis of referenced unit files that have not been modified. This approach substantially reduces re-analysis times.

Once the analysis is completed, MrSpidey displays the program's main file with the usual static debugging mark-ups. To view other source files, select the File|Open ... option from the MrSpidey window. This option displays a dialog box containing all the program's source files, and allows the programmer to select the file of interest. A typical dialog box is contained in figure D.8. Alternatively:

- The File|Open All option opens a MrSpidey windows for each source file; and
- The File|Load All option loads all source files into memory but does not immediately display them.

The File|Close All option closes all the MrSpidey windows.

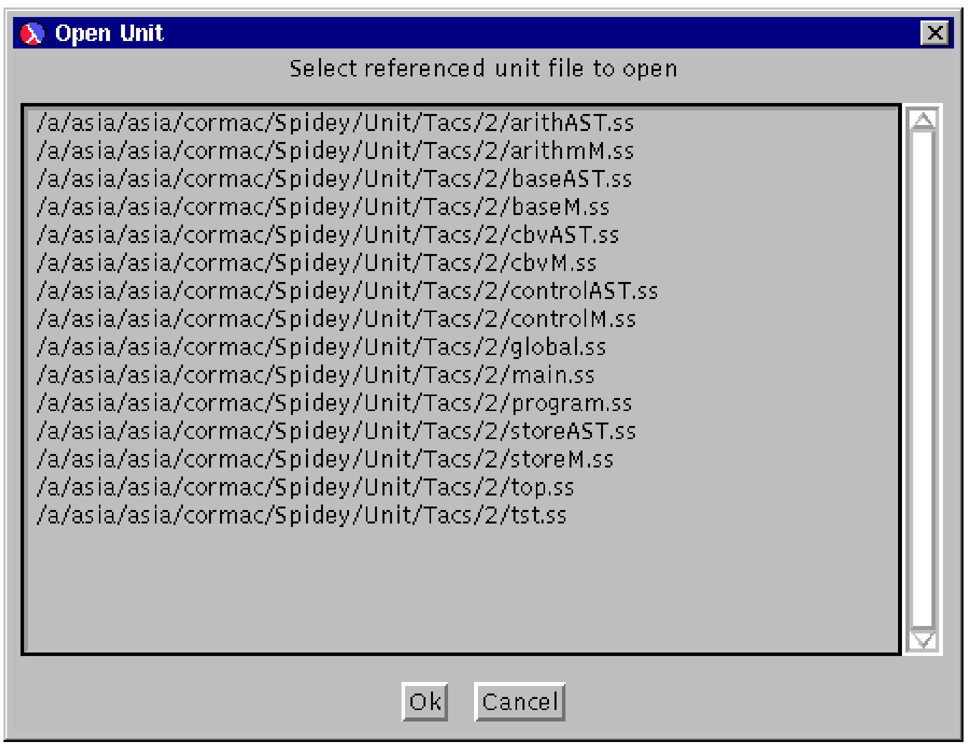


**Figure D.8** The File|Open ... dialog box

### D.3.1 Inter-File Arrows

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: see figure D.9. If a *margin arrow* is painted red, then it refers to an expression in a file that has not yet been loaded, and clicking on the arrow provides the option to load the file. A blue margin arrow refers to an expression in a file that has been loaded. Clicking on a blue arrow provides the option to zoom to and highlight the term at the other end of the arrow, as shown in figure D.10. These facilities are useful for following the flow of values through multi-file programs.

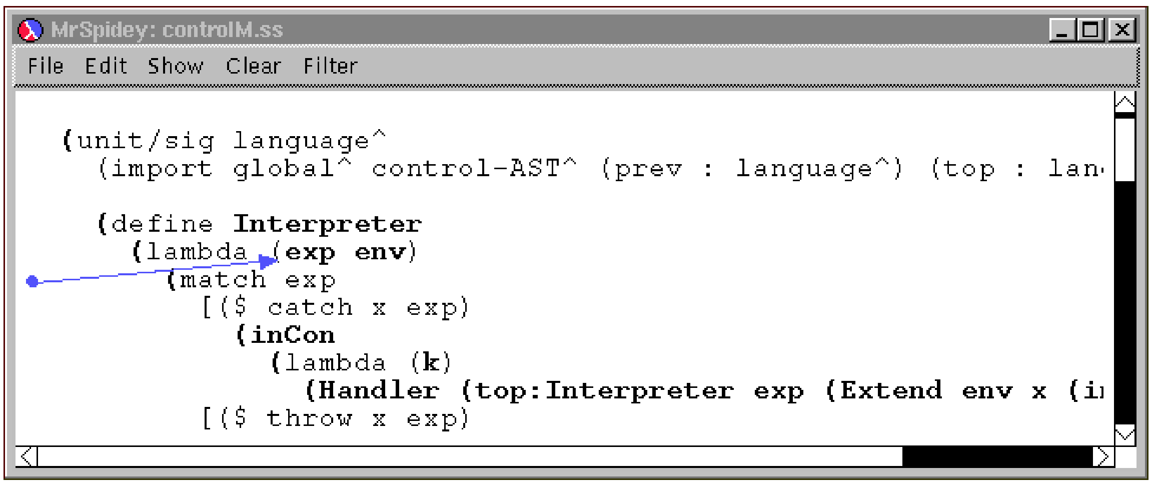


**Figure D.9** Source in another loaded file

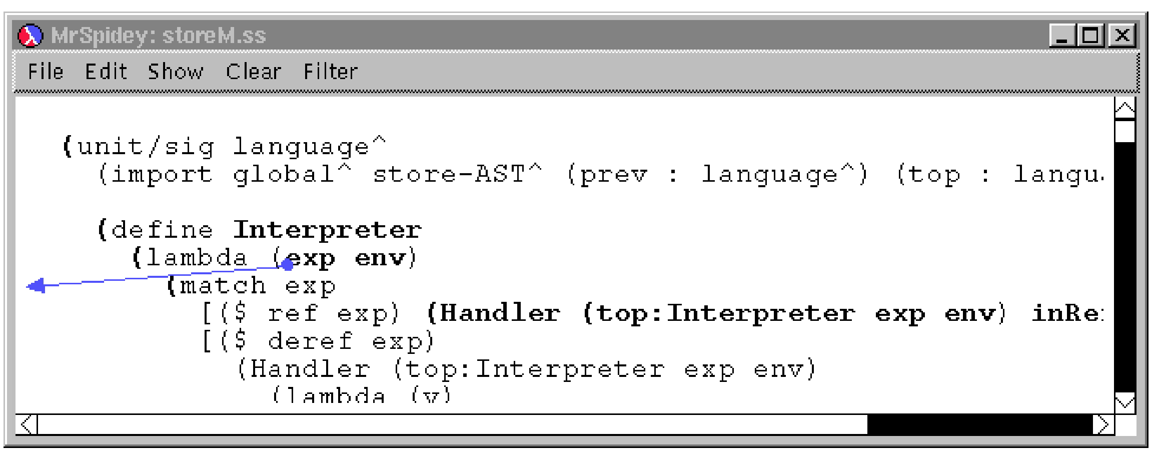


**Figure D.10** The highlighted source in the other file

## D.4 The Type Language

The language of basic types in MrSpidey is defined as follows:

*type* is one of:
- *set-variable*
- `empty`
- *zeroary-constructor*
- (*constructor type* ⋯ *type*)
- `(union` *type* ⋯ *type*)
- `(rec ([`*set-variable type*`]` ⋯) *type*)
- `(class [`*ivar type*`]` ⋯)
- `(object [`*ivar type*`]` ⋯)
- *function-type*
- *type-abbreviation*

*set-variable* is:
- *identifier*

*zeroary-constructor* is one of:
- `nil num sym str char void true false bool eof`

*constructor* is one of:
- *zeroary-constructor*
- `vec box promise mvalues iport oport` — `Unary constructors`
- `cons` — `Binary constructors`
- *define-structure-constructor*
- *user-defined-constructor*

*define-structure-constructor* is:
- *identifier*

*user-defined-constructor* is:
- *identifier*

*function-type* is one of:

| | |
|---|---|
| (*type* ⋯ *type* `->` *type*) | |
| (*type* ⋯ *type* `*->` *type*) | `Rest argument` |
| (*type* ⋯ *type* `->*` *type*) | `Return value list` |
| (*type* ⋯ *type* `*->*` *type*) | `Rest argument and return value list` |
| (`lambda` *type* *type*) | `same as` (*type* `*->*` *type*) |

*type-abbreviation* is one of:

| | Abbreviates |
|---|---|
| (`MU` *set-variable* *type*) | (`rec` ([*set-variable* *type*]) *set-variable*) |
| `noarg` | `nil` |
| (`arg` *type* *type*) | (`cons` *type* *type*) |
| (`list` *type* ⋯ *type*) | (`cons` *type* (`cons` ⋯(`cons` *type* `nil`))) |
| (`listof` *type*) | (`MU l (union (cons` *type* `l) nil)`) |
| `null` | `nil` |
| `bool` | `(union true false)` |
| `atom` | `(union nil num sym str char bool)` |
| `sexp` | `(MU x (union atom (cons x x) (vec x)))` |

The behavior of primitive operations is defined using *multiple-arity schemas*. For each reference to a primitive operation, MrSpidey retrieves the corresponding multiple-arity schema and selects the schema appropriate for the number of arguments given to the primitive (or the last schema if the primitive is used in a higher-order manner). It then instantiates the schema by replacing the quantified set variables by set variables, and converts the resulting basic type into a constraint system. Multiple-arity schemas are defined as follows:

*multiple-arity-schema* is one of:

*schema*

(`case->` *schema* ⋯)

*schema* is one of:

*type*

(`forall` (*set-variable* $\cdots$) *type*)

### D.4.1 Accurate Numeric Operations

When the Accurate numeric operations control in the MrSpidey Analysis preferences window is turned on, MrSpidey performs a more accurate analysis of numeric operations, as follows.

The type language is extended with the unary constructors `apply+`, `apply-`, `apply*` and `apply/`. The return type of the numeric operations `+`, `-`, `*`, and `/` record information about the numeric operation and its argument value sets. For example, the type returned by the operation `+` is `(apply+` *arglist*`)`, where *arglist* is the argument list to `+`. The resulting types are simplified before being presented to the programmer. For example, type `(apply+ (list x1 ... xn))` is transformed into `(+ x1 ... xn)`, etc.

In addition, the binary constructors `=`, `not=`, `<`, `<=`, `>` and `>=` are added to the type language. The meaning of the type `(< X Y)` is the set of numbers $x$ in `X` such that there exists some $y$ in `Y` with $x < y$. MrSpidey generates these types for `if`-expressions where the predicate is one of `zero?`, `=`, `<`, `<=`, `>` or `>=`.

## D.5 Extensions to DrScheme

### D.5.1 Type Assertions

The form (: *exp type*) is an assertion that the values produced by *exp* must be contained in *type*. If MrSpidey is unable to prove that the type assertion is satisfied, then a warning is reported in the summary window. These : forms evaluate to void.

### D.5.2 Polymorphic Annotations

The form (**polymorphic** *exp*) causes the expression *exp* to be analyzed in a polymorphic manner. That is, if the result of (**polymorphic** *exp*) is immediately bound to an identifier (*e.g.* by **let** or **define**), then all references to that identifier that occur below that binding will be polymorphic. The annotation has no runtime effect.

### D.5.3 Declaring New Primitives

The form (**type:** *multiple-arity-schema*) declares new primitive of type *multiple-arity-schema*. Some example definitions are:

(**define** *my-car* (**type:** (*forall* (*a*) ((*cons a _*) -> *a*))))

(**define** *my-map* (**type:** (*forall* (*a r*) ((*a* -> *r*) (*listof a*) -> (*listof r*)))))

The expression (**type:** ...) evaluates to void.

### D.5.4 Declaring Constructors

The form (**define-constructor** *name modes* ...) adds a new type constructor to the type language. The arguments *mode* ... are all booleans, each specifying whether the corresponding field in the constructor is mutable.

### D.5.5 Declaring New Types

The form (**define-type** *name type*) adds a new type *name* that can later be used in type expressions.

## D.6 Restrictions on Source Programs

The following DrScheme facilities are not handled.

- Primitives for dynamic loading and evaluation: `load`, `load/cd`, `load-relative`, `load/use-compiled`, `current-load`, `require-library-use-compiled`, `compile`, `eval`, `current-eval`, `expand-defmacro` and `expand-defmacro-once`.
- Primitives that dynamically manipulate the top-level environment: `undefine`, `global-defined-value`, `invoke-open-unit` and `invoke-open-unit/sig`.
- MzScheme's interfaces, exception system and exception hierarchy.
- Primitives that access structures without the appropriate selectors: `struct-ref` and `struct->vector`.
- MrSpidey doesn't know about DrScheme language levels.
- Primitive names should not be assigned or defined.

Certain potential errors are not detected:

- Returning an inappropriate number of multiple values to a single value context.
- Index out of bounds on vector, string and list operations.
- `compound-unit` errors. Since `unit`'s are normally used in a first-order manner, these errors are typically easy to detect using the evaluator.
- The arguments to `primitive-name` and `primitive-result-arity` are only checked to be procedures, not primitive procedures.
- Errors in `read` due to ill-formed s-expression on the input port.

The analysis of certain kinds of code is not completely sound:

- Values passed to exception handlers.
- Tracking values through parameters.
- Tracking values through will executors.
- MrSpidey doesn't translate `unit/sig` into `unit`, as MzScheme does.

# Appendix E

# Implementation Details

## E.1 Zodiac

A useful interface for MrSpidey must present the results of the program analysis in terms of the original source program. Hence, the environment requires a front-end for processing source text that can correlate the internal representation of programs with their source location. For Scheme, this correlation task is complicated by the powerful macro systems of typical implementations because macros permit arbitrary rearrangements of syntax.

MrSpidey exploits Zodiac [32] for its front-end. Zodiac is a tool-kit for generating language front-ends that are suitable for interactive environments. It includes a hygienic high-level macro system that relates each expression in the macro-expanded code to its source location. MrSpidey exploits this information to associate value set invariants with expressions in the source program and to present portions of the value flow-graph as arrows relating terms in the program text.

## E.2 MrEd

MrSpidey's graphical component is implemented using MrEd [18], a Scheme-based engine for constructing graphical user interfaces. The core of the engine is a C++-like object system and a portable graphics library. This library defines high-level GUI elements, such as windows, buttons, and menus, which are embedded within Scheme as special primitive classes.

MrEd's graphical class library includes a powerful, extensible text editor class. This editor class is used in MrSpidey to display analyzed programs, including the boxes containing type invariants and the arrows describing the value flow graph. Type invariant boxes are easily embedded in the program text because an editor buffer can contain other buffers as part of its text. The arrows used to present flow information are not part of the editor's built-in functionality, but it was straightforward to extend the editor class with arrow drawing capabilities using other components of the graphical library.

MrEd's object system provides a robust integration between the Scheme implementation and the underlying graphical class library. The integration of the library through the object system is easily understood by GUI programmers. The object system also provides an important tool for designing and managing the components of a graphical interface. Because the implementation of MrSpidey exploits this object system, it can absorb future enhancements to the editor and it is easily integrated into the DrScheme environment.

Applications developed with MrEd—including MrSpidey and DrScheme—are fully portable across the major windowing systems (X–Windows, Microsoft Windows, and MacOS). MrEd's portability, its object system, and its rich class library enabled us to focus on the interesting and novel parts of MrSpidey's implementation.

## E.3 Multiple-Arity Functions

In contrast to the idealized languages of chapters 2 and 3, realistic languages such as Scheme [5] provide more flexible parameter passing mechanisms. In particular, Scheme allows multiple arguments to be passed to a function. It also allows a list of arguments to be passed, via the `apply` primitive, and it allows for the incoming list of arguments to be bound to a formal parameter, via the syntax

(**lambda** $x$ ...)

To cope with these multiple argument passing and binding possibilities, MrSpidey models each function as taking a single argument, which is a list of the function's *actual* arguments. Thus the function (**lambda** ($x$ $y$) ...) takes an argument list $l$, extracts the *car* of $l$ into $x$, and the *cadr* of $l$ into $y$. Conversely, at an application site, the arguments to the callee are wrapped up in a list, which is then passed as the single argument to the callee. Thus the application ($f$ $x$ $y$) actually applies $f$ to (*list* $x$ $y$). The various other parameter passing and binding modes of Scheme, including the `apply` primitive, can also be modeled within this framework.

## E.4 Multiple Values

Scheme [5] allows expressions to return a collection of multiple values, which can be converted into an argument list via the *call-with-values* primitive.

To cope with multiple values, MrSpidey models each expression as returning a list of return values, in a manner analogous to the way functions take a list of argument values. Thus, the expression:

$(\mathit{values}\ 1\ 2)$

is modeled in MrSpidey as returning the value:

(list 1 2)

Similarly, the expressions $(\mathit{values}\ 1)$ and 1 (which are equivalent) both return (list 1).

Thus each expression must return a list of multiple values, and each context needs to expect such a list. This convention substantially simplifies the analysis, but it does cause problems in explaining of the program's value flow. If we consider the binding expression:

(**let** $([x\ M])\ \ldots)$

then we would expect that there should be a value flow arrow from the expression $M$ to the variable $x$. In the absence of multiple values, this arrow corresponds to a subset constraint $[\alpha_M \leq \alpha_x]$ in the solved constraint system, where $\alpha_M$ and $\alpha_x$ are the set variables for $M$ and $x$, respectively.

Once we introduce multiple values, this direct correspondence between the intuitive value flow arrows and subset constraints no longer exists. That is, in the presence of multiple values, the constraint corresponding to the above **let** expression is:

$$\texttt{car}(\alpha_M) \leq \alpha_x$$

since $\alpha_M$ actually denotes a list of multiple values, which in this case should contain a single element, and that element should be extracted and bound to the variable $x$.

In order to produce intuitive value flow arrows that do not correspond to subset constraints, MrSpidey needs to preserve additional information for certain binding constructs such as the above **let** expression.

## E.5 Checking Scheme Primitives

Section 4.3 describes how to identify unsafe operations in the idealized language $\Lambda^p$. However, realistic languages such as MzScheme [17] contains a large number of primitive procedures in addition to those in $\Lambda^p$. The set of valid argument values for each of these primitives is often more complicated than for the simple primitives **car** and **cdr**. For example, the primitive procedure **member** requires two arguments, the second of which must be a list, and each element of the list must be a pair. MrSpidey uses *type schemas* to describe both the behavior of each primitive procedure and the appropriate argument set for that primitive.

### E.5.1 Type Schemas

Type schemas are used to describe the behavior and domain of primitive procedures, and have the form:

$$\mu \quad \in \quad Schema \quad = \quad \forall\overline{\alpha}.\omega$$

For example, the operation **car** has the associated type schema:

$$\mu_{\texttt{car}} = \forall\beta,\gamma.((\textbf{cons}\ \beta\ \gamma)\ \ \rightarrow^{\{\texttt{car}\}}\ \ \beta)$$

where `car` is the function tag associated with the primitive **car**.

We define the semantics of type schemas via the function $\mathcal{M}.[\cdot] : SetEnv \times Schema \longrightarrow \mathcal{P}(\mathcal{D})$:

$$\mathcal{M}_\rho[\forall\alpha_1,\ldots,\alpha_n.\omega] \quad = \quad \{\mathcal{M}_{\rho'}[\omega] \mid \rho' = \rho[\alpha_i \mapsto X_i], X_i \in \mathcal{D}\}$$

Thus the meaning of a type schema $\forall\alpha_1,\ldots,\alpha_n.\omega$ is the set of meanings for $\omega$ as the universally quantified variables $\alpha_1,\ldots,\alpha_n$ range over elements of $\mathcal{D}$. For a closed type schema $\mu$, we define:

$$\mathcal{M}[\mu] = \mathcal{M}_\rho[\mu]$$

where $\rho$ can be chosen arbitrarily, and does not affect the definition.

### E.5.2 New Constraint Classes

For each reference to a primitive operation, MrSpidey needs to translate the corresponding type schema into a constraint system. To assist in this process, we introduce two new classes of constraints: *checking constraints* and *constructor constraints*.

**Checking Constraints:** A checking constraint is of the form:

$$\alpha \leq C$$

for some $C \subseteq Const$, and the semantics of checking constraints is defined by:

$$\rho \models \alpha \leq C \quad \Longleftrightarrow \quad const(\rho(\alpha)) \subseteq C$$

Checking constraints are used to describe restrictions on the set of appropriate arguments to a primitive operation. For example, suppose the program refers to the primitive procedure **car**, and that $\alpha$ is the set variable corresponding to that reference. Then the data-flow behavior of **car** can be extracted from the corresponding type schema $\mu_{\texttt{car}}$ as the simple constraint system:

$$\{\texttt{car} \leq \alpha, \texttt{dom}(\alpha) \leq \alpha_d, \texttt{car}(\alpha_d) \leq \beta, \texttt{cdr}(\alpha_d) \leq \gamma, \beta \leq \texttt{rng}(\alpha)\}$$

and the restriction on appropriate arguments to **car** can also be extracted from $\mu_{\texttt{car}}$ as the checking constraint:

$$\alpha_d \leq \{\texttt{pair}\} \; .$$

**Constructor Constraints:** A constructor constraint is of the form:

$$\omega \leq \alpha \quad | \quad \alpha \leq \omega$$

and the semantics of constructor constraints is defined by:

$$\begin{array}{rcl} \rho \models \omega \leq \alpha & \iff & \mathcal{M}_\rho[\omega] \sqsubseteq \rho(\alpha) \\ \rho \models \alpha \leq \omega & \iff & \rho(\alpha) \sqsubseteq \mathcal{M}_\rho[\omega] \end{array}$$

A constructor constraint can be converted into an equivalent collection of simple and checking constraints via the function $CS$ described in figure E.1. That is:

- $\rho \models CS[\omega \leq \alpha]$ if and only if $\rho \models \omega \leq \alpha$, and
- $\rho \models CS[\alpha \leq \omega]$ if and only if $\rho \models \alpha \leq \omega$.

The cases for constructor constraints of the form $\omega \leq \alpha$ are straightforward. The cases for constructor constraints of the form $\alpha \leq \omega$ are more complicated. The function $CS$ first extracts all the top-level constants in $\omega$ via the function call $TLC[\omega]$, and creates a checking constraint that ensures that $\alpha$ only contains constants in $TLC[\omega]$. It then calls the function $CS'[\alpha \leq \omega]$ to handle containment within subcomponents of $\omega$.

$$
\begin{array}{rcl}
CS[c \le \alpha] & = & \{c \le \alpha\} \\
CS[\beta \le \alpha] & = & \{\beta \le \alpha\} \\
CS[\bot_s \le \alpha] & = & \emptyset \\
CS[(\omega_d \ \to^T \ \omega_r) \le \alpha] & = & \{T \le \alpha, \texttt{dom}(\alpha) \le \alpha_d,\ \alpha_r \le \texttt{rng}(\alpha)\} \\
 & & \cup CS[\omega_r \le \alpha_r] \cup CS[\alpha_d \le \omega_d] \\
 & & \text{where } \alpha_d, \alpha_r \text{ are fresh} \\
CS[(\texttt{cons}\ \omega_a\ \omega_d) \le \alpha] & = & \{\alpha_a \le \texttt{car}(\alpha),\ \alpha_r \le \texttt{rng}(\alpha)\} \\
 & & \cup CS[\omega_a \le \alpha_a] \cup CS[\omega_d \le \alpha_d] \\
 & & \text{where } \alpha_a, \alpha_d \text{ are fresh} \\
CS[\omega_1 \cup \omega_2 \le \alpha] & = & CS[\omega_1 \le \alpha] \cup CS[\omega_2 \le \alpha] \\
CS[(\texttt{rec}\ ([\alpha_1\ \omega_1]\ \ldots\ [\alpha_n\ \omega_n])\ \omega) \le \alpha] & = & CS[\alpha_i \le \omega_i] \cup CS[\omega_i \le \alpha_i] \cup CS[\omega \le \alpha] \\
 & & \\
CS[\alpha \le \omega] & = & \{\alpha \le TLC[\omega]\} \cup CS'[\alpha \le \omega]
\end{array}
$$

$$
\begin{array}{rcl}
CS'[\alpha \le C] & = & \emptyset \\
CS'[\alpha \le \bot_s] & = & \emptyset \\
CS'[\alpha \le \beta] & = & \{\alpha \le \beta\} \\
CS'[\alpha \le (\omega \ \to^T \ \omega)] & = & \{\ \alpha_d \le \texttt{dom}(\alpha),\ \texttt{rng}(\alpha) \le \alpha_r\} \\
 & & \cup CS[\omega \le \alpha_d] \cup CS[\alpha_r \le \omega] \\
CS'[\alpha \le (\texttt{cons}\ \omega_a\ \omega_d)] & = & \{\texttt{car}(\alpha) \le \alpha_a,\ \texttt{cdr}(\alpha) \le \alpha_d\} \\
 & & \cup CS[\alpha_a \le \omega_a] \cup CS[\alpha_d \le \omega_d] \\
CS'[\alpha \le \omega_1 \cup \omega_2] & = & CS'[\alpha \le \omega_1] \cup CS'[\alpha \le \omega_2]
\end{array}
$$

**Figure E.1** Converting constructor constraints to simple and checking constraints

The auxiliary function $TLC : Type \longrightarrow \mathcal{P}(Const)$ extracts the top level constants in a type, and is defined as follows:

$$
\begin{array}{rcl}
TLC[c] & = & \{c\} \\
TLC[\alpha] & = & \emptyset \\
TLC[\bot_s] & = & \emptyset \\
TLC[(\omega \ \to^T \ \omega)] & = & T \\
TLC[(\texttt{cons}\ \omega\ \omega)] & = & \{\texttt{pair}\} \\
TLC[\omega_1 \cup \omega_2] & = & TLC[\omega_1] \cup TLC[\omega_2] \\
TLC[(\texttt{rec}\ ([\alpha_1\ \omega_1]\ \ldots\ [\alpha_n\ \omega_n])\ \omega)] & = & \\
\end{array}
$$

$$
TLC[\omega] \cup \bigcup_{\alpha_j} TLC[(\texttt{rec}\ ([\alpha_1\ \omega_1]\ \ldots\ [\alpha_n\ \omega_n])\ \omega_j)]
$$

where $\alpha_j$ is mentioned at top level in $\omega$

### E.5.3 Converting Type Schemas to Constraints

For each reference to a primitive operation, MrSpidey converts the corresponding type schema into collection of simple and checking constraints. This conversion process involves two steps.

**Instantiating the Type Schema:** First, the type schema is instantiated. For $\mu = \forall \overline{\alpha}.\omega$, this instantiation involves replacing references in $\omega$ to the universally quantified variables $\alpha_1, \ldots, \alpha_n$ by fresh variables, producing an instantiated type $\omega'$.

**Converting the Instantiated Type into Constraints:** Second, the instantiated type $\omega'$ is converted into a collection of simple and checking constraints via the function $CS$. That is, if $\alpha$ is the set variable for the primitive procedure reference, then:

$$\mathcal{S} \cup \mathrm{S} = CS[\omega' \leq \alpha]$$

where $\mathcal{S}$ is a simple constraint system and $\mathrm{S}$ is a checking constraint system.

Thus MrSpidey converts each reference to a primitive operation into two constraint systems: a simple constraint system and a checking constraint system. The simple constraint system is passed on to the set-based analysis algorithm and the analysis proceeds as usual. The analysis later terminates, yielding a closed constraint system $\mathcal{S}$ for the analyzed program. Since this constraint system is closed under $\Theta$, it is straightforward to check if the least solution of this constraint system satisfies the checking constraints for the primitive operation. Specifically, $\mathit{LeastSoln}(\mathcal{S})$ satisfies the checking constraint $\alpha \leq C$ if and only if:

$$\{c \mid [c \leq \alpha] \in \mathcal{S}\} \subseteq C$$

If the least solution satisfies the checking constraints for a primitive operation, then that operation is only applied to valid arguments at run-time, and hence that operation is safe. Conversely, if the least solution does not satisfy the checking constraints, then the primitive operation may be applied to erroneous arguments at run-time, and the operation should be marked as unsafe.

# Appendix F

# Notations

| Symbol | Meaning | Section | Page |
|---|---|---|---|
| $M \in \Lambda$ | Terms | 2.1 | 11 |
| $V \in \mathit{Value}$ | Values | 2.1 | 11 |
| $x \in \mathit{Var}$ | Variables | 2.1 | 11 |
| $b \in \mathit{BasicConst}$ | Basic constants | 2.1 | 11 |
| $t \in \mathit{FnTag}$ | Function tags | 2.1 | 11 |
| $l \in \mathit{Label}$ | Labels | 2.1 | 11 |
| | | | |
| $\beta_v, \beta_{let}, \mathit{unlabel}$ | Reduction rules | 2.1.2 | 12 |
| $\longrightarrow$ | Reduction relation | 2.1.2 | 12 |
| $\mathcal{E}$ | Evaluation contexts | 2.1.2 | 12 |
| $\longmapsto, \longmapsto^*$ | Standard reduction relation | 2.1.2 | 12 |
| $\mathit{eval}$ | Evaluator | 2.1.2 | 12 |
| | | | |
| $\tau \in \mathit{SetExp}$ | Set expressions | 2.2 | 12 |
| $\alpha, \beta, \ldots \in \mathit{SetVar}$ | Set variables | 2.2 | 12 |
| $c \in \mathit{Const}$ | Constants | 2.2 | 12 |
| $\mathtt{dom}, \mathtt{rng}$ | Type expression constructors | 2.2 | 12 |
| $\mathsf{C} \in \mathit{Constraint}$ | Constraints | 2.2 | 12 |
| $\mathsf{S} \in \mathit{ConstraintSystem}$ | Constraint systems | 2.2 | 12 |
| $\mathsf{S}\mid_E$ | Restriction of a constraint system | 2.2 | 13 |
| | | | |
| $\mathcal{P}$ | Power set constructor | 2.2 | 13 |
| $\mathcal{D}$ | Domain for constraints | 2.3.1 | 14 |
| $\mathit{const}, \mathit{dom}, \mathit{rng}$ | Extract components of element of $\mathcal{D}$ | 2.3.1 | 14 |
| $\mathit{in}$ | Values described by constants | 2.3.1 | 14 |
| $\sqsubseteq, \top, \bot, \sqcup, \sqcap$ | Ordering, elements and operations on $\mathcal{D}$ | 2.3.1 | 14 |

| | | | |
|---|---|---|---|
| $\Lambda^u$ | Language $\Lambda^!$ plus units | 3.6 | 42 |
| $\mathtt{ui}, \mathtt{ue}$ | Selectors for units | 3.6.2 | 44 |
| $\Lambda^c$ | Language $\Lambda^!$ plus objects | 3.7 | 45 |
| $\mathtt{cl\text{-}obj}, \mathtt{ivar}_z^+, \mathtt{ivar}_z^-$ | Selectors for classes | 3.7.2 | 46 |
| | | | |
| $E$ | External variables | 6.1 | 63 |
| $\Delta$ | Inference rules on constraint systems | 6.2 | 64 |
| $\kappa$ | Non-constant set expression | 6.2 | 64 |
| $\mathbf{C} \in \mathit{CmpdConstraint}$ | Compound constraints | 6.2 | 65 |
| $\mathbf{S} \in \mathit{CmpdConSystem}$ | Compound constraint systems | 6.2 | 65 |
| $\Psi$ | Inference rules on constraint systems | 6.2 | 66 |
| $\Pi$ | Inference rules on constraint systems | 6.2 | 67 |
| $\vdash^E_{\Psi\Theta}, =^E_{\Psi\Theta}$ | Relations on constraint systems | 6.2 | 67 |
| | | | |
| $G$ | Grammar | 6.3.1 | 70 |
| $G_r$ | Function producing regular grammar | 6.3.1 | 70 |
| $\alpha_L, \alpha_U$ | Grammar non-terminals | 6.3.1 | 70 |
| $\mathcal{L}_G(X)$ | Language for $X$ in $G$ | 6.3.1 | 70 |
| $G_t$ | Function producing RTG | 6.3.3 | 71 |
| $R$ | Root non-terminal | 6.3.3 | 71 |
| $\mathcal{R}$ | Relation for computing entailment | 6.3.4 | 72 |
| | | | |
| $p, q \in \mathit{Path}$ | Paths | C.2 | 114 |
| $\mathit{arity}, \pi$ | Arity function | C.2 | 114 |
| $\leq_i$ | Either $\leq$ or $\geq$ | C.2 | 114 |
| $\sqsubseteq_i, \sqsupseteq_i$ | Either $\sqsubseteq$ or $\sqsupseteq$ | C.2 | 114 |
| $\subseteq_i$ | Either $\subseteq$ or $\supseteq$ | C.2 | 114 |
| $\bigsqcup_i$ | Either $\bigsqcup$ or $\sqcap$ | C.2 | 114 |
| $X@p$ | Injection function | C.2 | 114 |

# Bibliography


[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] Bourdoncle, F. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* (June 1993), pp. 46–55.

[4] Cartwright, R., and Felleisen, M. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software.*

[5] Clinger, W., and Rees, J. (Eds.). The revised$^4$ report on the algorithmic language scheme. *ACM Lisp Pointers 4*, 3 (July 1991).

[6] Cooper, K. D., Hall, M. W., Hood, R., Kennedy, K., McKinley, K., Mellor-Crummey, J., Torczon, L., and Warren, S. The Parascope parallel programming environment. *Proceedings of the IEEE* (February 1993), 244–263.

[7] 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.

[8] Detlefs, D. An overview of the extended static checking system. In *Proceedings of The First Workshop on Formal Methods in Software Practice* (January 1996), ACM (SIGSOFT), pp. 1–9.

[9] Deutsch, A., and Heintze, N. Partial solving of set constraints. Unpublished manuscript.

[10] 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).

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

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

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

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

[15] Flanagan, C., and Felleisen, M. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (June 1997), pp. 235–248.

[16] 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.

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

[18] Flatt, M. MrEd: An engine for portable graphical user interfaces. Technical Report TR-96-258, Rice University, 1996.

[19] Flatt, M., and Felleisen, M. First-class compilation units. Unpublished manuscript.

[20] Flatt, M., Krishnamurthi, S., and Felleisen, M. Mixins for java. POPL'97 submission.

[21] GÉCSEG, F., AND STEINBY, M. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[22] GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification.* Addison-Wesley, 1996.

[23] HEINTZE, N. *Set Based Program Analysis.* PhD thesis, Carnegie Mellon University, 1992.

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

[25] HINDLEY, R. J., AND SELDIN, J. P. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[26] HIRANANDANI, S., KENNEDY, K., TSENG, C.-W., AND WARREN, S. The D editor: A new interactive parallel programming tool. In *Proceedings of Supercomputing* (1994).

[27] 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.

[28] 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.

[29] 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.

[30] KENNEDY, K., MCKINLEY, K., AND TSENG, C.-W. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July 1991).

[31] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice-Hall, 1988.

[32] KRISHNAMURTHI, S. Zodiac: A programming environment builder. Technical Report TR-96-259, Rice University, 1996.

[33] Miller, B., Koski, D., Lee, C. P., Maganty, V., Murthy, P., Natarajan, A., and Steidl, J. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Computer Science Department, University of Wisconsin, 1995.

[34] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

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

[36] 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.

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

[38] Pugh, W. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing* (1991).

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

[40] Reynolds, J. C. The essence of ALGOL. *Algorithmic Languages* (1981), 345–372.

[41] Shei, B., and Gannon, D. Sigmacs: A programmable programming environment. In *Advances in Languages and Compilers for Parallel Computing.* The MIT Press, August 1990.

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

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

[44] Wolfe, M. J. The Tiny loop restructuring research tool. In *Proceedins of the 1991 International Conference on Parallel Processing* (August 1991).

[45] Wright, A. *Practical Soft Typing for Scheme.* PhD thesis, Rice University, 1994.

[46] Wright, A., and Cartwright, R. A practical soft type system for scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994), pp. 250–262.

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

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