Implementing a Static Debugger for a First-Order Functional Programming Language

Matthias Felleisen and Paul A. Steckler

Rice COMP TR00-359

June 2000
Revised: March 2001

Department of Computer Science
Rice University
P.O. Box 1892
Houston, TX 77005-1892

Abstract

A static debugger assists a programmer in finding potential errors in programs. The key to a static debugger is set-based analysis (SBA). Many authors have described *formulations* of SBA, but leave open gaps among that theory, its implementation, and its use for a particular purpose. An implementation needs to confront these practical issues. While some of the implementation proceeds directly from the formal description of the analysis, there is much fine detail in the code. With a series of reports, we intend to bridge the gap between theory and implementation. In this first report, we implement an analyzer for a simple, first-order functional language and show how to use the analysis in a static debugger.

1 Introduction

A static debugger assists a programmer in finding potential errors in programs. Flanagan's MrSpidey [3] is a static debugger for the MzScheme programming language, supplied as an extension of the DrScheme programming environment. MrSpidey works in three stages. First, it analyzes the program using an adaptation of set-based analysis (SBA) [5, 6]. Second, it uses the results of the analysis to inspect all program operations for potential violations of MzScheme's safety invariants. Finally, it translates the result of the inspection into a program annotation. All provably safe program operations are colored green; all others are colored red.

The key to a static debugger is set-based analysis. Many authors have described *formulations* of SBA [1, 3, 6, 7]. Such formulations are effective for understanding the theory behind SBA, but leave open gaps among that theory, its implementation, and its use for a particular purpose. An implementation needs to confront practical issues not directly addressed by formal descriptions of SBA. While some of the implementation proceeds directly from the formal description of the analysis, there is much fine detail in the code.

With a series of reports, beginning with this one, we intend to bridge the gap between theory and implementation. In this report, we implement an analyzer for a simple, first-order functional language and show how to use the analysis in a static debugger. In subsequent reports, we will extend the language to be analyzed and refine the analysis.

In Section 2, we describe the source language to be analyzed. In Section 3, we present the language of set constraints used in our SBA. Section 4 describes how constraints are generated, how they are closed under propagation rules, and how to build a global flow graph. In Section 5, we give an upper bound for the time needed to compute our SBA. Section 6 shows how to translate constraint solutions to printable types for source terms. In Section 7, we explain how constraint information is used to mark program points that may result in possible safety violations. Section 8 contains the complete Scheme source code.

Many of the ideas in this series of reports are presented in similar form in the work of Cormac Flanagan [3, 5]. Flanagan applied these ideas in the implementation of MrSpidey, a set-based analyzer for Scheme [4]. The code in this report is new, and not derived from the MrSpidey codebase.

2 Terms

The set of source terms is given by the following grammar: [1]

$$
\begin{aligned}
D &::= (\textbf{define}\ x\ V) \mid (\textbf{define}\ x\ (\textbf{lambda}\ (y^{+})\ M)^{\ell}) \\
M &::= V \mid x \mid (x\ M^{+}) \mid \\
&\quad\ (\textbf{cond}\ (M\ M)^{+}) \mid (\textbf{cond}\ (M\ M)^{*}\ (\textbf{else}\ M)) \mid \\
&\quad\ (\textbf{cons}\ M\ M) \mid (\textbf{car}\ M) \mid (\textbf{cdr}\ M) \mid (O\ M\ M) \\
O &::= + \mid - \mid * \mid \backslash \\
V &::= c
\end{aligned}
$$

As usual, we impose the restriction that variables in the argument list to a **lambda** must be distinct. Variables in the function position of an application must be globally-defined. The analysis does not rely on this latter restriction. Rather, the restriction is pedagogic, as in the DrScheme Beginning Student language level [8].

The metavariable c ranges over constants, including numbers, characters, strings, symbols, the empty list, and the special constant `void`. The metavariables x and y range over an infinite set of variables; ℓ is a *label* metavariable. A label indicates a particular occurrence of a **lambda** subterm. We write $\texttt{ar}(\ell)$ for the arity of the **lambda** subterm labelled by ℓ.

A *program* is a sequence of definitions followed by a sequence of expressions:

$$Program ::= D^{*}\ M^{*}$$

For programs, we impose the restriction that a variable may be **define**'d only once.

We have deliberately restricted the source language to make the description of its set-based analysis simple. Definitions have either constants or procedures as their second argument. With that restriction, the analysis need not be concerned about the order of definitions. For example, the order of the definitions is not significant in the program:

$$
\begin{aligned}
&(\textbf{define}\ f\ (\textbf{lambda}\ (z)\ (+\ x\ 1))) \\
&(\textbf{define}\ x\ 42) \\
&(f\ 39)
\end{aligned}
$$

because `x` is bound when `f` is applied.

The list constructor **cons**, the list destructors **car** and **cdr**, and arithmetic operators are supported as special forms. As a consequence, syntactic criteria can be used to determine where to generate the constraints associated with those primitives.

The dynamic semantics of this language is that of Beginning Student in the DrScheme programming environment [8]. The possible are run-time errors are

[1] A $*$ superscript indicates zero or more repetitions of a subterm. A $+$ superscript indicates one or more repetitions of a subterm.

1. **apply-error**: a variable in function position evaluates to a non-procedural value

2. **arity-error**: a procedure is applied to an incorrect number of arguments

3. **cons-error**: the second operand of a **cons** evaluates to a non-list

4. **car-error**, **cdr-error**: the operand of a **car** or **cdr** evaluates to a non-list

5. **mathop-error**: an operand of an arithmetic form evaluates to a non-number

A static debugger should flag those expressions where these errors may occur.

3 The language of constraints

Our SBA formulation is modified from Flanagan's disseratation [3, pp. 12, 31–32]. A *set expression* is of the form

$$\begin{aligned}\tau ::= {} & \alpha \mid \\ & \texttt{num} \mid c \mid \ell \mid \\ & \texttt{var}(\ell) \mid \texttt{dom}_n^m(\tau) \mid \texttt{rng}(\tau) \mid \\ & \texttt{pair} \mid \texttt{car}(\tau) \mid \texttt{cdr}(\tau)\end{aligned}$$

where

- α is a *set variable*,
- c ranges over constants, and
- ℓ ranges over term labels.

The symbols $\texttt{dom}_n^m$, $\texttt{rng}$, $\texttt{car}$, and $\texttt{cdr}$ are *selectors* that describe components of a set variable's denotation.[2] $\texttt{dom}_n^m$ has *arity index* m and *argument index* n, and so denotes the set of values in the domain associated with the nth argument of a procedure with arity m.[3] The arity index prevents the propagation of data to the actual arguments of a procedure in case of an arity mismatch. `pair` is a token that denotes the infinite set of all possible pairs of values. Similarly, `num` is a token denoting all numbers. $\texttt{var}(\ell)$ is the set variable associated with the term labelled by ℓ.

A *constraint* is an inequality relating two set expressions:

$$\tau \leq \tau'$$

A *constraint system* is a finite set of constraints. We write C for a constraint system.

[2]Flanagan describes a semantics of set expressions [3, Section 2.3.2].
[3]The idea of adding an arity index to the `dom` selector is due to Kevin Charter.

4 Generating and propagating constraints

There are two distinct phases in the analysis: constraint derivation and constraint propagation. In constraint derivation, the analyzer derives constraints for each syntax node in a program. During constraint propagation, the analyzer uses inference rules to generate new constraints. Additionally during the propagation phase, the analyzer builds a flow graph that describes the source of values.

4.1 Constraint generation

We write judgements of the form $S, \Gamma \vdash M : \alpha, C$, where

- S is a *top-level context* that maps term variables to set variables.
- Γ is a *lexical context* that maps term variables to set variables.

We distinguish these contexts because top-level variables — those not lexically bound — need to be handled differently than lexical variables. We could have merged top-level and lexical contexts for the language under study, but we believe that keeping them separate will be useful for future language extensions.

The constraint generation rules for definitions appear in Figure 1. The rules in Figure 2 derive constraints for the λ-calculus fragment of the source language. Figure 3 shows the generation rules for operations, and Figure 4 gives the rules for conditionals.

When applying these rules, we use a top-level context S_0 that maps each distinct top-level variable in the program to a fresh set variable. This strategy allows the analysis to track dataflow from definitions to uses of top-level variables.

4.2 Constraint propagation

The rules for generating constraints capture *local* flow conditions associated with program syntax. By combining such local constraints, we obtain *global* data flow conditions. The *propagation rules* in Figure 5 simulate data flow throughout the program.

The `trans-const` rule uses three predicates on set expressions. The predicate $\mathbf{const}?(\tau)$ holds iff τ is a constant; $\mathbf{label}?(\tau)$ holds iff τ is a label; $\mathbf{token}?(\tau)$ holds if τ is a token, that is, either `pair` or `num`. Therefore, this rule propagates only constant and constant-like data, such as labels.

The `trans-sel` rule propagates data flowing through a selector. The predicate $\mathbf{selector}?(\tau)$ holds iff τ is of the form $\mathtt{dom}_n^m(\alpha)$, $\mathtt{rng}(\alpha)$, $\mathtt{car}(\alpha)$, or $\mathtt{cdr}(\alpha)$. If τ is a `dom` selector, the arity index and argument index of the two occurrences of the selector in the premise must be identical.

The rules `rng-prop` and `dom-prop` propagate information about functions. Consider the first of these. The set expression $\mathtt{rng}(\beta)$ represents the set of

$$\frac{S, \Gamma \vdash M : \beta, C}{S, \Gamma \vdash (\textbf{define}\ x\ M) : \alpha, C \cup \{\beta \leq S(x), \texttt{void} \leq \alpha\}} \quad \text{(define)}$$

$$\frac{S, \Gamma[x_1 \mapsto \alpha_1] \cdots [x_n \mapsto \alpha_n] \vdash M : \beta, C'}{S, \Gamma \vdash (\textbf{define}\ x\ (\textbf{lambda}\ (x_1\ \ldots\ x_n)\ M)^{\ell}) : \alpha, C \cup C'} \quad \text{(define-fun)}$$

$$\text{where } C = \left\{ \begin{array}{c} \texttt{var}(\ell) \leq S(x) \\ \texttt{void} \leq \alpha \\ \ell \leq \texttt{var}(\ell) \\ \texttt{dom}_1^n(\texttt{var}(\ell)) \leq \alpha_1 \\ \cdots \\ \texttt{dom}_n^n(\texttt{var}(\ell)) \leq \alpha_n \\ \beta \leq \texttt{rng}(\texttt{var}(\ell)) \end{array} \right\}$$

Figure 1: Constraint generation rules for definitions.

$$S, \Gamma \vdash c : \alpha, \{c \leq \alpha\} \quad \text{(const)}$$

$$S, \Gamma[x \mapsto \beta] \vdash x : \alpha, \{\beta \leq \alpha\} \quad \text{(var-lexical)}$$

$$\frac{x \notin \mathrm{Dom}(\Gamma)}{S, \Gamma \vdash x : \alpha, \{S(x) \leq \alpha\}} \quad \text{(var-top)}$$

$$\frac{S, \Gamma \vdash x : \beta_1, C_1 \quad S, \Gamma \vdash M_i : \beta_i, C_i \quad i \in [2..n]}{S, \Gamma \vdash (x\ M_2\ \ldots\ M_n) : \alpha, \displaystyle\bigcup_{i \in [1..n]} C_i \cup C} \quad \text{(app)}$$

$$\text{where } C = \left\{ \begin{array}{c} \beta_2 \leq \texttt{dom}_1^{n-1}(\beta_1) \\ \cdots \\ \beta_n \leq \texttt{dom}_{n-1}^{n-1}(\beta_1) \\ \texttt{rng}(\beta_1) \leq \alpha \end{array} \right\}$$

Figure 2: Constraint generation rules for core syntax.

$$\frac{S,\Gamma \vdash M_1 : \alpha_1, C_1 \quad S,\Gamma \vdash M_2 : \alpha_2, C_2}{S,\Gamma \vdash (\texttt{cons}\ M_1\ M_2) : \alpha, C_1 \cup C_2 \cup \left\{ \begin{array}{l} \texttt{pair} \leq \alpha \\ \alpha_1 \leq \texttt{car}(\alpha) \\ \alpha_2 \leq \texttt{cdr}(\alpha) \end{array} \right\}} \quad \text{(cons)}$$

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

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

$$\frac{S,\Gamma \vdash M_1 : \alpha_1, C_1 \quad S,\Gamma \vdash M_2 : \alpha_2, C_2}{S,\Gamma \vdash (O\ M_1\ M_2) : \alpha, C_1 \cup C_2 \cup \{\texttt{num} \leq \alpha\}} \quad \text{(math-op)}$$

Figure 3: Constraint generation rules for operations.

results returned by all functions associated with β. By covariance, we know that

$$\texttt{rng}(\beta) \leq \texttt{rng}(\gamma)\,.$$

By transitivity, we may conclude that

$$\alpha \leq \texttt{rng}(\gamma)\,.$$

Symmetrically, the domains of two abstract function values are contravariantly related. That is,

$$\texttt{dom}_n^m(\gamma) \leq \texttt{dom}_n^m(\beta)$$

if $\beta \leq \gamma$, given any choice of m and n. Hence, the propagation rule is justified for any arity and specific argument position.

The propagation of pair data is handled by the rules `car-prop` and `cdr-prop`. In the first of these, β must be associated with pair data, hence so is γ. Since $\beta \leq \gamma$, we have $\texttt{car}(\beta) \leq \texttt{car}(\gamma)$. By transitivity, we conclude $\alpha \leq \texttt{car}(\gamma)$. The rule `cdr-prop` is justified similarly.

4.3 Building a flow graph

For debugging purposes, we wish to know the source of values produced by a term. As we have mentioned, the propagation rules simulate dataflow within a program. In particular, the *trans-const* rule propagates constant and constant-like data within a program. Each set variable is associated with a particular term. Therefore, each firing of this rule simulates the flow of a constant or constant-like datum from one term to another. We can track the sources of data by building a directed graph whose nodes correspond to subterms, and whose edges are in the reverse direction to dataflow.

(cond)

$$\frac{S, \Gamma \vdash M_{i,j} : \alpha_{i,j}, C_{i,j} \quad i \in [1..n], j \in [1,2]}{S, \Gamma \vdash (\textbf{cond}\ (M_{1,1}\ M_{1,2})\ \ldots\ (M_{n,1}\ M_{n,2})) : \alpha, \bigcup_{\substack{i \in [1..n] \\ j \in [1..2]}} C_{i,j} \cup C}$$

$$\text{where } C = \left\{ \begin{array}{c} \alpha_{1,2} \leq \alpha \\ \cdots \\ \alpha_{n,2} \leq \alpha \end{array} \right\}$$

(cond-else)

$$\frac{S, \Gamma \vdash M_{i,j} : \alpha_{i,j}, C_{i,j} \quad S, \Gamma \vdash M_{n+1,2} : \alpha_{n+1,2}, C_{n+1,2} \quad i \in [1..n], j \in [1,2]}{S, \Gamma \vdash (\textbf{cond}\ (M_{1,1}\ M_{1,2})\ \ldots\ (M_{n,1}\ M_{n,2})\ (\textbf{else}\ M_{n+1,2})) : \alpha, \bigcup_{\substack{i \in [1..n] \\ j \in [1..2]}} C_{i,j} \cup C}$$

$$\text{where } C = \left\{ \begin{array}{c} \alpha_{1,2} \leq \alpha \\ \cdots \\ \alpha_{n+1,2} \leq \alpha \end{array} \right\}$$

Figure 4: Constraint generation rules for conditionals.

We construct the graph as follows. Each subterm of the program generates a graph node. The propagation rule *trans-const* mentions two set variables, α and β, which have corresponding source terms. Each time we fire this propagation rule, we add an edge from node for the term for β to the node for the term for α.

Why are the edges reversed? Suppose we want to know the sources of values for some term M. To do so, we simply trace along paths leading from M. Were the edges in the same direction as dataflow, we would have to scan the parse tree for edges leading to M. If we wish to draw dataflow paths for a user, as in MrSpidey, we can always draw edges pointing from sources to sinks, though the data structure for edges represents them in the opposite sense.

For example, consider the program

$$\begin{array}{l} (\textbf{define}\ f\ (\textbf{lambda}\ (x)\ x)) \\ (\textbf{define}\ g\ (\textbf{lambda}\ (y)\ y)) \\ (f\ (g\ 42)) \end{array}$$

The flow graph created for this program contains the directed subgraph

$$(f\ (g\ 42)) \rightarrow x \rightarrow x_{bv} \rightarrow (g\ 42) \rightarrow y \rightarrow y_{bv} \rightarrow 42$$

where the *bv* subscript on a variable indicates the binding occurrence of that variable. This subgraph shows that the edges are opposite in sense to the actual

$$\frac{\tau \leq \alpha \quad \alpha \leq \beta \quad \mathbf{const}?(\tau) \ \vee \ \mathbf{label}?(\tau) \ \vee \ \mathbf{token}?(\tau)}{\tau \leq \beta} \quad \text{(trans-const)}$$

$$\frac{\alpha \leq \tau \quad \tau \leq \beta \quad \mathbf{selector}?(\tau)}{\alpha \leq \beta} \quad \text{(trans-sel)}$$

$$\frac{\alpha \leq \mathtt{rng}(\beta) \quad \beta \leq \gamma}{\alpha \leq \mathtt{rng}(\gamma)} \quad \text{(rng-prop)}$$

$$\frac{\mathtt{dom}_n^m(\alpha) \leq \beta \quad \alpha \leq \gamma}{\mathtt{dom}_n^m(\gamma) \leq \beta} \quad \text{(dom-prop)}$$

$$\frac{\alpha \leq \mathtt{car}(\beta) \quad \beta \leq \gamma}{\alpha \leq \mathtt{car}(\gamma)} \quad \text{(car-prop)}$$

$$\frac{\alpha \leq \mathtt{cdr}(\beta) \quad \beta \leq \gamma}{\alpha \leq \mathtt{cdr}(\gamma)} \quad \text{(cdr-prop)}$$

Figure 5: Constraint propagation rules.

data flow: the edges all point to the right, yet the constant 42, all the way on the right, is the result of $(f\ (g\ 42))$, all the way on the left.

5 Complexity

While most accounts of set-based analysis claim that it may be performed in time $O(n^3)$, the analysis here is different enough to give a higher time complexity. By reduction to a variant of transitive closure, we can give a bound of $O(n^6)$. With a simple restriction on programs, we can reduce the bound to $O(n^3)$. We provide a proof sketch.

Constraint derivation takes time $O(n^2)$, because $O(n)$ constraints are generated for each syntax tree node.

It is propagation time that dominates the computation. We reduce the propagation problem to a modified graph transitive closure problem, as follows. For each possible set expression generated by a program, add a node in the graph. Whenever there is a derived constraint between set expressions, add an edge between corresponding nodes. As for graph transitive closure, create a matrix A such that $A_{i,j}$ contains 1 if $i = j$ or there is an edge between node i and node j [2]. Next, run a modified graph transitive closure algorithm, using the propagation rules to decide when a 1 should be set in the matrix. Assume that given the index for α, the index for $\mathtt{rng}(\alpha)$ is computable in constant time, and similarly for other selectors. Assume the reverse lookup is also constant-time. When the algorithm completes, throw out the reflexive edges, since they are not used.

The time to compute the closure algorithm depends on the number of nodes in the graph. There are $O(n)$ many set variables. For each set variable, there are $O(n)$ many `dom` selectors, because there are $O(n)$ formal parameters and $O(n)$ actual parameters, each of which generates a `dom` selector. Hence, there are $O(n^2)$ `dom` selectors applied to set variables. That means there must be $O(n^2)$ nodes in the graph. Graph transitive closure takes time $\Theta(N^3)$, where N is the number of graph nodes [2]. So the time for the algorithm is $\Theta(O(n^2))^3 = O(n^6)$.

If all procedures have one argument and all application sites have one operand, there is only one possible `dom` selector, $\mathtt{dom}_1^1$. In that case, there are only $O(n)$ many set expressions of the form $\mathtt{dom}_1^1(\alpha)$. With that restriction, the closure algorithm takes time $O(n^3)$.

There may well be a better available bound than the one just given. Our implementation does not use the transitive closure-like algorithm presented, since it is rather indirect, as well as expensive. However, we have been unable so far to give a definite bound for our implemented algorithm.

6 Types

After closing a constraint system under the propagation rules, we can extract a printable *type* from a set variable that concisely describes the data flowing to the term associated with the set variable.[4]

Types have the syntax

$$\begin{aligned}\tau ::= {} & \alpha \mid \mathtt{num} \mid c \mid (\mathtt{union}\ \tau\ \tau^+) \mid (\mathtt{cons}\ \tau\ \tau) \mid \\ & (\mathtt{rec}\ ((\alpha\ \tau)^+)\ \tau) \mid (\tau^+\ \rightarrow\ \tau) \mid \\ & \mathtt{empty}\end{aligned}$$

where `empty` denotes no data.

6.1 Building types

Given a set variable α, we construct a corresponding type in two stages [3, Chap. 4].

In the first stage, we build the recursive type

$$\mathit{MkType}(\alpha) = (\mathtt{rec}\ ((\alpha_1\ \omega_1)\dots(\alpha_n\ \omega_n))\ \alpha)$$

The α_i's are all the set variables that have been derived during the construction of the initial top-level context and the constraint derivation phases. Each ω_i is produced by an auxiliary procedure, $\mathit{MkTypeAux}(\alpha_i)$.

[4]The word *type* here refers to a description of a set expression, rather than as a constraint on valid programs.

In order to describe *MkTypeAux*, we need *Set2Type*, a function that takes sets of types to types:

$$
\mathit{Set2Type}(T) = \begin{cases} (\texttt{union}\ \tau_1 \ldots \tau_n) & \text{where } \tau_i \in T, \text{ if } |T| > 1 \\ \tau & \text{if } T = \{\tau\} \\ \texttt{empty} & \text{if } T = \emptyset \end{cases}
$$

Then

$$
\mathit{MkTypeAux}(\alpha) = (\texttt{union}\ \omega_c\ \omega_{\texttt{cons}}\ \omega_{\rightarrow})
$$

where

$$
\begin{array}{rcl}
\omega_c & = & \mathit{Set2Type}(\{c \mid c \leq \alpha\}) \\[1em]
\omega_{\texttt{cons}} & = & \begin{cases} (\texttt{cons}\ \omega_{\texttt{car}}\ \omega_{\texttt{cdr}}) & \text{if } \texttt{pair} \leq \alpha \\ \texttt{empty} & \text{otherwise} \end{cases} \\
& & \text{where} \\
& & \begin{array}{rcl} \omega_{\texttt{car}} & = & \mathit{Set2Type}(\{\beta \mid \beta \leq \texttt{car}(\alpha)\}) \\ \omega_{\texttt{cdr}} & = & \mathit{Set2Type}(\{\beta \mid \beta \leq \texttt{cdr}(\alpha)\}) \end{array} \\[1em]
\omega_{\rightarrow} & = & \begin{cases} (\texttt{union}\ (\omega_{\texttt{dom}_1^{\texttt{ar}(\ell_1)}} \ldots \omega_{\texttt{dom}_{\texttt{ar}(\ell_1)}^{\texttt{ar}(\ell_1)}} \rightarrow \omega_{\texttt{rng}}) & \text{if } \exists \ell . \ell \leq \alpha \\ \qquad \ldots & \\ \qquad (\omega_{\texttt{dom}_n^{\texttt{ar}(\ell_n)}} \ldots \omega_{\texttt{dom}_{\texttt{ar}(\ell_n)}^{\texttt{ar}(\ell_n)}} \rightarrow \omega_{\texttt{rng}})) & \\ \qquad \text{for each } \ell_i \text{ such that } \ell_i \leq \alpha & \\ \texttt{empty} & \text{otherwise} \end{cases} \\
& & \text{where} \\
& & \begin{array}{rcl} \omega_{\texttt{dom}_j^m} & = & \mathit{Set2Type}(\{\beta \mid \alpha \leq^* \delta, \beta \leq \texttt{dom}_j^m(\delta)\}) \\ \omega_{\texttt{rng}} & = & \mathit{Set2Type}(\{\beta \mid \beta \leq \texttt{rng}(\alpha)\}) \end{array}
\end{array}
$$

Even in our restricted language, a term may be associated with more than one procedure, even though at call sites, the operator evaluates to at most one procedure. Consider the program

(**define** f (**lambda** (x) ($+$ x 17))
(**define** g (**lambda** (y) ($-$ y 5)))
(**cond** (`#t` f) (**else** g))

Although only one branch of the condition is taken, the analysis cannot detect that fact. Hence the type of the conditional is a union of arrow types.

6.2 Simplifying types

The second stage of the type construction process simplifies the result of *MkType*. Usually, the recursive types built by *MkType* contains more information than needed. Each recursive type contains a binder for *every* set variable created. Most of these binders are semantically irrelevant for a given recursive type. We can simplify types according to the reduction rules in Figure 6. These rules simplify the presentation of types, without altering their semantics [3, p. 53].

$$\frac{\omega_i \longrightarrow \omega_i'}{(\texttt{rec}\ (\ldots(\alpha_i\ \omega_i)\ldots)\ \tau) \longrightarrow (\texttt{rec}\ (\ldots(\alpha_i\ \omega_i')\ldots)\ \tau)} \quad \text{(rec)}$$

$$\frac{\tau_i \longrightarrow \tau_i' \quad i \in [1..n]}{(\texttt{union}\ \tau_1 \ldots \tau_n) \longrightarrow (\texttt{union}\ \tau_1' \ldots \tau_n')} \quad \text{(union)}$$

$$\frac{\tau_i \longrightarrow \tau_i' \quad i \in [1,2]}{(\texttt{cons}\ \tau_1\ \tau_2) \longrightarrow (\texttt{cons}\ \tau_1'\ \tau_2')} \quad \text{(cons)}$$

$$\frac{\tau_i \longrightarrow \tau_i' \quad i \in [1..n]}{(\tau_1 \ldots \tau_{n-1} \longrightarrow \tau_n) \longrightarrow (\tau_1' \ldots \tau_{n-1}' \longrightarrow \tau_n')} \quad \text{(fun)}$$

$$(\texttt{rec}\ ()\ \tau) \quad \longrightarrow \quad \tau \qquad \text{(rec-elim)}$$

$$\begin{array}{l} (\texttt{rec}\ (\ldots_1\ (\alpha_i\ \omega_i)\ \ldots_2)\ \tau) \quad \longrightarrow \\ \qquad\qquad (\texttt{rec}\ (\ldots_1\ \ldots_2)\ \tau)[\omega_i/\alpha_i] \\ \qquad \text{provided } \alpha_i \text{ does not occur in } \omega_i \end{array} \qquad \text{(unfold)}$$

$$(\texttt{union}\ \ldots_1\ \texttt{empty}\ \ldots_2) \quad \longrightarrow \quad (\texttt{union}\ \ldots_1\ \ldots_2) \qquad \text{(empty-filter)}$$

$$\begin{array}{l} (\texttt{union}\ \ldots_1\ (\texttt{union}\ \ldots_2)\ \ldots_3) \quad \longrightarrow \\ \qquad\qquad (\texttt{union}\ \ldots_1\ \ldots_2\ \ldots_3) \end{array} \qquad \text{(union-merge)}$$

$$(\texttt{union}\ \tau) \quad \longrightarrow \quad \tau \qquad \text{(union-elim)}$$

Figure 6: Type reductions.

7 Static debugging

A set-based analysis yields a set of abstract values that each expression in a program may produce. Using these sets, a static debugger can determine whether a program operation may signal a violation of a safety constraint.

Recall that the semantics of our language yields these possible safety violations: **apply-error**, **arity-error**, **cons-error**, **car-error**, and **cdr-error**.[5]

Suppose α is the set variable associated with a term M. Then the solution set for α is defined as

$$sba_C(\alpha) = \{\ \tau \mid \textbf{const}?(\tau)\ \vee\ \textbf{label}?(\tau)\ \vee\ \textbf{token}?(\tau)\ \wedge\ \tau \leq \alpha \in C\ \}$$

where C is some constraint system.

[5] In addition to the list given here, a static debugger might flag division as unsafe for almost all programs because it is rarely possible to determine whether or not the divisor is zero.

Now consider this program containing an instance of **apply-error**:

(**define** x 3)
(x 17)

Suppose α is the set variable associated with the x in the application. If sba_C includes any elements other than function labels, we mark the operation as possibly causing an error.

An example of **arity-error** is:

(**define** f (**lambda** (x) (+ x 14)
(f 97 103)

Let α_1 be the set variable associated with the variable f in the application. The set $sba_C(\alpha_1)$ contains at most one label, which indicates the procedure that may be applied at the site. If the arity of that procedure differs from the number of operands at the site, the procedure and application site are marked as possibly causing an arity mismatch.

The expression

(**cons** 37 3)

is an example of **cons-error**. Consider a **cons** site:

(**cons** M_1 M_2)

Suppose α_3 is the set variable associated with M_2. If $sba_c(\alpha_3)$ contains elements other than the `pair` token and the empty-list constant, then M_2 is possibly not a list. In that case, we mark the `cons` site as possibly causing an error. The analyses for **car-error** and **cdr-error** is similar to that for **cons-error**. For those cases, we examine data flowing to the argument of the **car** or **cdr**. If the solution to the set variable for that argument contains any element other than the `pair` token, the site is marked as possibly causing an error.

The expression

(+ 4 `#f`)

is an example of **mathop-error**. To detect this error, we need to consider both operands of a mathematical operation. Let α_4 and α_5 be the set variables associated with the operands. If $sba_C(\alpha_4)$ or $sba_C(\alpha_5)$ contains elements other than numbers or the `num`token, then the mathematical operation is marked as the source of a potential safety violation.

The errors that are marked are only *potential* errors. An execution of a program may not even evaluate a marked term, so that a safety violation may not occur.

Programmers can use the flow graph to trace the source of errors. Consider the program

(**define** z `#f`)
(**define** f (**lambda** (x) (+ x 97)))
(f z)

The analyzer flags the addition as a possible error. Because the second operand is a constant, the only possible source of error is the first operand x. The flow graph contains the subgraph

$$x \to x_{bv} \to z \to z_{bv} \to \texttt{\#f}$$

where z_{bv} is the occurrence of c in the **define**. From the flow graph, the programmer can determine the source of the troublesome value for x.

8 Implementation

In this section, we present Scheme code that implements constraint generation, constraint propagation, and the marking of possible errors. The code is distributed over several files, which may be loaded as a DrScheme *project*. We elide some of the code not directly related to the static analysis, such as the code for pretty-printing of terms and constraints.

8.1 Overview of the code

Source terms, set expressions, and constraints are encoded as MzScheme structures. Each constraint is stored twice:

1. in a list of constraints
2. in a map from set expressions to a pair consisting of a list of lower bounds and a list of upper bounds

The latter map is implemented as a hash table. Because MzScheme hash tables use intensional equality, the keys in those maps are symbol representations of set expressions. Whenever a constraint is added in the derivation phases or during constraint propagation, both the constraint list and bounds hash table are updated.

Constraint derivation and the creation of the initial store context S_0 are done in a single pass over the program syntax tree. For each top-level variable, we create an entry in S_0 the first time an occurrence of that variable is traversed. During the traversal of the program, we may encounter a top-level variable reference before seeing its definition, as when the program contains mutually-recursive procedures:

$$(\textbf{define}\ f\ (\textbf{lambda}\ (x)\ \ldots\ g\ \ldots))$$
$$(\textbf{define}\ g\ (\textbf{lambda}\ (y)\ \ldots\ f\ \ldots))$$

Only one entry in S_0 is created for a given top-level variable, no matter how many occurrences of that variable exist.

The constraint list allows for easy iteration over all constraints. The list of lower bounds is used by the implementation of *MkType*, during constraint propagation, and during error checking. The list of upper bounds is primarily used during constraint propagation. Each propagation rule is of the form

$$\frac{\tau_1 \leq \tau_2 \quad \tau_3 \leq \tau_4}{\cdots}$$

where τ_3 is syntactically related to either τ_1 or τ_2. Thus if we have a constraint $\tau_1 \leq \tau_2$, the list of upper bound allows us to quickly check whether τ_4 is among the upper bounds for τ_3. Symmetrically, by using the list of lower bounds, if we find $\tau_3 \leq \tau_4$, we can check whether τ_1 is among the lower bounds for τ_2.

For constraint propagation, we iterate over the list of constraints generated in the derivation phase exactly once. For each constraint, for each propagation rule, we check whether the constraint matches the first premise in the rule. If so, we use the list of upper bounds to see if there is a constraint matching the second premise in the rule. In that case, a new constraint is added and checked to see if it triggers any propagation rule. Symmetrically, we check whether the constraint matches the second premise for each rule. If there is a match, we check for the existence of a suitable constraint matching the first premise. When that premise exists, a new constraint is checked against the propagation rules by a recursive call to the propagation procedure.

8.2 Representing terms

We create a datatype `T` to represent all possible terms. Note that a `Lambda` is not a `T`. The datatype `Term` corresponds to the nonterminal M in our grammar on page 2.

```
;;; terms.ss

;; datatypes for top-level terms

; T = Define | Term

; Lambda = (make-Lambda (listof sym) Term)

; Define = (make-Define sym Term)
;        | (make-Define sym Lambda)

; Term =   (make-Var sym)
;        | (make-Const (union char sym string bool number
;                       '() 'void 'empty))
;        | (make-App Var Term)
;        | (make-Cons Term Term)
;        | (make-Car-app Term)
;        | (make-Cdr-app Term)
;        | (Mathop (union '+ '- '/ '*) Term Term)
;        | (make-Cond (listof Term) (listof Term))
```

```
;        | (make-Cond-else (listof Term) (listof Term) Term)

(define-struct Lambda (xs body label))
(define-struct Define (name exp))
(define-struct Var (name))
(define-struct Const (val)) ; also considered a set expression
(define-struct App (rator rands))
(define-struct Cons (car cdr))
(define-struct Car-app (cons))
(define-struct Cdr-app (cons))
(define-struct Mathop (op rand1 rand2))
; preds, exps are equal-length lists
(define-struct Cond (preds exps))
; preds, exps are equal-length lists
(define-struct Cond-else (preds exps else-exp))

; label structure, used for Lambda's
; name is a symbol
(define-struct Label (name))

; -> Label
(define (gen-label) (make-Label (gensym)))
```

We define a couple of useful constants. Recall that constants are also types.

```
(define *empty-list* (make-Const '()))
(define *void-type* (make-Const 'void))
```

8.3 Representing set expressions

Next we define structures for the datatypes in Section 3.

```
;;; setexp.ss

; set expressions

; Set-exp = (make-Set-var sym)
;         | (make-Dom positive-exact-integer
;                     positive-exact-integer Set-var)
;         | (make-Rng Set-var)
;         | (make-Car Set-var)
;         | (make-Cdr Set-var)
;         | (make-Token (union 'pair 'num 'empty))

(define-struct Set-var (name))

; set expression constructors
```

```
(define-struct Rng (set-var))
(define-struct Dom (arity pos set-var))
(define-struct Car (set-var))
(define-struct Cdr (set-var))

; set expression tokens
(define-struct Token (name))

(define *pair-token* (make-Token 'pair))
(define *num-token* (make-Token 'num))
(define *empty-type* (make-Token 'empty))

; -> Set-var
(define (gen-set-var)
  (make-Set-var (gensym)))

; Type -> bool
(define (empty-type? set-exp)
  (eq? set-exp *empty-type*))
```

8.4 Global store

We use a hash table to represent the global store, S in the constraint derivation rules on page 4. Notice `reset-the-store`, which allows us to obtain an empty store. Later, we introduce reset procedures for various other global data structures.

```
;;; store.ss

;; global store

; (hash-table-of sym Set-var)
(define *the-store* (make-hash-table))

; -> void
(define (reset-the-store)
  (set! *the-store* (make-hash-table)))

; sym Set-var -> void
(define (extend-store name set-var)
  (hash-table-put! *the-store* name set-var))

; sym -> Set-var
(define (lookup-in-store-by-symbol sym)
  (hash-table-get *the-store* sym (lambda () #f)))
```

```
; Var -> Set-var
(define (lookup-in-store x)
  (lookup-in-store-by-symbol (Var-name x)))
```

8.5 Representing constraints

A constraint consists of lower and upper bounds. We maintain two data structures for constraints: a list of all constraints and a map from set expression to a pair consisting of a list of lower bounds and a list of upper bounds. The list allows iteration over all constraints. The other map, implemented as a hash table, allows quick searches of corresponding bounds, given a set expression.

MzScheme hash tables use `eq?` on hash keys. Because set expressions may be constructed more than once, we cannot use them as hash keys directly. Instead, we *encode* set expressions as symbols to be used as hash keys. We omit the code that performs the encoding.

```
;;; constraints.ss

; Constraint = (make-constraint Set-exp Set-exp)

(define-struct constraint (lo hi))

(define *empty-pairs-thunk* (lambda () '(() ())))

; (listof Constraint)
(define *the-constraints* '())

; hash table from set expression to lists of lower, upper bounds
; (hash-table-of Set-exp (list (listof Set-exp) (listof Set-exp))
(define *constraint-table* (make-hash-table))

; -> void
(define (reset-the-constraints)
  (set! *the-constraints* '())
  (set! *constraint-table* (make-hash-table)))

; sym -> (list (listof Set-exp) (listof Set-exp))
(define (lookup-coded-in-constraint-table coded-set-exp)
    (hash-table-get *constraint-table* coded-set-exp
                    *empty-pairs-thunk*))

; Set-exp -> (list (listof Set-exp) (listof Set-exp))
(define (lookup-in-constraint-table set-exp)
    (lookup-coded-in-constraint-table (set-exp->symbol set-exp)))

; ((list (listof Set-exp) (listof Set-exp)) -> (listof Set-exp)) ->
```

```
;   Set-exp -> (listof Set-exp)
(define (make-lookup-in-constraint-table accessor)
  (lambda (set-exp)
    (accessor (lookup-in-constraint-table set-exp))))

; Set-exp -> (listof Set-exp)
(define lookup-in-lo-constraint-table
  (make-lookup-in-constraint-table cadr))

; Set-exp -> (listof Set-exp)
(define lookup-in-hi-constraint-table
  (make-lookup-in-constraint-table car))

; (hash-table-of Set-exp
;   (list (listof Set-exp) (listof Set-exp)) ->
;     sym -> (listof Set-exp)
(define (make-lookup-coded-in-constraint-table tbl)
  (lambda (coded-set-exp accessor)
    (accessor (lookup-coded-in-constraint-table coded-set-exp))))

; sym -> (listof Set-exp)
(define lookup-coded-in-lo-constraint-table
  (make-lookup-coded-in-constraint-table cadr))

; sym -> (listof Set-exp)
(define lookup-coded-in-hi-constraint-table
  (make-lookup-coded-in-constraint-table car))

; sym (listof Set-exp) (listof Set-exp) -> void
(define (set-bounds! coded-set-exp los his)
  (hash-table-put! *constraint-table*
                   coded-set-exp
                   (list los his)))

; Set-exp
;   ((listof Set-exp) -> (listof Set-exp))
;   ((listof Set-exp) -> (listof Set-exp)) ->
;   void
(define (add-new-bound! set-exp lo-fun hi-fun)
  (let* ([coded-set-exp (set-exp->symbol set-exp)]
         [all-bounds
          (lookup-coded-in-constraint-table coded-set-exp)]
         [old-los (car all-bounds)]
         [old-his (cadr all-bounds)])
    (set-bounds! coded-set-exp
                 (lo-fun old-los) (hi-fun old-his))))
```

```
; Set-exp Set-exp -> void
(define (add-new-upper-bound! set-exp hi)
  (add-new-bound! set-exp
                  identity
                  (lambda (old-his)
                    (cons hi old-his))))

; Set-exp Set-exp -> void
(define (add-new-lower-bound! set-exp lo)
  (add-new-bound! set-exp
                  (lambda (old-los)
                    (cons lo old-los))
                  identity))

; Set-exp Set-exp -> (union #f (listof Set-exp))
; could just as well have used lo-constraint-table
(define (exists-constraint lo hi)
 (member lo (lookup-in-hi-constraint-table hi)))
```

When we add a new constraint, we update the lists of lower and upper bounds. In case the lower bound of the constraint contains a `dom` selector, we update a table of domain arities, as discussed in the next subsection.

```
; returns #t if constraint added, else #f
; Constraint -> bool
(define (add-constraint constraint)
  (let* ([lo (constraint-lo constraint)]
         [hi (constraint-hi constraint)])
    (if (exists-constraint lo hi)
        #f
        (begin
          (when (Dom? lo)
                (add-to-dom-table lo))
          (add-new-lower-bound! hi lo)
          (add-new-upper-bound! lo hi)
          (set! *the-constraints*
                (cons constraint *the-constraints*))
          #t))))

; Set-exp Set-exp -> bool
; returns #t if constraint added, else #f
(define (add-constraint-with-bounds lo hi)
  (add-constraint (make-constraint lo hi)))
```

8.6 Global tables

The code contains several global tables. For example, we use a hash table mapping set variables to a list of arities. This table is used when considering the *dom-prop* propagation rule. If we have a match for the second premise of that rule, we need to know the possible arities that might be seen in a constraint for the first premise.

```
;;; globmaps.ss

; (hash-table-of Set-var (listof positive-integer))
(define *dom-table* (make-hash-table))

; -> void
(define (reset-dom-table)
  (set! *dom-table* (make-hash-table)))

; Set-var -> (listof positive-integer)
(define (lookup-set-var-arities set-var)
  (hash-table-get *dom-table* set-var
                  (lambda () '())))

; (make-Dom arity pos Set-var) -> void
(define (add-to-dom-table set-exp)
  (let* ([arity (Dom-arity set-exp)]
         [set-var (Dom-set-var set-exp)]
         [curr-arities (lookup-set-var-arities set-var)])
    (unless (memq arity curr-arities)
            (hash-table-put! *dom-table* set-var
                             (cons arity curr-arities)))))
```

We wish to obtain a set variable given a term, and vice-versa, so we maintain appropriate hash tables:

```
; associate terms with set variables

; (hash-table-of T Set-var)
(define *term-map* (make-hash-table))

; -> void
(define (reset-term-map)
  (set! *term-map* (make-hash-table)))

; T -> Set-var
(define (lookup-in-term-map t)
  (hash-table-get *term-map* t
                  (lambda ()
```

```
                    (error
                     "Term not in term map: " t))))

; the inverse map: set variables to terms

; (hash-table-of Set-var T)
(define *set-var-map* (make-hash-table))

; -> void
(define (reset-set-var-map)
  (set! *set-var-map* (make-hash-table)))

; Set-var -> T
(define (lookup-in-set-var-map set-var)
  (hash-table-get
   *set-var-map* set-var))

; T Set-var -> void
(define (associate-term-with-set term sv)
  (hash-table-put! *term-map* term sv)
  (hash-table-put! *set-var-map* sv term))
```

For building types, we'll need a list of all set variables created during the analysis.

```
; -> (listof Set-var)
(define (get-all-set-vars)
  (hash-table-map *set-var-map* (lambda (sv t) sv)))
```

We may repeatedly need the arity of procedures. Rather than repeatedly taking the length of procedure argument lists, we maintain a map from labels to arities.

```
; hash table from Labels to arities
; (hash-table-of Label nonnegative-exact-integer)
(define *label-arity-map* (make-hash-table))

; -> void
(define (reset-label-arity-map)
  (set! *label-arity-map* (make-hash-table)))

; Label nonnegative-exact-integer -> void
(define (set-label-arity! ell n)
  (hash-table-put! *label-arity-map* ell n))

; Label -> (union nonnegative-exact-integer 'not-found)
(define (lookup-label-arity ell)
```

```
  (hash-table-get *label-arity-map* ell
                  (lambda () 'not-found)))
```

The analyzer solves for sets of labels. For the user, we want to print out the terms associated with labels. Accordingly, we maintain a map from labels to procedures.

```
; (hash-table Label Lambda)
(define *label-lambda-map* (make-hash-table))

; -> void
(define (reset-label-term-map)
  (set! *label-lambda-map* (make-hash-table)))

; Label Lambda -> void
(define (associate-label-with-lambda ell t)
  (hash-table-put! *label-lambda-map* ell t))

; Label -> (union Lambda 'not-found)
(define (lookup-label-lambda ell)
  (hash-table-get *label-lambda-map* ell
                  (lambda () 'not-found)))
```

Because the analyzer may be run more than once, we wish to clear the state of global maps. We have not yet seen all the global maps yet, but it is sensible to reset them all in one procedure.

```
; -> void
(define (reset-system)
  (reset-label-arity-map)
  (reset-label-term-map)
  (reset-the-store)
  (reset-the-constraints)
  (reset-dom-table)
  (reset-term-map)
  (reset-set-var-map)
  (reset-flow-graph)
  (reset-type-tables)
  (collect-garbage))
```

8.7 Constraint environments

Constraint environments, Γ's in the derivation rules, are implemented as association lists.

```
;;; env.ss

;; constraint environments (Gammas)
```

```
; let Env be the type of Gammas

; -> empty-list
(define (make-constraint-env) '())

; Var (listof (sym Set-var)) -> (union Set-var #f)
(define (lookup-in-env x Gamma)
  (let ([binding (assq (Var-name x) Gamma)])
    (if binding
        (cadr binding)
        #f)))

; Env (listof sym) (listof Setvar) -> Env
; the two input lists must be of equal length
(define (extend-environment env vars set-exps)
  (append (map list vars set-exps) env))
```

8.8 Constraint derivation

Here we implement the constraint derivation rules in Figures 1 through 4, simultaneously building the initial store context S_0:

```
; constraint derivation

; T -> Set-var
(define (derive-top-term-constraints Gamma M)
  (let ([alpha (gen-set-var)])
    (match M
           [($ Const _)
            (add-constraint-with-bounds M alpha)]
           [($ Var name)
            (cond
             [(or (lookup-in-env M Gamma)
                  (lookup-in-store M))
              =>
              (lambda (set-var)
                (add-constraint-with-bounds set-var alpha))]
             [else
              (let ([new-set-var (gen-set-var)])
                (extend-store name new-set-var)
                (add-constraint-with-bounds new-set-var alpha))])]
           [($ Define name exp)
            (derive-top-term-constraints Gamma name)
            (let ([name-var (lookup-in-store name)])
              (associate-term-with-set name name-var)
```

```
   (add-constraint-with-bounds *void-type* alpha)
   (if (Lambda? exp)
       (let* ([xs (Lambda-xs exp)]
              [body (Lambda-body exp)]
              [label (Lambda-label exp)]
              [xs-vars (map
                         (lambda (_) (gen-set-var)) xs)]
              [max-n (length xs)]
              [body-env (extend-environment Gamma
                                          (map Var-name xs)
                                          xs-vars)]
              [beta (derive-top-term-constraints
                     body-env body)]
              [ell-var (gen-set-var)])
         (for-each
          (lambda (x x-var)
            (associate-term-with-set x x-var))
          xs xs-vars)
         (associate-term-with-set exp ell-var)
         (associate-label-with-lambda label exp)
         (set-label-arity! label max-n)
         (add-constraint-with-bounds ell-var name-var)
         (add-constraint-with-bounds label ell-var)
         (let loop ([n 1]
                    [xs-vars xs-vars])
           (unless (null? xs-vars)
                   (add-constraint-with-bounds
                    (make-Dom max-n n ell-var)
                    (car xs-vars))
                   (loop (add1 n) (cdr xs-vars))))
         (add-constraint-with-bounds beta
                                  (make-Rng ell-var)))
       ; not a Lambda
       (let ([beta (derive-top-term-constraints Gamma exp)])
         (add-constraint-with-bounds beta name-var))))]
[($ App rator rands)
 (let ([rator-var
         (derive-top-term-constraints Gamma rator)]
       [rands-vars
        (map (lambda (rand)
               (derive-top-term-constraints Gamma rand))
             rands)]
       [max-n (length rands)]
       [n 0])
   (let loop ([n 1]
              [rands-vars rands-vars])
```

```
     (unless (null? rands-vars)
             (add-constraint-with-bounds
              (car rands-vars)
              (make-Dom max-n n rator-var))
             (loop (add1 n) (cdr rands-vars))))
   (add-constraint-with-bounds
     (make-Rng rator-var) alpha))]
[($ Cons the-car the-cdr)
 (let ([car-var (derive-top-term-constraints Gamma the-car)]
       [cdr-var (derive-top-term-constraints Gamma the-cdr)])
   (add-constraint-with-bounds *pair-token* alpha)
   (add-constraint-with-bounds car-var (make-car alpha))
   (add-constraint-with-bounds cdr-var (make-cdr alpha)))]
[($ Car-app the-cons)
 (let ([cons-var
          (derive-top-term-constraints Gamma the-cons)])
   (add-constraint-with-bounds (make-car cons-var) alpha))]
[($ Cdr-app the-cons)
 (let ([cons-var (derive-top-term-constraints Gamma the-cons)])
   (add-constraint-with-bounds (make-cdr cons-var) alpha))]
[($ Mathop op rand1 rand2)
 (derive-top-term-constraints Gamma rand1)
 (derive-top-term-constraints Gamma rand2)
 (add-constraint-with-bounds *num-token* alpha)]
[($ Cond preds values)
 (let* ([f (lambda (val)
            (derive-top-term-constraints Gamma val))]
        [value-vars (map f values)])
   (for-each f preds)
   (for-each
    (lambda (value-var)
      (add-constraint-with-bounds value-var alpha))
    value-vars))]
[($ Cond-else preds values else-exp)
 (let* ([f (lambda (val)
            (derive-top-term-constraints Gamma val))]
        [value-vars (map f values)]
        [else-var (derive-top-term-constraints Gamma else-exp)])
   (for-each f preds)
   (for-each
    (lambda (value-var)
      (add-constraint-with-bounds value-var alpha))
    value-vars)
   (add-constraint-with-bounds else-var alpha))]
[_ (error (format "derive-top-term-constraints: Unknown term ~a"
                 M))])
```

```
    ; associate M with fresh set variable
    (associate-term-with-set M alpha)

    ; return the set variable
    alpha))

; (listof T) -> void
(define (derive-top-level-constraints . top-terms)
  (let ([Gamma (make-constraint-env)])
    (for-each
     (lambda (t)
       (derive-top-term-constraints Gamma t))
     top-terms)))
```

8.9 Constraint propagation

The following code implements the propagation rules in Figure 5. We iterate over the list of all constraints exactly once. For each constraint, we search for a corresponding constraint that could be used to fire one of the propagation rules. When a constraint is added, we see if it triggers any rules itself. New constraints may appear as the result of the search when applying the rules to existing constraints.

```
;;; propagate.ss

; Set-exp -> bool
(define (constant-set-exp? tau)
  (or (Const? tau) (Label? tau) (Token? tau)))

; Set-exp -> bool
(define (selector? tau)
  (or (Dom? tau) (Rng? tau)
      (Car? tau) (Cdr? tau)))

; Set-exp -> bool
(define (var-or-selector? tau)
  (or (Set-var? tau) (selector? tau)))

; -> void
(define (propagate-constraints)
  (letrec ([add-constraint-and-propagate
            (lambda (lo hi)
              (let ([ct (make-constraint lo hi)])
                (when (add-constraint ct)
                      (propagate-one-constraint ct))))]
```

```
[fire-covariant-selector-rule
 ; for 1st premises
 (lambda (lo1 hi2s make-selector)
   (let ([hi3s (map make-selector (filter Set-var? hi2s))])
     (for-each
      (lambda (hi3)
        (add-constraint-and-propagate lo1 hi3))
      hi3s)))]
[fire-covariant-selector-rule-prime
 ; for 2nd premises
 (lambda (lo1 hi1 make-selector)
       (let ([svs (lookup-hi-and-filter Set-var?
                                        (make-selector lo1))]
             [hi2 (make-selector hi1)])
         (for-each
          (lambda (sv)
            (add-constraint-and-propagate sv hi2))
          svs)))]
[fire-contravariant-selector-rule
 (lambda (hi1 hi2s make-selector)
   (let ([lo3s (map make-selector (filter Set-var? hi2s))])
     (for-each
      (lambda (lo3)
        (add-constraint-and-propagate lo3 hi1))
      lo3s)))]
[lookup-and-filter-maker
 (lambda (lookup-proc)
   (lambda (pred set-exp)
     (filter pred
             (lookup-proc set-exp))))]
[lookup-lo-and-filter
 (lookup-and-filter-maker lookup-in-lo-constraint-table)]
[lookup-hi-and-filter
 (lookup-and-filter-maker lookup-in-hi-constraint-table)]
[propagate-one-constraint
 (lambda (ct)
   (let ([lo1 (constraint-lo ct)]
         [hi1 (constraint-hi ct)])

     ; rules are of form

     ;           ct1 ct2
     ;           -------
     ;             ct3

     ; consider ct as ct1, search for ct2
```

```
; rule trans-const
 (when (and (constant-set-exp? lo1)
       (Set-var? hi1))
  (let ([hi2s (lookup-lo-and-filter Set-var? hi1)])
    (for-each
     (lambda (hi2)
       (add-constraint-and-propagate lo1 hi2)
       (add-flow-edge hi2 hi1))
     hi2s)))
; rule trans-sel
(when (and (selector? hi1)
           (Set-var? lo1))
 (let ([hi2s (lookup-lo-and-filter Set-var? hi1)])
   (for-each
    (lambda (hi2)
      (add-constraint-and-propagate lo1 hi2))
    hi2s)))
; rule dom-prop
(when (and (Dom? lo1)
      (Set-var? hi1))
 (let ([hi2s (lookup-in-lo-constraint-table
              (Dom-set-var lo1))]
       [max-n (Dom-arity lo1)]
       [n (Dom-pos lo1)])
   (fire-contravariant-selector-rule
    hi1 hi2s
    (lambda (set-var)
      (make-Dom max-n n set-var)))))
; rule rng-prop
(when (and (Rng? hi1)
           (Set-var? lo1))
      (let ([hi2s (lookup-in-lo-constraint-table
              (Rng-set-var hi1))])
        (fire-covariant-selector-rule lo1 hi2s make-Rng)))
; rule car-prop
(when (and (Car? hi1)
           (Set-var? lo1))
      (let ([hi2s (lookup-in-lo-constraint-table
                   (Car-set-var hi1))])
        (fire-covariant-selector-rule lo1 hi2s make-Car)))
; rule cdr-prop
(when (and (Cdr? hi1)
           (Set-var? lo1))
      (let ([hi2s (lookup-in-lo-constraint-table
                   (Cdr-set-var hi1))])
```

```
        (fire-covariant-selector-rule lo1 hi2s make-Cdr)))

; consider ct as ct2, search for ct1

(when (and (Set-var? lo1)
           (Set-var? hi1))

  ; all rules but trans-sel fit here

  ; trans-const
  (for-each
   (lambda (c)
     (add-constraint-and-propagate c hi1)
     (add-flow-edge hi1 lo1))
   (lookup-hi-and-filter constant-set-exp? lo1))
  ; rng-prop
  (fire-covariant-selector-rule-prime lo1 hi1 make-Rng)
  ; dom-prop
  ; test all possible arities
  ; for each arity, test all possible arg positions
  (for-each
   (lambda (arity)
     (let loop ([pos 1])
       (when (<= pos arity)
             (let ([new-dom (make-Dom arity pos hi1)])
               (for-each
                (lambda (sv)
                  (add-constraint-and-propagate
                   new-dom
                   sv))
                (lookup-lo-and-filter
                 Set-var?
                 (make-Dom arity pos lo1))))
             (loop (add1 pos)))))
   (lookup-set-var-arities lo1))
  ; car-prop
  (fire-covariant-selector-rule-prime lo1 hi1 make-Car)
  ; cdr-prop
  (fire-covariant-selector-rule-prime lo1 hi1 make-Cdr))
; rule trans-sel
(when (and (selector? lo1)
           (Set-var? hi1))
      (for-each
       (lambda (sv)
         (add-constraint-and-propagate sv hi1))
       (lookup-hi-and-filter Set-var? lo1)))))])
```

```
    (for-each
     propagate-one-constraint
     *the-constraints*)))
```

8.10 Flow graphs

We store the flow graph as a hash table of edges, where the key is a set variable source node, and the value is a list of set variable target nodes.

```
;;; flow.ss

; (hash-table-of Set-var (listof Set-var))
; for example, key = alpha1 value = (alpha2,alpha3)
;  indicates directed edges (alpha1,alpha2) (alpha1,alpha3)
(define *flow-graph-edges* (make-hash-table))

; -> void
(define (reset-flow-graph)
  (set! *flow-graph-edges* (make-hash-table)))

; Set-var Set-var -> void
(define (add-flow-edge sv1 sv2)
  (let ([existing-nodes (hash-table-get *flow-graph-edges* sv1
                                        (lambda () '()))])
    (unless (memq sv2 existing-nodes)
            (hash-table-put! *flow-graph-edges*
                             sv1 (cons sv2 existing-nodes)))))

; Set-var -> (listof Set-var)
(define (get-flow-nodes set-var)
  (hash-table-get *flow-graph-edges* set-var
                  (lambda () '())))
```

8.11 Types

We build a datatype corresponding to the grammar on page 9. Notice that we do not actually create any new structures, since types are built from existing structures and ordinary lists.

```
;;; types.ss

; Type = Set-var
;      | (make-Const (union char string symbol bool number
;                     '() 'void 'empty))
;      | (list 'rec (list (Set-var Type) ...) Type) |
;      | (list 'union Type ...)
;      | (list 'cons Type Type)
```

```
;       | (list Type ... '-> Type)
```

We build a map from set variables to types, corresponding to the pairs found in the binder lists of recursive types. Those pairs are the same for all types, so memoizing that information is useful.

The `unfold` rule in Figure 6 requires knowledge of whether a set variable in a recursive type binder occurs in its corresponding type. We also memoize that information, since we need it repeatedly.

```
; (hash-table-of Set-var Type)
(define *type-table* (make-hash-table))

; Set-var Type -> void
(define (add-to-type-table sv omega)
  (hash-table-put! *type-table* sv omega))

; (hash-table-of Type (union #f (listof Set-var)))
(define *type-occurs-table* (make-hash-table))

; Set-var Type -> void
(define (add-to-type-occurs-table sv omega)
  (hash-table-put! *type-occurs-table*
                   sv (memq sv (get-type-set-vars omega))))

; Set-var Type -> void
(define (add-to-type-tables sv omega)
  (add-to-type-table sv omega)
  (add-to-type-occurs-table sv omega))

; -> void
(define (reset-type-tables)
  (set! *type-table* (make-hash-table))
  (set! *type-occurs-table* (make-hash-table)))

; Set-var -> Type
(define (lookup-in-type-table set-var)
  (hash-table-get *type-table* set-var
                  (lambda () *empty-type*)))

; Set-var -> (union (listof Set-var) #f)
(define (lookup-in-occurs-table sv)
  (hash-table-get *type-occurs-table* sv
                  (lambda () #f)))

; -> void
(define (build-type-tables)
  (let ([all-svs (get-all-set-vars)])
```

```
    (for-each
     (lambda (sv)
       (let ([omega (type-reduce (mk-type-aux sv))])
         (add-to-type-tables sv omega)))
     all-svs)))
```

We represent sets of types as lists. Hence our implementation of *Set2Type* is named `types->type`.

```
; (listof Type) -> Type
(define (types->type tys)
  (let ([filtered-tys (filter
                       (lambda (ty)
                         (not (eq? ty *empty-type*)))
                       tys)])
    (cond
     [(null? tys) *empty-type*]
     [(null? (cdr tys)) (car filtered-tys)]
     [else (mk-union-type tys)])))
```

Computing $\omega_{dom_n^m}$, on page 10, requires finding all set variable upper bounds given a set variable lower bound. We have to keep track of variables already seen, since there may be cycles in the constraints.

```
; Set-var -> (listof Set-var)
(define (set-var-upper-bounds set-var accum)
  (let* ([ubs (filter Set-var?
                      (lookup-in-lo-constraint-table set-var))]
         [new-ubs (filter
                   (lambda (ub) (not (memq ub accum)))
                   ubs)]
         [new-accum (cons set-var (append new-ubs accum))])
    (remove-duplicates-eq
     (cons
      set-var
      (apply append
             (map (lambda (ub)
                    (set-var-upper-bounds ub new-accum))
                  new-ubs))))))
```

Here is *MkTypeAux*:

```
; Set-var -> Type
(define (mk-type-aux set-var)
  (let* ([lookup-filtered-set-exp
          (lambda (pred set-exp)
            (types->type
             (filter pred
```

```
            (lookup-in-hi-constraint-table set-exp))))]
[omega-c
 (lookup-filtered-set-exp
  (lambda (s)
    (or (Const? s)
        (eq? s *num-token*))) set-var)]
[pair-token-or-empty
 (lookup-filtered-set-exp
  (lambda (set-exp)
    (eq? set-exp *pair-token*))
  set-var)]
[omega-cons
 (if (empty-type? pair-token-or-empty)
     *empty-type*
     (let ([omega-car
            (lookup-filtered-set-exp Set-var? (make-Car set-var))]
           [omega-cdr
            (lookup-filtered-set-exp Set-var? (make-Cdr set-var))])
       (mk-cons-type omega-car omega-cdr)))]
[fun-labels
 (filter Label? (lookup-in-hi-constraint-table set-var))]
[omega-fun
 (if (null? fun-labels)
     *empty-type*
     (let ([omega-rng
            (lookup-filtered-set-exp Set-var? (make-Rng set-var))])
       (types->type
        (map
         (lambda (ell)
           (let*
               ([max-arity (lookup-label-arity ell)]
                [omega-doms
                ; deltas = { delta | label-set-var <=* delta }
                 (let ([deltas (set-var-upper-bounds set-var '())])
                   (let loop ([n 1])
                     (if (> n max-arity)
                         '()
                         (cons
                          (types->type
                           (remove-duplicates-eq
                            (apply
                             append
                             (map
                              (lambda (delta)
                                (filter
                                 Set-var?
```

```
                                          (lookup-in-hi-constraint-table
                                           (make-Dom max-arity n delta))))
                                       deltas))))
                                   (loop (add1 n))))))])
                      (mk-fun-type omega-doms omega-rng)))
                  fun-labels))))])
    (types->type
     (list omega-c omega-cons omega-fun))))
```

Next are some utility procedures for handling types.

```
; sym -> Type -> bool
(define (make-prefix-constructed-type-pred prefix)
  (lambda (ty)
    (and (list? ty)
         (eq? (car ty) prefix))))

; (listof Type) -> Type
(define (mk-union-type elts)
  (cons 'union elts))

; Type -> bool
(define union-type?
  (make-prefix-constructed-type-pred 'union))

; Type -> (listof Type)
(define union-type-types cdr)

; (list (list (Set-Var Type) ...) Type [bool]) -> Type
; optional argument indicates whether to sort binders
(define (mk-rec-type binders body . sort?)
  (let ([actual-binders
         (cond
          [(null? sort?) binders]
          [(null? (cdr sort?))
           (if (car sort?)
               (quicksort
                binders
                (lambda (b _)
                  (not (set-var-occurs-in-binding? (car b)))))
               binders)]
          [else (error "mk-rec-type: too many arguments")])])
  ; maintain invariant that binders where alpha_i not in omega_i
  ; listed first
  (list 'rec actual-binders body)))

; Type -> bool
```

```
(define rec-type?
  (make-prefix-constructed-type-pred 'rec))

; Type -> Type
(define rec-type-binders cadr)

; Type -> Type
(define rec-type-body caddr)

; Type Type -> Type
(define (mk-cons-type the-car the-cdr)
  (list 'cons the-car the-cdr))

; Type -> bool
(define cons-type?
  (make-prefix-constructed-type-pred 'cons))

; Type -> Type
(define cons-type-car cadr)

; Type -> Type
(define cons-type-cdr caddr)

; (listof Type) Type -> Type
(define (mk-fun-type doms rng)
  (append doms (list '-> rng)))

; Type -> bool
(define (fun-type? ty)
  (and (list? ty)
       (memq '-> ty)))

; Type -> (listof Type)
(define (fun-type-doms ty)
  (let loop ([lst ty])
    (let ([the-car (car lst)])
      (if (eq? the-car '->)
          '()
          (cons the-car
                (loop (cdr lst)))))))

; Type -> Type
(define (fun-type-rng ty)
  (cadr (memq '-> ty)))

; (TST -> bool) -> (listof TST) -> (listof TST)
```

```
(define (remove-dup-maker mem-pred)
  (letrec ([rd
            (lambda (lst)
              (if (null? lst) '()
                  (let ([the-car (car lst)]
                        [the-cdr (cdr lst)])
                    (if (mem-pred the-car the-cdr)
                        (rd the-cdr)
                        (cons the-car
                              (rd the-cdr))))))])
    rd))

; (listof TST) -> (listof TST)
(define remove-duplicates-eq
  (remove-dup-maker memq))

; (listof TST) -> (listof TST)
(define remove-duplicates-equal
  (remove-dup-maker member))

; Type -> (listof Set-var)
(define (get-type-set-vars ty)
  (let ([map-and-flatten
         (lambda (lst)
           (apply append (map get-type-set-vars lst)))])
    (cond
     ; constructors
     [(rec-type? ty)
      (let ([binder-fvs
             (map-and-flatten
                    (map cadr (rec-type-binders ty)))]
            [body-fvs (get-type-set-vars (rec-type-body ty))])
        (remove-duplicates-eq
         (append binder-fvs body-fvs)))]
     [(or (union-type? ty) (cons-type? ty))
      (remove-duplicates-eq (map-and-flatten (cdr ty)))]
     [(fun-type? ty)
      (remove-duplicates-eq
       (map-and-flatten
        (filter
         (lambda (elt) (not (eq? elt '->))) ty)))]
        [(Set-var? ty) (list ty)]
     [(Const? ty) '()]
     [(Token? ty) '()]
     [else (error "get-type-set-vars: Unknown type" ty)])))
```

```
; Type (list Set-var Type) -> Type
(define (type-subst ty sub)
  (cond
   [(union-type? ty)
    (mk-union-type (map
                    (lambda (a-ty) (type-subst a-ty sub))
                    (union-type-types ty)))]
   [(cons-type? ty)
    (mk-cons-type (type-subst (cons-type-car ty) sub)
                  (type-subst (cons-type-cdr ty) sub))]
   [(fun-type? ty)
    (mk-fun-type (map (lambda (a-ty)
                        (type-subst a-ty sub))
                      (fun-type-doms ty))
                 (type-subst (fun-type-rng ty) sub))]
   [(rec-type? ty)
    (let* ([binders (rec-type-binders ty)]
           [svs (map car binders)]
           [new-omegas (map (lambda (omega)
                              (type-subst omega sub))
                            (map cadr binders))]
           [new-binders (map list svs new-omegas)]
           [new-body (type-subst (rec-type-body ty) sub)])
      ; binders are already sorted according to occurs-check
      ; sorting unaffected by substitution
      (mk-rec-type new-binders new-body))]
   [(Const? ty) ty]
   [(Token? ty) ty]
   [(Set-var? ty)
    (if (eq? ty (car sub))
        (cadr sub)
        ty)]
   [else
    (error "type-subst: Unknown type" ty)]))

; Set-var -> (union #f (listof Set-var))
(define set-var-occurs-in-binding? lookup-in-occurs-table)

; Type -> bool
(define (closed-type? ty)
  (null? (get-type-set-vars ty)))
```

The type reductions in Figure 6 are implemented by `type-reduce`.

```
; Type -> Type
(define (type-reduce ty)
  (cond
```

```
[(rec-type? ty)
 (let ([bs (rec-type-binders ty)]
       [body (rec-type-body ty)])
   (if (or (null? bs)
           (closed-type? body))
       (type-reduce body)
       (let* ([first-b (car bs)]
              ; rely on sorting invariant of binders
              ; if not first binder not eligible, done with reduction
              [eligible-sub?
               (not (set-var-occurs-in-binding? (car first-b)))])
         (if eligible-sub?
             (type-reduce
              (type-subst
               (mk-rec-type
                (cdr bs)
                (type-reduce body))
               first-b))
             ty))))]
[(union-type? ty)
 (let* ([elts (union-type-types ty)]
        [unique-elts (remove-duplicates-equal elts)]
        ; we have at most one *empty-type*, OK to stop
        ; filtering when we find one -- otherwise, we'd
        ; use MzScheme's filter
        [filtered-elts
         (let loop ([elts unique-elts])
           (if (null? elts)
               '()
               (let ([the-car (car elts)])
                 (if (eq? the-car *empty-type*)
                     (cdr elts)
                     (cons the-car (loop (cdr elts)))))))]
        ; rule union
        [reduced-elts (map type-reduce filtered-elts)]
        ; rule union-merge
        [merged-elts
         (let loop ([elts reduced-elts])
           (if (null? elts)
               '()
               (let ([the-car (car elts)]
                     [tail (loop (cdr elts))])
                 (if (union-type? the-car)
                     (append (union-type-types the-car) tail)
                     (cons the-car tail)))))])
   (cond
```

```
      ; special case
      [(null? merged-elts) *empty-type*]
      ; union-elim
      [(null? (cdr merged-elts)) (type-reduce (car merged-elts))]
      ; done
      [(equal? elts merged-elts) ty]
      [else (type-reduce (mk-union-type merged-elts))]))]
  [(cons-type? ty) ; rule cons
   (let* ([old-car (cons-type-car ty)]
          [old-cdr (cons-type-cdr ty)]
          [new-car (type-reduce old-car)]
          [new-cdr (type-reduce old-cdr)])
     (if (and (equal? old-car new-car)
              (equal? old-cdr new-cdr))
         ty
         (mk-cons-type new-car new-cdr)))]
  [(fun-type? ty) ; rule fun
   (let ([doms (map type-reduce (fun-type-doms ty))]
         [rng (type-reduce (fun-type-rng ty))])
     (mk-fun-type doms rng))]
  [else ty]))
```

Here is *MkType*:

```
; Set-var -> Type
(define (mk-type set-var)
  (let ([omega-set-var (lookup-in-type-table set-var)]
        [all-svs (get-all-set-vars)])
    (type-reduce
     (mk-rec-type
      (map list all-svs (map lookup-in-type-table all-svs))
      omega-set-var #t))))
```

8.12 Static debugging

The procedure `check-term-error` indicates possibly unsafe operations, as described in Section 7.

```
;;; errors.ss

; T -> void
(define (check-term-error t)
  (let* ([lookup-constant-los
          (lambda (set-var)
            (filter
             (lambda (tau)
               (not (var-or-selector? tau)))
```

```
          (lookup-in-hi-constraint-table set-var)))]
       [check-for-pairs-only
        (lambda (the-cons term-name)
          (let* ([cons-set-var (lookup-in-term-map the-cons)]
                 [cons-los (lookup-constant-los cons-set-var)])
            (when (ormap
                   (lambda (lo)
                     (not (eq? lo *pair-token*)))
                   cons-los)
                  (printf "~a: ~a may be unsafe~n"
                          term-name (pretty-term t)))))])
(match t
       [($ Define name t)
        (if (Lambda? t)
            (check-term-error (Lambda-body t))
            (check-term-error t))]
       [($ Var _) void]
       [($ Const _) void]
       [($ App rator rands)
        (let* ([rator-set-var (lookup-in-term-map rator)]
               [rator-los (lookup-constant-los rator-set-var)]
               [rator-labels (filter Label? rator-los)]
               [app-arity (length rands)])
          ; check for arity mismatch
          (for-each
           (lambda (ell)
             (let ([ell-arity (lookup-label-arity ell)])
               (unless (eq? app-arity ell-arity)
                       (printf (string-append
                                "Procedure defined by ~a~n"
                                "may give an arity mismatch at call site "
                                "~a~n")
                               (pretty-term (lookup-label-lambda ell))
                               (pretty-term t)))))
           rator-labels)
          ; check for unsafe application
          (when (and (not (null? rator-los))
                     (ormap
                      (lambda (lo) (not (Label? lo)))
                      rator-los))
                (printf "Application: ~a may be unsafe~n"
                        (pretty-term t)))
          (check-term-error rator)
          (for-each check-term-error rands))]
       [($ Cons the-car the-cdr)
        (let* ([cdr-set-var (lookup-in-term-map the-cdr)]
```

```
               [rator-los (lookup-constant-los cdr-set-var)])
          (when (and (not (null? rator-los))
                     (ormap
                      (lambda (lo)
                        (not (or (eq? lo *empty-list*)
                                 (eq? lo *pair-token*))))
                      rator-los))
                (printf "Cons: ~a may be unsafe~n"
                        (pretty-term t)))
          (check-term-error the-car)
          (check-term-error the-cdr))]
       [($ Car-app the-cons)
        (check-for-pairs-only the-cons "Car")
        (check-term-error the-cons)]
       [($ Cdr-app the-cons)
        (check-for-pairs-only the-cons "Cdr")
        (check-term-error the-cons)]
       [($ Mathop op rand1 rand2)
        (let* ([rand1-set-var (lookup-in-term-map rand1)]
               [rand1-los (lookup-constant-los rand1-set-var)]
               [rand2-set-var (lookup-in-term-map rand2)]
               [rand2-los (lookup-constant-los rand2-set-var)])
          (when (and
                 (not (null? rand1-los))
                 (not (null? rand2-los))
                 (ormap
                  (lambda (lo)
                    (not (or (eq? lo *num-token*)
                             (and (Const? lo)
                                  (number? (Const-val lo))))))
                  (append rand1-los rand2-los)))
                (printf "~a: ~a may be unsafe~n"
                        op (pretty-term t))))
        (check-term-error rand1)
        (check-term-error rand2) ]
       [($ Cond preds exps)
        (for-each check-term-error preds)
        (for-each check-term-error exps)]
       [($ Cond-else preds exps else-exp)
        (for-each check-term-error preds)
        (for-each check-term-error exps)
        (check-term-error else-exp)]
       [_ (error (format "check-term-error: Unknown term ~a" t))])))

; (listof T) -> void
(define (check-for-errors ts)
```

```
  (for-each check-term-error ts))
```

8.13 Analyzer driver

Finally we have the main driver procedure, which takes a program given as a list of `T`'s:

```
;;; analyze.ss

(require-library "functio.ss")
(require-library "match.ss")

; (listof T) -> void
(define (analyze . ts)
  (reset-system)
  (for-each derive-top-level-constraints ts)
  (propagate-constraints)
  (build-type-tables)
  (for-each display-term-annotation ts)
  (check-for-errors ts))
```

8.14 Obtaining the code

All source code is available via anonymous ftp from

`ftp.cs.rice.edu/public/steck/newspidey.tar.gz`.

See the `README` file for directions on loading and running.

References

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(1):79–111, 1999.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, MA/New York, 1990.

[3] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, Houston, Texas, May 1997.

[4] Cormac Flanagan. *PLT MrSpidey: Static Debugger Manual*. Rice University, 1999.

[5] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. *ACM Transactions on Programming Languages and Systems*, 21(2):369–415, February 1999.

[6] Nevin Heintze and Joxan Jaffar. Set constraints and set-based analysis. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 281–298. Springer-Verlag, 1994.

[7] François Pottier. A framework for type inference with subtyping. In *Proc. 3rd ACM SIGPLAN International Conf. on Functional Programming (ICFP'98)*, pages 228–238, September 1998.

[8] Rice University. *PLT DrScheme: Programming Environment Manual*, 1999. Version 102.