Catching Bugs in the Web of Program Invariants

Cormac Flanagan Matthew Flatt Shriram Krishnamurthi Stephanie Weirich
Matthias Felleisen

Department of Computer Science,
Rice University,
Houston, Texas 77005-1892*
cormac@cs.rice.edu

Abstract

MrSpidey is a user-friendly, interactive static debugger for Scheme. A static debugger supplements the standard debugger by analyzing the program and pinpointing those program operations that may cause run-time errors such as dereferencing the null pointer or applying non-functions. The program analysis of MrSpidey computes value set descriptions for each term in the program and constructs a value flow graph connecting the set descriptions. Using the set descriptions, MrSpidey can identify and highlight potentially erroneous program operations, whose cause the programmer can then explore by selectively exposing portions of the value flow graph.

1 Introduction

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 array indexing 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 [17] 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 [20]. In contrast, safe languages such as SML [21] and Scheme [3] equip all program operations with appropriate run-time checks. These checks guarantee that misapplications of program operations immediately raise an error signal, instead of returning random bit-patterns. Although this solution ensures that programs don't return random results, it is unsatisfactory because errors are not signaled until run-time. What is needed instead, is a static analysis tool that assists the programmer in verifying the preconditions of program operations. This kind of tool is a static debugger.

Recent advances in proof technology have brought static debugging within reach. Methods like abstract interpretation [5], control-flow analysis [25, 26, 14] or set-based analysis [11, 10] establish invariants about the sets of values that variables and expressions may assume. Thus, if an array index expression does not assume values outside of the appropriate range, an array bound check is superfluous, and an indexing error will never be signaled for this expression. Or, if the value set of a function variable only contains closures, the function application does not need to be checked, and will always succeed. Past research on static debuggers mainly focused on the synthesis of the invariants [2]. 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 set 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.

We have developed a static debugger for Scheme, called MrSpidey, which allows the programmer to browse program invariants and their derivations. We selected set-based analysis as the underlying proof technology of MrSpidey, for the following three reasons. First, set-based analysis produces accurate program invariants for Scheme-like languages, even in the presence of complex control-flow and data-flow patterns. Second, set-based analysis is intuitive. It interprets program operations as naïve set-theoretic operations on sets of run-time values,

*The authors are supported in part by NSF grants CCR 91-22518 and CDA-9414170.

and propagates these sets of values along the program's data-flow paths, in a manner that is easily understood by the programmer. Third, by appropriately annotating the set-based analysis algorithm, we can provide a supporting explanation in the form of a *value flow graph* for each invariant produced by the analysis.

Since MrSpidey is to be used as part of the typical program development cycle, we have integrated it with DrScheme, of our program development environment. On demand, MrSpidey marks up the program in the editor without distorting its lexical or syntactic structure. The mark-ups visibly identify those program operations that are not provably safe. Associated hyper-links provide:

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

The programmer can browse the information and thus improve his understanding of the program's execution behavior.

The rest of the paper proceeds as follows. The following section outlines the information computed by MrSpidey. The user interface that presents this information to the programmer is described in the third section, and the fourth section presents the results of a preliminary experiment evaluating the usefulness of MrSpidey. Technical details involved in the implementation are covered in the fifth section, and the sixth section describes related work. The seventh section presents our conclusions and future research directions.

2 Set-Based Analysis

MrSpidey's underlying set-based analysis algorithm handles all fundamental constructs of Scheme, including conditionals, assignable variables, mutable structures, and first-class continuations. In this section, we outline the information produced by the analysis in terms of a simple functional subset of Scheme. For a full presentation of set-based analysis for a realistic language, we refer the interested reader to a related report [6].

2.1 The Source Language

The sample language is a simplified, λ-calculus-like language: see Figure 1. The language includes the primitives **cons**, **car**, and **cdr** for list manipulation, which will serve to illustrate the treatment of primitive operations, and a number of basic constants. The semantics of the source language can be formulated as a variant of Plotkins λ_v-calculus [22]. Each term in the language is labeled, and we assume that all labels in a program are distinct. For clarity, labels are occasionally omitted.

$$
\begin{array}{rcllll}
P & \in & \mathit{Program} & ::= & (\mathbf{define}\ x\ M)\ \ldots \\
M, N & \in & \Lambda & ::= & V^l \\
 & & & & \mid (M\ M)^l \\
 & & & & \mid (\mathbf{cons}\ M\ M)^l \\
 & & & & \mid (\mathbf{car}\ M)^l \\
 & & & & \mid (\mathbf{cdr}\ M)^l \\
V & \in & \mathit{Value} & ::= & c \mid x \\
 & & & & \mid (\lambda x.\, M) \\
 & & & & \mid (\mathbf{cons}\ V\ V) \\
c & \in & \mathit{Const} & = & \mathit{Num} \cup \{\texttt{nil}, \ldots\} \\
n & \in & \mathit{Num} & = & \{0, 1, 2, \ldots\} \\
x & \in & \mathit{Vars} & = & \{x, y, z, \ldots\} \\
l & \in & \mathit{Label} & &
\end{array}
$$

Figure 1: The Source Language Λ

2.2 Set-Based Analysis

Set-based analysis computes information about the sets of values that program variables and terms may assume during execution. Because these sets of values are typically infinite, set-based analysis uses a finite number of program-dependent *abstract values*. Each abstract value corresponds to a particular constructor expression in the analyzed program, and represents the set of run-time values that can be created by that constructor expression. The set of abstract values used for the analysis of a program P is:[1]

$$
\begin{array}{rl}
\mathit{AbsValue}_P = & \{(\mathbf{cons}\ l_1\ l_2) \mid (\mathbf{cons}\ M_1^{l_1}\ M_2^{l_2}) \in P\} \\
 & \cup\{(\lambda x.\, M) \mid (\lambda x.\, M) \in P\} \\
 & \cup\{c \mid c \in P\}
\end{array}
$$

The abstract value $(\mathbf{cons}\ l_1\ l_2)$ represents the set of values $(\mathbf{cons}\ V_1\ V_2)$ that may be returned by the expression $(\mathbf{cons}\ M_1^{l_1}\ M_2^{l_2})$, where each V_i is a possible value of the term $M_i^{l_i}$. Similarly, the abstract value $(\lambda x.\, M)$ represents the set of closures that may be created from the λ-expression $(\lambda x.\, M)$. The set of abstract values also contains all constants occurring in the program.

Set-based analysis produces a finite table called an *abstract store*, which maps variables and labels to sets of abstract values:

$$\mathcal{S} \in \mathit{AbsStore} = (\mathit{Vars} \cup \mathit{Label}) \longrightarrow \mathcal{P}(\mathit{AbsValue}_P)$$

An abstract store is *valid* if it conservatively approximates the sets of values that variables and terms assume during an execution.

2.3 Deriving Abstract Stores

MrSpidey employs a two stage algorithm to derive a valid abstract store for a program. First, it derives constraints in a syntax-directed manner from the program text. These constraints conservatively approximate the

[1] We use the following notations throughout the paper: $\mathcal{P}$ denotes the power-set constructor; $f : A \longrightarrow B$ denotes that f is a total function from A to B; and $M \in P$ denotes that the term M occurs in the program P.

dataflow relationships of the analyzed program. Second, it determines the minimal (*i.e.*, most accurate) abstract store satisfying these constraints. This abstract store is a valid abstract store for the analyzed program.

Let us illustrate the syntax-directed derivation of constraints for two kinds of sentences:

- $(\mathbf{cons}\ M_1^{l_1}\ M_2^{l_2})^l$

  The evaluation of this term produces the result $(\mathbf{cons}\ V_1\ V_2)$, if V_1 and V_2 are values of the terms $M_1^{l_1}$ and $M_2^{l_2}$ respectively. Since all such results are represented by the abstract value $(\mathbf{cons}\ l_1\ l_2)$, the analyzer adds the constraint

  $$(\mathbf{cons}\ l_1\ l_2) \in \mathcal{S}(l)$$

  to the global set of program constraints, which ensures that $\mathcal{S}(l)$ represents all possible results of the `cons` expression.

- $(M_1^{l_1}\ M_2^{l_2})^{l_3}$

  Suppose the value set for the function expression $M_1^{l_1}$ includes the function $(\lambda x.\, N^l)$. Then the variable x may be bound to the result of the argument expression $M_2^{l_2}$. In addition, the result of the function body N^l is returned as the result of the application expression itself. By adding the constraint

  $$\frac{(\lambda x.\, N^L) \in \mathcal{S}(l_1)}{\mathcal{S}(l_2) \subseteq \mathcal{S}(x) \qquad \mathcal{S}(L) \subseteq \mathcal{S}(l_3)}$$

  where L ranges over the set of labels, the analyzer captures this potential flow of values.

Constraints for the remaining classes of terms can be constructed in an analogous manner. Solving the derived constraints to produce a valid abstract store is straightforward [11].

2.4 A Sample Analysis

Consider the following toy program:

$$
\begin{array}{l}
(\mathbf{define}\ \mathit{sum} \\
\quad (\lambda\, \mathit{tree}. \\
\quad\quad (\mathbf{if}\ (\mathit{number?}\ \mathit{tree}) \\
\quad\quad\quad \mathit{tree} \\
\quad\quad\quad (+\ (\mathit{sum}\ (\mathbf{car}\ \mathit{tree}^{l_1})^{l_2}) \\
\quad\quad\quad\quad (\mathit{sum}\ (\mathbf{cdr}\ \mathit{tree})))))) \\
\\
(\mathit{sum}\ (\mathbf{cons}\ (\mathbf{cons}\ \mathtt{nil}^{l_3}\ 1^{l_4})^{l_5}\ 2^{l_6})^{l_7})
\end{array}
$$

The program defines the function *sum*, which computes the sum of all leaves in a binary numeric tree. Such a tree is either a leaf, represented as a number, or an interior node containing two sub-trees, represented as a pair. However, *sum* is applied to the ill-formed tree (**cons** (**cons** `nil` 1) 2). When executed, the program raises an error because the primitive operation **car** is applied to the inappropriate argument `nil`.

$$(\lambda tree.\ \cdots) \in \mathcal{S}(sum) \tag{1}$$

$$\mathcal{S}(tree) \setminus Num \subseteq \mathcal{S}(l_1) \tag{2}$$

$$\frac{(\mathbf{cons}\ L_x\ L_y) \in \mathcal{S}(l_1)}{\mathcal{S}(L_x) \subseteq \mathcal{S}(l_2)} \tag{3}$$

$$\frac{(\lambda x.\, N) \in \mathcal{S}(sum)}{\mathcal{S}(l_2) \subseteq \mathcal{S}(x)} \tag{4}$$

$$\mathtt{nil} \in \mathcal{S}(l_3) \tag{5}$$

$$1 \in \mathcal{S}(l_4) \tag{6}$$

$$(\mathbf{cons}\ l_3\ l_4) \in \mathcal{S}(l_5) \tag{7}$$

$$2 \in \mathcal{S}(l_6) \tag{8}$$

$$(\mathbf{cons}\ l_5\ l_6) \in \mathcal{S}(l_7) \tag{9}$$

$$\frac{(\lambda x.\, N) \in \mathcal{S}(sum)}{\mathcal{S}(l_7) \subseteq \mathcal{S}(x)} \tag{10}$$

Figure 2: Simplified Constraints for *sum*

Since the complete list of set constraints for this program is quite long, we present only (simplified versions of) the set constraints that affect position l_1 which represents the potential arguments of **car**: see Figure 2.

Constraints (4) and (10) model the flow of argument values into the formal parameter x at two of the application sites of the function *sum*. Constraint (3) models the behavior of the operation **car**. Constraints (1) and (5) through (9) arise from syntactic values in the program. Constraint (2) captures the notion that the possible values of *tree* in the else part of the conditional expression cannot include numbers, because of the *number?* predicate.

To solve these constraints for $\mathcal{S}(l_1)$, we derive the implied invariants:

From (1), (10) :	$\mathcal{S}(l_7) \subseteq \mathcal{S}(tree)$	(11)
From (9), (11) :	$(\mathbf{cons}\ l_5\ l_6) \in \mathcal{S}(tree)$	(12)
From (2), (12) :	$(\mathbf{cons}\ l_5\ l_6) \in \mathcal{S}(l_1)$	(13)
From (3), (13) :	$\mathcal{S}(l_5) \subseteq \mathcal{S}(l_2)$	(14)
From (7), (14) :	$(\mathbf{cons}\ l_3\ l_4) \in \mathcal{S}(l_2)$	(15)
From (1), (4) :	$\mathcal{S}(l_2) \subseteq \mathcal{S}(tree)$	(16)
From (15), (16) :	$(\mathbf{cons}\ l_3\ l_4) \in \mathcal{S}(tree)$	(17)
From (2), (17) :	$(\mathbf{cons}\ l_3\ l_4) \in \mathcal{S}(l_1)$	(18)
From (3), (18) :	$\mathcal{S}(l_3) \subseteq \mathcal{S}(l_2)$	(19)
From (5), (19) :	$\mathtt{nil} \in \mathcal{S}(l_2)$	(20)
From (16), (20) :	$\mathtt{nil} \in \mathcal{S}(tree)$	(21)
From (2), (21) :	$\mathtt{nil} \in \mathcal{S}(l_1)$	(22)

No further invariants relating to $\mathcal{S}(l_1)$ are implied by the set constraints. Hence:

$$\mathcal{S}(l_1) = \{(\textbf{cons } l_3\ l_4), (\textbf{cons } l_5\ l_6), \texttt{nil}\}$$

This information provides a warning that the set of arguments of the operation **car** may include the inappropriate value `nil`.

2.5 Identifying Potential Run-Time Errors

By inspecting the set invariants for the arguments of each program operation, MrSpidey can identify those program operations that may cause run-time errors and can flag them for inspection by the programmer. For example, in the program *sum*, since the value set at position l_1 contains the value `nil`, the evaluation of $(\textbf{car}\ tree^{l_1})$ may raise a run-time error. An inspection of the value flow graph explains why `nil` may appear at l_1.

At the moment, MrSpidey checks the uses of Scheme primitives, the function position of applications, and the parameter lists of functions (for potential arity conflicts). Future extensions are discussed below.

2.6 Value Flow Information

While deriving an abstract store for the analyzed program, the set-based analysis algorithm also constructs a *flow graph* [14] from the subset relations. The flow graph models how values "flow" through a program during an execution, and provides an intuitive explanation for each value-set invariant produced by the analysis.

Let us illustrate this idea by considering how the value `nil` flows through the program *sum*. During an execution, the expression $\texttt{nil}^{l_3}$ simply returns the value `nil`, which becomes the first element of the pair created by $(\textbf{cons}\ \texttt{nil}^{l_3}\ 1^{l_4})^{l_5}$. Since this pair is a result value of the expression $tree^{l_1}$, the value `nil` is extracted by $(\textbf{car}\ tree^{l_1})^{l_2}$. This value is then bound to the formal parameter *tree* via the function call $(sum\ (\textbf{car}\ tree^{l_1})^{l_2})$, and gets returned as the result of the expression $tree^{l_1}$.

This flow of values is modeled by the subset invariants produced during set-based analysis. The specific invariants that describe the flow of the value `nil` through the program, from the constructor expression $\texttt{nil}^{l_3}$ to the expression $tree^{l_1}$, are:

$$\begin{array}{rcl} \texttt{nil} & \in & \mathcal{S}(l_3) \\ \mathcal{S}(l_3) & \subseteq & \mathcal{S}(l_2) \\ \mathcal{S}(l_2) & \subseteq & \mathcal{S}(tree) \\ \mathcal{S}(tree) \setminus Num & \subseteq & \mathcal{S}(l_1) \end{array}$$

3 The User Interface of MrSpidey

A program invariant browser must fit seamlessly into a programmer's work pattern. It must provide the programmer with useful information in a natural, easily accessible manner, and with a minimum of disruption to the program development cycle. For these reasons, we integrated MrSpidey with DrScheme, our Scheme programming environment.

MrSpidey uses *program mark-ups* to characterize a program's run-time behavior in an easily accessible manner. The mark-ups are simple font and color changes that do not affect the lexical or syntactic structure of the program. They represent information about the program's behavior. By clicking on one of the marked-up tokens or phrases, a programmer can make the information visible.

3.1 Identifying Potential Run-Time Errors

Program operations that may signal run-time errors during an execution are a natural starting point in the static debugging process. MrSpidey identifies these potentially erroneous operations by highlighting them 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). A run-time argument check is required at each of these potentially faulty operations. Primitive operations that never raise errors are shown in green. These operations do not require run-time checks. Any function that may be applied to an incorrect number of arguments is highlighted by displaying the `lambda` keyword in red (or underlined), and any application expression where the function subexpression may return a non-closure is highlighted by displaying the enclosing parentheses in red (or underlined) . Figure 3 contains examples of these three kinds of potential errors.

MrSpidey also presents summary information describing the number and type of potential run-time errors in each top-level definition, together with a hyperlink to that definition. By following these hyperlinks, the programmer can directly access the potentially erroneous expressions.

3.2 Presenting Value Set Information

MrSpidey provides an inferred value set invariant for each variable and term in the program. Because the value set representation used in abstract stores is verbose and difficult to read, MrSpidey uses the following *set-description language* (*SDL*) to describe these sets:

$$\begin{array}{rclll} \tau & \in & SDL & ::= & c \\ &&&& \mid (\lambda x.\, M) \\ &&&& \mid (\textbf{cons}\ \tau\ \tau) \\ &&&& \mid (+\ \tau_1\ \ldots\ \tau_n) \\ &&&& \mid (\textbf{rec}\ ([\alpha_1\ \tau_1]\ \ldots\ [\alpha_n\ \tau_n])\ \tau) \\ &&&& \mid \alpha \\ \alpha & \in & SDV & = & \{\alpha, \beta, \gamma, \ldots\} \end{array}$$

The expression $(+\ \tau_1\ \ldots\ \tau_n)$ denotes the union of the sets of values described by τ_1 through τ_n. The recursive set-description expression $(\textbf{rec}\ ([\alpha_1\ \tau_1]\ \ldots\ [\alpha_n\ \tau_n])\ \tau)$

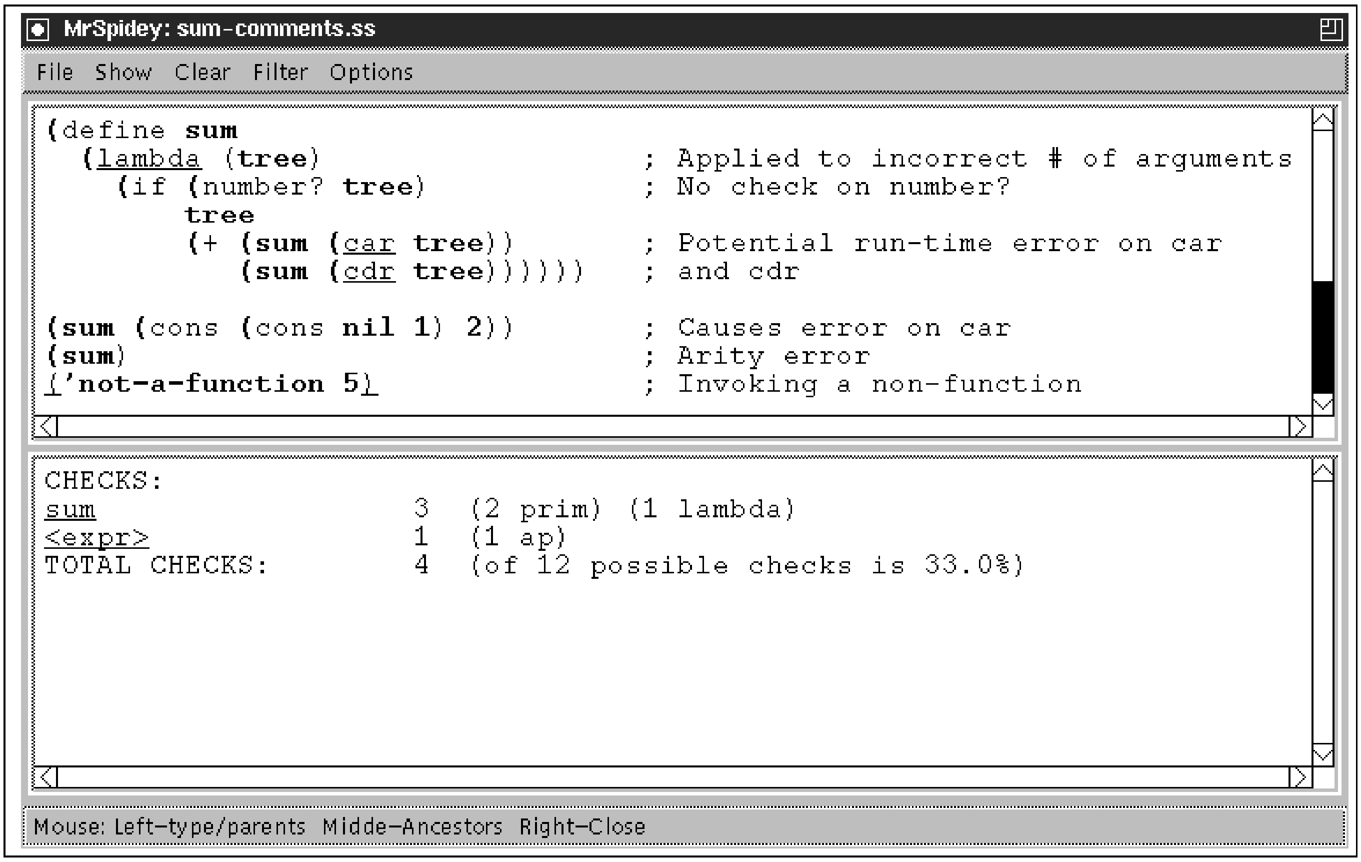

Figure 3: Identifying Potential Run-Time Errors

binds the *set-description variables* (*SDV*) $\alpha_1, \ldots, \alpha_n$, and these bindings are visible within $\tau_1, \ldots, \tau_n, \tau$. The meaning of this recursive set-description expression is the set of values described by τ, where each α_i is bound to τ_i. The transformation from an abstract store to set-description expressions is described in Subsection 5.2.

MrSpidey computes a closed set-description expression for each variable and term in the program from the abstract store produced by set-based analysis. It relates each program variable to its set-description expression via a hyperlink on that variable. It also relates each compound term in the program with a set-description expression via a hyperlink on the opening parenthesis of that term. Clicking on a hyperlink causes a box containing the corresponding set-description expression to be inserted to the right of the phrase in the buffer. Figure 4 shows the set-description expression displayed by clicking on the variable *tree*.

3.3 The Value Flow Browser

During the constraint derivation phase of the set-based analysis, MrSpidey constructs a value flow graph from subset relations. The value flow graph models the flow of values between various points in the program. Each edge in this graph is presented 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. To inspect the incoming edges for a given program term, the programmer clicks on the value set invariant of that term. Figure 5 shows the incoming edges for the parameter *tree*.

Hyperlinks 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 larger programs. Clicking on one end of an arrow moves the focus of the editor buffer to the term at the other end of the arrow, and also causes the value set invariant for that term to be displayed.

Using these facilities, a programmer who encounters a surprising value set invariant can proceed in an iterative fashion to expose the portions of the value flow graph that influence that invariant. To expedite this iterative process, MrSpidey also provides an *ancestor* facility that automatically exposes all portions of the value flow graph that influence a particular invariant, thus providing the programmer with a complete explanation for that invariant.

In some cases, the number of arrows presented by

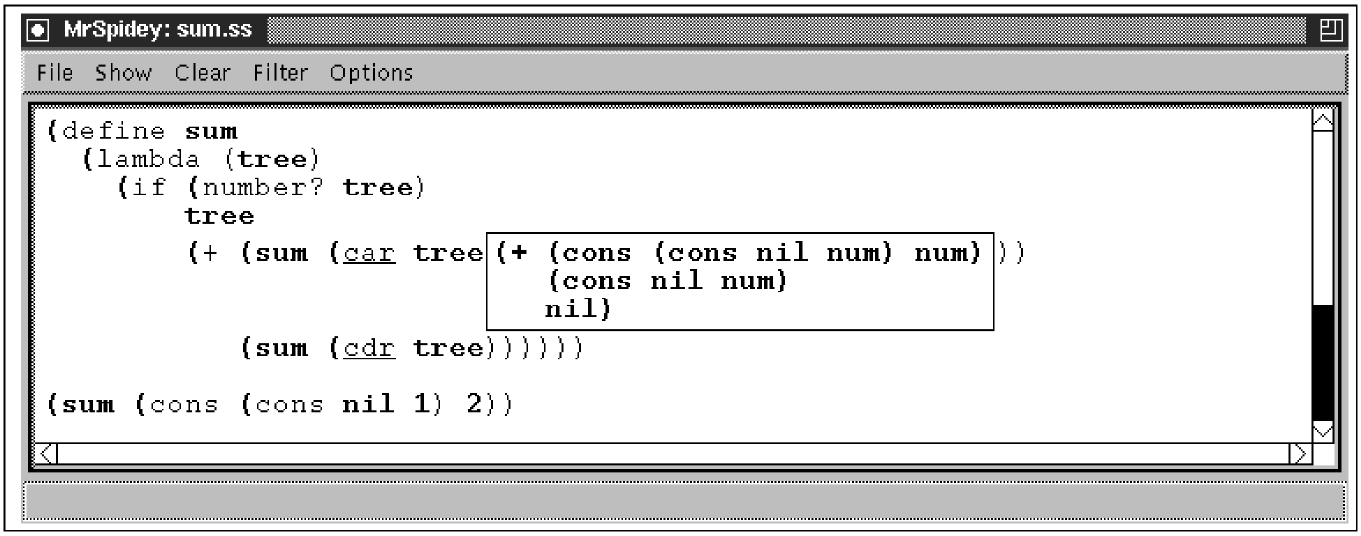

Figure 4: Value Set Information

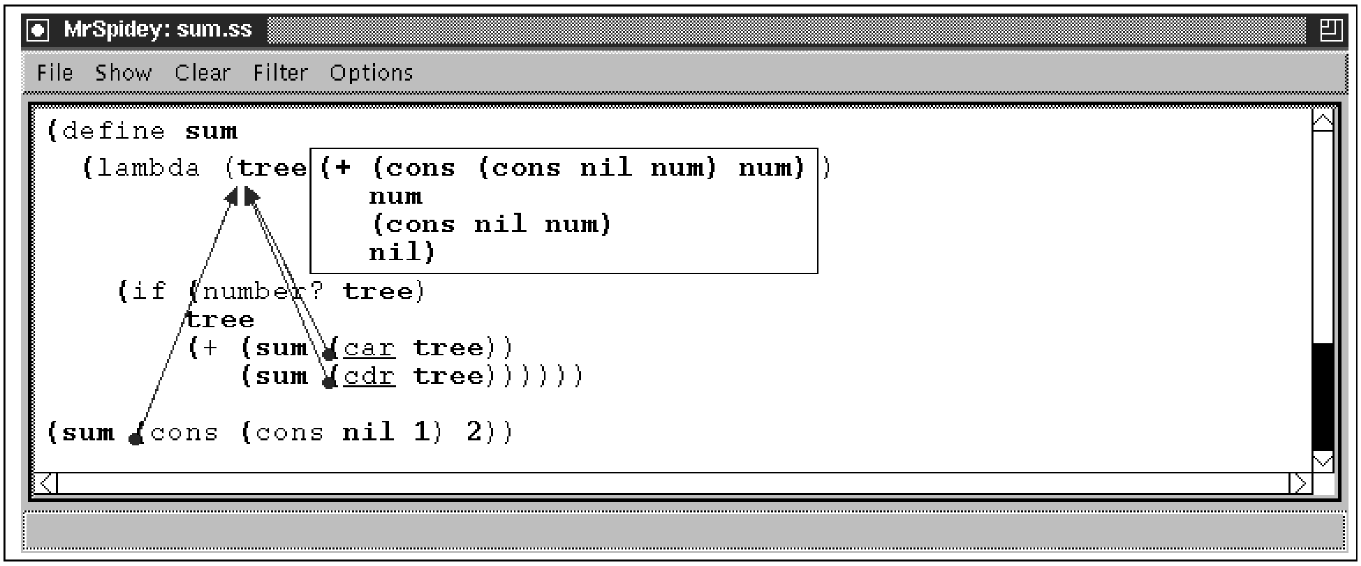

Figure 5: Value Source Information

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 edges 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.

3.4 A Sample Debugging Session

To illustrate the effectiveness of MrSpidey as a static program debugger, we describe how this tool can be used to identify and eliminate the potential run-time error in the program *sum*.

When MrSpidey is invoked, the primitive operation **car** is highlighted, indicating that this operation may raise a run-time error. Inspecting the value set for the operation argument, *tree*, (see Figure 4) shows that this set includes the inappropriate argument `nil`. By using the ancestor and filter facilities, we can view how this erroneous value flows through the program: see Figure 6. 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 space restrictions force us to present a triv-

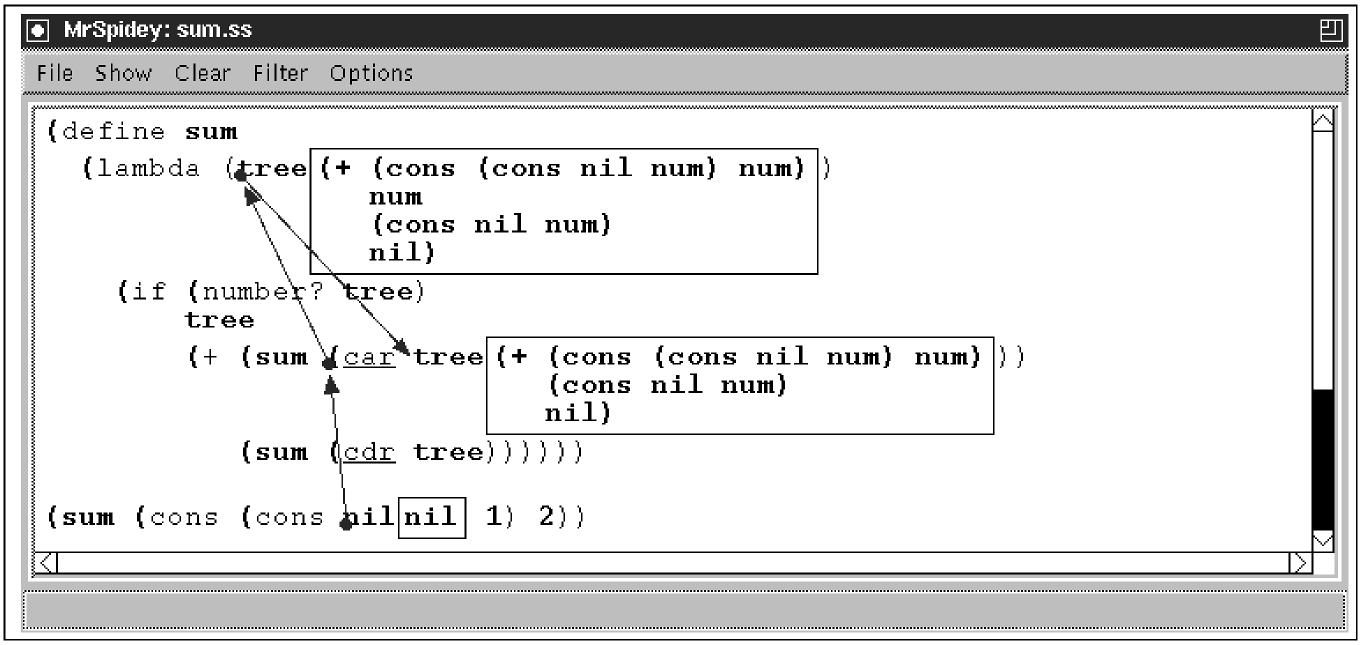

Figure 6: Flow of `nil`

ial program, our initial experiences indicate that the same static debugging process scales well to large programs.

4 Experimental Results

We evaluated the effectiveness of MrSpidey's explanatory mode with an experiment. We constructed a text-based system, called MrSpidey/textual, that resembles an ML-style type-checker and Wright and Cartwright's soft typer [28]. MrSpidey/textual produces an annotated version of the source program, which identifies those program operations that require run-time checks. Using additional commands, a programmer may inquire about the value sets of global and local definitions, and the mismatch between the expected arguments and inferred value sets of program operations.

The participating group consisted of 15 students taking an undergraduate course on programming languages. The students had been introduced to the basics of types and type-safety. All students had a working knowledge of Scheme; none had previously used either static debugger. They were given a 25 minute lecture on set-based analysis and a 45 minute tutorial on the use of the two systems.

The participants were split into two groups, with people of comparable skill separated based on homework assignments. For each test program, the participants in one group used MrSpidey/graphical, while those in the other group used MrSpidey/textual, with the groups alternating analysis tools between programs. The participants were not allowed to execute the programs.

The participants had thirty minutes to work on each of six programs. The programs were excerpts, ranging from one to four pages in length, of larger projects. Each excerpt contained at least one bug and possibly additional run-time checks inserted due to the approximate nature of the analysis. The participants were asked to classify the cause of each run-time check as either:

- a program error, or
- a weakness in the proof system.

In the first case, the participants were asked to fix the program; in the second case, they were asked to explain why this check would never raise an error at run-time.

We observed the progress of the participants and their interaction with the tools. The participants typically used most, if not all, of the facilities of MrSpidey. With the graphical version, students used both the ancestor and filter facilities to display portions of the flowgraph explaining the derivation of certain value set invariants. They mentioned that the ability to track down the origin of values, and especially to focus attention on selected classes of values, was particularly useful in understanding and eliminating checks and errors.

Our observations also suggest that the graphical user interface provides much easier access to the results of the analysis than the textual interface. The users of MrSpidey/textual typically had to work with three different sources of information: the source program, the annotated code, and the console window. The users of MrSpidey/graphical could avoid this context switching since all of these information sources were combined into a single window. Indeed, one user of MrSpidey/textual tried to reconstruct exactly the information provided by MrSpidey/graphical. The student began annotating the printed copy of a test program with value set descriptions and with arrows describing portions of the value flow graph.

Our productivity measurements were inconclusive due to what we believe was an overly artificial experimental setup. We intend to continue our observations through the rest of the semester and to report on them in an expanded version of the paper.

5 The Implementation of MrSpidey

MrSpidey is a component of DrScheme, a comprehensive Scheme development environment. DrScheme consists of several components. Its core component is MrEd; the syntax interface is Zodiac. The following subsections describe these components in some details and provide pointers to relevant technical reports.

5.1 Macro Expansion

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 [18] 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.

5.2 Converting an Abstract Store to Set-Description Expressions

MrSpidey computes a set-description expression for each term M^l from the abstract store representation as follows: First, it views the set environment as a regular-tree grammar with root non-terminal l, and uses a standard algorithm to simplify this grammar [9]. Second, it eliminates unnecessary non-terminals from this grammar, by replacing references to these non-terminals with the right-hand side of the appropriate production rules. If the resulting grammar contains only the single non-terminal l, it expresses the grammar as a non-recursive set-description expression. Otherwise, it expresses the grammar as a recursive set-description expression, where the remaining non-terminals (other than l) become set-description variables.

5.3 Identifying Potential Run-Time Errors

Scheme contains a large number of primitive procedures. MrSpidey represents the set of appropriate arguments for each primitive procedure as a regular-tree grammar (or RTG). Since the value set of each argument expression is also represented as an RTG, deciding whether a primitive is used correctly reduces to the inclusion question between two RTGs. Although the general question is PSPACE-complete [9], for our specific application it can be decided in time linear in the size of the argument's RTG, because the RTG of the expected arguments is deterministic and small.

5.4 Graphical Engine

MrSpidey's graphical component is implemented using MrEd [8], a Scheme-based engine for constructing graphical user interfaces. The core of the engine is am 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 value set information and the arrows describing the value flow graph. Value set 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 standard built-in functionality, but it was easy 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 intergrated 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 MrSpideyś implementors to focus on the interesting parts of their application.

6 Related Work

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

Syntox [2] 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 [2:Section 5].

Several environments [16, 4, 13, 27, 24] have been built for parallel programming languages to expose dependencies, thus allowing the programmer to tune programs to minimize these dependencies. In particular, MrSpidey has many similarities to the ParaScope [16, 4] and D editors [13]. 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.

7 Summary and Future Work

MrSpidey is an interactive static debugging tool that supports the production of reliable software. It identifies the program operations that may signal errors during an execution and describes the sets of erroneous argument values that may cause those errors. Unlike previous systems, it also provides an explanation of how those erroneous values flow through the program. Its graphical user interface presents this information to the programmer in a natural and intuitive manner. Experimental results support our belief that these this information facilitates static program debugging.

MrSpidey also functions as an interactive optimization tool. Using MrSpidey, the programmer can tune a program so that its value set invariants accurately characterize its execution behavior, thus enabling numerous program optimizations that depend on these invariants, including variant check elimination [6, 15, 28, 1, 12], synchronization optimization [7], partial evaluation [19], closure analysis [23], dead-code elimination and constant-folding. To investigate this potential, we implemented variant check elimination as part of MrSpidey. Preliminary results indicate that the resulting tool expedites the production of efficient programs. We intend to investigate this area in more depth.

We adapted set-based analysis for use as the underlying proof technology used in MrSpidey. Set-based analysis can be extended to produce accurate information on numeric ranges [10]. This information is useful for eliminating array bounds checks and for array data dependence analysis. Other program analyses that produce information similar to set-based analysis but which provide alternative cost/accuracy tradeoffs could also be adapted for use in MrSpidey [14, 15, 12, 1].

Availability DrScheme, including MrSpidey, is available at `http://www.cs.rice.edu/ scheme/packages/drscheme`.

Acknowledgments We thank Corky Cartwright and Bruce Duba for discussions concerning the philosophy of soft typing and Nevin Heintze for hints on the implementation of set-based analysis. We also gratefully acknowledge the students in the 1996 COMP311 programming languages course at Rice University for their participation in the experiment.

References

[1] 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.

[2] 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.

[3] Clinger, W., and Rees, J. (Eds.). The revised^4 report on the algorithmic language scheme. *ACM Lisp Pointers 4*, 3 (July 1991).

[4] 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.

[5] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analyses of programs by construction or approximation of fixpoints. In *Proceedings of the ACM Sigplan Conference on Principles of Programming Languages* (1977), pp. 238–252.

[6] Flanagan, C., and Felleisen, M. Set-based analysis for full Scheme and its use in soft-typing. Rice University Computer Science TR95-253.

[7] Flanagan, C., and Felleisen, M. The semantics of future and its use in program optimizations. In *Proceedings of the ACM Sigplan Conference on Principles of Programming Languages* (1995), pp. 209–220.

[8] Flatt, M. MrEd: An engine for portable graphical user interfaces. Rice University Computer Science TR-96-258, Rice University.

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

[10] Heintze, N. Set based analysis of arithmetic. Tech. Rep. CMU-CS-93-221, Carnegie Mellon University, December 1993.

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

[12] Henglein, F. Dynamic typing: syntax and proof theory. *Science of Computer Programming 22* (1994), pp. 197–230.

[13] Hiranandani, S., Kennedy, K., Tseng, C.-W., and Warren, S. The D editor: A new interactive parallel programming tool. In *Proceedings of Supercomputing* (1994).

[14] Jagannathan, S., and Weeks, S. A unified treatment of flow analysis in higher-order languages. In *22nd ACM Symposium on Principles of Programming Languages* (1995), pp. 393–407.

[15] 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. Preliminary version appears as part of Technical Report DAIMI-PB 493, Åarhus University, May 1995.

[16] 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).

[17] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, 1988.

[18] Krishnamurthi, S. Zodiac: A programming environment builder. Rice University Computer Science TR-96-259, Rice University.

[19] Malmkjær, K., Heintze, N., and Danvy, O. ML partial evaluation using set-based analysis. Tech. Rep. CMU-CS-94-129, Carnegie Mellon University, 1994.

[20] 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.

[21] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[22] Plotkin, G. D. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Comput. Sci. 1* (1975), 125–159.

[23] Shao, Z., and Appel, A. Space-efficient closure representations. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (1994), pp. 150–161.

[24] Shei, B., and Gannon, D. Sigmacs: A programmable programming environment. In *Advances in Languages and Compilers for Parallel Computing.* The MIT Press, August 1990.

[25] Shivers, O. *Control-flow Analysis of Higher-Order Languages, or Taming Lambda.* PhD thesis, Carnegie-Mellon University, 1991.

[26] Stefanescu, D., and Zhou, Y. An equational framework for the flow analysis of higher order functional programs. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994), pp. 318–327.

[27] Wolfe, M. J. The Tiny loop restructuring research tool. In *Proceedins of the 1991 International Conference on Parallel Processing* (August 1991).

[28] 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.