A New Way of Debugging Lisp Programs

Cormac Flanagan
Compaq Systems Research Center
130 Lytton Acenue
Palo Alto, CA 94301
`flanagan@pa.dec.com`

Matthias Felleisen
Department of Computer Science
Rice University
Houston, Texas 77005-1892
`matthias@cs.rice.edu`

Abstract

A major cause of software unreliability is the misapplication of primitive operations, such as taking the `car` of `nil`, dividing by zero, or using an invalid array index. These errors are traditionally discovered by extensive testing and debugging, but this approach is unsatisfactory because it is time-consuming and may not identify all potential errors. We suggest to address this problem with sophisticated static debugging systems.

Recent advances in proof technology have brought such advanced static debugging systems within reach. Methods like control-flow analysis or set-based analysis establish invariants that can identify potentially faulty program operations. Past research, however, only focused on the synthesis of the invariants and completely neglected their presentation to the programmer. We believe that the programmer must be able to inspect the inferred invariants *and* to browse their underlying proof. Then, if some set invariant contains surprising elements, the programmer can determine whether the result is a weakness in the proof system or whether it uncovers a flaw in the program.

This paper presents MrSpidey, a user-friendly, interactive static debugger that is completely integrated into DrScheme, our program development environment. MrSpidey exposes those operations that may signal errors during an execution; it describes the sets of values the program expressions may assume; and it also provides a graphical explanation of how values flow through the program. Using MrSpidey, the programmer can easily identify and eliminate the causes of potential run-time errors. Experimental results support our belief that MrSpidey expedites the process of program debugging.

Keywords: Static debugging, programming environments, interactive methods.

1 Introduction

A reliable program does not mis-apply primitive 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 [18] 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 Lisp, Scheme, ML, and Java check primitive program operations. The latter two use a mixture of compile-time type checking and insertion of run-time checks. The first two give the programmer even more power and exclusively rely on run-time checks.[1] 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 primitive 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 [21, 25, 15] or set-based analysis [12, 11] 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 contains only closures (of the appropriate arity), 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. 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 the necessarily conservative nature of the proof system.

We have developed a static debugger, called MrSpidey, for the Scheme dialect of Lisp. This static debugger allows the programmer to browse program invariants and their derivations. Since MrSpidey is to be used as part of the typical program development cycle, we have integrated it with DrScheme, our Scheme program development environment. On demand, MrSpidey statically analyzes the program and uses the resulting invariants to iden-

[1] We ignore Common Lisp's type annotations here.

tify and highlight program operations that are not provably safe (according to MrSpidey's underlying proof system). Associated hyper-links provide access to additional information, including:

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

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

A complete description of the underlying proof technology used in MrSpidey can be found in related papers [6, 9]. In this paper, we report on our experience using MrSpidey. Section 2 describes the user interface that presents the inferred invariants to the programmer, and section 3 describes our experience using MrSpidey on a variety of programs, including both mature systems and a system under development. Section 4 describes related work. Section 5 presents our conclusions and future research directions.

2 Using MrSpidey

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

On demand, MrSpidey statically analyzes the current program and infers useful static debugging information about that program. Specifically, MrSpidey identifies unsafe primitive operations, derives an appropriate value set invariant for each program expression, and provides a graphical explanation of each derived invariant.

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

2.1 Displaying Unsafe Operations

Unsafe program operations that may signal run-time errors are natural starting points in the static debugging process. MrSpidey highlights these unsafe operations via font and color changes. Any primitive operation that may be applied to inappropriate arguments is highlighted in red (or underlined on monochrome screens). Conversely, primitive operations that never raise errors are shown in green. Any function definition that may be applied to an incorrect number of arguments is highlighted by displaying the `lambda` keyword in red (or underlined). Figure 1 contains examples of such unsafe operations.

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

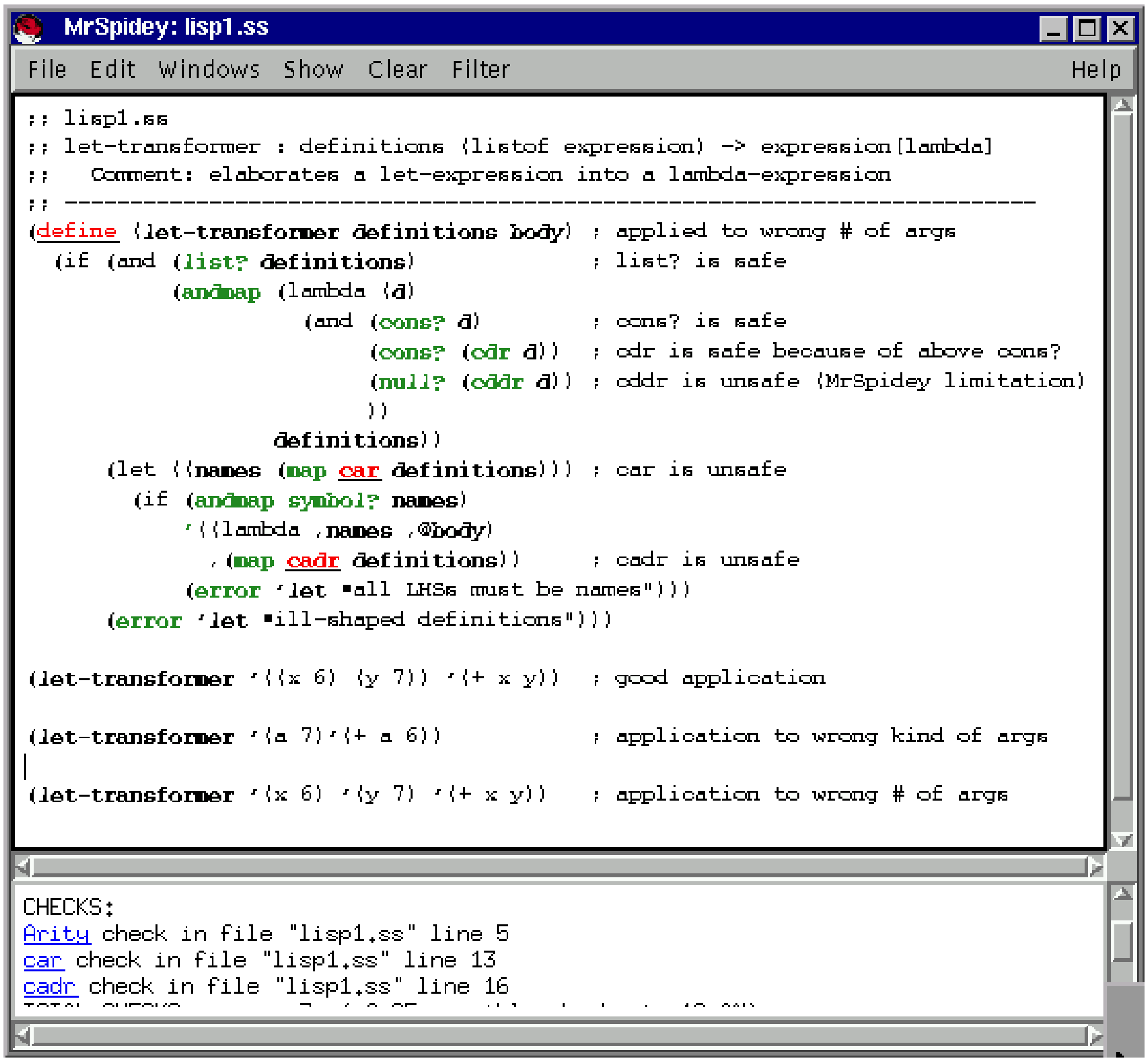

Figure 1: Identifying unsafe operations

MrSpidey also infers additional information for each expression in the analyzed program. This information cannot be immediately displayed, since it would simply result in "information overload." Instead, MrSpidey provides this information on a demand-driven basis via a pop-up menu associated with each expression. The information available through the menu is described in the following sections.

2.2 Presenting Value Set Information

MrSpidey infers an inferred value set invariant for each expression in the program. This value set invariant is expressed using the following language of *set-description expressions* (*SDE*):

$$
\begin{array}{lllll}
\tau & \in & \mathit{SDE} & ::= & \texttt{nil} \mid \texttt{num} \mid \texttt{sym} & \text{(constants)} \\
& & & & \mid (\texttt{cons}\ \tau_1\ \tau_2) & \text{(pairs)} \\
& & & & \mid (\tau_1\ \ldots\ \tau_n\ \texttt{->}\ \tau) & \text{(functions)} \\
& & & & \mid \texttt{empty} & \text{(empty set)} \\
& & & & \mid (\texttt{union}\ \tau_1\ \ldots\ \tau_n) & \text{(unions)} \\
& & & & \mid (\texttt{rec}\ ([\alpha_1\ \tau_1]\ \ldots\ [\alpha_n\ \tau_n])\ \tau) & \text{(recursive invariants)} \\
& & & & \mid \alpha & \text{(set description variables)}
\end{array}
$$

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

On demand, MrSpidey computes a value set invariant for each program expression. and displays that invariant in a box inserted to the right of the expression in the buffer. Figure 2 shows the inferred invariant for the variable *definitions*.

2.3 The Value Flow Browser

A programmer who encounters a value set invariant with bad values needs to understand the derivation of that invariant. MrSpidey provides such an explanation in the form of arrows overlaid on the program text. These arrows describe the data-flow paths of the program. Because a large numbers of arrows would clutter the program text, these arrows are presented in a demand-driven fashion.

The Ancestors option in the pop-up menu for an expression allows the programmer to view all portions of the program's data-flow that influence a particular invariant, thus providing the programmer with a complete explanation of the derivation of that invariant. For example, figure 3 shows the sources for the variable *definitions*.

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

In some cases, the number of arrows presented by the ancestor facility is excessive. Since the programmer is typically interested only in a particular class of values, MrSpidey

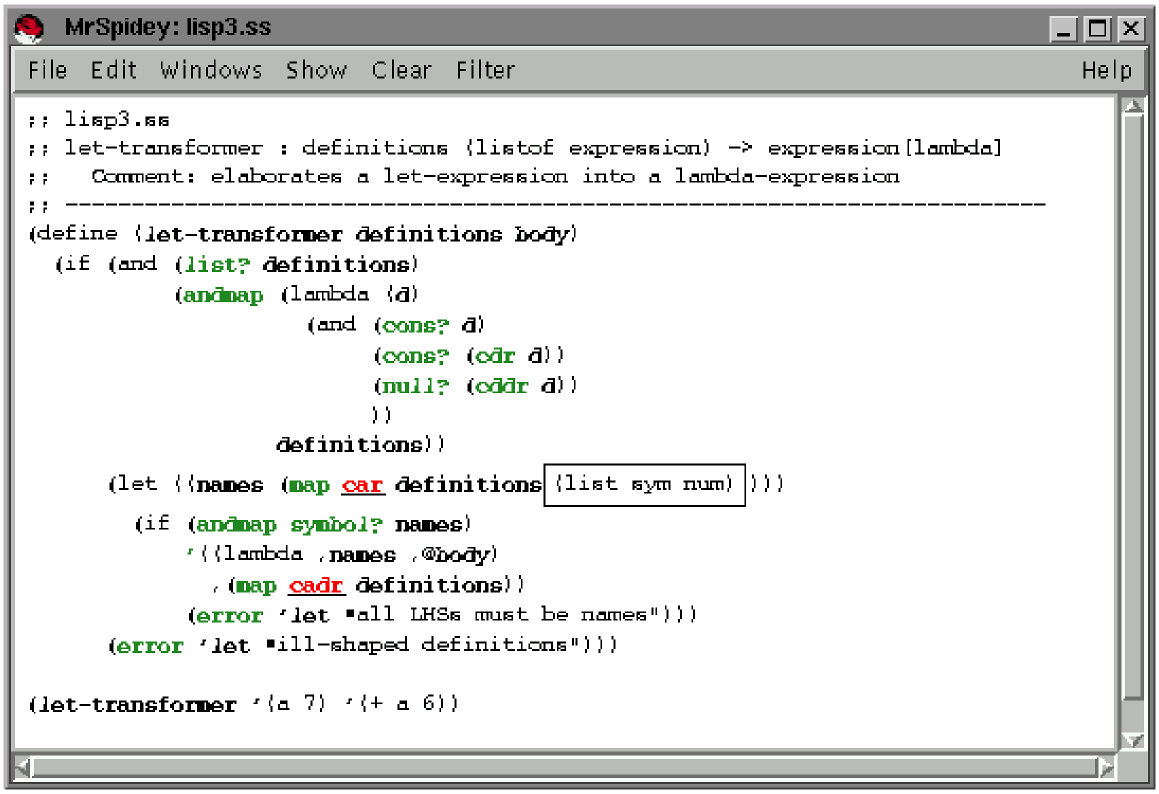

Figure 2: The value set invariant of *definitions*

incorporates a *filter* facility that allows the programmer to restrict the displayed arrows to those that affect the flow of certain kinds of values. This facility is extremely useful for quickly understanding why a primitive operation may be applied to inappropriate argument values.

By using an appropriate combination of the ancestor and filter facilities, the programmer can quickly view the flow of a particular class of value through the program. For example, figure 4 shows the derivation of the `sym` component in the value set invariant for *definitions*.

2.4 A Sample Debugging Session

Let us illustrate the usefulness of MrSpidey's explanatory capabilities with our running example. For the code in `lisp4.ss`, MrSpidey highlights the primitives `map` and `car` in

(`map car` *definitions*)

as unsafe (see Figure 3), indicating that these operations may signal run-time errors. In order to understand the problem, a programmer will inspect the value set invariants for the arguments to these primitives. In the case of `map`, an inspection of its second argument, *definitions*, shows that this set includes symbols. Since `map` can process only lists, it would have to signal an error if it were applied to symbols. By using the ancestor and filter

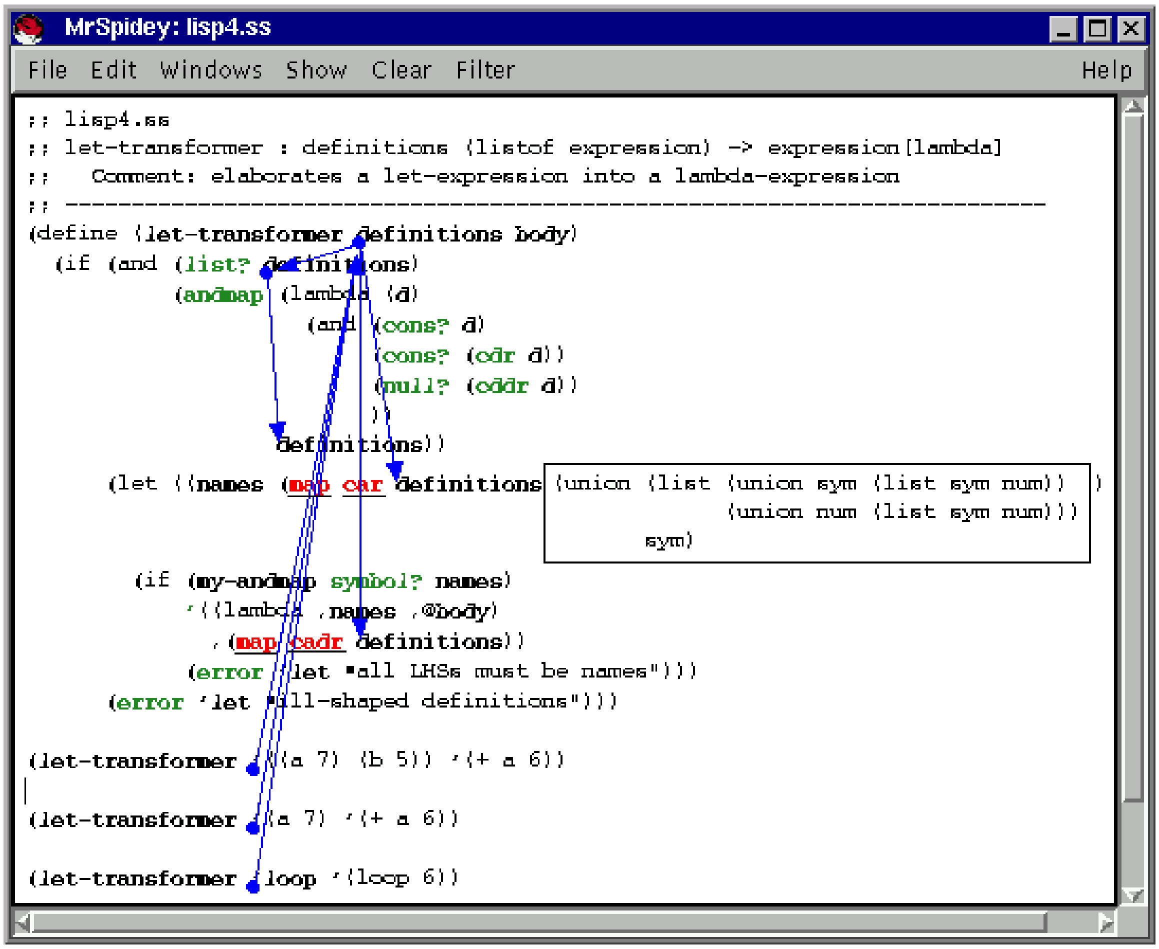

Figure 3: Value source information

facilities, a programmer can quickly determine how the symbol 'loop flows through various applications to the application of map: see figure 4.

In the case of car, a programmer must first understand that it receives its arguments indirectly via map. This means that map may apply car to inappropriate values, say, the empty list or symbols. And indeed, another look at the value set invariant of map's second argument shows that it may contain a list whose head is a symbol. If map traverses such a list, it will pass the symbol to car, which will trigger an error message. A value flow analysis similar to the one in the first case quickly points to the second call site of *let-transformer* as the culprit.

Although statically debugging programs with specific test cases is a good start, a programmer should also formulate a general description of the possible inputs for a function. In the case of *let-transformer*, the function is used by the MzScheme reader, which always applies the transformer to an S-expression and a list of S-expressions. A programmer

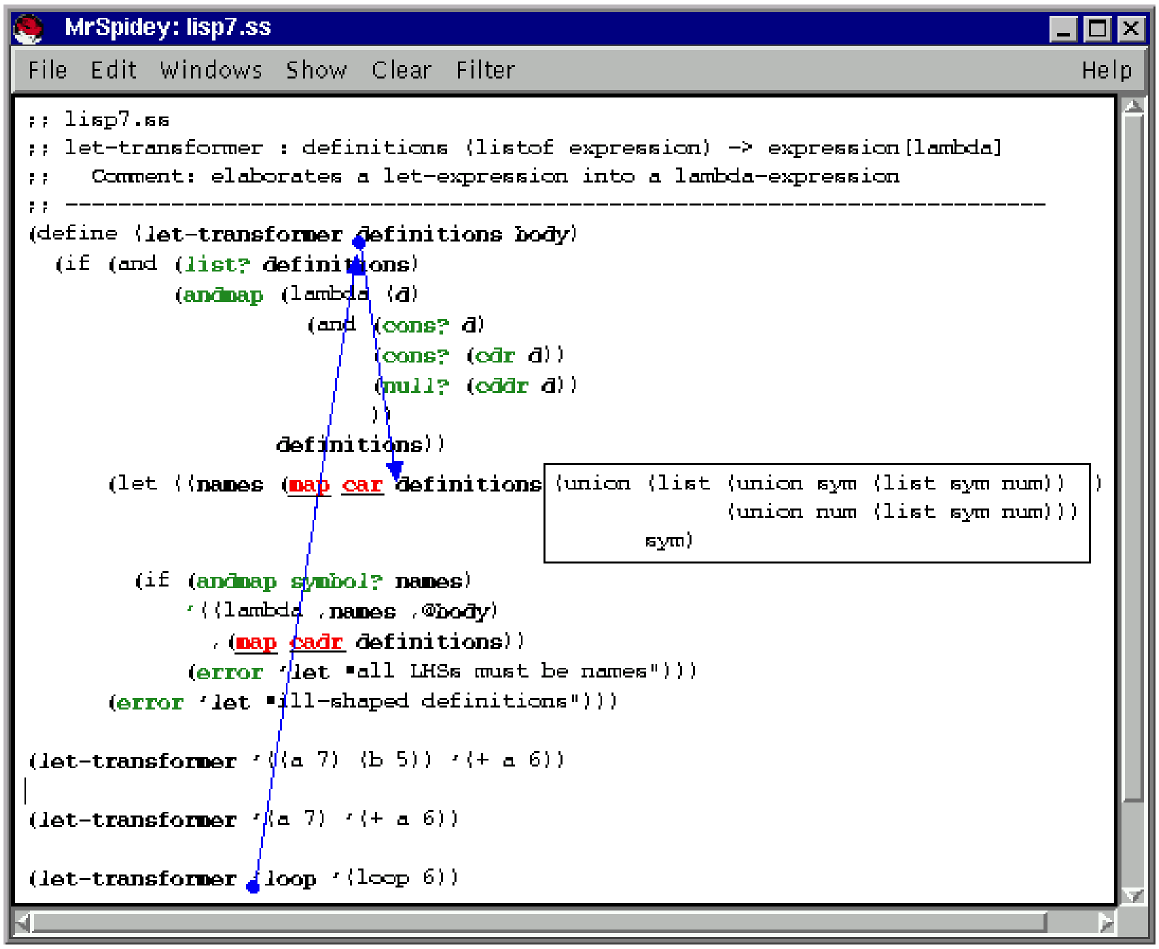

Figure 4: Flow of sym

can express this fact with **type:** annotations as shown in figure 5. Using this annotation, MrSpidey can then check how many primitives are unsafe for the most general case.

As it turns out, MrSpidey marks a number of these primitives as unsafe, including map, car, and cadr inside of the **let**-expression. If MrSpidey had used the **if**-condition in its analysis of the **then**-branch, it would not have highlighted map, car, and cadr. Since the current theorem proofing engine cannot perform this kind of conditional reasoning in an inexpensive manner, the programmer must formulate a stronger set invariant for the potential inputs. Specifically, the programmer can formulate a set description for those inputs of *let-transformer* that pass the **if**-test. MrSpidey will then analyze the **then**-branch using the assumptions expressed in the conditional. As figure 6 shows, using this second static debugging step proves that *let-transformer* is perfectly safe.

The example in this section is simple. Still, it does provide a good example of the explanatory capabilities of MrSpidey. The following section describes our experience using

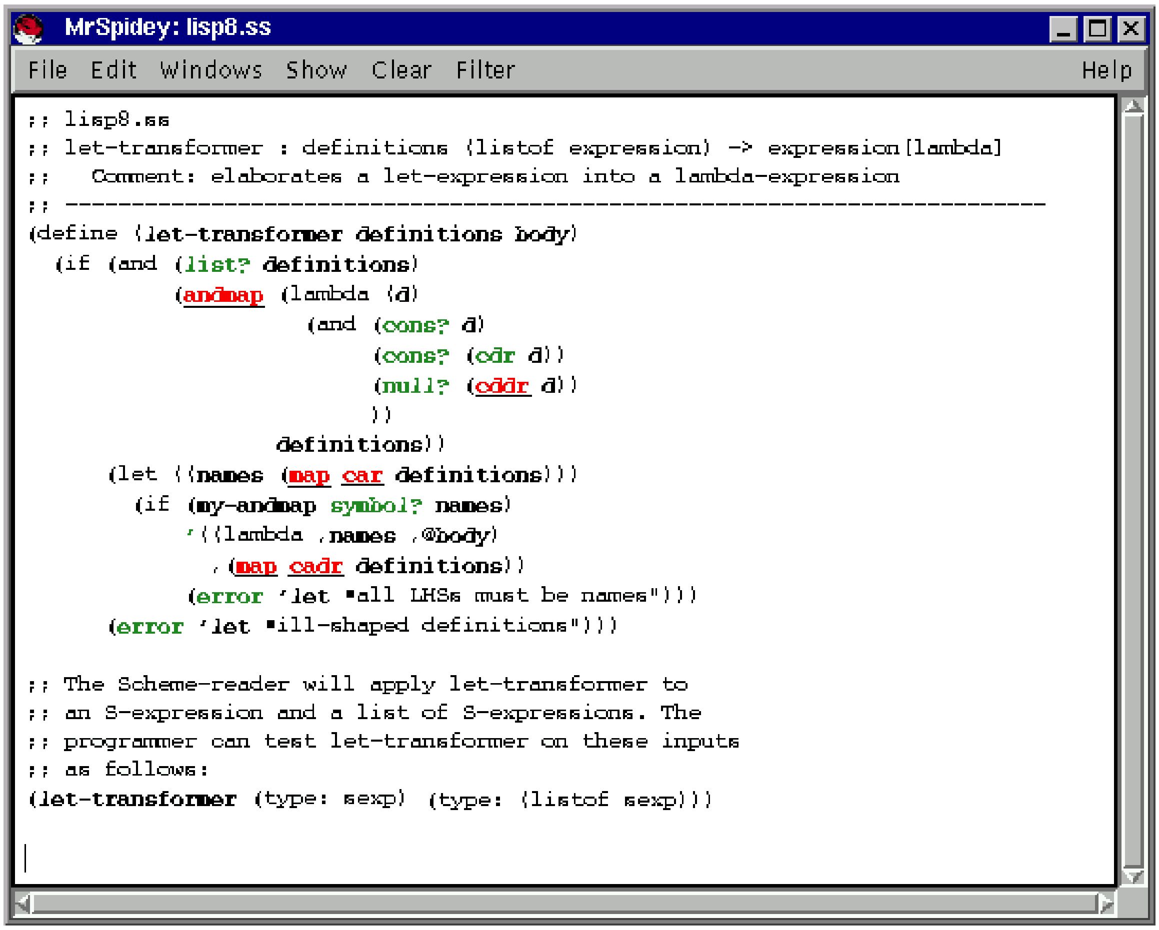

Figure 5: Expressing invariants

MrSpidey on significantly larger programs.

3 Evaluation of MrSpidey

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

3.1 Verifying a Web Server

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

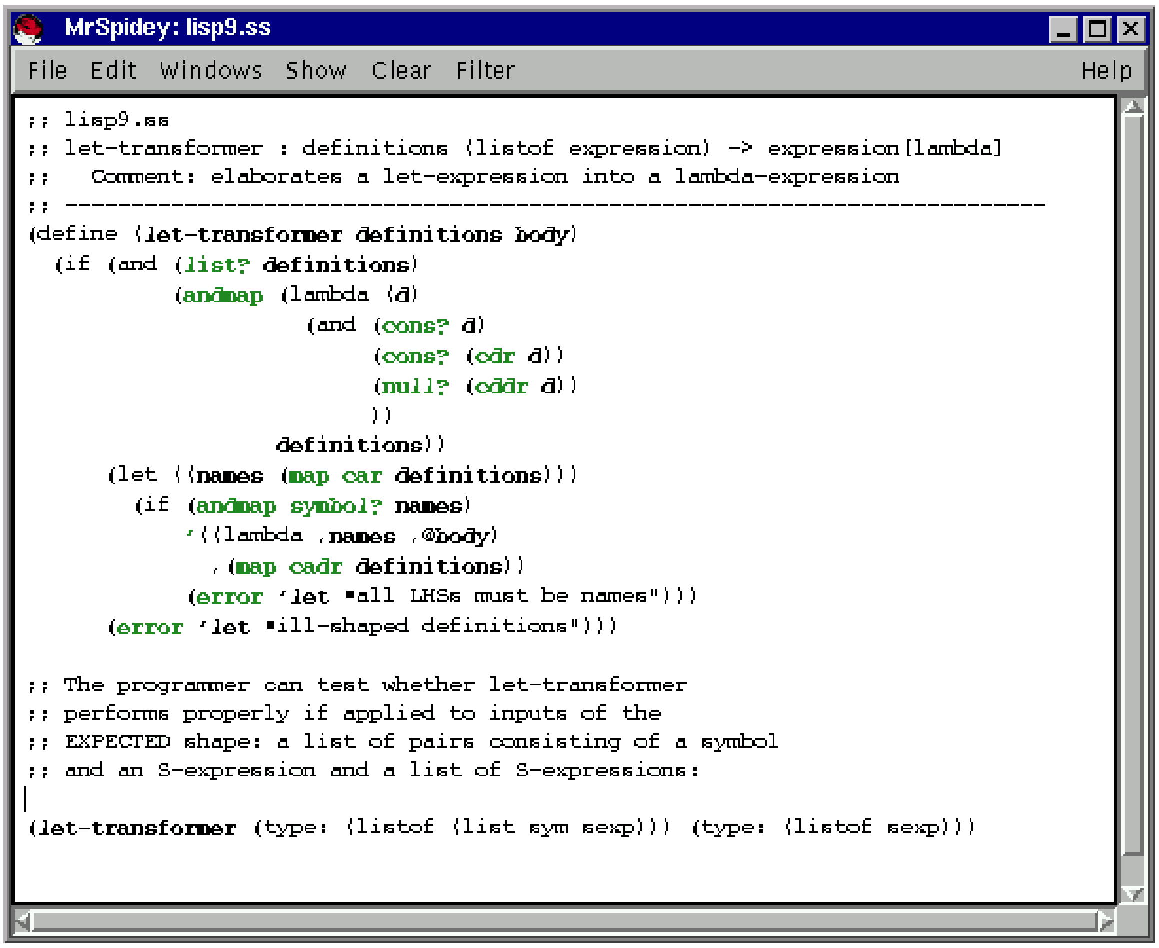

Figure 6: Expressing stronger invariants

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

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

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

3.2 Verifying gunzip

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

We used MrSpidey's ancestor and filter facilities to identify why MrSpidey could not prove the safety of these operations. In most cases, the cause was a code pattern that was too complicated for MrSpidey to understand. Simplifying and cleaning up the code in various places reduced the number of unsafe operations to 7.

All of the remaining unsafe operations are actual errors that resulted from failing to detect and handle a truncated input file. Thus the original program would crash on a truncated input file with a message such as:

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

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

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

3.3 Verifying an Extended Direct Semantics Interpreter

Extended direct semantics is a format for denotational language specifications that accommodates orthogonal extensions of a language without changing the denotations of existing phrases [3]. The semantics of a language is specified in this format using a tower of interpreters. The tower starts with a basic interpreter, which can interpret only certain trivial expressions. This basic interpreter is then composed with additional interpreters for the various constructs in the language.

We used MrSpidey to investigate an interpreter expressed in this style. In addition to the basic interpreter, this program contains interpreters for arithmetic operations (integer constants, `add1` and `sub1`), call-by-value functions (variables, functions and applications), control operations (`catch` and `throw`), and assignments (`ref`, `deref` and `setref`).

We ported the program to DrScheme's module system [10]. During this porting process, we inadvertently introduced an error, where one of the module was linked with an incorrect number of imports. When this program was analyzed, MrSpidey produced the warning:

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

together with a hyper-link to the relevant module.

After this bug was fixed, MrSpidey was successfully able to verify the safety of the entire program. Part of the reason that MrSpidey is so successful on this program is that the program had already been carefully written so that Soft Scheme [27] could verify its safety, and thus the style of the program was already well-suited for automatic analysis techniques.

3.4 Statically Debugging HHL

We also used MrSpidey to statically debug a program under development. This program, called `HHL`, is a hardware verifier using heterogeneous logic. It consists of 3312 lines of Scheme code distributed over 12 files, and interfaces to the Omega calculator [22].

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

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

The remaining eight unsafe operations appear to be caused by limitations in MrSpidey's underlying analysis.

4 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 [17, 4, 14, 26, 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 [17, 4] and D editors [14]. 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.

5 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 [8, 16, 27, 1, 13], 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 [11]. 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 [15, 16, 13, 1].

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

Acknowledgments We thank Corky Cartwright and Bruce Duba for discussions concerning the philosophy of soft typing; Nevin Heintze for hints on the implementation of set-based analysis; and Matthew Flatt, Shriram Krishnamurthi, and Robby Findler for feedback on various aspects of MrSpidey.

References

[1] Aiken, A., Wimmers, E. L., and Lakshman, T. K. Soft typing with conditional types. In *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] Cartwright, R., and Felleisen, M. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software*.

[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 *Principles of Programming Languages* (1977), pp. 238–252.

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

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

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

[9] Flanagan, C., and Felleisen, M. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (June 1997), pp. 235–248.

[10] Flatt, M., and Felleisen, M. Units: Cool modules for HOT languages. In *Programming Language Design and Implementation* (1998).

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

[12] Heintze, N. Set-based analysis of ML programs. In *Lisp and Functional Programming* (1994), pp. 306–317.

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

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

[15] 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.

[16] JAGANNATHAN, S., AND WRIGHT, A. K. Effective flow analysis for avoiding run-time checks. In *Second International Static Analysis Symposium, LNCS 983* (September 1995), Springer-Verlag, pp. 207–224.

[17] 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).

[18] KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language*. Prentice-Hall, 1988.

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

[22] PUGH, W. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing* (1991).

[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] STEFANESCU, D., AND ZHOU, Y. An equational framework for the flow analysis of higher order functional programs. In *Lisp and Functional Programming* (1994), pp. 318–327.

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

[27] WRIGHT, A., AND CARTWRIGHT, R. A practical soft type system for scheme. In *Lisp and Functional Programming* (1994), pp. 250–262.