ACM SIGPLAN
Functional Programming

Editor: *Philip Wadler, Bell Labs, Lucent Technologies*; *wadler@research.bell-labs.com*

The DrScheme Project: An Overview

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi
Department of Computer Science
Rice University
Houston, Texas
`http://www.cs.rice.edu/CS/PLT/`

Abstract

DrScheme provides a graphical user interface for editing and interactively evaluating Scheme programs on all major graphical platforms (Windows 95/nt, MacOs, Unix/X). The environment is especially well-suited to beginning programmers because it supports a tower of Scheme subsets. Each level corresponds to a particular stage in a typical introductory Scheme course and implements a stringent set of syntactic checks. The environment also pinpoints run-time exceptions in a graphical manner and implements a mostly functional read-eval-print loop.

DrScheme's most advanced component is a powerful static debugger. It permits programmers to inspect programs for potential safety violations before running them. If the debugger discovers a potential problem, it explains the problem by drawing a value-flow graph over the program text. The value-flow graphs shows how an inappropriate value may reach a program operation and trigger a run-time check.

The development of DrScheme in Scheme validated the strengths of Scheme, but also revealed several weaknesses. To overcome the latter, the underlying Scheme implementation was extended with a class-based object system, a language of program units, and a sophisticated GUI engine. All of these extensions are available to the programmer, who can thus interactively create fully portable, graphical applications.

1 Origins and Goals

Over the past ten years, Scheme [1] has become the most widely used functional programming language in introductory courses in the United States [11, 12]. When Rice University implemented an introductory course using Scheme, the instructors noticed three significant problems with its popular implementations. First, although Scheme's parenthesized prefix notation is extremely simple, beginning students often encounter surprising syntactic and run-time errors due to the transition from infix algebraic syntax to prefix Scheme syntax. Second, Scheme implementations provide little or no source information about run-time errors, even though such information is far more useful for Scheme than for C++ because the former is safe and the latter isn't. Finally, the traditional Scheme read-eval-print loop obscures the algebraic nature of values and introduces subtle bugs due to its hidden state.

In response to these observations, Rice's programming languages team (PLT) decided to launch the DrScheme project. From the beginning, the project had two goals:

- The first goal is to develop a modern programming environment for Scheme. The environment must support the teaching of programming principles in a pedagogically meaningful way. As part of this goal, we have always aimed for a completely portable graphical environment.
- The second goal is to equip the environment with "smart" (i.e., semantics-based) programming tools that assist advanced programmers with the development of robust software. The first innovation in this direction is *MrSpidey*, a static debugger. This new kind of software tool statically analyzes programs, reports potential fault sites to the programmer, and constructs graphical explanations of its reasoning on demand.

At the same time, the project team realized that the implementation of a large system would be a good chance to evaluate the use of Scheme for non-trivial software projects. In the past, Scheme has been used successfully for the implementation of tools that process languages, *e.g.*, abstract machines, compilers, interpreters, and type checkers. Since these applications are heavily tree-oriented, Scheme is a natural choice. For other contexts, however, especially that of graphical programming environments, Scheme does not seem to offer any particular advantages. Still, Scheme's expressiveness makes it a strong candidate for a thorough evaluation.

Finally, the project naturally yields a significant body of Scheme code that is used on a daily basis. The examples and impetus provided by working with large programs are invaluable in improving the environment and its smart tools. The eventual goal is to produce a self-applicable programming environment and to prove the usefulness of the smart tools in this context.

This column primarily presents DrScheme and MrSpidey and explains how these tools support teaching and programming. Two short sections briefly discuss the subsidiary goals. The reference section provides some pointers for complementary reports on the project and its contributions.

2 DrScheme: The Environment

DrScheme integrates program editing and program evaluation in a seamless manner (see figure 1). To overcome the problems of traditional Scheme implementations, its editor and evaluator support: a hierarchy of four Scheme subsets whose choice is pedagogically motivated, source correlation at all execution steps, and a new kind of read-eval-print loop. In addition, DrScheme also offers two pedagogic tools: a symbolic evaluator and a context-sensitive syntax checker. Finally, DrScheme includes a static debugger, which analyzes programs and exposes potential safety violations prior to execution. The following three subsections provide an overview of DrScheme; for more detailed information, we refer the reader to an extended report [5].

2.1 Pedagogic Enhancements

Language Levels: University courses typically introduce students to Scheme in discrete segments. The first segment covers first-order functional programming, the second one higher-order functions and data structuring, and the third one imperative facilities like **set!**, operations that mutate data, and call/cc. Given this widespread practice, DrScheme permits users to choose one of these levels and then strictly enforces correspondingly restrictive syntactic rules.

The strict enforcement of syntactic rules solves numerous notational problems for beginners who, through school mathematics and high school programming courses, have become used to infix operators and operator precedence. For example, the author of the program

```
(define (length l)
  (cond
    [(null? l) 0]
    [else 1 + (length (rest l))]))
```

has lapsed into algebraic syntax in the second clause of the **cond**-expression. Since in standard Scheme the value of a **cond**-clause is the value of its last expression, this version of **length** always returns **0**, regardless of its input. Other lapses into algebraic syntax may yield similarly inexplicable results or, even worse, error messages from the run-time system that make no sense for a beginning student.

In DrScheme, beginners are protected from such mishaps. By choosing the language level "Beginner", a programmer installs a stringent set of syntactic checks, which recognizes lapses into algebraic notation as easily explicable syntax mistakes. A beginner can then fix these mistakes before they cause additional run-time problems. The other language levels ("Intermediate", "Advanced", and "QuasiR4RS") address similar, but less severe problems with Scheme's syntax.

Run-Time Errors: Many modern Pascal and C++ environments highlight a source location when a program causes a core dump. Unfortunately, this source correlation is in general completely useless because the corresponding segmentation fault or bus error is not a direct consequence of the abuse of a computational primitive. The primitive has been misapplied much earlier, but since low-level languages do not enforce abstraction invariants between the computer architecture and the programming language, nonsensical bit patterns may flow through the program arbitrarily long before an error is signaled, if at all.

Scheme and other functional languages are safe and through a mixture of syntactic and run-time checks enforce invariants and, in turn, the intended level of data abstraction. More technically, each primitive operation

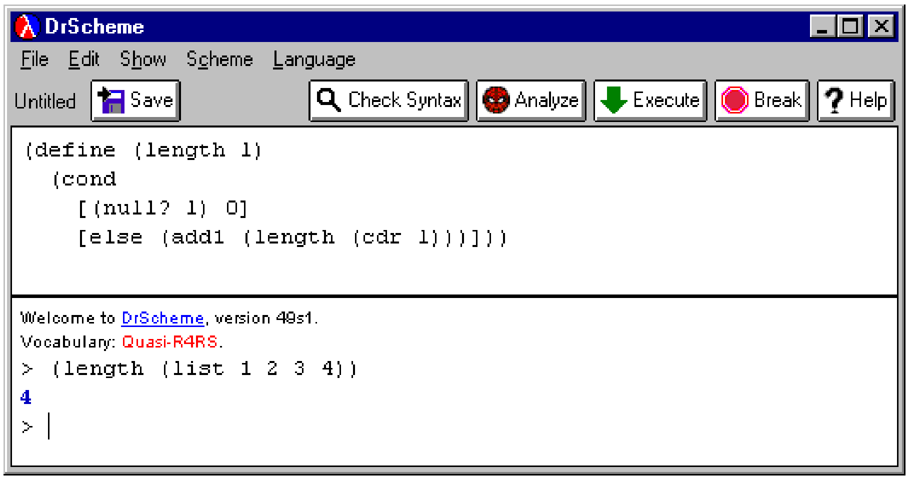

Figure 1: The DrScheme Window (Windows 95/NT version)

(for which it is not possible to enforce its invariants statically) checks at run-time whether or not its arguments and results are in the proper range. Examples of operations that check their arguments at run-time are arithmetic operations, array indexing, and "pointer" dereferencing. When an operation detects a problem with its arguments or results, it aborts the program execution. Unfortunately, conventional Scheme environments do not connect such run-time errors with the corresponding source location and thus force the programmer to search through the program for the error.

In contrast, in DrScheme a failed safety check does not only signal the nature of an error but also highlights its location in the program. To implement the second part, the underlying Scheme parser keeps track of source locations even across general macro expansions [3, 10]. The evaluator uses this source information, if desired, by setting a special "source register" ahead of primitive computational steps. This source register contains enough information to highlight the primitive application if it fails. The strategy ensures source correlation and preserves the desired tail-call optimization of Scheme [1].

A Transparent REPL**:** One distinct advantage of Scheme over conventional languages is its read-eval-print loop (REPL). Using the REPL, students can easily experiment with individual expressions, procedures, or compilation units. They can also change a program during execution to fix a bug on the fly or to observe and measure certain quantities. While the REPL is an excellent tool for gentle introductions to computing, it often causes subtle bugs in the various stages of an introductory course.

The traditional REPL interferes with teaching in two ways. First, Scheme's REPL inherited the LISP printer, which displays results in a list-oriented notation that resembles but is not identical to the input notation. Although this form of printing is useful for experienced programmers, especially in the context of program-writing programs, it is unintuitive for beginners who learn to compute the value of a Scheme program using ordinary algebra. Second, a REPL uses a modifiable table, the *namespace*, to keep track of definitions. Consequently, the REPL is a state-oriented element in a world that otherwise has the appearances of an implementation of algebra. If the REPL is used in a careless manner, it can introduce or shadow program bugs in a way that is utterly confusing to beginning students.

DrScheme overcomes both problems with a new REPL, which differs from Scheme's traditional read-eval-print loop in that it is parameterized over the printer and the namespace of the evaluator. The printer parameterization permits matching the language level and the printer. Thus, for a beginner, DrScheme prints the algebraic form of a value. After the introduction of data mutation, it exposes the sharing among nodes in a value. For flexibility, the user can change the printer that is provided by default. The choices also include the traditional Scheme printer, and a printer that supports both program-writing programs and an algebraic understanding of program execution (via **quasiquote**).

The namespace parameterization enables the environment to start each program execution with a clean slate. That is, every time the programmer clicks on the "Execute" button (see figure 1), the REPL loads the current set of definitions into a new namespace, which eliminates all legacy definitions from the preceding series of interactions. The environment thus efficiently mimics an inefficient and error-prone technique that is used by experienced Scheme programmers to avoid legacy problems.

2.2 Pedagogic Tools

The Symbolic Stepper: Scheme courses invariably introduce Scheme's basic functional core via a reduction semantics. The semantics extends three groups of algebraic laws that students are (or should be) intimately familiar with: the laws of primitive operations (like addition); the law of function application (β_v-reduction); and the law of replacement of equals by equals. This reduction semantics scales up to full Scheme [4]

DrScheme includes a tool that enables students to reduce a program to a value, step by step. It can deal with all the features used in Rice University's course, including the entire functional sub-language, structure definitions, variable assignment, data structure mutation, exceptions, and other control mechanisms. A student invokes the stepper on the current program by choosing Tools|Stepper.[1] By default, the stepper shows every reduction step of a program evaluation. While this default is useful for a complete novice, a full reduction sequence contains too much information for programmers with some experience. Hence the stepper permits the student to choose which reduction steps are shown or which sub-expressions the stepper is to focus on. The student can change these controls to view a more detailed reduction sequence at each stop.

Students use the stepper for two purposes. First, they use it to understand the meaning of new language features as they are introduced in the course. A few sessions with the stepper illustrates the behavior of new language constructs better than any blackboard explanation. Second, students use the stepper to find bugs in small programs. The stepper stops when it encounters a run-time error and permits students to move backwards through the reduction sequence. This usage quickly explains the reasons for bugs and even suggests fixes.

Syntactic-Lexical Annotations: Beginning programmers need help understanding the syntactic and lexical structure of their programs. DrScheme provides a syntax checker that annotates the source text of syntactically correct programs based on the syntactic and lexical structure of the program. The syntax checker marks up the source text based on five syntactic categories: primitives, keywords, bound variables, free variables, and constants. On demand, the syntax checker also displays arrows that point from bound identifiers to their binding occurrence, and from binding identifiers to all of the their bound occurrences. Since the checker processes the lexical structure of the program, a program can use it to α-rename bound and defined identifiers.

2.3 MrSpidey: Towards DrScheme2

For the second generation of DrScheme, we intend to develop "smart tools" whose purpose it is to help the programmer validate weak invariants of the program. A first extension in this direction is MrSpidey, a static debugger and soft typer[7]. It subsumes the syntax checker, but is computationally far more expensive.

MrSpidey inspects a program for *potential* safety violations. That is, it attempts to prove that the arguments of all program operations belong to appropriate sets. If MrSpidey cannot establish this legality condition for some operation, it annotates the operation in red. Since the underlying proof system is complex but necessarily conservative, the static debugger can explain its annotations. More specifically, it can insert the inferred value set for any expressions into the program display and it can draw the value flow graph on top of the program. Using these explanations, a programmer can then decide whether the annotated operation may indeed fail or whether the underlying proof system is too weak to prove the correctness of the invariant. Depending on the answer, the programmer will fix the bug or rewrite the program to improve the analysis.

For an illustration, consider the buggy program in the top part of figure 2. The program is simplistic and extremely small but suffices to demonstrate MrSpidey's basic capabilities. After the static debugger completes its analysis, it displays the program with an annotated `add1` operation (underlined in the figure). This annotation indicates that the static debugger cannot prove that the argument will always be a number. The programmer can now ask for the value set of `add1`'s argument, to which the static debugger responds by inserting the box to the right of `add1`'s argument: see top of figure 2. The box describes the value set for the argument: `(union num null)`. Since this inferred set contains `null`, the static

[1] The stepper is not available in DrScheme Version 50.

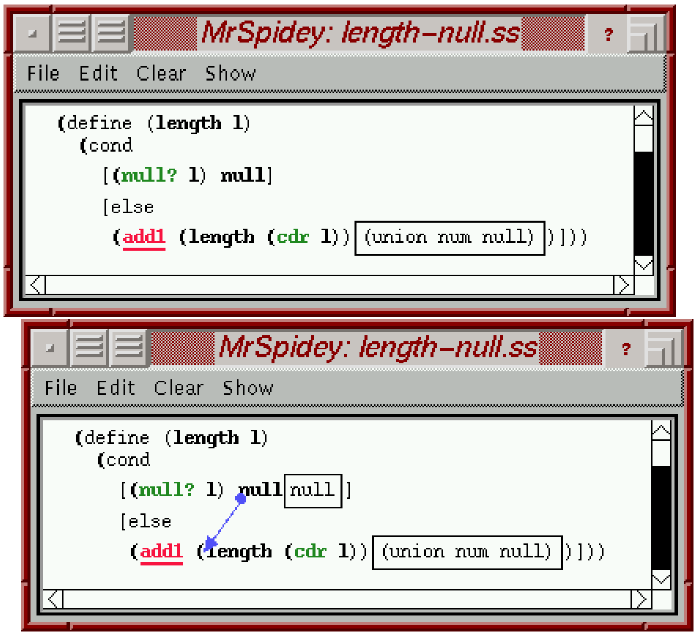

Figure 2: MrSpidey: The static debugger (X version)

debugger had to annotate add1 and warn of a potential mis-application. To see how null can flow into the argument of add1, the static debugger can also overlay the program with a slice of the value flow graph. The slice shows how the offending value might flow into the argument position for add1. In this example, the graph is a single arrow from null to *length* because a recursive call may return this value: see bottom of figure 2.

When MrSpidey is used on realistic multi-module programs, it analyzes the entire program at a coarse level except for the module that the programmer is working on. For the latter, MrSpidey performs an ordinary analysis and displays the required information. In addition, for all intra-module value flows, MrSpidey performs as before. If, however, some flow of values crosses the module boundary, it is indicated with an arrow that crosses the window boundary: see top of figure 3. By clicking on this arrow, the programmer can demand a fine analysis of the "target module" and MrSpidey will respond by displaying the annotated version. In particular, the display contains an arrow that crosses the window boundary and shows how values flow across boundaries: see bottom of figure 3. For more information on the multi-module analysis, the reader may wish to consult Flanagan's dissertation [7].

3 A Scheme for Large Systems

The development of DrScheme in Scheme has produced valuable insight into Scheme's capabilities for building large systems. Not surprisingly, Scheme's core language has served its role well as a tool in which programmers can quickly explore ideas and create prototypes. The most important features proved to be the dynamic type (unitype) system, higher-order procedures, and threads (continuations). Nevertheless, the project also demonstrated that for the purpose of GUI-oriented projects, Scheme should be extended with a number of features, in particular an exception system, an object system for interfacing with GUI libraries, facilities for encapsulating and linking program units, and a foreign function interface.

We have overcome these deficiencies with a new Scheme implementation, MzScheme, and an accompanying GUI engine, MrEd. As we gathered experience developing DrScheme, we refined MzScheme and MrEd. MzScheme's object system now supports composable classes, which greatly simplify the implementation of DrScheme's complex graphical interface. Its program unit system permits the treatment of program units as first-class values, mutually recursive references among procedures across unit boundaries, and the dynamic loading/linking of units. Finally, MrEd integrates GUI classes

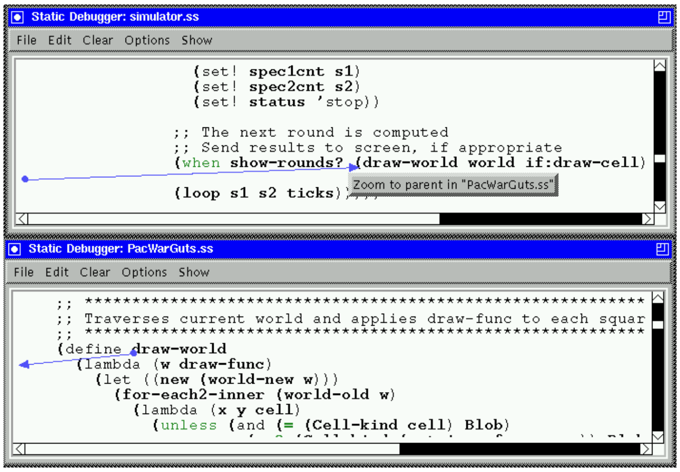

Figure 3: Analyzing Multiple Modules

with MzScheme's object and thread systems, providing multiple *event spaces* so that DrScheme's GUI can securely co-exist with a GUI program created by the user. We report on our extensions to Scheme in more detail elsewhere [8, 9].

4 DrScheme on DrScheme

A significant milestone towards our primary goal is to develop a programming environment that we can use to produce DrScheme itself. To prove the feasibility, we have developed MrSpidey to the point where it can be applied to large portions of DrScheme's Scheme code (around 70Kloc). The experiment indicates the potential benefits and the problems we are facing. While a smart tool like MrSpidey can reveal bugs that hundreds of users have not been able to find, an appropriate programming environment requires much more computational resources than typical workstations provide. For more information on this experiment (and others), we refer the reader to Flanagan's dissertation [7]. We expect that further experiments of this kind will yield additional insight into the requirements for smart tools and the software engineering process.

5 Conclusion

The construction of DrScheme overcomes the pedagogic problems of Scheme with a strong integration of the editing and evaluation processes. Our experience with DrScheme at Rice is positive. DrScheme has been used in our introductory course and has significantly strengthened our course. Starting this fall DrScheme is also used in local secondary schools for introductory programming. The students are excited about the graphical, interactive mode of experimentation. Since we have made DrScheme publicly available, over 100 non-Rice users/sites have signed up for the DrScheme announcement list.

Many aspects of DrScheme apply to languages other than Scheme. Any language becomes more accessible to the beginner in an environment that provides a tower of well-chosen, strictly separated language levels, a mostly functional read-eval-print loop, accurate source highlighting for safety violations, and a stepping tool that reinforces the algebraic view of computation. In addition, typed languages can benefit from graphical explanations of type errors like those of MrSpidey.

In conclusion, we believe that the continued development of DrScheme will provide new inspiration into the development of Scheme, related functional and object-oriented languages, and programming environments.

Acknowledgments: The development of DrScheme benefited from many contributions. Cormac Flanagan created MrSpidey, the static debugger. Gann Bierner, Richard Cobbe, Daniel Grossman, Mark Krentel and Stephanie Weirich contributed various pieces. Corky Cartwright and Bruce Duba made important suggestions in numerous discussions. The authors also gratefully acknowledge the patience of those people who used early versions of DrScheme. The project is partially supported by grants from the National Science Foundation and the Department of Education.

References

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

[2] Dybvig, R. K. *The Scheme Programming Language*. Prentice-Hall, 1987.

[3] Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[4] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 235–271, 1992.

[5] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for scheme. In *Programming Languages: Implementations, Logics, and Programs*, *LNCS* 1292, 369–388, Southampton, UK, 1997.

[6] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching Bugs in the Web of Program Invariants. In *Programming Language Design and Implementation*, 23–32, 1996.

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

[8] Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *Programming Languages: Design & Implementation*, 1998.

[9] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Principles of Programming Languages*, 1998.

[10] Krishnamurthi, S. Zodiac: A framework for building interactive programming tools. Technical Report TR96-262, Rice University, 1996.

[11] Reid, R. J. First-course language for computer science majors. *Posting to comp.edu*, 1995.

[12] Schemer's Inc. and Terry Kaufman. Scheme in colleges and high schools. Available on the web. URL: `http://www.schemers.com/schools.html`.

Matthias Felleisen's (Rice University) research interests cover the entire spectrum of programming languages: design, environments, implementation, logics, and semantics. Over the past few years, he has also become interested in the logical principles of programming and in teaching them to beginning programmers, including high school students. Felleisen has published some 50 papers and four Little books.

Robert Findler, Matthew Flatt, and Shriram Krishnamurthi are currently members of the Rice Programming Languages Team. Findler is the architect of DrScheme. Flatt designed, implemented, and maintains MzScheme and the graphical run-time environment; he is now in the process of modeling the major innovations of these components so that others can benefit, too. Krishnamurthi developed the tower of Scheme subsets; he will translate his experience into theories on and tools for patterns of software construction.