DrScheme: A Programming Environment for Scheme[*]

Robert Bruce Findler, John Clements[†], Cormac Flanagan[‡], Matthew Flatt[§], Shriram Krishnamurthi[¶], Paul Steckler[†], and Matthias Felleisen[†]

Department of Computer Science
Rice University
Houston, Texas 77005-1892, USA

Abstract

DrScheme is a programming environment for Scheme. It fully integrates a graphics-enriched editor, a parser for multiple variants of Scheme, a functional read-eval-print loop, and an algebraic printer. The environment is especially useful for students, because it has a tower of syntactically restricted variants of Scheme that are designed to catch typical student mistakes and explain them in terms the students understand. The environment is also useful for professional programmers, due to its sophisticated programming tools, such as the static debugger, and its advanced language features, such as units and mixins.

Beyond the ordinary programming environment tools, DrScheme provides an algebraic stepper, a context-sensitive syntax checker, and a static debugger. The stepper reduces Scheme programs to values, according to the reduction semantics of Scheme. It is useful for explaining the semantics of linguistic facilities and for studying the behavior of small programs. The syntax checker annotates programs with font and color changes based on the syntactic structure of the program. On demand, it draws arrows that point from bound to binding occurrences of identifiers. It also supports α-renaming. Finally, the static debugger provides a type inference system that explains specific inferences in terms of a value-flow graph, selectively overlaid on the program text.

Keywords. Programming Environments, Programming, Scheme, Pedagogy, Reduction Semantics, Static Debugging.

1 Problems with Teaching Scheme

Over the past fifteen years, Scheme (Clinger & Rees, 1991) has become the most popular functional programming language for introductory courses. Scheme's success is primarily due to Abelson and Sussman's seminal book (Abelson *et al.*, 1985)

[*] A preliminary version of this paper appeared at *Programming Languages: Implementations, Logics, and Programs* in 1997.
[†] Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA, 02115 USA
[‡] Compaq System Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA
[§] Dept. of Computer Science, 50 Central Campus Dr. Rm 3190, Salt Lake City, UT 84112, USA
[¶] Computer Science Department, Brown University, Box 1910, Providence, RI 02912, USA

on their introductory course at MIT. Their course proved that introductory programming courses can expose students to the interesting concepts of computer science instead of just the syntactic conventions of currently fashionable programming languages.

When Rice University implemented an MIT-style course, the instructors encountered four significant problems with Scheme and its implementations (Cadence Research Systems, 1994; Hanson *et al.*, 1993; Schemer's Inc., 1991; Texas Instruments, 1988):

1. Simple notational mistakes produced inexplicable results or incomprehensible error messages because the syntax of standard Scheme is extremely liberal.
2. The available implementations did not pinpoint the source location of run-time errors.
3. The Lisp-style output syntax obscured the pedagogically important connection between program execution and algebraic expression evaluation.
4. The hidden imperative nature of Scheme's read-eval-print loop introduced subtle bugs that easily frustrate students.

In contrast to experienced Scheme programmers who have, often unconsciously, developed work-arounds for these problems, students are confounded by the resulting effects. Consequently, some students dismiss the entire functional approach to programming because they mistake these environmental problems for flaws of the underlying methodology.

To address these problems we built DrScheme, a Scheme programming environment initially targeted at students. The environment eliminates all problems mentioned above by integrating program editing and evaluation in a semantically consistent manner. DrScheme also contains three tools that facilitate the teaching of functional programming and the development of Scheme programs in general. The first is a symbolic stepper. It models the execution of Scheme programs as algebraic reductions of programs to answers without needing to resort to the underlying machine model. The second tool is a syntax checker. It annotates programs with font and color changes based on the syntactic structure of the program. It also permits students to explore the lexical structure of their programs in a graphical manner and to α-rename identifiers. The third tool is a static debugger that infers what set of values an expression may produce and how values flow from position to position in the source text. It exposes potential safety violations and, upon demand, explains its reasoning by drawing value flow graphs over the program text.

Although DrScheme was first designed for beginning students, the environment has grown past its original goals. It is now useful for the development of complex Scheme applications, including DrScheme itself.

The second section of this paper discusses the pedagogy of Rice University's introductory course and motivates many of the fundamental design decisions of DrScheme. The third section presents DrScheme and how it supports teaching functional programming with Scheme. The fourth section briefly explains the additional tools. The next three sections discuss related work, present our experiences, and suggest possible adaptations of DrScheme for other functional languages.

2 Rice University's Introductory Computing Course

Rice University's introductory course on computing focuses on levels of abstraction and how the algebraic model and the physical model of computation give rise to the field's fundamental concerns. The course consists of three segments. The first segment covers (mostly) functional program design and algebraic evaluation. The second segment is dedicated to a study of the basic elements of machine organization, machine language, and assembly language. The course ends with an overview of the important questions of computer science and the key elements of a basic computer science curriculum (Felleisen *et al.*, 2001).

The introduction to functional program design uses a subset of Scheme. It emphasizes program design based on data analysis and computation as secondary school algebra. The course starts out with the design of inductively defined sets of values. Students learn to describe such data structures rigorously and to derive functions from these descriptions. In particular, the course argues that a program consumes and produces data, and that the design of programs must therefore be driven by an analysis of these sets of data.

Students begin with a formal description of the data that a function processes:

A *list of numbers* is either:

1. *empty* (the empty list), or
2. (*cons n lon*) where *n* is a number and *lon* is a *list of numbers*.

and an English description of the function, say, *Length*. Then, they use a five step recipe to derive *Length*, as follows:

1. Write the function header, including both a one-line description of the function and its type.
2. Develop examples that describe the function's behavior.
3. Develop the function template, based on the data definition. If the data definition has more than one alternative, add a **cond**-expression with one case per alternative in the data definition. Add selector expressions to each **cond**-clause, based on the input data in that clause. Add a recursive call for each self-reference in the data definition.
4. Finish the program, starting with the base cases and using the examples.
5. Test the examples from step 2.

This is the final product:

```
;; Length : list of numbers → number
;; to compute the number of elements in the list
(define (Length a-list)
  (cond
    [(empty? a-list) 0]
    [else (add1 (Length (rest a-list)))]))

;; Tests and Examples
```

```
(= (Length empty) 0)
(= (Length (cons 1 (cons 2 empty))) 2)
```

Steps one through three and five are derived mechanically from the data definition and the problem specification. Step 4 is the creative part of the process.

Once the program is designed, students study how it works based on the familiar laws of secondary school algebra. Not counting the primitive laws of arithmetic, two laws suffice: (1) the law of function application and (2) the law of substitution of equals by (provably) equals. A good first example is an application of the temperature conversion function:

```
(define (fahrenheit→celsius f)
  (* 5/9 (- f 32)))

  (fahrenheit→celsius (/ 410 10))
= (fahrenheit→celsius 41)
= (* 5/9 (- 41 32))
= 5
```

Students know this example from secondary school and can identify with it.

For examples that involve lists, students must be taught the basic laws of list-processing primitives. That is, (*cons* v l) is a *list value* if v is a value and l is a list value; (*first* (*cons* v l)) $=$ v and (*rest* (*cons* v l)) $=$ l, for every value v and list value l. From there, it is easy to illustrate how *Length* works:

```
  (Length (cons 8 (cons 33 empty)))
= (add1 (Length (rest (cons 8 (cons 33 empty)))))
= (add1 (Length (cons 33 empty)))
= (add1 (add1 (Length (rest (cons 33 empty)))))
= (add1 (add1 (Length empty)))
= (add1 (add1 0))
= 2
```

In short, algebraic calculations completely explain program execution. No references to the underlying hardware or the runtime context of the code are needed.

As the course progresses, students learn to deal with more complex forms of data definitions, non-structural recursion, and accumulator-style programs. At the same time, the course gradually introduces new linguistic elements as needed. Specifically, for the first three weeks, students work in a simple functional language that provides only function definitions, conditional expressions, data definitions, and basic boolean, arithmetic, and list-processing primitives. Then the language is extended with a facility for local definitions. The final extension covers variable assignment, data mutation, and higher-order functions. With each extension of the language, the course also introduces a set of appropriate design recipes and rewriting rules that explain the new language features (Felleisen, 1988; Felleisen, 1991; Felleisen & Hieb, 1992).

At the end of the segment on program design, students understand how to

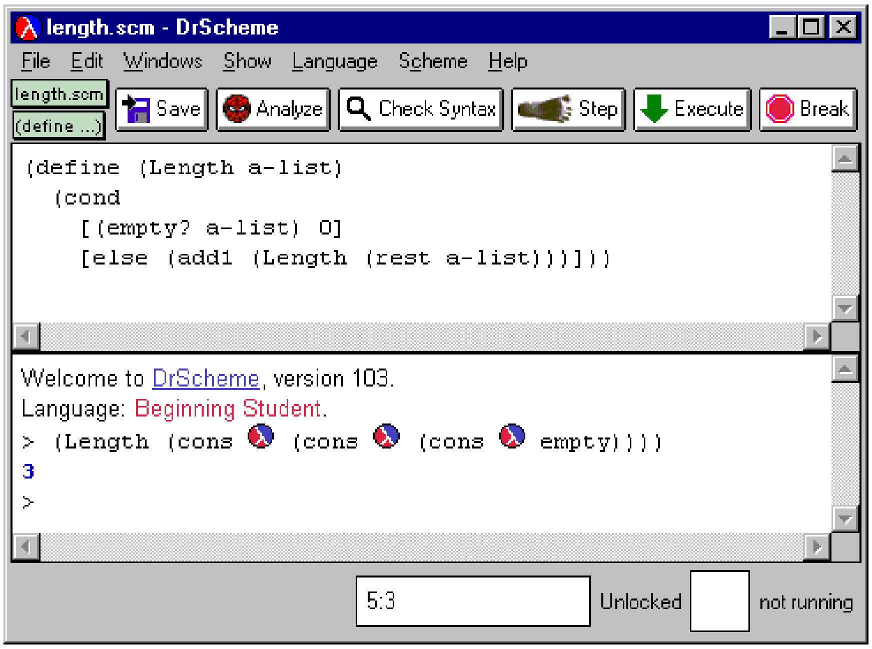

Figure 1. The DrScheme Window (Windows 95/98/NT/2000 version).

construct programs as (collections of) functions and as (object-oriented[1]) history-sensitive procedures. They can evaluate programs by reducing them algebraically to their values and effects, and understand how to use these evaluations to reason about the correctness and complexity of their designs.

For more details on the course itself, see *How to Design Programs* (Felleisen *et al.*, 2001).

3 The Programming Environment

DrScheme runs under Microsoft Windows 95/98/NT/2000, MacOS, and the X Window System. When DrScheme starts up, it presents the programmer with a menubar[2] and a window consisting of three pieces: the control panel, the definitions (upper) window, and the interactions (lower) window (see figure 1). The control panel has buttons for important actions, *e.g.*, Save and Execute. The definitions window is an editor that contains a sequence of definitions and expressions. The interactions window, which provides the same editing commands as the defini-

[1] Functional programming is a data-centric style of programming, just like object-oriented programming. It only lacks dispatch and inheritance, but is the best starting point for teaching object-oriented program design to novice programmers.

[2] Under Windows and X, the menubar appears at the top of the window; under MacOS, the menubar appears at the top of the screen.

tions window, implements a novel read-eval-print loop that supports an algebraic printer and stateless program execution.

DrScheme's menubar provides seven menus: File, Edit, Windows, Show, Scheme, Language, and Help. The File and Edit menus contain the standard menu items. In addition, the latter provides the Edit|Insert Image... menu item, which allows the programmer to insert images into the program text. Images are treated as ordinary values, like numbers or symbols. The Windows menu lists the open DrScheme frames. The Show menu controls the visibility of the sub-windows in a frame. The Scheme menu allows programmers to indent, comment, and uncomment regions of text in the definitions window, and create launchers that enable them to run their programs outside of the environment. The Language menu allows the student to choose which sub-languages of Scheme the syntax checker and evaluator accept. The Help menu provides access to Help Desk, DrScheme's documentation center.

The control panel contains six[3] buttons: Save, Analyze, Check Syntax, Step, Execute, and Break. The Save button saves the definitions from the definitions window as a file. The Analyze button invokes the static debugger (described in section 4.3) on the program in the definitions window. Clicking the Check Syntax button ensures that the definitions window contains a correctly formed program, and then annotates the program based on its syntactic and lexicographical structure (described in section 4.2). The Step button invokes the symbolic stepper (described in section 4.1). The Execute button runs the program in the definitions window, in a fresh environment. All allocated OS resources (including open files, open windows, and running threads) from the last run of the program are reclaimed before the program is run again. Finally, the Break button stops the current computation.

The definitions and interactions windows contain editors that are compatible with typical editors on the various platforms. For example, the editor has many of the Emacs (Stallman, 1987) key bindings. Additionally, the Windows and MacOS versions have the standard key bindings for those platforms.

The remainder of this section motivates and describes the novel aspects of the core programming environment. In particular, the first subsection describes how DrScheme can gradually support larger and larger subsets of Scheme as programmers gain more experience with the language and the functional programming philosophy. The second subsection describes how the definitions window and the interactions window (read-eval-print loop) are coordinated. Finally, the third subsection explains how DrScheme reports run-time errors via source locations in the presence of macros. The remaining tools of DrScheme are described in section 4.

3.1 Language Levels

Contrary to oft-stated claims, learning Scheme syntax poses problems for beginning students who are used to conventional algebraic notation. Almost any program with matching parentheses is syntactically valid and therefore has some meaning.

[3] The standard distribution of DrScheme does not include the static debugger. We describe the version where the static analysis package has been installed.

For beginning programmers that meaning is often unintended, and as a result they receive inexplicable results or incomprehensible error messages for their programs.

For example, the author of the program

```
(define (Length1 l)
  (cond
    [(empty? l) 0]
    [else 1 + (Length1 (rest l))]))
```

has lapsed into algebraic syntax in the second clause of the **cond**-expression. Since Scheme treats each clause in the **cond** expression as an implicit sequence, the value of a **cond**-clause is the value of its last expression. Thus, $Length_1$ always returns 0 as a result, puzzling any beginning programmer.

Similarly, the program

```
(define (Length2 l)
  (cond
    [empty? (l) 0]
    [else (+ 1 (Length2 (rest l)))]))
```

is syntactically valid. Its author also used algebraic syntax, this time in the first **cond**-clause. As a result, $Length_2$ erroneously treats its argument, *e.g.*, (*cons* 1 (*cons* 2 (*cons* 3 *empty*))), as a function and applies it to no arguments; the resulting error message, *e.g.*, "`apply: (1 2 3) not a procedure`", is useless to beginners.

Even though these programs are flawed, the students that wrote them should receive encouragement since the flaws are merely syntactic. They clearly understand the inductive structure of lists and its connection to the structure of recursive programs. Since Scheme's normal response does not provide any insight into the actual error, the students' learning experience suffers. A good programming environment should provide a correct and concise explanation of the students' mistakes.

Students may also write programs that use keywords as identifiers. If the student has not yet been taught those keywords, it is not the student's fault for misusing the keywords. A programming environment should limit the language to the pieces relevant for each stage of a course rather than leaving the entire language available to trap unwary students. For example, a student that has not yet been introduced to sequencing might write:

```
(define (Length3 l start)
  (cond
   [(empty? l) start]
   [else (Length3 (rest l) (add1 begin))]))
```

This program is buggy; it has an unbound identifier *begin*. But a conventional Scheme implementation generates a strange syntax error:

```
compile: illegal use of a syntactic form name in: begin
```

Students cannot possibly understand that they have uncovered a part of the programming language that the course has not yet introduced.

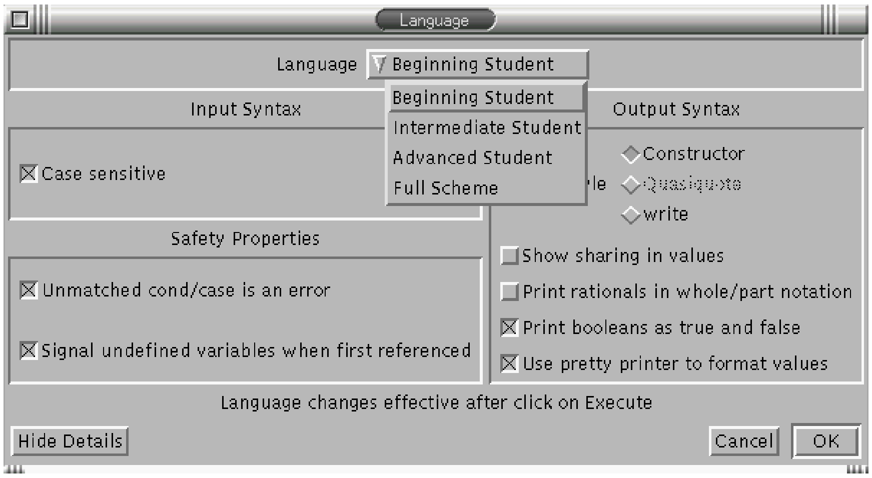

Figure 2. DrScheme's Language Configuration Dialog Box (X version).

Eager students also attempt to use features that they have not yet seen in class, often incorrectly. For example, many students might try to use local definitions before scope is described in class. Others try to return more than one value from a function by juxtaposing several expressions behind **lambda**. Students with prior experience in C or Pascal might solve a simple functional exercise with imperative features. Again, a good pedagogic programming environment should protect the student from using language features that are inconsistent with the pedagogic goals of a particular phase of the course.

A natural solution for all of these problems is to stratify the programming language into several levels. Each level should provide enough power to teach a new set of constructs and programming paradigms, and it must not allow irrelevant language features to interfere with the goals of a teaching unit. In short, a pedagogic programming environment must be able to grow along with the students through a course. DrScheme implements this stratification with four language levels (Flatt, 1997; Krishnamurthi *et al.*, 1999b).

DrScheme's current language is set via the Language|Configure Language... menu item. Choosing Language|Configure Language... opens a window with a choice dialog item that displays the current language level. The choice dialog item describes the student's language level. A language consists of several independent settings, which are normally hidden from the student. Clicking the Show Details button enlarges the dialog, bringing a panel with all of the language settings into view. Figure 2 shows the enlarged dialog.

Each language level consists of three parts: input syntax, safety properties, and output syntax (described in section 3.3). The input syntax is specified through the Case Sensitive check box and the language level itself. The four languages are: *Beginning Student*, *Intermediate Student*, *Advanced Student*, and *Full Scheme*. Each

Beginning has:
: **define**, first-order procedures with at least one argument, **cond**, **if**, **quote**'d symbols, **define-struct**, and various functional primitives. It is also case-sensitive.

Intermediate adds:
: **quote**'d lists, **quasiquote**, **unquote**, higher-order functions with at least one argument, and lexical scope: **let**, **local**, and **letrec**.

Advanced adds:
: **delay**, *force*, **set!**, **begin**, **when**, **unless**, *call/cc*, higher-order functions (possibly with zero arguments), various imperative primitives, and sharing in the REPL printer.

Full adds:
: the full set of Scheme primitives (Clinger & Rees, 1991; Flatt, 1997), and improper lists. *Full* is not case-sensitive, unmatched cond/case expressions result in (*void*), and sharing is not shown in the REPL printer. *Full* optionally adds a full-featured GUI programming toolkit (Flatt & Findler, 1997).

Figure 3. Language Level Quick Reference.

language corresponds to a stage in Rice University's introductory course. Figure 3 specifies the content for each of the language levels.

The safety properties check boxes allow the student to choose between conformance with R^5RS and more sensible error reporting. They can be specified with two check boxes:

- Unmatched cond/case is an error
- Signal undefined variables when first referenced

If Unmatched cond/case is an error is on, DrScheme inserts an implicit **else** clause in **cond** and **case** expressions that signals a run-time error. If it is off, the implicit **else** clause returns a dummy value. The Signal undefined variables when first referenced check box controls the implementation's behavior when evaluating recursive definitions. DrScheme evaluates recursive binding expressions by initializing all identifiers being bound to a special tag value, and then evaluating each definition and re-binding each identifier. If the checkbox is on, an error is signaled when a variable still bound to one of the tag values is evaluated, and if off, errors are only signaled if the special tag value flows into a primitive function.

Although the restrictions of the teaching languages are important for good feedback to students, it is also important to allow students access to modern features of computers such as a GUI toolkit, TCP/IP network connections, and so on. DrScheme permits this through the TeachPack mechanism. A TeachPack is a library that is implemented in the Full Scheme language and whose exports are made available (as primitive operations) to students programming in the teaching levels. Using TeachPacks, students can plug in the core functionality of a web-server or a graphical game into an instructor-provided infrastructure. This lets the students play the game with all of the fancy graphics or interact with the web-server in their own browser without having to waste time learning libraries that will probably be out of date by the time they graduate.

Although Eli Barzilay, Max Halipern, and Christian Quennec have customized DrScheme's language levels for their courses, this is a complex task and is not yet well-supported in DrScheme. Since the task of designing a set of appropriate language levels is inherently difficult, we probably cannot provide a simple mechanism to adapt the pre-defined language levels to a particular course. We do plan, however, to support new language levels in a black-box manner and with good documentation in the next version of DrScheme.

3.2 Interactive Evaluation

Many functional language implementations support the interactive evaluation of expressions via a read-eval-print loop (REPL). Abstractly, a REPL allows students to define new functions and to evaluate expressions in the context of a program's definitions. A typical REPL implements those operations by prompting the students to input program fragments. The implementation evaluates the fragments, and prints their results.

Interactivity is primarily used for program exploration, *e.g.* the process of evaluating stand-alone expressions in the context of a program to determine its behavior. It is critical for novice programmers because it eliminates the need to learn specialized input-output libraries. Also, frequent program exploration during development saves large amounts of conventional debugging time. Programmers use interactive environments to test small components of their programs and determine where their programs go wrong.

While interactive REPLs are superior to batch execution for program development, they can introduce subtle and confusing bugs into programs. Since they allow ad-hoc program construction, REPLs cause problems for the beginning student and experienced programmer alike. For example, a student who practices accumulator-style transformations may try to transform the program

```
(define (Length l)                (define (length-helper l n)
  (length-helper l 0))              (cond
                                     [(empty? l) n]
                                     [else (length-helper (rest l) (add1 n))]))
```

into a version that uses local definitions:

```
(define (Length l)
  (local [(define (helper l n)
            (cond
             [(empty? l) n]
             [else (length-helper (rest l) (add1 n))]))]
    (helper l 0)))
```

Unfortunately, the student has forgotten to change one occurrence of *length-helper* to *helper*. Instead of flagging an error when this program is run, the traditional

Scheme REPL calls the old version of *length-helper* when *Length* is applied to a non-empty list. The new program has a bug, but the confusing REPL semantics hides the bug.

Similar but even more confusing bugs occur when programmers use higher-order functions. Consider the program:

```
(define (make-adder n)
  (lambda (m)
    (* m n)))

(define add11 (make-adder 11))
```

The programmer quickly discovers the bug by experimenting with *add11*, replaces the primitive $*$ with $+$, and reevaluates the definition of *make-adder*. Unfortunately, the REPL no longer reflects the program, because *add11* still refers to the old value of *make-adder* and therefore still exhibits the buggy behavior, confusing the student. The problem is exacerbated when higher-order functions are combined with state, which is essential for explaining object-oriented ideas.

Experienced functional programmers have learned to avoid this problem by using their REPL in a fashion that mimics batch behavior for definitions and interactive behavior for expressions. They exit the REPL, restart the evaluator, and re-load a program file after each change. This action clears the state of the REPL, which eliminates bugs introduced by ghosts of old programs. Unfortunately, manually restarting the environment is both time-consuming and error-prone.

DrScheme provides and enforces this batch-oriented style of interactive program evaluation in a natural way. When the programmer is ready to test a program, a click on the Execute button submits the program to the interactions window. When the programmer clicks on Execute, the REPL is set to its initial state and the text from the definitions window is evaluated in the fresh environment. Thus, the REPL namespace exactly reflects the program in the definitions window. Next, the programmer evaluates test expressions in the REPL. After discovering an error, the programmer edits the definitions and clicks the Execute button. If the programmer forgets to execute the program and tries to evaluate an expression in the REPL, DrScheme informs the programmer that the text of the program is no longer consistent with the state of the REPL. In short, after every change to the program, the programmer starts the program afresh, which eliminates the problems caused by traditional REPLs. For large programs, restarting the entire program to test a single change can be time consuming. Although restarting the REPL in this manner can be time-consuming for large programs, the first author still uses this style REPL to develop DrScheme itself, albeit with selected, unchanged libraries pre-compiled to an intermediate byte-code representation.

3.3 Output Syntax

As discussed in section 2, Rice University's introductory course emphasizes the connection between program execution and algebraic expression evaluation. Students learn that program evaluation consists of a sequence of reduction steps that transform an expression to a value in a context of definitions.

Unfortunately, traditional Scheme implementations do not reinforce that connection; they typically use one syntax for values as input and a different syntax for values as output (Wadler, 1987). For example the expression:

(*map add1* (*cons* 2 (*cons* 3 (*cons* 4 *empty*))))

prints as

(3 4 5)

which causes students to confuse the syntax for application with the syntax for lists.

DrScheme uses an output syntax for values called constructor syntax that matches their input syntax. Constructor syntax treats the primitives *cons*, *list*, *vector*, *box*, *etc.*, as constructors. In other words, the type of value is clear from its printed representation.

In the the above example, DrScheme prints the value of:

(*map add1* (*cons* 2 (*cons* 3 (*cons* 4 *empty*))))

as either

(*cons* 3 (*cons* 4 (*cons* 5 *empty*)))

or, if the student has turned on a concise printing mode (introduced automatically in the second language level)

(*list* 3 4 5)

Thus, DrScheme's REPL produces the same syntax for the values that Scheme's reduction semantics produces. This also reinforces the data driven design methodology in section 2.

The standard Scheme printer is useful, however, for programs that manipulate program text. Often, it is convenient to create and manipulate lists of lists and symbols, especially when they represent programs. Scheme supports this with the **quote** and **quasiquote** operators. For example, the list containing the three elements: a symbol 'lambda, a list of symbols representing parameters, and an expression (again represented as a list of lists and symbols), can be written '(lambda (x y z) (+ x y)). When writing programs that treat quoted s-expressions in this way, the output of the original Scheme matches the programmer's mental model of the data. Accordingly, DrScheme provides a quasiquote (Pitman, 1980) printer that uses the same structure, but matches Scheme's quasiquote input syntax, thus preserving the algebraic model of evaluation.

3.4 Error Reporting

A programming environment must provide good run-time error reporting; it is crucial to a student's learning experience. The programming environment must

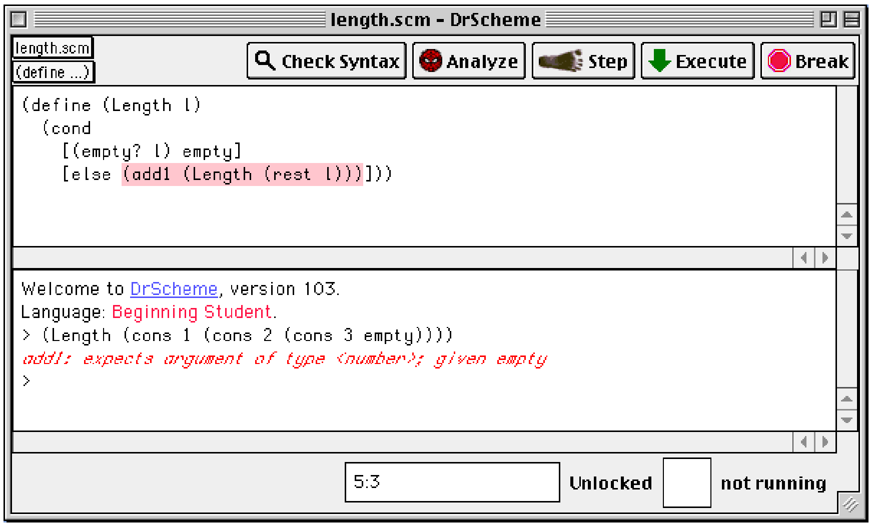

Figure 4. DrScheme, with a Run-time Error Highlighted (MacOS version).

catch errors as soon as they occur and provide meaningful explanations for them. The explanations must include the run-time values that caused the errors as well as the source location of the misapplied primitives.

Traditional Scheme programming environments fail in this regard for two reasons. First, with the exception of EdScheme (Schemer's Inc., 1991), Scheme compilers and interpreters only implement a simplistic read-eval-print loop. If this REPL is an command shell that is not coupled to the rest of the programming environment, it is impossible to relate errors to source locations in general. The historical solution is to execute the REPL in an Emacs buffer. This solution, however, does not truly integrate the REPL and its editing environment, so the full graphical capabilities of modern displays remain unexploited.

Second, Scheme's macro facility (Kohlbecker *et al.*, 1986; Kohlbecker, 1986) tremendously complicates the mapping from a run-time error to its source location. Since Scheme's macro language allows arbitrary transformations of program text during compilation, preserving the original source locations for pieces of program text is difficult. For example, Scheme's **let**∗ macro expands to a sequence of nested **let** expressions, and those **let** expressions then expand into **lambda** expressions. Other macros duplicate or delete portions of source text.

Since DrScheme integrates the editor and the Scheme implementation, it can support a much tighter interaction than standard Scheme implementations. The underlying Scheme implementation is safe and completely integrated into the editing environment. Furthermore, the front-end of the Scheme implementation maintains a correlation between the original program text and its macro-expanded version (Krishnamurthi *et al.*, 1999a; Krishnamurthi *et al.*, 1999b). This correlation allows DrScheme to report the source location of run-time errors.

Consider the example in figure 4. The student has written an erroneous version of *Length*. When it is applied to (*cons* 1 (*cons* 2 (*cons* 3 *empty*)))), *Length* traverses the list and is eventually applied to *empty*. The function then returns *empty*, which flows into the primitive +, generating a run-time error. At this point, DrScheme catches the run-time error and highlights the source location of the misapplied primitive. With almost no effort, any beginning student can now find and fix the bug.

4 DrScheme Tools

Thus far we have seen how DrScheme stratifies Scheme into pedagogically useful pieces, improves the read-eval-print loop, and provides better error reporting. This section focuses on the additional program understanding tools that DrScheme provides.

4.1 Supporting Reduction Semantics: The Stepper

DrScheme includes a stepper that enables students to reduce a program to a value in a series of reduction steps analogous to the calculations in secondary school algebra. At each step of the evaluation, this tool highlights the reducible expression and shows its replacement.

Figure 5 shows the basic layout of the stepper window. The window is separated by divider lines into four panes. The topmost pane shows the evaluated definitions. In general, this pane contains all values and definitions that have been reduced to a canonical form. The second and third panes show the current reduction step. In this case, the expression (+ 4 5) is reduced to 9. The reducible expression is highlighted in green, while the result of the reduction is highlighted in purple. The final pane, below the other panes, is used for top-level forms that have not yet been evaluated.

Figure 6 also illustrates the reduction of a procedure call, using the *Length* function defined earlier. In this step, the call to *Length* is replaced by the body of the procedure. Every occurrence of *Length*'s parameter, *l*, is replaced by the argument, (*cons* 'banana *empty*). Again, this mirrors the standard evaluation rules taught for functions.

Using the stepper, students see that a computation in the beginner level consists of a series of local transformations; nothing outside the colored boxes is changed. The stepping model is familiar, as it mirrors the reduction rules taught in grade school arithmetic and secondary school algebra.

DrScheme's intermediate level introduces the **local** binding construct, which allows students to create local bindings. Since local-bindings are recursive and a locally defined procedure may (eventually) escape this scope, the reduction rules are more complex than the simple substitution-based **let** and application rules. Instead, the reduction semantics lifts and renames the **local** definitions to the top-level, as shown in figure 7. The bindings are renamed during lifting to avoid capture in case a local expression is evaluated more than once and to avoid collision with other top-level variables.

When runtime errors occur, the stepper displays the error along with the expres-

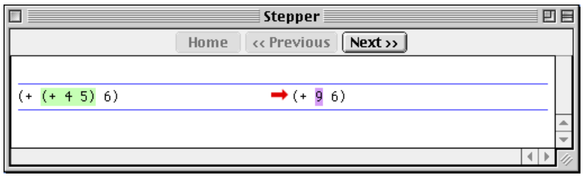

Figure 5. An Arithmetic Reduction in the Stepper (MacOS version).

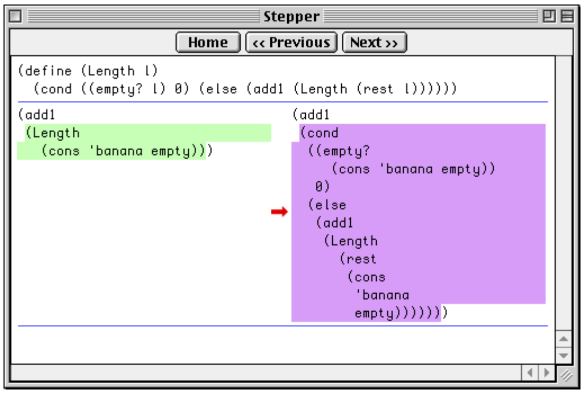

Figure 6. A Procedure Application in the Stepper (MacOS version).

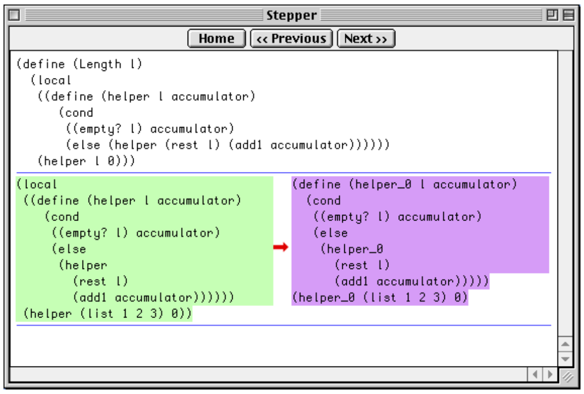

Figure 7. Reducing **local** (MacOS version).

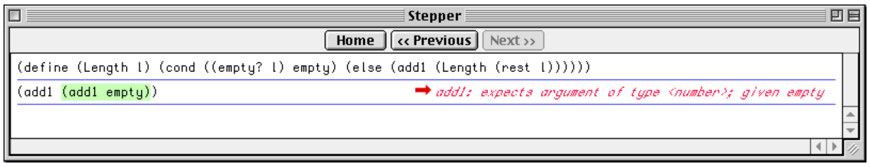

Figure 8. An Error in the Stepper (MacOS version).

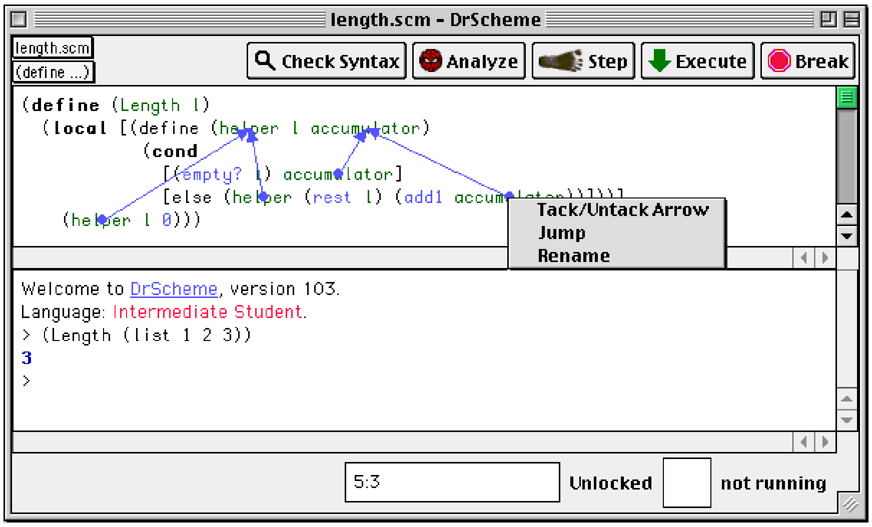

Figure 9. DrScheme's Syntax Checker (MacOS version).

sion that caused it. Figure 4, in section 3.4, showed a runtime error in DrScheme's evaluator. In figure 8, the same error is shown as it occurs during the stepper's evaluation. To locate the source of the offending *empty*, students may wish to view earlier steps in the evaluation. The stepper therefore retains all evaluation steps, so that students may step backward and forward as desired, using the Previous and Next buttons.

The stepper is useful for students in two ways. First, it concretely demonstrates the semantics of the language. Students that do not understand the evaluation rules of Scheme can observe them in action and formulate their own examples. This is particularly useful for students that prefer to learn by generalizing from examples, rather than working directly from an abstract model. Secondly, it is used as a debugging tool. Students observe the evaluation of their own programs, and can step forward and backward to locate their mistakes.

The key to the construction of the stepper is the introduction of *continuation marks* (Clements *et al.*, 2001). Continuation marks allow the stepper to re-use the underlying Scheme implementation, without having to re-implement the evaluator. As a result, the stepper and the compiler always have the same semantics.

Future versions of DrScheme will extend the stepper to handle the Advanced and Full Scheme levels. This will include side-effects, higher-order procedures, *call/cc*, and multi-threaded programs. These extensions are based on Felleisen and Hieb's work (Felleisen & Hieb, 1992).

4.2 Syntax Checking

Programmers need help understanding the syntactic and lexical structure of their programs. DrScheme provides a syntax checker that annotates the source text of

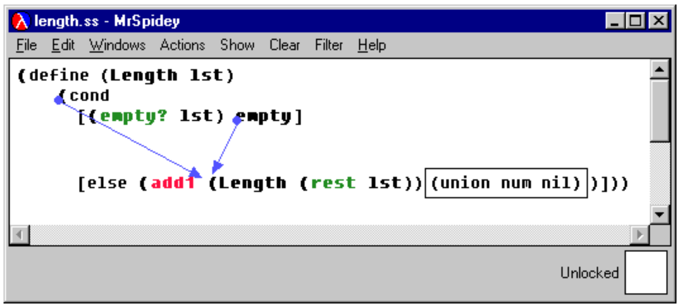

Figure 10. MrSpidey: The static debugger (Windows version).

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.

When the programmer moves the mouse over an identifier, the syntax checker displays arrows that point from bound identifiers to their binding occurrence, and and vice-versa (see figure 9). The checker can also α-rename bound identifiers. To do so, the user clicks on either a binding or bound occurrence of the identifier. This pops up a menu (as shown in figure 9). After choosing Rename and specifying a new name for the identifier, DrScheme renames all occurrences of the identifier to the new name. In the teaching languages, students can click on a primitive to see its contract and one-line description, as in step 1 of the design recipe in section 2.

4.3 Static Debugging

The most advanced DrScheme tool is MrSpidey, a static debugger (Bourdoncle, 1993; Flanagan *et al.*, 1996) that uses a form of set-based analysis (Flanagan & Felleisen, 1997; Heintze, 1994) to perform type inference and to mark potential errors. The static debugger tool infers constraints on the flow of values in a Scheme program. From those constraints, it infers value-set descriptions for each subexpression. On demand, it builds a graph that demonstrates how values flow though the program. For each primitive program operator,[4] the static debugger determines whether its potential argument values are valid inputs.

Based on the results of the analysis, the static debugger annotates the program with font and color changes. Primitive operations that may be misapplied are highlighted in red, while the rest are highlighted in green. By choosing from a popup menu, the static debugger can display an inferred “value set” for program points, and arrows overlaid on the program text describe data flow paths into program points. Unlike conventional unification-based type systems, MrSpidey’s arrows provide a clear description of its inferences, which is especially helpful when searching for the sources of potential errors. For each red-colored primitive, the programmer

[4] Primitive program operators include **if**, **cond**, and procedure application.

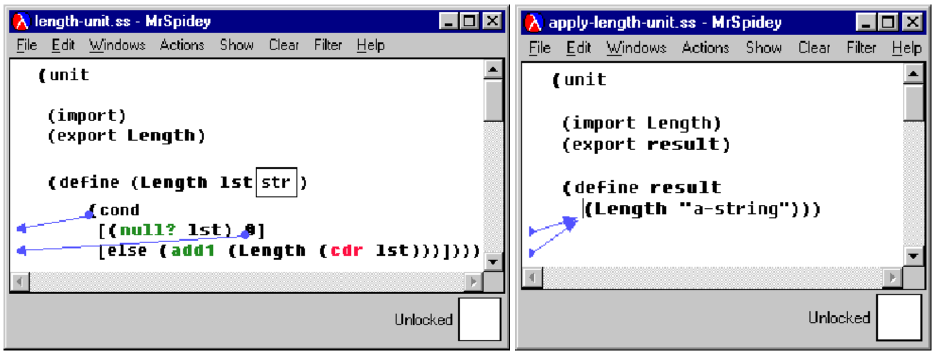

Figure 11. Analyzing multiple units in MrSpidey (Windows version).

can use type and data flow path information to determine if a program has an error. In some cases, MrSpidey's analysis is too weak to capture a program invariant. In those cases, the programmer might still use MrSpidey's information to tune the program to the analysis.

For an illustration, reconsider the flawed *Length* program from Section 3.4, shown in Figure 10. After the static debugger completes its analysis, it opens a window containing the analyzed program, with each primitive colored either red or green. In this example, *add1* is the only primitive colored red, indicating that the static debugger cannot prove that the argument is always a number. The programmer can now display the type of *add1*'s argument. The static debugger inserts a box that contains the type of the argument to the right of its text, as shown in Figure 10. In this case, the type contains *nil*, the type of *empty*, explaining why the static debugger concluded that *add1* may be misapplied. To explain how *empty* may flow into the argument of *add1*, the static debugger displays an arrow pointing from the constant *empty* to the argument of *add1*, as shown in Figure 10. At this point, the static debugger has given the programmer enough information to uncover the bug, that a recursive application of *Length* can result in the flow of *empty* into the argument of *add1*.

MrSpidey can also perform *componential analysis* on programs divided into units in separate files. Figure 11 shows the analysis of a program containing two units, where each unit is contained in a separate file. One unit defines a procedure *Length*, which (this time correctly) calculates the length of a list. The other unit imports the *Length* procedure, and applies it to a string. MrSpidey is able to track the data path for the definition of *Length* across unit boundaries. As shown in the figure, the instances of the variable *lst* in the procedure body are inferred to be strings. Hence, the occurrence of *cdr* is highlighted as unsafe.

MrSpidey is useful for realistic programs. Recently the authors, in collaboration with Paul Graunke, designed and implemented an extensible Web server (Graunke *et al.*, 2001). We developed the server in DrScheme and used MrSpidey to verify its basic correctness properties. Currently, the server consists of about one thousand lines of code. Using MrSpidey, the authors uncovered several small bugs in the implementation that tests had not revealed. MrSpidey also pinpointed a potential fault site in a string-processing primitive that the authors did not understand until after deployment of the server. While using MrSpidey for large projects is not yet simple enough for ordinary Scheme programmers, we believe that including tools such as MrSpidey in modern programming environments illustrates their use and creates future demand.

5 Related Work

DrScheme integrates a number of ideas that are important for programming with functional languages, especially at the introductory level: well-defined simple sub-languages, a syntax checker with lexical scope analysis, a read-eval-print loop (REPL) with transparent semantics, precise run-time error reporting, an algebraic printer, an algebraic stepper, and a full-fledged static debugger. The restriction of the full language to a hierarchy of simplified sub-languages, the syntax checker, the algebraic stepper for full Scheme, the transparent REPL, and the static debugger are novel environment components that no other programming environment provides.

Most of the Scheme implementations available are simple REPLs that do not provide a development environment. One notable exception is Bee (Serrano, 2000), a programming environment designed for professional programmers. It has many advanced features and outstanding debugging tools, in particular an excellent memory profiler and memory debugger (Serrano & Boehm, 2000).

In lieu of good error reporting, other Scheme implementations provide tracers, stack browsers, and conventional breakpoint-oriented debuggers. In our experience, these tools are too complex to help novice students. Worse, they encourage students with prior experience in Pascal or C++ to fall back into the traditional tinker-until-it-works approach to program construction.

Other functional language environments provide some of the functionality of DrScheme. Specifically, SML/NJ provides a REPL similar to the one described here for the module language of ML (Blume, 1995; Harper *et al.*, 1994). Unfortunately this is useless for beginners, who mostly work with the core language. Also, OCaml (Leroy, 1997), MLWorks (Harlequin Inc., 1996), and SML/NJ (AT&T Bell Labratories, 1993) have good source reporting for run-time errors but, due to the unification-based type inference process, report type errors of programs at incorrect places and often display incomprehensible messages.

Commercial programming environments (Borland, 1983, 2000; Metrowerks, 1993–1996; Microsoft, 1995) for imperative programming languages like C++ incorporate a good portion of the functionality found in DrScheme. Their editors use online real-time syntax coloring algorithms,[5] the run-time environments trap segmentation faults and highlight their source location, but that is much less useful than catching safety violations. Their debuggers serve as primitive REPLs, though with much less flexibility than the REPLs that come with Scheme or ML. None of these

[5] These syntax coloring tools typically ignore macro systems, making the task much easier.

tools, however, provides language levels, full-fledged algebraic printers, steppers, α-renaming or static debuggers, which we have found to be extremely useful for teaching and development.

Also, commercial environments have sophisticated, integrated, online documentation systems. DrScheme's Help Desk is a sophisticated, online documentation system, but it is not as well integrated into the environment as those available in commercial development systems. Since Scheme's facilities for constructing programs (macros, *load*, *eval*, etc.) are so permissive it is impossible to know much about the structure of the program without evaluating it. Future releases[6] of DrScheme will incorporate program construction features that allow us to integrate the online documentation and other tools more tightly, without giving up on the expressiveness of the conventional Scheme top-level.

One other Scheme programming environment was specifically designed for beginners: EdScheme (Schemer's Inc., 1991). It supports a rich set of pedagogic graphics libraries. Compared to DrScheme, however, it lacks a sophisticated editor, language levels, and the tools that DrScheme supports.

6 Experience

All five faculty members who teach the introductory computer science course at Rice University use DrScheme. DrScheme is used on a daily basis for the lectures, tutorials, and homework assignments. Also, several upper level courses at Rice have used DrScheme, including the programming languages course, a program construction course, the artificial intelligence course, the graphics course and a reduction semantics course. Programs in these courses range from 10 lines to 5000 lines.

We have also received enthusiastic reports from professors and teachers around the world who use DrScheme in their classes. Our mailing lists consist of several hundred people in academia and industry who use DrScheme and its application suite. DrScheme is used in several hundred colleges, universities, and high schools. We are also aware of several commercial efforts that are incorporating portions of our suite into their products.

DrScheme has grown past its original goals as a pedagogic programming environment. DrScheme now has libraries for most common tasks, including networking, and full-featured GUI programs. DrScheme has a class-based object system that supports mixins (Flatt *et al.*, 1999a) and a sophisticated module system that supports component programming (Flatt & Felleisen, 1998). DrScheme also includes a project manager to manage multi-file programs that is integrated with the syntax checker. In fact, the authors use DrScheme almost exclusively when developing DrScheme, which consists of more than 200,000 lines of Scheme code.

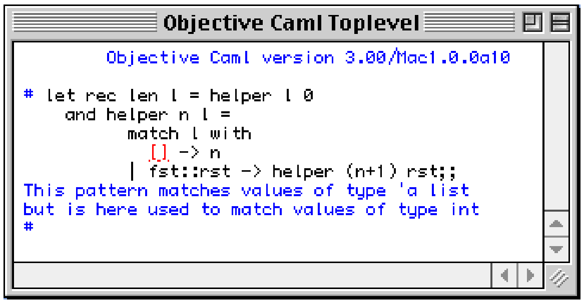

Figure 12. Erroneous ML program and OCaml's response (MacOS version).

7 From Scheme to ML and Haskell

Many of DrScheme's ideas apply to ML and Haskell programming environments *mutatis mutandis*. For example, many functional programming language implementations already have an algebraic printer. The syntax checker applies equally well to any other programming language whose binding structure is syntactically apparent. A stepper should be as useful for ML and Haskell as it is for Scheme. The stepper's implementation technique applies to both ML and Haskell, since it supports state, continuations, multiple threads of control and lazy evaluation (Clements *et al.*, 2001).

In principle, Haskell and ML would be excellent choices for teaching introductory programming. Indeed, their type discipline helps enforce our design discipline from section 2. Unfortunately, the error messages for programs that do not type-check are often confusing to beginners. Consider the program in figure 12. OCaml highlights the first pattern in the definition of *helper*, which is not where where the logical error occurs.

While Duggan (Duggan & Bent, 1996), Wand (Wand, 1986), and others have studied the problem of understanding type errors, their solutions do not apply to introductory programming environments, since they require the student to understand the type system. DrScheme's development suggests an alternative to these approaches. Specifically, stratifying both the type language and the core language into several levels would benefit both Haskell and ML. For example, the first teaching level could enforce the syntax of first-order, explicitly mono-typed function definitions. The next level could introduce polymorphism and first-class functions. In Haskell, the introductory languages could hide type classes. If, in addition, the elaborator is aware of these language levels, the programming environment could produce type errors that are appropriate to the teaching level. The programming environment could also mimic DrScheme's source-sensitive macro-elaborator and thus produce error messages in terms of the source text, instead of the elaborated core-language program.

[6] At the time of this writing, the current release of DrScheme is 103.

Finally, the semantics of ML's "infinite let" style REPL can mask bugs in students' programs, just as Scheme's REPL can. Consider this erroneous program:

$-$ **fun** *helper* [] $n = n$
  | *helper* $(x{::}l)$ $n =$ *helper* l n;
val *helper* $=$ *fn* : α *list* $\rightarrow \beta \rightarrow \beta$
$-$ **fun** *len* $l =$ *helper* l 0;
val *len* $=$ *fn* : α *list* $\rightarrow$ *int*

Suppose a student decides to fix the program by resubmitting the *helper* function to the REPL:

$-$ **fun** *helper* [] $n = n$
  | *helper* $(x{::}l)$ $n =$ *helper* l $(n{+}1)$;
val *helper* : α *list* $\rightarrow$ *int* $\rightarrow$ *int*

When the student then tests the new function:

$-$ *len* [1, 2, 3];
val *it* $=$ 0 : *int*

the bug is still present, even though *helper* has been fixed. Thus, Haskell and ML environments would benefit from a DrScheme-style REPL.

8 Conclusion

The poor quality of programming environments for Scheme distracts students from the study of computer science principles. DrScheme is our response to these problems. We hope that DrScheme's success with students, teachers, and programmers around the world inspires others to build programming environments for functional languages.

DrScheme is available on the web at `http://www.drscheme.org/`.

Acknowledgments

Thanks to R. Cartwright and D.P. Friedman for valuable feedback on early drafts of this paper. Thanks to the JFP reviewers and the PLILP reviewers and audience for their valuable comments and feedback. Thanks also to our users for their patience and feedback, which has helped improve DrScheme tremendously over the years.

References

Abelson, Harold, Sussman, Gerald Jay, & Sussman, Julie. (1985). *Structure and interpretation of computer programs.* MIT Press.

AT&T Bell Labratories. (1993). *Standard ML of New Jersey.* AT&T Bell Labratories.

Blume, Matthias. (1995). *Standard ML of New Jersey compilation manager.* Manual accompanying SML/NJ software.

Borland. (1983, 2000). *Borland C++Builder 5 developer's guide.* INPRISE Corporation, Inc.

Bourdoncle, François. (1993). Abstract debugging of higher-order imperative languages. *Pages 46–55 of: ACM SIGPLAN conference on Programming Language Design and Implementation.*

Cadence Research Systems. (1994). *Chez Scheme Reference Manual.*

Clements, John, Flatt, Matthew, & Felleisen, Matthias. (2001). Modeling an algebraic stepper. *European Symposium on Programming.*

Clinger, William, & Rees, Jonathan. (1991). The revised^4 report on the algorithmic language Scheme. *ACM Lisp pointers*, **4**(3).

Duggan, Dominic, & Bent, Frederick. 1996 (June). Explaining type inference. *Science of Computer Programming.*

Felleisen, Matthias. (1988). An extended λ-calculus for Scheme. *Pages 72–84 of: ACM symposium on Lisp and Functional Programming.*

Felleisen, Matthias. (1991). On the expressive power of programming languages. *Science of Computer Programming*, **17**, 35–75.

Felleisen, Matthias, & Hieb, Robert. (1992). The revised report on the syntactic theories of sequential control and state. *Pages 235–271 of: Theoretical Computer Science.*

Felleisen, Matthias, Findler, Robert Bruce, Flatt, Matthew, & Krishnamurthi, Shriram. (2001). *How to Design Programs.* MIT Press.

Flanagan, Cormac, & Felleisen, Matthias. (1997). Componential set-based analysis. *ACM SIGPLAN conference on Programming Language Design and Implementation.*

Flanagan, Cormac, Flatt, Matthew, Krishnamurthi, Shriram, Weirich, Stephanie, & Felleisen, Matthias. 1996 (May). Catching bugs in the web of program invariants. *Pages 23–32 of: ACM SIGPLAN conference on Programming Language Design and Implementation.*

Flatt, Matthew. (1997). *PLT MzScheme: Language manual.* Technical Report TR97-280. Rice University.

Flatt, Matthew, & Felleisen, Matthias. 1998 (June). Units: Cool modules for HOT languages. *Pages 236–248 of: ACM SIGPLAN conference on Programming Language Design and Implementation.*

Flatt, Matthew, & Findler, Robert Bruce. (1997). *PLT MrEd: Graphical toolbox manual.* Technical Report TR97-279. Rice University.

Flatt, Matthew, Krishnamurthi, Shriram, & Felleisen, Matthias. (1999a). A programmer's reduction semantics for classes and mixins. *Formal syntax and semantics of Java*, 241–269. preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998.

Flatt, Matthew, Findler, Robert Bruce, Krishnamurthi, Shriram, & Felleisen, Matthias. 1999b (Sept.). Programming languages as operating systems (*or* revenge of the son of the Lisp machine). *Pages 138–147 of: ACM SIGPLAN International Conference on Functional Programming.*

Francez, Nissim, Goldenberg, Shalom, Pinter, Ron Y., Tiomkin, Michael, & Tsur, Shalom. (1985). An environment for logic programming. *SIGPLAN Notices*, **20**(7), 179–190.

Graunke, Paul, Krishnamurthi, Shriram, Hoeven, Steve Van Der, & Felleisen, Matthias. 2001 (April). Programming the web with high-level programming languages. *European symposium on programming.*

Hanson, Chris, The MIT Scheme Team, & A Cast of Thousands. (1993). *MIT Scheme Reference.*

Harlequin Inc. (1996). *MLWorks.* Harlequin Inc.

Harper, Robert, Lee, Peter, Pfenning, Frank, & Rollins, Eugene. (1994). *Incremental* *recompilation for Standard ML of New Jersey*. Technical Report CMU-CS-94-116. Carnegie Mellon University.

Heintze, Nevin. (1994). Set based analysis of ML programs. *ACM symposium on Lisp and Functional Programming*.

Hsiang, J., & Srivas, M. 1984 (July). *A Prolog environment*. Tech. rept. 84-074. State University of New York at Stony Brook, Stony Brook, New York.

Kohlbecker, Eugene E. 1986 (Aug.). *Syntactic extensions in the programming language Lisp*. Ph.D. thesis, Indiana University.

Kohlbecker, Eugene E., Friedman, Daniel P., Felleisen, Matthias, & Duba, Bruce F. (1986). Hygienic macro expansion. *Pages 151–161 of: ACM symposium on Lisp and Functional Programming*.

Komorowski, Henryk Jan, & Omori, Shigeo. (1985). A model and an implementation of a logic programming environment. *SIGPLAN Notices*, **20**(7), 191–198.

Koschmann, Timothy, & Evens, Martha Walton. (1988). Bridging the gap between object-oriented and logic programming. *IEEE software*, **5**(July), 36–42.

Krishnamurthi, Shriram, Erlich, Yan-David, & Felleisen, Matthias. 1999a (Mar.). Expressing structural properties as language constructs. *Pages 258–272 of: European Symposium on Programming*. Lecture Notes in Computer Science, no. 1576.

Krishnamurthi, Shriram, Felleisen, Matthias, & Duba, Bruce F. 1999b (Sept.). From macros to reusable generative programming. *Pages 105–120 of: International Symposium on Generative and Component-Based Software Engineering*. Lecture Notes in Computer Science, no. 1799.

Lane, A. (1988). Turbo Prolog revisited. *Byte*, **13**(10), 209–212.

Leroy, Xavier. (1997). *The Objective Caml system, Documentation and User's guide*.

Metrowerks. (1993–1996). *CodeWarrior*. Metrowerks.

Microsoft. (1995). *Microsoft Developer Studio*. Microsoft.

Pitman, Kent M. (1980). Special forms in lisp. *Pages 179–187 of: Lisp conference*.

Schemer's Inc. (1991). *EdScheme: A Modern Lisp*.

Serrano, Manuel. (2000). Bee: an integrated development environment for the Scheme programming language. *Journal of Functional Programming*.

Serrano, Manuel, & Boehm, Hans J. (2000). Understanding memory allocation of Scheme programs. *Pages 245–256 of: ACM SIGPLAN International Conference on Functional Programming*.

Stallman, Richard. (1987). *GNU Emacs Manual*. Free Software Foundation Inc., 675 Mass. Ave., Cambridge, MA 02139.

Texas Instruments. (1988). *PC Scheme User's Guide & Language Reference Manual—Student Edition*.

Wadler, Philip. (1987). A critique of Abelson and Sussman, or, why calculating is better than scheming. *SIGPLAN Notices*, **22**(3).

Wand, Mitch. (1986). Finding the source of type errors. *Pages 38–43 of: ACM conference principles of programming languages*.