DrScheme: A Pedagogic Programming Environment for Scheme

Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen

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

Abstract. Teaching introductory computing courses with Scheme elevates the intellectual level of the course and thus makes the subject more appealing to students with scientific interests. Unfortunately, the poor quality of the available programming environments negates many of the pedagogic advantages. To overcome this problem, we have developed DrScheme, a comprehensive programming environment for Scheme. It fully integrates a graphics-enriched editor, a multi-lingual parser that can process a hierarchy of syntactically restrictive variants of Scheme, a functional read-eval-print loop, and an algebraically sensible printer. The environment catches the typical syntactic mistakes of beginners and pinpoints the exact source location of run-time exceptions.
DrScheme also provides an algebraic stepper, a syntax checker and a static debugger. The first reduces Scheme programs, including programs with assignment and control effects, to values (and effects). The tool 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. It also draws arrows on demand that point from bound to binding occurrences of identifiers. The static debugger, roughly speaking, provides a type inference system with explanatory capabilities. Preliminary experience with the environment shows that Rice University students find it helpful and that they greatly prefer it to shell- or Emacs-based systems.

Keywords. Programming Environments, Scheme, Programming, Pedagogy, Algebraic Evaluation, Static Debugging. Teaching programming to beginning students.

1 Problems with Teaching Scheme

Over the past ten years, Scheme [7] has become the most widely used functional programming language in introductory courses. A United States-wide count in 1995 put Scheme in fourth place with 11%, behind Pascal (35%), Ada (17%), and C/C++ (17%) (when grouped together) [32, 34]. SML [31] is the only other functional language listed, at 2%. Scheme's success is primarily due to Abelson and Sussman's seminal book [1] 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 listing 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 [6, 17, 33, 36]:

1. Since the syntax of standard Scheme is extremely liberal, simple notational mistakes produce inexplicable results or incomprehensible error messages.
2. The available implementations do not pinpoint the source location of run-time errors.
3. The Lisp-style output syntax obscures the pedagogically important connection between program execution and algebraic expression evaluation.
4. The hidden imperative nature of Scheme's read-eval-print loop introduces 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. As a result, some students dismiss the entire mostly-functional approach to programming because they mistake these environmental problems for flaws of the underlying functional methodology.

To address these problems we have built DrScheme, a Scheme environment targeted at beginning students. The environment eliminates all problems mentioned above by integrating program editing and evaluation in a semantically consistent manner. DrScheme also contains three additional tools that facilitate teaching functional programming. The first one is a symbolic stepper. It models the execution of functional and imperative Scheme programs as algebraic reductions of programs to answers and their effects. The second tool is a syntax checker. It annotates programs with font and color changes based on the syntactic structure of the program and permits students to explore the lexical structure of their programs with arrows overlaid on the program text. The third auxiliary tool is a static debugger that infers what set of values an expression may produce and how values flow from expressions into variables. It exposes potential safety violations and, upon demand, explains its reasoning by drawing value flow graphs over the program text.

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 but it should be skipped on a first reading if the reader is familiar

Data Description: A *list of numbers* is either:
1. null (the empty list), or
2. (cons *n* *lon*) where *n* is a number and *lon* is a list of numbers.

End

```
(define (length a-lon)
  (cond
    [(null? a-lon) 0]
    [(cons? a-lon) (add1 (length (cdr a-lon)))]))

(define (fahrenheit→celsius d)
  (* 5/9 (− d 32)))
```

Fig. 1. The design of a function

with teaching Scheme. The third section presents DrScheme and explains how it solves the above problems, especially in the context of Rice University's introductory course. The fourth section briefly explains the additional tools. The last three sections discuss related work, present preliminary experiences, and suggest possible uses in the functional programming community.

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 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.

The introduction to functional program design uses a subset of Scheme. It emphasizes program design and the connection between functional programming and secondary school algebra. In particular, the course first argues that a program (fragment) is a function that consumes and produces data, and that the design of programs (or fragments) must therefore be driven by an analysis of these sets of data. The course starts out with the design of list-processing functions, without relying on the fact that lists are a built-in type of data. Students quickly learn to describe such data structures rigorously and to derive functions from these descriptions: see figure 1.

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 from figure 1:

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

Students know this example from their early schooling 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 *value* if v is a value and l a list; (car (cons v l)) = v and (cdr (cons v l)) = l, for every value v and list l. From there, it is easy to illustrate how the sample program works:

```
  (length (cons 41 (cons 23 null)))
= (add1 (length (cdr (cons 41 (cons 23 null)))))
= (add1 (length (cons 23 null)))
= (add1 (add1 (length (cdr (cons 23 null)))))
= (add1 (add1 (length null)))
= (add1 (add1 0))
= 2
```

In short, algebraic calculations completely explain program execution without any references to the underlying hardware or the runtime context of the code.

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, and basic boolean, arithmetical, and list-processing primitives. Then the language is extended with a facility for defining new data constructors, and parallel and recursive local definitions. The final extension covers variable assignment and data mutation. 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 [9, 10, 11].

At the end of the segment on program design, students understand how to construct programs as (collections of) functions and as (object-oriented) 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.

3 The Programming Environment

DrScheme runs under Microsoft Windows 95, Windows NT, MacOS, and the X

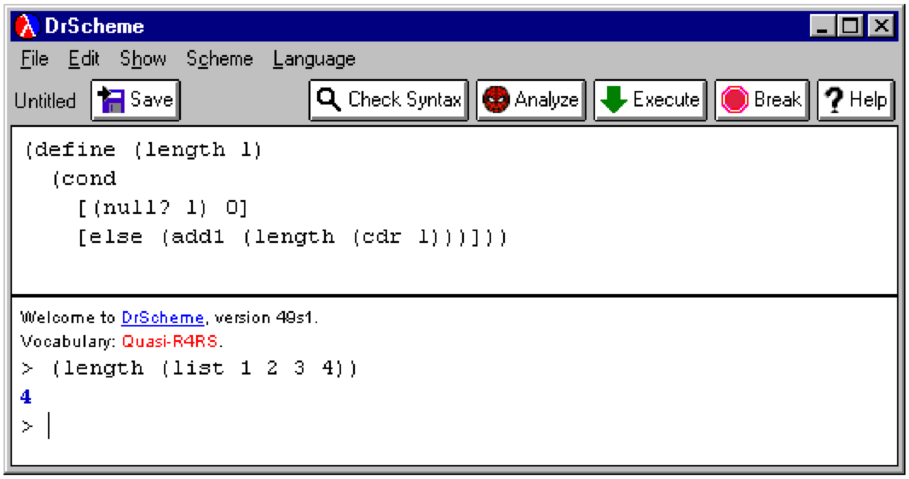

Fig. 2. The DrScheme window (Windows 95/NT version)

Window System. When it starts up, it presents the programmer with a menubar[1] and a window consisting of three pieces: the control panel, the definitions (upper) window, and the interactions (lower) window (see figure 2). The control panel has buttons for important actions, *e.g.*, Save and Help. 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 definitions window, implements a novel read-eval-print loop.

DrScheme's menubar provides five menus: File, Edit, Show, Scheme, and Language. 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 Show menu controls the visibility of the sub-windows. The Scheme menu allows programmers to indent, comment, and uncomment regions of text in the definitions window. The Language menu allows the student to choose which sub-languages of Scheme the syntax checker and evaluator accept.

The control panel contains six buttons: Save, Check Syntax, Analyze, Execute, Break and Help. The Save button saves the definitions from the definitions window as a file. 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 lexical structure (see section 4.2). The Analyze button invokes the static debugger (described in section 4.3) on the contents of the

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

definitions window. The Execute button executes the program in the definitions window. The Break button stops the current computation, and the Help button summons the on-line help facility for DrScheme.

The definitions and interactions windows contain editors that are compatible with typical editors on the various platforms. Under X, the editor has many of the Emacs [35] key bindings. The Windows and MacOS versions have the standard key bindings and menu items for those platforms.

The remainder of this section motivates and describes the new aspects of the core programming environment. In particular, the first subsection describes how DrScheme can gradually support larger and larger subsets of Scheme as students 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 elements 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. For beginning programmers that meaning is often unintended, and as a result they receive inexplicable results or incomprehensible error messages for essentially correct programs.

For example, the author of the program

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

has lapsed into algebraic syntax in the second clause of the **cond**-expression. Since the value of a **cond**-clause is the value of its last expression, this version of *length* always returns 0 as a result, puzzling any programmer, and especially beginning programmers.

Similarly, the program

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

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

While these programs are flawed, their student authors 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 response does not provide any insight into the actual error, the students' learning experience suffers. A good pedagogic programming environment should provide a correct and concise explanations of the students' mistakes.

Students also write programs that use keywords, that they have not yet been taught, as identifiers. It is not the students' fault for using those keywords incorrectly. 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 might write:

```
(define (length l start)
  (cond
   [(null? l) start]
   [else (length (cdr l) (add1 begin))]))
```

This program is buggy; it has an unbound identifier **begin**. But, it generates a strange syntax error: "`compile: illegal use of a syntactic form name in: begin`". The student cannot understand that they have uncovered a new part of the programming language.

Eager students also attempt to use features that they have not yet seen in class. For example, they might try to use local definitions before scope is described in class. Many students 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 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 [26]. The student can choose the appropriate language level 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 mirrors the student's language level. A language consists of several independent settings, which are normally hidden from the student. Clicking on the Show Details button enlarges the dialog, bringing a panel with all of the language settings into view. Figure 3 shows the enlarged dialog.

The description of a language level consist of three parts: input syntax, safety properties, and output syntax. The input syntax is specified through the Case

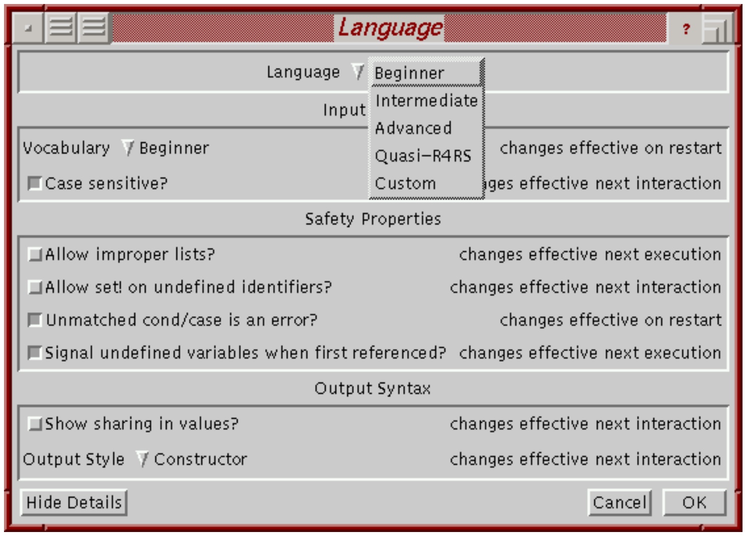

Fig. 3. DrScheme's language configuration dialog box (X version)

Sensitive? check box and a *vocabulary*, a set of syntactic forms. The four predefined vocabularies are: *Beginner*, *Intermediate*, *Advanced*, and *Quasi-R⁴RS*. Each vocabulary corresponds to a stage in Rice University's introductory course:

Beginner includes definitions, conditionals and a large class of functional primitives.

Intermediate extends *Beginner* with structure definitions and the local binding constructs: **local**, **let**, **let**$*$, and **letrec**.

Advanced adds support for variable assignments, data mutations, as well as implicit and explicit sequencing.

Quasi-R⁴RS subsumes the Scheme language [7, 12, 15].

The first three blocks of the table in figure 4 specify the exact syntactic content of the various language levels. The last block describes other properties of the four language levels.

The safety properties of DrScheme allow the student to choose between conformance with R^4RS and more sensible error reporting. They can be specified with four check boxes:

- Allow improper lists?,
- Allow set! on undefined identifiers?,
- Unmatched cond/case is an error?, and
- Signal undefined variables when first referenced?.

	Beginner	Intermediate	Advanced	Quasi-R^4RS
define	√	√	√	√
lambda	√	√	√	√
cond, **if**	√	√	√	√
quote'd symbols	√	√	√	√
define-struct		√	√	√
local, **let**, **let**∗, **letrec**		√	√	√
delay, *force*		√	√	√
set!			√	√
begin, **begin0**			√	√
implicit **begin**			√	√
named let, **recur**			√	√
quote'd lists			√	√
quasiquote			√	√
unquote			√	√
call/cc,*let/cc*			√	√
when, **unless**			√	√
if without else			√	√
scheme primitives	√	√	√	√
case sensitive	√	√	√	√
sharing in values			√	
Allow improper lists?				√
Allow **set!** on undefined identifiers				√
Unmatched cond/case is an error?				√

Fig. 4. Language Level Quick Reference

When the Allow improper lists? is unchecked, cons can only be used to construct lists; its second argument must always be either null or a cons cell. The check box Allow set! on undefined identifiers? controls whether **set!** creates a new name at the top-level or signals an error for unbound identifiers. If Unmatched cond/case is an error? is on, the implicit **else** clause in **cond** and **case** expressions signal 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 language's behavior when evaluating potentially circular definitions. Scheme evaluates recursive binding expressions by initializing all identifiers being bound to a special tag value, and then evaluating each definition and rebinding 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 initial value flows into a primitive function.

The output syntax is determined by the Show sharing in values? check box and the Printing choice. When the Show sharing in values? is on, all sharing within data structures is displayed in the output. The Printing choice provides three alternatives: constructor style, quasiquote style or R^4RS style. Under constructor style, the list containing the numbers 1, 2, and 3 prints out as (list 1 2 3). Because it mirrors the input syntax for values, constructor style output is useful for general programs and mandatory for pedagogic programming (see section 4.1). In contrast, quasiquote style is a compromise between the constructor style output and the standard Scheme output style [7]. Like the former, the quasiquote-style output matches quasiquote input syntax. But, by dropping the leading quasiquote, the output can also be used as program text, just like the output of the standard Scheme printer.

3.2 Interactive Evaluation

Many functional languages support the interactive evaluation of expressions via a read-eval-print loop (REPL). Abstractly, a REPL allows students to both construct new programs and evaluate expressions in the context of a program's definitions. A typical REPL implements those operations by prompting the students to input programs fragments. The fragments are then evaluated, and their results are printed.

Interactivity is primarily used for program exploration, the process of evaluating expressions in the context of a program to determine its behavior. 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. They also patch their programs with the REPL in order to test potential improvements or bug fixes by rebinding names at the top-level.

While interactive REPLs are superior to batch execution for program development, they can introduce confusing bugs into programs. Since they allow ad-hoc program construction, REPLs cause problems for the beginner 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
                                     [(null? l) n]
                                     [else (length-helper (cdr l) (add1 n))]))
```

into a version that uses local definitions:

```
(define (length l)
  (letrec ([helper (lambda (l n)
                     (cond
                      [(null? l) n]
                      [else (length-helper (cdr 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 students program with higher-order functions. Consider the program:

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

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

A student will quickly discover the bug by experimenting with *add11*, replace the primitive $*$ with $+$ and reevaluate the definition of *make-adder*. Unfortunately, the REPL no longer reflects the program, because *add11* still refers to the old value of *make-adder*. Consequently *add11* will still exhibit the bug, confusing the student. The problem is exacerbated when higher-order functions are combined with state.

Experienced functional programmers have learned to avoid this problem by using their REPL in a batch-oriented fashion. They exit, restart 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 student is ready to test a program, a click on the Execute button submits the program to the interactions window. When the student 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 student evaluates test expressions in the REPL. After discovering an error, the student edits the definitions and executes the program again to test the new definitions. In short, after every change to the program, the student starts the program afresh, which eliminates the problems caused by traditional REPLs.

3.3 Error Reporting

A pedagogic programming environment must provide good run-time error reporting; it is crucial to a student's learning experience. The programming environment must 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 [33], Scheme compilers and interpreters only implement a simplistic read-eval-print loop. If this REPL is executed in a plain command shell, it is impossible to relate errors to source locations in general. The historical solution is to execute the REPL in an Emacs buffer. This

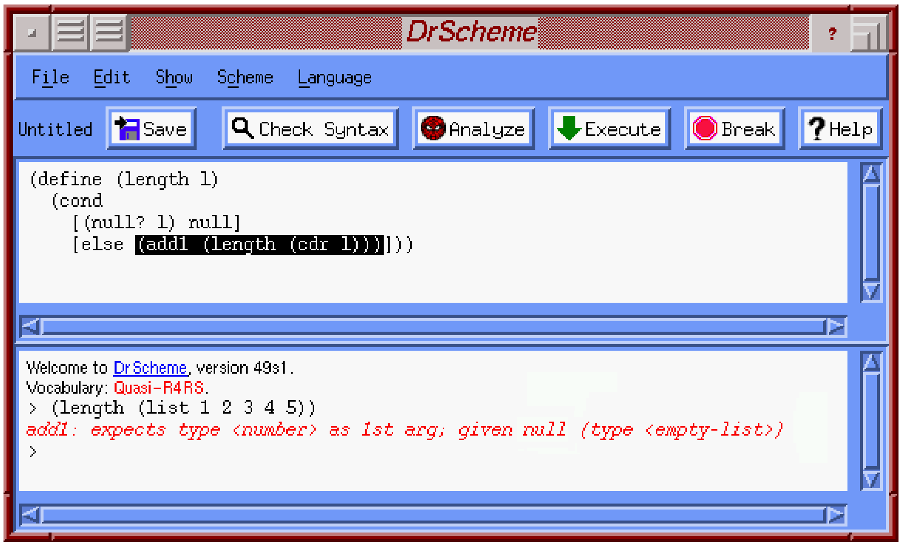

Fig. 5. DrScheme, with a run-time error highlighted (X Motif version)

solution, however, does not truly integrate the REPL and its editing environment, so that the graphical capabilities of modern displays remain unexploited.

Second, Scheme's macro facility [22, 23] tremendously complicates the mapping from a run-time error to its source location [8]. Since Scheme's macro language allows arbitrary mappings on 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.

DrScheme overcomes all three problems. 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 [26]. This correlation allows DrScheme to report the source location of run-time errors.

Consider the example in figure 5. The student has written an erroneous version of *length*. When it is applied to (list 1 2 3 4 5), it recurs down the list, and is applied to null. The function then returns null, which flows into the primitive $+$, generating a run-time error. Then, the run-time error is caught by DrScheme, and the source location of the misapplied primitive is highlighted. With a little effort, any beginning student can now fix the bug.

4 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: Printing and The Stepper

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 reductions that transform an expression to a value in a context of definitions.

Unfortunately, traditional Scheme implementations do not reinforce that connection [37]. They typically use one syntax for values as input and a different syntax for values as output. For example the expression:

```
(map add1 (list 2 3 4))
```

evaluates to

```
(3 4 5)
```

which gives students the mistaken impression that the original expression has evaluated to an application of 3 to 4 and 5.

DrScheme uses an output syntax for values called constructor syntax that matches their input syntax. Constructor syntax treats the primitives cons,[2] vector, box, *etc.*, as constructors. Thus, when a value is printed the initial constructor shows which subset contains the value.

So, in the the above example, DrScheme prints the value of:

```
(map add1 (list 2 3 4))
```

as

```
(list 3 4 5)
```

DrScheme's printer produces the same syntax for the values that Scheme's reduction semantics produces.

More importantly, DrScheme also includes a tool that enables students to reduce a program to a value step by step. This symbolic stepper is based on Felleisen and other's work on reduction semantics for Scheme- and ML-like languages [9, 10, 11] and 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 by choosing Tools|Stepper.[3] By default, the stepper shows every reduction step of a program evaluation. While this is useful

[2] list is used as shorthand for consecutive conses ending in null.

[3] The stepper is not available in DrScheme version 49.

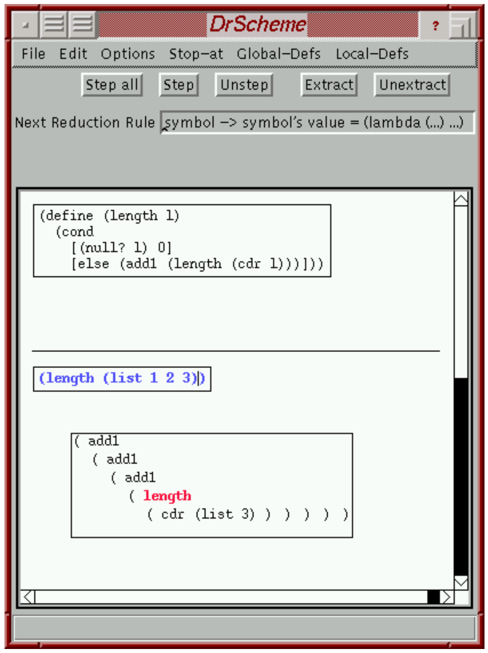

Fig. 6. The stepper (X version)

for a complete novice, a full reduction sequence contains too much information in general. Hence the stepper permits the student to choose which reduction steps are shown or which sub-expressions the stepper is to focus on. At each step, the student can change these controls to view a more detailed reduction sequence.

The stepper window always displays the original program expression and definitions together with the current state of the evaluation as a program (see figure 6). For each step, the stepper explains its action before it proceeds. In the figure, it indicates that it is about to lookup *length* and replace it with its value. If reduction rules for imperative language facilities require memory allocations, they are introduced as global definitions. To focus students' attention, these auxiliary definitions are hidden until mutated. Students may also choose to view them at intermediate stops.

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.

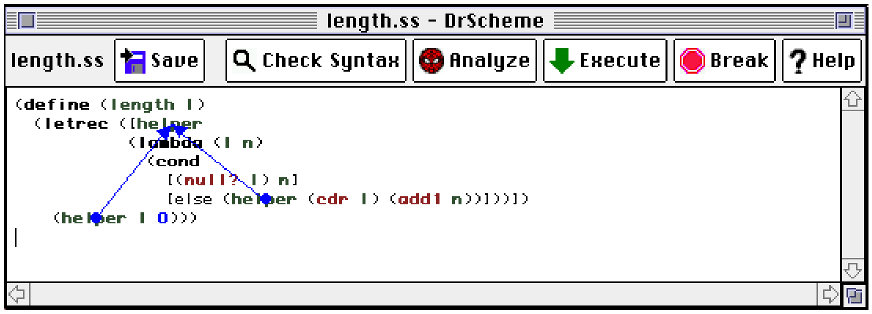

Fig. 7. DrScheme's syntax checker (MacOS version)

4.2 Syntax Checking

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 displays arrows that point from bound identifiers to their binding occurrence, and from binding identifiers to all of the their bound occurrences, see figure 7. Since the checker processes the lexical structure the program, it can also be used to α-rename bound identifiers. If a student checks a syntactically incorrect program, the first incorrect portion of the text is highlighted, and an error message is printed.

4.3 Static Debugging

The most advanced DrScheme tool is MrSpidey, a static debugger [5, 14], which subsumes the syntax checker, but is computationally far more expensive. It analyzes the current program, using a new form of set-based analysis [13, 20], for potential safety violations. That is, the tool infers constraints on the potential value flow in the Scheme program, similar to the equational constraints of a Hindley-Milner type checker, and builds a value flow graph for the program. For each primitive, the static debugger determines whether or not the potential argument values are legal 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, and those that the static debugger can prove are always correctly applied are highlighted in green. In print and on monochrome displays, all primitives are boldfaced, and red primitives are underlined. On demand, the static debugger annotates each program point with:

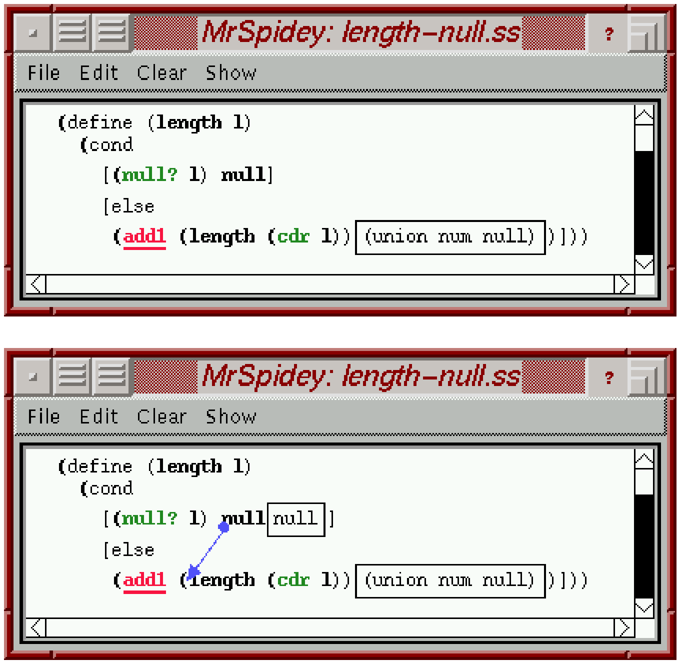

Fig. 8. MrSpidey: The static debugger (X version)

– its inferred value set, and
– arrows describing the inferred flow of values that produced the value set.

Using these mark-ups, a student can browse the invariants that approximate the dynamic behavior of a program to identify which operations are not provably safe. The student can evaluate the coverage of the set of program test cases, and tune the program to improve the accuracy of the generated invariants.

For an illustration, consider the program in the top of figure 8. 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 colored red, which indicates that the static debugger cannot prove that the argument will always be a number. The student, using a pop-up menu, can now display the value set of `add1`'s argument. The static debugger responds by inserting a box that contains the value set for the argument to the right of the argument's text as in the top of figure 8. This value set contains `null`, which is why the static debugger concluded that `add1` may be misapplied. To see how `null` can flow into the argument of `add1`, the static debugger can also annotate the program with an arrow pointing from the argument of `add1` to the source of `null`, as in the bottom of figure 8. Since a recursive application of *length* can trigger the flow of `null` into `add1` (as discussed in section 3.3) the static debugger has given the student the information needed to uncover the bug.

5 Related Work

DrScheme seamlessly integrates a number of ideas that are important for teaching courses 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.

In lieu of source locations for run-time errors, 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 tinker-until-it-works approach of traditional imperative 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 SML [3, 19]. Unfortunately this is useless for beginners, who mostly work with the core language. Also, CAML [28], MLWorks [18], and SML/NJ [2] 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 [4, 29, 30] for imperative programming languages like C++ incorporate a good portion of the functionality found in DrScheme. Their editors use on-line real-time syntax coloring algorithms, the run-time environments trap segmentation faults and highlight their source location, which is much less useful than catching safety violations but still superior to stand-alone REPLs of Scheme and other functional languages. Indeed, their debuggers serve as primitive REPLs, though with much less flexibility than the REPLs that come with Scheme or SML. None of them, however, provides language levels, full-fledged algebraic printers, steppers, or static debuggers, which we have found to be extremely useful for teaching purposes.

6 User Experiences

All four of the professors who teach the introductory computer science course at Rice University (three of whom are independent of the development group) use DrScheme for the course. DrScheme is used in the course on a weekly basis for both the tutorials and the homework assignments. Also, the programming languages course at Rice uses DrScheme for the weekly programming assignments.

Unfortunately, we do not have any quantitative data for comparing DrScheme to other programming environments, since all sections of the introductory course and the programming languages course use DrScheme. Still, students who know both Emacs-based Scheme environments and DrScheme typically prefer the latter.

We have also received enthusiastic reports from professors and teachers in several countries of North America and Europe who use DrScheme in their classes. Our announcement mailing list consists of nearly one hundred people in academia and industry who use DrScheme and its application suite. We are also aware of several commercial efforts that are incorporating portions of our suite into their products.

Most of the complaints about DrScheme fall into two categories. First, running DrScheme requires at least 24 megabytes of memory on most operating systems and machines. If a machine has less than the required minimum, DrScheme is extremely slow and may even crash. Second, since the development team only uses a small subset of the supported platforms, small platform-dependent errors can go undetected for some time. We expect to eliminate both problems with future research.

7 Conclusion

The poor quality of programming environments for functional languages distracts students from the study of computer science principles. The construction of DrScheme overcomes these problems for Scheme.

Many aspects of DrScheme apply to functional languages other than Scheme. Any functional language becomes more accessible to the beginner in an environment that provides several well-chosen language levels, a functional read-eval-print loop, accurate source highlighting for run-time errors, 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 the static debugger. In general, we hope that DrScheme's success with students and teachers around the world inspires others to build programming environments for functional languages based on pedagogic considerations.

DrScheme is available on the web at `http://www.cs.rice.edu/CS/PLT/`.

Acknowledgments

We greatly appreciate valuable feedback from R. Cartwright and D.P. Friedman on early drafts of this paper.

References

1. Abelson, H., G. J. Sussman and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.
2. AT&T Bell Labratories. *Standard ML of New Jersey*, 1993.

3. Blume, M. Standard ML of New Jersey compilation manager. Manual accompanying SML/NJ software, 1995.
4. Borland. *Borland C++*, 1987, 1994.
5. Bourdoncle, F. Abstract debugging of higher-order imperative languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 46–55, 1993.
6. Cadence Research Systems. *Chez* Scheme Reference Manual, 1994.
7. Clinger, W. and J. Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.
8. Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
9. Felleisen, M. An extended λ-calculus for Scheme. In *ACM Symposium on Lisp and Functional Programming*, pages 72–84, 1988.
10. Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.
11. Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. In *Proceedings of Theoretical Computer Science*, pages 235–271, 1992.
12. Findler, R. B. and M. Flatt. PLT MrEd: Graphical toolbox manual. Technical Report TR97-279, Rice University, 1997.
13. Flanagan, C. and M. Felleisen. Componential set-based analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.
14. Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.
15. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
16. Francez, N., S. Goldenberg, R. Y. Pinter, M. Tiomkin and S. Tsur. An environment for logic programming. *SIGPLAN Notices*, 20(7):179–190, July 1985.
17. Hanson, C., The MIT Scheme Team and A Cast of Thousands. *MIT Scheme Reference*, 1993.
18. Harlequin Inc. *MLWorks*, 1996.
19. Harper, R., P. Lee, F. Pfenning and E. Rollins. Incremental recompilation for Standard ML of New Jersey. Technical Report CMU-CS-94-116, Carnegie Mellon University, 1994.
20. Heintze, N. Set based analysis of ML programs. In *ACM Symposium on Lisp and Functional Programming*, 1994.
21. Hsiang, J. and M. Srivas. A Prolog environment. Technical Report 84-074, State University of New York at Stony Brook, Stony Brook, New York, July 1984.
22. Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
23. Kohlbecker Jr, E. E. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, August 1986.
24. Komorowski, H. J. and S. Omori. A model and an implementation of a logic programming environment. *SIGPLAN Notices*, 20(7):191–198, July 1985.
25. Koschmann, T. and M. W. Evens. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 5:36–42, July 1988.
26. Krishnamurthi, S. Zodiac: A framework for building interactive programming tools. Technical Report TR96-262, Rice University, 1996.

27. Lane, A. Turbo Prolog revisited. *BYTE*, 13(10):209–212, October 1988.
28. Leroy, X. *The Objective Caml system, documentation and user's guide*, 1997.
29. Metrowerks. *CodeWarrior*, 1993–1996.
30. Microsoft. *Microsoft Developer Studio*, 1995.
31. Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
32. Reid, R. J. First-course language for computer science majors. *Posting to comp.edu*, October 1995.
33. *EdScheme: A Modern Lisp*, 1991.
34. Schemer's Inc. and Terry Kaufman. Scheme in colleges and high schools. Available on the web.
    URL: `http://www.schemers.com/schools.html`.
35. Stallman, R. *GNU Emacs Manual.* Free Software Foundation Inc., 675 Mass. Ave., Cambridge, MA 02139, 1987.
36. Texas Instruments. *PC Scheme User's Guide & Language Reference Manual—Student Edition*, 1988.
37. Wadler, P. A critique of Abelson and Sussman, or, why calculating is better than scheming. *SIGPLAN Notices*, 22(3), March 1987.

This article was processed using the LaTeX macro package with LLNCS style