Debugging Macros

Ryan Culpepper
Northeastern University
ryanc@ccs.neu.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

Over the past two decades, Scheme macros have evolved into a powerful API for the compiler front-end. Like Lisp macros, their predecessors, Scheme macros expand source programs into a small core language; unlike Lisp systems, Scheme macro expanders preserve many desirable properties, including lexical scope and source location. Using such macros, Scheme programmers now routinely develop the ultimate abstraction: embedded domain-specific programming languages.

Unfortunately, Scheme programming environments provide little support for macro development. This lack makes it difficult for programmers to debug their macros and for novices to study the behavior of macros. In response, we have developed a stepping debugger specialized to the concerns of macro expansion. It presents the macro expansion process as a linear rewriting sequence of annotated terms; it graphically illustrates the binding structure of the program as expansion reveals it; and it adapts to the programmer's level of abstraction, hiding details of syntactic forms that the programmer considers built-in.

1. The Power of Macros

Modern programming languages support a variety of abstraction mechanisms: higher-order functions or delegates, class systems, expressive type systems, module systems, and many more. With these, programmers can develop code for reuse; establish single points of control for a piece of functionality; decouple distinct components and work on them separately. As Paul Hudak [21] has argued, however, "the ultimate abstraction is a ... language." The ideal programming language should allow programmers to develop, encapsulate, and reuse entire sub-languages.

The Lisp and Scheme family of languages empower programmers to do just that. Through macros, they offer the programmer the ability to define new forms of expressions and definitions with custom behavior. These *syntactic abstractions* may introduce new binding positions, perform analyses on their subterms, and change the context of their subexpressions. Because macros provide a uniform mechanism for extending the definition and expression forms of the language, programmers can mix language extensions freely. Because macros are part of the language, they respect the language's other abstraction mechanisms and avoid many of the problems associated with preprocessors and program generators. As some Scheme implementers have put it, macros have become a true API to the front-end of the compiler.

If attached to an expressive language [9] macros suffice to implement many general-purpose abstraction mechanisms as libraries. For example, programmers have used macros to extend Scheme with constructs for pattern matching [36], relations in the spirit of Prolog [8, 30, 18, 23], extensible looping constructs [7, 29], class systems [27, 1, 17] and component systems [34, 14, 5], among others. In addition, programmers have also used macros to handle metaprogramming tasks traditionally implemented outside the language using preprocessors or special compilers: Owens et al. [26] have added a parser generator library to Scheme; Sarkar et al. [28] have created an infrastructure for expressing nano-compiler passes; and Herman and Meunier [20] have used macros to improve the static analysis of Scheme programs. As a result, implementations of Scheme such as PLT Scheme [13] have a core of a dozen or so syntactic constructs but appear to implement a language the size of Common Lisp [32].

To support these increasingly ambitious applications, macro systems had to evolve, too. True syntactic abstractions are indistinguishable from features directly supported by the implementation's compiler or interpreter. In Lisp systems, macros are compile-time functions over program fragments, usually plain S-expressions. Unfortunately, these simplistic macros don't really define abstractions. For example, because Lisp macros manipulate variables by name alone, references they introduce can be captured by bindings in the context of the macro's use. This tangling of scopes reveals implementation details instead of encapsulating them. In response, researchers developed the notions of macro hygiene [24], referential transparency [3, 6], and phase separation [12] and implemented macro systems guaranteeing those properties. Now macros manipulate program representations that carry information about lexical scope; affecting scope takes deliberate effort and becomes a part of the macro's specification.

Given the power of macros and the degree to which Scheme programmers benefit from them, it is astonishing that no Scheme implementation offers solid debugging support for macros. The result is that programmers routinely ask on Scheme mailing lists about unforeseen effects and subtle errors arising from a faulty or incomplete understanding of macros. While experts always love to come to their aid, these questions demonstrate the need for software tools that help programmers explore their macro programs.

In this paper, we present the first macro stepper for Scheme, hoping to help programmers develop powerful abstractions and explore macros. In the next section we discuss the properties of macro programming that inform the design of our debugger, and in the subsequent sections we discuss the implementation of the stepper.

Note: A preliminary version of this work appeared at the Scheme workshop, which publishes its proceedings as a technical report.

2. Design Considerations for a Macro Debugger

A debugger must be tailored to the language it debugs. Debugging a logic program is fundamentally different from debugging assembly code. Most importantly, a debugger should reflect the programmer's mental model of the language. This model determines what information the debugger displays as well as how the programmer accesses it. It determines whether the programmer inspects unification variable bindings or memory locations and whether the programmer explores a tree-shaped logical derivation or steps through statements.

Following this analysis, the design of our debugger for macros is based on three principles of Scheme macros:

1. Macros are rewriting rules.
2. Macros respect lexical scoping.
3. Macros define layered abstractions.

The rest of this section elaborates these three principles of Scheme macros and explains how they influence the design of our debugger.

2.1 Macros Are Rewriting Rules

Intuitively, macro definitions are rewriting rules, and macro expansion is the process of applying these rules to the program until all of the macros have been eliminated. The expander does not rewrite terms freely. Rather, the base language provides a set of primitive forms, and expansion occurs only in primitive contexts. For example, `lambda` is a primitive form, and expansion occurs in the body of a `lambda` expression, but not in its formal parameter list. The final program contains only the primitive forms of the language.

The notation programmers use to express the rewriting rules varies among different Lisp and Scheme implementations. Lisp macros are generally written as procedures using Lisp's quasiquotation facility. The current Scheme report (R^5RS) [22] defines a pattern-based macro facility called `syntax-rules`. Many Scheme implementations also allow programmers to write macros using arbitrary transformer procedures; drafts of the next Scheme report [31] describe such a facility called `syntax-case` [6].

A macro definition in R^5RS Scheme has the following form:

```
(define-syntax macro
  (syntax-rules (literal ...)
    [pattern_1 template_1]
    ...
    [pattern_n template_n]))
```

This definition directs the macro expander to replace any occurrences of terms matching one of the listed patterns with the corresponding template, substituting the occurrence's subterms for the pattern variables. The `literals` determine which identifiers in the pattern are matched for equality rather than treated as pattern variables.

For example, here is a definition for a macro that helps a programmer protect shared resources. By using such a macro, the programmer ensures that the enclosed expression is only executed in the proper context: with the lock acquired. In addition, the programmer can be sure that the calls to acquire and release the lock are balanced.

```
;; (with-lock expr) acquires a lock, evaluates
;; the expression, and releases the lock.
;; The resulting value is the void value.
(define-syntax with-lock
  (syntax-rules ()
    [(with-lock body)
     (begin (acquire-the-lock)
            body
            (release-the-lock))]))
```

When a programmer wraps an expression in a `with-lock` form, the macro expander inserts the calls to acquire and release the lock.[1]

Given the macro definition above, macro expansion consists of scanning through a program searching for occurrences of `with-lock` to replace. For example, the following program contains one occurrence of the macro:

```
(define (print-items header items)
  (with-lock
   (begin (print header)
          (for-each print items))))
```

The expander rewrites the program to the following:

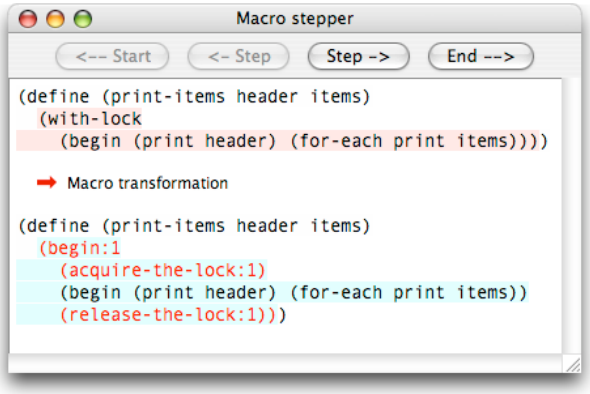
```
(define (print-items header items)
  (begin (acquire-the-lock)
         (begin (print header)
                (for-each print items))
         (release-the-lock)))
```

This term contains no occurrences of the macro: macro expansion is complete.

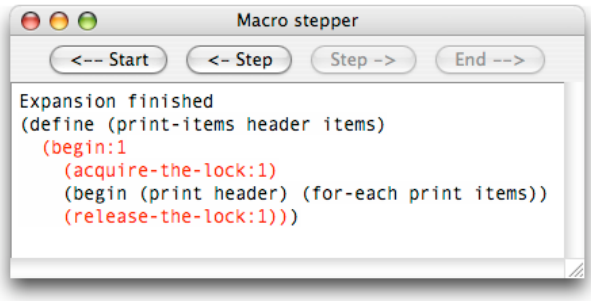
Our macro stepper shows the expansion process to the programmer as a sequence of rewriting steps. Each step consists of two program texts that differ by the application of a single macro rewriting rule. The stepper highlights the site of the rule's application.

The tool's graphical user interface permits programmers to go back and forth in an expansion and also to skip to the beginning or end of the rewriting sequence. The ideas for this interface have been borrowed from the run-time stepper for PLT Scheme [2, 25, 33].

Here is the program from above in the macro stepper:



The stepper shows the original program and the result of applying the `with-lock` rewriting rule. If the programmer navigates forward, the stepper indicates that expansion is complete and shows the final program:



[1] A more robust version would take exceptions and continuations into account, but this paper is not about Scheme control mechanisms.

The final program displayed by the stepper is *almost* what one would expect from naively applying the rewrite rules, but some of the identifiers have a ":1" suffix. These come from the macro system's lexical scoping mechanism. In this particular case, the marks do not affect the meaning of the program. The next section explains these marks in more detail.

2.2 Macros Respect Lexical Scoping

Scheme requires macro expansion to be hygienic—to respect lexical scoping according to the following two principles [3, 24]:

1. Free identifiers introduced by a macro are bound by the binding occurrence apparent at the site of the macro's definition.
2. Binding occurrences of identifiers introduced by the macro (not taken from the macro's arguments) bind only other identifiers introduced by the same macro transformation step.

The gist of these requirements is that macros act "like closures" at compile time. More generally speaking, macro definitions and macro uses are properly separated by an abstraction boundary. The meaning of a macro—the meaning of the variables and keywords in the macro's right-hand sides—can be determined from the macro definition's context alone; it does not depend on the context the macro is used in. Furthermore, if the macro creates a lexical binding for a variable whose name is given directly in the body of the macro, that binding does not interfere with expressions that the macro receives as arguments—unless the programmer explicitly programs such a violation using the hygiene-breaking features of a system such as the `syntax-case` system.

Thinking of macro expansion in terms of substitution provides additional insight into the problem and its solution. There are two conceptual occurrences of substitution and two kinds of capture to avoid. The first substitution consists of bringing the macro body to the use site; this substitution must not allow bindings in the context of the use site to capture names present in the macro body (condition 1). The second consists of substituting the macro's arguments into the body; names in the macro's arguments must avoid capture by bindings in the macro body (condition 2), even though the latter bindings are not immediately apparent in the macro's result.

To see how bindings might not be apparent, consider the following definition:

```
(define-syntax (munge stx)
  (syntax-rules ()
    [(munge e)
     (mangle (x) e)]))
```

It is not clear what role the template's occurrence of `x` plays. If `mangle` acts like `lambda`, then `x` would be a binding occurrence, and by the hygiene principle it must not interfere with any free occurrences of `x` in `e`. On the other hand, if `mangle` acts like `begin`, then the occurrence of `x` is a reference to an existing variable or macro defined in the context of `munge`'s definition. The second hygiene principle guarantees that if `x` is a reference, it refers to the `x` in scope where the macro was *defined*. Without performing further expansion steps, the macro expander cannot tell. Thus it must delay its resolution of `x` to its meaning until the use of `mangle` has been replaced with core syntax. The program representation must contain enough information to allow for both possibilities.

In short, macro expansion discovers the program's binding structure gradually. It does not know the precise structure until the program is completely expanded.

Hygienic macro systems annotate their program representations with timestamps. The timestamp indicates which macro rewriting step introduced an identifier. To illustrate, here is a macro that introduces a binding for a temporary variable:

```
;; (myor e1 ... eN) evaluates its subexpressions
;; in order until one of them returns a non-false
;; value. The result is that non-false value, or
;; false if all evaluated to false.
(define-syntax myor
  (syntax-rules ()
    [(myor e) e]
    [(myor e1 e ...)
     (let ([r e1])
       (if r r (myor e ...)))]))
```

and here is a program that uses it:

```
(define (nonzero? r)
  (myor (negative? r) (positive? r)))
```

If macro expansion followed the naive rewriting process, the temporary variable `r` introduced by the macro would interfere with the reference to the formal parameter named `r`.

```
;; NAIVE, CAUSES CAPTURE
(define (nonzero? r)
  (let ([r (negative? r)])
    (if r r (positive? r))))
```

Instead, the macro expander marks every macro-introduced identifier with a timestamp. Using our notation for timestamps, the Scheme expander gives us this term:

```
(define (nonzero? r)
  (let:1 ([r:1 (negative? r)])
    (if:1 r:1 r:1 (myor:1 (positive? r)))))
```

The macro expander marks introduced identifiers indiscriminately, since it cannot predict which identifiers eventually become variable bindings. When a marked identifier does not occur inside of a binding of the same marked identifier, such as `let:1`, the mark is ignored: hence the `let:1` above means the same thing as `let`. In contrast, the two references to `r:1` occur inside of a binding for `r:1`, so they refer to that binding. Finally, the occurrence of the unmarked `r` is *not* captured by the binding of `r:1`; it refers to the formal parameter of `nonzero?`.

In the example from the last section, then, the marks on `begin:1`, `acquire-the-lock:1`, and `release-the-lock:1` didn't affect the meanings of those identifiers.

Marks are a crucial part of Scheme's macro expansion process. Our macro debugger visually displays this scope information at every step. The display indicates with numeric suffixes[2] from which macro expansion step every subterm originated.

Here is what the macro stepper displays for the expansion of `nonzero?`:

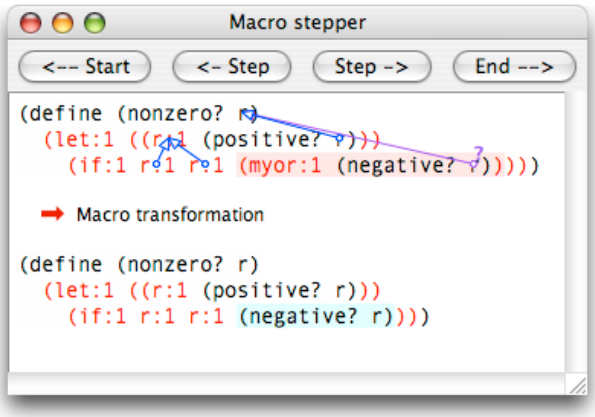
[2] The stepper also uses different text colors to display this information.

The introduced binding has a mark, and the mark matches the introduced references.

Recall that macro expansion gradually discovers the binding structure of the program. The discovery process occurs in two parts. When the macro expander sees a primitive binding form, it discovers the *bindings* of its bound variables. However, it does not yet know what matching occurrences of those identifiers constitute *references* to those bindings. Other intervening macros may introduce bindings of the same name. Only when the macro expander encounters a variable reference does the binding become definite.

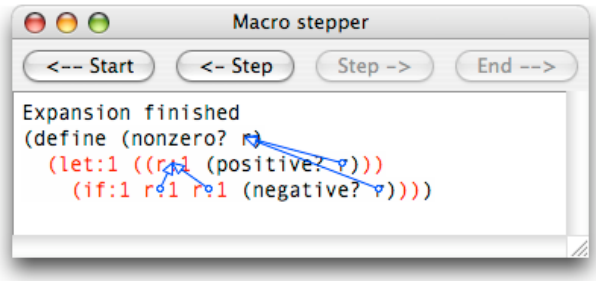
The macro stepper displays its knowledge of the program's binding structure through binding arrows, inspired by the arrows DrScheme's Check Syntax tool [10]. When the macro expander uncovers a binding, the macro stepper draws a "tentative" binding arrow from the binding occurrence to every identifier that may possibly be bound by it. The stepper annotates tentative binding arrows with a "?" symbol at the bound end. When the macro expander resolves a variable to a definite binding, the arrow becomes "definite" and the question mark disappears.

Here is the next step in the expansion with some of the binding arrows shown:



Note that the references to the marked `r:1` variables definitely belong to the binding of `r:1`, and the first reference to the unmarked `r` is definitely bound by the procedure's formal parameter. The final occurrence of `r` is not definite because macro expansion is not finished with its enclosing term.

Stepping forward once more reveals the fully expanded program. Since all of the references have been uncovered by the macro expander, all of the binding arrows are definite:



Binding information is important in debugging macros. Because macro expansion often manipulates the lexical scope of a program in a subtle manner, the shape of an intermediate term alone may not reveal an error. The program seems right, but variables don't get bound, or get bound in the wrong place. In these cases, the stepper's visual presentation of lexical binding is critical.

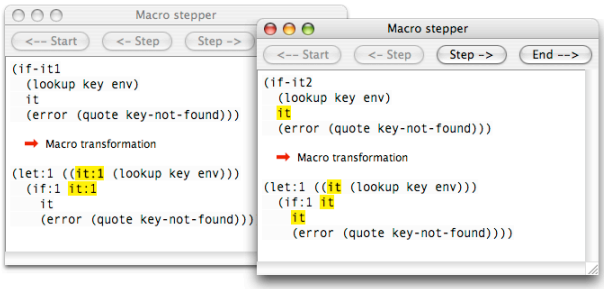
To illustrate this point, consider the creation of an `if-it` macro, a variant of `if` that binds the variable named `it` to the result of the test expression for the two branches:

```
(define-syntax (if-it1 stx)
  (syntax-case stx ()
    [(if-it1 test then else)
     #'(let ([it test]) (if it then else))]))

(define-syntax (if-it2 stx)
  (syntax-case stx ()
    [(if-it2 test then else)
     (with-syntax
         ([it (datum->syntax-object #'if-it2 'it)])
       #'(let ([it test]) (if it then else))))]))
```

These two attempts at the macro are written in the `syntax-case` macro system [6, 31]. Their right-hand sides can contain Scheme code to perform the macro transformation, including syntax-manipulating procedures such as `datum->syntax-object` and binding forms such as `with-syntax`. These facilities offer the programmer explicit control over lexical scoping, necessary when "breaking hygiene" as the programmer intends with this macro.

One of these macro definitions achieves its purpose, and one does not. A Scheme novice taking a first step beyond basic macro programming cannot tell which one is correct, or perhaps can guess but cannot explain *why*. A Scheme programmer equipped with the macro stepper can tell easily:



On the right, all four of the `it` identifiers have the same marks, but on the left they do not.

The macro stepper also explains `datum->syntax-object` as a function that transfers the marks from its first argument to its second. That is, the occurrence of `it` in `if-it2`'s expansion is unmarked because the `if-it2` keyword was unmarked. Thus the macro stepper helps reinforce and enhance the programmer's mental model of hygienic macro expansion.

Note: PLT Scheme attaches additional properties to tokens and preserves them across expansion. The programmer can view those properties on demand.

2.3 Macros Define Layered Abstractions

Preprocessors and compilers translate from a source language to a target language. So do macros, but in a system with macros, there are many source languages and many target languages, depending on one's perspective.

For example, PLT Scheme's new trait library [15] is implemented as a macro that translates trait declarations to mixin declarations [16]. Mixins (classes parameterized over their superclass) have been part of PLT Scheme's standard library for a long time—and they are implemented as macros in terms of "basic" Scheme. This "basic" language, however, is itself implemented using macros atop a smaller language kernel.

Such layers of linguistic abstraction are common in languages that support macros, and they demand special support from debug-

gers. After all, a debugger for a high-level language should not bother the programmer with low-level details. In a program with layered abstractions, however, the line between high-level and low-level is fluid. It varies from one debugging session to another. The debugger must be able to adjust accordingly.

A macro debugger that operates at too low a level of abstraction burdens the programmer with extra rewriting steps that have no bearing on the programmer's problem. In addition, by the time the stepper reaches the term of interest, the context has been expanded to core syntax. Familiar landmarks may have been transformed beyond recognition. Naturally, this prevents the programmer from understanding the macro as a linguistic abstraction in the original program. For the `class` example, when the expander is about to elaborate the body of a method, the `class` keyword is no longer visible; field and access control declarations have been compiled away; and the definition of the method no longer has its original shape. In such a situation, the programmer cannot see the forest for all the trees.

The macro debugger overcomes this problem with *macro hiding*. Specifically, the debugger implements a policy that determines which macros the debugger considers *opaque*; the programmer can modify this policy as needed. The macro debugger does not show the expansion of any opaque macros, but if an occurrence of a macro has subexpressions, it does display the expansions of those subexpressions in the context of the original macro form. That is, hiding a macro effectively removes its rewriting rules and adds expansion contexts, as if it were truly a primitive of the language.

In doing this, the debugger presents steps that actually never happen and it presents terms that the expander actually never produces. Still, these intermediate terms are plausible and instructive, and for well-behaved macros[3] macro hiding preserves the meaning of the program.

Consider the `if-it2` macro from the previous subsection. After testing the macro itself, the programmer wishes to employ it in the context of a larger program, an evaluator for arithmetic expressions:

```
(define (aeval expr)
  (match expr
    [(cons op args)
     (apply (eval op) (map eval args))]
    [(? number? n) n]
    [(? symbol? x)
     (if-it2 (lookup x)
             (fetch it)
             (error 'unbound))]))
```

This code uses the pattern-matching form called `match` from a standard library.

Since `match` is a macro, the stepper would normally start by showing the expansion of the `match` expression. It would only show the expansion of `if-it2` later, within the code produced by `match`. That code is of course a tangled web of nested conditionals, intermediate variable bindings, and failure continuations, all of which is irrelevant and distracting for someone interested in inspecting the behavior of `if-it2`. The left-hand side of Fig. 1 shows this view of the program's expansion. Note the hiding policy editor, which includes no directive about `match`.

To eliminate the noise and to focus on just the behavior of interest, the programmer instructs the macro stepper to consider `match` an opaque form. The macro stepper allows the programmer to update the policy by simply clicking on an occurrence of a macro and changing its designation in the policy editor. When the policy changes, the stepper immediately updates the display.

[3] For example, a macro that clones one of its subexpressions or inspects the structure of a subexpression is not well-behaved.

When the programmer hides `match`, instead of treating `match` as a rewriting rule, the stepper treats `match` as a primitive form that contains expansion contexts: the expression following the `match` keyword and the right-hand side of each clause. With `match` considered primitive, there is only one rewriting rule that applies to this program. See the right-hand side of Fig. 1.

Now the programmer can concentrate on `if-it2` in its original context, without the distraction of irrelevant expansion steps. As the screenshot shows, macro hiding does not interfere with the macro stepper's ability to determine the program's binding structure.

In principle, a macro hiding policy is simply a predicate on macro occurrences that determines whether to show or hide its details. In practice, programmers control the macro hiding policy by designating particular macros or groups of macros as opaque, and that designation applies to all occurrences of a macro.

The policy editor lets programmers specify the opacity of macros at two levels. The coarse level includes two broad classes of macros: those belonging to the base language [13] and those defined in standard libraries. These classes of macros correspond to the "Hide mzscheme syntax" and "Hide library syntax" checkboxes. For finer control over the hiding policy, programmers can override the designation of individual macros.

We could allow policies to contain more complicated provisions, and we are still exploring mechanisms for specifying these policies. We expect that time and user feedback will be necessary to find the best ways of building policies. So far we have only discovered two additional useful policies. The first hides a certain class of macros that implement contract checking [11]. The second hides the expansion of macros used in the right-hand sides of macro definitions: for example, uses of `syntax-rules` and `syntax-case`, which are actually macros. We have added support for these cases in an ad hoc manner.

3. Previous offerings

Most Scheme implementations provide limited support for inspecting macro expansion. Typically, this support consists of just two procedures: `expand` and `expand-once` (sometimes called `macroexpand` and `macroexpand-1` [32]). Some implementations also provide a procedure called `expand-only`.

When applied to a term, `expand` runs the macro expander on the term to completion. It does not give the programmer access to any intermediate steps of the expansion process. If expansion aborts with an error, `expand` usually shows the faulty term, but it does not show the term's context. The second common tool, `expand-once`, takes a term and, if it is a macro use, performs a single macro transformation step. Since `expand-once` returns only a simple term, it discards contextual information such as the syntactic environment and even the place where the macro expander left off. Finally, `expand-only` acts like `expand` but takes as an extra argument the list of macros to expand. Macro hiding can be seen as a generalization of `expand-only`.

Neither of these tools suffices for debugging complex macros. In general, macro programmers need to debug both syntactic and logical errors in their macros. Some macro bugs, such as using a macro in a way that doesn't match its rewriting rules, for example, cause the macro expander to halt with an error message. Other bugs pass through the expander undetected and result in a well-formed program—but the *wrong* well-formed program.

Programmers using `expand` may catch syntactic bugs, but they usually have a difficult time figuring out what sequence of transformations produced the term that finally exposed the flaw. The error raised by `expand` only reports the faulty term itself, not the context the term occurs in. Furthermore, `expand` races past logical errors; it is up to the programmer to decipher the fully-expanded code and deduce at what point the expansion diverged from the ex-

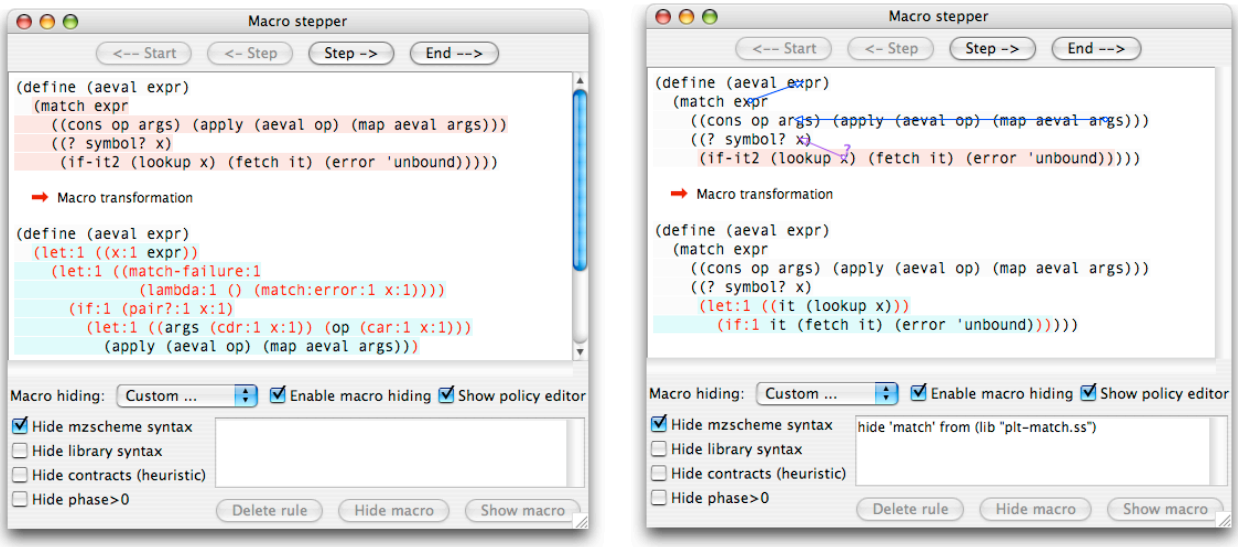


Figure 1. Example of macro hiding

pected path. While `expand-only` has some of the virtues of macro hiding, it has the same flaws as `expand`.

Using `expand-once`, on the other hand, leads the programmer through expansion step by step—but only for the root term. Thus `expand-once` is useless for finding errors that occur within non-trivial contexts. Such errors can occur in macros that depend on bindings introduced by other macros.

Implementing a better set of debugging tools than `expand` and `expand-once` is surprisingly difficult. It is impossible to apply many techniques that work well in run-time debugging. For example, any attempt to preprocess the program to attach debugging information or insert debugging statements fails for two reasons: first, until parsing and macro expansion happens, the syntactic structure of the tree is unknown; second, because macros can inspect their arguments, annotations or modifications are likely to change the result of the expansion process [35].

These limitations imply that a proper debugger for macros cannot be built atop the simplistic tools exposed to Scheme programmers. Implementing the macro stepper requires cooperation with the macro expander itself.

4. Implementation

The structure of our macro debugger is like the structure of a compiler: it has a front end that sits between the user and the intermediate representation (IR), a "middle end" or optimizer that performs advantageous transformations on the intermediate representation, and a back end that connects the intermediate representation to the program execution. While information in a compiler flows from the front end to the back end, information in a debugger starts at the back end and flows to the front end. The debugger's back end monitors the execution of the program; the front end displays the symbolic steps to the user. When present, the debugger's middle end is responsible for "optimizing" the IR for user comprehension.

In the macro stepper, the back end is connected to the macro expander, which we have instrumented to emit events that describe three aspects of the process: its progress, the intermediate terms, and the choices of fresh names and marks. The back end parses this stream of events into a structure that represents the call-tree of the expander functions for the expansion of the program. This structure serves as the macro stepper's intermediate representation. The middle end traverses the tree, hiding extraneous details. Finally, the front end turns the tree representation into a sequence of rewriting steps.

In this section we discuss the implementation of the macro stepper for a representative subset of Scheme. We show the instrumentation of the macro expander and we define parser that reconstructs the expander call tree. We describe how the macro hiding algorithm—the middle end—transforms the IR tree by detecting the contexts of hidden macros. Finally, we describe how the front end turns the IR into the rewriting sequences shown in Sect. 2.

4.1 The Macro Expander

The macro expander is implemented as a set of mutually recursive functions. PLT Scheme uses the `syntax-case` algorithm [6] for macro expansion, and we describe our implementation in the context of that algorithm, but our implementation strategy applies in principle to all Scheme macro expanders.

Figure 2 lists pseudocode for `expand-term`, the main expander function. Figure 3 provides definitions of some of the primitive expander functions; they recursively call `expand-term` on their subterms, as needed.

The `expand-term` function distinguishes macro applications, primitive syntax, and variable references with a combination of pattern matching on the structure of the term and environment lookup of the leading keyword.

Macro uses are handled according to the transformation rules associated with the macro. First, the expander creates a fresh mark and stamps it on the given term. Second, it transforms the term using the macro's transformation rules. Finally, it applies the mark again, canceling out marks on subterms of the original term. Naturally, the expander recurs on the result to eliminate macros in the resulting term. The self-cancelling marking is an efficient implementation of timestamps due to Dybvig et al. [6].

Primitive forms are handled by calling the primitive expander function associated with the keyword. The initial environment maps the name of each primitive (such as `lambda`) to its primitive expander (`expand-primitive-lambda`). When the primitive expander returns, expansion of that term is complete.

```
expand-term(term, env) =
  emit-event(Visit, term)
  case term of
    (kw . _)
      where lookup(resolve(kw), env)
              = ("macro", rules)
      => emit-event(EnterMacro)
         let M = fresh-mark
         let termM = mark(term, M)
         emit-event(Mark(M))
         let term2M = transform(rules, termM)
         let term2 = mark(term2M, M)
         emit-event(ExitMacro(term2))
         return expand-term(term2, env)
    (kw . _)
      where lookup(resolve(kw), env)
              = ("primitive", expander)
      => let term2 = expander(term, env)
         emit-event(Return(term2))
         return term2
    id
      where lookup(resolve(id), env)
              = "variable"
      => emit-event(Variable)
         emit-event(Return(term))
         return term2
    else
      => emit-event(Error)
         raise syntax error
```

Figure 2. Expansion function

4.2 Instrumenting the Expander

The shaded code in Fig. 2 and Fig. 3 represents our additions to the expander to emit debugging events.

The calls to `emit-event` send events through a private channel of communication to the macro stepper. The events carry data about the state of the macro expander. Figure 4 shows the event variants and the types of data they contain.

A Visit event indicates the beginning of an expansion step, and it contains the term being expanded. Likewise, the expansion of every term ends with a Return event that carries the expanded term.

The EnterMacro and ExitMacro events surround macro transformations. The Mark event carries the fresh mark for that transformation step, and the ExitMacro event carries the term produced by the transformation. The macro-handling case of `expand-term` does not include a Return event because the expander has not completed expansion of that term.

For every primitive, such as `if`, there is an event (PrimIf) that indicates that the macro expander is in the process of expanding that kind of primitive form. Primitives that create and apply renamings to terms send Rename events containing the fresh symbols that the bound names are renamed to.

4.3 Reconstruction

The back end of the macro stepper listens to the low-level events emitted by the instrumented macro expander and parses the sequence into a tree-structured intermediate representation. The kinds of events in the stream determine the variants of the tree's nodes, and the nodes' slots contain the data carried by the events.

Figure 5 lists the variants of the IR tree datatype. There is one variant for each primitive syntactic form plus one variant for a macro rewriting step. Every variant in the tree datatype starts with two fields that contain the initial term and final term for

```
expand-prim-lambda(term, env) =
  emit-event(PrimLambda)
  case term of
    (kw formals body)
      where formals is a list of identifiers
      => let newvars = freshnames(formals)
         let env2 = extend-env(env, newvars, "variable")
         let formals2 = rename(formals, formals, newvars)
         let body2 = rename(body, formals, newvars)
         emit-event(Rename(newvars))
         let body3 = expand-term(body2, env2)
         return (kw formals2 body3)
    else => emit-event(Error)
            raise syntax error

expand-prim-if(term, env) =
  emit-event(PrimIf)
  case term of
    (if test-term then-term else-term)
      => let test-term2 = expand-term(test-term, env)
         let then-term2 = expand-term(then-term, env)
         let else-term2 = expand-term(else-term, env)
         return (kw test-term2 then-term2 else-term2)
    else => emit-event(Error)
            raise syntax error

expand-primitive-let-syntax(term, env)
  emit-event(PrimLetSyntax)
  case term of
    (kw ([lhs rhs] ...) body)
      where each lhs is a distinct identifier
        and each rhs is a valid transformer
      => let lhss = (lhs ...)
         let rhss = (rhs ...)
         let newvars = freshnames(lhss)
         let env2 = extend-env(env, newvars, rhss)
         let body2 = rename(body, lhss, newvars)
         emit-event(Rename(newvars))
         return expand-term(body2, env2)
    else => emit-event(Error)
            raise syntax error

expand-primitive-application(term, env)
  emit-event(PrimApp)
  case term of
    (app . subterms)
      => let subterms2 =
           for t in subterms:
             expand-term(t, env)
         return subterms2
    else => emit-event(Error)
            raise syntax error
```

Figure 3. Expansion functions for primitives and macros

Events	*event*	::=	Visit(*expr*)
		|	Return(*expr*)
		|	EnterMacro
		|	Mark(*mark*)
		|	ExitMacro(*expr*)
		|	PrimLambda
		|	PrimIf
		|	PrimLetSyntax
		|	Variable
		|	Rename(*symbols*)
Traces	T	::=	list of events

Figure 4. Expansion events

IRTree	*ir*	::=	MacroNode(*expr*, *expr*, *mark*, *expr*, *ir*)
		|	VariableNode(*expr*, *expr*)
		|	LambdaNode(*expr*, *expr*, *symbols*, *ir*)
		|	IfNode(*expr*, *expr*, *ir*, *ir*, *ir*)
		|	AppNode(*expr*, *expr*, *irlist*)
		|	LetSyntaxNode(*expr*, *expr*, *symbols*, *ir*)

Figure 5. Intermediate representation structures

that expansion. The remainder of the fields of each variant are determined by the expander's behavior on that variant of term. The MacroNode variant contains a field for the timestamp associated with that step, the result of the macro transformation, and the subtree corresponding to the expansion of the macro's result. The LambdaNode variant contains a list of symbols for the fresh names chosen for the `lambda`-bound variables and an *IRTree* field that represents the expansion of the renamed body expression. The VariableNode variant contains no additional information.

Given the instrumented macro expander, we can easily read off the grammar for event streams. The terminals of the grammar consist of exactly the expansion events described in Fig. 4. There are two nonterminals: *ExpandTerm*, which corresponds to the events produced by an invocation of `expand-term`, and an auxiliary *ExpandTerm**. Each call to `emit-event` corresponds to a terminal on the right hand side, and each recursive call to `expand-term` corresponds to an occurrence of *ExpandTerm* on the right hand side. Figure 6 shows the grammar, $\mathcal{G}$; nonterminal names are italicized.

The back end parses the event streams into the intermediate representation. Figure 7 shows the parser function.[4]

The recursive structure of the parser is identical to the recursive structure of the expander.[5] The environment is not explicit in the IR datatype, but it can be reconstructed from a node's context as follows: traverse the IR tree from the root (the node corresponding to the expansion of the entire program) to the given node, adding to the environment on every LambdaNode or LetSyntaxNode node.

The macro stepper also handles expansions that halt because of syntax errors. Handling errors requires extending both the intermediate representation and the parser. We extend the IR by introducing wrappers that record the degree of completeness of the underlying tree node. Our approach is similar to, but less general than, one proposed by Gunter and Rémy [19] for big-step semantics. We extend the parser by annotating the grammar with possible error locations; we then transform the grammar to handle incomplete event streams and construct wrapped IR trees accordingly.

[4] In our implementation of the macro stepper, the parser is constructed from the grammar using a standard LR parser generator [26].

[5] We have proved the correctness of our reconstruction approach with respect to an annotated big-step semantics. See our technical report [4] for details.

4.4 Macro Hiding

Once the back end has created an IR structure, the macro stepper processes it with the user-specified macro hiding policy to get a new IR tree. Because the user can change the policy many times during the debugging session, the macro stepper retains the original IR tree, so that updating the display involves only reprocessing the tree and re-running the front end.

We refer to the tree produced by applying the macro policy as the *synthetic tree*. The datatype of synthetic trees contains an additional variant of node that does not correspond to any primitive syntactic form. This node contains a list of subterm expansions, where each subterm expansion consists of a derivation structure and a path representing the context of the subterm in the node's original syntax.

The macro hiding algorithm consists of two modes: hide and seek. In hide mode, the macro stepper recursively traverses the IR tree looking for macro-rewriting nodes to hide. In seek mode, the macro stepper is traversing the tree corresponding to a hidden macro's expansion, looking for subtrees corresponding to terms that existed in the macro occurrence. The algorithm is parameterized over the hiding policy.

More specifically, the hiding function consumes an IR tree. If it is a macro node and the policy says to hide the macro, the stepper switches to seek mode with the macro node's subtree and the macro expression. Seek mode returns a list of subtrees, and the stepper replaces the original macro node with a synthetic node containing the subtrees.

If the node encountered in hiding mode is not a a hidden macro, it simply recurs on the node's subtrees, updating the final term with any changes in the subtrees: for example, when a macro in a `lambda` body is hidden, the final form of the `lambda` term changes also.

In seek mode, the macro stepper is looking for expansion subtrees for terms that occurred in the original macro expression. The existence of such a subtree implies that the subterm occupied an *expansion context* of the hidden macro. When the stepper finds these subtrees, it switches back to hiding mode for the subtree, then pairs the resulting synthetic tree with the context of the hidden macro expression that it occurred in.

When in seek mode, the debugger also collects renaming steps performed by the primitive binding forms. It updates the original macro expression with the renaming step and adds the renaming step into the subtree list. This allows the front end to find bindings and draw binding arrows correctly even when the primitive binding form isn't visible in the macro stepper.

As a side benefit, macro hiding enables the stepper to detect a common beginner mistake: putting multiple copies of an input expression in the macro's output. If macro hiding produces a list of paths with duplicates (or more generally, with overlapping paths), the stepper warns the programmer of the potential error.

4.5 Front End

Once macro hiding has produced an IR tree adjusted to the programmer's level of abstraction, the macro stepper's front end translates the tree into a sequence of rewriting steps. This sequence is displayed by the macro stepper's graphical user interface. Each step contains the program before expansion and the program after expansion, along with a context specification that indicates where the step takes place. In addition, the data representation for a rewrite step records the bindings and references known at the time of the rewriting step. The macro stepper uses this information to draw the binding arrows.

$$
\begin{array}{lcl}
\mathit{ExpandTerm} & ::= & \mathsf{Visit\ EnterMacro\ Mark\ ExitMacro}\ \mathit{ExpandTerm} \\
& | & \mathsf{Visit\ PrimLambda\ Rename}\ \mathit{ExpandTerm}\ \mathsf{Return} \\
& | & \mathsf{Visit\ PrimIf}\ \mathit{ExpandTerm}\ \mathit{ExpandTerm}\ \mathit{ExpandTerm}\ \mathsf{Return} \\
& | & \mathsf{Visit\ PrimApp}\ \mathit{ExpandTerm}\text{*}\ \mathsf{Return} \\
& | & \mathsf{Visit\ PrimLetSyntax\ Rename}\ \mathit{ExpandTerm}\ \mathsf{Return} \\
& | & \mathsf{Visit\ Variable\ Return} \\
\mathit{ExpandTerm}\text{*} & ::= & \varepsilon \\
& | & \mathit{ExpandTerm}\ \mathit{ExpandTerm}\text{*}
\end{array}
$$

Figure 6. Grammar of event streams

$$
\begin{array}{l}
\mathrm{parse}(\mathsf{Visit}(\mathit{expr}); \mathsf{EnterMacro}; \mathsf{Mark}(m); \mathsf{ExitMacro}(\mathit{expr}'); \mathit{ExpandTerm}_n) = \\
\quad \mathsf{MacroNode}(\mathit{expr}, t_n\,\texttt{.finalTerm}, m, \mathit{expr}', t_n) \\
\mathrm{parse}(\mathsf{Visit}(\mathit{expr}); \mathsf{PrimLambda}; \mathsf{Rename}(\mathit{newvars}); \mathit{ExpandTerm}_b; \mathsf{Return}(\mathit{expr}')) = \\
\quad \mathsf{LambdaNode}(\mathit{expr}, \mathit{expr}', \mathit{base}, \mathit{newvars}, \mathrm{parse}(\mathit{ExpandTerm}_b)) \\
\mathrm{parse}(\mathsf{Visit}(\mathit{expr}); \mathsf{Variable}; \mathsf{Return}(\mathit{expr}')) = \\
\quad \mathsf{VariableNode}(\mathit{expr}, \mathit{expr}') \\
\mathrm{parse}(\mathsf{Visit}(\mathit{expr}); \mathsf{PrimIf}; \mathit{ExpandTerm}_1; \mathit{ExpandTerm}_2; \mathit{ExpandTerm}_3; \mathsf{Return}(\mathit{expr}')) = \\
\quad \mathsf{IfNode}(\mathit{expr}, \mathit{expr}', \mathrm{parse}(\mathit{ExpandTerm}_1), \mathrm{parse}(\mathit{ExpandTerm}_2), \mathrm{parse}(\mathit{ExpandTerm}_3)) \\
\mathrm{parse}(\mathsf{Visit}(\mathit{expr}); \mathsf{PrimApp}; \mathit{ExpandTerm}\text{*}; \mathsf{Return}(\mathit{expr}')) = \\
\quad \mathsf{AppNode}(\mathit{expr}, \mathit{expr}', \mathrm{parse}^*(\mathit{ExpandTerm}\text{*})) \\
\mathrm{parse}(\mathsf{Visit}(\mathit{expr}); \mathsf{PrimLetSyntax}; \mathsf{Rename}(\mathit{newvars}); \mathit{ExpandTerm}_b; \mathsf{Return}(\mathit{expr}')) = \\
\quad \mathsf{LetSyntaxNode}(\mathit{expr}, \mathit{expr}', \mathit{newvars}, \mathrm{parse}(\mathit{ExpandTerm}_b)) \\
\mathrm{parse}^*(\mathit{ExpandTerm}_1\ \mathit{ExpandTerm}\text{*}_2) = \\
\quad \mathsf{Cons}(\mathrm{parse}(\mathit{ExpandTerm}_1), \mathrm{parse}^*(\mathit{ExpandTerm}\text{*}_2)) \\
\mathrm{parse}^*(\varepsilon) = \mathsf{Empty}
\end{array}
$$

Figure 7. Parser

When the front end encounters a macro node in its traversal of the IR tree, it creates a rewriting step and adds it to the sequence generated by the macro node's continuation subtree. The step contains the program term before and after applying the macro. The front end constructs the second program term by inserting the macro's result term into the inherited program context.

When the front end encounters a primitive node in the IR tree, it generally recurs on the subtrees of the primitive node, extending the expansion context and threading through the program term and binding information. The subsequences from all of the node's subtrees are appended together.

The resulting rewriting sequence also stores information about the known bindings and references. Whenever the front end encounters a binding node—LambdaNode, LetSyntaxNode, or SyntheticNode containing renaming steps—it adds the bound occurrences to the accumulator for known bindings. Likewise, when it encounters a variable node or macro node, it adds the variable or macro keyword to the list of known references. Synthetic nodes do not directly add to the list of references; instead, they contain variable or macro nodes in their list of subterm expansion trees.

5. Experience

We have implemented a macro stepper for PLT Scheme following the design presented in this paper. The actual macro stepper handles the full PLT Scheme language, including modules, procedural macro transformers and their accompanying phase separation [12], source location tracking, and user-defined syntax properties. The macro stepper also handles additional powerful operations available to macro transformers such as performing local expansion and lifting expressions to top-level definitions.

The full language poses additional complications to the macro hiding algorithm. For example, PLT Scheme partially expands the contents of a block (the body of a `lambda` term, for example) to expose internal definitions, then transforms the block into a `letrec` expression and finishes handling the block by expanding the intermediate `letrec` expression. Advanced operations available to macro writers cause similar problems. In most cases the macro stepper is able to cleanly adapt the macro hiding algorithm; in a few cases the macro stepper compromises.

The macro stepper is available in the standard distribution of PLT Scheme. Macro programmers have been using it since its first release via nightly builds, and their practical experience confirms that it is an extremely useful tool. It has found use in illustrating macro expansion principles on small example programs, occasionally supplementing explanations given on the mailing list in response to macro questions. It has also helped researchers debug the large multi-module systems of co-operating macros that implement their language extensions. Our users report that the macro stepper has significantly increased their productivity.

In particular, their reports support the importance of macro hiding, the parameterization of the base language that allows programmers to work at the abstraction level of their choice. This feature of the debugger is critical for dealing with nontrivial programs. Otherwise, programmers are overwhelmed by extraneous details and their programs change beyond their recognition as the macro step-

per elaborates away the constructs that give programs their structure.

The macro stepper even assisted its own development as its first major application. We naturally implemented the grammar transformation for handling errors via macros, and our first attempt produced incorrect references to generated nonterminal names—a kind of lexical scoping bug. Fortunately, we were able to use a primitive version of the macro stepper to debug the grammar macros.

References

[1] Eli Barzilay. Swindle. `http://www.barzilay.org/Swindle`.

[2] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proc. 10th European Symposium on Programming Languages and Systems*, pages 320–334, 2001.

[3] William Clinger and Jonathan Rees. Macros that work. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[4] Ryan Culpepper and Matthias Felleisen. Debugging macros. Technical Report NU-CCIS-07-01, Northeastern University, April 2007.

[5] Ryan Culpepper, Scott Owens, and Matthew Flatt. Syntactic abstraction in component interfaces. In *Proc. Fourth International Conference on Generative Programming and Component Engineering*, pages 373–388, 2005.

[6] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

[7] Sebastian Egner. Eager comprehensions in Scheme: The design of SRFI-42. In *Proc. Sixth Workshop on Scheme and Functional Programming*, September 2005.

[8] Matthias Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University, 1985.

[9] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[10] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[11] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[12] Matthew Flatt. Composable and compilable macros: you want it when? In *Proc. Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.

[13] Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[14] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 236–248, 1998.

[15] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems (APLAS) 2006*, pages 270–289, November 2006.

[16] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University TR 97-293, June 1999.

[17] Daniel Friedman. Object-oriented style. In *International LISP Conference*, October 2003. Invited talk.

[18] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, July 2005.

[19] Carl A. Gunter and Didier Rémy. A proof-theoretic assessment of runtime type errors. Technical Report 11261-921230-43TM, 600 Mountain Ave, Murray Hill, NJ 07974-2070, 1993.

[20] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *Proc. Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 16–27, 2004.

[21] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[22] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[23] Oleg Kiselyov. A declarative applicative logic programming system, 2004–2006. `http://kanren.sourceforge.net`.

[24] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proc. 1986 ACM Conference on LISP and Functional Programming*, pages 151–161, 1986.

[25] Henry Lieberman. Steps toward better debugging tools for LISP. In *Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 247–255, 1984.

[26] Scott Owens, Matthew Flatt, Olin Shivers, and Benjamin McMullan. Lexer and parser generators in Scheme. In *Proc. Fifth Workshop on Scheme and Functional Programming*, pages 41–52, September 2004.

[27] PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR2006-4-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[28] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass framework for compiler education. *Journal of Functional Programming*, 15(5), September 2005. Educational Pearl.

[29] Olin Shivers. The anatomy of a loop: a story of scope and control. In *Proc. Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 2–14, 2005.

[30] Dorai Sitaram. Programming in schelog. `http://www.ccs.neu.edu/home/dorai/schelog/schelog.html`.

[31] Michael Sperber, William Clinger, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, and Jonathan Rees (Editors). Revised$^{5.91}$ report of the algorithmic language Scheme, 2006. Draft. Available at `http://www.r6rs.org`.

[32] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, 1990.

[33] Andrew P. Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[34] Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–215, 1999.

[35] Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.

[36] Andrew Wright and Bruce Duba. Pattern matching for Scheme, 1995. Unpublished manuscript.