A Stepper for Scheme Macros

Ryan Culpepper
Northeastern University
ryanc@ccs.neu.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

Even in the days of Lisp's simple `defmacro` systems, macro developers did not have adequate debugging support from their programming environment. Modern Scheme macro expanders are more complex than Lisp's, implementing lexical hygiene, referential transparency for macro definitions, and frequently source properties. Scheme implementations, however, have only adopted Lisp's inadequate macro inspection tools. Unfortunately, these tools rely on a naive model of the expansion process, thus leaving a gap between Scheme's complex mode of expansion and what the programmer sees.

In this paper, we present a macro debugger with full support for modern Scheme macros. To construct the debugger, we have extended the macro expander so that it issues a series of expansion events. A parser turns these event streams into derivations in a natural semantics for macro expansion. From these derivations, the debugger extracts a reduction-sequence (stepping) view of the expansion. A programmer can specify with simple policies which parts of a derivation to omit and which parts to show. Last but not least, the debugger includes a syntax browser that graphically displays the various pieces of information that the expander attaches to syntactic tokens.

1. The Power of Macros

Modern functional programming languages support a variety of abstraction mechanisms: higher-order functions, expressive type systems, module systems, and more. With functions, types, and modules, programmers can develop code for reuse; establish single points of control for a piece of functionality; decouple distinct components and work on them separately; and so on. As Paul Hudak [18] has argued, however, "the ultimate abstraction of an application is a . . . language." Put differently, the ideal programming language should allow programmers to develop and embed 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 *syntactic abstractions* that manipulate binding structure, perform some analyses, re-order the evaluation of expressions, and generally transform syntax in complex ways—all at compile time. As some Scheme implementors have put it, macros have become a true compiler (front-end) API.

In the context of an expressive language [9] macros suffice to implement many general-purpose abstraction mechanisms as libraries that are *indistinguishable from built-in features*. For example, programmers have used macros to extend Scheme with constructs for pattern matching [32], relations in the spirit of Prolog [8, 27, 15, 20], extensible looping constructs [7, 26], class systems [24, 1, 14] and component systems [30, 13, 5], among others. In addition, programmers have also used macros to handle tasks traditionally implemented as *external* metaprogramming tasks using preprocessors or special compilers: Owens et al. [23] have added a parser generator library to Scheme; Sarkar et al. [25] have created an infrastructure for expressing nano-compiler passes; and Herman and Meunier [17] have used macros to improve the set-based analysis of Scheme. As a result, implementations of Scheme such as PLT Scheme [12] have a core of a dozen or so constructs but appear to implement a language the size of Common Lisp.

To support these increasingly ambitious applications, macro systems had to evolve, too. In Lisp systems, macros are compile-time functions over program fragments, usually plain S-expressions. Unfortunately, these naive macros don't really define abstractions. For example, these macros interfere with the lexical scope of their host programs, revealing implementation details instead of encapsulating them. In response, Kohlbecker et al. [21] followed by others [4, 6, 11] developed the notions of macro hygiene, referential transparency, and phase separation. In this world, macros manipulate syntax tokens that come with information about lexical scope; affecting scope now takes a deliberate effort and becomes a part of the macro's specification. As a natural generalization, modern Scheme macros don't manipulate S-expressions at all but opaque syntax representations that carry additional information. In the beginning, this information was limited to binding information; later Dybvig et al. [6] included source information. Now, Scheme macros contain arbitrary properties [12] and programmers discover novel uses of this mechanism all the time.

Although all these additions were necessary to create true syntactic abstraction mechanisms, they also dramatically increased the complexity of macro systems. The result is that both inexperienced and advanced users routinely ask on Scheme mailing lists about unforeseen effects, subtle errors, or other seemingly inexplicable phenomena. While "macrologists" 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 and debugger. Constructing this tool proved surprisingly complex. The purpose of the next section is to explain the difficulties abstractly, before we demonstrate how our tool works and how it is constructed.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

2. Explaining Macros

Macro expansion takes place during parsing. As the parser traverses the concrete syntax,[1] it creates abstract syntax nodes for primitive syntactic forms, but stops when it recognizes the use of a macro. At that point, it hands over the (sub)phrases to the macro, which, roughly speaking, acts as a rewriting rule.

In Lisp and in some Scheme implementations, a macro is expressed as a plain function; in R^{5}RS Scheme [19], macros are expressed in a sub-language of rewriting rules based on patterns. Also in Lisp, concrete syntax are just S-expressions; Lisp macro programming is thus typically first-order functional programming[2] over pairs and symbols. The most widely used Scheme implementations, however, represent concrete syntax with structures that carry additional information: lexical binding information, original source location, code security annotations, and others. Scheme macro programming is therefore functional programming with a rich algebraic datatype.

Given appropriate inputs, a Lisp macro can go wrong in two ways. First, the macro transformer itself may raise a run-time exception. This case is naturally in the domain of run-time debuggers; after all, it is just a matter of traditional functional programming. Second, a Lisp macro may create a new term that misuses a syntactic form, which might be a primitive form or another macro. This kind of error is not detected when the macro is executing, but only afterwards when the parser-expander reaches the misused term.

Modern Scheme macros might go wrong in yet another way. The additional information in a syntax object interacts with other macros and primitive special forms. For example, macro-introduced identifiers carry a mark that identifies the point of their introduction and binding forms interpret identifiers with different marks as distinct names. Scheme macros must not only compute a correct replacement tree but also equip it with the proper additional properties.

Even in Lisp, which has supported macros for almost 50 years now, macros have always had impoverished debugging environments. A typical Lisp environment supports just two procedures/tools for this purpose: `expand` and `expand-once` (or `macroexpand` and `macroexpand-1` [28]). All Scheme implementations with macros have adapted these procedures.

When applied to a term, `expand` completely parses and expands it; in particular, it does not show the intermediate steps of the rewriting process. As a result, `expand` distracts the programmer with too many irrelevant details. For example, Scheme has three conditional expressions: `if`, `cond`, and `case`. Most Scheme implementations implement only `if` as a primitive form and define `cond` and `case` as macros. Whether or not a special form is a primitive form or a macro is irrelevant to a programmer except that macro expansion reveals the difference. It is thus impossible to study the effects of a single macro or a group of related macros in an expansion, because `expand` processes *all* macros and displays the entire abstract syntax tree.

The task of showing individual expansion steps is left to the second tool: `expand-once`. It consumes a macro application, applies the matching macro transformer, and returns the result. In particular, when an error shows up due to complex macro interactions, it becomes difficult to use `expand-once` easily because the offending or interesting pieces are often hidden under a large pile of syntax. Worse, iterated calls to `expand-once` lose information between expansion steps, because lexical scope and other information depends on the context of the expansion call. This problem renders `expand-once` unfit for serious macro debugging.

[1] We consider the result of `(read)` as syntax.

[2] Both Lisp and Scheme macro programmers occasionally use side-effects but aside from `gensym` it is rare.

Implementing a better set of debugging tools than `expand` and `expand-once` is surprisingly difficult. It is apparently impossible to adapt the techniques known from run-time debugging. For example, any attempt to pre-process the syntax and 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 inspect their arguments, annotations or modifications are likely to change the result of the expansion process [31].

While these reasons explain the dearth of macro debugging tools and steppers, they don't reduce the need for them. What we present in this paper is a mechanism for instrumenting the macro expander and for displaying the expansion events and intermediate stages in a useful manner. Eventually we also hope to derive a well-founded model of macros from this work.

3. The Macro Debugger at Work

The core of our macro debugging tool is a stepper for macro expansion in PLT Scheme. Our macro debugger shows the macro expansion process as a reduction sequence, where the redexes are macro applications and the contexts are primitive syntactic forms, i.e., nodes in the final abstract syntax tree. The debugger also includes a syntax display and browser that helps programmers visualize properties of syntax objects.

The macro stepper is parameterized over a set of "opaque" syntactic forms. Typically this set includes those macros imported from libraries or other modules. The macro programmers are in charge, however, and may designate macros as opaque as needed. When the debugger encounters an opaque macro, it deals with the macro as if it were a primitive syntactic form. That is, it creates an abstract syntax node that hides the actual expansion of the macro. Naturally, it does show the expansion of the subexpressions of the macro form. The parameterization of primitive forms thus allows programmers to work at the abstraction level of their choice. We have found this feature of the debugger critical for dealing with any nontrivial programs.

The rest of this section is a brief illustrative demonstration of the debugger. We have picked three problems with macros from recent discussions on PLT Scheme mailing lists, though we have distilled them into a shape that is suitably simple for a technical paper.

3.1 Plain Macros

For our first example we consider a debugging scenario where the macro writer gets the form of the result wrong. Here are three different versions of a sample macro that consumes a list of identifiers and produces a list of trivial definitions for these identifiers:

1. in Lisp, the macro writer uses plain list-processing functions to create the result term:

```
(define-macro (def-false . names)
  (map (lambda (a) `(define ,a #f)) names))
```

2. in R^{5}RS the same macro is expressed with a rewriting rule notation like this:

```
(define-syntax def-false
  (syntax-rules ()
    [(def-false a ...) ((define a #f) ...)]))
```

3. in major alternative Scheme macro systems, the rule specification is slightly different:

```
(define-syntax (def-false stx)
  (syntax-case stx ()
    [(_ a ...) (syntax ((define a #f) ...))]))
```

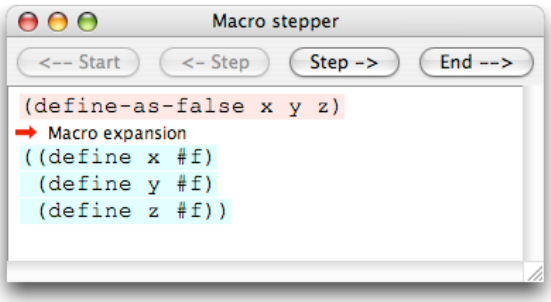
The macro definition is a function that consumes a syntax tree, named `stx`. The `syntax-case` construct de-structures the tree and binds pattern variables to its components. The `syntax` constructor produces a new syntax tree by replacing the pattern variables in its template with their values.

Using the macro, like thus:

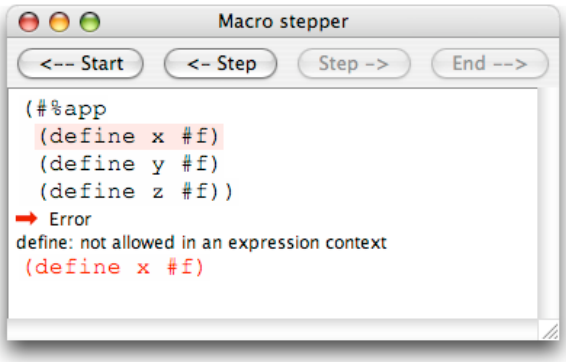
```
(def-false x y z)
```

immediately exposes a problem. The macro expander fails with an error explaining that definitions can't occur in an expression context. Of course, the problem is that the macro produces a list of terms, which the macro expander interprets as an application and which, in turn, may not contain any definitions.

Our macro stepper shows the sequence of macro expansion steps, one at a time:



Here we can see both the original macro form and the output of the macro application. The original appears at the top, the output of the first step at the bottom. The highlighted subterms on the top and bottom are the redex and contractum, respectively. The separator explains that this is a macro expansion step. At this point, an experienced Lisp or Scheme programmer recognizes the problem. A novice may need to see another step:



Here the macro expander has explicitly tagged the term as an application. The third step then shows the syntax error, highlighting the term *and* the context in which it occurred.

The macro debugger actually expands the entire term before it displays the individual steps. This allows programmers to skip to the very end of a macro expansion and to work backwards. The stepper supports this approach with a graphical user interface that permits programmers to go back and forth in an expansion and also to skip to the very end and the very beginning. The ideas for this interface have been borrowed from Clements's algebraic run-time stepper for PLT Scheme [3]; prior to that, similar ideas appeared in Lieberman's stepper [22] and Tolmach's SML debugger [29].

3.2 Syntax properties

Nearly all hygienic macro papers use the `or` macro to illustrate the problem of inadvertent variable capture:

```
(define-syntax (or stx)
  (syntax-case stx ()
    [(or e1 e2)
     (syntax (let ([tmp e1]) (if tmp tmp e2)))]))
```

In Scheme, the purpose of `(or a b)` is to evaluate `a` and to produce its value, unless it is false; if it is false, the form evaluates `b` and produces its value as the result.

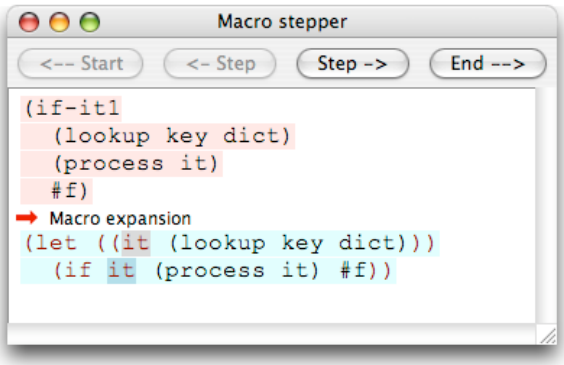
In order to keep `or` from evaluating its first argument more than once, the macro introduces a new variable for the first result. In Lisp-style macro expanders (or Scheme prior to 1986), the new `tmp` binding captures any free references to `tmp` in `e2`, thus interfering with the semantics of the macro and the program. Consequently, the macro breaks abstraction barriers. In Scheme, the new `tmp` identifier carries a mark or timestamp—introduced by the macro expander—that prevents it from binding anything but the two occurrences of `tmp` in the body of the macro-generated `let` [21]. This mark is vital to Scheme's macro expansion process, but no interface exists for inspecting the marks and the marking process directly.

Our macro debugger visually displays this scope information at every step. The display indicates with different text colors[3] from which macro expansion step every subterm originated. Furthermore, the programmer can select a particular subterm and see how the other subterms are related to it. Finally, the macro stepper can display a properties panel to show more detailed information such as identifier bindings and source locations.

The following example shows a programmer's attempt to create a macro called `if-it`, a variant of `if` that tries to bind the variable `it` to the result of the test expression for the two branches:

```
(define-syntax (if-it1 stx) ;; WARNING: INCORRECT
  (syntax-case stx ()
    [(if-it1 test then else)
     (syntax
      (let ([it test]) (if it then else)))]))
```

The same mechanism that prevents the inadvertent capture in the `or` example prevents the *intentional* capture here, too. With our macro debugger, the puzzled macro writer immediately recognizes why the macro doesn't work:



When the programmer selects an identifier, that identifier and all others with compatible binding properties are highlighted in the same color. Thus, in the screenshot above, the occurrence of `it` from the original program is *not* highlighted while the two macro-introduced occurrences are.

[3] Or numeric suffixes when there are no more easily distinguishable colors.

For completeness, here is the macro definition for a working version of `if-it`:

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

This macro creates an identifier named `it` with the lexical context of the original expression. The `syntax` form automatically unquotes `it` and injects the new identifier, with the correct properties, into the output. When the programmer examines the expansion of `(if-it2 a b c)`, all occurrences of `it` in `then` and `else` are highlighted now.

3.3 Scaling up

The preceding two subsections have demonstrated the workings of the macro debugger on self-contained examples. Some macros cannot be tested in a stand-alone mode, however, because it is difficult to extract them from the environment in which they occur.

One reason is that complex macros add entire sub-languages, not just individual features to the core. Such macros usually introduce local helper macros that are valid in a certain scope but nowhere else. For example, the `class` form in PLT Scheme, which is implemented as a macro, introduces a `super` form—also a macro—so that methods in derived classes can call methods in the base class. Since the definition of `super` depends on the rest of the class, it is difficult to create a small test case to explore its behavior. While restructuring such macros occasionally improves testability, *requiring restructuring for debugging is unreasonable*.

In general, the problem is that 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 macros on this list, but it does display the expansions of the subtrees in the context of the original macro form. That is, 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,[4] they have the same meaning as the original and final programs.

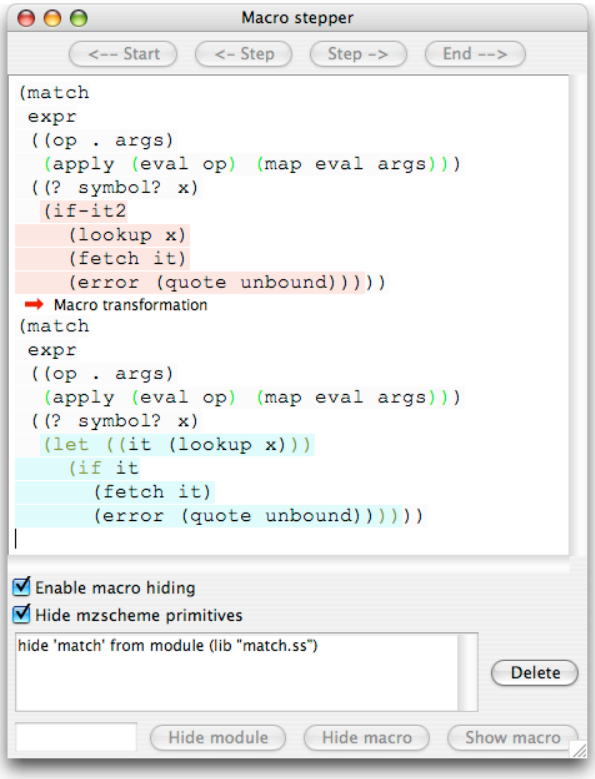
Consider the `if-it2` macro from the previous subsection. After testing the macro itself, the programmer wishes to employ it in the context of a larger program:

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

This snippet uses the pattern-matching form called `match` from a standard (macro) library.

If the debugger had no "opaqueness policy" covering `match`, the macro expander and therefore the stepper would show the expansion of `if-it2` within the code produced by `match` macro. 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 the implementor of `if-it2`.

To eliminate the noise and focus on just the behavior of interest, the programmer instructs the macro debugger to consider `match` an opaque form. Then the macro debugger shows the expansion of the code above as a single step:



Although macro hiding and opaqueness policies simplify the story of expansion presented to the programmer, it turns out that implementing them is difficult and severely constrains the internal organization of the macro stepper. Before we can explain this, however, we must explain how Scheme expands macros.

4. Macro Expansion

Our model of macro expansion is an adaptation of Clinger and Rees's model [4], enriched with the lexical scoping mechanism of Dybvig et al. [6] and the phase separation of Flatt [11]. Figures 1 through 3 display the details.

Macro expansion is a recursive process that takes an expression and eliminates the macros, resulting in an expression in the core syntax of the language. The macro expander uses an environment to manage bindings and a phase number to manage staging. We use the following judgment to say the term $expr$ fully macro expands into $expr'$ in the syntactic environment E in phase number p:

$$p, E \vdash expr \Downarrow expr'$$

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

Terms	$expr$	::=	$identifier$
		|	$datum$
		|	$(expr\ \cdots\ expr)$
Identifiers	x, kw	::=	$symbol$
		|	$\mathrm{mark}(identifier, mark)$
		|	$\mathrm{subst}(ident, ident, symbol)$
Symbols	s	::=	countable set of names
Marks	$mark$	::=	countable set
Phases	p	::=	number: $0, 1, 2, \ldots$
Denotation	d	::=	`variable`
		|	$\langle$`primitive`, $symbol\rangle$
		|	$\langle$`macro`, $transformer\rangle$

Environments $E : symbol \times phase \rightarrow denotation$

Expansion relation	$p, E \vdash expr \Downarrow expr$
Macro step relation	$p, E \vdash expr \rightarrow expr$
Evaluation relation	$p, E \vdash expr \Downarrow_{eval} expr$

Figure 1. Semantic domains and relations

Figure 1 summarizes the domains and metavariables we use to describe the semantics.[5]

The structure of expressions varies from system to system. Lisp macro expanders operate on simple concrete syntax trees. Scheme macro systems are required to be hygienic—that is, they must respect lexical binding—and the expressions they manipulate are correspondingly complex. The hygiene principle [21, 4] of macro expansion places two requirements on macro expansion's interaction with lexical scoping:

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 expansion step.

The gist of the hygiene requirement is that macros act "like closures" at compile time. Hence the meaning of a macro can be determined from its environment and its input; it does not depend on the context the macro is used in. Furthermore, if the macro creates a binding for a name that comes from the macro itself, it doesn't affect expressions that the macro receives as arguments.

Thinking of macro expansion in terms of substitution provides additional insight into the problem. There are two occurrences of substitution, and two kinds of capture to avoid. The first substitution consists of copying the macro body down 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 (hygiene 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 (hygiene condition 2), even though the latter bindings are not immediately apparent in the macro's result.

Consider the following sample macro:

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

This macro puts its argument in the context of a use of a `mangle` macro. Without performing further expansion steps, the macro expander cannot tell if the occurrence of `x` is a binding occurrence. The expander must keep enough information to allow for both possibilities and delay its determination of the role of `x`.

The hygiene requirement influences the way lexical bindings are handled, and that in turn influences the structure of the expression representation. Technically, the semantics utilizes substitution and marking operations on expressions:

$$\mathrm{subst} : Expr \times Identifier \times Symbol \rightarrow Expr$$
$$\mathrm{mark} : Expr \times Mark \rightarrow Expr$$

Intuitively, these operations perform renaming and reversible stamping on all the identifiers contained in the given expression. These operations are generally done lazily for efficiency. There is an accompanying forcing operation called resolve

$$\mathrm{resolve} : Identifier \rightarrow Symbol$$

that sorts through the marks and substitutions to find the meaning of the identifier. Dybvig et al. [6] explain identifier resolution in detail and justify the lazy marking and renaming operations.

The expander uses these operations to implement variable renaming and generation of fresh names, but they don't carry the *meaning* of the identifiers; that resides in the expander's environment. This *syntactic environment* maps a symbol and a phase to a macro, name of a primitive form, or the designator `variable` for a value binding.

Determining the meaning of an identifier involves first resolving the substitutions to a symbol, and then consulting the environment for the meaning of the symbol in the current phase.

4.1 Primitive syntactic forms

Handling primitive syntactic forms generally involves recursively expanding the expression's subterms; sometimes the primitive applies renaming steps to the subterms before expanding them. Determining which rule applies to a given term involves resolving the leading keyword and consulting the environment. Consider the `lambda` rule from Figure 2. The keyword may be something other than the literal symbol `lambda`, but the rule applies to any form where the leading identifier has the *meaning* of the primitive `lambda` in the current environment.

The `lambda` syntactic form generates a new name for each of its formal parameters and creates a new body term with the old formals mapped to the new names—this is the renaming step. Then it extends the environment, mapping the new names to the `variable` designator, and expands the new body term in the extended environment. Finally, it re-assembles the `lambda` term with the new formals and expanded body.

When the macro expander encounters one of the `lambda`-bound variables in the body expression, it resolves the identifier to the fresh symbol from the renaming step, checks to make sure that the environment maps it to the `variable` designator (otherwise it is a misused macro or primitive name), and returns the resolved symbol.

The `if` and `app` (application) rules are simple; they just expand their subexpressions in the same environment.

The `let-syntax` form, which introduces local macro definitions, requires the most complex derivation rule (Figure 3). Like `lambda`, it constructs fresh names and applies a substitution to the body expression. However, it expands the right hand sides of the bindings using a phase number one greater than the current phase number. This prevents the macro transformers, which exist at compile time, from accessing *run-time* variables.[6] The macro transformers are then *evaluated* in the higher phase, and the envi-

[5] For simplicity, we do not model the store. Flatt [11] presents a detailed discussion of the interaction between phases, environments, and the store.

[6] Flatt [11] uses phases to guarantee separate compilation on the context of modules that import and export macros, but those issues are beyond the scope of this discussion.

ronment is extended in the *original* phase with the mapping of the macro names to the resulting values. The body of the `let-syntax` form is expanded in the extended environment, but with the original phase number.

For our purposes the run-time component of the semantics is irrelevant; we simply assume that the macro transformers act as functions that compute the representation of a new term when applied to a representation of the macro use. Thus, we leave the evaluation relation ($\Downarrow_{eval}$) unspecified.

4.2 Macros

Expanding a macro application involves performing the immediate macro transformation and then expanding the resulting term. The transformation step, represented by judgments of the form:

$$p, E \vdash expr \rightarrow expr$$

essentially consists of retrieving the appropriate macro transformer function from the environment and applying it to the macro use. We assume an invertible mapping from terms to data:

$$\begin{array}{rcl} \text{reflect} & : & Expr \rightarrow Datum \\ \text{reify} & : & Datum \rightarrow Expr \end{array}$$

$$\text{reify}(\text{reflect}(expr)) \quad = \quad expr$$

Like the evaluation relation ($\Downarrow_{eval}$), the details [6] of this operation are unimportant.

The interesting part of the macro rule is the marking and unmarking that supports the second hygiene condition. Identifiers introduced by the macro should not bind occurrences of free variables that come from the macro's arguments. The macro expander must somehow distinguish the two. The expander marks the macro arguments with a unique mark. When the macro transformer returns a new term, the parts originating from arguments still have a mark, and those that are newly introduced do not. Applying the same mark again cancels out the mark on old expressions and results in marks on only the introduced expressions.

When a marked identifier is used in a binding construct, the substitution only affects identifiers with the same name *and* the same mark. This satisfies the requirements of the second hygiene condition.

4.3 Syntax properties

The semantics shows how one kind of syntax property (scoping information) is manipulated and propagated by the macro expansion process.

The macro systems of real Scheme implementations define various other properties. For example, some put source location information in the syntax objects, and this information is preserved throughout macro expansion. Run time tools such as debuggers and profilers in these systems can then report facts about the execution of the program in terms of positions in the original code.

PLT Scheme allows macro writers to attach information keyed by arbitrary values to syntax. This mechanism has given rise to numerous lightweight protocols between macros, primitive syntax, and language tools.

5. Implementation

Programmers think of macros as rewriting specifications, where macro uses are replaced with their expansions. Therefore a macro debugger should show macro expansion as a sequence of rewriting steps. These steps are suggestive of a reduction semantics, but in fact we have not formulated a reduction semantics for macro expansion.[7] For a reduction semantics to be as faithful as the natural semantics we have presented, it would have to introduce administrative terms that obscure the user's program. We prefer to present an incomplete but understandable sequence of steps containing only terms from the user's program and those produced by macro expansion.

This section describes how we use the semantics in the implementation of the stepper, and the relationship between the semantics and the information displayed to the user.

5.1 Overview

The structure of a debugger is like the structure of a compiler. It has a front end that sits between the user and the debugger's internal representation of the program execution, a "middle end" or optimizer that performs translations on the internal representation, and a back end that connects the internal representation to the program execution.

While information flows from a compiler's front end to the back end, information in a debugger starts at the back end and flows through the front end to the user. The debugger's back end monitors the low-level execution of the program, and the front end displays an abstract view of the execution to the user. The debugger's middle end is responsible for finessing the abstract view, "optimizing" it for user comprehension.

Figure 4 displays the flow of information through our debugger. We have instrumented the PLT Scheme macro expander to emit information about the expansion process. The macro debugger receives this low-level information as a stream of events that carry data representing intermediate subterms and renamings. The macro debugger parses this low-level event stream into a structure representing the derivation in the natural semantics that corresponds to the execution of the macro expander. These derivations constitute the debugger's intermediate representation.

The debugger's middle end operates on the derivation generated by the back end, computing a new derivation tree with certain branches pruned away in accordance with the macro hiding policy. Finally, the front end traverses the optimized intermediate representation of expansion, turning the derivation structure into a sequence of rewriting steps, which the debugger displays in a graphical view. This view supports the standard stepping navigation controls. It also decorates the text of the program fragments with colors and mark-ups that convey additional information about the intermediate terms.

5.2 The Back End: Instrumentation

The macro expander of PLT Scheme is implemented as a collection of mutually recursive functions. Figure 5 presents a distilled version of the main function (in pseudocode).

The `expand-term` function checks the form of the given term. It 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 by applying the corresponding transformer to a marked copy of the term, then unmarking and recurring on the result. Primitive forms are handled by calling the corresponding primitive expander function from the environment. 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 is complete for that term. The definitions of some of the primitive expander functions are

[7] Bove and Arbilla [2], followed by Gasbichler [16], have formulated reduction systems that present the macro expansion process as an ordered sequence of states. Each state is a term in an explicit substitution syntax (plus additional attributes). Unfortunately, these semantics are complex and unsuitable as simple specifications for a reduction system. In comparison, Clements [3] uses beta-value and delta-value rules as a complete specification and proves his stepper correct.

$$\textsc{Lambda}\ \frac{E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{lambda}\rangle \qquad \mathit{formals} \text{ is a list of distinct identifiers}}{\begin{array}{c}\mathit{formals}' = \text{freshnames}_E(\mathit{formals}) \qquad \mathit{body}' = \text{subst}(\mathit{body}, \mathit{formals}, \mathit{formals}') \\ E' = E[\langle \mathit{formals}', p\rangle \mapsto \texttt{variable}] \qquad p, E' \vdash \mathit{body}' \Downarrow \mathit{body}'' \\ \hline p, E \vdash \texttt{(}kw\ \ \mathit{formals}\ \ \mathit{body}\texttt{)} \Downarrow \texttt{(}kw\ \ \mathit{formals}'\ \ \mathit{body}''\texttt{)}\end{array}}$$

$$\textsc{Variable}\ \frac{\text{resolve}(x) = x' \qquad E(x', p) = \texttt{variable}}{p, E \vdash x \Downarrow x'}$$

$$\textsc{If}\ \frac{\begin{array}{c}E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{if}\rangle \\ p, E \vdash \mathit{test} \Downarrow \mathit{test}' \qquad p, E \vdash \mathit{then} \Downarrow \mathit{then}' \qquad p, E \vdash \mathit{else} \Downarrow \mathit{else}'\end{array}}{p, E \vdash \texttt{(}kw\ \ \mathit{test}\ \ \mathit{then}\ \ \mathit{else}\texttt{)} \Downarrow \texttt{(}kw\ \ \mathit{test}'\ \ \mathit{then}'\ \ \mathit{else}'\texttt{)}}$$

$$\textsc{Application}\ \frac{\begin{array}{c}E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{app}\rangle \\ \forall i \le n : p, E \vdash \mathit{expr} \Downarrow \mathit{expr}'\end{array}}{p, E \vdash \texttt{(}kw\ \ \mathit{expr}_0\ \cdots\ \mathit{expr}_n\texttt{)} \Downarrow \texttt{(}kw\ \ \mathit{expr}'_0\ \cdots\ \mathit{expr}'_n\texttt{)}}$$

Figure 2. Primitive syntactic forms

$$\textsc{Let-Syntax}\ \frac{\begin{array}{c}E(\text{resolve}(kw), p) = \langle \texttt{primitive}, \texttt{let-syntax}\rangle \qquad \text{each } \mathit{var}_i \text{ is a distinct identifier} \\ \forall i \le n : \mathit{var}'_i = \text{freshname}_E(\mathit{var}_i) \\ \mathit{vars} = \{\mathit{var}_0, \ldots, \mathit{var}_n\} \qquad \mathit{vars}' = \{\mathit{var}'_0, \ldots, \mathit{var}'_n\} \qquad \mathit{body}' = \text{subst}(\mathit{body}, \mathit{vars}, \mathit{vars}') \\ \forall i \le n : p+1, E \vdash \mathit{rhs}_i \Downarrow \mathit{rhs}'_i \qquad \forall i \le n : p+1, E \vdash \mathit{rhs}'_i \Downarrow_{\mathit{eval}} \mathit{transformer}_i \\ E' = E[\{\langle \mathit{var}'_i, p\rangle \mapsto \mathit{transformer}_i\}] \qquad p, E' \vdash \mathit{body}' \Downarrow \mathit{body}''\end{array}}{p, E \vdash \texttt{(}kw\ \ \texttt{((}\mathit{var}_0\ \ \mathit{rhs}_0\texttt{)}\ \cdots\ \texttt{(}\mathit{var}_n\ \ \mathit{rhs}_n\texttt{))}\ \ \mathit{body}\texttt{)} \Downarrow \mathit{body}''}$$

$$\textsc{Macro}\ \frac{p, E \vdash \mathit{expr} \to \mathit{expr}' \qquad p, E \vdash \mathit{expr}' \Downarrow \mathit{expr}''}{p, E \vdash \mathit{expr} \Downarrow \mathit{expr}''}$$

$$\textsc{Macro-Step}\ \frac{\begin{array}{c}E(\text{resolve}(kw), p) = \langle \texttt{macro}, \mathit{transformer}\rangle \qquad \mathit{mark} = \text{freshmark}_E \\ \mathit{expr} = \texttt{(}kw\ \ \mathit{form}_1\ \cdots\ \mathit{form}_n\texttt{)} \qquad \mathit{stx} = \text{reflect}(\text{mark}(\mathit{expr}, \mathit{mark})) \\ p+1, E \vdash \texttt{(}\mathit{transformer}\ \ \mathit{stx}\texttt{)} \Downarrow_{\mathit{eval}} \mathit{stx}' \qquad \mathit{expr}' = \text{mark}(\text{reify}(\mathit{stx}'), \mathit{mark})\end{array}}{p, E \vdash \texttt{(}kw\ \ \mathit{form}_1\ \cdots\ \mathit{form}_n\texttt{)} \to \mathit{expr}'}$$

Figure 3. Macro definitions and uses

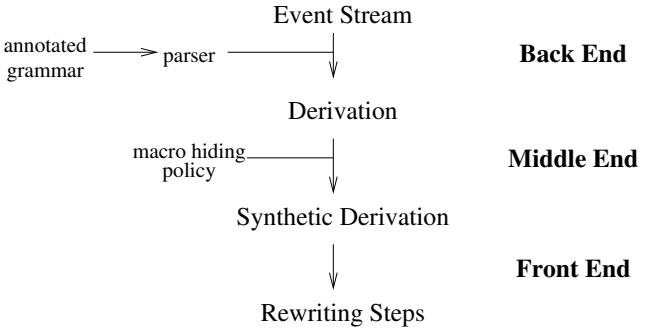


Figure 4. Implementation overview

given in Fig. 6; they recursively call `expand-term` on their subterms, as needed.

The shaded code in Fig. 5 and Fig. 6 represents our additions to the expander to emit debugging events. The calls to `emit-event` correspond to our instrumentation for the macro debugger. A call to `emit-event` send an event through a channel of communication to the macro debugger.

The events carry data about the state of the macro expander. Figure 7 shows a few of the event variants and the types of data they contain. A `visit` event indicates the beginning of an expansion step, and it contains the syntax being expanded. Likewise, the expansion of every term ends with a `return` event that carries the expanded syntax. The `enter-macro` and `exit-macro` events surround macro transformation steps, and the `macro-pre` and `macro-post` contain the marked versions of the starting and resulting terms. The `enter-primitive` and `exit-primitive` events surround all primitive form expansions. For every primitive, such as `if`, there is an event (`primitive-if`) 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 renamed syntax. The `next` signal separates the recursive expansion of subterms; `next-part` separates different kinds of subterms for primitives such as `let-syntax`.

5.3 The Back End: From Events to Derivations

The back end of the macro debugger transforms the low-level event stream from the instrumented expander into a derivation structure. This derivation structure corresponds to the natural semantics account of the expansion process.

The kinds of events in the stream determine the structure of the derivation, and the information carried by the events fills in the fields of the derivation objects. Figure 8 lists a few of the variants of the derivation datatype.

```
expand-term(term, env, phase) =
  emit-event("visit", term)
  case term of
    (kw . _)
      where lookup(resolve(kw), env, phase)
              = ("macro", transformer)
    => emit-event("enter-macro", term)
       let M = fresh-mark
       let term/M = mark(term, M)
       emit-event("macro-pre", term/M)
       let term2/M = transformer(term/M)
       emit-event("macro-post", term/M)
       let term2 = mark(term2/M, M)
       emit-event("exit-macro", term2)
       return expand-term(term2, env, phase)
    (kw . _)
      where lookup(resolve(kw), env, phase)
              = ("primitive", expander)
    => emit-event("enter-primitive", term)
       let term2 = expander(term, env, phase)
       emit-event("exit-primitive", term2)
       emit-event("return", term2)
       return term2
    id
      where lookup(resolve(id), env, phase)
              = "variable"
    => emit-event("enter-primitive", id)
       let term2 = expand-variable(id, env, phase)
       emit-event("exit-primitive", term2)
       emit-event("return", term2)
       return term2
    else
      => raise syntax error
```

Figure 5. Expansion function

Recall the inference rules from Fig. 2 and Fig. 3. The corresponding derivation structures contain essentially the same information in a different form. The first two fields of all derivation variants are the terms before and after expansion. The remaining fields are specific to the variant. In the `mrule` variant, the remaining field contains the derivation of the macro's result. In the `lambda` variant, the third field contains the new formal parameters and body expression after renaming, and the final field contains the derivation that represents the expansion of the renamed body expression. In the `if` variant, the three additional fields are the derivations for the three `if` subexpressions. The phase and environment parameters are not stored explicitly in the derivation structures, but they can be reconstructed for any subderivation from its context.

Creating structured data from unstructured sequences is a parsing problem. By inspecting the order of calls to `emit-event`, recursive calls to `expand-term`, and calls to other auxiliary functions, it is possible to specify a grammar that describes the language of event streams from the instrumented expander. Figure 9 shows such a grammar. By convention, non-terminal names start with upper-case letters and terminal names start with lower-case letters.

The `ExpandTerm` non-terminal describes the events generated by the `expand-term` function (Fig. 5) in expanding a term, whether it is the full program or a subterm. It has two variants: one for macros and one for primitive syntax. The `Primitive` non-terminal has a variant for each primitive syntactic form, and the productions

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

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

expand-variable(id, env, phase) =
  let id2 = resolve(id)
  emit-event("variable", id2)
  return id2

expand-primitive-let-syntax(term, env, phase)
  emit-event("primitive-let-syntax", term)
  case term of
    (kw ([lhs rhs] ...) body)
      where each lhs is a distinct identifier
      => let lhss = (lhs ...)
         let rhss = (rhs ...)
         let lhss2 = freshnames(lhss)
         let body2 = rename(body, lhss, lhss2)
         emit-event("rename", lhss2 body2)
         let rhss2 =
           for each rhs in rhss:
             emit-event("next")
             expand-term(rhs, env, phase+1)
         let transformers =
           for each rhs2 in rhss2:
             eval(rhs2, env)
         emit-event("next-part")
         let env2 =
           extend-env(env, lhss2, transformers, phase)
         let body3 = expand-term(body2, env2, phase)
         return body3
    else => raise syntax error
```

Figure 6. Expansion functions for primitives and macros

`visit`	:	$Syntax$
`return`	:	$Syntax$
`enter-macro`	:	$Syntax$
`exit-macro`	:	$Syntax$
`macro-pre`	:	$Syntax$
`macro-post`	:	$Syntax$
`enter-primitive`	:	$Syntax$
`exit-primitive`	:	$Syntax$
`rename`	:	$Syntax \times Syntax$
`next`	:	()
`next-part`	:	()
`primitive-lambda`	:	()
`primitive-if`	:	()
`primitive-let-syntax`	:	()

Figure 7. Selected primitive events and the data they carry

for each primitive reflect the structure of the primitive expander functions from Fig. 6.

The macro debugger parses event streams into derivations according to this grammar. After all, a parse tree is the derivation proving that a sentence is in a language.[8] The action routines for the parser simply combine the data carried by the non-terminals—that is, the events—and the derivations constructed by the recursive occurrences of `ExpandTerm` into the appropriate derivation structures. There is one variant of the derivation datatype for each inference rule in the natural semantics (see Fig. 8).

5.4 Handling syntax errors

Both the derivation datatype from Fig. 8 and the grammar fragment in Fig. 9 describe only successful macro expansions, but a macro debugger must also deal with syntax errors. Handling such errors involves two additions to our framework:

1. new variants of the derivation datatype to represent interrupted expansions; after all, a natural semantics usually does not cope with errors
2. representation of syntax errors in the event stream, and additional grammar productions to recognize the new kinds of event streams

5.4.1 Derivations representing errors

A syntax error affects expansion in two ways:

1. The primitive expander function or macro transformer raises an error and aborts. We call this the *direct occurrence* of the error.
2. Every primitive expander in its context is interrupted, having completed some but not all of its work.

It is useful to represent these two cases differently. Figure 10 describes the extended derivation datatype.

For example, consider the expansion of this term:

```
(if x (lambda y) z)
```

The form of the `if` expression is correct; `expand-primitive-if` expands its subterms in order. When the expander encounters the `lambda` form, it calls the `expand-primitive-lambda` function, which rejects the form of the `lambda` expression and raises a syntax error. We represent the failed expansion of the `lambda` expression by wrapping a `prim:lambda` node in an `error-wrapper` node. The error wrapper also includes the syntax error raised.

That failure prevents `expand-primitive-if` from expanding the third subexpression and constructing a result term—but it did successfully complete the expansion of its first subterm. We represent the interrupted expansion of the `if` expression by wrapping the partially initialized `prim:if` node with an `interrupted-wrapper`. The interrupted wrapper also contains a tag that indicates that the underlying `prim:if` derivation has a complete derivation for its first subterm, it has an interrupted derivation for its second subterm, and it is missing the derivation for its third subterm.

5.4.2 The Error-handling Grammar

When an expansion fails, because either a primitive expander function or a macro transformer raises an exception, the macro expander places that exception at the end of the event stream as an `error` event. The event stream for the bad syntax example in Sec. 5.4.1 is:

```
visit enter-prim prim-if
  next visit enter-prim variable exit-prim return
  next visit enter-prim prim-lambda error
```

To recognize the event streams of failed expansions, we extend the grammar in the following way: for each non-terminal representing *successful* event streams, we add a new non-terminal that represents interrupted event streams. We call this the *interrupted* non-terminal, and by convention we name it by suffixing the original non-terminal name with "`/Error`." This interruption can take the form of an `error` event concerning the current rule or an interruption in the processing of a subterm.

Figure 11 shows two examples of these new productions.[9] The first variant of each production represents the case of a direct error, and the remaining variants represent the cases of errors in the expansions of subterms. The action routines for the first sort of error uses `error-wrapper`, and those for the second sort use `interrupted-wrapper`.

Finally, we change the start symbol to a new non-terminal called `Expansion` with two variants: a successful expansion `ExpandTerm` or an unsuccessful expansion `ExpandTerm/Error`.

The error-handling grammar is roughly twice the size of the original grammar. Furthermore, the new productions and action routines share a great deal of structure with the original productions and action routines. We therefore create this grammar automatically from annotations rather than manually adding the error-handling productions and action routines. The annotations specify positions for potential errors during the expansion process and potentially interrupted subexpansions. They come in two flavors: The first is the site of a potential error, written `(! tag)`, where tag is a symbol describing the error site. The second is a non-terminal that may be interrupted, written `(? NT tag)`, where `NT` is the non-terminal.

Figure 12 gives the definitions of the `PrimitiveLambda` and `PrimitiveIf` non-terminals from the annotated grammar. From these annotated definitions we produce the definitions of both the successful and interrupted non-terminals from Fig. 9 and Fig. 11, respectively.

The elaboration of the annotated grammar involves splitting every production alternate containing an error annotation into its successful and unsuccessful parts. This splitting captures the meaning of the error annotations:

- A potential error is either realized as an error that ends the event stream, or no error occurs and the event stream continues normally.

[8] Parser generators are widely available; we used the PLT Scheme parser generator macro [23].

[9] Figure 11 also shows that we rely on the delayed commitment to a particular production possible with LR parsers.

```
Derivation  ::=  (make-mrule Syntax Syntax Derivation)
             |   (make-prim:if Syntax Syntax Derivation Derivation Derivation)
             |   (make-prim:lambda Syntax Syntax Renaming Derivation)
             |   (make-prim:let-syntax Syntax Syntax Renaming DerivationList Derivation)
             |   ...
Renaming    ::=  Syntax × Syntax
```

Figure 8. Derivation structures

```
ExpandTerm ::= visit enter-macro macro-pre macro-post exit-macro ExpandTerm
             | visit enter-primitive Primitive exit-primitive return

Primitive ::= PrimitiveLambda
            | PrimitiveIf
            | PrimitiveApp
            | PrimitiveLetSyntax
            | ...

PrimitiveLambda ::= primitive-lambda rename ExpandTerm

PrimitiveIf ::= primitive-if next Expandterm next ExpandTerm next ExpandTerm

PrimitiveLetSyntax ::= primitive-let-syntax rename NextExpandTerms next-part ExpandTerm

NextExpandTerms ::= ε
                  | next ExpandTerm NextExpandTerms
```

Figure 9. Grammar of event streams

```
Derivation  ::=  ...
             |   (make-error-wrapper Symbol Exception Derivation)
             |   (make-interrupted-wrapper Symbol Derivation)
```

Figure 10. Extended derivation datatype

```
PrimitiveLambda/Error ::= primitive-lambda error
                        | primitive-lambda renames ExpandTerm/Error

PrimitiveIf/Error ::= primitive-if error
                    | primitive-if next ExpandTerm/Error
                    | primitive-if next ExpandTerm next ExpandTerm/Error
                    | primitive-if next ExpandTerm next ExpandTerm next ExpandTerm/Error
```

Figure 11. Grammar for interrupted primitives

```
PrimitiveLambda ::= primitive-lambda (! 'malformed) renames (? ExpandTerm)

PrimitiveIf ::= primitive-if (! 'malformed) next (? ExpandTerm 'test) next (? ExpandTerm 'then)
                next (? ExpandTerm 'else)
```

Figure 12. Grammar with error annotations

- A potentially interrupted subexpansion is either interrupted and ends the event stream, or it completes successfully and the event stream continues normally.

Naturally, we use a macro to elaborate the annotated grammar into the error-handling grammar. This is a nontrivial macro, and we made mistakes, but we were able to debug our mistakes using an earlier version of the macro stepper itself.

5.5 The Middle End: Hiding Macros

Once the back end has created a derivation structure, the macro debugger processes it with the user-specified macro hiding policy to get a new derivation structure. The user can change the policy many times during the debugging session. The macro debugger retains the original derivation, so updating the display involves only redoing the tree surgery and re-running the front end, which produces the rewriting sequence.

Conceptually, a macro hiding policy is simply a predicate on macro derivations. In practice, the user of the debugger controls the macro hiding policy by designating modules and particular macros as opaque or transparent. The debugger also provides named collections of modules, such as "mzscheme primitives," that can be hidden as a group. Policies may also 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.

We refer to the original derivation as the true derivation, and the one produced by applying the macro policy as the synthetic derivation. The datatype of synthetic derivations 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 middle end constructs the synthetic derivation by walking the true derivation and applying the policy predicate to every macro step. When the policy judges a macro step opaque, the debugger hides that step and the derivation for the macro's result, replacing the macro node with a synthetic node. The debugger then searches for *expanded subterms* within the hidden derivation. While searching through the subterms, it keeps track of the path through the opaque term to the subterms. When it finds an expanded subterm, it adds the derivation for the subterm's expansion, together with the path to that subterm, to the synthetic node.

Figure 13 illustrates one step of the macro hiding process. Suppose that the expansion of `let/cc` is marked as hidden. The debugger searches through the hidden derivation for subderivations corresponding to subterms of the opaque term. In the example from the figure, there is no subderivation for `k`, but there is a subderivation for $e1$. The macro hiding pass produces a new derivation with a *synthetic node* containing the derivation of $e1$ and the *path* to $e1$ in the original term. In this case, $e1$ can be reached in the term `(let/cc k` $e1$`)` through the path `(-- -- [])`.

If the expansion of $e1$ involves other opaque macros, then the debugger processes the derivation of $e1$ and places its corresponding synthetic derivation in the list of subterm derivations instead.

As a side benefit, macro hiding enables the debugger 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 debugger reports an error to the programmer.

Engineering note 1: Macro hiding is complicated slightly by the presence of renaming steps. When searching for derivations for subterms, if the macro hider encounters a renaming step, it must also search for derivations for any subterms of the renamed term that correspond to subterms of the original term.

Engineering note 2: Performing macro hiding on the full language is additionally complicated by internal definition blocks. PLT Scheme partially expands the contents of a block to expose internal definitions, then transforms the block into a `letrec` expression and finishes handling the block by expanding the intermediate `letrec` expression.[10] Connecting the two passes of expansion for a particular term poses significant engineering problems to the construction of the debugger.

5.6 The Front End: Rewriting Steps

Programmers think of macro expansion as a term rewriting process, where macro uses are the redexes and primitive syntactic forms are the contexts. The front end of the debugger displays the process of macro expansion as a reduction sequence. More precisely, the debugger displays one rewriting step at a time, where each step consists of the term before the step and the term after the step, separated by an explanatory note.

The macro stepper produces the rewriting steps from the derivation produced by the middle end, which contains three sorts of derivation node. A macro step (`mrule`) node corresponds to a rewriting step, followed of course by the steps generated by the derivation for the macro's result. A primitive node generates rewriting steps for the renaming of bound variables, and it also generates rewriting steps from the expansion of its subterms. These rewriting steps occur in the context of the primitive form, with all of the previous subterms replaced with the results of their expansions.

For example, given subderivations `test`, `then`, and `else` for the three subterms of an `if` expression, we can generate the reduction sequence for the entire expression. We simply generate all the reductions for the first derivation and plug them into the original context. We build the next context by filling the first hole with the expanded version of the first subterm, and so on.

Opaque macros also act as expansion contexts. The synthetic nodes that represent opaque macros contain derivations paired with paths into the macro use's term. The paths provide the location of the holes for the contexts, and the debugger generates steps using the subderivations just as for primitive forms.

6. Conclusion

Despite the ever increasing complexity of Scheme's syntactic abstraction system, Scheme implementations have failed to provide adequate tools for stepping through macro expansions. Beyond the technical challenges that we have surmounted to implement our macro stepper, there are additional theoretical challenges in proving its correctness. Macro expansion is a complex process that thus far lacks a simple reduction semantics. We have therefore based our macro stepper on a natural semantics, with an ad hoc translation of derivations to reduction sequences.

First experiences with the alpha release of the debugger suggest that it is a highly useful tool, both for experts and novice macro developers. We intend to release the debugger to the wider Scheme community soon and expect to refine it based on the community's feedback.

Acknowledgments

We thank Matthew Flatt for his help in instrumenting the PLT Scheme macro expander.

References

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

[10] PLT Scheme macros are also allowed to partially expand their subterms and include the results in their output.

True derivation (before macro hiding):

$$\frac{p, E \vdash \texttt{(let/cc k } e1\texttt{)} \rightarrow \texttt{(call/cc (lambda (k) } e1\texttt{))} \qquad \dfrac{\dfrac{\Delta}{p, E' \vdash e1 \Downarrow e1'}}{p, E \vdash \texttt{(call/cc (lambda (k) } e1\texttt{))} \Downarrow expr'}}{p, E \vdash \texttt{(let/cc k } e1\texttt{))} \Downarrow expr'}$$

Synthetic derivation (after macro hiding):

$$\frac{\left\langle \texttt{(-- -- [])}, \dfrac{\Delta}{p, E' \vdash e1 \Downarrow e1'} \right\rangle}{p, E \vdash \texttt{(let/cc k } e1\texttt{)} \Downarrow_h \texttt{(let/cc k } e1'\texttt{)}}$$

Figure 13. Macro hiding

[2] Ana Bove and Laura Arbilla. A confluent calculus of macro expansion and evaluation. In *Proc. 1992 ACM Conference on LISP and Functional Programming*, pages 278–287, 1992.

[3] 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.

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

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

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

[13] 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.

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

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

[16] Martin Gasbichler. *Fully-parameterized, First-class Modules with Hygienic Macros*. PhD thesis, Eberhard-Karls-Universität Tübingen, February 2006.

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

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

[25] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. In *Proc. Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 201–212, 2004.

[26] 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.

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

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

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

[30] 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.

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

[32] Andrew Wright and Bruce Duba. Pattern matching for scheme, 1995.