Debugging Hygienic Macros [1]

Ryan Culpepper, Matthias Felleisen
College of Computer and Information Science, Northeastern University, Boston MA 02115, USA

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 lexical scoping, and advanced Scheme macro systems handle other important properties such as source location. Using such macros, Scheme programmers now routinely develop the ultimate abstraction: embedded domain-specific programming languages.

Unfortunately, a typical Scheme programming environment provides 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. This debugger 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 therefore 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. In some systems, macros can share information, perform sophisticated computation, and detect and report errors, all at compile time. As some Scheme implementers have put it, macros have become a true API to the front-end of the compiler.

Macro definitions are a uniform mechanism for extending the expression and definition forms of the language. Programmers can mix extensions freely using Scheme's scoping mechanisms, such as modules and local bindings, rather than needing preprocessors, specialized compilers, or program generators. If attached to an expressive language [10] macros suffice to implement many general-purpose abstraction mechanisms as

[1] This paper revises and extends a conference publication [5]. This work was partially supported by several NSF grants.

libraries. For example, programmers have used macros to extend Scheme with constructs for pattern matching [37], relations in the spirit of Prolog [9,18,29], extensible looping constructs [8,28], class systems [1,26] and component systems [6,15,35], 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. [25] have added a parser generator library to Scheme; Sarkar et al. [27] 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 [14] have a core of a dozen or so syntactic constructs but appear to implement a language as rich in features as Common Lisp [32].

To support these increasingly ambitious applications, macro systems have had to evolve, too, because true syntactic abstractions must be indistinguishable from features directly supported by the implementation's compiler or interpreter. Historically, macros had humble beginnings in Lisp systems, where they were compile-time functions over program fragments, usually plain S-expressions. Unfortunately, these simplistic macros do not 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 [22], referential transparency [4,7], and phase separation [13] and implemented macro systems guaranteeing those properties. Now macros manipulate program representations that carry information about lexical scope; affecting scope takes deliberate effort on the part of the macro writer.

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 the macro system. While experts always love to come to their aid, these questions demonstrate that programmers need tools for exploring macro expansion just as they need tools for exploring ordinary program execution.

In this paper, we present the first macro stepper for Scheme. In the next section we discuss the properties of macro programming that inform the design of our debugger, and in subsequent sections we discuss the implementation of the stepper, describe its foundations with a formal model, and prove those foundations correct.

2. Design Considerations for a Macro Debugger

A debugger must be tailored to its target language. 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 variable bindings resulting from unification or memory locations and whether the programmer explores a tree-shaped logical derivation or steps through line-sized statements.

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

(i) Macros are rewriting rules.
(ii) Macros respect lexical scoping.
(iii) Macros define layered abstractions.

This section elaborates these three principles and their influence on the design of our debugger.

2.1. *Macros are rewriting rules*

Intuitively, macro definitions specify rewriting rules, and macro expansion is the process of applying these rules to the program until all of the macros have been eliminated and the program uses only the primitive forms of the language.

The expander does not rewrite terms freely. Rather, it only looks at terms in *expansion contexts* such as at the top-level and in certain positions within primitive syntactic forms. The body of a `lambda`-form, for example, is an expansion position, but its formal parameter list is not. Of course, a term that is not in an expansion context may eventually be placed into an expansion context when enclosing macros are rewritten. We reserve the words *expression* and *definition* for terms that ultimately occur in an expansion context. The expression structure of a program, then, is gradually discovered during macro expansion.

Terms that are not expressions may be identifiers or constants, or they may be part of a macro's structured syntax. Macros commonly perform case analysis on their structured syntax, extracting subexpressions and other meaningful syntax such as variable binding lists and embedding these fragments in a new expression. Modern macro systems provide convenient notations for this case analysis and transformation of syntax.

Two ways of writing macro definitions are defined in the current Scheme report (R^6RS) [30]. One is a limited pattern-based macro facility [23] called `syntax-rules`. The other allows programmers to compute syntax transformations using the full power of Scheme plus a pattern-matching form called `syntax-case` [7].

A macro definition using `syntax-rules` 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 `literal`s determine which identifiers in the pattern are matched for equality rather than treated as pattern variables.

For example, here is the definition of a macro that helps a programmer protect shared resources:

```
;; (with-lock expr) acquires a lock, evaluates the expression, and releases the lock.
(define-syntax with-lock
  (syntax-rules ()
    [(with-lock body)
     (dynamic-wind
       (lambda () (acquire-the-lock))
       (lambda () body)
       (lambda () (release-the-lock)))]))
```

By using such a macro, the programmer ensures that the enclosed expression is executed only in the proper context: with the lock acquired. In addition, the programmer can be sure that the function calls to acquire and to release the lock are balanced (for all possible control flows).

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

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

The expander rewrites the procedure definition to the following:

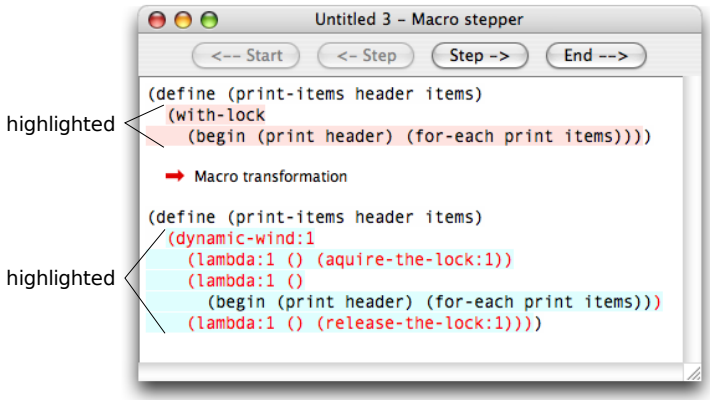
```
(define (print-items header items)
  (dynamic-wind
    (lambda () (acquire-the-lock))
    (lambda () (begin (print header) (for-each print items)))
    (lambda () (release-the-lock))))
```

The result 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.

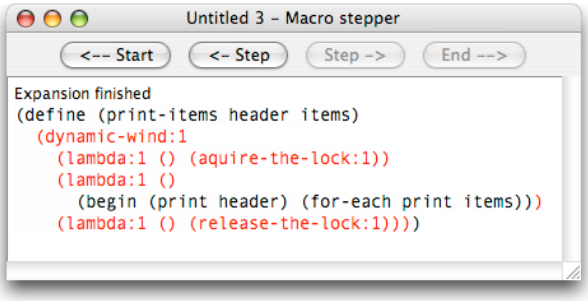
The tool's graphical user interface permits programmers to go back and forth in an expansion and also to skip to the beginning or end of the rewriting sequence. The ideas for this interface have been borrowed from the run-time stepper for PLT Scheme and similar steppers for Lisp-like languages [3,24,34].

Here is the program from above in the macro stepper:



The stepper shows the original code and the result of applying the `with-lock` rewriting rule. The figure indicates the highlighted regions: the macro use and its expansion.

If the programmer navigates forward, the stepper indicates that expansion is complete and shows the final program:



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 [4,22]—to respect lexical scoping according to the following two principles: [2]

(i) References introduced by a macro (not taken from the macro's arguments) refer to the binding occurrence apparent at the site of the macro's definition.

(ii) 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.

[2] Unless the programmer explicitly programs such a violation using the hygiene-breaking features of a system such as the `syntax-case` system.

The gist of these requirements is that macros act "like closures" at compile time. For example, the writer of `with-lock` can be certain of the meaning of `acquire-the-lock`, `release-the-lock`, and even `lambda`, regardless of whether some of those names have different bindings where the macro is used. Similarly, a macro can create a temporary variable binding without that binding capturing references in expressions that the macro receives as arguments.

Thinking of macro expansion in terms of substitution provides additional insight into the problem and its solution. A single macro expansion step involves two sorts of substitution and thus there are two sorts 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 might not be 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 capture any free occurrences of `x` in the expression `e`.[3] On the other hand, if `mangle` acts like `begin`, then the occurrence of `x` is a reference to a name 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 the role `x` plays and consequently which hygiene principle governs its behavior. Thus it must delay the treatment of `x` 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, the binding structure of a program is gradually discovered during macro expansion. The precise structure is not known until the program is completely expanded, although partial information is known earlier.

Hygienic macro systems usually implement the proper lexical scoping by annotating their program representations with timestamps, or *marks*. An identifier's mark, if one is present, indicates which macro rewriting step introduced it. Even though we customarily represent marks as numbers, we only care about their identity, not the order in which they occur. 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)))
```

[3] There are devious macros, related to the "ill-behaved macros" discussed in Sect. 2.3, that challenge conventional intuitions about hygiene by searching through expressions looking for identifiers to capture. Regardless, the hygiene principles we describe give a useful, if imperfect, guide to the scoping properties of macros.

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

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

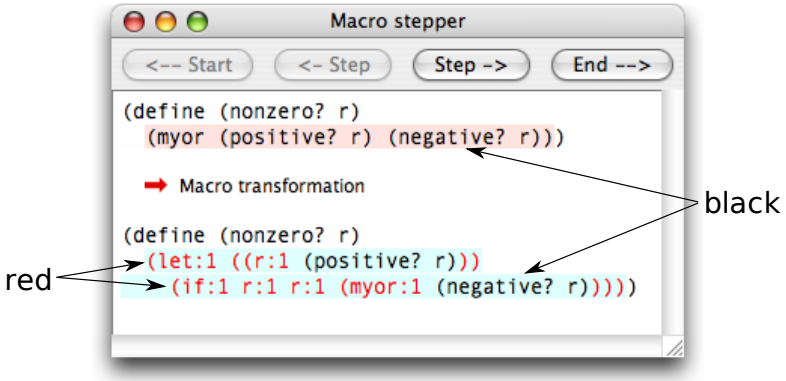
Instead, the macro expander adds a mark to every macro-introduced identifier. Using our notation for marks, the Scheme expander gives us this term:

```
(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 such as `let:1` does not occur in the scope of a binding of the same marked identifier, 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?`.

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 from which macro expansion step every subterm originated by rendering terms from different steps in different colors. If there are too many steps to give each one a distinguishable color, then the stepper adds numeric suffixes to marked identifiers, such as `let:1`.[4]

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



The introduced binding has a mark, and the mark matches the introduced references. The marked identifiers are rendered in red and the unmarked identifiers are rendered in black.

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. It does not yet know, however, 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.

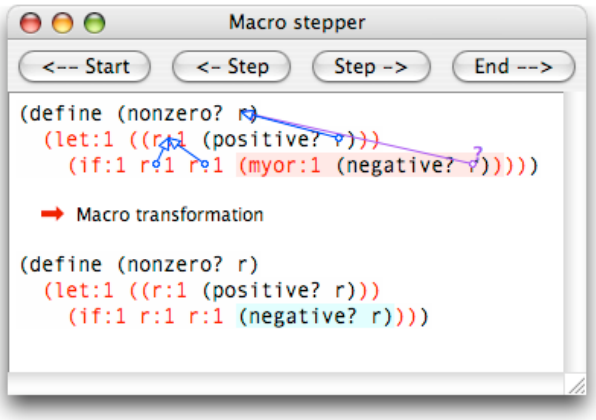
The macro stepper displays its knowledge of the program's binding structure through binding arrows, inspired by the arrows of DrScheme's Check Syntax tool [11]. When the macro expander uncovers a binding, the macro stepper draws a *tentative* binding arrow from the binding occurrence to every identifier that may possibly be bound by it.[5] The stepper annotates tentative binding arrows with a "?" symbol at the bound

[4] Because of the colors, it is never unclear whether a suffix is part of the actual identifier or a macro stepper annotation.

[5] The macro stepper cannot predict references created using hygiene-breaking facilities like `datum->syntax`, of course, but once they are created the macro stepper does identify them as potential references.

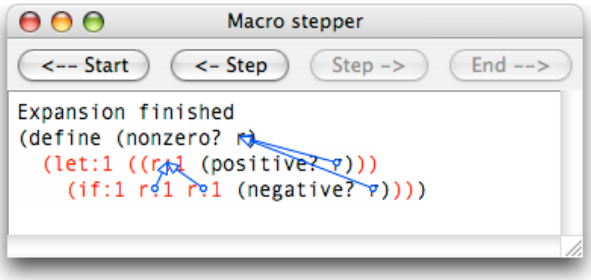
end. When the macro expander finally resolves a variable to a specific binding, the arrow becomes *definite* and the question mark disappears. Alternatively, macro expansion may uncover a closer binding occurrence of the same name, in which case the target of the tentative arrow changes.

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 still tentative 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 macros often manipulate the lexical scope in a subtle manner, the shape of an intermediate term alone may not reveal an error. The program seems correct, but variables do not get bound or get bound in the wrong place. In these cases, the stepper's visual presentation of lexical binding is critical.

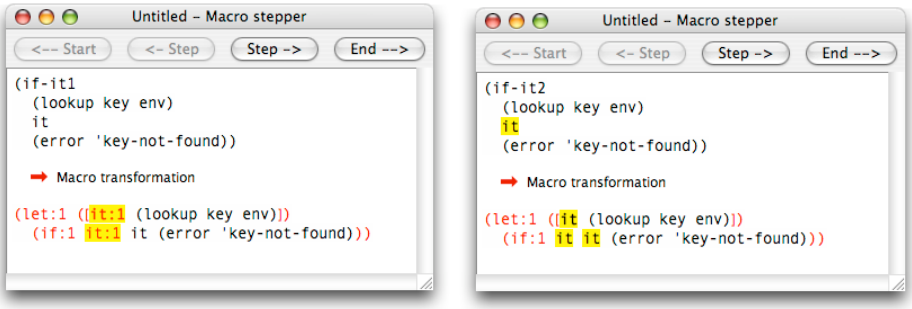
To illustrate this point, consider the creation of an `if-it` macro, a variant of `if` that binds the variable named `it` to the result of the test expression for the two branches:

```
(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 #'if-it2 'it)])
       #'(let ([it test]) (if it then else)))]))
```

These two attempts at the macro are written in the `syntax-case` macro system [7,30]. Their right-hand sides can contain Scheme code to perform the macro transformation, including syntax-manipulating procedures such as `datum->syntax` 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 easily tell:



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` 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 lexical tokens and syntax trees during macro expansion. The programmer can view those properties, on demand, in a separate part of the macro stepper.

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.

For example, PLT Scheme's new trait library [16] is implemented as a macro that translates trait declarations to mixin declarations [17], i.e., as functions from classes to classes. Mixins 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, following Steele's original precedent [31].

Such layers of linguistic abstraction are common in languages that support macros, and they demand special support from debuggers. 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 syntactic 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*. Designating a macro as opaque effectively removes its rewriting rules and adds expansion contexts, as if it were truly a primitive of the language.

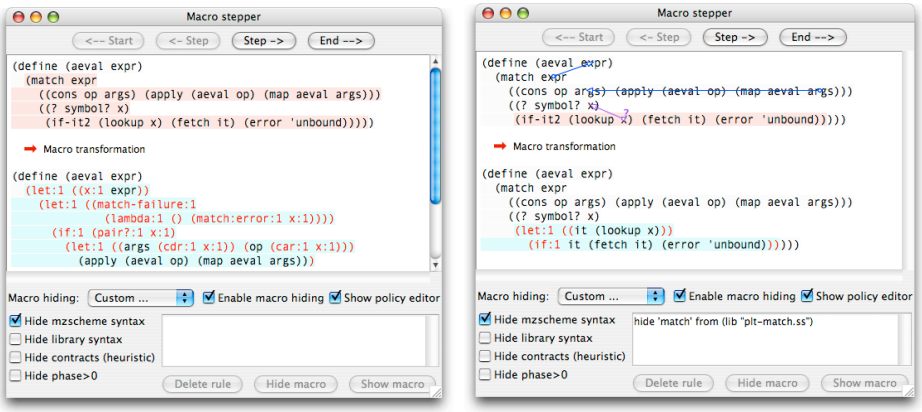


Fig. 1. Example of macro hiding

The programmer can modify the 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.

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 (aeval op) (map aeval 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.

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 allows programmers to specify the opacity of macros at two levels. The coarse level includes two broad classes of macros: those belonging to the base language [14] and those defined in standard libraries; hiding these allows the programmer to focus on the behavior of their own macros, which are more likely to contain bugs than library macros. 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 discovered only two additional useful policies. The first hides a certain class of macros that implement run-time contract checking [12]. The second hides the expansion of macros used in the right-hand sides of macro definitions, e.g., uses of `syntax-rules` and `syntax-case`, which are actually macros. We have added support for these cases in an ad hoc manner.

Because of macro hiding, our debugger presents steps that actually never happen and it presents terms that the expander actually never produces. In most cases these spurious terms are plausible and instructive, but for some macros, macro hiding produces terms that are not equivalent to the original program. These faulty terms generally arise when a macro inspects one of its subexpressions or otherwise treats it as more than just an expression. In a sense, such macros violate the expression abstraction by inspecting the expression's underlying representation.

Macro hiding relies on the expression abstraction to justify its reordering of expansion steps. One way of breaking the expression abstraction is to quote a subexpression, as in this example:

```
(define-syntax time
  (syntax-rules ()
    [(time e) (time-apply (quote e) (lambda () e))]))
```

If `time` is opaque, then macro hiding shows expansion in the context of the `time` macro. But the resulting terms do not preserve the meaning of the original term, because they report different quoted constants. Thus we regard this macro as ill-behaved *with respect to the expression abstraction.*

A more subtle way of breaking the expression abstraction is analyzing a subexpression or a subterm that is only sometimes used as an expression:

```
(define-syntax no-constants
  (syntax-rules (quote)
    [(no-constants (quote c)) (error "no constants please")]
    [(no-constants e) e]))
```

Here's the expansion of one use of `no-constants`:

$$\texttt{(no-constants (quasiquote x))} \Longrightarrow \texttt{(quasiquote x)} \Longrightarrow \texttt{(quote x)}$$

With `no-constants` hidden, the expansion is this:

$$\texttt{(no-constants (quasiquote x))} \Longrightarrow \texttt{(no-constants (quote x))}$$

But if the final term were evaluated, it would result in an error rather than the expected constant `x`.

The macro debugger also cannot hide a macro that clones a subexpression. Since the subexpression occurs multiple times in the output, the macro debugger finds multiple expansion subsequences associated with the same context, the context of the subexpression in the original macro use. When it discovers the contexts are the same (or overlap), the macro stepper issues a warning and continues, showing that instance of the macro. This limitation is indirectly beneficial, since expression cloning is almost always an error in the macro definition.

In general, the expansions of a cloned expression may be different, so there is no good way to fit them into the same context. As a special case, however, duplicated variable references are allowed, since their expansions are trivial and no conflicts arise.

Note that not every duplication of a subterm is a duplication of an expression. For example, consider the following macro:

```
(define-syntax push!
  (syntax-rules ()
    [(push! s e)
     (set! s (cons e s))]))
```

The macro duplicates `s`. But since `s` appears as the target of an assignment, `s` must be a variable name, and that occurrence is not an expression. The other occurrence of `s` is as a variable reference expression, which has no expansion steps. So this macro is regarded as well-behaved by macro hiding. If, however, `set!` allowed targets that included subexpressions (as Lisp's `setf` does, for example), then this `push!` macro would be capable of duplicating subexpressions.

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.

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 does not 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 expected path.

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 nontrivial contexts. Such errors can occur in macros that depend on bindings introduced by other macros.

In fact, `expand` suffers from one more problem: it reveals too much. 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 a special form is a primitive form or a macro is irrelevant to the programmer, but 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, even those the programmer is not concerned with. Using `expand-only` avoids this problem. but it otherwise exhibits the same flaws as `expand`. Our idea of macro hiding can be seen as a generalization of `expand-only`.

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 [36].

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 describe the parser that reconstructs the expander call tree. We explain how the macro hiding algorithm—the middle end—transforms the IR tree by detecting the contexts of hidden macros. Finally, we discuss 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 procedures. PLT Scheme uses the `syntax-case` algorithm [7] for macro expansion, and we describe our implementation in the context of that algorithm, but our implementation strategy applies in principle to other macro expanders.

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

The `expand-term` procedure 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-canceling marking is an efficient implementation of timestamps due to Dybvig et al. [7].

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

4.2. *Instrumenting the Expander*

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

```
expand-term(term, env) =
  emit-event(Visit, term)
  case term of
    (kw . _)
      when 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 . _)
      when lookup(resolve(kw), env)
             = ("primitive", expander)
      => let term2 = expander(term, env)
         emit-event(Return(term2))
         return term2
    id
      when lookup(resolve(id), env)
             = "variable"
      => emit-event(Variable)
         emit-event(Return(term))
         return term
    else
      => emit-event(Error)
         raise syntax error
```

Fig. 2. Expansion procedure

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 to which the bound names are renamed.

4.3. *Reconstruction*

The back end of the macro stepper consumes the sequence of low-level events from the instrumented macro expander, parsing it 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.

```
expand-prim-lambda(term, env) =
  emit-event(PrimLambda)
  case term of
    (kw formals body)
      when 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
    (kw 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)
      when 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
```

Fig. 3. Expansion procedures for primitives and macros

$$
\begin{aligned}
\text{Events } \mathit{event} \;::=\; & \mathsf{Visit}(\mathit{expr}) \mid \mathsf{Return}(\mathit{expr}) \\
& \mid \mathsf{EnterMacro} \mid \mathsf{Mark}(\mathit{mark}) \mid \mathsf{ExitMacro}(\mathit{expr}) \\
& \mid \mathsf{PrimLambda} \mid \mathsf{PrimIf} \mid \mathsf{PrimApp} \mid \mathsf{PrimLetSyntax} \\
& \mid \mathsf{Variable} \mid \mathsf{Rename}(\mathit{symbols}) \\
\text{Traces } T \;::=\; & \text{list of events}
\end{aligned}
$$

Fig. 4. Expansion events

$$
\begin{array}{rcl}
\text{IRTrees } ir & ::= & \mathsf{MacroNode}(expr, expr, mark, expr, ir) \\
& | & \mathsf{VariableNode}(expr, expr) \\
& | & \mathsf{LambdaNode}(expr, expr, symbols, ir) \\
& | & \mathsf{IfNode}(expr, expr, ir, ir, ir) \\
& | & \mathsf{AppNode}(expr, expr, irlist) \\
& | & \mathsf{LetSyntaxNode}(expr, expr, symbols, ir)
\end{array}
$$

Fig. 5. Intermediate representation structures

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

Fig. 6. Grammar of event streams

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

Fig. 7. Parser

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 that expansion. The functions `initialterm` and `finalterm` serve as accessors for those fields. The remaining fields of each variant are determined by the expander's behavior on that term variant. The MacroNode variant contains a field for the mark 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: *Expand*, which corresponds to the events produced by an invocation of `expand-term`, and an auxiliary *Expand**. 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 *Expand* 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 pseudocode for the parser function.[6] The recursive structure of the parser is identical to the recursive structure of the expander. 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.

4.4. *Handling syntax errors*

The macro stepper also handles expansions that halt because of syntax errors. Handling such errors requires extending both the intermediate representation and the parser.

We extend the IR with a notion of an *incomplete* node. We add fields at every position where an error can occur. For example, the `if` keyword checks to make sure it has exactly three subexpressions before starting to expand the test expression, so IfNode gets one additional field:

$$\mathsf{IfNode}(\mathit{expr}, \mathit{expr}, \mathit{optexn}, \mathit{ir}, \mathit{ir}, \mathit{ir})$$

If an *optexn* field holds an exception value, then all the fields that follow it are absent. Furthermore, if an *ir* field holds an incomplete node, then the fields that follow it are absent.

We extend the parser with a new nonterminal, *Fail*, that represents the event streams of failed expansions:

$$\begin{aligned} \mathit{Fail} ::= {} & \mathsf{Visit\ PrimLambda\ Error} \\ | \ & \mathsf{Visit\ PrimLambda\ Rename}\ \mathit{Fail} \\ | \ & \ldots \end{aligned}$$

As with *Expand*, the alternatives can be read off the instrumented code, picking only the cases where an error is raised or a recursive call fails. The start symbol of the grammar goes to either *Expand* or *Fail*.

The error-handling grammar is roughly twice the size of the original grammar. Furthermore, the new alternatives and action routines share a great deal of structure with the original alternatives and action routines. We therefore create this grammar automatically from annotations rather than adding the error-handling parts by hand.

4.5. *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

[6] In our implementation of the macro stepper, the parser is constructed from the grammar using PLT Scheme's macro-based parser generator library [25].

```
hide : IR -> IR
hide(MacroNode(e1,e2,mark,emid,irk)) =
  if should-hide(e1)
  then let table = table-of-proper-subterms(e1)
           subs = seek(MacroNode(e1,e2,mark,emid,irk), table)
           e2' = substitute-trees-with-contexts(e1, subs)
       in SynthNode(e1, e2', subs)
  else let irk' = hide(irk)
       in MacroNode(e1, finalterm(irk'), mark, emid, irk')
hide(LambdaNode(e1,e2,newvars,irbody)) =
  let irbody' = hide(irbody)
      e2' = substitute-lambda-body(e2, finalterm(irbody'))
  in LambdaNode(e1, e2', newvars, irbody')
...
```

Fig. 8. hide function

```
seek : (IR, Table) -> list((path, term))
seek(ir, table) =
  if table contains initialterm(ir) at path
  then [(path, hide(ir))]
  else seek-within(ir, table)

seek-within : (IR, Table) -> list((path, term))
seek-within(MacroNode(e1, e2, mark, emid, irk), table) =
  let table' = adjust-for-marks(table, e1, mark, emid)
  in seek(irk, table')
seek-within(LambdaNode(e1, e2, newvars, irbody), table) =
  let table' = adjust-for-lambda-rename(table, e1, newvars)
  in seek(irbody, table')
seek-within(IfNode(e1, e2, irtest, irthen, irelse), table) =
  append(seek(irtest, table),
         seek(irthen, table),
         seek(irelse, table))
...
```

Fig. 9. seek function

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 IR node variant that does not correspond to any primitive syntactic form. This node contains a list of subterm expansions, where each subterm expansion consists of an IR tree and a path representing the context of the subterm in the node's original syntax.

The macro hiding algorithm is a traversal of the IR tree in two alternating modes, hide and seek. In hide mode, the algorithm looks for macro-rewriting nodes to hide. In seek mode, the algorithm is traversing the tree corresponding to a hidden macro's expansion, looking for subtrees to show; that is, subtrees corresponding to terms that exist in the macro occurrence to be hidden. When it finds such a subtree, it switches back to hide mode. The algorithm is parameterized over the hiding policy, which we denote as `should-hide(·)`.

Figure 8 shows the essence of the hide function, which consumes an IR tree and produces an IR tree. If the input is a macro node and the policy says to hide the macro, the stepper switches to seek mode to continue the traversal of the macro node. Seek mode returns a list of subtrees with contexts, and the hide function replaces the original macro node with a synthetic node containing the subtrees. When the hide function gets a primitive syntax node or a macro node that should not be hidden, it just recurs on the node's subtrees.

In addition to removing macro expansion nodes from the tree, the hide function must also recompute the final term for every node that it produces to account for hidden macros. This calculation involves the node's processed subtrees and their contexts, which are fixed for primitive nodes and calculated by the seek phase for synthetic nodes. The auxiliary function substitute-trees-with-contexts, which computes the final term of

a synthetic node, checks that all of the contexts it receives are disjoint. If contexts overlap (due to cloned subexpressions), the macro stepper shows the current node instead.

In seek mode (see Fig. 9), the algorithm is looking for 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 the seek function encounters a binding form, it must adjust the set of terms it is looking for to account for the renaming done. Suppose that we are processing the following expression that uses the macro `or`, which we would like to hide:

$$\texttt{(or X Y)} \Longrightarrow \texttt{(let ([t X]) (if t t Y))}$$

When the algorithm initially enters seek mode, the term to search is the original use of `or`. But `let` is a binding form and applies a renaming to its body, including the subexpression `Y`. So in the body of the `let` form, the algorithm should not be looking for the original `Y`, but for `Y` wrapped with the `let`'s renaming. The algorithm does this by updating the table whenever it passes through a binding primitive.

In fact, the algorithm tracks the identity of the syntax objects rather than comparing them for textual equality, so it does not confuse subterms of a hidden macro with terms arising later in the macro's expansion. Consequently, it also enforces a strict interpretation of "subexpression": if a macro takes apart a term and reassembles it, the macro hiding algorithm does not consider them the same term. We consider this the proper behavior: by analyzing and rebuilding an expression a macro takes responsibility for its contents. Of course, untouched subexpressions of an analyzed term are still recognized as the same.

Although it is not shown in the algorithm sketch, the debugger also collects renaming steps performed by the primitive binding forms and adds them to the synthetic nodes. This extra information allows the macro debugger to recognize binding positions within macros that act as binding forms. The front end uses the information to draw binding arrows correctly even when the primitive binding form isn't visible in the macro stepper.

4.6. *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 binding arrows and distinguish between tentative and definite references.

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.

Terms	*expr*	::=	*id*
		|	*datum*
		|	(*expr* $\cdots$ *expr*)
Identifiers	*id*, *kw*	::=	*s*
		|	mark(*id*, *mark*)
		|	subst(*id*, *id*, *s*)
Symbols	*s*	::=	countable set of names
Marks	*mark*	::=	countable set
Denotations	*d*	::=	`variable`
		|	⟨`primitive`, *symbol*⟩
		|	⟨`macro`, *transformer*⟩
Environments	*E*	:	Symbol → Denotation
Stores	*S*	:	sets of Symbols and Marks

Expansion relation $E \vdash expr, S \Downarrow expr, S$

Fig. 10. Domains and relations

5. Formal Model

The reliability of the macro stepper depends on the faithfulness of its reconstruction of the recursive expansion process. Rather than assign meaning directly to the instrumented expander functions, we model the behavior of the expander with a big-step semantics. We extend the usual model of macro expansion to include the event stream generated by our instrumentation. We then prove a correspondence between the IR trees produced by the back end's parser and the extended big-step derivations.

5.1. *Macro expansion*

Our model of macro expansion is an adaptation of the `syntax-case` model of Dybvig et al. [7].[7] We use their program representation and lexical scoping mechanism, but we adapt their formulation of expansion via recursive equations to a big-step semantics (similar to an earlier formulation by Clinger and Rees [4]). In the next section, we show how the big-step derivations correspond to the structures produced by the macro debugger's back end.

The expansion process operates on terms, using an environment to record the meanings of bound names and a store for the generation of fresh names and marks. The basic judgment thus has the shape

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

and says that the term *expr* fully macro expands into *expr*′ in the environment E, transforming the store S into S'. Figure 10 summarizes the domains and metavariables of our semantic framework.

To respect lexical scope, Scheme macro systems enrich their representations of programs with lexical scoping information and timestamps that record when identifiers are introduced. The representation used by the `syntax-case` system introduces two new variants of identifiers beyond simple symbols. One variant

[7] Ideally, we would like to use rewriting models like those of Bove and Arbilla [2] and Gasbichler [19], but we have found those models unsuitable as the specification of our debugger.

$$\begin{aligned}
\text{resolve}(s) &= s \\
\text{resolve}(\text{mark}(id, m)) &= \text{resolve}(id) \\
\text{resolve}(\text{subst}(id, id', s)) &= \begin{cases} s & \text{if } id =_{\text{bound}} id' \\ \text{resolve}(id) & \text{otherwise} \end{cases} \\
\text{marksof}(s) &= \emptyset \\
\text{marksof}(\text{mark}(id, mark)) &= \text{marksof}(id) \uplus \{mark\} \\
\text{marksof}(\text{subst}(id, id', s)) &= \text{marksof}(id)
\end{aligned}$$

$$\begin{aligned}
id_1 =_{\text{free}} id_2 &\text{ iff } \text{resolve}(id_1) = \text{resolve}(id_2) \\
id_1 =_{\text{bound}} id_2 &\text{ iff } id_1 =_{\text{free}} id_2 \text{ and } \text{marksof}(id_1) = \text{marksof}(id_2)
\end{aligned}$$

Fig. 11. Operations and relations on syntax

represents an identifier with a timestamp, or *mark* in `syntax-case` terminology. The other represents a delayed alpha-renaming; this operation is applied by binding forms in the macro expansion process.

The macro expander uses an environment, called the syntactic environment, to carry the meaning of identifiers. The environment maps a symbol to its denotation: the name of a primitive syntactic form, a macro with its rewriting rules, or the designator `variable`. 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.

Figure 11 defines the relevant operations on identifiers:

- resolve(id) applies all delayed alpha-renaming operations and produces the symbol for the binding that id refers to.
- marksof(id) is the set of marks that the macro expander has applied to the identifier; identifiers in the original program start out with no marks.
- $id_1 =_{\text{free}} id_2$ if id_1 and id_2 are equivalent as references; that is, if they both refer to the same binding.
- $id_1 =_{\text{bound}} id_2$ if id_1 and id_2 are equivalent for the purposes of creating new bindings; that is, if a binding of one would capture the other.

Dybvig et al. [7] explain this program representation in greater detail, especially the algebraic properties of marking and renaming.

The language of our model is similar to Scheme, but greatly simplified. It has only a few special forms, including the binding forms `lambda` and `let-syntax`; we omit constants and we give application an explicit keyword. For simplicity, we include only `syntax-rules` macros. Figures 12 and 13 lists the expansion rules for macros and primitive forms. Ignore the shaded parts of the semantics for now. Determining which rule applies to a term involves resolving its leading keyword and consulting the environment for its meaning.

A macro use is distinguished by a leading keyword that is mapped to transformation rules in the syntactic environment. The expander uses those rules to transform the macro use into a new term. We use the $\langle rules, expr \rangle \Downarrow_{tr} expr'$ judgment to indicate this process. We omit the definition of the transformation relation, since it does not affect the structure of our debugger. The expander marks the expression before transformation and then marks the transformation's result. Identifiers in the original term get the mark twice, and these marks cancel out (marks are self-canceling). Identifiers introduced by the rule's template get the mark only once, and the mark thus acts as a timestamp for the identifier's introduction. When a marked identifier is used in a binding construct, the alpha-renaming affects only identifiers with the same name *and* the same marks.

The `lambda` syntactic form generates a new name for each of its formal parameters and creates a new formals list and a new body term with the old formals mapped to the new names. 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 reassembles the `lambda` term with the new formals list and expanded body.

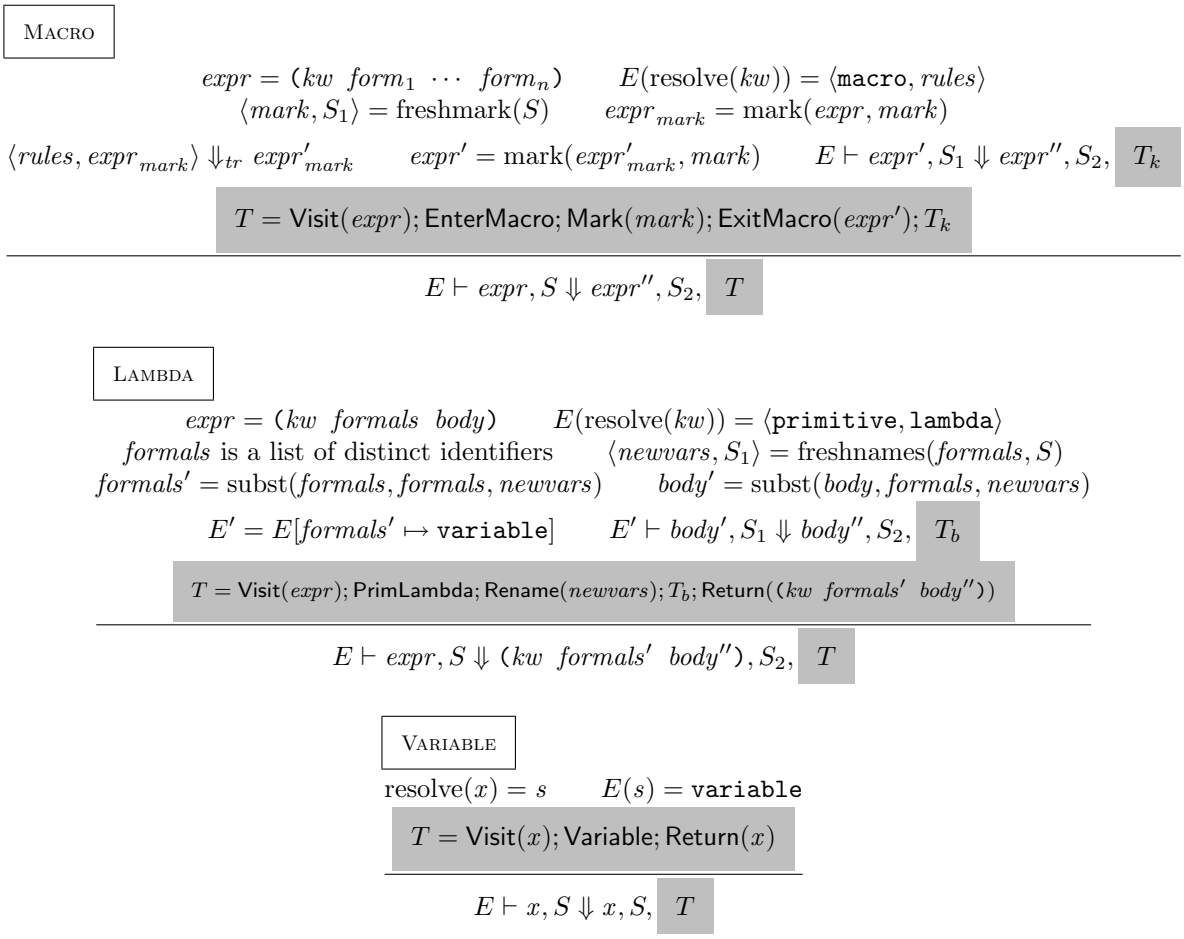


Fig. 12. Semantics of macro expansion with expansion events

When the macro expander encounters one of the `lambda`-bound variables in the body expression, it resolves the identifier to the symbol from the renaming step and discovers that the environment maps it to the `variable` designator (otherwise it is a misused macro or primitive name—an error). It then returns the original identifier.

The IF and APP (application) rules are simple; they just expand their subexpressions without extending the environment.

The `let-syntax` form works like `lambda` but binds `macro` transformers instead of simple `variable`s. Once the body has been fully expanded, there is no more need for the macro bindings, so it does not recreate the `let-syntax` expression.

5.2. *Correctness*

We model the behavior of the instrumented expander with a small syntactic change to the big-step semantics. The shaded parts of Figs. 12 and 13 represent the synthesis of the event sequence. Given the semantics with event sequences, we can prove a correspondence between the IR trees produced by the back end's parser and the extended big-step derivations.

Given a derivation Δ, we write $[\![\Delta]\!]$ for its representation as an IR tree. The mapping is the obvious interpretation of the IR trees, discussed informally in Sect. 4.3. For example, if Δ is a derivation for $E \vdash expr, S \Downarrow expr', S'$ with LAMBDA as the final inference rule, *newvars* as the fresh variable names, and Δ_b as

IF

$$\frac{\begin{array}{c} expr = \texttt{(}kw\ \ test\ \ then\ \ else\texttt{)} \qquad E(\text{resolve}(kw)) = \langle \texttt{primitive}, \texttt{if} \rangle \\ E \vdash test, S \Downarrow test', S_1, \boxed{T_1} \qquad E \vdash then, S_1 \Downarrow then', S_2, \boxed{T_2} \qquad E \vdash else, S_2 \Downarrow else', S_3, \boxed{T_3} \\ \boxed{T = \mathsf{Visit}(expr); \mathsf{PrimIf}; T_1; T_2; T_3; \mathsf{Return}(\texttt{(}kw\ \ test'\ \ then'\ \ else'\texttt{)})} \end{array}}{E \vdash expr, S \Downarrow \texttt{(}kw\ \ test'\ \ then'\ \ else'\texttt{)}, S_3, \boxed{T}}$$

APPLICATION

$$\frac{\begin{array}{c} expr = \texttt{(}kw\ \ arg_1\ \ \cdots\ \ arg_n\texttt{)} \qquad E(\text{resolve}(kw)) = \langle \texttt{primitive}, \texttt{app} \rangle \\ \forall i \le n : E \vdash arg_i, S_{i-1} \Downarrow arg_i', S_i, \boxed{T_i} \\ \boxed{T = \mathsf{Visit}(expr); \mathsf{PrimApp}; T_1 \cdots T_n; \mathsf{Return}(\texttt{(}kw\ \ expr'_1\ \ \cdots\ \ expr'_n\texttt{)})} \end{array}}{E \vdash expr, S_0 \Downarrow \texttt{(}kw\ \ arg_1'\ \ \cdots\ \ arg_n'\texttt{)}, S_n, \boxed{T}}$$

LET-SYNTAX

$$\frac{\begin{array}{c} expr = \texttt{(}kw\ \ \texttt{((}var_1\ \ rhs_1\texttt{)}\ \cdots\ \texttt{(}var_n\ \ rhs_n\texttt{))}\ \ body\texttt{)} \qquad E(\text{resolve}(kw)) = \langle \texttt{primitive}, \texttt{let-syntax} \rangle \\ \text{each } var_i \text{ is a distinct identifier} \qquad vars = \texttt{(}var_1\ \cdots\ var_n\texttt{)} \qquad \langle newvars, S_1 \rangle = \text{freshnames}(vars, S) \\ \texttt{(}newvar_1\ \cdots\ newvar_n\texttt{)} = newvars \qquad \forall i \le n : \text{ValidRule}(rhs_i) \qquad E' = E[\{newvar_i \mapsto \langle \texttt{macro}, rhs_i \rangle\}] \\ body' = \text{subst}(body, vars, newvars) \qquad E' \vdash body', S_1 \Downarrow body'', S_2, \boxed{T_b} \\ \boxed{T = \mathsf{Visit}(expr); \mathsf{PrimLetSyntax}; \mathsf{Rename}(newvars); T_b; \mathsf{Return}(body'')} \end{array}}{E \vdash expr, S \Downarrow body'', S_2, \boxed{T}}$$

Fig. 13. Semantics of macro expansion with expansion events, continued

the derivation corresponding to the body expression, then

$$[\![\Delta]\!] = \mathsf{LambdaNode}(expr, expr', newvars, [\![\Delta_b]\!])$$

Every inference rule has a corresponding datatype variant, and the fields of the variants suffice to determine the entire contents of the derivation if the initial store and environment are known. We omit the formal definition of $[\![\cdot]\!]$.

Now we are ready to state the correctness theorem. Informally, we want to prove that when the macro expander runs, starting with the initial environment and store, and the stepper constructs an IR tree, the IR tree corresponds to the derivation in the big-step semantics.

Theorem 1 *If Δ is a derivation of the judgment*

$$E \vdash expr, S \Downarrow expr', S', T$$

then $\text{parse}(T) = [\![\Delta]\!]$.

Proof sketch. First we show that T belongs to the language accepted by the grammar $\mathcal{G}$ (by Lem. 2) and that the parse function is defined on all traces in $\mathcal{G}$ (Lem. 3). Consequently, the parser is well-defined on T. Then we proceed by induction on the structure of Δ. For each inference rule, we show that the environments and stores in subjudgments correspond to those in the recursive calls to the parser, and we apply the induction hypothesis on those subderivations. ∎

Lemma 2 *If $E \vdash expr, S \Downarrow expr', S', T$, then T is in the language of $\mathcal{G}$.*

Proof sketch. By induction on the derivation of the judgment. Each inference rule contains a trace expression that corresponds exactly to a line of $\mathcal{G}$. ∎

Lemma 3 *The function* parse *is well-defined on all event sequences in the language of* $\mathcal{G}$.

Proof sketch. The definition of parse covers every alternate in the production of $\mathcal{G}$. The grammar is unambiguous; therefore, a single parse rule applies to any event sequence, and parse is a function. ∎

This result scales up to the error-handling version of the back end. We must simply add inference rules for failed expansions, with one inference rule for each possible point of failure. We must also add the corresponding patterns to the parse function. The structure of the proof stays the same.

6. 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 [13], 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, e.g., the body of a `lambda` term, to expose internal definitions so that the block can be transformed into a `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 and shows extra expansion details.

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. In Kathi Fisler's accelerated introductory course at Worcester Polytechnic Institute, a course that covers macros, the students discovered the macro stepper on their own and found it "cool."

Our users report that the macro stepper has significantly increased their productivity. It has helped several fellow researchers debug large multi-module systems of co-operating macros that implement their language extensions. One example is Typed Scheme [33], a typed dialect of PLT Scheme. The implementation of Typed Scheme is a large project that makes intensive use of the features of the PLT Scheme macro system, and its development benefited greatly from the macro stepper.

User reports support the importance of macro hiding, which 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 recognition as the macro stepper elaborates away the constructs that give programs their structure. At the same time, the need for more structured navigation is clear, especially when debugging large modules, and this is a priority of the continuing development of the macro stepper.

We ended up using the macro stepper for its own development. We naturally used macros to implement the grammar transformation for handling errors, 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.

Acknowledgments We are grateful to Matthew Flatt for his help and guidance in instrumenting the PLT Scheme macro expander. We thank Sam Tobin-Hochstadt for his feedback and bug reports. We also thank the anonymous referees of this paper and its conference version for their suggestions.

References

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

[2] A. Bove and L. Arbilla. A confluent calculus of macro expansion and evaluation. In *ACM Symposium on Lisp and Functional Programming*, pages 278–287, 1992.

[3] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *European Symposium on Programming*, pages 320–334, 2001.
[4] W. Clinger and J. Rees. Macros that work. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 155–162, 1991.
[5] R. Culpepper and M. Felleisen. Debugging macros. In *International Conference on Generative Programming and Component Engineering*, pages 135–144, 2007.
[6] R. Culpepper, S. Owens, and M. Flatt. Syntactic abstraction in component interfaces. In *International Conference on Generative Programming and Component Engineering*, pages 373–388, 2005.
[7] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, Dec. 1993.
[8] S. Egner. Eager comprehensions in Scheme: The design of SRFI-42. In *Workshop on Scheme and Functional Programming*, pages 13–26, Sept. 2005.
[9] M. Felleisen. Transliterating Prolog into Scheme. Technical Report 182, Indiana University, 1985.
[10] M. Felleisen. On the expressive power of programming languages. *Sci. Comput. Programming*, 17:35–75, 1991.
[11] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.
[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.
[13] M. Flatt. Composable and compilable macros: you want it when? In *ACM SIGPLAN International Conference on Functional Programming*, pages 72–83, 2002.
[14] M. Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.
[15] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, 1998.
[16] M. Flatt, R. B. Findler, and M. Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems (APLAS) 2006*, pages 270–289, November 2006.
[17] M. Flatt, S. Krishnamurthi, and M. 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.
[18] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. The MIT Press, July 2005.
[19] M. Gasbichler. *Fully-parameterized, First-class Modules with Hygienic Macros*. PhD thesis, Eberhard-Karls-Universität Tübingen, Feb. 2006.
[20] D. Herman and P. Meunier. Improving the static analysis of embedded languages via partial evaluation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 16–27, 2004.
[21] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.
[22] E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
[23] E. E. Kohlbecker and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 77–84, 1987.
[24] H. Lieberman. Steps toward better debugging tools for LISP. In *Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 247–255, 1984.
[25] S. Owens, M. Flatt, O. Shivers, and B. McMullan. Lexer and parser generators in Scheme. In *Workshop on Scheme and Functional Programming*, pages 41–52, Sept. 2004.
[26] PLT. PLT MzLib: Libraries manual. Technical Report PLT-TR2006-4-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.
[27] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass framework for compiler education. *Journal of Functional Programming*, 15(5):653–667, Sept. 2005. Educational Pearl.
[28] O. Shivers. The anatomy of a loop: a story of scope and control. In *ACM SIGPLAN International Conference on Functional Programming*, pages 2–14, 2005.
[29] D. Sitaram. Programming in schelog. `http://www.ccs.neu.edu/home/dorai/schelog/schelog.html`.
[30] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten (Editors). Revised^6 report of the algorithmic language Scheme, Sept. 2007. Available at `http://www.r6rs.org`.
[31] G. L. Steele, Jr. Rabbit: a compiler for Scheme. Technical Report 474, MIT Artificial Intelligence Laboratory, May 1978.
[32] G. L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, 1990.
[33] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 395–406, 2008.
[34] A. P. Tolmach and A. W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.
[35] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 203–215, 1999.
[36] M. Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3):189–199, 1998.
[37] A. Wright and B. Duba. Pattern matching for Scheme, 1995. Unpublished manuscript.