Syntactic Abstraction in Scheme

Robert Hieb[*], R. Kent Dybvig, and Carl Bruggeman
dyb or bruggema@cs.indiana.edu

Indiana University Computer Science Department
Technical Report #355

June 1992
(Revised July 3, 1992)

Abstract

Naive program transformations can have surprising effects due to the interaction between introduced identifier references and previously existing identifier bindings, or between introduced bindings and previously existing references. These interactions can result in the inadvertent binding, or capturing, of identifiers. A further complication results from the fact that the transformed program may have little resemblance to the original program, making correlation of source and object code difficult. We address both the capturing problem and the problem of source-object code correlation. Previous approaches to the capturing problem have been both inadequate and overly restrictive, and the problem of source-object code correlation has been largely unaddressed. Our approach is based on a new algorithm for implementing syntactic transformations along with a new representation for syntactic expressions. It allows the programmer to define program transformations using an unrestricted, general-purpose language, while at the same time it helps the programmer avoid capturing problems and maintains a correlation between the original code and the transformed code.

1 Introduction

A fundamental problem with most Lisp macro systems is that they do not respect lexical scoping. When one expression is substituted for another, apparent bindings can be shadowed, resulting in unintended capture of variable references. This is the source of many serious and difficult to find bugs. This problem was first addressed by Kohlbecker, Friedman, Felleisen, and Duba [15], who proposed a *hygiene condition* for macros and a macro-expansion algorithm that enforces this condition. Unfortunately, the KFFD algorithm increases the computational complexity of the macro expansion process.

An additional problem of equal practical importance is that Lisp macro systems cannot track source code through the macro-expansion process. In order for the compiler, run-time system, and debugger to communicate with the programmer in terms of the original program, it is necessary to be able to reliably correlate source code and macro-expanded code. This source-object correlation problem has been addressed by optimizing compiler writers [13, 6, 18]. However, if the macro

[*]Robert Hieb died in an automobile accident on April 30, 1992

expansion process loses the correlation between source and macro-expanded code, the efforts of the compiler writer in this regard may be wasted. This is especially problematic in Scheme, where nearly all syntactic forms are implemented as macros. Unfortunately, techniques that have been applied to optimizing compilers are not directly applicable to macro processors, since in the former the transformations performed are known to the compiler writer, whereas in the latter the transformations are written by the user in a general purpose language.

This paper presents a macro system for Scheme that preserves hygiene automatically while adding only constant overhead to the macro expansion process and solving the source-object correlation problem. This system also maintains "referential transparency" as defined for local macros by Clinger and Rees [4] (see Section 2). Also presented is a mechanism that supports a controlled form of variable capture that allows most common "capturing" macros to be written without violating the spirit of hygiene. Our system is based on a notion of syntax objects, and our algorithm has at its core a "lazy" variant of the original KFFD algorithm. This algorithm is extended to support pattern variables, controlled variable capture, and source-object correlation. Our system is applicable to macros written in a general purpose programming language (Scheme), and it extends to macros written in high-level pattern languages such as *extend-syntax* [14, 7].

Other systems have been proposed to solve some of these problems; many of these are described in the following section. However, ours is the only system proposed to date that:

- enforces hygiene for macros written in a general-purpose language with constant overhead,
- solves the source-object correlation problem for variables and constants as well as structured expressions,
- supplies a hygiene-preserving mechanism for controlled variable capture, and
- maintains referential transparency for all local macros.

The remainder of this paper is structured as follows. Section 2 discusses related work. Section 3 describes the interface to our macro system and examples of its use. Section 4 presents our algorithm. It begins with a variant of the KFFD algorithm applied to abstract syntax, demonstrates how delaying part of the work of this algorithm reduces expansion overhead to a constant, and shows how the correlation between source and object code is maintained.

2 Background

Various *ad hoc* techniques have been developed to help prevent unintended variable captures. Capturing problems have been avoided by using generated names, special naming conventions, or the careful use of lexical-scoping combined with local procedure definitions. Unfortunately, these approaches make capturing the *default*, since the programmer has to do something special to avoid it, when in fact capturing is rarely desired. What is worse, macros that can cause unintended captures often do not do so immediately, but lie dormant in the program, waiting for the unsuspecting programmer to insert just the right (wrong!) variable name into a macro call or its context.

These problems were first addressed by Kohlbecker, Friedman, Felleisen, and Duba [15], who develop an algorithm in which the macro system *automatically* renames bound variables to prevent inadvertent capturing. The fundamental notion underlying the KFFD algorithm is *alpha equivalence*, which equates terms that differ only in the names of bound variables. Alpha equivalence is the basis for Barendregt's *variable convention*, which assumes that the bound variables in terms used in definitions and proofs are always chosen to be different from the free variables [1, page 26]. The KFFD algorithm respects the variable convention and thus is said to be "hygienic." It traverses each expression after it is rewritten in order to give identifiers "time stamps," which are used during alpha conversion to distinguish identifiers that were introduced at different times in the transformation process. Unfortunately, since the algorithm completely traverses each expression after it is rewritten, it increases the worst-case time complexity of the macro expansion process from linear to quadratic with respect to the number of expressions present in the source code or introduced during macro expansion. This is a serious problem for large programs that make heavy use of macros, *e.g.*, nearly all large Scheme programs.

Clinger and Rees [4] present an algorithm for hygienic macro transformations that does not have the quadratic time complexity of the original KFFD algorithm. They mark only the new identifiers introduced at each iteration of the macro transformation process, as opposed to all of the identifiers as in the original algorithm. However, their system allows macros to be written only in a restricted high-level specification language in which it is easy to determine where new identifiers will appear in the output of a macro. Since some macros cannot be expressed easily using this language, they have developed a low-level interface that requires new identifiers to be marked explicitly [3].

Bawden and Rees [2] approach the capturing problem from a different angle. Rather than providing automatic hygiene, their system forces the programmer to make explicit decisions about the resolution of free identifier references and the scope of identifier bindings. Borrowing from the notion of *closures* as procedures that embody environments for their free identifiers, they allow the programmer to create *syntactic closures*. The result is a system that allows the programmer not only to avoid unwanted capturing but also to introduce arbitrary capturing. Unlike traditional closures, however, syntactic closures and their environments must be constructed explicitly. As a result, the mechanism is difficult to use and definitions created using it are difficult to understand and verify. To alleviate this, Hanson [12] has recently demonstrated that the restricted high-level specification language supported by Clinger and Rees can be built on top of a modified version of syntactic closures.

With both the Clinger and Rees and syntactic closures approaches, responsibility for enforcing the hygiene convention is placed on the macro writer rather than on the underlying transformation algorithm for a large class of macros that cannot be written in the high-level specification language. Furthermore, the convenient pattern matching facilities provided by the specification language must be completely abandoned for the same class of macros.

Both algorithms also support local macro definitions that are "referentially transparent" [17] in the sense that a macro-introduced identifier refers to the binding present at the point of definition of the macro rather than the point of use of the macro. This extends the notion of hygiene to local macro definitions, which were not handled by KFFD algorithm. Like automatic hygiene, however, this transparency is not present in the low-level system.

Griffin [11] describes a theory of syntactic definitions in the context of interactive proof development systems. He supports high-level definitions of derived notations in such a way that the definitions have certain formal properties that make them easy to reason about. As a result, however, his system is very restrictive with respect to the sort of macros that can be defined.

Dybvig, Friedman and Haynes [9, 10] address the source-object correlation problem, demonstrating that their proposed macro expansion protocol, *expansion-passing style*, is capable of maintaining source-object correlation even in the presence of arbitrary user-defined macros. However, their mechanism handles only structured expressions; in particular, it does not handle variable references.

The Revised[4] Report on Scheme [5] includes an appendix that contains a proposed macro system for Scheme. The "high-level" (**syntax-rules**) system described therein is a version of Kohlbecker's **extend-syntax** [14] with the same restrictions imposed by Clinger and Rees [4]. The revised-report appendix also describes a "low-level" system that, although it automatically preserves hygiene and referential transparency, requires manual destructuring of the input and restructuring of the output. The low-level system described in the revised-report appendix was proposed by the authors of this paper and is the predecessor of the system described here. The new system provides only a "high-level" pattern language, similar to the one provided by **syntax-rules**, which is nonetheless powerful enough to provide the functionality of the "low-level" system while maintaining automatic hygiene, referential transparency, and source-object correlation. A detailed description of the features available in the system described in this paper and an extensive set of annotated examples can be found in [8].

3 The language

New syntactic forms are defined by associating *keywords* with transformation procedures, or *transformers*. At top level, syntactic definitions take the form of:

(**define-syntax** *keyword transformer-expression*)

The *transformer-expression* must be an expression that evaluates to a transformer. When the expander encounters an expression of the form (*keyword subform* ...), the expression is passed to the associated transformer to be processed; the expansion process is repeated for the result returned by the transformer until no macro forms are left.

The scope of syntactic definitions can be limited by using the lexical binding forms:

(**let-syntax** ((*keyword transformer-expression*) ...) *body*)
(**letrec-syntax** ((*keyword transformer-expression*) ...) *body*)

In both of these forms a *keyword* denotes new syntax in *body*; in the case of **letrec-syntax** the binding scope also includes each *transformer-expression*.

At the language level, the fundamental characteristic of our system is the abstract nature of the arguments passed to macro transformers. The argument to a macro transformer is a *syntax object*, which contains contextual information about an expression in addition to its structure. This contextual information is used by the expander to maintain hygiene and referential transparency. Traditional Lisp macro systems use ordinary list-structured data to represent syntax. Although such list structures are convenient to manipulate, crucial syntactic information cannot be maintained. For the macro writer, the ability to distinguish between different identifiers that share the same name is of paramount importance. Information to allow these distinctions to be drawn is contained within each abstract syntax object. Transformers can compare identifiers according to their *intended use* as free identifiers, bound identifiers, or symbolic data.

Syntax objects may contain other syntactic information that is not of direct interest to the macro writer. In our system, for example, syntax objects can contain source annotations that allow the evaluator to correlate the final object code with the source code that produced it. Or, as discussed in Section 4, syntax objects may contain information that for efficiency reasons has not yet been fully processed.

Transformers decompose their input using **syntax-case** and rebuild their output using **syntax**. A **syntax-case** expression takes the following form:

(**syntax-case** *input-expression* (*literal* ...) (*pattern output-expression*) ...)

syntax-case first evaluates *input-expression*, then attempts to match the resulting value with the pattern from the first *clause*. This value is usually a syntax object, but it may be any Scheme list structure. If the value matches the pattern, *output-expression* is evaluated and its value returned as the value of the **syntax-case** expression. If the value does not match the pattern, the value is compared against the next clause, and so on. An error is signaled if the value does not match any of the patterns.

List structure within a pattern specifies the basic structure required of the value returned by *input-expression*. Identifiers not found in the literals list (*literal* ...) are pattern variables which match arbitrary substructure. Literals and constants specify atomic pieces that must match exactly. An ellipsis, represented by the special identifier ..., specifies zero or more occurrences of the subpattern it follows.

Within *output-expression*, pattern variables may be used within **syntax** expressions to reference matching portions of *input-expression*. Although pattern variable bindings created by **syntax-case** can shadow (and be shadowed by) lexical and macro keyword bindings, pattern variables can only be referenced within **syntax** expressions.

The syntactic construct (**syntax** *template*) is used to introduce syntax objects. This form evaluates to a syntax object representing *template* in much the same way as (**quote** *expression*) or (**quasiquote** *expression*) evaluates to a "raw" representation for *expression*. However, there are two important differences: the values of pattern variables appearing within *template* are inserted into the output, and contextual syntactic information contained within *template* is retained.

The expressive power of **syntax-case** and **syntax** renders a "low level" system unnecessary. Unlike simple rewrite-rule systems, like **syntax-rules**, input patterns are associated with output *expressions*, not output patterns. Arbitrary transformations may be performed since the output expression may be any Scheme expression. The only restriction is that "raw" symbols cannot be used in the construction of the result; identifiers must always be introduced using **syntax**.

Unlike other low-level proposals, **syntax-case** relieves the programmer from the tedium of pattern matching, destructuring, and restructuring expressions, and provides a level of syntax checking that macro programmers usually do not provide.

Nothing is lost by providing **syntax-case** and **syntax** in place of **syntax-rules**; in fact **syntax-rules** is easily defined as a macro:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((syntax-rules (i ...) (in out) ...)
       (lambda (x) (syntax-case x (i ...) (in (syntax out)) ...))))))
```

Although we use **syntax-case** rather than **syntax-rules** in our examples, some could be written using **syntax-rules**, and in those cases, the choice of which to use is a matter of taste.

Identifiers within a **syntax** expression, like ordinary variable references, refer to the closest lexical binding. For example:

```
(let ((− +))
  (let-syntax ((minus (lambda (x)
                        (syntax-case x ()
                          ((minus e1 e2) (syntax (− e1 e2)))))))
    (let ((− *)) (minus 2 1))))
```

returns 3 since the "−" inserted by the macro *minus* refers to the outer **let** binding, rather than 1, as would be the case if the reference were made global, or 2, as would be the case if the reference were captured by the inner **let** binding. All identifier references in our system are referentially transparent in the sense discussed in Section 2.

Symbolic names alone do not distinguish identifiers unless they are actually being used as symbolic data. The predicates *free-identifier=?* and *bound-identifier=?* are used to compare identifiers according to their *intended use* as free references or bound variables in a given context. The predicate *free-identifier=?* is used to determine whether two identifiers would be equivalent if they appeared as free identifiers in the output of a transformer. Because identifier references are lexically scoped, this means (*free-identifier=?* id_1 id_2) is true if and only if the identifiers id_1 and id_2 have the same binding. Literal identifiers appearing in **syntax-case** patterns (such as **else** in **cond**) are matched using *free-identifier=?*.

Similarly, the predicate *bound-identifier=?* is used to determine if two identifiers would be equivalent if they appeared as bound identifiers in the output of a transformer. In other words, if *bound-identifier=?* returns true for two identifiers, then a binding for one will capture references to the other within its scope. In general, two identifiers are *bound-identifier=?* only if both are present in the original program or both are introduced by the same macro application (perhaps implicitly; see *implicit-identifier* below). The predicate *bound-identifier=?* can be used for detecting duplicate variable names in a binding construct, or for other preprocessing of a binding construct that requires detecting instances of the bound variables.

Two identifiers that are *bound-identifier=?* are also *free-identifier=?*, but two identifiers that are *free-identifier=?* may not be *bound-identifier=?*. This is a consequence of the fundamental premise of hygienic macro expansion: an identifier introduced by a macro transformer should not bind, or be bound by, an identifier in the input to the transformer. Yet an identifier introduced by a macro transformer may refer to the same binding as an identifier not introduced by the transformer if they both refer to the same lexical or top-level binding.

It is also possible to distinguish identifiers that are intended to be used as symbolic data using the procedure *syntax-object->datum*. This procedure strips all syntactic information from a syntax object and returns the corresponding Scheme "datum." Identifiers stripped in this manner are converted to their symbolic names, which can then be compared with *eq?*. Thus, *symbolic-identifier=?* might be defined as follows:

```
(define symbolic-identifier=?
  (lambda (x y)
    (eq? (syntax-object->datum x)
         (syntax-object->datum y))))
```

Below is a macro definition for **let** that uses *bound-identifier=?* to detect duplicate identifiers. In order to make this macro completely robust, it should ensure that the bound variables are indeed identifiers using the predicate *identifier=?*, which returns true if and only if its argument is an identifier.

```
(define-syntax let
  (lambda (x)
    (define unique-ids?
      (lambda (ls)
        (or (null? ls)
            (and (let notmem? ((x (car ls)) (ls (cdr ls)))
                      (or (null? ls)
                          (and (not (bound-identifier=? x (car ls)))
                               (notmem? x (cdr ls)))))
                 (unique-ids? (cdr ls))))))
    (syntax-case x ()
      ((let ((i v) ...) e1 e2 ...)
       (if (unique-ids? (syntax (i ...)))
           (syntax ((lambda (i ...) e1 e2 ...) v ...))
           (error (syntax-object->datum x) "duplicate identifier found"))))))
```

Occasionally it is useful to define macros that introduce bindings for identifiers that are not supplied explicitly in each macro call. For example, we might wish to define a **loop** macro that binds the implicit variable *exit* to an escape procedure within the loop body. However, strict automatic hygiene prevents introduced bindings from capturing existing variable references. Previous hygienic systems have provided mechanisms for *explicit capturing*, typically by allowing a macro to insert a symbol into an expansion as if it were part of the original source program [15]. Unfortunately, this means that macros cannot reliably expand into macros that use explicit capturing.

Our system provides a more consistent way to accommodate such macros. A macro may construct an *implicit identifier* that behaves as if it were present in the macro call. Implicit identifiers are created by providing the procedure *implicit-identifier* with a *template identifier* and a *symbol.*

The template identifier is typically the macro keyword itself, extracted from the input, and the symbol is the symbolic name of the identifier to be constructed. The resulting identifier behaves as if it had been introduced when the template identifier was introduced. For example, the **loop** macro mentioned above may be defined as follows:

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      ((loop e1 ...)
       (with-syntax ((exit-id (implicit-identifier (syntax loop) 'exit)))
         (syntax (call-with-current-continuation
                   (lambda (exit-id)
                     (let f () e1 ... (f))))))))))
```

(The **with-syntax** form introduces pattern variable bindings within the scope of its body; its definition is shown later.)

This same mechanism may be used to create aggregate identifier names typically required when defining structure-definition constructs such as Common Lisp **defstruct** [16], as macros. The procedure below constructs an implicit identifier using an aggregate name of the form "⟨structure name⟩-⟨field name⟩," from a structure name identifier *s-id* and a field name identifier *f-id*:

```
(define aggregate-identifier
  (lambda (s-id f-id)
    (let ((s-name (symbol->string (syntax-object->datum s-id)))
          (f-name (symbol->string (syntax-object->datum f-id))))
      (let ((sym (string->symbol (string-append s-name "-" f-name))))
        (implicit-identifier s-id sym)))))
```

A **defstruct** form would expand into a set of definitions, including accessors for each field whose names are constructed using *aggregate-identifier*.

The **with-syntax** form used in the **loop** example above often provides the easiest way to establish new pattern variable bindings. It is possible to use **syntax-case** instead, although this usually results in less readable code. For example, the **loop** macro could be defined as follows:

```
(define-syntax loop
  (lambda (x)
    (syntax-case x ()
      ((loop e1 ...)
       (syntax-case (implicit-identifier (syntax loop) 'exit) ()
         (exit-id (syntax (call-with-current-continuation
                            (lambda (exit-id)
                              (let f () e1 ... (f)))))))))))
```

Given that **syntax-case** can be used in this manner, it is not surprising that **with-syntax** can be defined as a macro in terms of **syntax-case**:

```
(define-syntax with-syntax
  (lambda (x)
    (syntax-case x ()
      ((with-syntax ((p e0) ...) e1 e2 ...)
       (syntax (syntax-case (list e0 ...) ()
                 ((p ...) (begin e1 e2 ...))))))))
```

A more detailed description of the macro language and many examples of its use can be found in [8].

4 The algorithm

4.1 Traditional macro systems

Traditional Lisp macro systems rely on the fact that programs and data have the same representation, both textually and as internal structures. This shared representation is exploited not only for macro expansion but also for program evaluation; most Lisp systems provide an evaluation procedure so that programs can construct and execute programs. Consequently, the concrete syntax of Lisp can best be seen as consisting of internal data structures rather than text. We assume a concrete syntax of expressions ($e \in \mathit{Exp}$) defined as a data type consisting of an unspecified set of constants ($c \in \mathit{Const}$), symbols ($s \in \mathit{Sym}$), and structures built by pairing. The following signature specifies the abstract data type *Exp*:

$$
\begin{array}{rcl}
\mathit{Sym} & \subset & \mathit{Exp} \\
\mathit{Const} & \subset & \mathit{Exp} \\
\mathit{cons} & : & \mathit{Exp} \times \mathit{Exp} \rightarrow \mathit{Exp} \\
\mathit{car} & : & \mathit{Exp} \rightarrow \mathit{Exp} \\
\mathit{cdr} & : & \mathit{Exp} \rightarrow \mathit{Exp} \\
\mathit{pair?} & : & \mathit{Exp} \rightarrow \mathit{Bool} \\
\mathit{sym?} & : & \mathit{Exp} \rightarrow \mathit{Bool}
\end{array}
$$

The subset *Sym* (symbols) of *Exp* are atomic elements. *Const* includes such traditional Lisp constants as booleans, numbers and the empty list. The variables e, s, and c range over *Exp*, *Sym* and *Const*, respectively. The usual equations for elements of *Exp* hold:

$$
\begin{array}{rcl}
\mathit{car}(\mathit{cons}(e_1, e_2)) & = & e_1 \\
\mathit{cdr}(\mathit{cons}(e_1, e_2)) & = & e_2 \\
\mathit{pair?}(\mathit{cons}(e_1, e_2)) & = & \mathit{True} \\
\mathit{sym?}(\mathit{cons}(e_1, e_2)) & = & \mathit{False} \\
\mathit{pair?}(s) & = & \mathit{False} \\
\mathit{sym?}(s) & = & \mathit{True} \\
\mathit{pair?}(c) & = & \mathit{False} \\
\mathit{sym?}(c) & = & \mathit{False}
\end{array}
$$

$$expand : Exp \times Env \rightarrow ExpandedExp$$

$$
\begin{array}{l}
expand(e, r) = \\
\quad \text{case } parse(e, r) \text{ of:} \\
\qquad \begin{array}{rcl}
constant(c) & \rightarrow & symbolic\text{-}data(c), \\
variable(s) & \rightarrow & variable(s), \\
application(e_1, e_2) & \rightarrow & application(expand(e_1, r), expand(e_2, r)), \\
symbolic\text{-}data(e) & \rightarrow & symbolic\text{-}data(e), \\
function(s, e) & \rightarrow & function(s, expand(e, r[s := Variable])), \\
macro\text{-}application(s, e) & \rightarrow & expand(t(e), r) \\
 & & \text{where } t = r(s),
\end{array}
\end{array}
$$

Figure 1: A traditional macro-expansion algorithm.

$$parse : Exp \times Env \rightarrow ParsedExp$$

$$
\begin{array}{rcl}
parse(\llbracket c \rrbracket, r) & = & constant(c) \\
parse(\llbracket s \rrbracket, r) & = & variable(s) \quad \text{if } r(s) = Variable \\
parse(\llbracket (e_1\ e_2) \rrbracket, r) & = & application(e_1, e_2) \quad \text{if } e_1 \notin Sym \\
parse(\llbracket (s\ e) \rrbracket, r) & = & application(s, e) \quad \text{if } r(s) = Variable \\
parse(\llbracket (s\ e) \rrbracket, r) & = & macro\text{-}application(s, e) \quad \text{if } r(s) \in Transformer \\
parse(\llbracket (\mathbf{quote}\ e) \rrbracket, r) & = & symbolic\text{-}data(e) \quad \text{if } r(\llbracket \mathbf{quote} \rrbracket) = Special \\
parse(\llbracket (\mathbf{lambda}\ s\ e) \rrbracket, r) & = & function(s, e) \quad \text{if } r(\llbracket \mathbf{lambda} \rrbracket) = Special
\end{array}
$$

Figure 2: A traditional macro-expansion parser.

Figure 1 shows a traditional expansion algorithm for a simplified language. The expander is assumed to be part of a standard evaluation process where the value of a program e is obtained by $eval(expand(e, r_{init}))$. The symbols **quote** and **lambda** are bound to *Special* in the initial expansion environment r_{init}; all other symbols are bound to *Variable*.

$$
\begin{array}{rcl}
r & \in & Env = Sym \rightarrow Transformer + Variable + Special \\
t & \in & Transformer = Exp \rightarrow Exp
\end{array}
$$

Macro expansion and parsing are inextricably intertwined in Lisp. Although Figure 1 shows the expander driving the parser, the relationship could just as well be reversed. The parser is shown in Figure 2. Pattern matching is used to hide the details of accessing the expression parts. The constructors (such as *symbolic-data*) used to communicate between the parser and the expander are not fully specified (their definition is trivial). Since $ExpandedExp \subset ParsedExp$, the same constructors are used to communicate between the expander and the evaluator.

It is not hard to see that this expansion algorithm has serious hygiene problems. It does not prevent free variables inserted by a macro application from being captured by program bindings,

$$expand : Exp \times Env \rightarrow ParsedExp$$

$$
\begin{array}{l}
expand(e, r) = \\
\quad \text{case } parse(e, r) \text{ of:} \\
\quad \begin{array}{rcl}
variable(i) & \rightarrow & variable(resolve(i)), \\
application(e_1, e_2) & \rightarrow & application(expand(e_1, r), expand(e_2, r)), \\
symbolic\text{-}data(e) & \rightarrow & symbolic\text{-}data(strip(e)), \\
syntax\text{-}data(e) & \rightarrow & symbolic\text{-}data(e), \\
function(i, e) & \rightarrow & function(s, expand(subst(e, i, s), r[s := Variable])) \\
 & & \quad \text{where } s \text{ is fresh,} \\
pfunction(i, e) & \rightarrow & function(s, expand(subst(e, i, s), r[s := Pattern\text{-}Variable])) \\
 & & \quad \text{where } s \text{ is fresh,} \\
macro\text{-}application(i, e) & \rightarrow & expand(mark(t(mark(e, m)), m), r) \\
 & & \quad \text{where } t = r(resolve(i)) \text{ and } m \text{ is fresh,} \\
syntax\text{-}binding(i, e_1, e_2) & \rightarrow & expand(subst(e_2, i, s), r[s := t]) \\
 & & \quad \text{where } t = eval(expand(e_1, r)) \text{ and } s \text{ is fresh} \\
rec\text{-}syntax\text{-}binding(i, e_1, e_2) & \rightarrow & expand(subst(e_2, i, s), r[s := t]) \\
 & & \quad \text{where } t = eval(expand(subst(e_1, i, s), r)) \text{ and } s \text{ is fresh}
\end{array}
\end{array}
$$

Figure 3: A substitution-based macro-expansion algorithm.

nor does it prevent bindings introduced by macros from capturing free identifiers in the program.

4.2 A substitution-based macro system

In the λ-calculus, alpha conversion is used to circumvent hygiene problems caused by program transformations. Since the actual name of a bound variable is immaterial, a binding expression can be converted into an equivalent expression in which different names are used for the bound variables. Our algorithm uses alpha conversion to preserve hygiene during macro expansion.

Since symbols as data and symbols as variables cannot be distinguished reliably until after macro expansion, and since the name of a symbol, when used as data, is always important, naive alpha conversion is not viable in traditional macro expansion algorithms. Our algorithm makes alpha conversion possible by abandoning the traditional Lisp identification of variables and symbols. Instead, we introduce a new abstraction, the *identifier*, which maintains both symbolic names and variable names until an identifier's role in a program is determined. Alpha conversion is accomplished by replacing only the variable names of bound identifiers.

Figure 3 shows the substitution-based macro-expansion algorithm. The parser, shown in Figure 4, has been modified to operate on identifiers rather than symbols and to recognize several new forms: **let-syntax**, **letrec-syntax**, **syntax**, and **plambda**. To simplify the presentation, **let-syntax** and **letrec-syntax** are each restricted to a single binding. We have also restricted the subform of a **syntax** expression to a single identifier. If the identifier is a pattern variable, the **syntax** form evaluates to the value of the pattern variable; otherwise, the result is the identifier itself. The unrestricted version is a straightforward generalization. We have also chosen to

$$parse : Exp \times Env \rightarrow ParsedExp$$

$$\begin{array}{rcl}
parse([\![c]\!], r) & = & symbolic\text{-}data(c) \\
parse([\![i]\!], r) & = & variable(i) \quad \text{if } r(resolve(i)) = Variable \\
parse([\![(e_1\ e_2)]\!], r) & = & application(e_1, e_2) \quad \text{if } e_1 \notin Sym \\
parse([\![(i\ e)]\!], r) & = & application(i, e) \quad \text{if } r(resolve(i)) = Variable \\
parse([\![(i\ e)]\!], r) & = & macro\text{-}application(i, e) \quad \text{if } r(resolve(i)) \in Transformer \\
parse([\![(\mathbf{quote}\ e)]\!], r) & = & symbolic\text{-}data(e) \quad \text{if } r([\![\mathbf{quote}]\!]) = Special \\
parse([\![(\mathbf{lambda}\ i\ e)]\!], r) & = & function(i, e) \quad \text{if } r([\![\mathbf{lambda}]\!]) = Special \\
parse([\![(\mathbf{plambda}\ i\ e)]\!], r) & = & pfunction(i, e) \quad \text{if } r([\![\mathbf{plambda}]\!]) = Special \\
parse([\![(\mathbf{syntax}\ i)]\!], r) & = & syntax\text{-}data(i) \quad \text{if } r(resolve(i)) \neq Pattern\text{-}Variable \\
parse([\![(\mathbf{syntax}\ i)]\!], r) & = & variable(i) \quad \text{if } r(resolve(i)) = Pattern\text{-}Variable \\
parse([\![(\mathbf{let\text{-}syntax}\ (i\ e_1)\ e_2)]\!], r) & = & syntax\text{-}binding(i, e_1, e_2) \quad \text{if } r([\![\mathbf{let\text{-}syntax}]\!]) = Special \\
parse([\![(\mathbf{letrec\text{-}syntax}\ (i\ e_1)\ e_2)]\!], r) & = & rec\text{-}syntax\text{-}binding(i, e_1, e_2) \\
 & & \quad \text{if } r([\![\mathbf{letrec\text{-}syntax}]\!]) = Special
\end{array}$$

Figure 4: A substitution-based macro-expansion parser.

add **plambda**, which binds a single pattern variable within it's body, rather than **syntax-case**, which can be defined in terms of **plambda**. Pattern variables are bound to *Pattern-Variable* in the expansion environment.

$$Env = Sym \rightarrow Transformer + Variable + Pattern\text{-}Variable + Special$$

The function *resolve* is used by *expand* to complete alpha substitution and determine the actual variable name of an identifier. The variable name is used in the output for program variables and to look up transformers for syntactic variables (keywords). When expanding a binding expression, *subst* replaces the variable name of the bound identifier with a fresh variable name. To distinguish new identifiers introduced by a transformer, both input to the transformer and output from the transformer are freshly marked. Since identical marks cancel each other, only new syntax retains the mark.[1] The expander handles two sorts of data, symbolic (introduced by **quote** expressions) and syntactic (introduced by **syntax** expressions). Symbolic data is stripped of identifier substitutions and markings, whereas syntactic data is left intact.

Since *mark* and *subst* both generate elements of *Exp*, they can be treated as constructors in an extended *Exp* algebra.

$$\begin{array}{rcl}
mark & : & Exp \times Mark \rightarrow Exp \\
subst & : & Exp \times Ident \times Sym \rightarrow Exp
\end{array}$$

[1]For the simplified language considered here it would be adequate to mark only the input to the transformer. However, this approach would not work for more complex language constructs in which internal definitions are expanded separately and then recombined into a binding expression. It would also cause complexity problems for the delayed substitution mechanism described in Section 4.4.

Marks ($m \in \mathit{Mark}$) can be any countably infinite set. Identifiers ($e \in \mathit{Ident}$) are a subset of the expanded *Exp* domain. An identifier is a symbol that has been subjected to zero or more marking and substitution operations. The intent of $\mathit{subst}(e, i, s)$ is to replace the variable name of the identifier i in the expression e with the symbol s.

Since marking and substitution operations are of interest only insofar as they affect identifiers, it is convenient to think of them as identity operations on constants and as being immediately propagated to the components of a pair. For now, we assume that this is the case, although later we abandon this assumption in order to avoid complexity problems.

$$\begin{aligned}
\mathit{mark}(c, m) &= c && (1)\\
\mathit{subst}(c, i, s) &= c && (2)\\
\mathit{mark}(\mathit{cons}(e_1, e_2), t) &= \mathit{cons}(\mathit{mark}(e_1, m), \mathit{mark}(e_2, m)) && (3)\\
\mathit{subst}(\mathit{cons}(e_1, e_2), i, s) &= \mathit{cons}(\mathit{subst}(e_1, i, s), \mathit{subst}(e_2, i, s)) && (4)
\end{aligned}$$

The function *resolve* is used to determine the variable name of an identifier. It resolves substitutions using the criterion that a substitution should take place if and only if both identifiers have the same marks and the same variable name.

$$\begin{aligned}
\mathit{resolve} &: \mathit{Ident} \to \mathit{Sym}\\
\mathit{resolve}(s) &= s\\
\mathit{resolve}(\mathit{mark}(i, m)) &= \mathit{resolve}(i)\\
\mathit{resolve}(\mathit{subst}(i_1, i_2, s)) &= \begin{cases} s & \text{if } \mathit{marksof}(i_1) = \mathit{marksof}(i_2) \\ & \text{and } \mathit{resolve}(i_1) = \mathit{resolve}(i_2) \\ \mathit{resolve}(i_1) & \text{otherwise} \end{cases}
\end{aligned}$$

The auxiliary function *marksof* determines an identifier's mark set:

$$\begin{aligned}
\mathit{marksof} &: \mathit{Ident} \to \mathit{MarkSet}\\
\mathit{marksof}(s) &= \emptyset\\
\mathit{marksof}(\mathit{mark}(i, m)) &= \mathit{marksof}(i) \uplus \{m\}\\
\mathit{marksof}(\mathit{subst}(i_1, i_2, s)) &= \mathit{marksof}(i_1)
\end{aligned}$$

The operator $\uplus$ forms an exclusive union, which cancels identical marks.

The function *strip* simply undoes marking and substitution operations:

$$\begin{aligned}
\mathit{strip} &: \mathit{Exp} \to \mathit{Exp}\\
\mathit{strip}(s) &= s\\
\mathit{strip}(c) &= c\\
\mathit{strip}(\mathit{cons}(e_1, e_2)) &= \mathit{cons}(\mathit{strip}(e_1), \mathit{strip}(e_2))\\
\mathit{strip}(\mathit{mark}(e, m)) &= \mathit{strip}(e)\\
\mathit{strip}(\mathit{subst}(e, i, s)) &= \mathit{strip}(e)
\end{aligned}$$

Two identifiers i_1 and i_2 are *free-identifier=?* if and only if $\mathit{resolve}(i_1) = \mathit{resolve}(i_2)$. Two identifiers i_1 and i_2 are *bound-identifier=?* if and only if $\mathit{resolve}(\mathit{subst}(i_1, i_2, s)) = s$ for a fresh symbol s.

So far the *Exp* algebra has been considered abstractly. A concrete algebra must ensure that the primitive accessors behave as specified. By virtue of equations (1)–(4), constants and pairs can use traditional representations. Identifiers can be represented as distinguished triples of the form $\langle s_1, s_2, \{m \ldots\}\rangle$, where s_1 is the symbolic name, s_2 is the variable name, and $\{m \ldots\}$ is a (possibly empty) set of marks. This representation takes advantage of the fact that *strip* is only interested in the symbolic name of an identifier, and that *resolve* is only interested in the final variable name of an identifier. Intermediate substitutions and substitutions that cannot succeed can be discarded without affecting the behavior of the accessors. The mark set for an identifier i is just $\mathit{marksof}(i)$. Marks can be represented as integers. Given this representation of identifiers, implementation of the primitive operators is straightforward. $\mathit{mark}(i, m)$ adds its mark to the mark field of i unless it is already present, in which case it removes it. $\mathit{subst}(i_1, i_2, s)$ replaces the variable name field of i_1 with s if the variable names and the marks of i_1 and i_2 are the same, otherwise it leaves the identifier unchanged. $\mathit{strip}(i)$ extracts the symbolic name of an identifier, whereas $\mathit{resolve}(i)$ extracts the variable name of an identifier,

4.3 Capturing

The procedure *implicit-identifier* must construct a new identifier that behaves as if it had been introduced where the template identifier was introduced. That is, if *implicit-identifier* is called with an identifier i_1 and a symbol s_2, where the symbolic name of i_1 is s_1, *implicit-identifier* should create the identifier i_2 that would have resulted had s_2 appeared in place of s_1 in the original input. The following definition captures this semantics:

$$
\begin{array}{rcl}
\mathit{imp\text{-}id} & : & \mathit{Ident} \times \mathit{Sym} \rightarrow \mathit{Ident} \\
\mathit{imp\text{-}id}(s_1, s_2) & = & s_2 \\
\mathit{imp\text{-}id}(\mathit{mark}(i, m), s) & = & \mathit{mark}(\mathit{imp\text{-}id}(i, s), m) \\
\mathit{imp\text{-}id}(\mathit{subst}(i_1, i_2, s_1), s_2) & = & \mathit{subst}(\mathit{imp\text{-}id}(i_1, s_2), i_2, s_1)
\end{array}
$$

Supporting *imp-id* means that the representation for identifiers cannot discard failed substitutions, since the new accessor *imp-id* can observe them. However, intermediate substitutions are still unimportant and substitutions that fail because of mismatched marks can still be discarded. Thus the representation of identifiers as triples can be adapted by replacing the variable name in an identifier triple with an environment that maps *Sym* to *Sym*. *resolve* must then apply the environment to the symbolic name to get the variable name. $\mathit{imp\text{-}id}(i, s)$ simply builds a new identifier triple from s and the environment and marks from i.

4.4 A lazy substitution-based macro system

The substitution-based macro system has the virtue of providing an intuitive, alpha substitution-based solution to the hygiene problem. Unfortunately, its implementation as suggested above is too expensive. The expense arises from the desire to make pairs transparent to hygiene operations. To maintain this transparency, every mark or substitution operation must be propagated immediately to all the identifiers in an expression. Consequently, the overhead incurred by the hygienic algorithm at each expansion step that uses these operations is proportional to the size of the expression, compared to constant overhead in a traditional system. This is precisely the source of the complexity problem for the KFFD algorithm.

We solve this problem by making substitutions and markings "lazy" on structured expressions. Eventually, however, someone must do the work of propagating identifier operations. If *Exp* were being used only as syntax, it would be reasonable to let structure accessors do the work. For instance, we could have:

$$car(mark(cons(e_1, e_2), m)) = mark(e_1, m).$$

However, rather than alter the definitions of *car* and other accessors, we provide an accessor that exposes the outermost structure of an expression by pushing identifier information down to its constituent parts:

$$\begin{array}{rcl} expose & : & Exp \to Exp \\ expose(i) & = & strip(i) \\ expose(mark(cons(e_1, e_2), m)) & = & cons(mark(e_1, m), mark(e_2, m)) \\ & etc. & \end{array}$$

The functionality of *expose* is required only in the implementation of **syntax-case**, which uses it to destructure the input value as far as necessary to match against the input patterns.

It remains to construct a concrete *Exp* algebra. Expressions with pending substitutions or marks can be represented as distinguished triples (*wrapped expressions*) of the form $\langle e, r, \{m \ldots\}\rangle$, where e is an expression, r is an environment mapping *Ident* to *Sym*, and $\{m \ldots\}$ is a set of zero or more marks. Pairs, symbols and constants can use traditional representations. To avoid complexity problems, additional constraints must be imposed. In particular, the expression component of a wrapped expression must not be another wrapped expression. This property can be maintained by having *expose* combine mark sets and environments when it pushes a wrapping onto another wrapped expression. Since marks "stick" only to new elements introduced by macro transformers, a wrapped expression will have more than one mark only if it is generated by a macro whose definition was itself generated by a macro. In practice the mark field of a wrapped expression rarely has more than one mark. Consequently, handling marks is cheap and the complexity problems caused by "eager" mark propagation in the KFFD algorithm are avoided.

4.5 Source-object correlation

Finally, we note that the lazy substitution model can be adapted easily to support source-object correlation. We allow an expression to be annotated with information about its source by extending *Exp* with additional constructor:

$$source : Exp \times Annotation \to Exp$$

An annotation ($a \in Annotation$) is an unspecified data structure that provides information about the source of an expression, such as its location in a file.

Source annotations can be passed along by the expander to the evaluator, where they can be used to provide debugging information. Thus we might add to the definition of *expand*:

$$expand(source(e, a), r) = source(expand(e, r), a)$$

The expander can also use source annotations to report errors it detects.

Otherwise, operations ignore or drop source annotations. For instance:

$$\begin{array}{rcl} expose(source(e, a)) & = & expose(e) \\ mark(source(e, a), m) & = & source(mark(e, i), m) \\ id?(source(e, a)) & = & id?(e) \\ resolve(source(i, a)) & = & resolve(i) \end{array}$$

Since *expose* drops annotations, they are invisible to procedures that need to examine the structure of an expression. Previously constants were the sole class of expressions unaffected by syntactic operations. However, since constants can also be annotated, they too must be "exposed" before they can be examined. Source annotations can be implemented by adding another field to the wrapped expression structure of Section 4.4.

5 Conclusions

Our system, syntactic closures as augmented by Hanson [2, 12], and the Clinger and Rees "explicit renaming" system [4, 3] are all compatible with the "high-level" facility (**syntax-rules**) described in the Revised^4 Report on Scheme [5]. Thus, the three systems differ primarily in the treatment of "low-level" macros. Our system extends automatic hygiene and referential transparency to the low level, whereas the other systems require explicit renaming of identifiers or construction of syntactic closures, which can be tedious and error-prone. In addition, we have extended the automatic syntax checking, input destructuring, and output restructuring previously available only to high-level macros into the low level. In fact, our system draws no distinction between high- and low-level macros, so there is never a need to completely rewrite a macro originally written in a high-level style because it needs to perform some low-level operation. We have also provided a mechanism for correlating source and object code and introduced a hygiene-preserving mechanism for controlled variable capture, both of which are unique to our system.

An important aspect of our work is its thoroughgoing treatment of identifiers. Since identifiers cannot be treated as simple symbolic data in hygienic systems, the macro writer must be given tools that respect their essential properties. We provide tools for introducing new identifiers in a hygienic and referentially-transparent manner, for constructing macros that implicitly bind identifiers, and for comparing identifiers according to their intended use as free identifiers, bound identifiers, or symbolic data.

Although our work is designed to provide a macro system with automatic hygiene for Scheme and other Lisp dialects, it could be adapted to languages in which programs and data do not share the same structure. Rather than providing mechanisms to translate syntactic structures to and from standard data structures, accessors and constructors for the different types of syntax objects necessary to express the syntactic constructs of the language must be provided.

The macro system described in this paper has been fully implemented, and the implementation is available via "ftp" from *iuvax.cs.indiana.edu*. Contact the second author for details.

References

[1] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier Science Publishers, revised edition, 1984.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 86–95, July 1988.

[3] William Clinger. Hygienic macros through explicit renaming. *LISP Pointers*, 4(4), 1991.

[4] William Clinger and Jonathan Rees. Macros that work. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, January 1991.

[5] William Clinger, Jonathan A. Rees, et al. The revised^4 report on the algorithmic language Scheme. *LISP Pointers*, 4(3), 1991.

[6] D. Coutant, S. Meloy, and M. Ruscetta. Doc: A practical approach to source-level debugging of globally optimized code. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 125–134, July 1988.

[7] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[8] R. Kent Dybvig. Writing hygienic macros in scheme with syntax-case. Technical Report 356, Indiana Computer Science Department, June 1992.

[9] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: Beyond conventional macros. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 143–150, 1986.

[10] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: A general macro mechanism. *Lisp and Symbolic Computation*, 1(1):53–75, 1988.

[11] Timothy G. Griffin. *Definition and Top-Down Refinement for Interactive Proof Development Systems.* PhD thesis, Cornell University, August 1988.

[12] Chris Hanson. A syntactic closures macro facility. *LISP Pointers*, 4(4), 1991.

[13] J. Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, July 1982.

[14] Eugene Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, Bloomington, August 1986.

[15] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, 1986.

[16] Guy L. Steele Jr. *Common Lisp, the Language.* Digital Press, second edition, 1990.

[17] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[18] P. Zellweger. An interactive high-level debugger for control-flow optimized programs. In *Proceedings of the ACM Software Engineering Symposium on High-Level Debugging*, pages 159–171, August 1983.