Macros That Work

William Clinger
Department of Computer Science
University of Oregon

Jonathan Rees
Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

This paper describes a modified form of Kohlbecker's algorithm for reliably hygienic (capture-free) macro expansion in block-structured languages, where macros are source-to-source transformations specified using a high-level pattern language. Unlike previous algorithms, the modified algorithm runs in linear instead of quadratic time, copies few constants, does not assume that syntactic keywords (e.g. `if`) are reserved words, and allows local (scoped) macros to refer to lexical variables in a referentially transparent manner.

Syntactic closures have been advanced as an alternative to hygienic macro expansion. The problem with syntactic closures is that they are inherently low-level and therefore difficult to use correctly, especially when syntactic keywords are not reserved. It is impossible to construct a pattern-based, automatically hygienic macro system on top of syntactic closures because the pattern interpreter must be able to determine the syntactic role of an identifier (in order to close it in the correct syntactic environment) before macro expansion has made that role apparent.

Kohlbecker's algorithm may be viewed as a book-keeping technique for deferring such decisions until macro expansion is locally complete. Building on that insight, this paper unifies and extends the competing paradigms of hygienic macro expansion and syntactic closures to obtain an algorithm that combines the benefits of both.

Several prototypes of a complete macro system for Scheme have been based on the algorithm presented here.

1 Introduction

A macro is a source-to-source program transformation specified by programmers using a high-level pattern language. Macros add a useful degree of syntactic extensibility to a programming language, and provide a convenient way to abbreviate common programming idioms.

Macros are a prominent component of several popular programming languages, including C [7] and Common Lisp [11]. The macro systems provided by these languages suffer from various problems that make it difficult and in many cases impossible to write macros that work correctly regardless of the context in which the macros are used. It is widely believed that these problems result from the very nature of macro processing, so that macros are an inherently unreliable feature of programming languages [2].

That is not so. A macro system based upon the algorithm described in this paper allows reliable macros to be written, and is simple to use. The algorithm is nearly as fast as algorithms in current use.

This introduction illustrates several problems common to existing macro systems, using examples written in C and Scheme [10]. With one exception, all of these problems are solved automatically and reliably by the algorithm presented in this paper.

The original preprocessor for C allowed the body of a macro definition to be an arbitrary sequence of characters. It was therefore possible for a macro to expand into part of a token. For example,

```
#define foo "salad
printf(foo bar");
```

would expand into `printf("salad bar");`. We may say that this macro processor *failed to respect the integrity of lexical tokens.* This behavior caused so many problems that it has been banished from ANSI C except when special operators are used for the specific purpose of violating the integrity of lexical tokens.

Similar problems arise in macro systems that allow the body of a macro to expand into an arbitrary sequence of tokens. For example,

```
#define discriminant(a,b,c) b*b-4*a*c
  2*discriminant(x-y,x+y,x-y)*5
```

expands into `2*x+y*x+y-4*x-y*x-y*5`, which is then parsed as

```
(2*x)+(y*x)+y-(4*x)-(y*x)-(y*5)
```

instead of the more plausible

```
2*(x+y)*(x+y)-4*(x-y)*(x-y)*5.
```

We may say that systems such as the ANSI C preprocessor *fail to respect the structure of expressions.* Experienced C programmers know they must defend against this problem by placing parentheses around the macro body and around all uses of macro parameters, as in

```
#define discriminant(a,b,c) ((b)*(b)-4*(a)*(c))
```

This convention is not enforced by the C language, however, so the author of a macro who neglects this precaution is likely to create subtle bugs in programs that use the macro.

Another problem with the `discriminant` macro occurs when the actual parameter for `b` is an expression that has a side effect, as in `discriminant(3,x--,2)`. Of the problems listed in this introduction, the inappropriate duplication of side effects is the only one that is not solved automatically by our algorithm.

One can argue that actual parameter expressions should not have side effects, but this argument runs counter to the C culture. In a fully block-structured language such as Scheme, the author of a macro can defend against actual parameters with side effects by introducing a local variable that is initialized to the value obtained by referencing the macro parameter. In the syntax we adopt for this paper, which differs slightly from syntax that has been proposed for Scheme [3], a correct version of the `discriminant` macro can be defined by

```
(define-syntax discriminant
  (syntax-rules
    ((discriminant ?a ?b ?c)
     => (let ((temp ?b))
          (- (* temp temp) (* 4 ?a ?c))))))
```

Unfortunately there is no equivalent to this macro in C, since blocks are not permitted as expressions.

From the above example we see that macros must be able to introduce local variables. Local variables, however, make macros vulnerable to accidental conflicts of names, as in

```
#define swap(v,w) {int temp=(v); \
     (v) = (w); (w) = temp;}
  int temp = thermometer();
  if (temp < lo_temp) swap(temp, lo_temp);
```

Here the `if` statement *never* exchanges the values of `temp` and `lo_temp`, because of an accidental name clash. The macro writer's usual defense is to use an obscure name and *hope* that it will not be used by a client of the macro:

```
#define swap(v,w) {int __temp_obscure_name=(v); \
     (v) = (w); (w) = __temp_obscure_name;}
```

Even if the macro writer is careful to choose an obscure name, this defense is less than reliable. For example, the macro may be used within some other use of a macro whose author happened to choose the same obscure name. This problem can be especially perplexing to authors of recursive macros.

The most troublesome name clashes arise not from variables local to a macro, but from the very common case of a macro that needs to refer to a global variable or procedure:

```
#define complain(msg) print_error(msg);
  if (printer_broken()) {
      char *print_error = "The printer is broken";
      complain(print_error);
      }
```

Here the name given to the error message conflicts with the name of the procedure that prints the message. In this example the compiler will report a type error, but programmers are not always so lucky.

In most macro systems there is no good way to work around this problem. Programs would become less readable if obscure names were chosen for global variables and procedures, and the author of a macro probably does not have the right to choose those names anyway. Requiring the client of each macro to be aware of the names that occur free in the macro would defeat the purpose of macros as syntactic abstractions.

The examples above show that at least two classes of inadvertent name conflicts can be created by macro processors that *fail to respect the correlation between bindings and uses of names.* One class involves macros that introduce bound (local) variables. Another class involves macros that contain free references to variables or procedures.

Hygienic macro expansion [8] is a technology that maintains the correlation between bindings and uses of names, while respecting the structure of tokens and expressions. The main problem with hygienic macro expansion has been that the algorithms have required $O(n^2)$ time, where n is the time required by naive macro expansion as in C or Common Lisp. Although this is the worst case, it sometimes occurs in practice.

The purpose of this paper is to build on Kohlbecker's algorithm for hygienic macro expansion by incorporating ideas developed within the framework of syntactic closures [1]. Our result is an efficient and more complete solution to the problems of macro processing. Our algorithm runs in $O(n)$ time and solves several additional problems associated with local (scoped) macros that had not been addressed by hygienic macro expansion. In particular, our algorithm supports referentially transparent local macros as described below.

If macros can be declared within the scope of lexical variables, as in Common Lisp, then a seemingly new set of problems arises. We would like for free variables that occur on the right hand side of a rewriting rule for a macro to be resolved in the lexical environment of the macro definition instead of being resolved in the lexical environment of the use of the macro. Using `let-syntax` to declare local macros, for example, it would be nice if

```
(let-syntax ((first
              (syntax-rules
                ((first ?x) => (car ?x))))
             (second
              (syntax-rules
                ((second ?x) => (car (cdr ?x))))))
  (let ((car "duesenberg"))
    (let-syntax ((classic
                  (syntax-rules
                    ((classic) => car))))
      (let ((car "yugo"))
        (let-syntax ((affordable
                      (syntax-rules
                        ((affordable) => car))))
          (let ((cars (list (classic)
                            (affordable))))
            (list (second cars)
                  (first cars))))))))
```

could be made to evaluate to `("yugo" "duesenberg")`. This would be the case if the references to `car` that are introduced by the `first`, `second`, `classic`, and `affordable` macros referred in each case to the `car` variable within whose scope the macro was defined. In other words, this example would evaluate to `("yugo" "duesenberg")` if local macros were *referentially transparent.*

Our algorithm supports referentially transparent local macros. We should note that these macros may not be referentially transparent in the traditional sense because of side effects and other problems caused by the programming language whose syntax they extend, but we blame the language for this and say that the macros themselves are referentially transparent.

Now consider the following Scheme code:

```
(define-syntax push
  (syntax-rules
    ((push ?v ?x) => (set! ?x (cons ?v ?x)))))

(let ((pros (list "cheap" "fast"))
      (cons (list)))
  (push "unreliable" cons))
```

It is easy to see that the `cons` procedure referred to within the macro body is different from the `cons` variable within whose scope the macro is used. Through its reliance on the meaning of the syntactic keyword `set!`, the `push` macro also illustrates yet a third class of inadvertent name conflicts. If syntactic keywords are not reserved, then the `push` macro might be used within the scope of a local variable or macro named `set!`.

We can now identify four classes of *capturing* problems. The first class involves inadvertent capturing by bound variables introduced by a macro. The second class involves the inadvertent capture of free variable references introduced by a macro. The third is like the first, but involves inadvertent capturing by bindings of syntactic keywords (i.e. local macros) introduced by a macro. The fourth is like the second, but involves the inadvertent capture of references to syntactic keywords introduced by a macro.

Syntactic closures [1] have been advanced as an alternative to hygienic macro expansion. Syntactic closures can be used to eliminate all four classes of capturing problems, and can support referentially transparent local macros as well. Why then have we bothered to develop a new algorithm for hygienic macro expansion, when syntactic closures also run in linear time and solve all the problems that are solved by our algorithm?

The problem with syntactic closures is that they do not permit macros to be defined using the kind of high-level pattern language that is used in C and is proposed for Scheme [6]. The difficulty is that a pattern interpreter that uses syntactic closures must be able to determine the syntactic role of an identifier (to know whether to close over it, and if so to know the correct syntactic environment) before macro expansion has made that role apparent. Consider a simplified form of the `let` macro, which introduces local variables:

```
(define-syntax let
  (syntax-rules
    ((let ((?name ?val)) ?body)
     => ((lambda (?name) ?body) ?val))))
```

When this macro is used, the `?val` expression must be closed in the syntactic environment of the use, but the `?body` cannot simply be closed in the syntactic environment of the use because its references to `?name` must be left free. The pattern interpreter cannot make this distinction unaided until the lambda expression is expanded, and even then it must somehow remember that the bound variable of the lambda expression came from the original source code and must therefore be permitted to capture the references that occur in `?body`. Recall from the `swap` example that a reliable macro system must *not* permit bound variables introduced by macro expansion to capture such references.

Kohlbecker's algorithm can be viewed as a book-keeping technique for deferring all such decisions about the syntactic roles of identifiers until macro expansion is locally complete and has made those roles evident. Our algorithm is therefore based on Kohlbecker's but incorporates the idea of syntactic environments, taken from the work on syntactic closures, in order to achieve referentially transparent local macros.

$$\begin{aligned}
\text{env} &= \text{identifier} \to \text{denotation}\\
\text{denotation} &= \text{special} + \text{macro} + \text{identifier}\\
\text{special} &= \{\mathit{lambda}, \mathit{let\text{-}syntax}\}\\
\text{macro} &= (\text{pattern} \times \text{rewrite})^{+} \times \text{env}
\end{aligned}$$

$$\begin{aligned}
\mathit{ident} &\in \text{env}\\
\mathit{lookup} &\in \text{env} \times \text{identifier} \to \text{denotation}\\
\mathit{bind} &\in \text{env} \times \text{identifier} \times \text{denotation} \to \text{env}\\
\mathit{divert} &\in \text{env} \times \text{env} \to \text{env}
\end{aligned}$$

$$\begin{aligned}
\mathit{lookup}\,(\mathit{ident}, x) &= x\\
\mathit{lookup}\,(\mathit{bind}\,(e, x, v), x) &= v\\
\mathit{lookup}\,(\mathit{bind}\,(e, x, v), y) &= \mathit{lookup}\,(e, y) \qquad \text{if } x \neq y\\
\mathit{divert}\,(e, \mathit{ident}) &= e\\
\mathit{divert}\,(e, \mathit{bind}\,(e', x, v)) &= \mathit{bind}\,(\mathit{divert}\,(e, e'), x, v)
\end{aligned}$$

Figure 1: Syntactic environments.

The question arises: Don't the capturing problems occur simply because we use names instead of pointers, and trees instead of dags? If we first parsed completely before doing macro expansion, could we not replace all occurrences of names by pointers to symbol table entries and thereby eliminate the capturing problems?

The answer is no. One of the most important uses of macros is to provide syntactically convenient binding abstractions. The distinction between binding occurrences and other uses of identifiers should be determined by a particular syntactic abstraction, not predetermined by the parser. Therefore, even if we were to parse actual parameters into expression trees before macro expansion, the macro processor still could not reliably distinguish bindings from uses and correlate them until macro expansion is performed.

Consider an analogy from lambda calculus. In reducing an expression to normal form by textual substitution, it is sometimes necessary to rename variables as part of a beta reduction. It doesn't work to perform all the (naive) beta reductions first, without renaming, and then to perform all the necessary alpha conversions; by then it is too late. Nor does it work to do all the alpha conversions first, because beta reductions introduce new opportunities for name clashes. The renamings must be *interleaved* with the (naive) beta reductions, which is the reason why the notion of substitution required by the non-naive beta rule is so complicated.

The same situation holds for macro expansions. It does not work to simply expand all macro calls and then rename variables, nor can the renamings be performed before expansion. The two processes must be interleaved in an appropriate manner. A correct and efficient realization of this interleaving is our primary contribution.

2 The algorithm

Our algorithm avoids inadvertent captures by guaranteeing the following *hygiene condition*:

$$\frac{\mathit{lookup}\,(e,\mathbf{x}) \in \text{identifier}}{e \vdash \mathbf{x} \rightarrow \mathit{lookup}\,(e,\mathbf{x})} \qquad [\textit{variable references}]$$

$$\frac{\begin{array}{c}\mathit{lookup}\,(e,\mathbf{k}_0) = \mathit{lambda}\\ \mathit{bind}\,(e,\mathbf{x},\mathbf{x}') \vdash \mathbf{E} \rightarrow \mathbf{E}' \text{ (where } \mathbf{x}' \text{ is a fresh identifier)}\end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{(}\mathbf{x}\texttt{)}\ \mathbf{E}\texttt{)} \rightarrow \texttt{(lambda (}\mathbf{x}'\texttt{)}\ \mathbf{E}'\texttt{)}} \qquad [\textit{procedure abstractions}]$$

$$\frac{e \vdash \mathbf{E}_0 \rightarrow \mathbf{E}_0',\quad e \vdash \mathbf{E}_1 \rightarrow \mathbf{E}_1'}{e \vdash \texttt{(}\mathbf{E}_0\ \mathbf{E}_1\texttt{)} \rightarrow \texttt{(}\mathbf{E}_0'\ \mathbf{E}_1'\texttt{)}} \qquad [\textit{procedure calls}]$$

$$\frac{\begin{array}{c}\mathit{lookup}\,(e,\mathbf{k}_0) = \mathit{let\text{-}syntax}\\ \mathit{bind}\,(e,\mathbf{k},\langle\tau,e\rangle) \vdash \mathbf{E} \rightarrow \mathbf{E}'\end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{((}\mathbf{k}\ \tau\texttt{))}\ \mathbf{E}\texttt{)} \rightarrow \mathbf{E}'} \qquad [\textit{macro abstractions}]$$

$$\frac{\begin{array}{c}\mathit{lookup}\,(e,\mathbf{k}) = \langle\tau,e'\rangle\\ \mathit{transcribe}\,(\texttt{(}\mathbf{k}\ \ldots\texttt{)},\tau,e,e') = \langle\mathbf{E},e''\rangle\\ e'' \vdash \mathbf{E} \rightarrow \mathbf{E}'\end{array}}{e \vdash \texttt{(}\mathbf{k}\ \ldots\texttt{)} \rightarrow \mathbf{E}'} \qquad [\textit{macro calls}]$$

Figure 2: The modified Kohlbecker algorithm.

$$\frac{\begin{array}{c}\mathit{match}\,(\mathbf{E},\pi,e_{use},e_{def}) = \mathit{nomatch}\\ \mathit{transcribe}\,(\mathbf{E},\tau',e_{use},e_{def}) = \langle\mathbf{E}',e'\rangle\end{array}}{\mathit{transcribe}\,(\mathbf{E},\langle\langle\pi,\rho\rangle,\tau'\rangle,e_{use},e_{def}) = \langle\mathbf{E}',e'\rangle}$$

$$\frac{\begin{array}{c}\mathit{match}\,(\mathbf{E},\pi,e_{use},e_{def}) = \sigma\\ \mathit{rewrite}\,(\rho,\sigma,e_{def}) = \langle\mathbf{E}',e_{new}\rangle\end{array}}{\mathit{transcribe}\,(\mathbf{E},\langle\langle\pi,\rho\rangle,\tau'\rangle,e_{use},e_{def}) = \langle\mathbf{E}',\mathit{divert}\,(e_{use},e_{new})\rangle}$$

$\mathit{transcribe}\,(\mathbf{E},\langle\ \rangle,e_{use},e_{def})$ is an error.

Figure 3: Definition of $\mathit{transcribe}\,(\mathbf{E},\tau,e_{use},e_{def})$.

1. It is impossible to write a high-level macro that inserts a binding that can capture references other than those inserted by the macro.

2. It is impossible to write a high-level macro that inserts a reference that can be captured by bindings other than those inserted by the macro.

These two properties can be taken as the defining properties of hygienic macro expansion. The hygiene condition is quite strong, and it is sometimes necessary to write non-hygienic macros; for example, the `while` construct in C implicitly binds `break`, so if `while` were implemented as a macro then it would have to be allowed to capture references to `break` that it did not insert. We here ignore the occasional need to escape from hygiene; we have implemented a compatible low-level macro system in which non-hygienic macros can be written.

The reason that previous algorithms for hygienic macro expansion are quadratic in time is that they expand each use of a macro by performing a naive expansion followed by an extra scan of the expanded code to find and paint (i.e. rename, or time-stamp) the newly introduced identifiers. If macros expand into uses of still other macros with more or less the same actual parameters, which often happens, then large fragments of code may be scanned anew each time a macro is expanded. Naive macro expansion would scan each fragment of code only once, when the fragment is itself expanded.

Our algorithm runs in linear time because it finds the newly introduced identifiers by scanning the rewrite rules, and paints these identifiers as they are introduced during macro expansion. The algorithm therefore scans the expanded code but once, for the purpose of completing the recursive expansion of the code tree, just as in the naive macro expansion algorithm. The newly introduced identifiers can in fact be determined by scanning the rewrite rules at macro definition time, but this does not affect the asymptotic complexity of the algorithm.

The more fundamental difference between our algorithm and previous algorithms lies in the book-keeping needed to support referentially transparent local macros. The syntactic environments manipulated by this book-keeping are shown in Figure 1. These are essentially the same as the environments used by syntactic closures, but the book-keeping performed by our algorithm allows it to defer decisions that involve the syntactic roles of identifiers.

The overall structure of the algorithm is shown formally in Figure 2 for the lambda calculus subset of Scheme, extended by macro calls and a restricted form of `let-syntax`. The notation "$e \vdash \mathbf{E} \rightarrow \mathbf{E}'$" indicates that $\mathbf{E}'$ is the completely macro-expanded expression that results from expanding $\mathbf{E}$ in the syntactic environment e. For this simple language the initial syntactic environment e_{init} is

$$bind\,(bind\,(ident, \texttt{lambda}, lambda), \texttt{let-syntax}, let\text{-}syntax).$$

The variables bound by a procedure abstraction are renamed to avoid shadowing outer variables. Identifiers that denote variable references must therefore be renamed as well. Procedure calls are straightforward (but would be less so if expressions were allowed to expand into syntactic keywords; we have implemented this both ways). The rule for macro abstractions is also straightforward: the body is macro-expanded in a syntactic environment in which the syntactic keyword bound by the abstraction denotes the macro obtained from the transformation function τ (a set of rewrite rules) by closing τ in the syntactic environment of the macro definition.

The rule for macro calls is subtle. The macro call is transcribed using the transformation function τ obtained from the macro being called, the syntactic environment of the call, and the syntactic environment in which the macro was defined. This transcription yields not only a new expression $\mathbf{E}$ but a new syntactic environment e'' in which to complete the macro expansion. The key fact about algorithms for hygienic macro expansion is that any identifiers introduced by the macro are renamed, so the macro can only introduce fresh identifiers. The new syntactic environment binds these fresh identifiers to the denotations of the corresponding original identifiers in the syntactic environment of the macro definition. These will be the ultimate denotations of the fresh identifiers unless subsequent macro expansion exposes an intervening procedure abstraction that binds them. Since the identifiers were fresh, any such binding must have been introduced by the same macro call that introduced the references. Hence the hygiene condition will be satisfied.

Figure 3 defines the *transcribe* function in terms of *match* and *rewrite*. The *transcribe* function simply loops through each rewrite rule π `=>` ρ until it finds one whose pattern π matches the macro call. It then rewrites the macro call according to ρ and adds new bindings for any identifiers introduced by the rewrite to the syntactic environment of the use.

Actually, rather than matching the entire macro call to the pattern, it is sufficient to match only the tail (or "actual parameters") of the call against the tail ("formal parameters") of the pattern. This complication is not reflected in Figure 3.

The *match* function (Figure 4) delivers a substitution σ mapping pattern variables to components of the input expression. It is quite conventional except for one important wrinkle: Matching an identifier x in the macro use against a literal identifier y in the pattern succeeds if and only if $lookup\,(e_{use}, x) = lookup\,(e_{def}, y)$. This implies that identifiers in patterns are lexically scoped: bindings that intervene between the definition and use of a macro may cause match failure.

For simplicity, the definition of *match* in Figure 4 assumes that no pattern variable is duplicated.

The *rewrite* function (Figure 5) rewrites the macro call using the substitution σ generated by the matcher. It is also responsible for choosing the fresh identifiers that replace those that appear in the output pattern and establishing their proper denotations. *transcribe* uses *divert* to add these new bindings to the environment in which the macro output is expanded.

$$match\,(\mathbf{E}, \mathbf{?v}, e_{use}, e_{def}) = \{\mathbf{?v} \mapsto \mathbf{E}\}$$

$$\frac{lookup\,(e_{def}, \mathbf{x}') = lookup\,(e_{use}, \mathbf{x})}{match\,(\mathbf{x}, \mathbf{x}', e_{use}, e_{def}) = \{\}}$$

$$\frac{\begin{array}{c} match\,(\mathbf{E}_i, \pi_i, e_{use}, e_{def}) = \sigma_i,\ i = 1, \ldots, n \\ \sigma_i \neq nomatch,\ i = 1, \ldots, n \end{array}}{match\,((\mathbf{E}_1 \ldots \mathbf{E}_n), (\pi_1 \ldots \pi_n), e_{use}, e_{def}) = \sigma_1 \cup \ldots \cup \sigma_n}$$

$$match\,(\mathbf{E}, \pi, e_{use}, e_{def}) = nomatch, \quad \text{if the other rules are not applicable}$$

Figure 4: Definition of $match\,(\mathbf{E}, \pi, e_{use}, e_{def})$.

$$rewrite\,(\rho, \sigma, e_{def}) = \langle rewrite'(\rho, \sigma'), e_{new}\rangle,$$

where

$\sigma' = \sigma \cup \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$,
$x_1, \ldots, x_n$ are all the identifiers that occur in ρ,
$x'_1, \ldots x'_n$ are fresh identifiers,
$e_{new} = bind\,(\ldots bind\,(ident, x'_1, d_1) \ldots, x'_n, d_n)$,
and $d_i = lookup(e_{def}, x_i)$ for $i = 1, \ldots, n$.

$$rewrite'(\mathbf{?v}, \sigma) = \sigma(\mathbf{?v})$$

$$rewrite'(\mathbf{x}, \sigma) = \sigma(\mathbf{x})$$

$$rewrite'((\pi_1 \ldots \pi_n), \sigma) = (\mathbf{E}'_1 \ldots \mathbf{E}'_n),$$

where $\mathbf{E}'_i = rewrite'(\pi_i, \sigma),\ i = 1, \ldots, n$.

Figure 5: Definition of $rewrite\,(\rho, \sigma, e_{def})$.

Given constant-time operations on environments, this algorithm takes $O(n)$ time, where n is the time required by naive macro expansion.

3 Examples

To make the examples easier to follow, we'll assume that `let` is understood primitively by the expansion algorithm, in a manner similar to `lambda`. The initial environment e_{init} thus has a nontrivial binding for the identifier `let` in addition to `lambda` and `let-syntax`.

Example 1. Consider the problem of expanding the following expression in the initial syntactic environment:

```
(let-syntax ((push (syntax-rules
                    ((push ?v ?x)
                     => (set! ?x (cons ?v ?x))))))
  (let ((pros (list "cheap" "fast"))
        (cons (list)))
    (push "unreliable" cons)))
```

The rule for macro abstractions applies, because

$$lookup\,(e_{init}, \texttt{let-syntax}) = let\text{-}syntax.$$

The next step is to expand the body of the `let-syntax` expression in an augmented syntactic environment

$$e_1 = \mathit{bind}(e_{init}, \texttt{push}, \langle \tau, e_{init} \rangle)$$

where

$$\tau = \langle\langle \texttt{(?v ?x)}, \texttt{(set! ?x (cons ?v ?x))} \rangle\rangle$$

The pattern is `( ?v ?x)`, rather than `(push ?v ?x)`, because the rule will only be examined when the fact that the head of the form denotes the `push` macro is already apparent. Thus there is never a need to match the head of the pattern with the head of the expression to be expanded.

The initializers in the `let` expression expand to themselves, because $e_1 \vdash \texttt{list} \rightarrow \texttt{list}$. (Recall that e_{init}, and therefore e_1 as well, is based on the identity environment *ident*.)

Next we expand the body of the `let` expression in the augmented environment

$$e_2 = \mathit{bind}(\mathit{bind}(e_1, \texttt{cons}, \texttt{cons.1}), \texttt{pros}, \texttt{pros.1})$$

where `pros.1` and `cons.1` are fresh identifiers. In e_2, the identifier `push` denotes the macro $\langle \tau, e_{init} \rangle$, so the rule for macro calls applies to the `push` expression. We now compute

$$\mathit{transcribe}(\texttt{(push "unreliable" cons)}, \tau, e_2, e_{init})$$

τ consists of only one rule, which matches the input:

$$\begin{aligned} &\mathit{match}(\texttt{("unreliable" cons)}, \texttt{(?v ?x)}, e_2, e_{init}) \\ &\quad = \{\texttt{?v} \mapsto \texttt{"unreliable"}, \texttt{?x} \mapsto \texttt{cons}\} \end{aligned}$$

Note that *match* made no use of its two environment arguments, because the pattern `( ?v ?x)` didn't contain any identifiers. The `cond` example below gives a situation in which these environments are used.

Now we compute the replacement expression:

$$\begin{aligned} &\mathit{rewrite}(\texttt{(set! ?x (cons ?v ?x))}, \\ &\qquad \{\texttt{?v} \mapsto \texttt{"unreliable"}, \texttt{?x} \mapsto \texttt{cons}\}, \\ &\qquad e_{init}) \\ &\quad = \langle \texttt{(set!.2 cons (cons.2 "unreliable" cons))}, e_3 \rangle \end{aligned}$$

where

$$e_3 = \mathit{bind}(\mathit{bind}(\mathit{ident}, \texttt{set!.2}, \texttt{set!}), \texttt{cons.2}, \texttt{cons})$$

and `set!.2` and `cons.2` are fresh identifiers. *transcribe* now delivers the rewritten expression together with an environment

$$\begin{aligned} e_4 &= \mathit{divert}(e_2, e_3) \\ &= \mathit{bind}(\mathit{bind}(e_2, \texttt{set!.2}, \texttt{set!}), \texttt{cons.2}, \texttt{cons}) \end{aligned}$$

that gives bindings to use in expanding the replacement expression. The environment e_4 takes `set!.2` to `set!`, `cons.2` to `cons`, and `cons` to `cons.1`, so

$$\begin{aligned} e_4 \vdash\ &\texttt{(set!.2 cons (cons.2 "unreliable" cons))} \\ &\rightarrow \texttt{(set! cons.1 (cons "unreliable" cons.1))} \end{aligned}$$

The final result is

```
(let ((pros.1 (list "cheap" "fast"))
      (cons.1 (list)))
  (set! cons.1 (cons "unreliable" cons.1)))
```

Example 2. This example illustrates reliable reference to local variables that are in scope where the macro is defined.

```
(let ((x "outer"))
  (let-syntax ((m (syntax-rules
                    ((m) => x))))
    (let ((x "inner"))
      (m))))
```

To expand this expression in e_{init}, a fresh identifier `x.1` is chosen to replace the outer `x`, and the `let-syntax` expression is expanded in the syntactic environment

$$e_1 = \mathit{bind}(e_{init}, \texttt{x}, \texttt{x.1})$$

This leads to expanding the inner `let` expression in the syntactic environment

$$e_2 = \mathit{bind}(e_1, \texttt{m}, \langle\langle\langle \texttt{( )}, \texttt{x} \rangle\rangle, e_1 \rangle)$$

Finally a fresh identifer `x.2` is chosen to replace the inner `x`, and `(m)` is expanded in the syntactic environment

$$e_3 = \mathit{bind}(e_2, \texttt{x}, \texttt{x.2})$$

Now

$$\mathit{transcribe}(\texttt{(m)}, \langle\langle\langle \texttt{( )} \rangle, \texttt{x} \rangle\rangle, e_3, e_1) = \langle \texttt{x.3}, \mathit{bind}(e_3, \texttt{x.3}, \texttt{x.1}) \rangle$$

where `x.3` is a fresh identifer introduced for the right-hand side of the rewrite rule for the macro `m`. The denotation of `x.3` is the denotation of `x` in the environment of `m`'s definition, which is `x.1`. The final expansion is

```
(let ((x.1 "outer"))
  (let ((x.2 "inner"))
    x.1))
```

Example 3. This example illustrates lexical scoping of constant identifiers that occur in the left-hand side of rewrite rules. Following [9], we adopt the use of an ellipsis token `...` as part of the syntax of patterns, not as a meta-notation indicating that something has been elided from this example.

```
(define-syntax cond
  (syntax-rules
    ((cond) => #f)
    ((cond (else ?result ...) ?clause ...)
     => (begin ?result ...))
    ((cond (?test) ?clause ...)
     => (or ?test (cond ?clause ...)))
    ((cond (?test ?result ...) ?clause ...)
     => (if ?test
            (begin ?result ...)
            (cond ?clause ...)))))
```

The second pattern for this macro contains a fixed identifier `else`. This will only match a given identifer x in a use of `cond` if the denotation of x in the environment of use matches the denotation of `else` in the environment of `cond`'s definition. Typically x is `else` and is self-denoting in both environments. However, consider the following:

```
(let ((else #f))
  (cond (#f 3)
        (else 4)
        (#t 5)))
```

Expanding this expression requires calculating *match* (else, else, e_{use}, e_{def}) where

$$lookup\,(e_{use}, \texttt{else}) = \texttt{else.1},$$
$$lookup\,(e_{def}, \texttt{else}) = \texttt{else}.$$

Thus

$$match\,(\texttt{else}, \texttt{else}, e_{use}, e_{def}) = nomatch$$

and the final expansion is

```
(let ((else.1 #f))
  (if #f (begin 3)
      (if else.1 (begin 4)
          (if #t (begin 5) #f))))
```

4 Integration issues

We emphasize that our algorithm is suitable for use with any block-structured language, and does not depend on the representation of programs as lists in Scheme. Scheme's representation is especially convenient, however. This section explains how the algorithm can be made to work with less convenient representations.

The simplicity of the *match* function defined in Figure 4 results from the regularity of a Cambridge Polish syntax. For Algol-like syntaxes the matcher could be much more complicated. To avoid such complications, a macro system may eschew complex patterns and may specify a fixed syntax for macro calls.

Our algorithm must understand the structure of the source program, so Algol-like syntaxes require that the algorithm be integrated with a parser. If the macro language is sufficiently restricted, then it may be possible to parse the input program completely before macro expansion is performed. If the actual parameters of a macro call need to be transmitted to the macro in unparsed form, however, then parsing will have to be interleaved with the macro expansion algorithm.

For the algorithm to work at all, the parser must be able to locate the end of any macro definition or macro use. If parentheses are used to surround the actual parameters of a macro call, for example, then mismatched parentheses within an actual parameter cannot be tolerated unless they are somehow marked as such, perhaps by an escape character. This is a fundamental limitation on the generality of the syntactic transformations that can be described using a macro system based on our algorithm.

This is not a particularly burdensome limitation. For example, it is possible to design a hygienic macro system for C that allows the `for` construct to be described as a macro.

Experience with macros in Lisp and related languages has shown that macros are most useful when local variables can be introduced in any statement or expression context. Most Algol-like languages are not fully block-structured in this sense. C, for example, does not admit blocks as expressions, while Pascal and Modula-2 do not even admit blocks as statements. Fortunately this particular shortcoming can usually be overcome by the macro processor. Macros can be written as though the language admits blocks in all sensible contexts, and the macro processor itself can be responsible for replacing these blocks by their bodies while lifting all local declarations to the head of the procedure body within which the declarations appear.

In our algorithm, the matcher compares an identifier in the pattern against an identifier in the input by comparing their denotations. This makes it difficult to use such a sophisticated matcher in languages such as Common Lisp, where an identifier may denote many different things at once, and where the overloading is resolved by syntactic context. The problem is that the matcher cannot reliably determine the syntactic context in which the identifier will ultimately appear. One solution is to ban identifiers from patterns. Another is to resolve the overloading by relying on declarations provided by the author of the macro.

5 Previous work, current status, future work

The problems with naive macro expansion have been recognized for many years, as have the traditional work-arounds [2]. The capturing problems that afflict macro systems for block-structured languages were first solved by Kohlbecker's work on hygienic macro expansion. Bawden and Rees then proposed syntactic closures [1] as a more general and efficient but lower-level solution. Our algorithm unifies and extends this recent research, most of which has been directed toward the goal of developing a reliable macro system for Scheme.

At the 1988 meeting of the Scheme Report authors at Snowbird, Utah, a macro committee was charged with developing a hygienic macro facility akin to `extend-syntax` [5] but based on syntactic closures. Chris Hanson implemented a prototype and discovered that an implementation based on syntactic closures must determine the syntactic roles of some identifiers before macro expansion based on textual pattern matching can make those roles apparent [6]. Clinger observed that Kohlbecker's algorithm amounts to a technique for delaying this determination, and proposed a linear-time version of Kohlbecker's algorithm. Rees married syntactic closures to the modified Kohlbecker's algorithm and implemented it all, twice. Bob Hieb found some bugs in the first implementation and proposed fixes.

A high-level macro system similar to that described here is currently implemented on top of a compatible low-level system that is not described in this paper. Bob Hieb and Kent Dybvig have redesigned this low-level system to make it more abstract and easier to use, and have constructed yet another implementation. It is expected that both the high-level and low-level macro facilities will be described in a future report on Scheme [3].

Some problems remain. The algorithm we have described treats identifiers uniformly, but identifiers in Scheme are lexically indistinguishable from symbols that appear in constants. Consequently any symbols introduced by a macro will be renamed just as if they were identifiers, and must therefore be reverted after macro expansion has revealed that they are part of a constant. This means that constants introduced by a macro may have to be copied. In this respect our algorithm improves upon Kohlbecker's, which copied *all* constants, but is still not ideal.

More significantly, the pattern variables used to define macros might be lexically indistinguishable from identifiers and symbols. In order for macros that define other macros to remain referentially transparent, pattern variables must not be reverted to their original names even though they are represented as symbols in our existing implementation. We are not completely certain that this refinement eliminates all problems with macros that define other macros.

One project we intend to pursue is to integrate our algorithm with a module system for Scheme such as that described in [4]. For example, it should be possible for a module to export a macro without also having to export bindings needed by the macro's expansions.

An obvious application for this research is to develop better macro facilities for other block-structured languages such as Modula-2.

Acknowledgements

The authors thank an anonymous member of the program committee who helped us to write the introduction. Jim O'Toole and Mark Sheldon provided helpful comments on drafts of the paper.

References

[1] Alan Bawden and Jonathan Rees.
Syntactic closures.
1988 ACM Conference on Lisp and Functional Programming, pages 86–95.

[2] P. J. Brown.
Macro Processors and Techniques for Portable Software.
Wiley, 1974.

[3] William Clinger and Jonathan Rees, editors.
Revised^4 report on the algorithmic language Scheme.
University of Oregon Technical Report CIS-TR-90-02, to appear.

[4] Pavel Curtis and James Rauen.
A module system for Scheme.
1990 ACM Conference on Lisp and Functional Programming, pages 13–19.

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

[6] Chris Hanson.
Personal communication.

[7] Samuel P. Harbison and Guy L. Steele Jr.
C: A Reference Manual.
Prentice-Hall, second edition 1987.

[8] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba.
Hygienic macro expansion.
1986 ACM Conference on Lisp and Functional Programming, pages 151–159.

[9] Eugene E. Kohlbecker Jr.
Syntactic extensions in the programming language Lisp.
Technical report no. 199, Indiana University Computer Science Department, 1986.

[10] Jonathan A. Rees and William Clinger, editors.
Revised^3 report on the algorithmic language Scheme.
SIGPLAN Notices 21(12), pages 37–79, 1986.

[11] Guy L. Steele Jr.
Common Lisp: The Language.
Digital Press, second edition 1990.