Syntax Macros in M-LISP: A Representation Independent Dialect of LISP with Reduction Semantics*

(Extended Abstract)

Robert Muller†

Abstract

In this paper we present an efficient algorithm for avoiding unintended name captures during syntax macro transcription in LISP. The algorithm is a hybrid of Kohlbecker's *Macro-by-Example* and *Hygienic* algorithms adapted for a representation-independent dialect of LISP. The adaptation yields a substantially different model of syntax macros than is found in S-expression LISP dialects. The most important difference is that λ-binding patterns become apparent when an abstraction is first (i.e., partially) transcribed in the syntax tree. This allows us to avoid a larger class of name captures than is possible in S-expression dialects such as Scheme where λ-binding patterns are not apparent until the tree is completely transcribed.

1 Introduction

The subtle problem of *name capture* during syntax macro transcription in LISP has been discussed extensively in the literature [KFFD86, BR88, CR91]. Many of the recent attempts to come to grips with it are based on the observation in [KFFD86] that the problem is analogous to the classical problems with substitution in formal calculii. For example, if performed naively the substitution $(\lambda x.M)[y := N]$ results in capture if x should occur free in N (and y occurs free in M.) This is similar to the potential capture in the following *or* macro:

```
(extend-syntax (or)
 ((or e1 e2) => ((lambda (x) (if x x e2)) e1)))
```

A naive expansion of a call

*This paper appeared as Harvard University, CRCT Technical Report TR-04-90.

†The Author's address is: Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138. *email:* muller@harvard.edu. This research was supported in part by Defense Advanced Research Projects Agency N00039-88-C-0163.

```
(or M N) ==> ((lambda (x) (if x x N)) M)
```

results in an erroneous form whenever x occurs free in N. So by analogy with *hygienic* substitution in λ-calculus, the hygienic approach to macro expansion relies on an α-conversion process during transcription to ensure that these kinds of captures (what we call *vertical captures*) will not occur. It is shown that the algorithm respects the following *hygiene condition*: "Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated in the same transcription step." This approach has proved to be reasonably successful in guarding against this class of binding problems and it is the basis of several recent macro implementations in Scheme.

On the other hand, this characterization of the problem is lacking in some respects. It is possible to fully respect the hygiene condition but still have intended bindings broken during transcription. This is because the hygiene condition says nothing about identifiers that are introduced in the same step. For example, consider the macro:

```
(extend-syntax horizontal
  ((horizontal a (b c d)) => (b c (d a))))
```

which does some simple rearranging of its arguments. A macro call such as

```
(lambda (x) (horizontal x (lambda (y) f)))
```

transcribes to

```
==> (lambda (x) (lambda (y) (f x)))
```

On the other hand a call such as

```
(lambda (x) (horizontal x (lambda (x) f)))
```

results in a structurally different function

```
==> (lambda (x) (lambda (x) (f x)))
```

in which the associations between the binding and applied occurrences of the identifier `x` are changed. While this simple example might seem somewhat contrived, in general the identifiers in the two actual arguments to *horizontal* could have been produced in different branches of the same parallel transcription step (and therefore receive the same clock value) by textually remote macro definitions.

The example, highlights a fundamental distinction between hygienic substitution in λ-calculus and hygienic macro expansion in S-expression LISP: in the former *the names of bound variables are irrelevant*, in the latter they are not. Moreover, Users of syntax macros in S-expression LISP dialects have few guarantees about the integrity of bindings for function expressions that occur within the scope of a macro call. In S-expression LISP the macro expander treats the calling expressions as S-expressions and is insensitive to the real binding structure intended by the programmer. The binding structure of the program is not revealed until all macros are fully transcribed.

So while it is true that the hygienic approach will guard against vertical captures we are also interested in addressing the class of *horizontal* captures illustrated above. In this paper

we present an efficient expansion algorithm that addresses both vertical and horizontal capture. The algorithm is essentially a hybrid of Kohlbecker's *Macro-by-Example* (MBE) [KW87] and hygienic (HME) algorithms. Our key idea is that the problem lies not with either of these algorithms but rather with S-expression LISP's lack of syntactic structure.[1] So our adaptation of Kohlbecker's algorithm is applied in the M-LISP dialect [**?**], which has adequate structure. M-LISP is essentially a hybrid of McCarthy's original M-expression LISP and Scheme. Its essential abstract syntax is:

$$M ::= X \mid [] \mid [M\ .\ M] \mid x \mid (M\ M) \mid (\lambda x.M) \mid (\texttt{IF}\ M\ M\ M)$$

where X denotes the set of upper-case symbols, M-LISP's symbolic constants, $[M\ .\ M]$ denotes a pair and x denotes the set of lower-case symbols, M-LISP's identifiers. Its operational semantics is presented in Appendix A for reference. One important property of M-LISP is that it is independent of any representation of its programs. Since it is independent of McCarthy's original representation for M-expressions, M-LISP has no *quotation* form or any of its related forms *backquote*, *unquote* or *unquote-splicing*.

In M-LISP, rather than explicitly writing a syntactic transform function, a macro writer specifies the transformation in a pattern language superimposed on the core language. This is the method of Leavenworth [Lea66] in which responsibility for syntactic transformation lies with the compiler rather than with the programmer. For example, in M-LISP a *let* macro can be given by:

```
(MACRO
 ((LET ((i e) ...) b ...) ((LAMBDA i ... . b ...) e ...)))
```

1.1 Trading Expressiveness for Safety

While Kohlbecker's algorithms form the basis of our approach, their adaptation in this LISP dialect yields a substantially different model of macros. In general, we have followed a more conservative approach, trading off expressiveness for safety. For example, we provide no method for a macro to syntactically decomposing an abstraction $(\lambda x.M)$ into its parts x and M. Once it is introduced in the tree, an abstraction may subsequently be deleted or copied. If it is not deleted, macro calls in its body may be expanded yielding a new term $(\lambda x.M')$, but it cannot be broken into parts. As a consequence, λ-binding patterns become apparent when abstractions are first (i.e., partially) transcribed in the tree. This property will allow us to state and prove tighter restrictions on the capture of identifiers than is possible in S-expression LISP dialects in which the λ-binding patterns are not apparent until the tree is completely transcribed.

On the other hand, this is a significant restriction on the kind of macros that can be written. For example, in S-expression LISP one can write a *reference count* macro

[1] The problem with the syntactic structure of LISP is quite independent of the fact that its programs are understood to be represented by S-expressions. Rather, it is the *particular* representation defined in [McC60] which is problematic. In particular, one of the base cases in the original inductive coding of M-expressions as S-expressions for *eval* is not well-defined. A full account of the ramifications of this are beyond the scope of this paper and are covered in Section 2 of [**?**]. It is remarkable that this bug has been propagated throughout the LISP dialects including Common LISP and Scheme.

$$
\begin{array}{rcll}
\mathit{Program} & ::= & (\texttt{MACRO}\ [P\ P]\ \cdots)\ M\ \cdots & \\
M & ::= & X \mid [] \mid [M\ .\ M] \mid x_0 \mid (\lambda x_0.M) \mid (M\ \cdots) & (1) \\
P & ::= & X \mid [] \mid [P\ .\ P] \mid x \mid (\lambda x.P) \mid (P\ \cdots) \mid & (2) \\
 & & (P\ \cdots\ \ldots) \mid [P\ \cdots\ \ldots] \mid (\lambda x\ \cdots\ \ldots\ .P) & (3)
\end{array}
$$

Figure 1: Syntax

which takes an abstraction `(lambda (x) M)` and returns an abstraction `(lambda (x) M')` in which a counter is incremented each time the bound identifier `x` is referenced in `M`. This cannot be written as a macro in M-LISP. Moreover, with our strict pattern matching discipline it is impossible to determine to what kind of syntactic structure a particular pattern variable is bound.

Some of these restrictions can be overcome within the current framework simply by adopting a different coding style. Others would require an extension of the algorithm to include some of the features found in [BR88] and [CR91]. This raises the old question of just how expressive a macro facility ought to be — a question which has been largely ignored in the recent literature on name capture. Clearly there ought to be a balance between expressiveness and safety. The utility of the present work is predicated in part on the thesis that our trade-off is a reasonable one. Although the independent development of MBE lends some credence to this idea, this is a conjecture that can only be assessed in practice. Our initial experience with this system is encouraging and we believe that the system will be of interest to designers of other applicative programming languages such as ML and Haskell who wish to extend their languages with facilities for syntactic abstraction.

Among other points highlighted in the paper, M-LISP macros are distinguished by symbolic constants rather than identifiers. This eliminates *keywords* and provides visual information to readers of programs. The transcription algorithm is efficient because it indexes identifiers as they are transcribed. This is done in a separate pass through the tree in [KFFD86]. This is a property we share with the recent work in [CR91] which was developed independently of our work.

The remainder of this abstract is organized as follows. In Section 2 we introduce the syntax, define conventions and present the general framework for transcription. In Section 3 we present the transcription algorithm. In Section 3.1 we define the tree traversal algorithm and the structure of syntax tables. In Section 3.2 we define the binding valuation and in Section 3.3 we define the transcribing valuation. In Section 4 we state and prove new correctness conditions for transcription. Section 5 provides a comparison with related work and Section 6 contains concluding remarks. Appendix B contains definitions of auxiliary functions.

2 Preliminaries

The syntaxes of *programs*, *terms* and *patterns* are given in Figure 1. The centered dots "$\cdots$" denote sequencing in the metalanguage. Thus, a program is a sequence of macro definitions

followed by some terms. Equation (1) defines the syntax of terms. As in the abstract syntax the symbol X denotes the set of upper-case symbols. The symbol x_0 denotes the set of lower-case symbols indexed by 0. The indexes indicate that the identifiers occur in the source text. The rightmost clause defines both applications and macro calls. Equations (2) and (3) define the pattern language. Following the term language, upper-case symbols are pattern constants and lower-case symbols are pattern variables. Parentheses and brackets are used for grouping. Ellipses "..." denote sequencing.

We use the symbol $Term$ for the set of terms and the symbol Pat for the set of patterns. We use the metavariables M, M_1, M_2, $\cdots$ to range over $Term$, and the metavariables P, P_1, P_2, $\cdots$ to range over Pat. We use Z, Z_1, Z_2, $\cdots$ to range over $Term \cup Pat$. We use j for *clock* values $0, 1, \cdots$ and y_j, x_{j_1}, x_{j_2}, $\cdots$ to range over the set $Tvar$ of variables indexed by clock values. We use the metavariables x, y, v_1, v_2, ... to range over the set $Pvar$ of pattern variables. In contexts where both x and x_j appear, they are understood to have the same name. We use the single metavariable X to range over both symbols and pattern constants. We use the symbol $\equiv$ to denote syntactic equality.

As in MBE the transcription of a call term M is determined by pairs of patterns $[P_c\ P_t]$. The patterns are retained in *syntax tables* which are given by

$$\tau \in T = [T \to Term \to Nat \to Term]$$

The transcription of P_t is governed by an *environment* of pattern variable bindings obtained from matching the call term M to the call pattern P_c. Environments are drawn from the domain:

$$\rho \in Env = Pvar \to \langle Nat, Term^* \rangle$$

We use the symbol ρ_0 to denote the empty environment. The expression $\rho[x \mapsto \langle n, M \rangle]$ denotes the environment obtained from ρ by associating the pattern variable x with the tuple $\langle n, M \rangle$. The expression $(\langle n, M \rangle \downarrow 1)$ selects n, and $(\langle n, M \rangle \downarrow 2)$ selects M. In $\langle n, M \rangle$, n gives the level of ellipsis nesting. If $n = 0$ then the variable is not matched in the scope of an ellipsis, and M is a component of the call term. If $n > 0$ then the variable is governed by at least one ellipsis and M is a sequence of call subterms. For example, matching the pattern `[x ...]` to the call term `[A i (f j)]` produces an environment $[x \mapsto \langle 1, [\texttt{A i (f j)}]\rangle]$. While matching the pattern `[[x ...] ...]` to the call term `([a A] [true TRUE])` produces

$$[x \mapsto \langle 2, [[\texttt{a A}][\texttt{true TRUE}]]\rangle]$$

The ellipsis level will be used in the transcription process to guide transcription of occurrences of x.

2.1 Concrete and Abstract Syntax

Proper lists and functions of several arguments may be expressed in concrete syntax as defined by the following identities:

$$\begin{aligned} [Z_1\ Z_2\ \cdots\ Z_n] &\equiv [Z_1\ .\ [Z_2\ .\ [\ \cdots\ [Z_n\ .\ []]\ \cdots\ ]]] \\ (\lambda z_1 z_2\ \cdots\ z_n.Z) &\equiv (\lambda z_1.(\lambda z_2.(\ \cdots\ (\lambda z_n.Z)\ \cdots\ ))) \end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{C} & : & Term \to T \to U \to K \to Dv \\
\mathcal{D} & : & \mathit{Core\text{-}Term} \to U \to K \to Dv \\
\mathcal{W} & : & Term \to T \to Nat \to Term \\
\mathcal{E} & : & Pat_c \times Pat_t \to T \to T \\
\mathcal{B} & : & Pat_c \to Env \to Term \to Env \\
\mathcal{T} & : & Pat_t \to Env \to Nat \to Term
\end{array}
$$

Figure 2: Valuations

For example, matching the concrete expression `[A B C]` against the pattern `[first . rest]` produces the environment

$$[\mathit{first} \mapsto \langle 0, \texttt{A}\rangle][\mathit{rest} \mapsto \langle 0, \texttt{[B . [C . []]]}\rangle]$$

In the absence of syntax extensions, applications may also be defined in their curried abstract form $(Z\ Z)$, with the following correspondence between concrete and abstract syntax:

$$(Z_1\ Z_2\ Z_3\ \cdots) \equiv ((Z_1\ Z_2)\ Z_3\ \cdots) \tag{4}$$

In the presence of syntax extensions, however, it is more convenient to obtain the left-hand side of (4) through sequencing in the metalanguage since it preserves the concrete structure. This not only simplifies the call interface but it also simplifies the interaction of the ellipsis with the left-associativity of application. For example, consider the pattern `(x y ...)` and the call form `((f g) h i)`. The abstract form of the latter is `(((f g) h) i)`, but the preferred match:

$$[x \mapsto \langle 0, \texttt{(f g)}\rangle][y \mapsto \langle 1, \texttt{[h i]}\rangle]$$

is determined by the original concrete structure.

2.2 Valuations

The valuations of interest are shown in Figure 2. $\mathcal{C}$ corresponds to an interpreter or compiler and $\mathcal{D}$ defines the dynamic semantics. $\mathcal{W}$, $\mathcal{E}$, $\mathcal{B}$ and $\mathcal{T}$ collectively transcribe syntax macros. $\mathcal{C}$ is given by:

$$\mathcal{C}[\![M_0]\!] = \lambda\tau.\mathit{let}\ M_0' = \mathcal{W}[\![M_0]\!]\tau 1\ \mathit{in}\ \lambda\rho\kappa.\mathcal{D}[\![M_0']\!]\rho\kappa \tag{5}$$

The syntax tree M_0 is recursively walked by $\mathcal{W}$ with syntax table τ and an initial clock value. If the transcription process terminates the resulting core term M_0' will be evaluated according to the dynamic semantics $\mathcal{D}$ in environment ρ and continuation κ. Two notes about $\mathcal{D}$: it identifies all *free* variables of the same name x_{j_1}, x_{j_2} in M_0' even though their indexes may differ. Secondly, it applies (4) to the result of transcription. We will not consider $\mathcal{D}$ further here since it has no bearing on syntax transcription.

$$
\begin{array}{rcll}
\mathcal{W}[\![X]\!] & = & \lambda\tau j.X & \\
\mathcal{W}[\![[]]\!] & = & \lambda\tau j.[] & \\
\mathcal{W}[\![[M\ .\ N]]\!] & = & \lambda\tau j.[\mathcal{W}[\![M]\!]\tau j\ .\ \mathcal{W}[\![N]\!]\tau j] & \\
\mathcal{W}[\![x_{j'}]\!] & = & \lambda\tau j.x_{j'} & j' \geq 0 \\
\mathcal{W}[\![(\lambda x_{j'}.M)]\!] & = & \lambda\tau j.(\lambda x_{j'}.\mathcal{W}[\![M]\!]\tau j) & j' \geq 0 \\
\mathcal{W}[\![(M_1\ \cdots\ M_k)]\!] & = & \lambda\tau j.M_1 \equiv X \rightarrow & k \geq 1 \\
 & & \quad \tau\tau[\![(M_1\ \cdots\ M_k)]\!]j, & \\
 & & \quad (\mathcal{W}[\![M_1]\!]\tau j\ \cdots\ \mathcal{W}[\![M_k]\!]\tau j) &
\end{array}
$$

Figure 3: Syntax Tree Traversal

3 The Transcription Algorithm

3.1 Syntax Tree Traversal and Syntax Tables

The valuation $\mathcal{W}$ is given by induction on the structure of syntax trees in Figure 3. The final equation defines the semantics of terms of the form $(M_1\ \cdots\ M_k)$ which include both applications and macro calls. If M_1 is a symbolic constant then the term is taken as a macro call. To transcribe the term the expander applies the syntax table to itself, the call term and the current clock value. If M_1 is not a symbol then each $M_i, 1 \leq i \leq k$ is recursively walked.

The empty syntax table contains no transcription patterns, thus it maps all terms to themselves.

$$\tau_0 \equiv \lambda\tau M j.M$$

A table is extended when a macro definition is processed:

$$
\begin{array}{l}
\mathcal{E}[\![\ [P_c\ P_t]\ ]\!] = \lambda\tau\tau' M j. \\
\quad \mathcal{B}[\![P_c]\!]\rho_0 M \neq \perp \rightarrow \\
\quad\quad \mathcal{W}[\![\mathcal{T}[\![P_t]\!](\mathcal{B}[\![P_c]\!]\rho_0 M)j]\!]\tau'(j+1), \\
\quad\quad \tau\tau' M j
\end{array}
\tag{6}
$$

P_c represents the call pattern, P_t represents the transcription pattern, τ represents the macro-definition-time syntax table and τ' represents the outermost macro-transcription-time syntax table. M represents the call term and j is the clock value. The symbol $\perp$ denotes a failed match. If the call matches the call pattern then an environment is generated and used for the transcription process. Since the resulting syntactic structure may have macro calls, it is recursively walked with the outermost syntax table and an incremented clock value. If the call fails to match then it is checked against the next entry in the table.

We can inductively define the *length* of a transcription based on the the *then* branch of (6). The induction selects the maximum of the lengths of the parallel tree walks in the pair and application clauses of $\mathcal{W}$. Thus, the length is the highest clock value occurring during $\mathcal{W}[\![M_0]\!]\tau 1$.

$$
\begin{array}{rcl}
\mathcal{B}[\![\ X\ ]\!] & = & \lambda\rho M.M \equiv X \rightarrow \rho, \bot \\
\mathcal{B}[\![\ []\ ]\!] & = & \lambda\rho M.M \equiv [] \rightarrow \rho, \bot \\
\mathcal{B}[\![\ [P_1\ .\ P_2]\ ]\!] & = & \lambda\rho M.M \equiv [M_1\ .\ M_2] \rightarrow \mathit{fold}\ [P_1\ P_2][M_1\ M_2]\rho, \bot \\
\mathcal{B}[\![\ x\ ]\!] & = & \lambda\rho M.x \in \mathit{Dom}(\rho) \rightarrow ((\rho\ x) = \langle 0, M\rangle \rightarrow \rho, \bot), \rho[x \mapsto \langle 0, M\rangle] \\
\mathcal{B}[\![(P_1\ \cdots\ P_k)]\!] & = & \lambda\rho M.M \equiv (M_1 \cdots M_k) \rightarrow \mathit{fold}\ [P_1 \cdots P_k][M_1 \cdots M_k]\rho, \bot \\
\mathcal{B}[\![\ (P_1 \cdots P_k \ldots)\ ]\!] & = & \lambda\rho M.M \equiv (M_1 \cdots M_n) \rightarrow \mathit{seqmatch}\ [P_1 \cdots P_k]\ [M_1 \cdots M_n]\ \rho, \bot \\
\mathcal{B}[\![\ [P_1 \cdots P_k \ldots]\ ]\!] & = & \lambda\rho M.M \equiv [M_1 \cdots M_n] \rightarrow \mathit{seqmatch}\ [P_1 \cdots P_k]\ [M_1 \cdots M_n]\ \rho, \bot
\end{array}
$$

Figure 4: Environment Construction

$$
\begin{array}{rcl}
\mathcal{T}[\![\ X\ ]\!] & = & \lambda\rho j.X \\
\mathcal{T}[\![\ []\ ]\!] & = & \lambda\rho j.[] \\
\mathcal{T}[\![\ [P_1\ .\ P_2]\ ]\!] & = & \lambda\rho j.[\mathcal{T}[\![P_1]\!]\rho j\ .\ \mathcal{T}[\![P_2]\!]\rho j] \\
\mathcal{T}[\![\ (P_1\ \cdots\ P_k)\ ]\!] & = & \lambda\rho j.(\mathcal{T}[\![P_1]\!]\rho j\ \cdots\ \mathcal{T}[\![P_k]\!]\rho j)
\end{array}
$$

Figure 5: Simple Transcription Equations

3.2 Environment Construction

The environment is constructed by the binding valuation $\mathcal{B} : Pat_c \rightarrow Env \rightarrow Term \rightarrow Env$, which is defined by cases on patterns in Figure 4. If the call fails to match one of these patterns it denotes $\bot$. A pattern symbol matches a call symbol if they are identical symbols. A pair pattern matches a pair in the call if the cdr component of the pattern matches the cdr component of the call in the environment constructed by matching the respective cars. Matching is left-to-right. A pattern variable matches any call term provided that the variable is not already bound to a different value in ρ. Application patterns define the top level interface. These follow the left-to-right matching of a pair. A sequence pattern of the form $(P_1\ \cdots\ P_k\ \ldots)$ matches a call if the call is an application form, P_1 through P_{k-1} match the first $k-1$ components of the call form and P_k matches each of the remaining components of the form. The auxiliary functions *fold* and *seqmatch* are defined in the appendix.

We wish to emphasize that we define no semantics for matching a pattern of the form $(\lambda x.P)$. In M-LISP, abstractions cannot be decomposed. This is a key difference between our system and those proposed for Scheme.

3.3 Transcription

We now consider transcription of the pattern P_t in the environment $\mathcal{B}[\![P_c]\!]\rho_0 M$ and clock cycle j. The transcribing valuation $\mathcal{T} : Pat_t \rightarrow Env \rightarrow Nat \rightarrow Term$, is defined by cases on patterns. The simple equations are presented in Figure 5. Symbols and nil are introduced directly into the syntax tree. Pair and application patterns cause pair and application forms

to be introduced with recursively transcribed components. The equations for transcribing ellipsis patterns follow those defined in [Koh86] and are defined in the appendix. Two equations that require some elaboration appear in the text below.

3.3.1 Transcribing Pattern Variables

The transcription of a pattern variable is determined by its value in ρ. If the variable is defined in ρ with top level value M, then M is the transcription. If the pattern variable is unbound in ρ then it is taken to be a lambda variable rather than a pattern variable; it is introduced with the current clock value.

$$\mathcal{T}[\![\ x\ ]\!] = \lambda\rho j.x \in Dom(\rho) \rightarrow ((\rho\ x) = \langle 0, M\rangle \rightarrow M, \bot), x_j$$

3.3.2 Transcribing Abstraction Patterns

Procedure transcription breaks into two cases depending on the binding status of the pattern variable x in the formal parameter position. If x is bound in ρ, clause (7), then it is a compile-time pattern variable which should have matched some indexed run-time variable $y_{j'}$ in the call term. If it has, then a procedure is introduced into the tree with formal parameter $y_{j'}$ and a body determined by the recursive transcription of P. If x is unbound in ρ, clause (8), then x corresponds to a generated identifier. A procedure is introduced into the tree with the time stamped version of x (i.e., x_j) as its formal parameter. The body pattern is transcribed in an environment that maps the pattern variable to the new lambda variable.

$$\begin{aligned}\mathcal{T}[\![\ (\lambda x.P)\ ]\!] = \lambda\rho j.x \in Dom(\rho) \rightarrow \\ ((\rho\ x) = \langle 0, y_{j'}\rangle \rightarrow (\lambda y_{j'}.\mathcal{T}[\![P]\!]\rho j), \bot), \quad (7)\\ (\lambda x_j.\mathcal{T}[\![P]\!](\rho[x \mapsto \langle 0, x_j\rangle])j) \quad (8)\end{aligned}$$

This clause can be modified (at the expense of safety) to allow for intentional captures. Since our current emphasis is on safe transcription we do not consider this extension here.

4 Name Capture

In the following we use the familiar notion of contexts, $C[]$, in considering the tree structure M_0' which ultimately results from a terminating transcription of M_0 in (5).

4.1 Vertical Capture

We use the term *upward capture* to refer to captures in which, for some j, s, an applied occurrence is introduced in step $j+s$ within the scope of a binding occurrence of the same name introduced in step j. For example, taking $j = 0$ and $s = 1$:

```
(extend-syntax first
  ((first x) (car x)))
```

```
(lambda (car) (first z))
 => (lambda (car) (car z))
```

is an upward capture. We use the term *downward capture* when, for some j, s, an applied occurrence introduced in step j is captured by a binding occurrence introduced in step $j+s$. An example of this was illustrated in the introduction.

The proposition that guarantees the correct transcription of these *vertical* captures is:

Proposition 1 (Vertical Safety) *Let* $\mathcal{W}[\![M_0]\!]\tau 1 = M_0' \equiv C[(\lambda x_{j'}.M')]$. *Then every* $x_{j'} \in M'$ *was introduced in step* j'.

This is vacuously true since j increases monotonically at each transcription step. Since j' may be 0, the proposition is a generalization of the *hygiene* condition.

4.2 Horizontal Capture

As we have noted, neither the hygiene condition nor the proposition are sufficient to rule out broken bindings in general. The example in the introduction illustrates that capture may also occur horizontally within a single step by having an applied occurrence of an identifier $x_{j'}$ move within the scope of a binding occurrence of the same name that was generated in the same step. Similarly, an applied occurrences of $x_{j'}$ may occur in step j within the scope of a binding occurrence $\lambda x_{j'}$, and later moved out of that scope.

The difficulty in S-expression LISP is that it cannot, in general, be determined what role a symbol will ultimately play in M_0'. In particular, it cannot be determined whether or not a symbol is an identifier. This is not the case in M-LISP since it distinguishes between symbols and identifiers, has no quotation form and syntax transcription does not decompose abstractions. Thus, we can track the development of the term $(\lambda x_{j'}.M')$ from the time it is first (i.e., partially) introduced. For example, if (MA) transcribes to $(\lambda x_0.(\text{MB}))$ in step 1 and (MB) transcribes to x_2 in step 2. Then $(\lambda x_0.x_2)$ is fully transcribed in step 2 but introduced in step 1.

If $(\lambda x_{j'}.M')$ is introduced in the syntax tree in step j then, by convention, we say that it was *introduced as* $(\lambda x_{j'}.M)$ if this was introduced in step j and M transcribes to M'. There are three points at which a term $(\lambda x_{j'}.M') \in \mathcal{W}[\![M_0]\!]\tau 1$ can be introduced:

1. $j = 0$. Then $(\lambda x_{j'}.M') \equiv (\lambda x_0.M')$. Its introduced form $(\lambda x_0.M)$ occurs in the source tree and, since there is no binding equation $\mathcal{B}$ for λ, $\mathcal{W}[\![M]\!]\tau 1 = M'$.

2. $j > 0$ and $(\lambda x_{j'}.M')$ is established, for some pattern P, as $(\lambda x_{j'}.\mathcal{T}[\![P]\!]\rho j)$ such that for some $\rho, y, (\rho y) = \langle 0, x_{j'} \rangle$ (in general $j \neq j'$.)

3. $j > 0$ and $(\lambda x_{j'}.M')$ is established, for some pattern P, as $(\lambda x_j.\mathcal{T}[\![P]\!](\rho[x \mapsto \langle 0, x_j \rangle])j)$ such that, for $\rho, x, x \notin Dom(\rho)$.

Once a term $(\lambda x_{j'}.M)$ is introduced in the tree it may subsequently be deleted or copied. If it is not deleted macros in its body may be transcribed. We define the *scope* of its binding occurrence $\lambda x_{j'}$ to be the region between the "." and the ")". At step j this is occupied by M. For example, the rightmost x_0 in $(\lambda x_0.(\text{MA } x_0))$ occurs in the scope of λx_0 even though the syntax extension MA may operate on it.

Proposition 2 (Horizontal Safety) *Let the terminating transcription*

$$\mathcal{W}[\![M_0]\!]\tau 1 = C[(\lambda x_{j'}.M')]$$

occur in n steps. Let $(\lambda x_{j'}.M')$ *be introduced as* $(\lambda x_{j'}.M)$ *in step* j*. Then*

1. *No identifier occurrence* $y_{j_0} \in M$ *at the end of step* j *appears outside of the scope of* $\lambda x_{j'}$ *in step* $s,\ j \leq s \leq n$.
2. *If the identifier occurrence* $y_{j_0} \notin M$ *at the end of step* j*, then* y_{j_0} *can only appear in the scope of* $\lambda x_{j'}$ *in step* $s,\ j \leq s \leq n$*, if* $j_0 > j$.

Proof: (Sketch)

1. Assume the converse. Then, for some s', step $j + s'$ transcribes the identifier occurrence y_{j_0} outside the scope of $\lambda x_{j'}$. The transcription has a pattern variable which matches y_{j_0} in the scope of $\lambda x_{j'}$ and transcribes it outside. However, this is impossible since there is no binding equation for abstractions. The only pattern which matches $(\lambda x_{j'}.M)$ is a pattern variable.
2. Similar.

Remark: An identifier occurrence y_{j_0} in the scope of $\lambda x_{j'}$ may be left in place, copied, deleted or moved into a *hole* in the scope of $\lambda x_{j'}$. For example,

```
(MACRO
  [(M1 x)  (LAMBDA x . (M2 x x))]
  [(M2 x y)  [x . (LAMBDA x . x)]])

 (M1 a)
  => (LAMBDA a . (M2 a a))
      => (LAMBDA a . [a . (LAMBDA a . a)])
```

5 Related Work

Although M-LISP's syntax macro facility is based on the MBE and Hygienic algorithms there are a number of important points of contrast. These stem from M-LISP's structured (i.e., quotation-free) abstract syntax. This property was essential to the tightened correctness conditions. We briefly consider a few of the other distinctions. The Hygienic/MBE algorithm uses the following interface:

```
(extend-syntax <keyword-list> <key-identifier-list>
 (<pattern> <fender> <pattern>) ... )
```

The keyword list declares that a set of symbols are associated with syntax extensions and are not identifiers. Keywords are not required in M-LISP because syntax macros are associated with symbolic constants rather than identifiers. *Key* identifiers are those for which free occurrences in the macro call are intended to be associated with binding occurrences of the same name in the macro definition. For example, in

```
(MACRO
  [(LOOP e ...) (call/cc (LAMBDA exit-with-value . (WHILE TRUE e ...)))])
```

free occurrences of the key-identifier `exit-with-value` in the calling form are intended to be captured. We have omitted the modification to clause (8) which governs these controlled captures. This is not an essential restriction. The interface can be extended to include a key-identifier list and clause (8) can be modified to associate the generated binding instance with the applied instances in the transcribed body. Unfortunately, it is not clear how to distinguish between intended key-identifier captures and accidental captures.

Although the MBE interface is based on the *specification* of transformations, it nevertheless permits a macro writer to explicitly operate on the S-expression representation of the calling form through *fenders*. These are arbitrary LISP expressions evaluated during transcription to perform checks on the calling expression. Since this representation dependence is contrary to the design goals of the M-LISP macro facility, we provide no analog of fenders. While there are clearly many S-expression LISP macros which cannot be expressed in this style we wish to point out the parallels between Scheme's reformulation of functions from metalinguistic to linguistic status and our reformulation of macros. In LISP 1.5, for example, function representations could be dynamically constructed

```
(defun funmaker (args body)
  (cons (quote lambda) (cons args (list body))))
```

and then applied as in

```
> ((lambda (f) (f (quote a))) (funmaker (quote (x)) (quote x)))

a
```

In practice, the loss of such dynamically constructed functions has not proved to be a significant hardship in Scheme and we have reason to believe that this will prove true of our restructuring of macros as well.

Our work has parallels with the recent work of [CR91]. Contrary to their view that the main problem with the Kohlbecker's hygienic algorithm is its inefficiency, we believe that is main problem is that it does not address the kinds of captures considered above. We have not considered lexical nesting of macros in our work.

6 Conclusions

We have presented a framework for extending the syntactic structure of a simple representation-independent dialect of LISP which avoids a large class of possible name captures. The structure of the syntax was essential for the compiler to distinguish binding patterns *during* transcription.

A Operational Semantics of M-LISP

The operational semantics is given in the style of [Plo75]. Recall that the abstract syntax is:

$$M ::= X \mid [] \mid [M\ .\ M] \mid x \mid (M\ M) \mid (\lambda x.M) \mid (\texttt{IF}\ M\ M\ M)$$

Axioms:

$$\begin{array}{rcll}
((\lambda x.M)\ N) & \xrightarrow{v} & M[x := N] & \text{— } N \in V \\
(\texttt{CAR}\ [M_1\ .\ M_2]) & \xrightarrow{v} & M_1 & \text{— } M_1, M_2 \in V \\
(\texttt{CDR}\ [M_1\ .\ M_2]) & \xrightarrow{v} & M_2 & \text{— } M_1, M_2 \in V \\
(\texttt{EQ?}\ M_1\ M_2) & \xrightarrow{v} & \texttt{TRUE} & \text{— } M, N \text{ symbols}, M \equiv N \\
(\texttt{EQ?}\ M_1\ M_2) & \xrightarrow{v} & \texttt{FALSE} & \text{— } M, N \text{ symbols}, M \not\equiv N \\
(\texttt{ATOM?}\ M) & \xrightarrow{v} & \texttt{TRUE} & \text{— } M \in V, \text{ an atom} \\
(\texttt{ATOM?}\ M) & \xrightarrow{v} & \texttt{FALSE} & \text{— } M \in V, \text{ not an atom} \\
(\texttt{IF TRUE}\ M_1\ M_2) & \xrightarrow{v} & M_1 & \\
(\texttt{IF FALSE}\ M_1\ M_2) & \xrightarrow{v} & M_2 &
\end{array}$$

Inference rules:

$$\frac{M \xrightarrow{v} M'}{(M\ N) \xrightarrow{v} (M'\ N)}\ ,\ \frac{N \xrightarrow{v} N', M \in V}{(M\ N) \xrightarrow{v} (M\ N')}$$

$$\frac{M \xrightarrow{v} M'}{[M\ .\ N] \xrightarrow{v} [M'\ .\ N]}\ ,\ \frac{N \xrightarrow{v} N', M \in V}{[M\ .\ N] \xrightarrow{v} [M\ .\ N']}$$

$$\frac{M_1 \xrightarrow{v} M_1'}{(\texttt{IF}\ M_1\ M_2\ M_3) \xrightarrow{v} (\texttt{IF}\ M_1'\ M_2\ M_3)}$$

Figure 6: The Operational Semantics of M-LISP.

We use the symbol Λ_π to denote the set of terms. We extend the λ-calculus notion of substitution $N[x := M]$, "M for free occurrences of x in N" with the additional clauses:

$$\begin{array}{rcl}
X[x := M] & \equiv & X \\
[][x := M] & \equiv & [] \\
[N\ .\ L][x := M] & \equiv & [N[x := M]\ .\ L[x := M]]
\end{array}$$

and similarly for the conditional. We define the set V of *values* inductively as containing any term of the form X, x, $[]$, $(\lambda x.M)$ or $[M_1\ .\ M_2]$ whenever M_1 and M_2 are values. The operational semanitcs is then defined in Figure 6.

B Auxiliary Functions

The auxiliary function $\mathit{fold} : Pat^* \to Term^* \to Env \to Env$, is given by

$$\mathit{fold}\ [P_1\ \cdots\ P_n][M_1\ \cdots\ M_n]\rho = \mathcal{B}[\![P_n]\!](\cdots\ \mathcal{B}[\![P_1]\!]\rho M_1\ \cdots)M_n$$

The function $\mathit{seqmatch} : Pat^* \to Term^* \to Env \to Env$, matches a sequence of patterns to a sequence of call terms in environment ρ. The patterns match with call terms one-for-one until the last pattern P_k is reached. This pattern is mapped across the remaining call terms M_k through M_n. The resulting sequence of environments is reduced to a single environment by *combine*. For $0 \le i < k \le n$

$$\begin{array}{l}
\mathit{seqmatch}\ [P_i\ \cdots\ P_{k-1}\ P_k][M_i\ \cdots\ M_{k-1}\ M_k\ \cdots\ M_n]\ \rho = \\
\qquad \mathit{combine}\ (\mathit{fold}\ [P_1 \cdots P_{k-1}][M_1 \cdots M_{k-1}]\rho)[\mathcal{B}[\![P_k]\!]\rho_0 M_k \cdots \mathcal{B}[\![P_k]\!]\rho_0 M_n]
\end{array}$$

The function $\mathit{combine} : Env \to Env^* \to Env$, is given by

$$\mathit{combine}\ \rho\ [\rho_1\ \cdots\ \rho_n] = \lambda x.x \in Dom(\rho_1) \to \langle(\rho_1 x \downarrow 1) + 1, [(\rho_1 x \downarrow 2)\ \cdots\ (\rho_n x \downarrow 2)]\rangle, \rho x$$

The equation for a pair sequence pattern is

$$\begin{aligned}
&\mathcal{T}[\![\ [P_i\ \cdots\ P_k\ \ldots]\ ]\!] = \\
&\quad \lambda\rho j.\exists x \in Dom(\rho \mid varsof(P_k)) \text{ such that } (\rho\ x \downarrow 1) \geq 1 \rightarrow \\
&\quad\ let\ [\rho_1\ \cdots\ \rho_n] = decompose(\rho \mid varsof(P_k))\ in \\
&\quad\quad [\mathcal{T}[\![P_i]\!]\rho j\ \cdots\ \mathcal{T}[\![P_{k-1}]\!]\rho j\ \mathcal{T}[\![P_k]\!]\rho_1 j\ \cdots\ \mathcal{T}[\![P_k]\!]\rho_n j], \bot
\end{aligned}$$

The function $decompose : Env \rightarrow Env^*$, which takes an environment and splits it into a sequence of environments is given by

$$\begin{aligned}
decompose \quad = \quad & \lambda\rho.\exists v \in Dom(\rho) \text{ such that } (\rho\ v \downarrow 2) = [] \rightarrow [], \\
& \quad let\ \{v_1, \ldots, v_k\} = Dom(\rho)\ in \\
& \quad\quad [(\rho_0[v_1 \mapsto splita((\rho\ v_1))]\ \cdots\ [v_k \mapsto (splita\ (\rho\ v_k))])\ . \\
& \quad\quad decompose(\rho_0[v_1 \mapsto splitb(\rho\ v_1))]\ \cdots\ [v_k \mapsto splitb(\rho\ v_k))])]
\end{aligned}$$

The functions *splita* and *splitb* are given as in MBE by

$$\begin{aligned}
splita\langle n, M\rangle \quad &= \quad (n = 0) \rightarrow \langle 0, M\rangle, \langle n-1, hd(M)\rangle \\
splitb\langle n, M\rangle \quad &= \quad \langle n, (n = 0) \rightarrow M, tl(M)\rangle
\end{aligned}$$

References

[BR88] A. Bawden and J. Rees. Syntactic closures. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 86–95, 1988.

[CR91] W. Clinger and J. Rees. Macros that work. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[KFFD86] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 151–161, 1986.

[Koh86] E. Kohlbecker. *Syntax Extensions in the Programming Language LISP*. PhD thesis, Indiana University, 1986.

[KW87] E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[Lea66] B. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9:790–793, 1966.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, 1960.

[Plo75] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, **1**:125–159, 1975.